Estudo de caso: Implementação da *Fast Fourier Transform* em C

Kaique Guimarães Cerqueira*
Engenharia de Controle e Automação, Instituto Federal Fluminense.
(Dated: 24 de agosto de 2022)

O seguinte trabalho tem como objetivo investigar a implementação da *Fast Fourier Transform* em C, seguindo o algoritmo de Cooley-Tukey. Utilizado como forma de avaliação da disciplina de Processamento de Sinais. A escrita obedece a seguinte ordem: 1)Introdução ao tema, 2) Algoritmo de Cooley-Tukey 3) Implementação utilizada, 4) Análise numérica dos resultados, 5) Análise temporal e 6) Conclusões.

I. INTRODUÇÃO

A DFT, ou *Discrete Fourier Transform*, é o equivalente no domínio da frequência discreta a Transformada de Fourier. Sua fórmula pode ser descrita como:

$$X_k = \sum_{n=0}^{N-1} x_n \cdot e^{\frac{i2\pi}{N}kn} \tag{1}$$

Onde:

x_n são as amostras do sinal discreto

X_k são os pontos complexos resultantes da transformada

N é o número de amostras observadas

k é o índice das amostras no domínio da frequência discreta

Podemos observar que, para cada valor de k, executa-se N multiplicações complexas, onde k também varia entre 0 e $N-1$. Logo, para computar a transformada discreta de Fourier, seriam necessários $N \cdot N$ operações matemáticas ($O(N^2)$), o que se torna algo com o custo impraticável computacionalmente.

Técnicas para aumentar a eficiência e diminuir o tempo necessário para o cálculo foram desenvolvidas. Sua primeira citação na era moderna é chamada de algoritmo de *Cooley-Tukey* [1], apesar de Gauss ter usado a técnica 150 anos antes, a fim de calcular de forma recursiva a interpolação da trajetória dos asteroides *Pallas* e *Juno* [2].

II. ALGORITMO DE COOLEY-TUKEY

Esta técnica explora as propriedades de simetria e periodicidade dos chamados *twiddle factors*, que podem ser definidos como:

$$W_n = e^{-i2\pi/N}$$

Em particular, as propriedades podem ser explicitadas:

Simetria: $W_N^{k+N/2} = -W_N^k$
Periodicidade: $W_N^{k+N} = W_N^k$

Em suma, estas propriedades nos permitem dividir o problema da DFT em dois, separados as sequências de amostras de acordo com os índices pares e ímpares, de tamanho $\frac{N}{2}$ cada. A execução desses passos recursivamente otimiza os cálculos e reduz o algoritmo para a complexidade $N \log_2 N$. Esta decomposição gera o que chama-se de *butterfly multiplications*, que é basicamente o processo núcleo onde se aplica as propriedades citadas para a reutilização dos cálculos. Adotando E_k como a denotação para a DFT dos índices pares e O_k sendo a DFT dos índices ímpares, sua operação básica pode ser descrita como:

$$X_k = E_k + W_N^k \cdot O_k$$

$$X_{k+\frac{N}{2}} = E_k - W_N^k \cdot O_k$$

As *butterflies* são feitas em estágios e o algoritmo otimiza a sua computação ao reutilizar multiplicações intermediárias para calcular o resultado de múltiplas sub-DFTs.

Em um sistema com 8 amostras, os estágios das *butterflies* podem ser descritos como[3]:

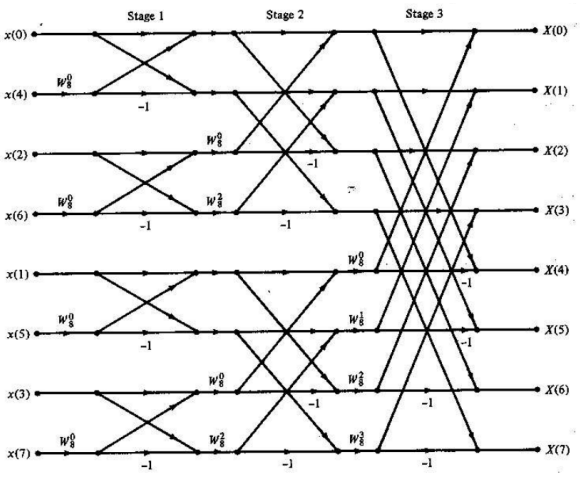


Figura 1: Estágios das *butterflies* calculados para 8 amostras

*Electronic address: kaique.cerqueira@gsuite.iff.edu.br

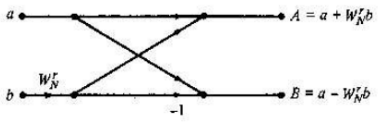


Figura 2: Computação básica da butterfly em FFTs de decimação no tempo

Após o processo de múltiplos butterflies, nota-se que as amostras passam por um processo de embaralhamento, resultado de sucessivas separações entre amostras de índices pares e ímpares. Ao observar a notação binária das amostras, percebe-se que a ordem dos *bits* estão invertidos. Por isso, é de costume utilizar um algoritmo de *bit-reversal* em uma etapa anterior às butterflies, com o intuito de que ao final das operações se obtenha o vetor de transformações ordenado corretamente.

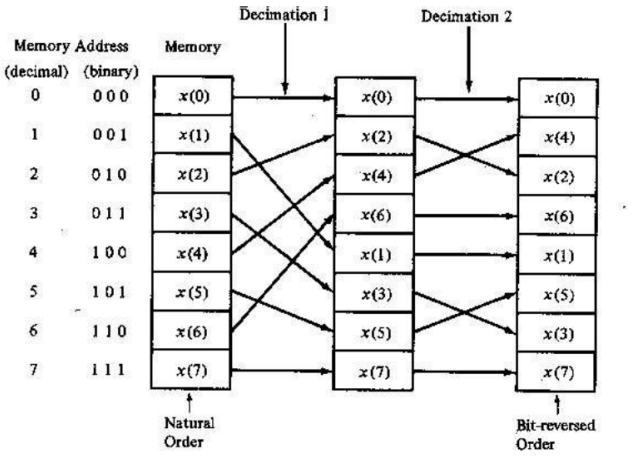


Figura 3: Representação do embaralhamento das posições após a multiplicação das *butterflies*

O resultado da implementação destes dois algoritmos é a FFT de base 2 com decimação no tempo, também chamado de algoritmo de Cooley-Tukey.

III. IMPLEMENTAÇÃO

Neste trabalho, o algoritmo da FFT foi implementado na linguagem C de forma iterativa [4], evitando assim o processo explícito de recursão. O pseudocódigo utilizado como base pode ser descrito na figura 4. Os dados foram gerados através do *software* MATLAB, que também foi utilizado para a análise posterior e serviu como ferramenta de comparação. O código fonte do projeto pode ser encontrado por completo em [5].

A. Sinal de *Benchmark*

Dentro do ambiente de desenvolvimento do MATLAB, um sinal de *benchmark* foi gerado para os testes. Esse

```
{Loop over the size of the IDFT}
for(int L=2; L<=N; L*=2)
   {Loop over the IDFT of one level}
   for(int k=0; k<N; k+=L)
      {perform all butterflies of one level}
      for(int j=0; j<L/2; j++) {
         {complex computation:}
         z ← ω_L^j * X[k+j+L/2]
         X[k+j+L/2] ← X[k+j] - z
         X[k+j] ← X[k+j] + z
      }
```

Figura 4: Implementação iterativa do cálculo das *butterflies*

sinal é uma composição de senos e cossenos de diferentes amplitudes, fases e frequências, a fim de poder diferenciar com facilidade a resposta em frequência dada como resultado da FFT. Sua função pode ser descrita como:

$$U(t) = \sin\left(2\pi \cdot 10t + \frac{\pi}{2}\right) + 1.5 \cdot \cos(2\pi \cdot 3t) + 0.75 \cdot \sin(2\pi \cdot 14t)$$

Dentro dele, podemos observar componentes em frequência de $3Hz$, $10Hz$ e $14Hz$, e para não ferir o teorema de Nyquist, foi escolhido a frequência de amostragem de 50Hz, o que ultrapassa com folga o dobro de seu maior componente em frequência.

A fim de aumentar a complexidade do sinal, implementa-se um ruído aleatório de distribuição normal, com amplitude que pode ser de até 2x maior que o maior componente em frequência. O resultado dessa composição pode ser visto a seguir:

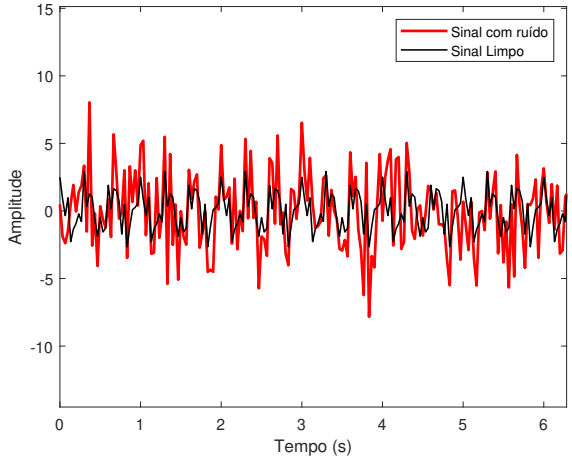


Figura 5: Sinal utilizado para as análises temporal e numérica

A curva foi então salva e utilizada posteriormente para os cálculos de análise temporal e numérica. Foram extraídos 4096 (2^{12}) amostras, com o intuito de formatar o sinal na representação em que a FFT é mais eficiente possível (N sendo potência de 2). Para garantir a maior

precisão numérica possível, os pontos foram armazenados em binário no formato de 64-bits especificado pela IEEE 754, que normaliza a representação de variáveis de ponto flutuante.

IV. ANÁLISE NUMÉRICA

Após computar a FFT, é preciso tratar o sinal para se tirar alguma informação tangível. O primeiro a se fazer é computar o módulo dos pontos complexos resultantes da transformada, a fim de se ter o plot conhecido no domínio da frequência (na forma de potência do sinal). Após isso, centraliza-se o resultado em torno do zero, já que segundo o teorema de Nyquist a maior componente em frequência possível de ser observada situa-se na metade da frequência de amostragem.

Feito isso, tem-se como resultado o seguinte gráfico:

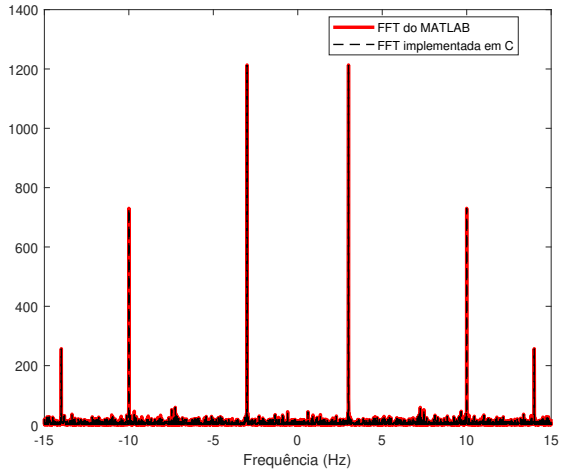


Figura 6: Comparação entre a FFT implementada em C e o MATLAB

É importante destacar a precisão apresentada no cálculo, já que os pontos ficam virtualmente nas mesmas posições no plano. A fim de confirmar isso, métricas podem ser calculadas, tais como:

1. **Erro quadrático médio**

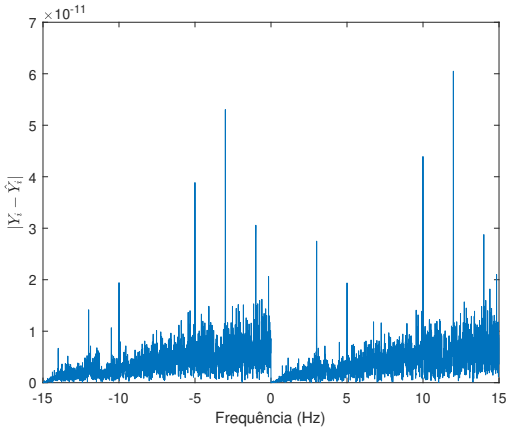
$$MSE = \frac{1}{n}\sum_{i=1}^{n}(Y_i - \hat{Y}_i)^2 = 2.913 \cdot 10^{-23}$$

   Onde o sinal computado pelo MATLAB é tido como Y_i e os valores dados pela implementação em C como $\hat{Y}_i$.

2. **Erro máximo**

$$M = max(|(Y_i - \hat{Y}_i)|) = 6.049 \cdot 10^{-11}$$

Com esses dados, é possível afirmar que a discrepância associada ao resultado do MATLAB em comparação com a implementação em C é praticamente inexistente. Na figura a seguir, isso se confirma ao olhar a escala do módulo do erro calculado em cada ponto em relação ao eixo y.



V. ANÁLISE TEMPORAL

Para obter métricas confiáveis em relação ao tempo de execução do código, o algoritmo da FFT foi executado 1 milhão de vezes no total. Como o programa executa de forma tão rápida que até por meio da contagem de ciclos de instrução se torna difícil de rastrear sua duração com uma boa precisão numérica, o código foi executado dividido em pacotes (*batches*). Cada *batch* rodava a FFT cem vezes, e retornava o tempo total gasto nestas computações. Esse dado então era publicado no terminal, e levado em seguida para o ambiente de desenvolvimento do MATLAB. No final, executaram-se 10000 *batches*, fornecendo assim uma grande quantidade de amostras para a análise.

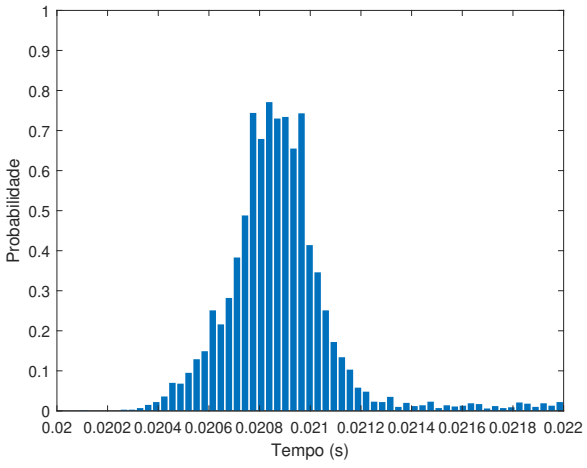


Figura 7: Recorte do histograma com a probabilidade relativa temporal do tempo de computação da FFT

Destes dados, também pode-se tirar métricas escalares, tais como:

Métrica	Valor (s)
Média	0.020455s
Desvio padrão	0.002727s
Variância	$7.438 \cdot 10^{-}6$s
Tempo máximo	0.037830s
Tempo mínimo	0.005980s

Tabela I: Métricas analíticas relacionadas ao tempo de computação

Com os pontos do histograma devidamente parametrizados, há uma última informação interessante que pode ser computada: ao linearizar uma curva com os pontos, é possível calcular a probabilidade de determinada faixa de tempo ocorrer ao estimar a integral da função linearizada e normalizada. Para isso, adotou-se como referência um somatório de curvas gaussianas, que podem ser representadas como:

$$f(x) = a_1 \cdot e^{-(\frac{x-b_1}{c_1})^2} + a_2 \cdot e^{-(\frac{x-b_2}{c_2})^2} + \cdots + a_8 \cdot e^{-(\frac{x-b_8}{c_8})^2}$$

Onde os valores de a_n, b_n e c_n podem ser consultados no apêndice deste documento.
Com isso, a curva linearizada fica:

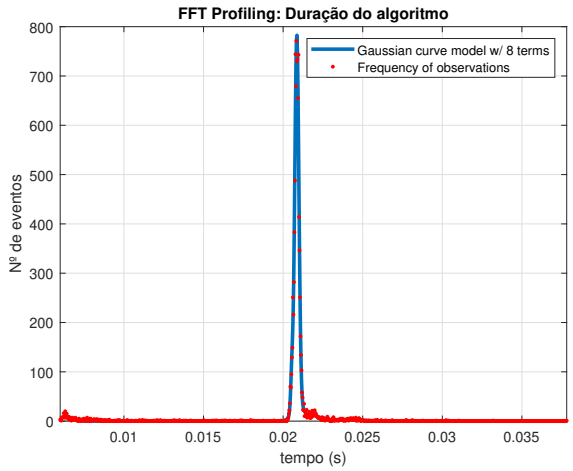


Figura 8: Curva gaussiana interpolada com 8 termos

Com isso, somos capazes de calcular a probabilidade de qualquer instante arbitrário, como por exemplo os eventos escolhidos:

Evento	Probabilidade
$t < 0.02s$	3.031%
$t > 0,025s$	0.270%
$0.02 < t < 0.022s$	93.08%

Tabela II: Eventos sortidos escolhidos pelo autor

No entanto, ainda assim a implementação do algoritmo feita pelo MATLAB é muito mais eficiente. As técnicas de otimização implementadas permitem reproduzir os mesmos cálculos em uma velocidade praticamente 10x mais rápido. Como não temos acesso ao código fonte do produto, foram utilizadas as funções *tic* e *toc* do próprio programa a fim de registrar o tempo de computação. A estratégia de *batches* foi mantida a mesma, tendo rodado o código 1 milhão de vezes no total.

Programa	Tempo médio (ms)
C	20.455
MATLAB	0.0685

Tabela III: Comparação entre o tempo das implementações

VI. CONCLUSÕES

Neste experimento, investigamos mais a fundo a implementação do cálculo da DFT através da *Fast Fourier Transform*. A respeito da precisão numérica, é possível afirmar que não há mudanças significativas nos cálculos para realizar tal tarefa, ou seja, a implementação foi um sucesso. No entanto, os resultados temporais nos apresentam discrepâncias relevantes. A implementação básica da FFT de base 2 com decimação no tempo de forma recursiva chega a ser mais de 100x mais lenta se comparada aos cálculos otimizados do algoritmo implementado pelo MATLAB.

VII. REFERÊNCIAS

[1] Cooley, J., and J. Tukey. *"An algorithm for machine calculation of complex Fourier series. Mathematics of computing, reprinted 1972."* Digital signal processing. IEEE Press, New York, NY (1965): 223-227.

[2] Heideman, Michael, Don Johnson, and Charles Burrus. *"Gauss and the history of the fast Fourier transform."* IEEE ASSP Magazine 1.4 (1984): 14-21.

[3] Wu, Ja-Ling J. *"Fast Fourier Transform (FFT)"*. [online] Available at: https://www.cmlab.csie.ntu.edu.tw/cml/dsp/training/coding/transform/fft.html [Accessed 21 Aug 2022]

[4] Bader, Michael *"Algorithms of Scientific Computing: Fast Fourier Transform"*. 2018. [online] Available at:

https://www5.in.tum.de/lehre/vorlesungen/asc/ss18/fft.pdf [Accessed 21 Aug 2022]
[5] PID Micro Repository. 2022. [online] Available at: https://github.com/kkkiq/myfft [Accessed 22 Aug 2022]

Apêndice A: Coeficientes da curva linearizada

$$
\begin{aligned}
a_1 &= 748.8;\\
b_1 &= 0.02086;\\
c_1 &= 0.0001824;\\
a_2 &= 32.27;\\
b_2 &= 0.02087;\\
c_2 &= 0.0004331;\\
a_3 &= 74.43;\\
b_3 &= 0.02055;\\
c_3 &= 0.0001032;\\
a_4 &= 9.984;\\
b_4 &= 0.02162;\\
c_4 &= 0.0007228;\\
a_5 &= 8.714;\\
b_5 &= 0.006366;\\
c_5 &= 0.0006109;\\
a_6 &= 8.03;\\
b_6 &= 0.02197;\\
c_6 &= 0.0001355;\\
a_7 &= 2.578;\\
b_7 &= 0.0234;\\
c_7 &= 0.001759;\\
a_8 &= 2.073;\\
b_8 &= 0.02444;\\
c_8 &= 0.000431;
\end{aligned}
$$

Apêndice B: Recursos computacionais

Todas as simulações e análises descritas neste artigo foram feitas utilizando os recursos do software *MATLAB*, da *MathWorks Inc.*; O código foi desenvolvido usando a IDE *Visual Studio Code*, da *Microsoft Inc.*; Todas as bibliotecas utilizadas no desenvolvimento do código são resguardadas pela GNU General Public License (GPL).