

Paradigmas de Solução de Problemas

Divisão e Conquista – Transformada Rápida de Fourier

Prof. Edson Alves - UnB/FGA

2020

1. Transformada de Fourier
2. Transformada Rápida de Fourier
3. Implementação
4. Referências

Transformada de Fourier

Série de Fourier

- Uma série de Fourier consiste na expansão de uma função periódica $f(x)$ em termos de senos e cosenos
- Isto possível porque as funções $\sin(mx)$ e $\sin(ny)$ são ortogonais para $m \neq n$ no intervalo $[-\pi, \pi]$:

$$\begin{aligned}\int_{-\pi}^{\pi} \sin(mx) \sin(nx) dx &= \int_{-\pi}^{\pi} \sin(mx) \cos(nx) dx \\ &= \int_{-\pi}^{\pi} \cos(mx) \cos(nx) dx = 0\end{aligned}$$

- Para $m = n$, segue que

$$\int_{-\pi}^{\pi} \sin^2(mx) dx = \int_{-\pi}^{\pi} \cos^2(mx) dx = \pi$$

- Deste modo,

$$f(x) = \frac{1}{2}a_0 + \sum_{n=1}^{\infty} a_n \cos(nx) + \sum_{n=1}^{\infty} b_n \sin(nx),$$

onde

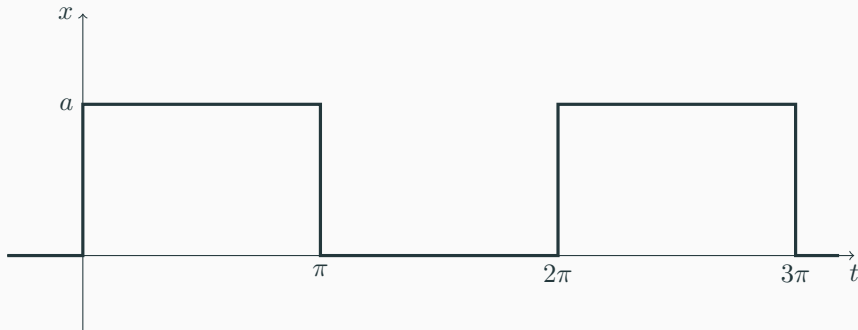
$$a_0 = \frac{1}{\pi} \int_{-\pi}^{\pi} f(x) dx$$

$$a_n = \frac{1}{\pi} \int_{-\pi}^{\pi} f(x) \cos(nx) dx$$

$$b_n = \frac{1}{\pi} \int_{-\pi}^{\pi} f(x) \sin(nx) dx$$

Exemplo: Onda Quadrada

Considere a onda quadrada abaixo:



Exemplo: Onda Quadrada

- O coeficiente a_0 é dado por

$$a_0 = \frac{1}{\pi} \int_{-\pi}^{\pi} f(t) dt = \frac{1}{\pi} \int_0^{\pi} a dt = a$$

- Os coeficientes a_n , para $n \geq 1$, são todos iguais a zero, pois

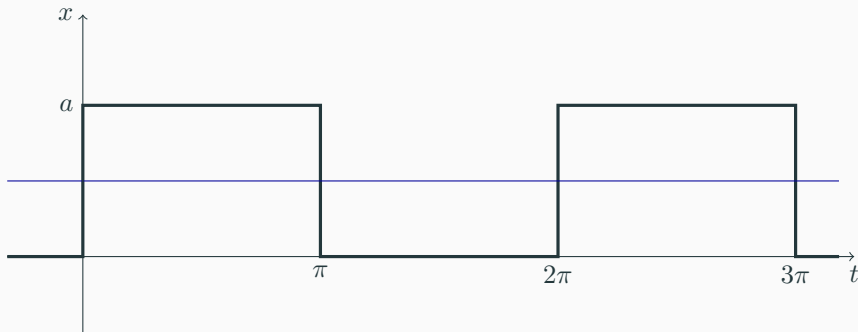
$$a_n = \frac{1}{\pi} \int_{-\pi}^{\pi} f(t) \cos(nt) dt = \frac{a}{\pi} \left[\frac{\sin(nt)}{n} \right]_0^{\pi} = 0$$

- Os coeficientes b_n são iguais a zero, para n par, e

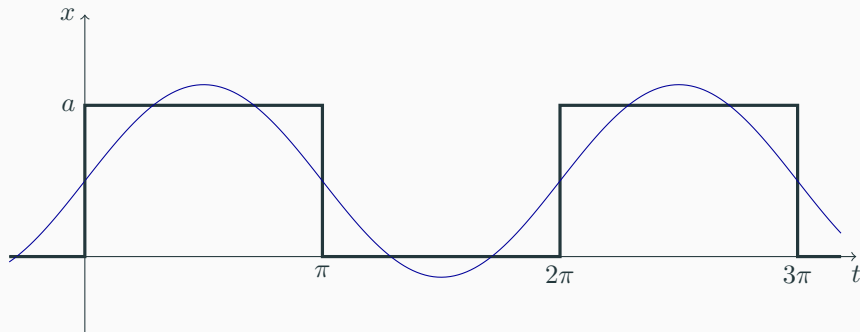
$$b_n = \frac{1}{\pi} \int_{-\pi}^{\pi} f(t) \sin(nt) dt = -\frac{a}{\pi} \left[\frac{\cos(nt)}{n} \right]_0^{\pi} = \frac{2a}{n\pi},$$

se n é ímpar

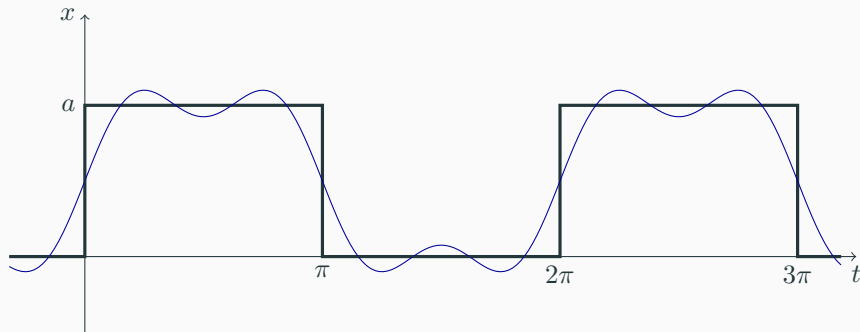
Exemplo: Aproximação da onda quadrada com $n = 0$



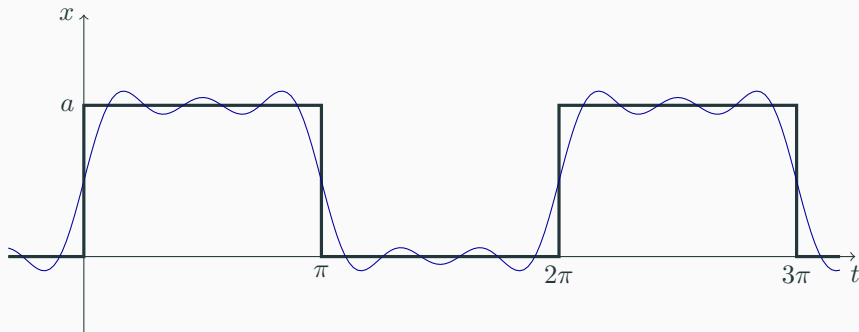
Exemplo: Aproximação da onda quadrada com $n = 1$



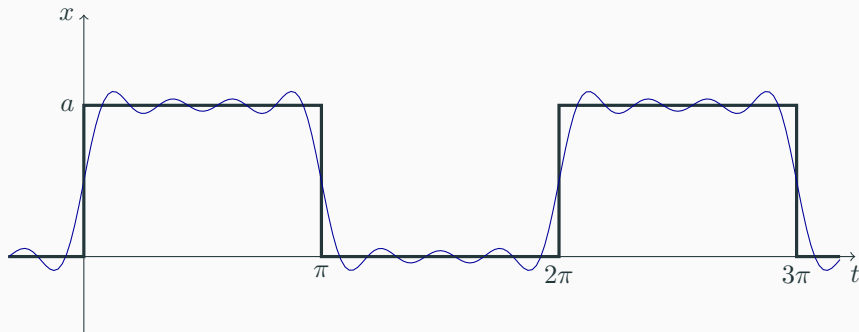
Exemplo: Aproximação da onda quadrada com $n = 3$



Exemplo: Aproximação da onda quadrada com $n = 5$



Exemplo: Aproximação da onda quadrada com $n = 7$



Série de Fourier com coeficientes complexos

- A série de Fourier pode ser estendida para coeficientes complexos a partir da observação que

$$e^{bi} = \cos b + i \sin b$$

- Seja $f(x)$ uma função nos reais. Faça

$$f(x) = \sum_{-\infty}^{\infty} A_n e^{inx}$$

- Assim, vale que

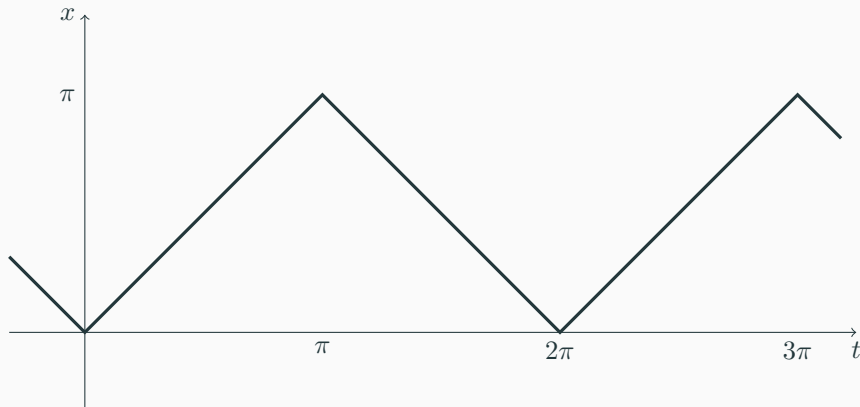
$$\int_{-\pi}^{\pi} f(x) e^{-imx} dx = 2\pi A_m,$$

de modo que

$$A_n = \frac{1}{2\pi} \int_{-\pi}^{\pi} f(x) e^{-inx} dx$$

Exemplo: Onda Triangular

Considere a onda triangular abaixo:



Exemplo: Onda Triangular

- No intervalo $[-\pi, \pi]$ temos que

$$f(x) = \begin{cases} -x, & \text{se } x \leq 0, \\ x, & \text{caso contrário} \end{cases}$$

- Daí

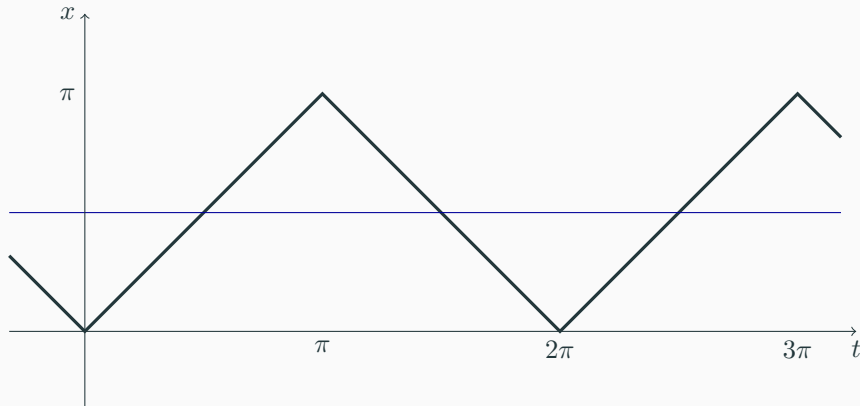
$$A_0 = \frac{1}{2\pi} \int_{-\pi}^{\pi} f(x) dx = \frac{\pi}{2}$$

- Para $n > 1$ ímpar vale que

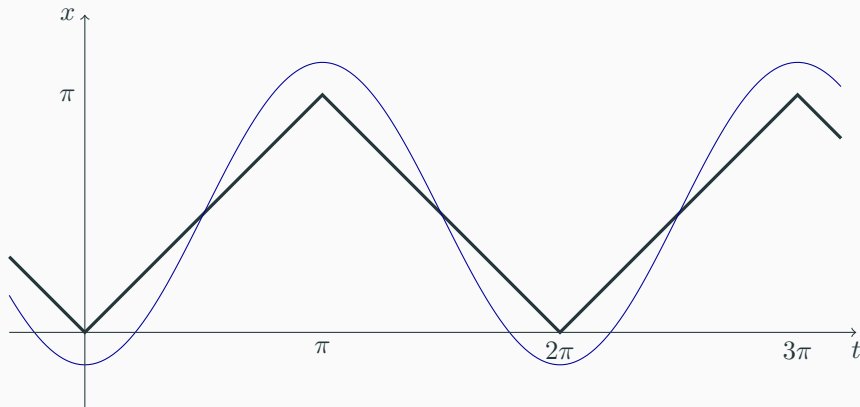
$$A_n = \frac{1}{2\pi} \int_{-\pi}^{\pi} f(x) e^{inx} dx = -\frac{4}{n^2}$$

- $A_n = 0$, se n é par

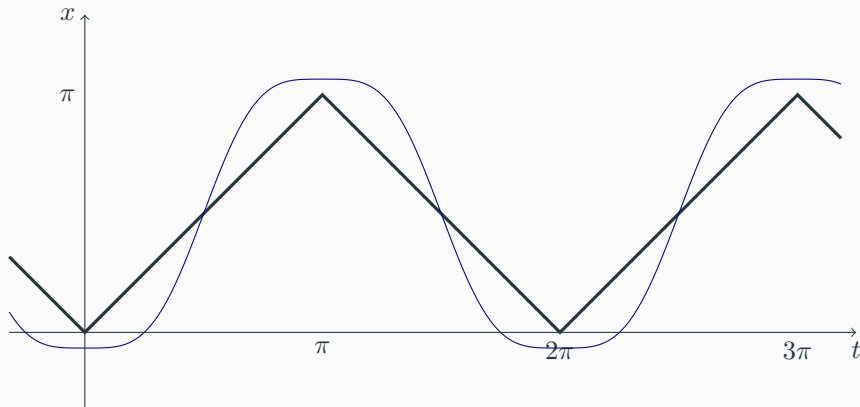
Exemplo: Aproximação com $n = 0$



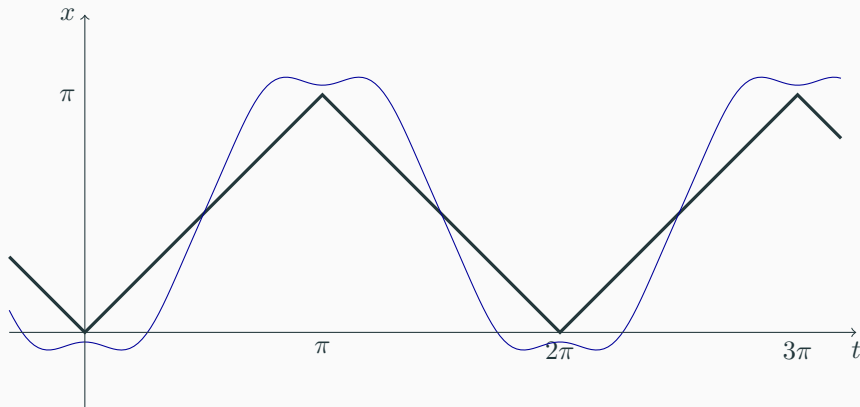
Exemplo: Aproximação com $n = 1$



Exemplo: Aproximação com $n = 3$



Exemplo: Aproximação com $n = 5$



Transformada de Fourier

- A transformada de Fourier é uma generalização das séries de Fourier com coeficientes complexos quando o período tende ao infinito
- Seja $f(x)$ uma função com um número finito de descontinuidades e tal existe a integral

$$\int_{-\infty}^{\infty} |f(x)| dx$$

- A Transformada de Fourier \mathcal{F} de $f(x)$ é dada por

$$\mathcal{F}[f(x)] = F(k) = \int_{-\infty}^{\infty} f(x) e^{-2\pi i k x} dx$$

- A Transformada Inversa \mathcal{F}^{-1} é dada por

$$\mathcal{F}^{-1}[F(k)] = f(x) = \int_{-\infty}^{\infty} F(k) e^{2\pi i k x} dk$$

- A Transformada de Fourier é linear:

$$\mathcal{F}[af(x) + bg(x)] = a\mathcal{F}[f(x)] + b\mathcal{F}[g(x)],$$

onde a e b são constantes

- A transformada da derivada da função está diretamente relacionada com a transformada da função

$$\mathcal{F}[f^{(n)}(x)](k) = (2\pi ik)^n \mathcal{F}[f(x)](k)$$

- **Teorema da Convolução:**

$$\mathcal{F}[f * g] = \mathcal{F}[f]\mathcal{F}[g]$$

Exemplo: Exponencial Decrescente

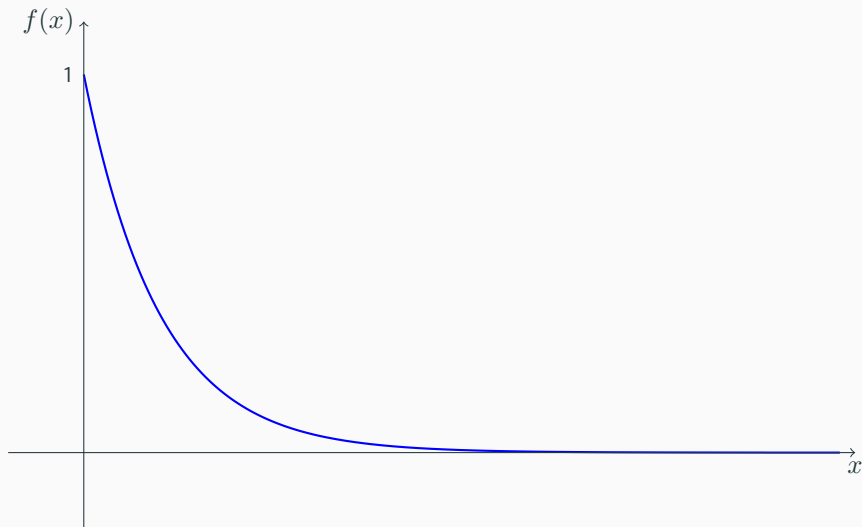
- Seja

$$f(x) = \begin{cases} e^{-x}, & \text{se } x \geq 0, \\ 0, & \text{caso contrário} \end{cases}$$

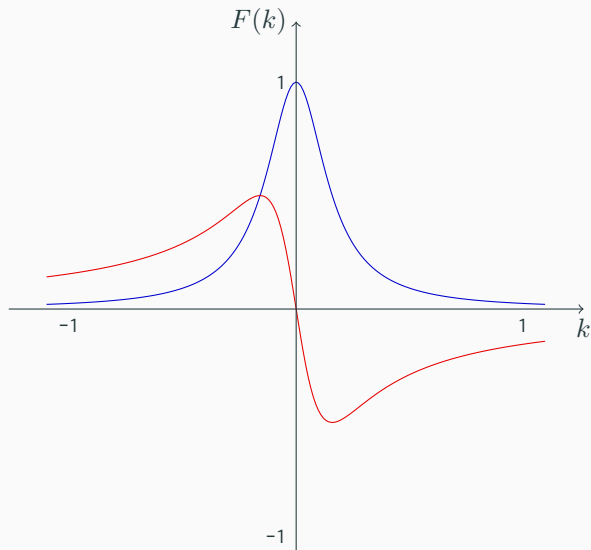
- Temos que

$$\begin{aligned} \mathcal{F}[f(x)] = F(k) &= \int_{-\infty}^{\infty} f(x)e^{-2\pi i k x} dx \\ &= \int_0^{\infty} e^{-x} e^{-2\pi i k x} dx = \int_0^{\infty} e^{-(1+2\pi i k)x} dx \\ &= -\frac{e^{-(1+2\pi i k)x}}{1+2\pi i k} \Big|_0^{\infty} \\ &= \frac{1}{1+2\pi i k} \end{aligned}$$

Visualização da função $f(x)$



Visualização da parte real (azul) e imaginária (vermelha) da função $F(k)$



Transformada Discreta de Fourier

- Uma série $x_i = \{x_0, x_1, \dots, x_{N-1}\}$ de N amostras de um sinal, igualmente espaçadas ao longo do tempo, pode ser interpretada como uma função y_i periódica de período N
- Para isso, defina $y(j) = x_i$, onde j é um inteiro tal que $j = N * q + i$, e $y(t) = 0$, se t não é inteiro
- Contudo, ao invés de fazer esta adaptação e utilizar a transformada de Fourier, é melhor utilizar a Transformada Discreta de Fourier (DFT):

$$X_k = \sum_{n=0}^{N-1} x_n e^{-2\pi i k n / N}$$

- A Transformada Discreta Inversa de Fourier (IDFT) é dada por:

$$x_n = \frac{1}{N} \sum_{k=0}^{N-1} F_k e^{2\pi i k n / N}$$

Implementação da DFT e da IDFT em C++ em $O(N^2)$

```
1 #include <bits/stdc++.h>
2 #include <complex>
3
4 using namespace std;
5
6 const double PI { acos(-1.0) };
7
8 template<typename T>
9 vector<complex<T>> dft(const vector<T>& xs)
10 {
11     int N = (int) xs.size();
12     vector<complex<T>> F(N, 0);
13
14     for (int k = 0; k < N; ++k)
15         for (int i = 0; i < N; ++i)
16             F[k] += xs[i]*exp(complex<T>(0, -2*PI*i*k/N));
17
18     return F;
19 }
20
```

Implementação da DFT e da IDFT em C++ em $O(N^2)$

```
21 template<typename T>
22 vector<T> idft(const vector<complex<T>>& Fs)
23 {
24     int N = (int) Fs.size();
25     vector<T> f(N, 0);
26
27     for (int x = 0; x < N; ++x)
28         for (int k = 0; k < N; ++k)
29             f[x] += (1.0/N)*(Fs[k]*exp(complex<T>(0, 2*PI*x*k/N))).real();
30
31     return f;
32 }
```

Aplicação da DFT: Multiplicação de Polinômios

- A convolução entre duas funções $f(x)$ e $g(x)$ é uma função $h(x) = f(x) * g(x)$ que representa como a forma de uma função é modificada pela outra
- Ela é a integral do produto de ambas funções, sendo que uma delas é invertida e deslocada:

$$h(x) = f(x) * g(x) = \int_{-\infty}^{\infty} f(\tau)(t - \tau) d\tau$$

- A convolução discreta de f e g é dada por

$$(f * g)[n] = \sum_{m=-\infty}^{\infty} f[m]g[n - m]$$

- Se $f(x)$ e $g(x)$ são sequências de coeficientes de dois polinômios, a convolução de ambas será igual ao produto destes polinômios

Visualização da multiplicação de polinômios

$$f(x) = x^2 - 5x + 6$$

$$g(x) = x^3 - 2x^2 + 3x - 1$$

6	-5	1	
-1	3	-2	1

Visualização da multiplicação de polinômios

$$f(x) = x^2 - 5x + 6$$

$$g(x) = x^3 - 2x^2 + 3x - 1$$

$$h(x) = -6$$

			6	-5	1
1	-2	3	-1		

Visualização da multiplicação de polinômios

$$f(x) = x^2 - 5x + 6$$

$$g(x) = x^3 - 2x^2 + 3x - 1$$

$$h(x) = \mathbf{23}x - 6$$

		6	-5	1
1	-2	3	-1	

Visualização da multiplicação de polinômios

$$f(x) = x^2 - 5x + 6$$

$$g(x) = x^3 - 2x^2 + 3x - 1$$

$$h(x) = -28x^2 + 23x - 6$$

	6	-5	1
1	-2	3	-1

Visualização da multiplicação de polinômios

$$f(x) = x^2 - 5x + 6$$

$$g(x) = x^3 - 2x^2 + 3x - 1$$

$$h(x) = 19x^3 - 28x^2 + 23x - 6$$

6	-5	1	
1	-2	3	-1

Visualização da multiplicação de polinômios

$$f(x) = x^2 - 5x + 6$$

$$g(x) = x^3 - 2x^2 + 3x - 1$$

$$h(x) = -7x^4 + 19x^3 - 28x^2 + 23x - 6$$

6	-5	1		
	1	-2	3	-1

Visualização da multiplicação de polinômios

$$f(x) = x^2 - 5x + 6$$

$$g(x) = x^3 - 2x^2 + 3x - 1$$

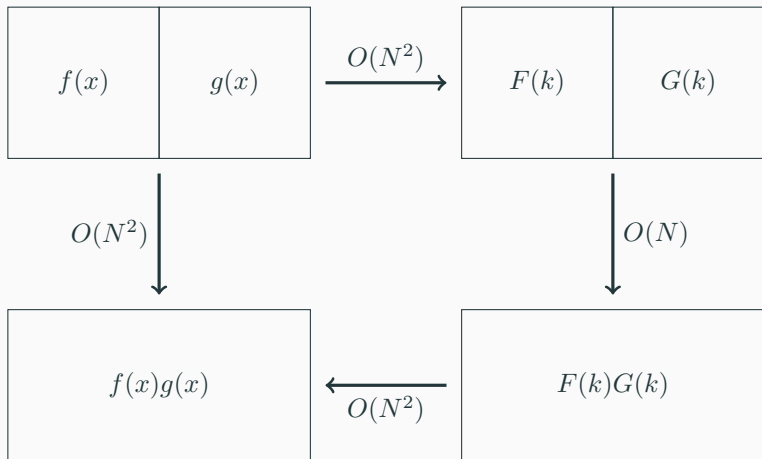
$$h(x) = x^5 - 7x^4 + 19x^3 - 28x^2 + 23x - 6$$

6	-5	1			
		1	-2	3	-1

Aplicação da DFT: Multiplicação de Polinômios

- Considere as transformadas $F(k)$ e $G(k)$ dos polinômios $f(x)$ e $g(x)$
- Pelo Teorema da Convolução, a transformada do produto será $H(k) = F(k)G(k)$, onde a multiplicação, neste caso, é termo a termo
- No domínio do tempo, onde estão os polinômios, a multiplicação polinomial é convolução, com complexidade $O(N^2)$, onde N é o maior dentre os graus
- No domínio das frequências, onde estão as transformadas, a convolução se torna uma multiplicação termo a termo, com complexidade $O(N)$
- Assim, é possível realizar a multiplicação de polinômios indiretamente, computando as transformadas $F(k)$ e $G(k)$, fazendo a multiplicação termo a termo, e computando a inversa de $H(k)$

Visualização da multiplicação indireta de polinômios



Implementação da multiplicação indireta de polinômios

```
34 vector<double>
35 operator*(const vector<double>& fx, const vector<double>& gx)
36 {
37     auto n = fx.size() - 1, m = gx.size() - 1;
38     vector<double> xs(n + m + 1), ys(n + m + 1);
39
40     copy(fx.begin(), fx.end(), xs.begin());
41     copy(gx.begin(), gx.end(), ys.begin());
42
43     auto Fk = dft(xs), Gk = dft(ys), Hk(Fk);
44
45     for (size_t i = 0; i < Hk.size(); ++i)
46         Hk[i] *= Gk[i];
47
48     return idft(Hk);
49 }
```

Transformada Rápida de Fourier

DFT em $O(N \log N)$

- A divisão e conquista pode ser aplicada no cálculo da DFT para reduzir sua complexidade assintótica
- Na etapa de divisão o sinal é dividido em duas partes de tamanhos aproximadamente iguais
- A conquista acontece quando o sinal tem uma única amostra: neste caso a transformada discreta coincide com a própria amostra
- A fusão permite o cálculo da DFT do sinal a partir das DFTs das duas partes
- Se a fusão for feita em $O(N)$, a recorrência se torna

$$f(N) = 2f(N/2) + (N)$$

- O Teorema Mestre nos diz que a complexidade da transformada passa a ser $O(N \log N)$
- Esta versão da DFT é denominada *Fast Fourier Transform* (FFT)

Decomposição do sinal FFT

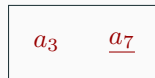
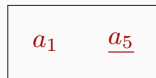
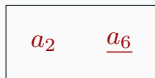
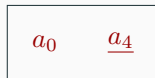
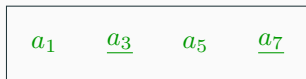
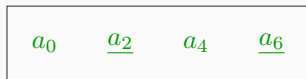
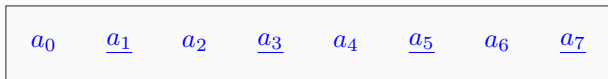
- Considere o sinal $(a_k) = a_0, a_1, \dots, a_{N-1}$
- Assuma, sem perda de generalidade, que $N = 2^t$, para algum t natural
- Se N não for uma potência de dois, basta adicionar um número suficiente de amostras $a_i = 0$ ao sinal até que N se torne uma potência de dois
- A etapa de divisão, também denominada decomposição do sinal, o sinal é separado em duas partes de tamanho $N/2$: as amostras cujos índices são pares (e_k) e as amostras cujos índices são ímpares (o_k)
- Assim,

$$(e_k) = a_0, a_2, a_4, \dots, a_{N-2}$$

e

$$(o_k) = a_1, a_3, a_5, \dots, a_{N-1}$$

Visualização da decomposição do sinal



Decomposição × ordenação

- Gerando a decomposição por meio da alocação de novos dois subvetores com as cópias dos elementos de índices pares e ímpares permite uma implementação *top-down* da FFT
- Para uma implementação *bottom-up*, é preciso entender o padrão subjacente que surge desta decomposição
- De fato, os elementos que ocupam as folhas nas árvores de decomposição tem índices que correspondem à ordenação dos números $\{0, 1, 2, \dots, N - 1\}$ usando como critério a inversão de sua representação binária
- Assim, por meio de um comparador customizado o este ordenação pode ser feita com complexidade $O(N \log N)$, o que não modifica a complexidade da FFT como um todo

Visualização da ordenação por padrão binário invertido

Índice	Padrão invertido	Padrão original
0	000	000
4	001	100
2	010	010
6	011	110
1	001	100
5	101	101
3	110	011
7	111	111

Implementação da ordenação por padrão binário

```
1 #include <bits/stdc++.h>
2
3 using namespace std;
4
5 int reversed(int x, int bits)
6 {
7     int res = 0;
8
9     for (int i = 0; i < bits; ++i)
10    {
11        res <<= 1;
12        res |= (x & 1);
13        x >>= 1;
14    }
15
16    return res;
17 }
18
```

Implementação da ordenação por padrão binário

```
19 template<typename T> vector<T> sortByBits(const vector<T>& xs)
20 {
21     int N = (int) xs.size(), bits = 1;
22
23     while ((1 << bits) != N)
24         ++bits;
25
26     vector<int> is(N);
27     iota(is.begin(), is.end(), 0);
28
29     sort(is.begin(), is.end(), [&bits](int x, int y) {
30         return reversed(x, bits) < reversed(y, bits);
31     });
32
33     vector<T> ans(N);
34
35     for (int i = 0; i < N; ++i)
36         ans[i] = xs[is[i]];
37
38     return ans;
39 }
```

- A etapa de conquista acontece em sinais como uma única amostra
- Aplicando o valor $N = 1$ na transformada discreta obtêm-se

$$X_0 = \sum_{n=0}^{N-1} x_n e^{-2\pi i k n / N} = \sum_{n=0}^0 x_0 e^{-2\pi i k n} = x_0$$

- Assim, a própria amostra corresponde à sua transformada
- É preciso atentar, contudo, que embora numericamente iguais, X_0 reside no domínio das frequências, enquanto que x_0 está no domínio do tempo
- Portanto esta etapa tem complexidade $O(1)$

Fusão (Síntese)

- A última etapa consiste em combinar as transformadas das duas partes (pares e ímpares) na transformada do sinal
- Lembrando que a Transformada de Fourier é Linear, a transformada de um sinal x_k pode ser computada como a soma de dois sinais distintos cuja soma resulte em x_k
- Considere os sinais e_k e o_k dados por

$$e_i = \begin{cases} x_i, & \text{se } i \text{ é par} \\ 0, & \text{caso contrário} \end{cases}$$

e

$$o_j = \begin{cases} x_j, & \text{se } j \text{ é ímpar} \\ 0, & \text{caso contrário} \end{cases}$$

Fusão (Síntese)

- Deste modo, $x_k = e_k + o_k$
- As transformadas de e_k e o_k tem um comportamento peculiar
- A transformada de e_k duplica seus resultados
- A transformada de o_k tem mesmo comportamento, porém com os valores multiplicados por uma componente sinusoidal
- Isto porque, em relação à x_k , o sinal o_k está deslocado no tempo em uma unidade
- Deslocar no tempo corresponde a convolução do sinal com uma função $\delta(t - a)$, onde a é o deslocamento
- A transformada da função $\delta(t - a)$ é uma exponencial complexa:

$$\mathcal{F}[\delta(t - a)] = \int_{-\infty}^{\infty} \delta(t - a) e^{-2\pi k i t} dt = e^{-2\pi k a i}$$

Visualização dos sinais no domínio do tempo

x_k	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8
-------	-------	-------	-------	-------	-------	-------	-------	-------

e_k	x_0	0	x_2	0	x_4	0	x_6	0
-------	-------	---	-------	---	-------	---	-------	---

o_k	0	x_1	0	x_3	0	x_5	0	x_7
-------	---	-------	---	-------	---	-------	---	-------

Visualização das transformadas no domínio das frequências

X_k	X_1	X_2	X_3	X_4	X_5	X_6	X_7	X_8
-------	-------	-------	-------	-------	-------	-------	-------	-------

E_k	E_1	E_2	E_3	E_4	E_1	E_2	E_3	E_4
-------	-------	-------	-------	-------	-------	-------	-------	-------

O_k	s_1O_1	s_2O_2	s_3O_3	s_4O_4	s_1O_1	s_2O_2	s_3O_3	s_4O_4
-------	----------	----------	----------	----------	----------	----------	----------	----------

Padrão borboleta

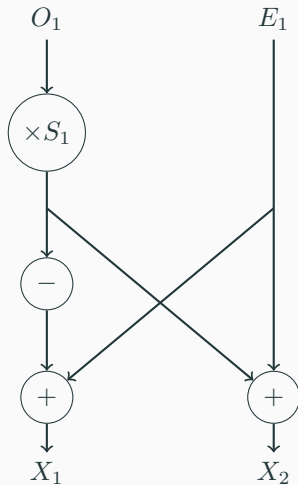
- Como as transformadas das duas partes foram computadas sem o deslocamento, é preciso compensá-lo na composição da transformada do todo
- As componentes oriundas da parte par E_k são somadas sem alteração em cada componente da transformada X_k
- Já as componentes de O_k devem ser multiplicadas pela componente sinusoidal

$$S_k = e^{\frac{2\pi k i}{N}}$$

antes de serem somadas

- Esta multiplicação afeta o sinal do termo: a primeira metade terá sinal negativo, e a segunda metade sinal positivo
- Este ajuste é denominado padrão borboleta, por conta da visualização do diagrama gerado

Padrão borboleta para $N = 2$



Implementação

Implementação recursiva *top-down*

- Uma forma de implementar a FFT é por meio de recursão
- A cada etapa, são criados dois subvetores e_k e o_k , de tamanho $N/2$, e a transformada é chamada em cada um destes subvetores
- O caso base acontece quando $N = 1$, onde a transformada é igual à própria amostra
- Na etapa de síntese ou fusão, os coeficientes da transformada X_k podem ser computado através das transformadas E_k e O_k dos subvetores:

$$X_j = \begin{cases} E_j + S_j O_j, & \text{se } 0 \leq j < N/2, \\ E_{j-N/2} - S_{j-N/2} O_{j-N/2}, & \text{caso contrário} \end{cases}$$

onde

$$S_j = e^{-2\pi j i}$$

Implementação recursiva *top-down*

- É possível implementar tanto a transformada quanto a inversa em uma função
- Primeiramente, é preciso uniformizar o tipo dos vetores
- Uma opção é que todos sejam vetores de complexos
- Em segundo lugar, são duas as diferenças entre a transformada e sua inversa:
 1. os sinais dos ângulos são opostos
 2. na inversa os coeficientes são divididos por N
- Uma *flag* booleana pode ser passada como parâmetro para decidir o sentido da transformada
- Como N é uma potência de 2, uma forma de realizar esta divisão por N é dividir, a cada etapa, os coeficientes por 2

Implementação *top-down* da FFT

```
1 #include <bits/stdc++.h>
2
3 using namespace std;
4
5 const double PI { acos(-1.0) };
6
7 void fft(vector<complex<double>>& xs, bool invert = false)
8 {
9     int N = (int) xs.size();
10
11     if (N == 1)
12         return;
13
14     vector<complex<double>> es(N/2), os(N/2);
15
16     for (int i = 0; i < N/2; ++i)
17         es[i] = xs[2*i];
18
19     for (int i = 0; i < N/2; ++i)
20         os[i] = xs[2*i + 1];
21
```

Implementação *top-down* da FFT

```
22     fft(es, invert);
23     fft(os, invert);
24
25     auto signal = (invert ? 1 : -1);
26     auto theta = 2 * signal * PI / N;
27     complex<double> S { 1 }, S1 { cos(theta), sin(theta) };
28
29     for (int i = 0; i < N/2; ++i)
30     {
31         xs[i] = (es[i] + S * os[i]);
32         xs[i] /= (invert ? 2 : 1);
33
34         xs[i + N/2] = (es[i] - S * os[i]);
35         xs[i + N/2] /= (invert ? 2 : 1);
36
37         S *= S1;
38     }
39 }
```

Implementação *bottom-up*, *in-place*

- Utilizando a ordenação por *bits* invertidos, é possível implementar a FFT *bottom-up*
- Além de dispensar a recursão, o custo de memória é reduzido, pois só é preciso copiar dois coeficientes a cada atualização, sem precisar copiar os subvetores a cada iteração
- A ordenação baseada em *bits* pode ser feita em $O(N)$: basta trocar de posição dos elementos de índices i e j tais que $i < j$ e que i e j são mutuamente reversos

Implementação *bottom-up, in-place* da FFT

```
1 #include <bits/stdc++.h>
2
3 using namespace std;
4
5 const double PI { acos(-1.0) };
6
7 int reversed(int x, int bits)
8 {
9     int res = 0;
10
11     for (int i = 0; i < bits; ++i)
12     {
13         res <<= 1;
14         res |= (x & 1);
15         x >>= 1;
16     }
17
18     return res;
19 }
20
```

Implementação *bottom-up, in-place* da FFT

```
21 void fft(vector<complex<double>>& xs, bool invert = false)
22 {
23     int N = (int) xs.size();
24
25     if (N == 1)
26         return;
27
28     int bits = 1;
29
30     while ((1 << bits) != N)
31         ++bits;
32
33     for (int i = 0; i < N; ++i)
34     {
35         auto j = reversed(i, bits);
36
37         if (i < j)
38             swap(xs[i], xs[j]);
39     }
40
```

Implementação *bottom-up, in-place* da FFT

```
41  for (int size = 2; size <= N; size *= 2)
42  {
43      auto signal = (invert ? 1 : -1);
44      auto theta = 2 * signal * PI / size;
45      complex<double> S1 { cos(theta), sin(theta) };
46
47      for (int i = 0; i < N; i += size)
48      {
49          complex<double> S { 1 }, k { invert ? 2.0 : 1.0 };
50
51          for (int j = 0; j < size / 2; ++j)
52          {
53              auto a { xs[i + j] }, b { xs[i + j + size/2] * S };
54
55              xs[i + j] = (a + b) / k;
56              xs[i + j + size/2] = (a - b) / k;
57              S *= S1;
58          }
59      }
60  }
61 }
```

Referências

1. **CHEEVER**, Erick. [The Fourier Series](#), acesso em 12/08/2020.
2. CP Algorithms. [Fast Fourier Transform](#), acesso em 13/08/2020.
3. **SMITH**, Steven W. [The Scientist and Engineer's Guide to Digital Signal Processing](#), acesso em 17/08/2020.
4. Stanford. [Lecture 11 – The Fourier Transform](#), acesso em 13/08/2020.
5. The Fourier Transform.com. [The Dirac-Delta Function – The Impulse](#), acesso em 18/08/2020.
6. Wikipédia. [Discrete Fourier Transform](#), acesso em 13/08/2020.
7. Wolfram. [Fourier Series](#), acesso em 12/08/2020.
8. Wolfram. [Fourier Transform](#), acesso em 13/08/2020.