

**UNIVERSIDADE FEDERAL DE SANTA CATARINA
CENTRO TECNOLÓGICO
CURSO DE BACHARELADO EM CIÊNCIAS DA
COMPUTAÇÃO**

**"COMPARAÇÃO DE EFICIÊNCIA COMPUTACIONAL
ENTRE AS TRANSFORMADAS RÁPIDAS DE FOURIER
(FFT) E DE HARTLEY (FHT)"**

Autor: Saulo Castilho

Orientador: Prof. Júlio Felipe Szeremeta

Florianópolis, SC, novembro de 2005

**UNIVERSIDADE FEDERAL DE SANTA CATARINA
CENTRO TECNOLÓGICO
CURSO DE BACHARELADO EM CIÊNCIAS DA
COMPUTAÇÃO**

**"COMPARAÇÃO DE EFICIÊNCIA COMPUTACIONAL
ENTRE AS TRANSFORMADAS RÁPIDAS DE FOURIER
(FFT) E DE HARTLEY (FHT)"**

Autor: Saulo Castilho

Orientador: Prof. Júlio Felipe Szeremeta

Florianópolis, SC

2005 / 2

FOLHA DE APROVAÇÃO

Trabalho de conclusão de curso submetido ao Curso de Ciências da Computação do Departamento de Informática e Estatística do Centro Tecnológico, da Universidade Federal de Santa Catarina como parte dos requisitos para obtenção do grau de Bacharel em Ciências da Computação.

AUTOR:

Saulo Castilho

Trabalho aprovado pela comissão examinadora em: __/__/____

COMISSÃO EXAMINADORA:

Prof. Júlio Felipe Szeremeta (Orientador)

Prof. Daniel Santana de Freitas

Prof. Sérgio Peters

DEDICATÓRIA

Aos meus pais que sempre me acompanharam de perto em minha jornada escolar e acadêmica, me dando todo o apoio e colocando sempre meu aprendizado e minha formação profissional como prioridade em suas vidas.

AGRADECIMENTOS

Agradeço aos meus amigos Marcus Vinicius e Alessandro por terem me ajudado com seus conhecimentos sobre a linguagem C++, sistemas operacionais, compiladores e sobre a própria FFT.

RESUMO

Este trabalho apresenta um estudo comparativo entre a transformada de Hartley e a transformada de Fourier, desde seus princípios matemáticos até a amostragem prática de seus respectivos tempos de processamento, passando por detalhes de suas implementações.

O ponto central deste estudo é a comparação de eficiência em termos de tempo de processamento necessário para ambas as transformadas quando submetidas às condições mais semelhantes possíveis. Os resultados obtidos nesta comparação são apresentados aqui de forma clara através de gráficos e tabelas.

ABSTRACT

This paper presents a comparative study between the Hartley transform and the Fourier transform, from their mathematical principles to the practical sampling of their processing time, even analyzing their implementation details.

The main point of this study is the efficiency comparison of the processing time demanded by both the transforms in as similar as possible conditions. The results from this comparison are presented here in a clear way through graphics and tables.

SUMÁRIO

Resumo.....	6
Abstract.....	7
Sumário.....	8
Lista de Figuras e Gráficos.....	9
Lista de Tabelas.....	9
Introdução.....	10
Objetivos.....	12
Capítulo 1: Conceituação.....	13
1.1: Histórico.....	13
1.2: Definições.....	14
1.3 Propriedades das Transformadas de Fourier e Hartley.....	19
Capítulo 2: Análise comparativa entre a DHT e a DFT.....	23
Capítulo 3: As Transformadas Rápidas.....	32
3.1: Transformada Rápida de Fourier (FFT).....	32
3.2: Transformada Rápida de Hartley (FHT)	35
3.3: FFT versus FHT	37
Conclusão.....	43
Referências Bibliográficas.....	46
Anexo: Código fonte.....	47

LISTA DE FIGURAS E GRÁFICOS

Gráfico 2.1.....	28
Gráfico 2.2.....	28
Gráfico 2.3.....	29
Gráfico 2.4.....	29
Gráfico 2.5.....	30
Gráfico 2.6.....	30
Figura 3.1.....	35
Figura 3.2.....	36
Figura 3.3.....	36
Gráfico 3.1.....	39
Gráfico 3.2.....	40
Gráfico 3.3.....	41
Gráfico 3.4.....	41

LISTA DE TABELAS

Tabela 2.1.....	25
Tabela 2.2.....	27
Tabela 3.1.....	39

INTRODUÇÃO

Dentre os vários campos de conhecimento que são abrangidos pela ciência da computação, a computação científica aplicada é o que abriga o tópico abordado neste trabalho. Mais especificamente, estão envolvidos métodos de Análise Numérica, a qual busca encontrar e avaliar algoritmos para resolver computacionalmente modelos matemáticos comuns e de grande importância em várias áreas. Devido a fatores tais com a imprecisão da representação numérica em computadores e a falta de métodos objetivos para resolver alguns desses problemas, dificilmente se encontra uma solução exata, ficando a cargo da análise numérica estudar métodos para se encontrar uma solução satisfatória, tão próxima da real quanto desejado.

O tópico que pode ser considerado o principal da análise numérica é a teoria da aproximação, que aborda aproximação de funções, que tanto podem ter expressões conhecidas, quanto podem ser definidas através de um conjunto discreto de pontos (x_k, y_k) . É neste tópico que geralmente é introduzida a análise de Fourier, com a finalidade de aproximar funções utilizando-se polinômios trigonométricos. Porém a importância dessa análise vai muito além da teoria da aproximação, sendo crucial em muitas áreas científicas, já que sua principal aplicação está na transformada de Fourier, que mapeia uma função do tempo em uma função da frequência.

No entanto, a transformada de Fourier como definida originalmente é inviável de ser utilizada na prática, devido a seu custo computacional, sendo o número de operações da ordem de $O(n^2)$. Para resolver este problema surgiu a Transformada Rápida de Fourier (FFT) que acelerou de forma impressionante este processo, reduzindo o custo para $O(n \log_2 n)$ operações de números complexos.

Por outro lado, uma solução muito semelhante surgiu como alternativa à transformada rápida de Fourier, com a vantagem de não utilizar operações aritméticas com números complexos, o que tornaria o processamento ainda mais rápido, já que estes demandam maior processamento para as operações e o

dobro de memória para o armazenamento. Esta solução é denominada Transformada Rápida de Hartley (FHT).

Por conseqüência surgiu a idéia para este trabalho, que é estudar mais a fundo os conceitos envolvidos nas duas transformadas e fazer sua comparação tanto nos conceitos teóricos, quanto nas implementações práticas. O que motiva esse esforço é que a transformada de Fourier tem enorme importância e um grande número de aplicações, tais como reconhecimento de padrões e processamento de sinais eletromagnéticos, análise de ondas e imagens, entre outras. A proposta da transformada de Hartley, com a mesma funcionalidade da de Fourier e supostamente mais eficiente continuou sendo quase desconhecida, aparecendo em raras publicações e em pouco substituiu a FFT. Dessa forma, a divulgação da FHT traria grandes benefícios às áreas citadas.

Neste texto serão relatados os resultados obtidos no estudo e comparação dessas transformadas tanto em teoria quanto na prática. No capítulo 1 as transformadas discretas de Fourier e Hartley serão apresentadas de forma que primeiro será feita uma contextualização histórica de seu surgimento, depois será feita sua conceituação e serão apresentadas algumas de suas propriedades. No capítulo 2 serão detalhadas implementações e testes feitos com ambas. O capítulo 3 é dedicado às transformadas rápidas, sendo tanto numa abordagem teórica como na abordagem prática, novamente através de implementações e testes, fechando o objetivo principal do trabalho que é a comparação entre a FHT e a FFT.

OBJETIVOS

Objetivo geral:

O objetivo geral deste trabalho é o estudo dos fundamentos de teóricos da Transformada Rápida de Hartley e sua comparação em relação à Transformada Rápida de Fourier.

Objetivos específicos:

- Entender o suporte teórico da FHT;
- Efetuar a algoritmização e implementação da Transformada de Hartley na linguagem C++;
- Testar a eficiência da FHT comparada com a FFT
- Divulgar a Transformada de Hartley e torná-la acessível como uma alternativa à Transformada de Fourier em casos onde o custo-benefício de sua implantação seja satisfatório.

CAPÍTULO 1 - CONCEITUAÇÃO

1.1 HISTÓRICO

O título deste trabalho deixa transparecer com muita clareza qual o principal objetivo a ser alcançado neste estudo. Porém antes de nos aprofundarmos tanto na parte matemática, quanto na parte prática das transformadas de Fourier e de Hartley é interessante sabermos que fatos estão por trás desses conceitos, ou seja, como e por que motivo eles surgiram.

Para isso, a história nos remete ao século XVIII, antes mesmo do nascimento de Jean Baptiste Joseph Fourier, fato que ocorreu no ano de 1768, quando já havia prévias daquilo que mais tarde seria conhecido como séries de Fourier, tanto em sua forma discreta, quanto contínua. Tais fórmulas surgiram do estudo de problemas como o da vibração de uma corda com extremidades presas e o da órbita de corpos celestiais, fruto do esforço de matemáticos como d'Alembert, Euler, Bernoulli, Clairaut e Lagrange, sendo que este último teria mais tarde influência direta no trabalho de Fourier.

Em 1807, Fourier apresentou formalmente seu trabalho intitulado “Teoria Analítica do Calor” ao Instituto de Paris, no qual ele afirmava que uma função arbitrária poderia ser expressa como uma série infinita de senos e co-senos. Em 1812 o mesmo trabalho foi enviado à competição matemática do mesmo Instituto, sendo premiado, porém com muitas críticas. Entre os juízes que avaliaram seu trabalho, encontravam-se Lagrange, Laplace e Legendre, os quais argumentaram que as equações e provas apresentadas por Fourier deixavam a desejar em generalidade e rigor, fato que atrasou a publicação de seu trabalho, o que aconteceu somente em 1822.

Paralelamente ao trabalho de Fourier, o qual deu ênfase à forma contínua da transformada que hoje leva seu nome, há registros de que o matemático alemão Gauss teria utilizado a forma discreta desta transformada, que como citado anteriormente já aparecia em estudos de outros matemáticos, em um

trabalho não publicado que utilizava interpolação, datado em 1805, caracterizando a primeira utilização da Transformada Rápida de Fourier.

A Transformada Rápida de Fourier só foi popularizada porém no ano de 1965, através de um trabalho publicado por John Tukey e James Cooley reinventado o algoritmo e descrevendo como implementá-lo convenientemente em um computador. A partir daí, essa transformada gerou grandes repercussões, com muitos estudos a seu respeito e uma vasta gama de aplicações, revolucionando várias áreas da ciência.

O outro objeto de estudo deste trabalho surgiu em 1942 ainda em sua forma contínua, criado por Ralph Vinton Lyon Hartley, um pesquisador na área de eletrônica. Ao contrário da transformada de Fourier que surgiu de problemas matemáticos, principalmente visando a interpolação e aproximação de funções, a transformada de Hartley foi proposta como uma alternativa à transformada de Fourier para algumas de suas aplicações.

A forma discreta da transformada de Hartley e a forma rápida para seu cálculo surgiram porém apenas em 1983 e 1984 respectivamente, através dos trabalhos de Bracewell. Com isto seu campo de aplicação expandiu para muitas áreas onde as transformadas discreta e rápida de Fourier possuem imensa importância. Porém apesar de argumentos de que a transformada de Hartley seria mais eficiente que a de Fourier, pouco se ouviu falar dela nos últimos anos. Por este motivo resolvemos retomar a discussão sobre a eficiência de ambas para concluirmos se Hartley poderia mesmo substituir Fourier completamente, em partes, ou de forma alguma, sendo justificável seu esquecimento.

1.2 DEFINIÇÕES

Até agora citamos fatos históricos e problemas matemáticos que motivaram o surgimento da transformada de Fourier e posteriormente da transformada de Hartley, porém sem nos aprofundarmos em detalhes

matemáticos. Nesta sessão definiremos os conceitos envolvidos nesta pesquisa, assim como sua motivação e significado matemático ou prático.

Como vimos, o surgimento da transformada de Fourier está diretamente relacionada com o problema de aproximação e interpolação de funções. Este problema consiste em utilizar determinados tipos de funções com características de interesse para aproximar outras funções cujo comportamento não apresenta essas características, ou ainda funções cuja expressão matemática é desconhecida. O tipo mais comum de função utilizada para aproximação é o das funções polinomiais, por possuírem muitas propriedades importantes, sendo que seu estudo compõe uma grande área da matemática. Porém neste caso estamos interessados na aproximação de funções através de uma classe especial de polinômios denominada polinômios trigonométricos, os quais definiremos a seguir:

Definição 1.1: Um polinômio trigonométrico de grau m é uma expressão do tipo:

$$P_m(x) = \frac{a_0}{2} + \sum_{k=1}^{m-1} [a_k \cos kx + b_k \sin kx] + a_m \cos mx \quad (1)$$

Nosso interesse aqui é aproximar uma função $f(x)$ através de polinômios trigonométricos. Com isso podemos trabalhar com esta função utilizando características das funções seno e co-seno, além de que através desta aproximação podemos obter o padrão de comportamento de $f(x)$, o que é muito útil em áreas como a de reconhecimento de padrões.

Definição 1.2: Para uma função $y = f(x)$, $x \in [-\pi, \pi]$ com expressão conhecida, sua série de Fourier é a função:

$$F(x) = \lim_{m \rightarrow \infty} P_m(x) = \frac{a_0}{2} + \sum_{k=1}^{\infty} [a_k \cos kx + b_k \sin kx] \quad (2)$$

Onde:

$$a_k = \frac{1}{\sqrt{\pi}} \int_{-\pi}^{\pi} (f(x) \cos kx) dx \quad (3)$$

$$b_k = \frac{1}{\sqrt{\pi}} \int_{-\pi}^{\pi} (f(x) \text{sen } kx) dx \quad (4)$$

Não provaremos aqui a convergência das séries de Fourier, porém de fato isto ocorre quando a função $f(x)$ satisfaz algumas propriedades. Vamos nos restringir apenas a funções contínuas e periódicas de período 2π , sendo que para estas a convergência da série de Fourier é garantida, assim podemos truncar sua série de forma a obter um polinômio trigonométrico aproximador com a precisão desejada. Note-se que qualquer função periódica $f(x)$, com um certo período τ pode ter seu período normalizado para 2π , através de uma mudança de variáveis, tomando $z = 2\pi x / \tau$, (o que implica $x = \tau z / 2\pi$) e definindo a função $g(z) = f(x) = f(\tau z / 2\pi)$, que é periódica de período 2π .

Porém, na prática geralmente não se possui uma função com expressão conhecida e sim um conjunto de pontos discretos que representam dados de imagens ou sons por exemplo, e precisamos detectar seu padrão de comportamento. Para este conjunto discreto podemos também definir uma aproximação via Fourier, de modo que neste caso o polinômio trigonométrico resultante serve também como interpolador.

Definição 1.3: Para um conjunto de $2n$ pares ordenados $\{ (x_i, y_i), i, j = 0..2n-1 \}$, $x_j \in [-\pi, \pi]$, seu ajuste de Fourier de grau m é o polinômio trigonométrico:

$$P_m(x) = \frac{a_0}{2} + \sum_{k=1}^{m-1} [a_k \cos kx + b_k \text{sen } kx] + a_m \cos mx \quad (5)$$

Onde os coeficientes a_k e b_k são obtidos via Mínimos Quadrados, resultando em:

$$a_k = \frac{1}{n} \sum_{j=0}^{2n-1} y_j \cos kx_j \quad (6)$$

$$b_k = \frac{1}{n} \sum_{j=0}^{2n-1} y_j \text{sen } kx_j \quad (7)$$

Agora que já vimos os princípios básicos do método de Fourier para aproximação de funções, podemos partir para a definição de sua transformada propriamente dita. O efeito desta transformada é mapear uma função $X(t)$ cujo domínio é o tempo em uma função $F(k)$ cujo domínio é a freqüência, tendo uma enorme importância em todas as áreas que envolvem o estudo de ondas.

Definição 1.4: Dada uma função contínua $X(t)$ no domínio do tempo, sua transformada contínua de Fourier é definida por:

$$F(k) = \frac{1}{2\pi} \int_{-\infty}^{\infty} X(t)e^{-i2\pi kt} dt \quad (8)$$

E sua inversa é expressa por:

$$X(t) = \int_{-\infty}^{\infty} F(k)e^{i2\pi kt} dk \quad (9)$$

O símbolo i representa o número imaginário $\sqrt{-1}$. Destaque-se que ambas as definições envolvem integrais impróprias e com primitivas difíceis, ou até impossíveis de serem obtidas analiticamente.

Como nos sistemas digitais o sinal não é contínuo mas discreto, obtido através de amostragem, e quantizado, isto é, com duração limitada, fazendo-se uso de ajustamento de curvas por mínimos quadrados a uma base de valores representativos para obter uma forma discreta para as funções $F(k)$ e $X(t)$.

Definição 1.5: Dada um conjunto de pontos $X(t)$, $t=0, 1, \dots, n-1$ no domínio do tempo, sua Transformada Discreta de Fourier (DFT) é definida por:

$$F(k) = \frac{1}{n} \sum_{t=0}^{n-1} X(t)e^{(-i2\pi kt/n)}, \quad k = 0, \dots, n-1 \quad (10)$$

E sua inversa é expressa por:

$$X(t) = \sum_{k=0}^{n-1} F(k)e^{(i2\pi kt/n)}, \quad t = 0, \dots, n-1 \quad (11)$$

Estes são os conceitos fundamentais na análise de Fourier. Com o desenrolar deste trabalho outros conceitos semelhantes aparecerão e serão explicados mais detalhadamente.

Passaremos agora aos conceitos análogos na teoria de Hartley:

Definição 1.6: Seja uma função contínua $X(t)$ no domínio do tempo e com valores reais, sua transformada de Hartley contínua é definida por:

$$H(k) = \frac{1}{2\pi} \int_{-\infty}^{\infty} X(t) \text{cas}(2\pi kt) dt, \quad (12)$$

Onde $\text{cas}(2\pi kt) = \cos(2\pi kt) + \sin(2\pi kt)$

E sua inversa é dada por:

$$X(t) = \int_{-\infty}^{\infty} H(k) \text{cas}(2\pi kt) dk \quad (13)$$

Definição 1.7: Seja um conjunto de pontos reais $X(t)$, $t=0, 1, \dots, n-1$ no domínio do tempo, sua Transformada Discreta de Hartley (DHT) é definida por:

$$H(k) = \frac{1}{n} \sum_{t=0}^{n-1} X(t) \text{cas}(2\pi kt/n), \quad k = 0, \dots, n-1 \quad (14)$$

E sua inversa e dada por:

$$X(t) = \sum_{k=0}^{n-1} H(k) \text{cas}(2\pi kt/n), \quad t = 0, \dots, n-1 \quad (15)$$

Assim, terminamos de apresentar as definições básicas que compõem este trabalho e na seqüência passaremos às próximas etapas: análise, implementação e bateria de testes.

1.3 PROPRIEDADES DAS TRANSFORMADAS DE FOURIER E HARTLEY

Todas as definições apresentadas até agora possuem propriedades muito interessantes, porém algumas matematicamente complexas e fora do escopo de nosso estudo, as quais não serão citadas. Também deixaremos de lado as transformadas contínuas, já que nosso objetivo é utilizar as transformadas para problemas discretos.

Como primeira questão de análise, note-se que as definições (10) e (14) podem também ser representadas como operadores lineares, isto é, operadores que envolvem multiplicação de matrizes. Daí temos que

$$(10) \text{ é equivalente a } F = (1/n) [W] \times [X]$$

$$(14) \text{ é equivalente a } H = (1/n) [U] \times [X]$$

Onde X é o vetor (equivalente a uma matriz coluna) contendo os n pontos de entrada, F e H são os vetores contendo as transformadas de Fourier e Hartley respectivamente para o vetor X , e finalmente W e U são matrizes quadradas de ordem $n \times n$. A título de exemplo, para $n=4$ e considerando para efeito de síntese que $w^j = e^{-i2\pi j/n}$ e $u^j = \cos(2\pi j/n)$, com $j = tk$, resultando em:

$$\begin{bmatrix} F_0 \\ F_1 \\ F_2 \\ F_3 \end{bmatrix} = \frac{1}{n} \begin{bmatrix} w^0 & w^0 & w^0 & w^0 \\ w^0 & w^1 & w^2 & w^3 \\ w^0 & w^2 & w^4 & w^6 \\ w^0 & w^3 & w^6 & w^9 \end{bmatrix} \times \begin{bmatrix} X_0 \\ X_1 \\ X_2 \\ X_3 \end{bmatrix}, \text{ e } \begin{bmatrix} H_0 \\ H_1 \\ H_2 \\ H_3 \end{bmatrix} = \frac{1}{n} \begin{bmatrix} u^0 & u^0 & u^0 & u^0 \\ u^0 & u^1 & u^2 & u^3 \\ u^0 & u^2 & u^4 & u^6 \\ u^0 & u^3 & u^6 & u^9 \end{bmatrix} \times \begin{bmatrix} X_0 \\ X_1 \\ X_2 \\ X_3 \end{bmatrix}$$

Desta forma, percebe-se trivialmente que a complexidade para obter tanto F quanto H no que se refere ao número de operações é da ordem de $O(n^2)$. Porém para F estas operações envolvem números complexos, enquanto para H apenas números reais.

A DHT possui a propriedade de ser sua própria inversa a menos do escalar n , que representa o número de pontos. A DFT, apesar de não tão

diretamente, também pode ter sua inversa explícita a partir dela mesma como: $F^{-1}(x) = F(x^*)^*/n$, onde * representa o conjugado complexo.

Uma propriedade importante que diz respeito à DHT, é a de que podemos substituir a expressão $\cos(2\pi kt / n)$ por $\sqrt{2} \cos(2\pi kt / n - \pi/4)$. Isso ocasiona um ganho imenso em termos de processamento, já que o cálculo de funções trigonométricas representa a parte mais pesada do processamento nestes algoritmos. A igualdade pode ser confirmada utilizando-se a propriedade trigonométrica $\cos(a - b) = \cos a \cos b + \sin a \sin b$, de onde temos que:

$$\sqrt{2} \cos(a - \pi/4) = \sqrt{2} [\cos a \cos \pi/4 + \sin a \sin \pi/4]$$

$$\sqrt{2} \cos(a - \pi/4) = \sqrt{2} [(\sqrt{2}/2) (\cos a + \sin a)]$$

$$\sqrt{2} \cos(a - \pi/4) = \cos a + \sin a$$

Um dos pontos cruciais da comparação entre as duas transformadas e que deve ser observado com cautela é o domínio em que elas agem, já que a transformada de Fourier opera sobre entradas complexas (ou reais, que podem ser consideradas caso particular das complexas) produzindo saídas complexas, enquanto a transformada de Hartley opera apenas no corpo dos números reais. Neste ponto, à primeira vista a DHT parece obter vantagem sobre a DFT, pois utiliza apenas aritmética real, cujos custos tanto de processamento quanto de armazenamento são menores. Para exemplificar, note-se que para armazenarmos um número complexo necessitamos do dobro do espaço necessário para armazenar um número real, a adição de números complexos equivale a duas adições de números reais e a multiplicação de dois números complexos equivale a quatro multiplicações e duas adições reais.

Por outro lado, a maioria das aplicações utiliza dados reais como entrada, pois a maioria das grandezas da natureza é representada através desse conjunto numérico. Além disso, em nosso caso específico, pretendemos comparar os algoritmos de Hartley e de Fourier nas condições mais próximas possíveis, incluindo os mesmos conjuntos de teste, portanto a utilização de dados complexos não faria sentido, já que a DHT só aceita entradas reais.

No caso particular de dados reais, a DFT pode sofrer modificações para ter seu processamento melhorado. Utilizando a fórmula de Euler ($e^{-j2\pi kt} = \cos(2\pi kt) - j \sin(2\pi kt)$), podemos separar a parte real e imaginária da DFT e calculá-las separadamente utilizando apenas aritmética real. Nesse caso a fórmula da DFT ficaria assim:

$$F(k) = \frac{1}{n} \sum_{t=0}^{n-1} X(t) [\cos(2\pi kt/n) - j \sin(2\pi kt/n)] \quad (16)$$

Separando as partes reais e imaginárias temos:

$$\text{Re}\{F(k)\} = \frac{1}{n} \sum_{t=0}^{n-1} X(t) \cos(2\pi kt/n) \quad (17)$$

$$\text{Im}\{F(k)\} = -\frac{1}{n} \sum_{t=0}^{n-1} X(t) \sin(2\pi kt/n) \quad (18)$$

Alguns autores consideram este não apenas como um caso particular, mas sim como uma outra transformada, denominada DFT Real, ou RDFT. Esta será a representante entre as transformadas de Fourier com a qual trabalharemos deste ponto em diante. Há ainda uma outra característica interessante nos resultados gerados por ela. Dado um conjunto de n entradas reais, o significado deste conjunto para as transformadas em questão é que eles representam valores igualmente espaçados dentro do intervalo equivalente a um período da função de tempo, portanto $F(0) = F(n)$, sendo este um número real. Podemos observar também que para n par, $F(n/2)$ também é real, e os outros pontos de saída satisfazem a propriedade $F(k) = F(n - k)^*$.

Com isso, podemos calcular a DFT apenas para metade das entradas, sendo que para os outros pontos basta espelhar os resultados mudando o sinal da parte imaginária, o que causa uma redução em quase 50% das operações, o que lhe dá alguma vantagem sobre Hartley, que não possui essa simetria.

Ainda não explicitamos a fórmula inversa da DFT real, porém sabemos que dados os pontos complexos resultantes da aplicação da transformada, utilizando sua inversa podemos reconstruir os dados de entrada. Através de um raciocínio simples podemos então notar que a transformada inversa está

diretamente relacionada com a aproximação e interpolação via Fourier. Mais precisamente, o ajuste via Fourier dado pela definição 1.3 é a inversa da DFT real, com coeficientes $a_k = 2 \operatorname{Re}\{F(k)\}$, e $b_k = -2 \operatorname{Im}\{F(k)\}$.

Outra propriedade muito interessante e que será muito útil em nosso estudo, é a de que através da DHT podemos obter a DFT e vice-versa. Explicitando a fórmula real para a DFT fica claro que os coeficientes da DHT podem ser obtidos por $H(f) = \operatorname{Re}\{F(k)\} - \operatorname{Im}\{F(k)\}$. O contrário não é tão evidente, porém de fato, temos que:

$$\operatorname{Re}\{F(k)\} = \frac{H(n-k) + H(k)}{2}$$

$$\operatorname{Im}\{F(k)\} = \frac{H(n-k) - H(k)}{2}$$

Após analisarmos estas propriedades, temos o material necessário para começarmos as implementações e testes em com a DFT e a DHT.

CAPÍTULO 2 – ANÁLISE COMPARATIVA ENTRE A DHT E A DFT

Neste capítulo começaremos finalmente a entrar no objetivo principal deste trabalho, que é o de fazer comparações. A primeira questão que devemos levar em conta para isso é nos certificarmos que ambos os objetos a serem comparados estão submetidos a condições iguais, ou seja, não há fatores externos que forneçam qualquer vantagem a um dos envolvidos. Certamente poderíamos encontrar várias implementações da transformada de Fourier (até mesmo algumas da transformada de Hartley) com uma rápida pesquisa na internet, porém os detalhes de cada implementação, o método como os cálculos são realizados, particularidades de linguagens, e até mesmo otimizações poderiam colocar uma das transformadas em vantagem. Por esse motivo foi optado por desenvolver implementações próprias semelhantes para a DHT e a DFT, de forma que as diferenças sejam apenas as conceituais.

O primeiro passo para a comparação entre as transformadas de Hartley e de Fourier foi a implementação e teste nas condições mais semelhantes possíveis das suas versões discretas conceituais. Neste ponto não estamos usando nenhum algoritmo para acelerar o cálculo dos coeficientes, estamos apenas utilizando as propriedades vistas no capítulo anterior sobre simetria e equivalência de expressões para a economia de cálculos. Com certeza esta não é a maneira mais eficiente de se efetuar ambas as transformadas, porém pode fornecer uma prévia em termos absolutos sobre a relação de tempo de processamento entre as duas transformadas no que diz respeito às operações envolvidas conceitualmente em seu cálculo.

Para a implementação foi utilizada a linguagem orientada a objetos C++, com a IDE Borland C++ Builder. A razão desta escolha foi o fato de que entre as linguagens de domínio do programador, esta é a que apresenta maior desempenho, enquanto a IDE escolhida oferece vantagens e facilidades para a programação, principalmente no que diz respeito à facilidade para criar uma interface gráfica na qual os resultados pudessem ser analisados.

Primeiramente foi criada uma classe na qual foram codificados os algoritmos para a obtenção da DFT, DHT e suas respectivas inversas. Juntamente com isso foi criada uma interface gráfica, utilizando-se os recursos da IDE, para que os resultados da aplicação das transformadas pudessem ser visualizados e sua correção verificada. A parte principal da interface gráfica consiste em uma lista na qual pode-se digitar pontos manualmente ou gerar um número desejado de pontos através de uma função pré-definida, botões de teste que ativam as transformadas implementadas sobre os dados de entrada e outras listas nas quais serão mostrados os dados de saída. Além disso, fazem parte da interface outros elementos gráficos que seriam úteis mais adiante no momento da comparação de velocidade.

Após ser feita a verificação dos algoritmos implementados, o resultado desejado foi obtido e suas características foram analisadas, confirmando todas as propriedades citadas anteriormente na parte teórica. Com isso os algoritmos já se encontram prontos para serem testados. Porém, antes de testá-los quanto ao seu tempo de processamento, pode-se fazer uma análise através do algoritmo para a estimativa do número de operações demandado por cada uma. Como exemplo, será analisada a seguir a DHT implementada e para isso o código será dividido em fragmentos, sendo consideradas apenas as linhas que contém operações relevantes.

```
double pi2_N = (M_PI + M_PI) / N;
```

1 adição + 1 divisão

```
for ( int k = 0; k < N; k++ )
```

n-1 adições (incrementos)

```
k2pi_N = k * pi2_N;
```

Repetido n vezes = n multiplicações

```
for ( int t = 0; t < N; t++ )
```

n-1 incrementos, repetidos n vezes = $n^2 - n$ adições

resultado[k] += M_SQRT2 * dados[t] * cos((k2pi_N * t) - M_PI_4);
n² (2 adições + 3 multiplicações + 1 cosseno)

resultado[k] = resultado[k] / N;
n divisões

As seguintes operações são necessárias caso deseje-se obter a saída da transformada de Fourier através da saída de Hartley:

for (int j = 1; j <= N * 0,5; j++)
1 multiplicação + [(n/2) - 1] adições

real[j] = (resultado[N - j] + resultado[j]) * 0.5;
img[j] = (resultado[N - j] - resultado[j]) * 0.5;
[(n/2) - 1] (4 adições + 2 multiplicações)

Vale frisar que estamos considerando os incrementos de variáveis inerentes a estruturas de repetição como adições, porém dependendo do processador e da otimização feita pelo compilador, estas operações podem ser menos dispendiosas do que as adições comuns.

Similarmente ao que fizemos no caso da DHT podemos estimar as operações necessárias para calcular a DFT implementada (em sua versão otimizada), obtendo a seguinte tabela.

	Adições	Multiplicações	Divisões	Cosseno	Seno
DHT	$3n^2$	$3n^2 + n$	$n + 1$	n^2	0
DFT via DHT	$3n^2 + 2,5n - 5$	$3n^2 + 2n - 1$	$n + 1$	n^2	0
DFT	$n^2 + 2n - 1$	$n^2 + n/2 + 1$	$n + 2$	$n^2/2$	$n^2/2$

Tabela 2.1: Operações para os algoritmos DFT e DHT implementados

Agora podemos passar à etapa de testes práticos para avaliar o desempenho de ambas as transformadas. A idéia é testar os algoritmos para diversos números de pontos e observar seu comportamento e a relação de tempo entre a DFT e a DHT e se essa relação é sempre a mesma ou se modifica conforme o número de pontos cresce.

Como já havia sido mencionado, devemos realizar os testes com o mínimo possível de influências externas, portanto devemos rodá-los em um computador sem outros programas simultâneos e com prioridade máxima para o processo (as threads foram configuradas para real time durante o processamento das transformadas). Mesmo assim, ainda podem ocorrer interferências no processo, portanto a estratégia adotada será a de repetir cada teste e tomar como resultado do tempo demandado a média dos tempos calculados para um mesmo número de pontos. Inicialmente a interface gráfica disponibilizava uma barra de progresso para que o usuário tivesse conhecimento da porcentagem do processamento dos dados para ter uma idéia do tempo que ainda faltaria para seu término. Porém a atualização da interface gráfica estava influenciando muito no tempo de processamento dos dados, e por isso foi retirada, fato que tornou mais difícil o acompanhamento dos testes, porém necessário.

Para os testes foi utilizando um computador com processador de 1,5GHz e 256MB de RAM rodando Windows XP. Os algoritmos testados foram, DHT, DHT com obtenção dos coeficientes de Fourier, DFT otimizado por simetria e DFT não otimizado, que apesar de não ser útil na prática, fornece uma idéia da demanda que seria necessária caso não houvesse simetria e todos os pontos tivessem que ser processados.

Os testes a seguir foram feitos dividindo-se a função a função $f(x) = x^4 - 3x^3 + x^2 - \tan x(x - 2)$, em n pontos igualmente espaçados no intervalo $[0, 2]$, porém qualquer que seja a função gerada para a obtenção dos pontos o efeito é o mesmo, já que estamos interessados apenas na análise do processamento, e não na interpretação matemática dos resultados.

Pontos	DHT	DFT por DHT	DFT simétrica	DFT não sim.
1024	194ms	202,2ms	176,4ms	346,6ms
2048	795,2ms	803,2ms	717ms	1406ms
4096	3,18s	3,2s	2,87s	5,57s
8192	12,57s	12,86s	11,6s	22,3s
16384	50,409s	51,6462s	46,15s	91,14s
32768	3,38min	3,39min	3,12min	6,17min
65536	13,59min	14,26min	12,85min	25,63min
131072	55,43min	58,18min	52,46min	104,59min
*262144	3,77h	3,96h	3,57h	7,11h
*524288	15,38h	16,14h	14,56h	29,02h
*1048576	62,74h	65,86h	59,39h	118,38h

Tabela 2.2: Tempo de processamento por número de pontos. (dados estimados).*

A idéia inicial desta tabela era realmente fazer os testes para todos os números de pontos sugeridos, porém conforme este número foi aumentando, seu teste tornou-se impraticável. No entanto, o comportamento dos testes mostrou uma tendência evidente que permitiu que o tempo de processamento fosse estimado para mais pontos.

Os resultados dos testes comprovaram o fator determinante de operações n^2 , sendo que ao se dobrar o número de pontos o tempo quadruplicou. No caso da DHT e da DFT os dados de entrada não necessariamente devem ser potências de 2, mas essa opção foi feita para possibilitar futuras comparações com algoritmos rápidos (FFT, FHT) que necessitam deste tipo de entradas.

O gráfico para a tabela 2.2 foi dividido em 4 gráficos menores para facilitar sua visualização em escalas diferentes.

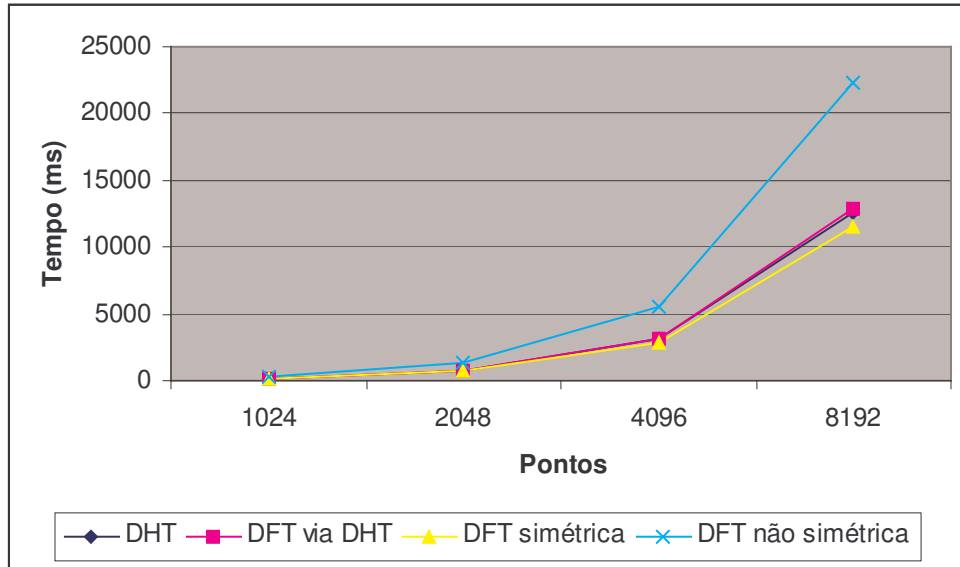


Gráfico 2.1: Tempo de processamento em milissegundos

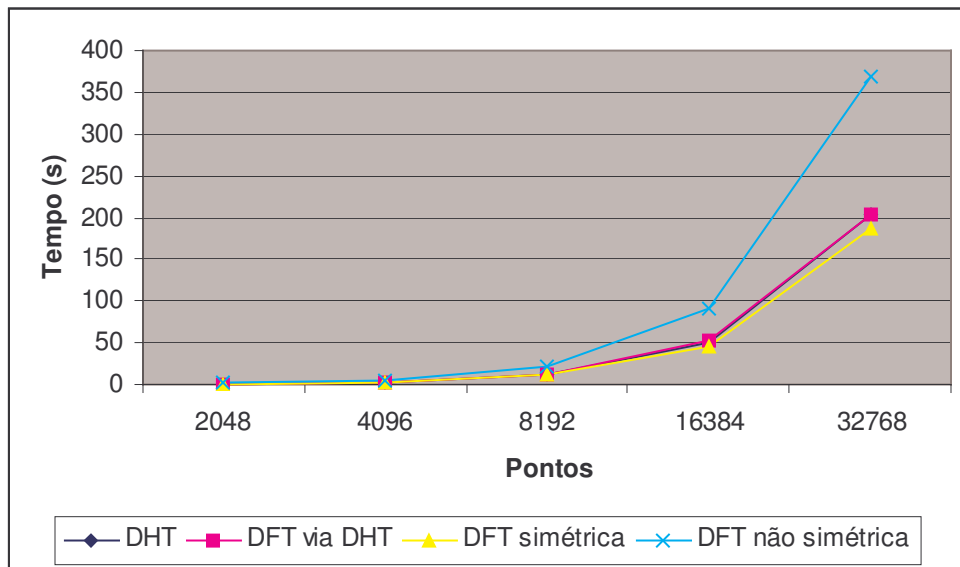


Gráfico 2.2: Tempo de processamento em segundos

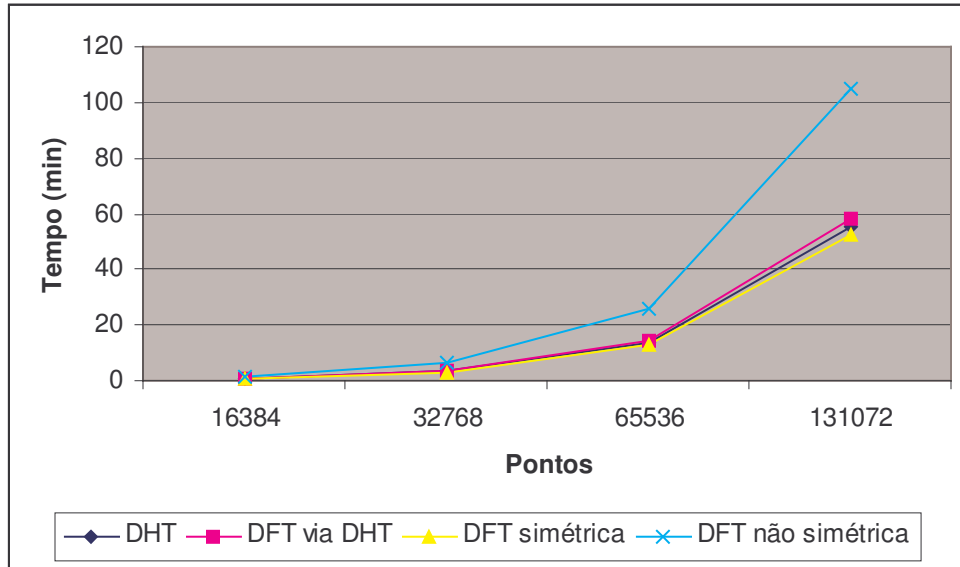


Gráfico 2.3: Tempo de processamento em minutos

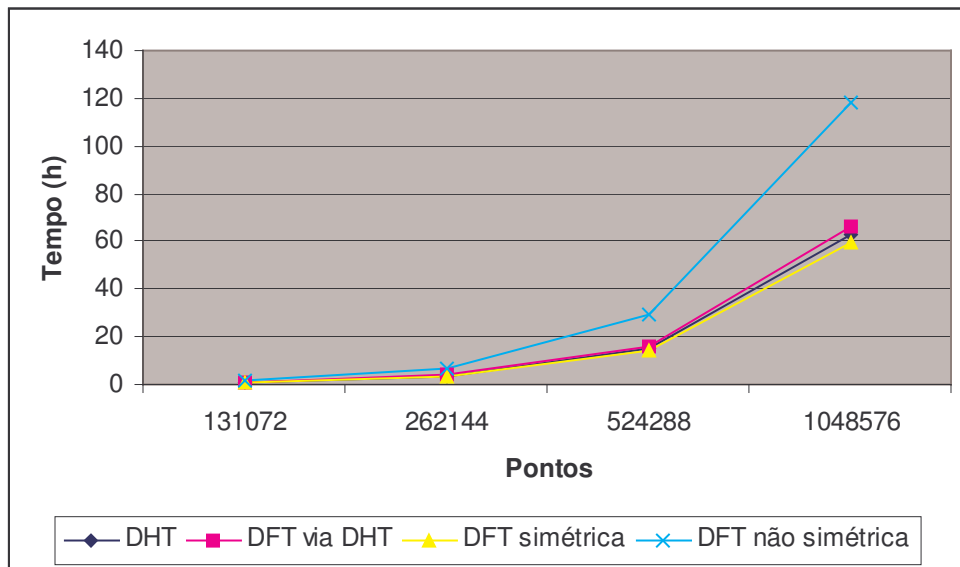


Gráfico 2.4: Tempo estimado de processamento em horas

Podemos notar a semelhança entre os gráficos acima que mostram que o padrão de comportamento do processamento em função dos dados é sempre o mesmo. O fato do eixo do número de pontos estar não em escala linear faz com que o gráfico mostre uma tendência exponencial, enquanto na verdade estas

funções possuem comportamento quadrático, que pode ser claramente observado no gráfico 2.6, que melhor representa o fenômeno.

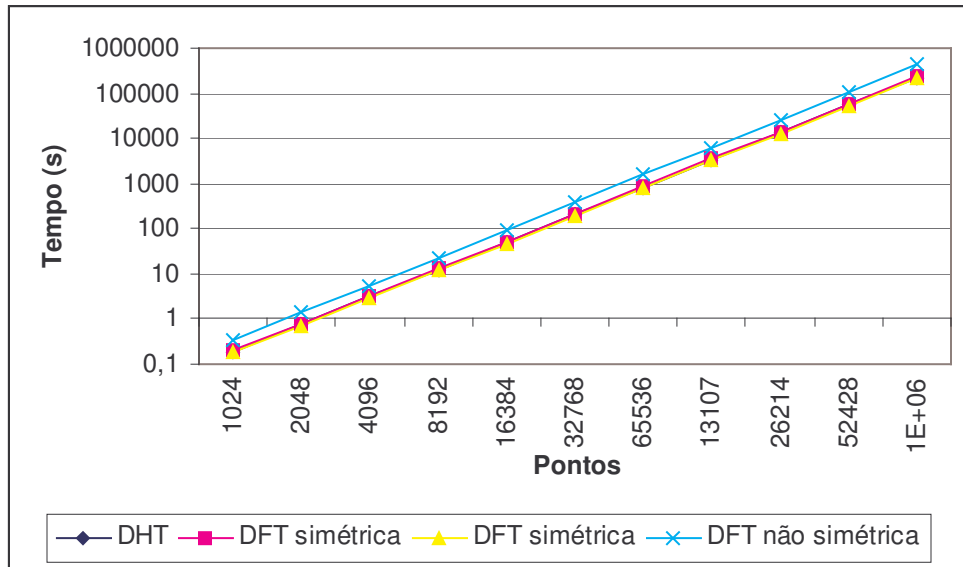


Gráfico 2.5: Tempo de processamento em escala logarítmica

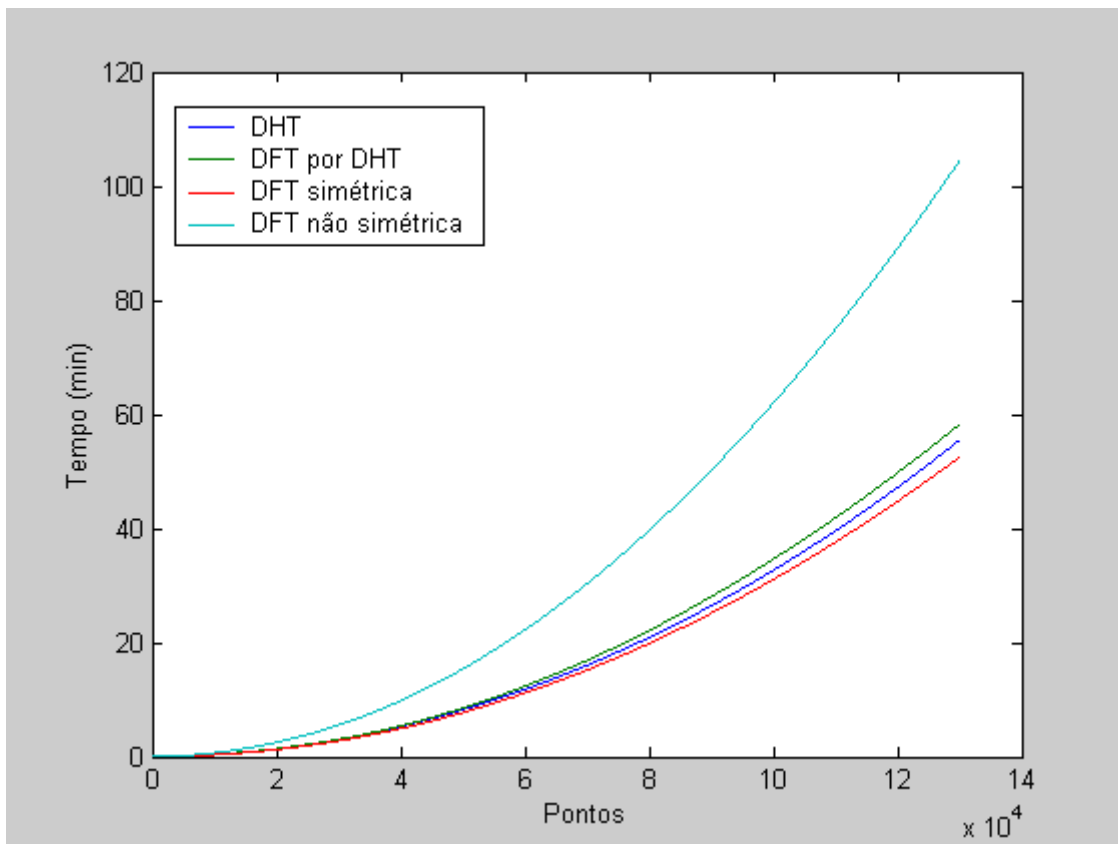


Gráfico 2.6: Gráfico linear interpolado do tempo de processamento

Os resultados práticos obtidos nos testes realizados comprovam as estimativas feitas anteriormente de que menos operações são necessárias para o cálculo da DFT otimizada para números reais através de simetria, do que para o cálculo da DHT. Porém, podemos notar que não fosse essa simetria, a DHT seria muito mais rápida, principalmente devido ao fato de utilizar metade das operações de cálculo de seno e cosseno necessárias para a obtenção da DFT sem otimização.

Já no que diz respeito à memória consumida, a DHT necessita armazenar n números reais como resultado e durante todo o processo apenas este consumo é significativo. Poderíamos pensar que a quantidade de memória ocupada pela DFT é o dobro da DHT, porém para sua versão otimizada, note que há a necessidade apenas de armazenar metade dos valores ($(n/2)+1$ valores reais e $(n/2)-1$ valores imaginários), pois os outros podem ser obtidos a partir destes. Logo o consumo de memória é equivalente.

Até aqui vemos que conceitualmente a transformada de Hartley não apresenta vantagens em relação à transformada de Fourier, porém sabemos que estas transformadas discretas não possuem utilização prática, portanto devemos partir para os testes com as transformadas rápidas para obtermos uma conclusão mais significativa.

CAPÍTULO 3 – TRANSFORMADAS RÁPIDAS

Vimos em teoria e comprovamos na prática que para um conjunto de n pontos são necessárias um fator de n^2 operações para calcular sua transformada discreta de Fourier ou de Hartley, de modo que ao duplicarmos o número de pontos de entrada, o tempo de processamento mais do que quadruplica.

Através dos testes efetuados pode-se estimar que o tempo de processamento para 10^6 pontos (um número comum ou até pequeno em aplicações reais) ficaria em torno de 60 horas para o computador de testes utilizado. Por isso a utilização direta destas transformadas é impraticável, e devido a isso surgiram as transformadas rápidas.

Primeiro será explicado o princípio da Transformada Rápida de Fourier (FFT), o que facilitará o entendimento da Transformada Rápida de Hartley (FHT), pois esta é muito semelhante, tendo como principal diferença uma assimetria não existente na FFT.

3.1 TRANSFORMADA RÁPIDA DE FOURIER (FFT)

Quando mencionamos FFT não nos referimos a um único algoritmo ou expressão matemática e sim a uma família de algoritmos e métodos para se calcular a DFT. Porém, em todos esses métodos o raciocínio é semelhante e pode ser explicado matematicamente de várias formas.

Uma das maneiras de explicar-se a FFT é utilizando sua forma matricial já explicitada anteriormente. Como a DFT pode ser considerada uma multiplicação de matrizes, podemos utilizar um princípio muito comum na álgebra linear que é o de utilizar propriedades de matrizes para fatorá-las e simplificar a obtenção de seu produto. Para os leigos, a fatoração de matrizes pode não parecer uma simplificação, já que transforma uma matriz em várias outras, sendo que uma multiplicação de matriz se transforma em várias multiplicações. Porém, com a utilização das propriedades das matrizes fatoradas, que podem ser simétricas,

ortogonais, esparsas, entre outros, estas multiplicações sucessivas tornam-se triviais, gerando menos esforço computacional do que a operação original.

Porém não nos aprofundaremos na explicação matricial da FFT e sim em um outro princípio chamado método da divisão, pois este consiste na explicação mais comum para a FFT, sendo que a implementação realizada neste trabalho foi baseada nele. Vamos assumir a partir de agora que o número de pontos de entrada será sempre uma potência de dois.

Este método consiste em bisseccionar os dados de entradas em dois outros sub-conjuntos e calcular a transformada para esses dois sub-conjuntos separadamente e depois combinar os dois sub-conjuntos de resultados obtendo a transformada original. Note que os subconjuntos gerados também podem ser divididos gerando um processo recursivo que termina no cálculo da transformada para um único ponto, sendo que ele é sua própria transformada.

Para entender o método da divisão, considere a transformada de Fourier no ponto k , desconsiderando o fator $1/n$ que pode ser multiplicado no fim do cálculo, utilizando a notação $w_n^{kt} = e^{i2\pi kt/n}$, temos que:

$$F(k) = \sum_{t=0}^{n-1} X(t)w_n^{-kt}$$

Agora separaremos $X(t)$ em dois outros conjuntos Y , Z de tamanho $n/2$, sendo $Y(t) = X(2t)$ e $Z = X(2t+1)$ para k indo de 0 a $(n/2) - 1$. Podemos dizer que Y é o subconjunto par e Z o subconjunto ímpar de X , e podemos reescrever $F(k)$ como:

$$F(k) = \sum_{t=0}^{(n/2)-1} Y(t)w_n^{-2kt} + \sum_{t=0}^{(n/2)-1} Z(t)w_n^{-(2t+1)k} \quad (19)$$

Note agora a seguinte igualdade:

$$w_n^{-2kt} = e^{-i4\pi kt/n} = e^{-i2\pi kt/(n/2)} = w_{n/2}^{-kt} \quad (20)$$

Com isso podemos reescrever (19) da seguinte forma:

$$F(k) = \sum_{t=0}^{(n/2)-1} Y(t)w_n^{-kt} + w_n^{-k} \sum_{t=0}^{(n/2)-1} Z(t)w_n^{-kt} \quad (21)$$

Podemos notar que $F(k)$ está expresso em função de duas transformadas menores:

$$F(k) = F(Y(k)) + w_n^{-k} F(Z(k)), \quad k = 0 \dots (n/2)-1 \quad (22)$$

Considerando que $w_n^{-n/2} = -1$, e que $Y(k)$ e $Z(k)$ são periódicos de período $n/2$ (pois X é periódico de período n), temos que:

$$F(k+(n/2)) = F(Y(k)) - w_n^{-k} F(Z(k)), \quad k = 0 \dots (n/2)-1 \quad (23)$$

Assim vemos que pode-se obter a transformada de um conjunto de pontos através da transformada de seus subconjuntos par e ímpar, e a aplicação sucessiva deste processo caracteriza a FFT. Normalmente as combinações são feitas duas a duas através de um diagrama conhecido como borboleta, gerando o efeito colateral de trocar os pontos de posição, o que pode ser facilmente corrigido através de uma permutação conhecida como reversão de bits, pois para se obter o índice da transformada final deve-se transformar o número em binário e espelhá-lo. Por exemplo para 16 pontos (representados por 4 bits), $F(4)$ está na posição 2, pois 4 em binário é 0100, que espelhando fica 0010, que é 2.

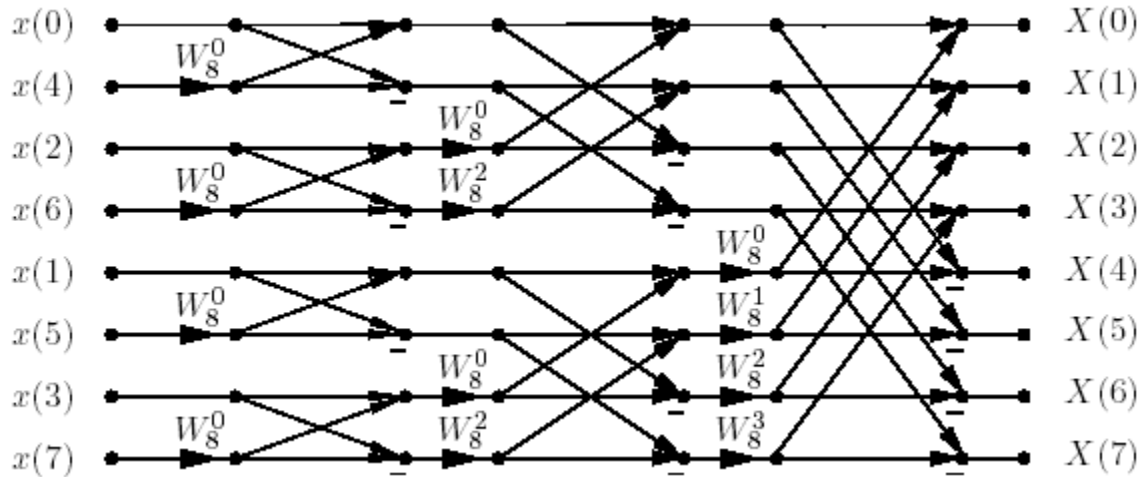


Figura 3.1: Diagrama borboleta para a FFT

3.2 – TRANSFORMADA RÁPIDA DE HARTLEY (FHT)

O princípio da FHT é muito semelhante ao da FFT, porém com uma diferença crucial. O raciocínio é o mesmo, dividir o conjunto em dois sub-conjuntos Y e Z, sendo Y o subconjunto par e Z o subconjunto ímpar. Porém desta vez a decomposição não depende apenas de dois termos, e sim de três, gerando uma assimetria. A fórmula de decomposição é a seguinte:

$$H(k) = H(Y(k)) + H(Z(k)) \cos(2\pi k / n) + H(Z(n - k)) \text{sen}(2\pi k / n) \quad (24)$$

$$H(k + (n/2)) = H(Y(k)) - H(Z(k)) \cos(2\pi k / n) - H(Z(n - k)) \text{sen}(2\pi k / n) \quad (25)$$

A figura abaixo ilustra esta decomposição:

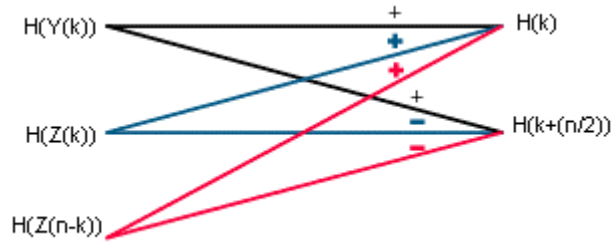


Figura 3.2: decomposição da FHT

O diagrama completo para uma base de 16 pontos fica assim:

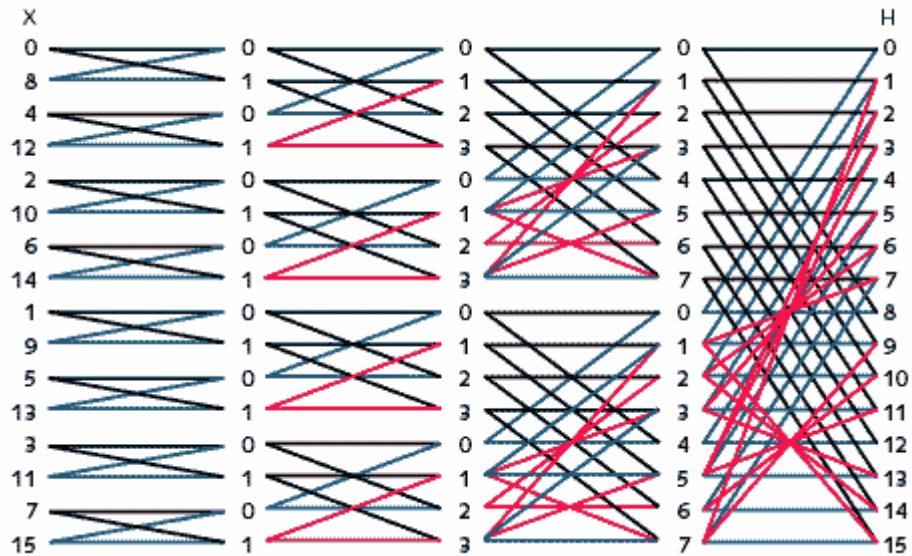


Figura 3.3: Diagrama borboleta para a FHT

Podemos notar que com a utilização dessa estratégia, são necessários apenas $\log_2 n$ passos para se bisseccionar toda a base de dados, portanto o número de operações necessárias para se calcular cada uma das transformadas não é mais da ordem de $O(n^2)$ e sim da ordem de $O(n \log_2 n)$.

3.3 – FHT VERSUS FFT

Com a fundamentação dos princípios nos quais se baseiam as famílias de transformadas rápidas, partimos para a implementação prática da FFT e da FHT para o teste de desempenho entre as duas. A abordagem das implementações foi a mesma para a comparação entre a DHT e a DFT, sempre tentando manter as maiores semelhanças e igualdade de condições para o funcionamento de ambas. A interface utilizada foi a mesma, deixando novamente de lado os elementos gráficos para focar no processamento. Foram criados três métodos novos na classe já existente para o cálculo das transformadas. Um método para calcular a FFT, outro para calcular a FHT e outro para obter a FFT utilizando os cálculos da FHT.

Na questão da comparação entre as transformadas há um detalhe que devemos lembrar dos testes entre a DFT e a DHT que foi a otimização da DFT para números reais. Essa otimização foi feita a partir de uma propriedade que surge quando se tem apenas números reais nas entradas que torna fácil a obtenção da segunda metade dos valores da saída, tendo-se a primeira metade, não sendo necessário calcular a transformada para todos os pontos e sim para apenas metade deles, reduzindo os cálculos praticamente à metade. Porém esta otimização não pode ser feita trivialmente na FFT, já que não se pode calcular apenas metade dos valores, já que o algoritmo da borboleta faz com que todas as saídas sejam calculadas simultaneamente sendo todas dependentes umas das outras. Neste caso também não se pode fugir das multiplicações entre números complexos, que consistem na parte mais dispendiosa de processamento, já que independente de as entradas serem apenas reais, a cada passo da borboleta há uma multiplicação por uma exponencial complexa, gerando um número complexo que no próximo passo multiplicará outro número complexo e assim por diante.

Foi visto anteriormente que para a DFT e DHT o número de pontos pode ser qualquer, porém para a FFT e FHT vistas e implementadas aqui é necessário que este número seja uma potência de 2. Os testes realizados com DFT e DHT já

foram feitos com potências de 2 para a comparação com as transformadas rápidas.

Os testes com a FFT e FHT, assim como no caso das transformadas não-rápidas, começaram a partir de 1.024 (2^{10}) pontos, porém com esta quantidade o tempo de resposta dos algoritmos foi inferior a 10 ms, impossibilitando qualquer comparação devido à falta de precisão no medidor de tempo utilizado. Para 2^{11} pontos o mesmo fato ocorreu. Devido a isso, as comparações efetivas foram iniciadas a partir de 2^{14} (16.384) pontos, quando o tempo médio ficou em torno de 80ms para a FHT, 90ms para a obtenção de Fourier através da FHT e 124ms para a FFT diretamente.

Prosseguiram-se os testes dobrando-se sucessivamente a quantidade de pontos, de forma a sempre haver uma potência de dois no número dos dados de entrada. Até 2^{21} pontos os resultados seguiram a mesma tendência, com velocidade satisfatória. Porém ao serem dobradas as entradas novamente (2^{22} pontos), a memória RAM do computador de testes não foi mais suficiente para os algoritmos, fazendo com que o programa alocasse em disco rígido uma parte dos dados processados, o que causou um atraso desproporcional nos cálculos, já que o acesso a disco é muito lento se comparado com o acesso à memória RAM. Como também não podemos ter controle sobre quantos dados são alocados em disco para cada algoritmo, não foi mais possível a comparação de eficiência e deu-se por encerrado o teste com dados de desempenho para até 2^{21} pontos.

A tabela a seguir mostra o resultado obtido, seguindo a estratégia anterior: são apresentados os valores de tempo de processamento obtidos para cada algoritmo através da média de 5 testes. Os dados da tabela encontram-se em segundos.

Pontos	FHT	FFT via FHT	FFT
16384	0,080	0,090	0,124
32768	0,180	0,194	0,249
65536	0,393	0,404	0,460
131072	0,877	0,911	1,085
262144	1,829	1,865	2,239
524288	3,907	3,976	4,743
1048576	8,291	8,424	9,948
2097152	17,519	17,738	20,920

Tabela 3.1: Tempo de processamento por número de pontos (em segundos)

Foi optado novamente pela divisão da tabela em mais de um gráfico para facilitar sua visualização, porém desta vez apenas dois gráficos foram suficientes, já que a diferença no tempo de processamento de cada etapa não foi tão brusca.

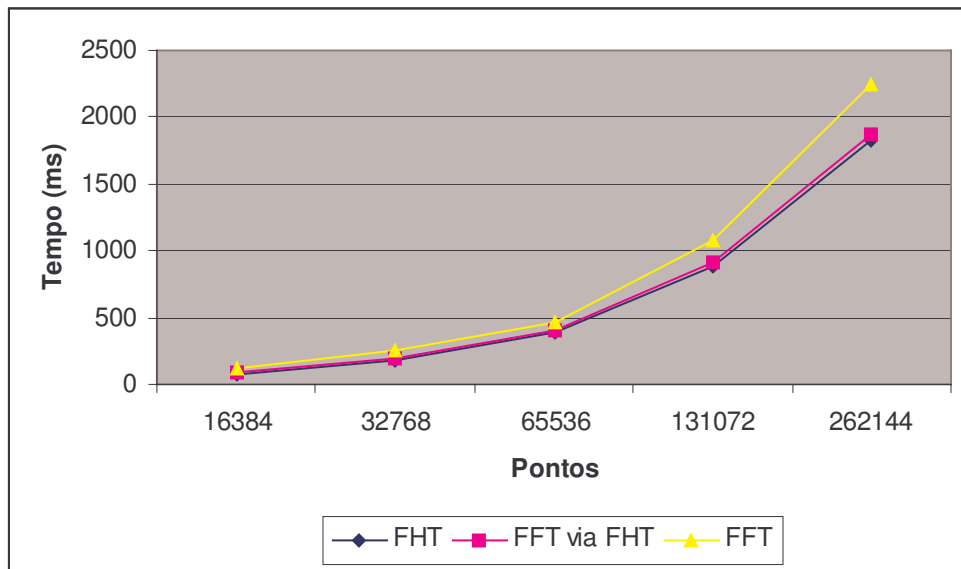


Gráfico 3.1: Tempo de processamento em milissegundos para as transformadas rápidas

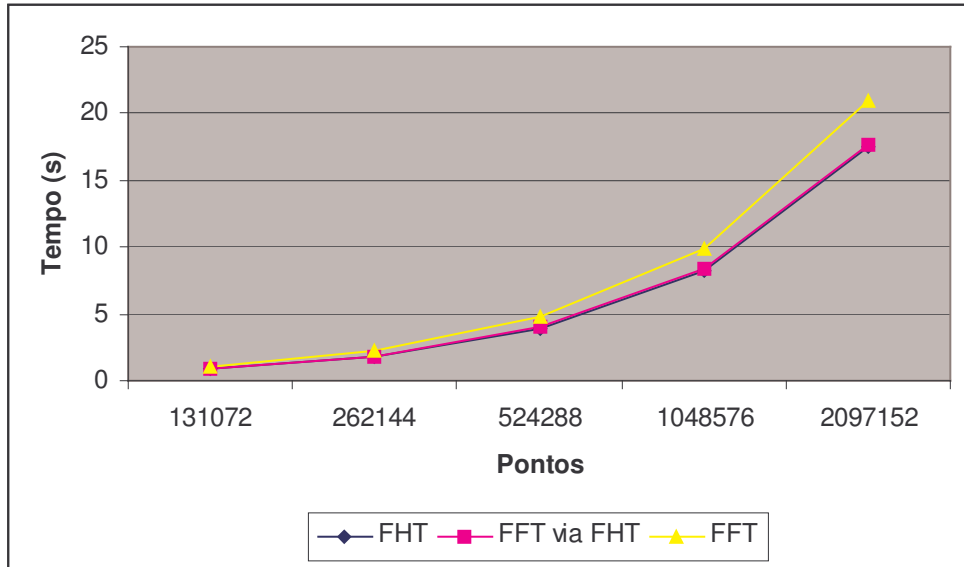


Gráfico 3.2: Tempo de processamento em segundos para as transformadas rápidas

Novamente notamos o comportamento semelhante entre os gráficos que mostra uma mesma tendência sempre, porém desta vez o tempo de processamento não mais quadruplica ao serem duplicadas as entradas. Em vez disso o tempo fica próximo de duplicar, o que mostra uma tendência quase que linear. Na verdade, como vimos, a tendência de processamento é da ordem de $O(n(\log_2 n))$.

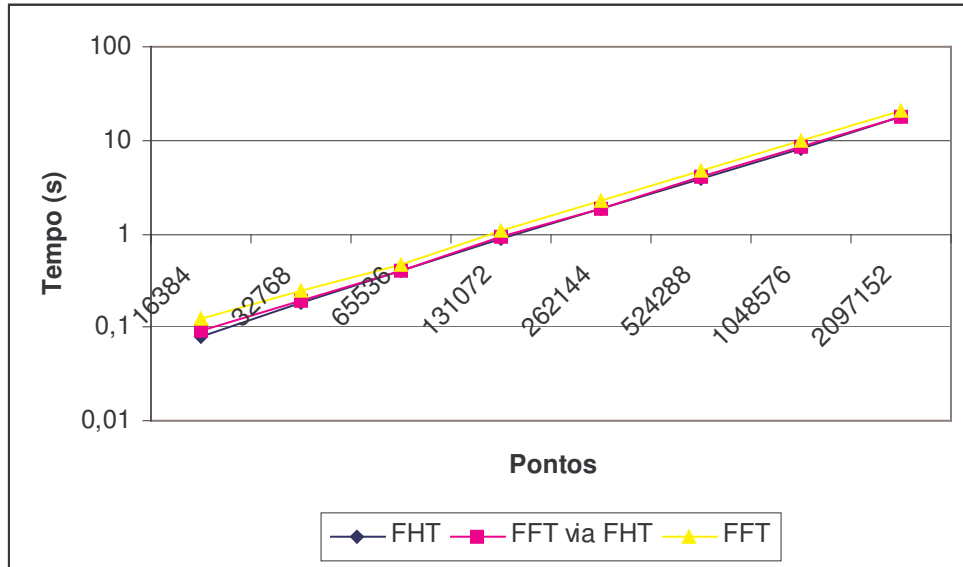


Gráfico 3.3: Tempo de processamento em escala logarítmica para as transformadas rápidas

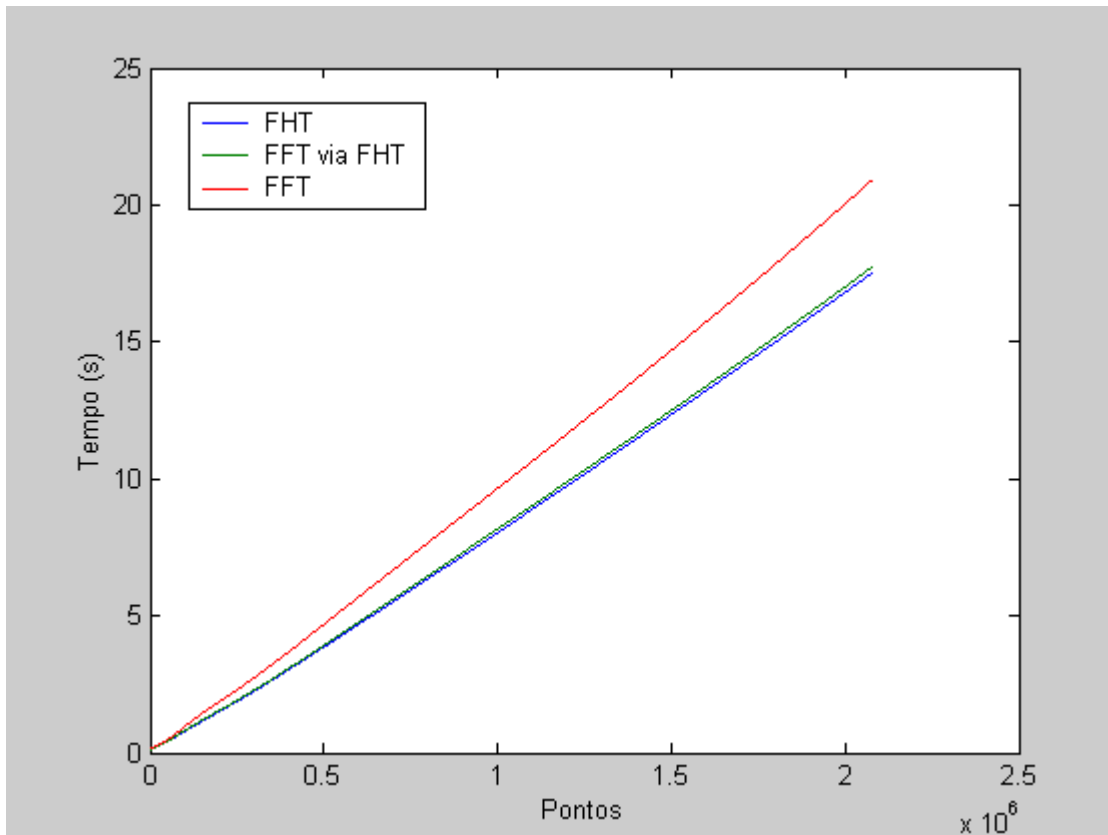


Gráfico 3.4: Gráfico linear interpolado do tempo de processamento para as transformadas rápidas

O gráfico 3.4 mostra os dados dispostos em escala linear, evidenciando a tendência de comportamento quase linear das transformadas rápidas. Assim, podemos notar um ganho absurdo de velocidade em relação às transformadas não rápidas testadas anteriormente.

Quanto à comparação entre as transformadas rápidas, percebe-se que a FHT obteve melhor resultado do que a FFT, e mesmo a obtenção da FFT utilizando a FHT foi mais rápida que a própria FFT. Esta última obteve resultados muito próximos aos da FHT, pois foi implementada substituindo-se o último passo da FHT que consiste na normalização dos dados (divisão por n), e com poucas operações extras já eram obtidas as saídas da FFT.

Os resultados práticos comprovaram os estudos teóricos, sendo que podemos justificar o melhor desempenho da FHT pelo fato de a FFT utilizar cálculos complexos.

Quanto ao consumo de memória, tanto a FHT quanto a FFT necessitam armazenar os resultados obtidos no passo anterior para calcular o passo seguinte, por tanto o custo principal de armazenamento durante os cálculos para a FHT é de $2n$ números reais e para a FFT $2n$ números complexos, os quais podem ser representados por $4n$ números reais. Note que na DFT pode-se armazenar apenas metades dos dados obtidos, já no caso da FFT, pelo menos durante a etapa de processamento é necessário armazenar todos os dados para as combinações entre eles. Assim temos que o consumo de memória da FFT é o dobro do consumo de memória da FHT.

Com isso encerra-se a etapa de testes e comparações entre as transformadas rápidas, sendo os resultados finais favoráveis à FHT. Porém, a conclusão deste trabalho levantará outras questões sobre as quais deve-se refletir para uma melhor interpretação dos dados contidos aqui.

CONCLUSÃO

Neste ponto conclui-se uma longa etapa que consistiu em pesquisa, análise, implementação e testes. Muitas dificuldades foram encontradas e barreiras transpostas, podendo considerar-se que a principal delas foi a falta de bibliografia sobre a transformada de Hartley e a dificuldade de acesso a ela. Mesmo quando teve-se acesso ao principal artigo que motivou esta pesquisa, notou-se que ele encontrava-se incompleto e com erros, fato que atrasou e dificultou muito a compreensão e a implementação do método da borboleta para a transformada rápida de Hartley.

As informações encontradas sobre a transformada rápida de Hartley também foram contraditórias, sendo que em alguns lugares dizia-se que seu desempenho era muito melhor que o da transformada rápida de Fourier, enquanto outras fontes desmentiam tal afirmação. Isso porém motivou nosso esforço na comparação entre as duas transformadas, para que pudéssemos chegar a uma conclusão satisfatória sobre o assunto.

No entanto, apesar dos esforços para as implementações com maior similaridade possível e testes em iguais condições, sem oferecer vantagens quaisquer a algum dos objetos da comparação em questão, obtendo resultados muito claros sempre dentro de uma tendência fácil de ser visualizada tanto em tabelas quanto em gráficos, mesmo assim não podemos cair na armadilha de inferir conclusões precipitadas.

Pode-se notar pelos resultados aqui expostos, que na primeira bateria de testes realizada, o melhor desempenho foi obtido pela DFT otimizada para números reais. Esse resultado, como vimos explica-se pelo fato de quando termos entradas reais a DFT pode ser otimizada devido à simetria dos dados de saída, sendo necessário apenas o cálculo da transformada para metade dos pontos. Além disto, este fato também faz com que o resultado da transformada possa ter suas partes real e imaginária calculadas separadamente apenas com operações utilizando aritmética de números reais, sendo que não há necessidade de

multiplicações complexas, cujo custo é alto se comparado às outras operações em questão.

Na segunda bateria de testes realizados, desta vez comparando as transformadas rápidas que são as que possuem efetivamente utilização prática, vimos que a transformada rápida de Hartley obteve sempre melhor resultado do que a já consagrada FFT. Será que então deveríamos concluir que a FHT é melhor que a FFT e propormos sua substituição por completo? Certamente não!

Vários aspectos devem ser levados em consideração antes de formularmos uma afirmação concreta a respeito dos resultados obtidos. Um deles é que as implementações visaram a maior semelhança e igualdade de condições para os algoritmos, descartando otimizações mais profundas que poderiam interferir nos resultados. Por exemplo para o caso da FFT não foi utilizada uma otimização que considerasse que as entradas seriam apenas reais, como no caso da DFT, pelo simples fato de que no caso da FFT esta otimização não seria trivial e poderia gerar um algoritmo com estratégia muito diferente da FHT, o que não seria desejado neste caso.

Outro aspecto é o da linguagem e ambiente de programação. Apesar de termos utilizado a linguagem C++, o ambiente utilizado para a programação priorizou a facilidade para os testes, porém não a otimização a baixo nível. Com isso, havia pouco controle sobre as instruções geradas pelo compilador ao converter o código para linguagem de máquina, sendo que o cuidado que houve para que os algoritmos fossem semelhantes pode não ter sido completamente respeitado a baixo nível. Para um teste mais preciso seria recomendável utilizar uma linguagem de mais baixo nível utilizando-se um ambiente de programação sem interface gráfica em um sistema operacional onde houvesse o controle para que apenas o processo referente ao cálculo das transformadas fosse executado no momento dos testes.

O importante a se ressaltar é que os resultados obtidos no presente trabalho não sejam tomados como verdades absolutas, muito menos sejam ignorados por completo, e sim sirvam como motivação para maiores pesquisas

nesta área, já que ainda há poucos estudos sobre a transformada de Hartley e como ela pode ser calculada e aplicada com eficiência.

Para trabalhos futuros podemos sugerir a análise mais detalhada dos aspectos citados acima, para gerar algoritmos com maiores otimizações e testes com maior precisão. A dica seria a de pesquisar mais profundamente as otimizações que podem ser feitas e de implementar os algoritmos utilizando uma linguagem como C ou Fortran, em conjunto com um compilador como o GCC e com sistema operacional Linux ou semelhante.

Porém, independente de trabalhos futuros que possam vir a comprovar ou desmentir os dados aqui obtidos, fica a certeza de que ambas as transformadas são muito úteis, e cada uma delas pode ter uma aplicação onde seu uso seja mais recomendável do que a outra. Por exemplo, imagine uma situação onde deseje-se economizar o máximo possível o consumo de memória, porém o processamento é muito rápido. Neste caso poderia-se calcular a transformada de Hartley, armazenar seus coeficientes com metade do consumo de memória do que a transformada de Fourier, e calcular esta última através de cálculos triviais com os dados armazenados. Por outro lado, em uma aplicação que utilize números complexos, a FHT fica completamente descartada, sendo necessário a utilização da FFT.

Assim, chegamos ao fim deste trabalho com a conclusão de que as transformadas de Hartley e de Fourier não competem entre si para que uma reine soberana e a outra caia no esquecimento, mas sim se completam, abrindo um leque de ferramentas disponíveis para a resolução de problemas com particularidades diferentes.

REFERÊNCIAS BIBLIOGRÁFICAS

- [1] O'NEILL, Mark A. Faster Than Fast Fourier. **Byte**. v. 13 n.4, p. 293-300, Abril, 1988.
- [2] SZEREMETA, Júlio F. **Apostila de Análise Numérica II**. Florianópolis, Setembro de 2002. 86 p.
- [3] GERALD, Curtis; WHEATLEY, Patrick. **Applied Numerical Analysis**. Addison-Wesley, 1999.
- [4] BRIGGS, William L; HENSON, Van E. **The DFT: An Owner's Manual for the Discrete Fourier Transform**. Society for Industrial and Applied Mathematics, Philadelphia. 1995.
- [5] FRIGO, Matteo; JOHNSON, Steven G. **The Design and Implementation of FFTW3**. Proceedings of the IEEE, v. 93, n. 2, p. 216-231, 2005.
- [6] **Wikipedia, the free encyclopedia**. Disponível em <<http://www.wikipedia.org>> , acesso em 31 de agosto de 2005.
- [7] SCOTT, Robert. **Doing Hartley Smartly**. Disponível em <<http://www.embedded.com/2000/0009/0009feat3.htm>>, acesso em 30 de setembro de 2005.
- [8] ERICKSON, Adam C.; FAGIN, Barry S. **Calculating The FHT in Hardware**. Disponível em <<http://www.faginfamily.net/barry/Papers/ieeetsp.htm>>, acesso em 15 de outubro de 2005.
- [9] LE-NGOC, Tho; VO, Minh Tue. **Implementation and Performance of the Fast Hartley Transform**. IEEE Micro, v. 9, n.5, p. 20-27, 1989.

ANEXO – CÓDIGO FONTE

Arquivo Transformadas.h

```
#ifndef TransformadasH
#define TransformadasH

#include <mmsystem.h>

double f(double x);

class Transformadas
{
public:
    double *DFT(double* dados, double *imag, int N);
    double *iDFT(double* A, double *B, int N, int grau); //inverse DFT
    double *DHT(double* dados, int N);
    double *iDHT(double* dados, int N); //inverse DHT
    double *FourierPorDHT(double* dados, double *img, int N);
    double *DFTopt(double* dados, double *imag, int N);

    double *FFT(double* dados, double *imag, int N);
    double *FHT(double* dados, int N);
    double *FFTviaFHT(double* dados, double *img, int N);

    void CoefFourierNorm(double* X, double* Y, double* A, double* B, int
N, int grau);
    double *AproxFourierFNorm(int numPontos, int grau, double* fX, double
*A, double *B);

};

#endif
```

Arquivo Transformadas.cpp

```
#include "Unit1.h"
#include <math.h>
#include "Transformadas.h"

double *Transformadas::DFT(double* dados, double *imag, int N)
{
    double *real = new double[ N ];
    double f2pi_N;
    double pi2_N = (M_PI + M_PI) / N;

    int tempo = GetTickCount();

    for ( int f = 0; f < N; f++ )
    {
        real[ f ] = 0;
        imag[ f ] = 0;
        f2pi_N = f * pi2_N;

        for ( int t = 0; t < N; t++ )
        {
            real[ f ] += dados[ t ] * cos( f2pi_N * t );
            imag[ f ] -= dados[ t ] * sin( f2pi_N * t );
        }
        real[ f ] = real[ f ] / N;
        imag[ f ] = imag[ f ] / N;
        // Form1->Atualiza(100.0 * (f + 1) / N);
    }

    tempo = GetTickCount() - tempo;
    Form1->Label4->Caption = tempo;

    return real;
}

double *Transformadas::DFTopt(double* dados, double *imag, int N)
{
```



```

double *real = new double[ N ];
double f2pi_N;
double pi2_N = (M_PI + M_PI) / N;
// 1 adição + 1 divisão

int tempo = GetTickCount();

real[ 0 ] = 0;
for ( int t = 0; t < N; t++ ) // N-1 adições
    real[ 0 ] += dados[ t ];

// N adições

real[ 0 ] = real[ 0 ] / N;
imag[ 0 ] = 0;

for ( int f = 1; f <= N * 0.5; f++ ) // 1 multiplicação + (N/2)-1
adições
{
    real[ f ] = 0;
    imag[ f ] = 0;
    f2pi_N = f * pi2_N;
    //(N/2) multiplicações

    for ( int t = 0; t < N; t++ ) //(N²-N)/2 adições
    {
        real[ f ] += dados[ t ] * cos( f2pi_N * t );
        // N²/2 (2 multiplicações + 1 adição + 1 cos)
        imag[ f ] -= dados[ t ] * sin( f2pi_N * t );
        // N²/2 (2 multiplicações + 1 adição + 1 sen)
    }

    real[ f ] = real[ f ] / N;
    imag[ f ] = imag[ f ] / N;
    //N divisões
    real[ N - f ] = real[ f ];
    imag[ N - f ] = - imag[ f ];
//    Form1->Atualiza(200.0 * (f + 1) / N);
}

```

```

tempo = GetTickCount() - tempo;
Form1->Label4->Caption = tempo;

return real;
}

void Transformadas::CoefFourierNorm(double* X, double* Y, double* A,
double* B, int N, int grau)
{

    int M = N / 2;

    A[ 0 ] = 0;
    B[ 0 ] = 0;
    A[ grau ] = 0;
    for ( int j = 0; j < N; j++ ) // calculo do a0
    {
        A[ 0 ] += Y[ j ];
        A[ grau ] += Y[ j ] * cos( grau * X[ j ] );
    }
    A[ 0 ] = A[ 0 ] / M;
    A[ grau ] = A[ grau ] / M;

    for ( int k = 1; k < grau; k++ )
    {
        A[ k ] = 0;
        B[ k ] = 0;
        for ( int j = 0; j < N; j++ )
        {
            A[ k ] += Y[ j ] * cos( k * X[ j ] );
            B[ k ] += Y[ j ] * sin( k * X[ j ] );
        }
        A[ k ] = A[ k ] / M;
        B[ k ] = B[ k ] / M;
    }
    // Form1->Atualiza(100 * (k + 1) / grau);
}

```

```

}

double *Transformadas::iDFT(double* A, double *B, int N, int grau) // não
testado
{
    double *inv = new double[ N ];

    double f2pi_N;
    double pi2_N = (M_PI + M_PI) / N;

    for (int j = 0; j < N; j++)
    {
        f2pi_N = j * pi2_N;

        inv[ j ] = (A[ 0 ] / 2) + (A [ grau ] * cos( grau * f2pi_N ) ) ;

        for (int k = 1; k < grau; k++)
        {
            inv[ j ] += ( A[ k ] * cos( k * f2pi_N ) )
                + ( B[ k ] * sin( k * f2pi_N ) );
        }
    }

    return inv;
}

double *Transformadas::iDHT(double* dados, int N)
{
    double *resultado = new double[ N ];
    double f2pi_N;
    double pi2_N = (M_PI + M_PI) / N;

    for ( int f = 0; f < N; f++ )
    {
        resultado[ f ] = 0;
        f2pi_N = f * pi2_N;
    }
}

```

```

        for ( int t = 0; t < N; t++ )
        {
            resultado[ f ] += M_SQRT2 * dados[ t ] * cos( (f2pi_N * t) -
M_PI_4 );
        }
//        Form1->Atualiza(100 * (f + 1) / N);
    }

    return resultado;
}

double *Transformadas::DHT(double* dados, int N)
{
    double *resultado = new double[ N ];
    double f2pi_N;
    double pi2_N = (M_PI + M_PI) / N;

    int tempo = GetTickCount();

    for ( int f = 0; f < N; f++ )
    {
        resultado[ f ] = 0;
        f2pi_N = f * pi2_N;
        for ( int t = 0; t < N; t++ )
        {
            resultado[ f ] += M_SQRT2 * dados[ t ] * cos( (f2pi_N * t) -
M_PI_4 );
        }
        resultado[ f ] = resultado[ f ] / N;
//        Form1->Atualiza(100 * f / N);
    }

    tempo = GetTickCount() - tempo;
    Form1->Label4->Caption = tempo;

    return resultado;
}

```

```

}

double *Transformadas::FourierPorDHT(double* dados, double *img, int N)
{
    double *resultado = new double[ N ];
    double *real = new double[ N ];
    double f2pi_N;
    double pi2_N = (M_PI + M_PI) / N; //1 adição + 1 divisão

    int tempo = GetTickCount();

    for ( int f = 0; f < N; f++ ) // N-1 adições (incrementos)
    {
        resultado[ f ] = 0;
        f2pi_N = f * pi2_N; // N multiplicações

        for ( int t = 0; t < N; t++ ) // N²-N adições
        {
            resultado[ f ] += M_SQRT2 * dados[ t ] * cos( (f2pi_N * t) -
M_PI_4 );
            // N² (2 adições + 3 multiplicações + cos)
        }
        resultado[ f ] = resultado[ f ] / N;
//        Form1->Atualiza(100 * f / N);
    }

    real[ 0 ] = resultado[ 0 ];
    img[ 0 ] = 0;

    for (int j = 1; j <= N * 0.5; j++)
    {
        real[ j ] = (resultado[ N - j ] + resultado[ j ]) * 0.5;
        real[ N - j ] = real[ j ];

        img[ j ] = (resultado[ N - j ] - resultado[ j ]) * 0.5;
        img[ N - j ] = - img[ j ];
    }
}

```

```

    tempo = GetTickCount() - tempo;
    Form1->Label2->Caption = tempo;

    delete resultado;
    return real;
}

double f(double x)
{
    return ( x * x * ( 2 + x * ( -3 + x ) ) ) - tan( x * ( x - 2 ) );
}

double z(double x)
{
    return M_PI * ( x - 1 );
}

double x(double z)
{
    return 1 + z / M_PI;
}

double *Transformadas::AproxFourierFNorm(int numPontos, int grau, double*
fX, double *A, double *B)
{
    double* X = new double [ numPontos ];
    double* Z = new double [ numPontos ];
    double* sX = new double [ numPontos ];
    double* Dif = new double [ numPontos ];

    double passo = 2.0 / numPontos;

    for (int j = 0; j < numPontos; j++)
    {
        X[ j ] = j * passo;
        fX[ j ] = f(X[ j ]);
        Z[ j ] = ( M_PI * X[ j ] );
    }
}

```

```

}

Transformadas FT;
FT.CoeffFourierNorm(Z, fX, A, B, numPontos, grau);

for (int j = 0; j < numPontos; j++)
{
    sX[ j ] = (A[ 0 ] / 2) + (A [ grau ] * cos( grau * Z[ j ] ) ) ;
    for (int k = 1; k < grau; k++)
    {
        sX[ j ] += ( A[ k ] * cos( k * Z[ j ] ) )
                + ( B[ k ] * sin( k * Z[ j ] ) );
    }

    Dif[ j ] = fX[ j ] - sX[ j ];
}

delete X;
delete Z;
delete sX;

return Dif;
}

```

```

double *Transformadas::FFT(double* dados, double *imag, int N)
{
    double *real = new double[ N ];
    double *tmpreal = new double[ N ];
    double *tmpimg = new double[ N ];
    int *pot = new int[ N ];
    int indicebase;
    int indice;

    double a,b,c,d; //para multiplicar complexos: (a+bi)*(c+di)
    double pi2_N = (M_PI + M_PI) / N;
    int N2 = N/2;

```

```

int passo = N2;

timeBeginPeriod(1);
int tempo = GetTickCount();

for (int j = 0; j < N; j++)
{
    pot[ j ] = 0;
    tmpreal[ j ] = dados[ j ];
    tmpimg[ j ] = 0;
}

//    for (int k = 1; k <= log2N; k++ )
for (int k = 1; k < N; k += k )
{

    for (int j = 0; j < N2; j++)
        pot[ j ] = pot[ j + j ];

    for (int j = 0; j < N2; j++)
        pot[ N2 + j ] = pot[ j ] + passo;

    indicebase = 0;

    for (int m = 0; m < k; m++)
    {
        for (int j = 0; j < passo; j++)
        {
            indice = indicebase + j;

            //xk+1[j] = xk[j] + W^pot[j] * xk[j+passo]
            a = tmpreal[ indice + passo ];
            b = tmpimg[ indice + passo ];
            c = cos(pi2_N * pot[ indice ]);
            d = -sin(pi2_N * pot[ indice ]);

            real[ indice ] = tmpreal[ indice ] + (a*c - b*d);
            imag[ indice ] = tmpimg[ indice ] + (b*c + a*d);
        }
    }
}

```



```

        c = cos(pi2_N * pot[ indice + passo ]);
        d = -sin(pi2_N * pot[ indice + passo ]);

        //xk+1[j+passo] = xk[j] + W^pot[j+passo] * xk[j+passo]
        real[ indice + passo ] = tmpreal[ indice ] + (a*c - b*d);
        imag[ indice + passo ] = tmpimg[ indice ] + (b*c + a*d);
    }
    indicebase += passo + passo;
}

for(int j = 0; j < N; j++)
{
    tmpreal[ j ] = real[ j ];
    tmpimg[ j ] = imag[ j ];
}
passo = passo * 0.5;
}

for(int j = 0; j < N; j++)
{
    real[ j ] = tmpreal[ pot[ j ] ] / N;
    imag[ j ] = tmpimg[ pot[ j ] ] / N;
}

tempo = GetTickCount() - tempo;
Form1->Label4->Caption = tempo;
timeEndPeriod(1);

delete tmpreal;
delete tmpimg;
delete pot;
return real;
}

double *Transformadas::FHT(double* dados, int N)
{
    double *result = new double[ N ];

```

```

double *tmpresult = new double[ N ];
int *permuta = new int[ N ];

int passo = N/2;

int indicebase;
int indice;

double pi2_N = M_PI;

timeBeginPeriod(1);
int tempo = GetTickCount();

permuta[0] = 0;
permuta[1] = 1;
for (int k = 2; k < N; k += k )
{
    for(int j = 0; j < k; j++)
    {
        permuta[j] += permuta[j];
        permuta[ j + k ] = permuta[j] + 1;
    }
}

for (int j = 0; j < N; j += 2) //passo1
{
    tmpresult[ j ] = (dados[ permuta[ j ] ] + dados[ permuta[ j + 1 ]
]);
    tmpresult[ j + 1 ] = (dados[ permuta[ j ] ] - dados[ permuta[ j +
1 ] ]));
}

for (int k = 2; k < N; k += k )
{
    passo = passo * 0.5;
    pi2_N = pi2_N * 0.5;
    indicebase = 0;
}

```

```

for (int m = 0; m < passo; m++)
{

    result[ indicebase ] = tmpresult[ indicebase ]
        + tmpresult[ indicebase + k ];

    result[ indicebase + k ] = tmpresult[ indicebase ]
        - tmpresult[ indicebase + k ];

    for (int j = 1; j < k; j++)
    {

        //xk+1[j] = xk[j] + W^pot[j] * xk[j+passo]
        indice = indicebase + j;
        result[ indice ] = tmpresult[ indice ]
            + ( tmpresult[ indice + k ] *
                cos( pi2_N * j ) )
            + ( tmpresult[ indicebase + k + k - j ] *
                sin( pi2_N * j ) );

        //xk+1[j+passo] = xk[j] + W^pot[j+passo] * xk[j+passo]
        result[ indice + k ] = tmpresult[ indice ]
            - ( tmpresult[ indice + k ] *
                cos( pi2_N * j ) )
            - ( tmpresult[ indicebase + k + k - j ] *
                sin( pi2_N * j ) );

    }

    indicebase += k + k;
}

for(int j = 0; j < N; j++)
    tmpresult[ j ] = result[ j ];

}

```

```

for(int j = 0; j < N; j++)
    result[ j ] = tmpresult[ j ] / N;

tempo = GetTickCount() - tempo;
Form1->Label6->Caption = tempo;
timeEndPeriod(1);

delete tmpresult;
delete permuta;

return result;
}

double *Transformadas::FFTviaFHT(double* dados, double *img, int N)
{
    double *result = new double[ N ];
    double *tmpresult = new double[ N ];
    int *permuta = new int[ N ];

    int passo = N/2;

    int indicebase;
    int indice;

    double pi2_N = M_PI;

    timeBeginPeriod(1);
    int tempo = GetTickCount();

    permuta[0] = 0;
    permuta[1] = 1;
    for (int k = 2; k < N; k += k )
    {
        for(int j = 0; j < k; j++)
        {
            permuta[j] += permuta[j];
            permuta[ j + k ] = permuta[j] + 1;
        }
    }
}

```

```

    }
}

for (int j = 0; j < N; j += 2) //passo1
{
    tmpresult[ j ] = (dados[ permuta[ j ] ] + dados[ permuta[ j + 1 ]
]) ;
    tmpresult[ j + 1 ] = (dados[ permuta[ j ] ] - dados[ permuta[ j +
1 ] ] );
}

for (int k = 2; k < N; k += k )
{
    passo = passo / 2;
    pi2_N = pi2_N / 2;
    indicebase = 0;

    for (int m = 0; m < passo; m++)
    {
        result[ indicebase ] = tmpresult[ indicebase ]
            + tmpresult[ indicebase + k ];

        result[ indicebase + k ] = tmpresult[ indicebase ]
            - tmpresult[ indicebase + k ];

        for (int j = 1; j < k; j++)
        {
            //xk+1[j] = xk[j] + W^pot[j] * xk[j+passo]
            indice = indicebase + j;
            result[ indice ] = tmpresult[ indice ]
                + ( tmpresult[ indice + k ] *
                    cos( pi2_N * j ) )
                + ( tmpresult[ indicebase + k + k - j ] *
                    sin( pi2_N * j ) );

            //xk+1[j+passo] = xk[j] + W^pot[j+passo] * xk[j+passo]
            result[ indice + k ] = tmpresult[ indice ]

```

```

        - ( tmpresult[ indice + k ] *
            cos( pi2_N * j ) )
        - ( tmpresult[ indicebase + k + k - j ] *
            sin( pi2_N * j ) );
    }

    indicebase += k + k;
}

for(int j = 0; j < N; j++)
    tmpresult[ j ] = result[ j ];
}

result[ 0 ] = tmpresult[ 0 ] / N;
img[ 0 ] = 0;

for (int j = 1; j <= N * 0.5; j++)
{
    result[ j ] = (tmpresult[ N - j ] + tmpresult[ j ]) / (N+N);
    result[ N - j ] = result[ j ];

    img[ j ] = (tmpresult[ N - j ] - tmpresult[ j ]) / (N+N);
    img[ N - j ] = - img[ j ];
}

tempo = GetTickCount() - tempo;
Form1->Label2->Caption = tempo;
timeEndPeriod(1);

delete tmpresult;
delete permuta;
return result;
}

#pragma package(smart_init)

```

Arquivo Unit1.h (interface gráfica)

```
//-----  
#ifndef Unit1H  
#define Unit1H  
//-----  
#include <Classes.hpp>  
#include <Controls.hpp>  
#include <StdCtrls.hpp>  
#include <Forms.hpp>  
#include <ExtCtrls.hpp>  
#include <ComCtrls.hpp>  
  
#include "CGAUGES.h"  
//-----  
class TForm1 : public TForm  
{  
__published:      // IDE-managed Components  
    TMemo *Memo1;  
    TButton *Button1;  
    TMemo *Memo2;  
    TMemo *Memo3;  
    TButton *Button2;  
    TButton *Button3;  
    TButton *Button4;  
    TCGauge *CGauge1;  
    TLabel *Label1;  
    TLabel *Label2;  
    TLabel *Label3;  
    TLabel *Label4;  
    TLabel *Label5;  
    TLabel *Label6;  
    TButton *Button5;  
    TButton *Button6;  
    TEdit *Edit1;  
    TButton *Button7;  
    TButton *Button8;  
    TButton *Button9;
```

```

TEdit *Edit2;
TLabel *Label7;
TLabel *Label8;
    void __fastcall Button1Click(TObject *Sender);
void __fastcall Button2Click(TObject *Sender);
void __fastcall Button3Click(TObject *Sender);
void __fastcall Button4Click(TObject *Sender);
void __fastcall Memo1Change(TObject *Sender);
void __fastcall Button5Click(TObject *Sender);
void __fastcall Button6Click(TObject *Sender);
void __fastcall Button7Click(TObject *Sender);
void __fastcall Button8Click(TObject *Sender);
void __fastcall Button9Click(TObject *Sender);
private:    // User declarations
public:        // User declarations
    void Atualiza(double X);

    __fastcall TForm1(TComponent* Owner);
};
//-----
extern PACKAGE TForm1 *Form1;
//-----
#endif

```

Arquivo Unit1.cpp (interface gráfica)

```

#include <vcl.h>
#pragma hdrstop

#include "Hartley.h"
#include "Unit1.h"
#include "Transformadas.h"
#include <math.h>

#pragma package(smart_init)
#pragma link "CGAUGES"

```



```

#pragma resource "*.dfm"
TForm1 *Form1;

void __fastcall TForm1::Button1Click(TObject *Sender)
{
    int N = Mem01->Lines->Count;
    double *p = new double[ N ];
    for( int i = 0; i < N; i++ )
        p[ i ] = StrToFloatDef( Mem01->Lines->Strings[ i ], 0 );

    HANDLE hProcess = GetCurrentProcess();
    HANDLE hThread = GetCurrentThread();

    SetPriorityClass(hProcess, REALTIME_PRIORITY_CLASS);
    SetThreadPriority(hThread, THREAD_PRIORITY_TIME_CRITICAL);

    Transformadas T;
    double *h = T.FHT(p, N);

    for( int j = 0; j < N; j++ )
        Memo2->Lines->Add( FloatToStr( h[ j ] ) );

    double *imag = new double [ N ];
    double *h1 = T.DHT(p, N);

    for( int k = 0; k < N; k++ )
        Memo3->Lines->Add( FloatToStr( h1[ k ] ) );

    delete p;
    delete h;
    delete h1;
    delete imag;

    SetThreadPriority(hThread, THREAD_PRIORITY_NORMAL);
    SetPriorityClass(hProcess, NORMAL_PRIORITY_CLASS);
}

```

```

void __fastcall TForm1::Button2Click(TObject *Sender)
{
    int N = Mem01->Lines->Count;
    double *p = new double[ N ];
    for( int i = 0; i < N; i++ )
        p[ i ] = StrToFloatDef( Mem01->Lines->Strings[ i ], 0 );

    Transformadas T;
    double *h = T.DHT(p, N);
    for( int j = 0; j < N; j++ )
        Memo2->Lines->Add( FloatToStr( h[ j ] ) );

    double *h1 = T.iDHT(h, N);

    for( int k = 0; k < N; k++ )
        Memo3->Lines->Add( FloatToStr( h1[ k ] ) );

    delete p;
    delete h;
    delete h1;
}

```

```

void __fastcall TForm1::Button3Click(TObject *Sender)
{
    HANDLE hProcess = GetCurrentProcess();
    HANDLE hThread = GetCurrentThread();
    SetPriorityClass(hProcess, REALTIME_PRIORITY_CLASS);
    SetThreadPriority(hThread, THREAD_PRIORITY_TIME_CRITICAL);

    int N = Mem01->Lines->Count;
    double *p = new double[ N ];
    for( int i = 0; i < N; i++ )
        p[ i ] = StrToFloatDef( Mem01->Lines->Strings[ i ], 0 );

    Transformadas T;
    double *h = T.DHT(p, N);

```

```

for( int j = 0; j < N; j++ )
    Memo2->Lines->Add( FloatToStr( h[ j ] ) );

Memo2->Lines->Add( FloatToStr( h[ 0 ] ) );
for( int j = 1; j < N; j++ )
    Memo2->Lines->Add( FloatToStr( (h[ j ] + h[ N - j ]) / 2 )
);

Memo2->Lines->Add( 0 );
for( int j = 1; j < N; j++ )
    Memo2->Lines->Add( FloatToStr( (h[ N - j ] - h[ j ] ) / 2 )
);

double *imag = new double [ N ];
double *h1 = T.DFT(p, imag, N);

for( int k = 0; k < N; k++ )
    Memo3->Lines->Add( FloatToStr( h1[ k ] ) );
for( int k = 0; k < N; k++ )
    Memo3->Lines->Add( FloatToStr( imag[ k ] ) );

delete p;
delete h;
delete h1;
delete imag;
SetThreadPriority(hThread, THREAD_PRIORITY_NORMAL);
SetPriorityClass(hProcess, NORMAL_PRIORITY_CLASS);

}

void __fastcall TForm1::Button4Click(TObject *Sender)
{

    int N = 10;
    int grau = 5;

    double* V = new double[ N ];
    double* A = new double[ grau + 1 ];

```

```

double* B = new double[ grau ];

Transformadas T;

double *dif = T.AproxFourierFNorm(N, grau, V, A, B);

for( int k = 0; k < N; k++ )
    Memo3->Lines->Add( FloatToStr( dif[ k ] ) );

for( int k = 0; k < N; k++ )
    Memo1->Lines->Add( FloatToStr( V[ k ] ) );

for( int k = 0; k <= grau; k++ )
    Memo2->Lines->Add( FloatToStr( A[ k ] ) );
Memo2->Lines->Add( '-' );
for( int k = 1; k < grau; k++ )
    Memo2->Lines->Add( FloatToStr( B[ k ] ) );

delete V;
delete A;
delete B;
delete dif;
}

void TForm1::Atualiza(double X)
{
    CGaugel->Progress = X;
    CGaugel->Refresh();
}

void __fastcall TForm1::Memo1Change(TObject *Sender)
{
    Label6->Caption = Memo1->Lines->Count;
}

void __fastcall TForm1::Button5Click(TObject *Sender)
{

```

```

HANDLE hProcess = GetCurrentProcess();
HANDLE hThread = GetCurrentThread();

SetPriorityClass(hProcess, REALTIME_PRIORITY_CLASS);
SetThreadPriority(hThread, THREAD_PRIORITY_TIME_CRITICAL);

int N = Memo1->Lines->Count;
double *p = new double[ N ];
for( int i = 0; i < N; i++ )
    p[ i ] = StrToFloatDef( Memo1->Lines->Strings[ i ], 0 );

Transformadas T;
double *img = new double [ N ];
double *h = T.FourierPorDHT(p, img, N);

for( int j = 0; j < N; j++ )
    Memo2->Lines->Add( FloatToStr( h[ j ] ) );

for( int j = 0; j < N; j++ )
    Memo2->Lines->Add( FloatToStr( img[ j ] ) );

double *imag = new double [ N ];
double *h1 = T.DFTopt(p, imag, N);

for( int k = 0; k < N; k++ )
    Memo3->Lines->Add( FloatToStr( h1[ k ] ) );
for( int k = 0; k < N; k++ )
    Memo3->Lines->Add( FloatToStr( imag[ k ] ) );

delete p;
delete h;
delete h1;
delete imag;
delete img;

SetThreadPriority(hThread, THREAD_PRIORITY_NORMAL);
SetPriorityClass(hProcess, NORMAL_PRIORITY_CLASS);

```

```

}

void __fastcall TForm1::Button6Click(TObject *Sender)
{
    int N = StrToIntDef( Edit1->Text, 0 );
    double *p = new double[ N ];
    double passo = 2.0 / N;
    for( int i = 0; i < N; i++ )
        p[ i ] = f( passo * i );

    for( int j = 0; j < N; j++ )
        Mem1->Lines->Add( FloatToStr( p[ j ] ) );

    delete p;
}

void __fastcall TForm1::Button7Click(TObject *Sender)
{
    Mem1->Lines->Clear();
    Mem2->Lines->Clear();
    Mem3->Lines->Clear();
}

void __fastcall TForm1::Button8Click(TObject *Sender)
{
    int N = Mem1->Lines->Count;
    double *p = new double[ N ];
    for( int i = 0; i < N; i++ )
        p[ i ] = StrToFloatDef( Mem1->Lines->Strings[ i ], 0 );

    HANDLE hProcess = GetCurrentProcess();
    HANDLE hThread = GetCurrentThread();

    SetPriorityClass(hProcess, REALTIME_PRIORITY_CLASS);
    SetThreadPriority(hThread, THREAD_PRIORITY_TIME_CRITICAL);

    Transformadas T;
}

```

```

double *img = new double [ N ];
double *h = T.FFTviaFHT(p, img, N);

for( int j = 0; j < N; j++ )
    Memo2->Lines->Add( FloatToStr( h[ j ] ) );

for( int j = 0; j < N; j++ )
    Memo2->Lines->Add( FloatToStr( img[ j ] ) );

double *imag = new double [ N ];
double *h1 = T.DFTopt(p, imag, N);

for( int k = 0; k < N; k++ )
    Memo3->Lines->Add( FloatToStr( h1[ k ] ) );
for( int k = 0; k < N; k++ )
    Memo3->Lines->Add( FloatToStr( imag[ k ] ) );

delete p;
delete h;
delete h1;
delete imag;
delete img;

SetThreadPriority(hThread, THREAD_PRIORITY_NORMAL);
SetPriorityClass(hProcess, NORMAL_PRIORITY_CLASS);

}

void __fastcall TForm1::Button9Click(TObject *Sender)
{

    int log2N = StrToIntDef( Edit2->Text, 0 );
    int N = pow(2, log2N);
    double *p = new double[ N ];
    double passo = 2.0 / N;
    for( int i = 0; i < N; i++ )
        p[ i ] = f( passo * i );
}

```

```

HANDLE hProcess = GetCurrentProcess();
HANDLE hThread = GetCurrentThread();

SetPriorityClass(hProcess, REALTIME_PRIORITY_CLASS);
SetThreadPriority(hThread, THREAD_PRIORITY_TIME_CRITICAL);

    Transformadas T;

    double *imag = new double [ N ];
    double *h1 = T.FFT(p, imag, N);

    double *img = new double [ N ];
    double *h2 = T.FHT(p, N);
    double *h = T.FFTviaFHT(p, img, N);

    delete p;
    delete h;
    delete h1;
    delete h2;
    delete imag;
    delete img;

SetThreadPriority(hThread, THREAD_PRIORITY_NORMAL);
SetPriorityClass(hProcess, NORMAL_PRIORITY_CLASS);
}

```