UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

Trabalho DFT e FFT

Projeto e Análise de Algoritmos

Autores:

Antonio César de Andrade Júnior - 473444

David Machado Couto Bezerra - 475664

Anderson Moura Costa do Nascimento - 473070

Professor: Criston Pereira de Souza

Quixadá - Ceará
30/11/21

Conteúdo

1 Introdução

Os algoritmos que realizam o processo da transformada de Fourier são importantes, pois são utilizados em vários tipos de aplicações envolvendo processamento de sinais digitais, entre eles, são exemplos: imagens e áudio. A transformada de Fourier é uma maneira de se obter uma nova visão de um sinal qualquer, levando para uma representação que traz informações sobre a frequência desse sinal.

Um exemplo simples é observado nas figuras 1 e 2.

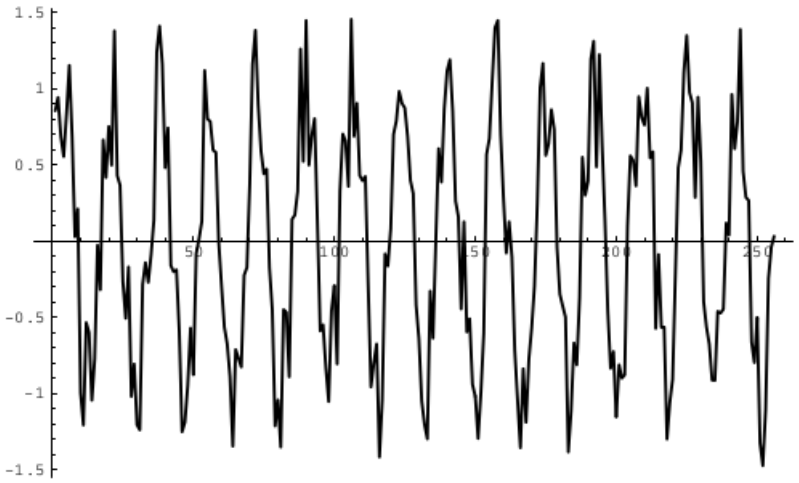

Figura 1: Sinal de entrada do algoritmo.

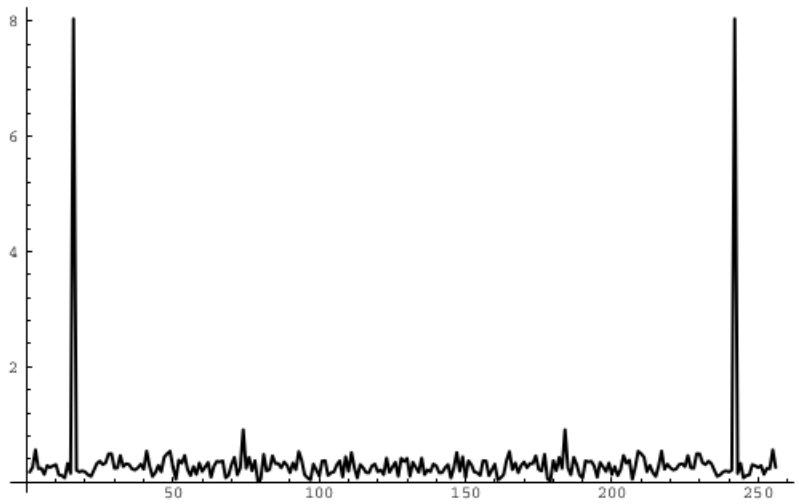

Figura 2: Transformada de Fourier do sinal de entrada.

O primeiro sinal é um vetor de entrada que é processado utilizando a transformada de Fourier de tempo discreto. Ela é definida pela seguinte fórmula:

$$\hat{f}_k = \sum_{i=0}^{n-1} f_i e^{\frac{-2\pi i jk}{n}}$$

Aonde f_k é um elemento na posição k do vetor de dados qualquer, $\hat{f}_k$ é o array resultante, j a unidade imaginária. Podemos definir o exponencial complexo como $\omega_n = e^{\frac{-2\pi j}{n}}$,

onde ω_n representa uma componente de frequência. O que pode ser visto pela fórmula, que é a mesma fórmula entre multiplicações de matrizes, com isso podemos definir uma estrutura de dados de matriz para resolver o problema.

A matriz pode ser vista a seguir

$$\begin{vmatrix} 1 & 1 & 1 & ... & 1 \\ 1 & \omega_n & \omega_n^2 & ... & \omega_n^{n-1} \\ 1 & \omega_n^2 & \omega_n^4 & ... & \omega_n^{2(n-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega_n^{(n-1)} & \omega_n^{2(n-1)} & ... & \omega_n^{(n-1)(n-1)} \end{vmatrix}$$

E o algoritmo da transformada discreta de Fourier se baseia em realizar a multiplicação dessa matriz com o vetor de dados que representa o sinal. O algoritmo do DFT realiza a multiplicação direta, enquanto o FFT faz uma fatoração da matriz realizando uma divisão e conquista.

$$\begin{vmatrix} \hat{f}_1 \\ \hat{f}_2 \\ \hat{f}_3 \\ \vdots \\ \hat{f}_n \end{vmatrix} = \begin{vmatrix} 1 & 1 & 1 & ... & 1 \\ 1 & \omega_n & \omega_n^2 & ... & \omega_n^{n-1} \\ 1 & \omega_n^2 & \omega_n^4 & ... & \omega_n^{2(n-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega_n^{(n-1)} & \omega_n^{2(n-1)} & ... & \omega_n^{(n-1)(n-1)} \end{vmatrix} \begin{vmatrix} f_1 \\ f_2 \\ f_3 \\ \vdots \\ f_n \end{vmatrix}$$

Com isso, os algoritmos se baseiam nesse cálculo para obter a saída, porém a diferença entre os dois se baseia em como vai ser feito essa multiplicação. A matriz em questão é uma famosa matriz que recebe o nome de matriz de Vandermonde.

A maneira que o algoritmo da FFT funciona se baseia na divisão e conquista da matriz do DFT, baseando na propriedade que $\omega_{2n}^2 = (e^{-2j\pi/2n})^2 = e^{-2j\pi/n} = \omega_n$, Considerando F_n a matriz de Vandermonde da DFT,com isso se pode escrever:

$$F_{2n} = \begin{vmatrix} I & D \\ I & D \end{vmatrix} \begin{vmatrix} F_n & \mathbf{0} \\ \mathbf{0} & F_n \end{vmatrix} P$$

Como pode ser visto, é possível quebrar a multiplicação da matriz de dimensões $2n$ em uma multiplicação de matrizes de dimensões n, e com isso entra a divisão em conquista. A matriz P é uma matriz de permutação que realiza a divisão dos elementos em pares e ímpares e assim fazendo a multiplicação com a primeira que possui os F_n e $\mathbf{0}$ é uma matriz de tamanho n com todos os valores n. A terceira matriz, é um matriz que possui duas matriz identidades I e matriz diagonais D:

$$D = \begin{vmatrix} 1 & 0 & 0 & ... & 0 \\ 0 & \omega & 0 & ... & 0 \\ 0 & 0 & \omega^2 & ... & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & ... & \omega^{n-1} \end{vmatrix}$$

Com isso temos que estamos utilizando de cálculos simples de matrizes que podem diminuir a complexidade alta que o DFT tinha, um exemplo é a matriz P que seu custo

é pequeno, assim como uma multiplicação por uma matriz identidade I e uma matriz diagonal D, e levando em conta que podemos quebrar uma matriz F_n em duas menores e assim indo, até o caso base. Assim podemos simplificar a complexidade para calcular a transformada de Fourier.

2 Pré-condições

- Tamanho do vetor de dados tem que ser maior que zero e potência de 2.
- Os elementos do vetor tem que ser valores reais.

3 Pós-condições

O algoritmo retorna um vetor de dados de valores complexos que representa a transformada do vetor original.

4 Transformada de Fourier discreta DFT

4.1 Corretude da parte 1

Para provar a corretude desse passo, consideramos o invariante do loop que afirma que vetor x_temp terá todos os seus valores, até então vistos, do tipo complexo.

Como na linha 2 o vetor não é composto por nenhum elemento, então ele satisfaz o invariante por vacuidade. Além disso, como é um laço for, então o laço termina. E tendo em vista que a cada iteração, o valor da posição i do vetor x_{in} primeiro tem seu tipo alterado de real para complexo e depois colocado em x_temp, então garantimos o invariante.

4.2 Corretude da parte 2

A corretude dessa parte é similar a anterior. Logo, consideramos o invariante do loop que afirma que vetor dft_mat terá todos os seus valores, até então vistos, do tipo complexo.

Como na linha 6 a matriz não é composta por nenhum elemento, então ele satisfaz o invariante por vacuidade. Além disso, como é dois laços for's, então ambos terminam quando $a > n$. Tendo em vista que a cada iteração do laço mais interno, um novo valor 0 do tipo complexo é colocado na matriz dft_mat, então garantimos o invariante.

4.3 Corretude da Parte 3

A corretude dessa parte está relacionada a preencher com exponenciais complexas a matriz DFT. Para invariante do loop podemos considerar sendo, para dada matriz resultado chamada dft_mat, todos os valores até então produzidos para ela serão exponenciais complexas.

Para a primeira iteração, a invariante de loop é satisfeita, pois dft_mat recebe uma lista de zeros de dimensão KN do tipo complexo da parte 2.

Em uma iteração qualquer que não seja a de parada do laço for mais externo, como os valores a e k são constantes que irão de 1 até n em ambos os for's, então não irá alterar o fato de ser gerado uma exponencial complexa na posição [k,a] da matriz dft. Assim, não invalidando a invariante.

Como estamos trabalhando com dois for's que não sofrem alterações nas variáveis a e k, ao alcançar o valor n, o algoritmo termina. Ao final dos laços for's podemos observar que é formada uma uma matriz dft composta por valores exponenciais complexos, satisfazendo nossas pós-condições.

4.4 Corretude da parte 4

Para inversão na dimensão do vetor x_{temp}, atribuimos o invariante relacionado ao fato de cada valor atribuído de x_{temp} para x seja complexo.

Como na linha 18, nenhum valor é atribuído de x_{temp} em x, então por vacuidade garantimos o invariante. Além disso, por ser um for que irá de 1 até o tamanho de x_{temp} (ou seja, n), o laço terminará. Assim, como em cada iteração do laço os valores de x_{temp} já são do tipo complexo e são apenas adicionados na lista (linha) da posição i da matriz x, então não invalida o invariante.

4.5 Corretude da parte 5

Na parte 5 é realizado a multiplicação da matriz dft_mat com a matriz x. Para provar a corretude desse passo, estabelecemos dois invariantes. O primeiro invariante é x ser complexo e o segundo é que dft_mat precisa ser uma matriz de dimensão NxN.

Como os valores de x está com tipo complexo pela parte 1 e dft_mat é uma matriz NxN obtida nos passos 2 e 3, então a primeira interação satisfaz as invariantes.

Por fim, a etapa final é realizar a multiplicação entre matrizes, assim implementado pelos for's da linha 31 até 39. Como a multiplicação gera uma nova matriz sem alterar os valores ou dimensão de dft_mat, então satisfazemos o segundo invariante. Da mesma forma, segue que x não possui o tipo dos seus dados alterados, continuando tendo dados do tipo complexo e satisfazendo o primeiro invariante.

4.6 Pseudocódigo

Algorithm 1 DFT(x_{in})

Require: $x_{in}(Array)$; Sinal a ser transformado
Ensure: $X(Array)$; Transformada de fourier do sinal x

$n \leftarrow len(x)$ ▷ Tamanho do array x_{in}
▷ Parte 1 - Inicio: Transformando os valores inteiros da lista x_{in} em complexo, colocando eles em x_{temp};
$x_{temp} \leftarrow []$
for `i in 1..len(`x_{in}`)` **do**
    $x_{temp}.append(complex(x_{in}[i]))$
end for ▷ Parte 1 - Fim
▷ Parte 2 - Inicio : Criando uma matriz nxn composta por valores complexos;
$dft\_mat \leftarrow []$
for `a in 1..n` **do**
    $dft\_mat.append([])$
    for `b in 1..n` **do**
        $dft\_mat[a].append(complex(0))$
    end for
end for ▷ Parte 2 - Fim
▷ Parte 3 - Inicio : Preenchendo a matriz dft_mat com exponenciais complexas;
for `a in 1..n` **do**
    for `k in 1..n` **do**
        $dft\_mat[k, a] = exp(-2j * \pi * k * a/n)$
    end for
end for ▷ Parte 3 - Fim
▷ Parte 4 - Inicio : Faço uma inversão na dimensão de x_{temp}, colocando os valores do vetor x_{temp} (com dimensão 1xn) na matriz x (com dimensão nx1);
$x \leftarrow []$
for `i in 1..len(`x_{temp}`)` **do**
    $x.append([])$
    $x[i].append(x_{temp}[i])$
end for ▷ Parte 4 - Fim
▷ Parte 5 - Inicio : Faço a multiplicação de matrizes entre dft_mat (com dimensão nxn) e x (com dimensão nx1);
$num\_linhas\_dft\_mat \leftarrow len(dft\_mat)$
$num\_colunas\_dft\_mat \leftarrow len(dft\_mat[0])$
$num\_linhas\_x \leftarrow len(x)$
$num\_colunas\_x \leftarrow len(x[0])$
if $num\_colunas\_dft\_mat \neq num\_linhas\_x$ **then**
    return Error
end if
$X \leftarrow []$ ▷ Onde salvo os valores da multiplicação
for `linha in 1..`*`num_linhas_dft_mat`* **do**
    $X.append([])$
    for `coluna in 1..`*`num_coluna_x`* **do**
        $X[linha].append(0)$
        for `k in 1..`*`num_linhas_dft_mat`* **do**
            $X[linha][coluna] \leftarrow X[linha][coluna] + (dft\_mat[linha][k] * x[k][coluna])$
        end for
    end for
end for ▷ Parte 5 - Fim
return X

4.7 Complexidade do algoritmo DFT

- **Linha 1:** Complexidade $O(n)$. Seguindo o material oficial do Python.

- **Parte 1:** Complexidade $O(n)$. Pois, na linha 2 temos complexidade $O(1)$, ja que estamos atribuindo uma lista vazia a x_{temp}. Além disso, na linha 3 temos um laço com n iterações, logo com complexidade $O(n)$. E na linha 4, temos complexidade $O(1)$ para cada iteração do laço da linha 3, já que estamos transformando os n's dados de x no tipo complexo.

- **Parte 2:** Complexidade $O(n^2)$. Pois, na linha 6 estamos atribuindo uma lista vazia a dft_mat (logo, complexidade $O(1)$), na linha 7 temos um laço com n iterações (logo, complexidade $O(n)$), na linha 8 temos uma append em cada uma das n's iterações do *for* da linha 7 (logo, complexidade $O(n)$) e na linha 9 temos um laço com n iterações que será repetido em cada uma das n's iterações do *for* da linha 7 (logo, complexidade $O(n^2)$). Por fim, na linha 10 temos complexidade $O(n^2)$, pois o comando com complexidade $O(1)$ está sendo repetido n vezes pelo laço da linha 9 em cada uma das n iterações do *for* da linha 7.

- **Parte 3:** Complexidade $O(n^2)$. Pois, na linha 13 temos n iterações (assim, essa linha tendo complexidade $O(n)$), enquanto na linha 14 temos um laço com n iterações que será executado em cada uma das n's iterações do *for* da linha 13 (assim, essa linha tendo complexidade $O(n^2)$) e na linha 14 uma atribuição para dft_mat[k,a], que será executada n vezes pelo laço da linha 14 em cada uma das n iterações do *for* da linha 13 (assim, essa linha tendo complexidade $O(n^2)$).

- **Parte 4:** Complexidade $O(n)$. Já que na linha 18 temos apenas um lista vazia sendo atribuída em x (assim, essa linha tendo complexidade $O(1)$), enquanto na linha 19 temos um laço com n iterações e nas linhas 20 e 21 operações com custos $O(1)$ em cada uma das n's iterações do laço da linha 19.

- **Parte 5:** Complexidade $O(n^2)$. Onde, da linha 23 até a linha 30 temos complexidade $O(1)$ e da linhas 31 até a linha 34 complexidade $O(n)$, já que as operações serão executadas n vezes por causa do laço da linha 31. Enquanto isso, nas linhas 35 e 36 temos complexidade $O(n^2)$, pelo fato dos laços das linhas 31,33 e 35 terem respectivamente n,1 e n iterações.

  Vale ressaltar, que a linha 33 terá apenas 1 iteração, já que a quantidade de colunas da matriz x é igual a 1. Por isso que ele não influenciará tanto a complexidade das operações internas em seu escopo. Além disso, que as operações da linha 36 tem complexidade $O(1)$, por serem apenas operações de multiplicação, adição e atribuição, mas que serão executadas uma quantidade de vezes determinada pelos laços das linhas 31,33 e 35 (assim tendo complexidade $O(n^2)$).

- **Linha 40:** Complexidade $O(1)$. Já que é apenas uma operação retorno.

4.8 Avaliação Empírica do DFT

Os resultados obtidos utilizaram o procedimento "razão dobrando"para determinar o grau do polinômio d, começando do valor $N = 16$:

Razão Dobrando		
Valor N	Valor da Razão	Tempo de execução (em segundos)
32	3.25666492420282	0.0014853477478
64	4.048314606741573	0.0060131549835
128	8.442805598509178	0.0507678985596
256	1.797455573505654	0.0912530422211
512	3.723853865387479	0.3398129940033
1024	3.819244512828716	1.2978289127349
2048	3.988439950921183	5.1763126850128
4096	3.991122806754116	20.659299612045
8192	4.048314606741573	83.136213541031
16384	4.153319718536371	345.29127502441

O procedimento foi executado até que a execução passe do tempo de 5 minutos, como visto no caso N = 16384, em que o tempo em segundos é aproximadamente 345, e convertendo em minutos é 5.75 minutos. Com essa quantidade de execuções do algoritmo podemos ver que o valor na qual a razão estava tendendo era 4 e assim fazendo o $\log_2$ desse valor obtemos o grau do polinômio, $\log_2 4 = 2$, com isso temos um polinômio de grau 2. O que combina com o polinômio de grau 2 presente na complexidade teórica de pior caso, $O(n^2)$.

Achar o valor de c, através da expressão $T(n) = cn^2$, pegando o caso n = 1024, temos que:

$$c = \frac{1.2978289127349}{1024^2} = 1.2377061011647 \cdot 10^{-6}$$

Com isso obtemos a expressão polinomial do tempo de execução do DFT. Agora vai ser feita a previsão do valor, utilizando como caso n = 8192, por conta que depois de 16384, programa fica sem memoria pra alocar.

$$T(16384) = 1.2377061011647 \cdot 10^{-6} \cdot 16384^2 = 332.24420166012$$

A diferença percentual com o valor medido é:

$$\text{Diferença percentual} = 100 \cdot \frac{|345.29127502441 - 332.24420166012|}{332.24420166012} = 3.92\%$$

4.9 Exemplo demonstrativo

É evidente que as dimensões da matriz dft (composta por exponenciais complexas) cresce de acordo com a variável n, mas sem alterar seus dados para um n especifico. Por isso, tivemos que escolher um valor de n pequeno ($n = 4$) para realizar um teste de mesa do algoritmo DFT. Além de escolhermos esse sinal de entrada sendo $\sin(40 \cdot 2 \cdot \pi \cdot t) + 0.5 \cdot \sin(90 \cdot 2 \cdot \pi \cdot t)$. Vale lembrar, que n é a quantidade de pontos do sinal de entrada.

A fim de melhor ilustrar o teste de mesa, faremos ele para cada parte do algoritmo separadamente. Sabendo que a entrada x_in é o seguinte vetor:

$$\begin{bmatrix} 0.00000000e+00 & -1.47019514e-15 & -6.85678411e-15 & -3.28322948e-14 \end{bmatrix}$$

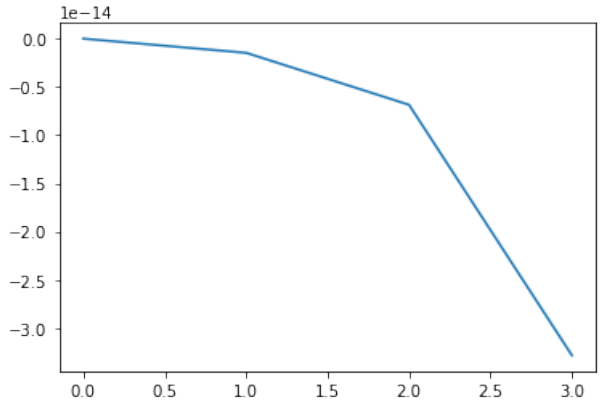

Figura 3: Entrada do exemplo de dft.

Na primeira parte, o sinal de entrada (x_in) tem o tipo dos seus dados transformados em complexo e salvos no vetor x_temp.

$$\left(0j,\ -1.4701951396134124e{-}15{+}0j,\ -6.856784113952001e{-}15{+}0j,\ -3.2832294849244245e{-}14{+}0j \right)$$

Enquanto na segunda parte, é criado a matriz dft_mat com dimensão 4x4 e que será formada por zeros do tipo complexo.

$$\begin{bmatrix} 0j & 0j & 0j & 0j \\ 0j & 0j & 0j & 0j \\ 0j & 0j & 0j & 0j \\ 0j & 0j & 0j & 0j \end{bmatrix}$$

Já na terceira parte, substituímos os valores complexos ($0j$) da posição (k,a) pelos valores obtidos da seguinte expressão:

$dtf\_mat[k, a] = e^{-2j \cdot \pi \cdot k \cdot a/n}$

Assim obtendo a seguinte matriz dft_mat para $n = 4$:

$$\begin{pmatrix} (1{+}0j) & (1{+}0j) & (1{+}0j) & (1{+}0j) \\ (1{+}0j) & (6.123233995736766e{-}17{-}1j) & (-1{-}1.2246467991473532e{-}16j) & (-1.8369701987210297e{-}16{+}1j) \\ (1{+}0j) & (-1{-}1.2246467991473532e{-}16j) & (1{+}2.4492935982947064e{-}16j) & (-1{-}3.6739403974420594e{-}16j) \\ (1{+}0j) & (-1.8369701987210297e{-}16{+}1j) & (-1{-}3.6739403974420594e{-}16j) & (5.51091059616309e{-}16{-}1j) \end{pmatrix}$$

Com objetivo de trabalhar com um produto entre matrizes, fizemos na parte 4 a inversão da dimensão do vetor x_{temp}, utilizando a variável x para não perdemos os dados de

x_{temp} no processo. Fazendo os dados do vetor x_{temp} sair de um vetor com dimensão 1x4 para uma matriz com dimensão 4x1.

$$\begin{bmatrix} (0\,j) \\ (-1.4701951396134124e-15+0\,j) \\ (-6.856784113952001e-15+0\,j) \\ (-3.2832294849244245e-14+0\,j) \end{bmatrix}$$

Por fim, na quinta parte e última parte, fizemos a multiplicação entre as matrizes dft_mat e x. Obtendo o seguinte resultado:

$$\begin{bmatrix} (-4.115927410280966e-14+0\,j) \\ (6.856784113952007e-15-3.136209970963083e-14\,j) \\ (2.744570587490566e-14+1.0563008672402537e-29\,j) \\ (6.856784113951983e-15+3.136209970963083e-14\,j) \end{bmatrix}$$

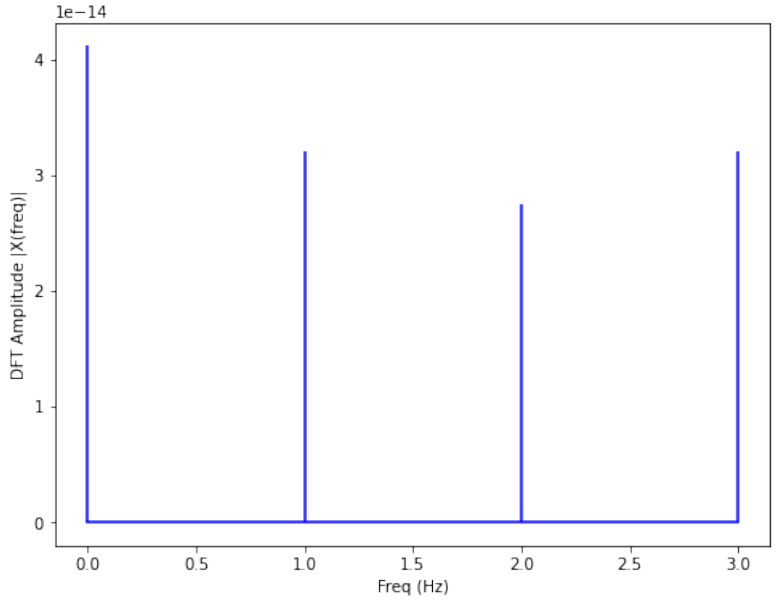

Figura 4: Entrada do exemplo de dft.

5 Transformada Rápida de Fourier FFT

5.1 Pseudocódigo

Algorithm 2 FFT(X)

1: $N \leftarrow len(x)$
2: **if** N == 1 **then** ▷ Caso base. Ou seja, quando N for igual a 1 o próprio x é retornado.
3: return x
4: **else**
5: $X_{even} \leftarrow FFT(x_0, x_2, x_4, ..., x_{2k})$ ▷ Faz a parte da divisão, da divisão e conquista, dividindo as componentes de x entre índices pares e ímpares.
6: $X_{odd} \leftarrow FFT(x_1, x_3, x_5, ..., x_{2k+1})$
7: $exp\_comp \leftarrow exp(-2j * pi * matriz(N)/N)$ ▷ Cria uma matriz de vandermonde de exponenciais complexas de tamanho N.
8: $X \leftarrow concatena(X_{even} + exp\_comp_{0->N/2} * X_{odd}, X_{even} + exp\_comp_{N/2->N-1} * X_{odd})$ ▷ Concatena a soma entre as componentes pares e o produto entre as componentes ímpares e matriz de exponenciais. Essa soma é dividida entre a primeira metade de matriz e segunda metade, indicadas por 0->N/2 e N/2->N-1, respectivamente. Ou seja, representa a conquista da divisão e conquista. A concatenação foi feita utilizando listas encadeadas, destruindo uma das listas quando uma aponta para a outra.
9: return X

5.2 Corretude por indução

Base: N=1. Como x possui apenas 1 valor, então é preciso retornar apenas x.

Hipótese: Suponha um sinal x de tamanho N=m, sendo m potência de 2, FFT(x) retorna a transformada de Fourier do sinal x. Ou seja, satisfaz as pré-condições e a pós-condição.

Passo indutivo: Assumindo agora um sinal x com tamanho N=2m, o algoritmo dividirá o vetor em 2 partes iguais, ou seja, as partes terão tamanho $\frac{2m}{2} = m$, porém, por hipótese, m satisfaz as pré-condições e a pós-condição, então 2m também é potência de 2 e possui valores reais, assim as pré-condições de N também são satisfeitas, fazendo assim FFT(x) retornar a transformada de x.

5.3 Prova da execução finita

Como N é de base 2, então em algum momento ele vai ser de tamanho 2, assim, dividindo-o por 2, o algoritmo entrará no caso base.

5.4 Exemplos

5.4.1 Exemplo N=4:

Sinal de Entrada: $\sin(40 \cdot 2 \cdot \pi \cdot t) + 0.5 \cdot \sin(90 \cdot 2 \cdot \pi \cdot t)$

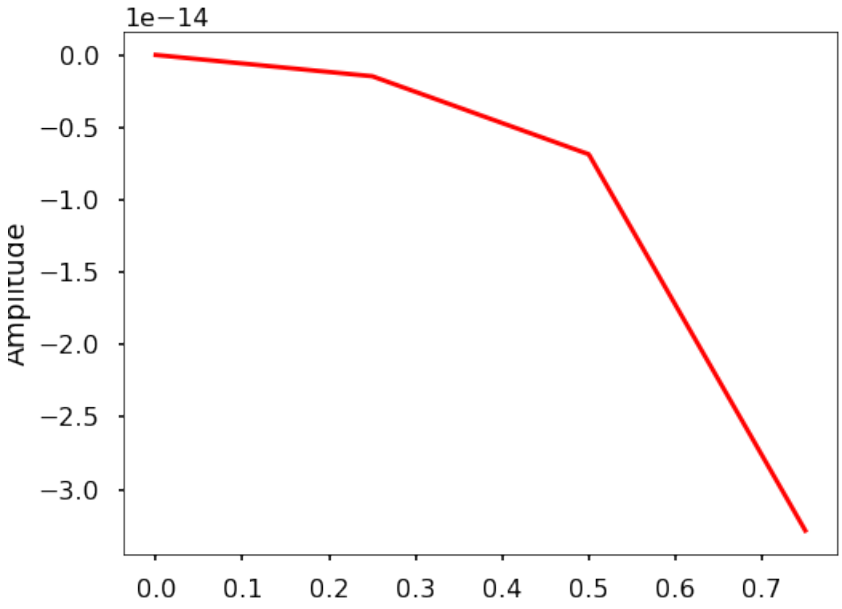

Figura 5: Sinal de entrada do exemplo de fft.

Saída:

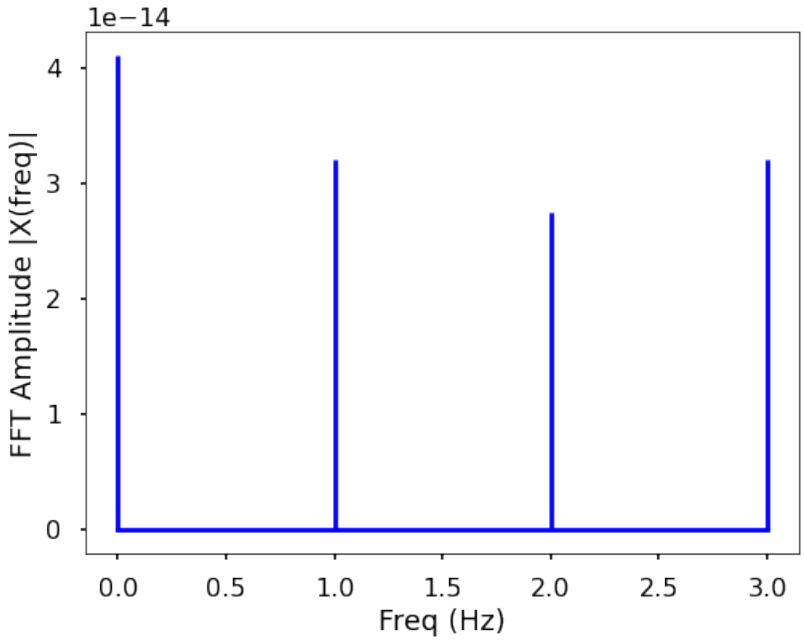

Figura 6: Saída do exempo do fft.

As chamadas recursivas irão gerar as sub-instâncias seguindo o diagrama da figura 10, em que cada nó é uma sub-instância gerada, N é o tamanho do vetor naquela instância, os nós a esquerda de seu nó pai referem-se as instâncias que possuem o vetor com os elementos de índices pares do nó anterior e aqueles a esquerda referem-se aos índices ímpares. Quando N=1 as chamadas acabam e será retornado X.

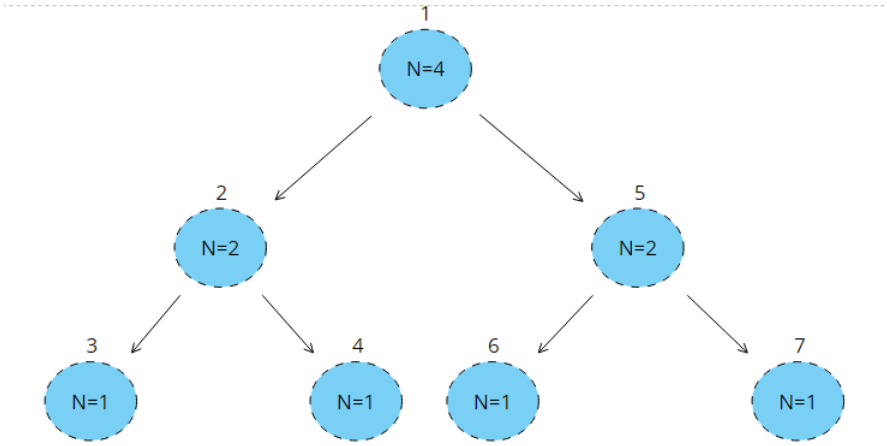

Figura 7: Diagrama das chamadas recursivas.

Depois da obtenção de uma matriz unitária de exponenciais complexas, é feita a concatenação, pelo eixo 0 (Seção 6.4), das somas das componentes retornadas, por exemplo, a primeira concatenação (referente a instância 2) será entre o valor da instância 3 (Xeven) e o valor da instância 4 (Xodd).

5.5 Medida de progresso

Mais da saída. Pois vai construindo as saídas ao decorrer das chamadas recursivas.

5.6 Instância

Vetor de entrada x com N índices.

5.7 Subinstâncias $\mathbf{I}_1$

Elementos de x com índice par.

5.8 Subinstâncias $\mathbf{I}_2$

Elementos de x com índice ímpar.

5.9 Complexidade FFT

5.9.1 Complexidade de cada linha

- **Linha 1:** Um comando de atribuição é O(1), a função len() também é O(1), então a complexidade dessa linha é O(1).
- **Linha 2 e 3:** Possuem apenas uma condicional if e um return, então tem complexidade O(1).
- **Linha 5 e 6:** Chamadas recursivas, portanto uma análise é feita depois.
- **Linha 7:** Um comando de atribuição é O(1). Para a geração da matriz de exponenciais foi usada a função arange, da biblioteca numpy, que possui complexidade O(N), ou seja, essa linha possui complexidade O(N).
- **Linha 8:** Um comando de atribuição é O(1). Para a concatenação foi utilizada a função concatenate, da biblioteca numpy, que possui complexidade O(1), pois, como citado antes, usa listas encadeadas e destrói uma das listas quando uma aponta para a outra, então essa linha tem complexidade O(1).
- **Linha 9:** Return possui complexidade O(1).

5.9.2 Complexidade da recursividade

N é dividido em 2 partes e FFT é chamada para as 2 partes, sendo assim, $a = b = 2$ e $log_b a = 1$, o que é igual a d, pois os custos adicionais em cada chamada recursiva é O(N) pela linha 7, e $e = 0$.

Com as afirmações anteriores, concluímos que a complexidade de FFT é O(NlogN).

5.10 Avaliação Empírica do FFT

Os resultados obtidos utilizaram o procedimento "razão dobrando"para determinar o grau do polinômio d, começando do valor $N = 16$:

Razão Dobrando		
Valor N	Valor da Razão	Tempo de execução (em segundos)
32	1.60418848167539	0.0003652572632
64	2.332898172323760	0.0008521080017
128	1.995523223279239	0.0017004013061
256	1.763740886146943	0.0029990673065
512	1.950473010573177	0.0058495998382
1024	2.970898716119829	0.0173785686492
2048	1.657419983262680	0.0288035869598
4096	1.666934302340019	0.0480136871338
8192	1.860852897946212	0.0893464088439
16384	1.983284678155337	0.1771993637085
32768	1.965460127982261	0.3482782840729
65536	2.009605794144929	0.6999020576477
131072	2.015222431378640	1.4104583263397
262144	2.012640192322678	2.8387451171875
524288	2.011908813292840	5.7112963199615
1048576	2.024955590529592	11.565121412277
2097152	1.992685831828329	23.045653581619
4194304	2.004500722668687	46.195029258728
8388608	2.017743760559544	93.209732055664
16777216	2.00541657055888	186.92434120178
33554432	2.01539923128655	376.72717356682

O procedimento foi executado até que a execução passe do tempo de 5 minutos, como visto no caso N = 33554432, em que o tempo em segundos é aproximadamente 377, e convertendo em minutos é 6.28 minutos. Com essa quantidade de execuções do algoritmo podemos ver que o valor na qual a razão estava tendendo era 2 e assim fazendo o $\log_2$ desse valor obtemos o grau do polinômio, $\log_2 2 = 1$, com isso temos um polinômio de grau 1.

Achar o valor de c, através da expressão $T(n) = cn$, pegando o caso n = 1048576, temos que:

$$c = \frac{11.565121412277}{1048576} = 0.000011029359257$$

Com isso obtemos a expressão polinomial do tempo de execução do FFT. Agora vai ser feita a previsão do valor, utilizando como caso n = 33554432.

$$T(33554432) = 0.000011029359257 \cdot 33554432 = 370.083885192$$

A diferença percentual com o valor medido é:

$$\text{Diferença percentual} = 100 \cdot \frac{|376.72717356682 - 370.083885192|}{370.083885192} = 1.8\%$$

Como o razão dobrando não reconhece logN, então a complexidade dada pelo mesmo é O(N), o que seria próximo do valor encontrado na submissão passada (NlogN).

6 Estruturas / abstrações adicionais

6.1 numpy.dot

Comando:

- numpy.dot(a, b, out=None)

Dot é o produto de dois arrays *a* e *b*, onde

- se *a* e *b* são arrays de uma dimensão, o resultado será um produto interno de vetores.
- Se *a* e *b* forem arrays de duas dimensões, então o resultado será uma multiplicação matricial.
- Se *a* ou *b* for um escalar, então o resultado será a multiplicação de um array por um escalar.
- Se *a* é um matriz com N dimensões, enquanto *b* um array de uma dimensão, então o resultado será um produto de soma sobre o último eixo de *a* e *b*.
- Se *a* for uma matriz com N dimensões e *b* uma matriz com M dimensões (onde $M \geq 2$), então o resultado será um produto de soma sobre o último eixo de *a* e o penúltimo eixo de *b*.

6.2 numpy.arange

Comando:

- numpy.arange([start,] stop[, step,], dtype=None, *, like=None)

Retorna valores espaçados uniformemente dentro de um determinado intervalo. Os valores são gerados dentro de um intervalo semi-aberto, onde é incluído no intervalo a entrada *start*, mas excluído a entrada *stop*. Onde, para argumentos inteiros, a função é equivalente à função *range* em Python, mas retorna um ndarray em vez de uma lista.

Ao utilizar na variável de entrada *step* valores não-inteiro, como 0.1, os resultados muitas vezes não serão consistentes.

6.3 numpy.complex128

Tipo de número complexo composto de dois números de ponto flutuante de precisão dupla, compatível com Python ‘complex‘.

6.4 numpy.concatenate

Em geral faz uma união entre duas sequências em Python (por exemplo dois arrays) na vertical (quando escolhido o eixo 0) ou na horizontal (quando escolhido o eixo 1).

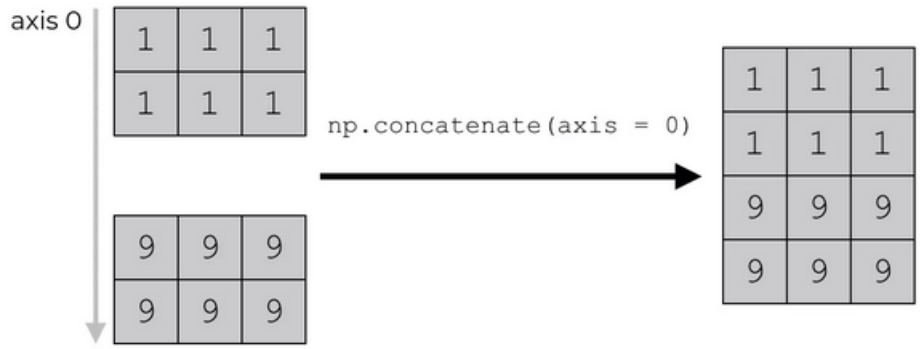

Figura 8: Concatenação sobre eixo 0.

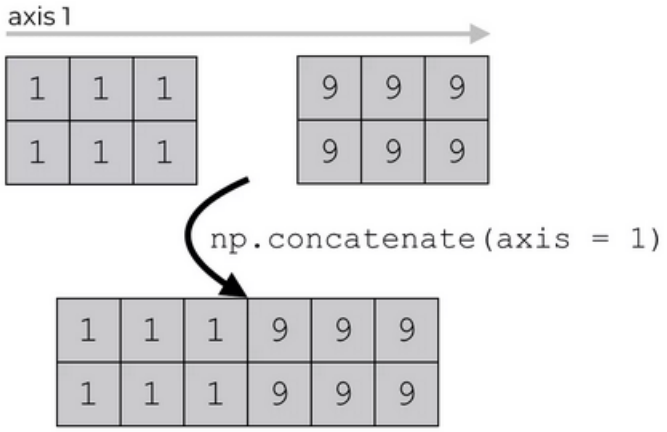

Figura 9: Concatenação sobre eixo 1.

7 Referências

1. (2022) https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-046j-design-and-analysis-of-algorithms-spring-2012/lecture-notes/MIT6_046JS12_lec05.pdf
2. (2022) https://ocw.mit.edu/courses/mathematics/18-06sc-linear-algebra-fall-2011/positive-definite-matrices-and-applications/complex-matrices-fast-fourier-transform-fft/MIT18_06SCF11_Ses3.2sum.pdf
3. (2022). NumPy. https://numpy.org/doc/stable/reference/index.html
4. Oppenheim, A. V., Willsky, A. S., and Nawab, S. H. (1996).Signals Systems (2nd Ed.).Prentice-Hall, Inc., USA.