

# Processamento Digital de Sinais - 2021.2

Ativando o ambiente de desenvolvimento do projeto para importar os pacotes instalados:

```

· begin
·   import Pkg
·   Pkg.activate(".")
· end

```

Activating project at 'C:\Users\hgtll\Projects\ecom063-pds' [?](#)

```

· using Random, Distributions, FFTW, Plots
· plotlyjs();

```

## Implementação DFT e iDFT

Começamos definindo o sinal senoidal, que será amostrado no tempo, dado por:

$$s(t) = 0.7\sin(2\pi 50t) + \sin(2\pi 120t)$$

Além disso podemos supor que o sinal  $S$  sofre uma perturbação pela ação de um ruído de amplitude  $A$  que segue uma distribuição normal:

$$x(t) = s(t) + An(t) \quad , \quad n(t) \sim \mathcal{N}(0, 1)$$

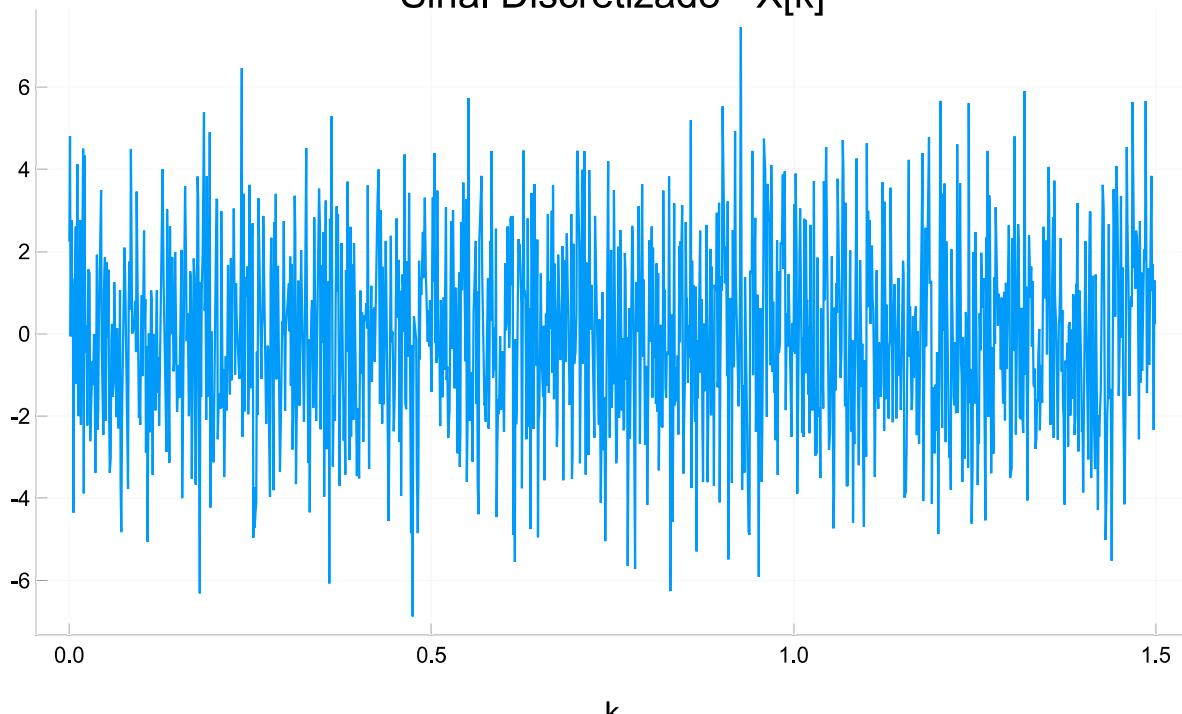
```

· begin
·   j = im # 😊
·   Fs=1000 # Frequência de amostragem
·   T=1/Fs # Período de amostragem
·   L=1500 # Tamanho do sinal
·   t=0:T:(L-1)*T # Vetor de tempo
·   A = 2 # Amplitude do ruído
·
·   S = 0.7*sin.(2π*50*t) + sin.(2π*120*t)
·   X = S + A*rand(Normal(), size(S))
· end;

```

Plotando o sinal gerado:

## Sinal Discretizado - X[k]



```

begin
    plot(t, X, label=nothing)
    title!("Sinal Discretizado - X[k]")
    xlabel!("k")
end

```

A transformada de fourier discreta do sinal  $x[k]$  é dada por:

$$X[k] = \sum_{n=0}^{N-1} x[n] e^{-j \frac{2\pi}{N} kn}, \quad k = 0, 1, \dots, N-1$$

Se definirmos  $\omega_N = e^{-2\pi j/N}$  a equacão anterior pode ser secreta de forma matricial como:

$$X = \begin{bmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & \omega_N & \omega_N^2 & \dots & \omega_N^{(n-1)} \\ 1 & \omega_N^2 & \omega_N^4 & \dots & \omega_N^{2(n-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega_N^{(n-1)} & \omega_N^{2(n-1)} & \dots & \omega_N^{(n-1)^2} \end{bmatrix} \begin{bmatrix} x[0] \\ x[1] \\ x[2] \\ \vdots \\ x[n-1] \end{bmatrix}$$

O que mostra que a DFT pode ser calculada através da multiplicacao do vetor  $x$  por uma matriz densa  $n \times n$ , requerendo  $\mathcal{O}(n^2)$  operaçoes.

Implementando a DFT:

```
dft (generic function with 1 method)
· function dft(x)
·   N = length(x)
·   Ω = Array{ComplexF64}(undef, N, N)
·   ωn = e-j*(2π/N)
·   for i ∈ 1:N
·     for j ∈ 1:N
·       Ω[i, j] = ωn((i-1)*(j-1))
·     end
·   end
·   return Ω * x
· end
```

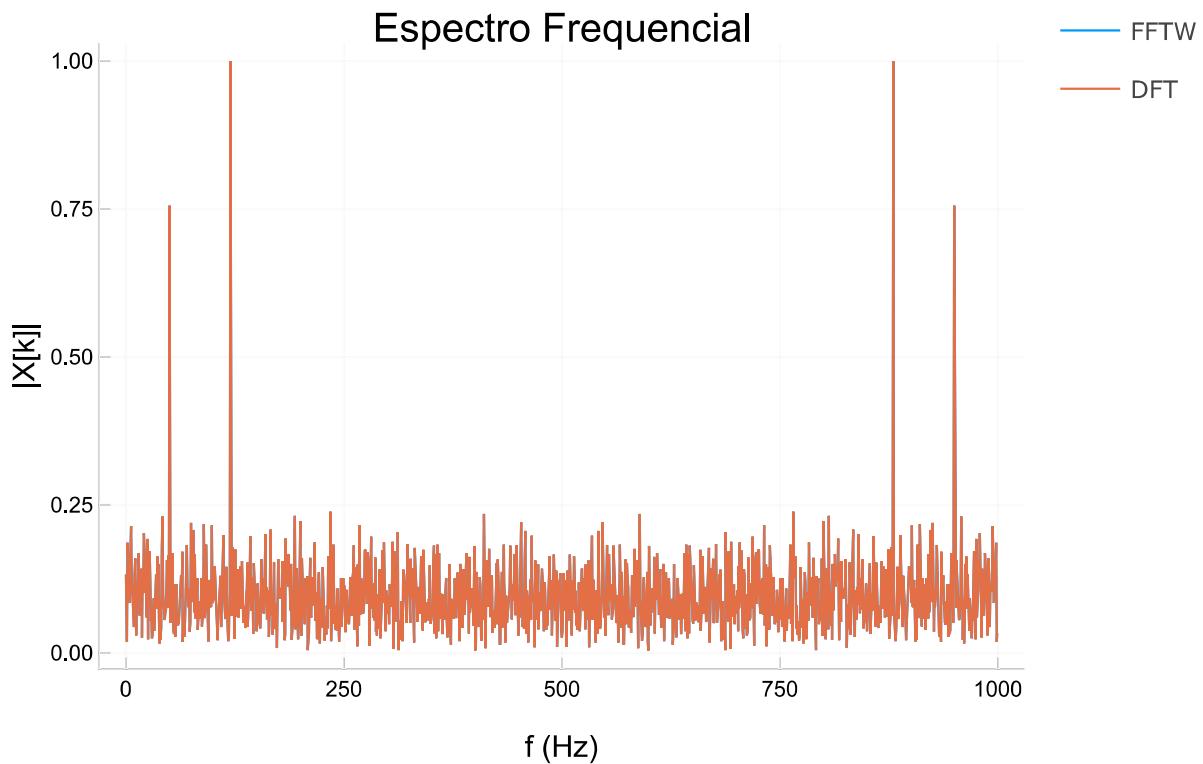
Em seguida, calculamos a DFT para o sinal amostrado e comparamos com a implementação disponível na biblioteca FFTW para validar o resultado:

```
dft_amp (generic function with 1 method)
```

```
· function dft_amp(X, foo)
·   Y = foo(X)
·   return 2*abs.(Y/L)
· end
```

```
plot_magnitude (generic function with 2 methods)
```

```
· function plot_magnitude(X, foo, foo_label, base=nothing)
·
·   if base != nothing
·     X = pad_zeros(X, base)
·   end
·
·   L = length(X)
·
·   Y1 = dft_amp(X, fft)
·   Y2 = dft_amp(X, foo)
·
·   f = Fs*(0:L-1)/L;
·   plot(f, Y1, label="FFTW")
·   plot!(f, Y2, label=foo_label)
·   title!("Espectro Frequencial")
·   xlabel!("f (Hz)")
·   ylabel!("|X[k]|")
· end
```

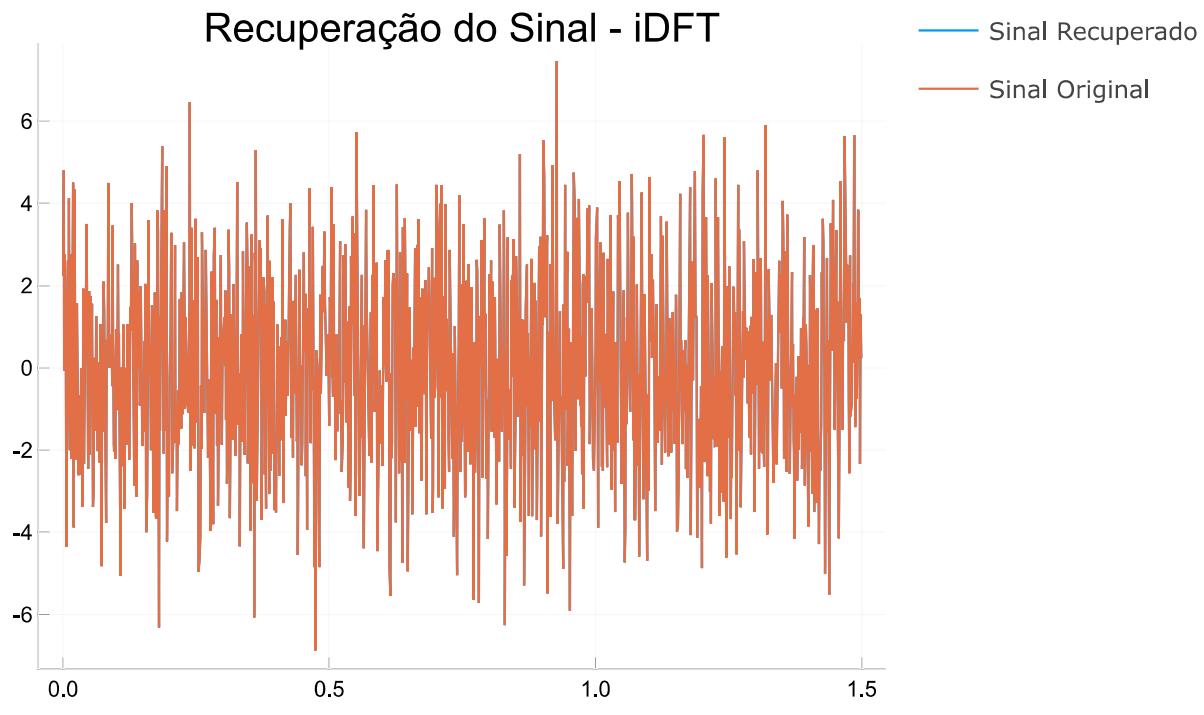


Para recuperar o vetor do sinal discretizado, utilizamos a transformada inversa de fourier discreta, dada por:

$$x[k] = \frac{1}{N} \sum_{n=0}^{N-1} X[n] e^{j \frac{2\pi}{N} kn} \quad , k = 0, 1, \dots, N - 1$$

```
idft (generic function with 1 method)
· function idft(x)
·     N = length(x)
·     Ω = Array{ComplexF64}(undef, N, N)
·     ω_n = e^(j*(2π/N))
·     for i ∈ 1:N
·         for j ∈ 1:N
·             Ω[i, j] = ω_n^((i-1)*(j-1))
·         end
·     end
·     return (Ω * x) ./ N
· end
```

Validando a implementação:



```

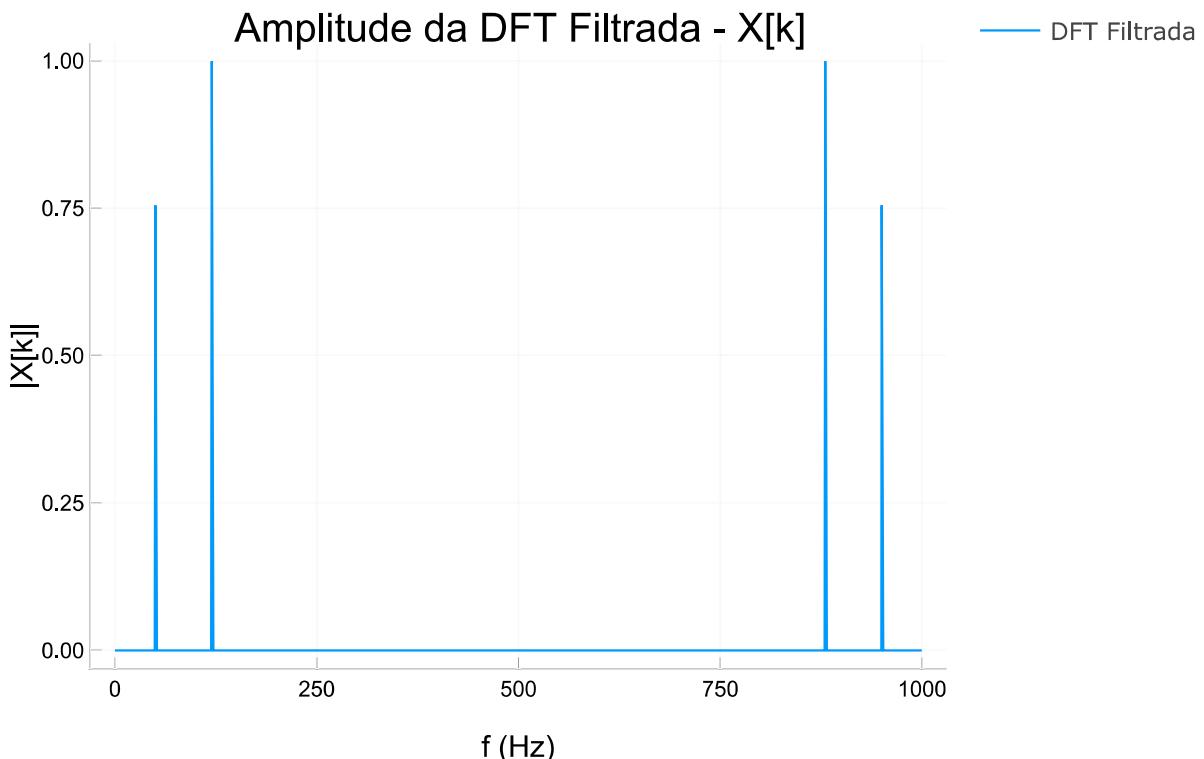
begin
    Y = dft(X)
    _X = idft(Y)

    plot(t, real(_X), label="Sinal Recuperado")
    plot!(t, X, label="Sinal Original")
    title!("Recuperação do Sinal - iDFT")
    xlabel!("k")
end

```

## Implementando um filtro simples

Se desejamos filtrar, digamos apenas a senoide de 50Hz podemos excluir com uma máscara o intervalo que contém amplitudes maiores que 0.7 no grafico do modulo do espectro e recuperar o sinal:

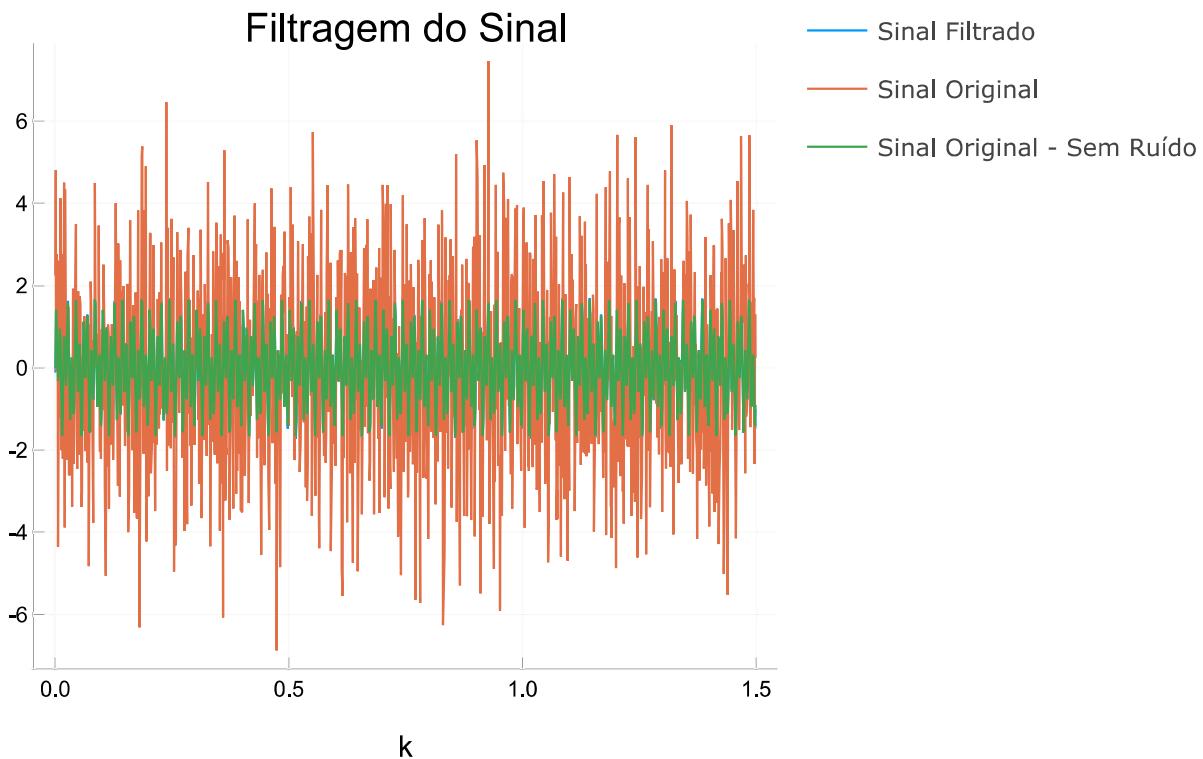


```

begin
    ε = 0.001
    index = findall(
        function compare(value)
            value >= 0.5 - ε
        end, dft_amp(X, dft)
    )
    mask = zeros(L)
    mask[index] .= 1.
    plot(Fs*(0:L-1)/L, dft_amp(X, dft) .* mask, label="DFT Filtrada")
    title!("Amplitude da DFT Filtrada - X[k]")
    xlabel!("f (Hz)")
    ylabel!("|X[k]|")
end

```

Recuperando o sinal filtrado:



```

begin
    Y_f = mask .* dft(X)
    plot(t, real.(idft(Y_f)), label="Sinal Filtrado")
    plot!(t, X, label="Sinal Original")
    plot!(t, S, label="Sinal Original - Sem Ruído")
    title!("Filtragem do Sinal")
    xlabel!("k")
end

```

## Implementação da FFT - Radix 2 e Radix 3

Apesar de simples, a implementação da DFT é custosa quando computada matricialmente como podemos observar comparando a execução com a FFT para a mesma entrada:

```
time_dft = 1.1773216
begin
    time_dft = @elapsed begin
        dft(X)
    end
end
```

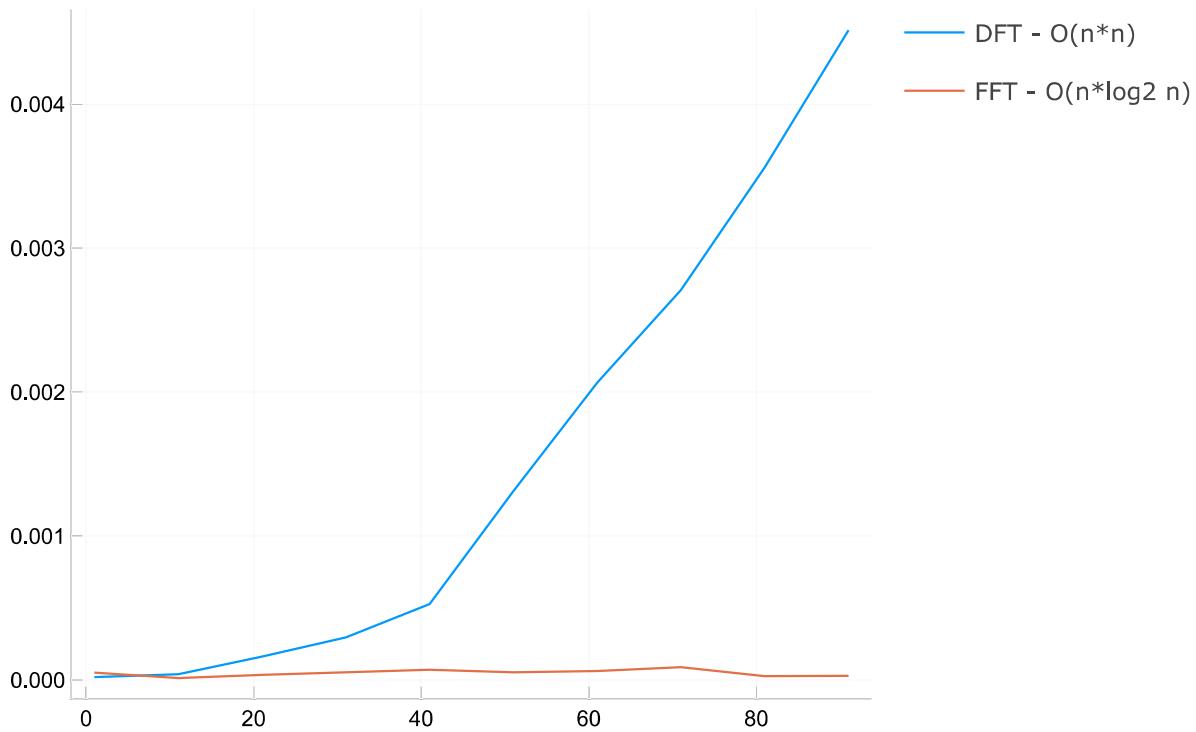
```
time_fft = 0.0001319
begin
    time_fft = @elapsed begin
        fft(X)
    end
end
```

Calculando a razão entre os tempos, podemos perceber o quanto a FFT chega a ser mais eficiente:

8925.865049279757

```
time_dft / time_fft
```

Para vetores X de tamanho crescente também podemos visualizar as curvas de desempenho:



```

begin
    sizes = 1:10:100
    function pad_zeros(x, b=2)
        N = length(x)
        pad = b^Int(ceil(log(b, N)))
        return vcat(x,zeros(pad - N))
    end
    function elapsed_time(size, foo, pad=nothing)
        if pad != nothing
            x = pad_zeros(rand(size), pad)
        else
            x = rand(size)
        end
        dt = @elapsed begin
            foo(x)
        end
    end
    times_fft = [elapsed_time(Int(s), fft) for s in sizes]
    times_dft = [elapsed_time(Int(s), dft) for s in sizes]
    plot(sizes, times_dft, label="DFT - O(n*n)")
    plot!(sizes, times_fft, label="FFT - O(n*log2 n)")
end

```

Para implementar a transformada de fourier rápida com decimação no tempo (Radix 2 FFT) observamos que a expressão da DFT pode ser reescrita da seguinte maneira:

$$X[k] = E_k + e^{-j\frac{2\pi}{N}k} O_k$$

$$X[k + \frac{N}{2}] = E_k - e^{-j\frac{2\pi}{N}k} O_k$$

Onde  $E_k$  e  $O_k$  são DFTs de tamanho  $N/2$  relativas aos indices pares e ímpares do vetor original  $x$ .

```

dit2fft (generic function with 1 method)
· function dit2fft(x)
·   N = length(x)

·
·   if N == 1
·     return x
·   else
·     odd_index = Int.(1:2:N)
·     X = Array{ComplexF64}(undef, N)
·     X[1:N÷2] = dit2fft(x[odd_index .+ 1])
·     X[N÷2+1:N] = dit2fft(x[odd_index])
·   end

·
·   for k ∈ 0:(N÷2)-1
·     p = X[k+1]
·     q = e^( -j * (2π/N) * k ) * X[k+1 + (N÷2)]
·     X[k+1] = p + q
·     X[k+1 + (N÷2)] = p - q
·   end

·
·   return X
· end

```

De maneira similar, podemos mostrar que para a transformada rápida de fourier (Radix 3 FFT) podemos escrever a DFT da seguinte maneira:

$$X[k] = A_k + \omega_N^k B_k + \omega_N^{2k} C_k$$

$$X\left[k + \frac{N}{3}\right] = A_k + e^{-j\frac{2\pi}{3}} \omega_N^k B_k + e^{-j\frac{4\pi}{3}} \omega_N^{2k} C_k$$

$$X\left[k + \frac{2N}{3}\right] = A_k + e^{-j\frac{4\pi}{3}} \omega_N^k B_k + e^{-j\frac{2\pi}{3}} \omega_N^{2k} C_k$$

Onde  $A_k$ ,  $B_k$  e  $C_k$  são DFTs de tamanho  $N/3$ .

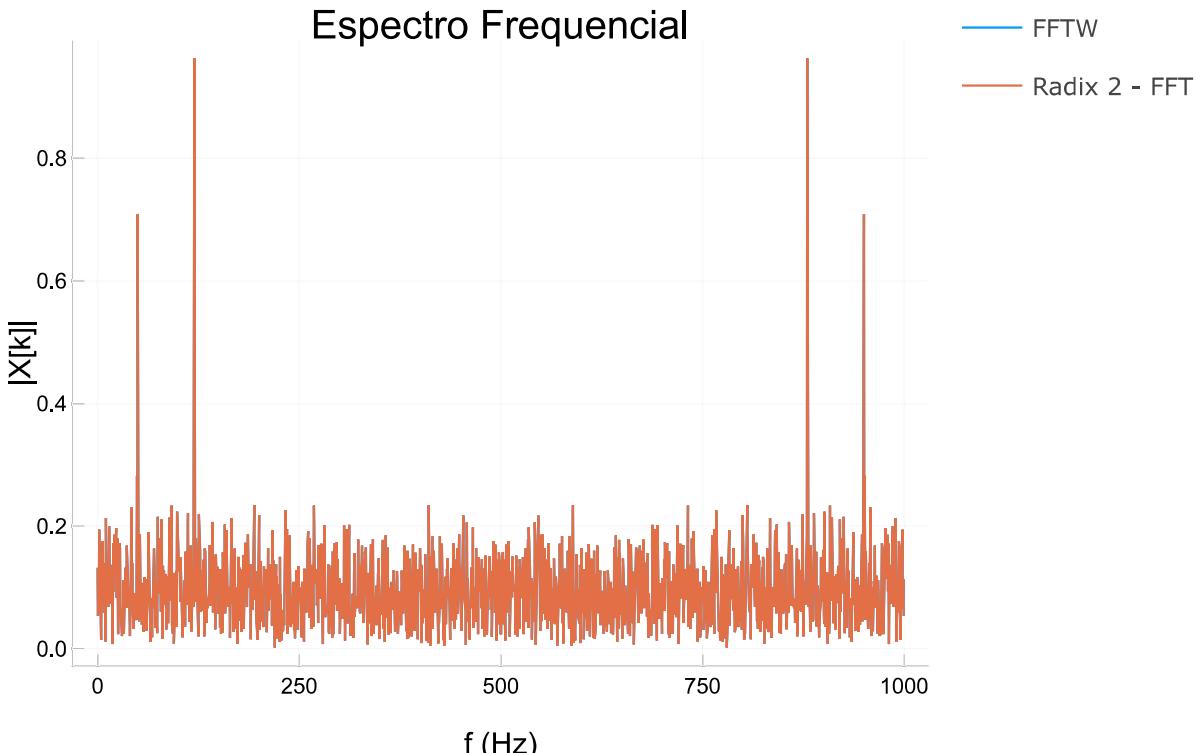
```
dit3fft (generic function with 1 method)
begin
    ω(n, N) = e^(-j*(2π/N)*n)
    function dit3fft(x)
        N = length(x)

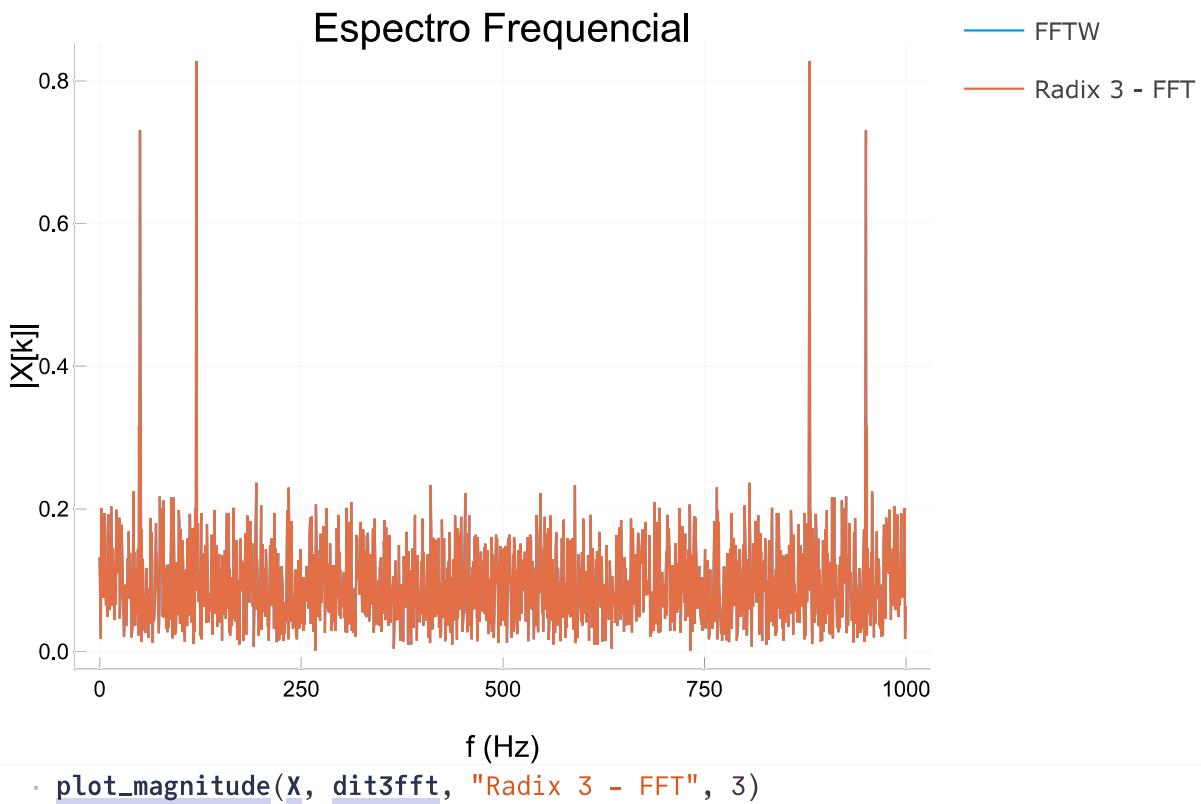
        if N == 1
            return x
        else
            index = Int.(1:3:N)
            X = Array{ComplexF64}(undef, N)
            X[1:N÷3] = dit3fft(x[index])
            X[N÷3+1:2*(N÷3)] = dit3fft(x[index .+ 1])
            X[2*(N÷3)+1:N] = dit3fft(x[index .+ 2])
        end

        for k ∈ 0:(N÷3)-1
            A = X[k+1]
            B = ω(k, N) * X[k+1 + (N÷3)]
            C = ω(2k, N) * X[k+1 + 2*(N÷3)]
            X[k+1] = A + B + C
            X[k+1 + (N÷3)] = A + ω(1, 3) * B + ω(2, 3) * C
            X[k+1 + 2*(N÷3)] = A + ω(2, 3) * B + ω(1, 3) * C
        end

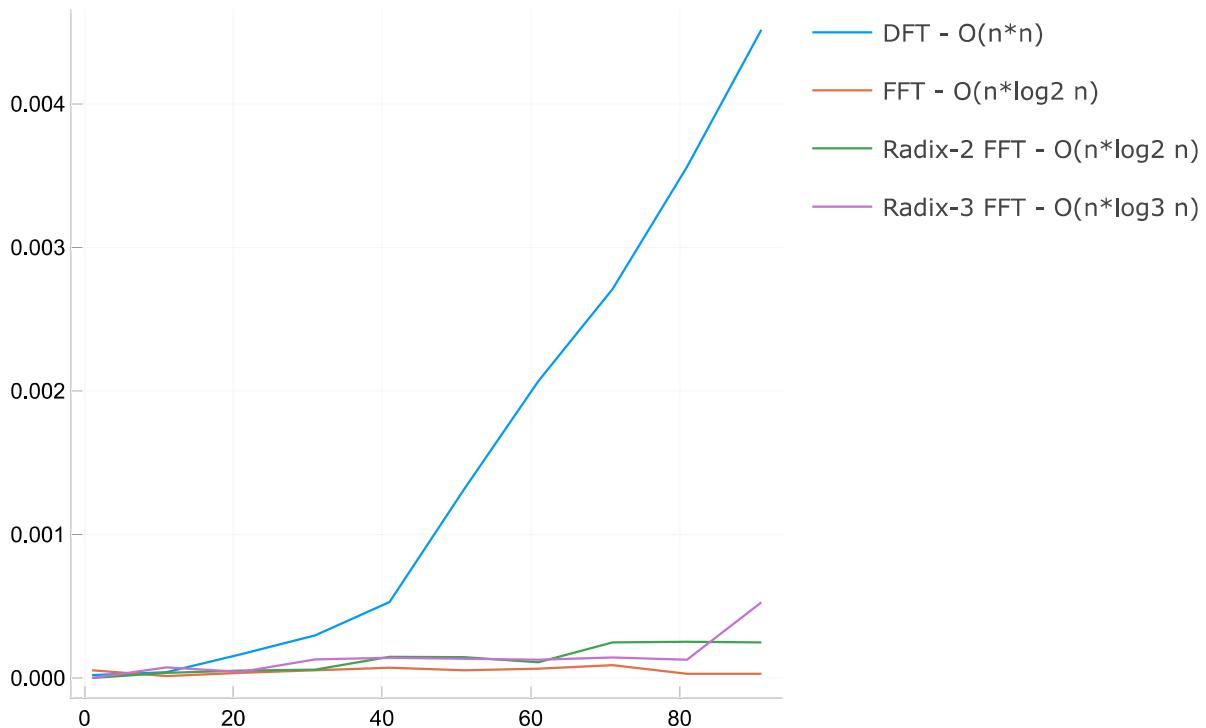
        return X
    end
end
```

Validando mais uma vez a implementação:





Comparando as curvas de desempenho:



```

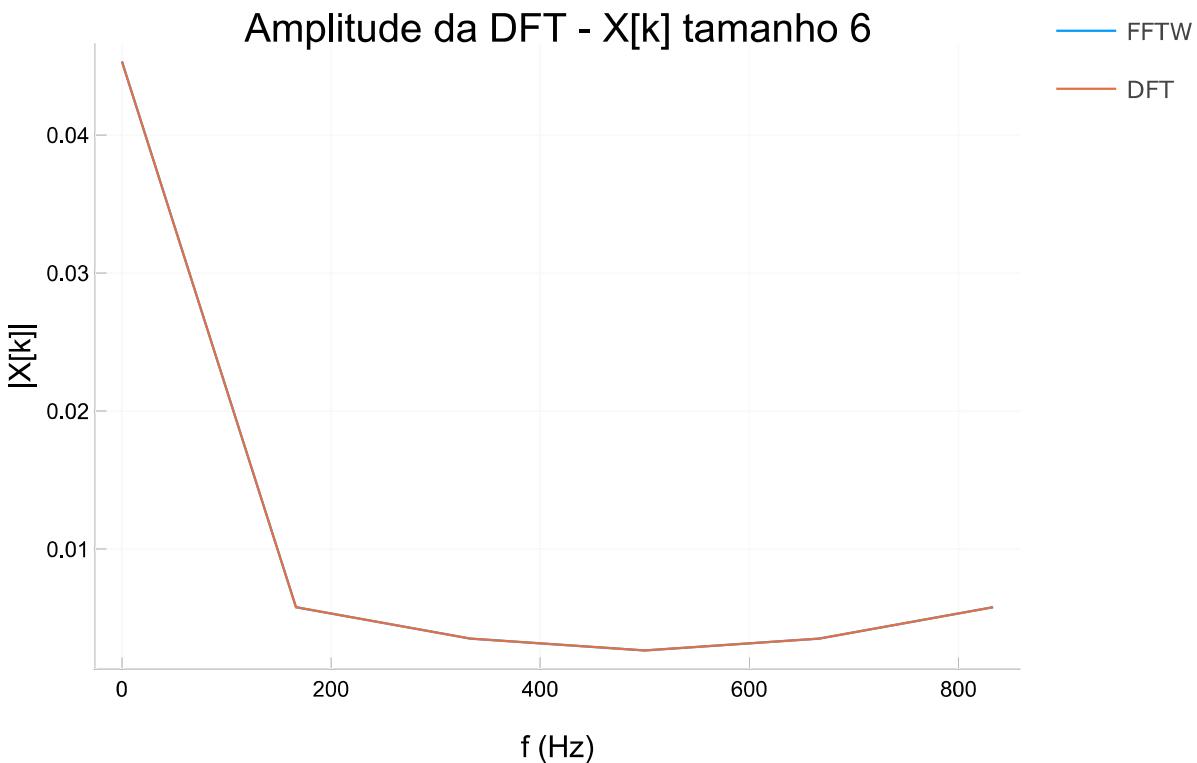
begin
    times_dit2fft = [elapsed_time(Int(s), dit2fft, 2) for s in 1:10:100]
    times_dit3fft = [elapsed_time(Int(s), dit3fft, 3) for s in 1:10:100]
    plot(sizes, times_dft, label="DFT -  $O(n \cdot n)$ ")
    plot!(sizes, times_fft, label="FFT -  $O(n \cdot \log_2 n)$ ")
    plot!(sizes, times_dit2fft, label="Radix-2 FFT -  $O(n \cdot \log_2 n)$ ")
    plot!(sizes, times_dit3fft, label="Radix-3 FFT -  $O(n \cdot \log_3 n)$ ")
end

```

# Atividade 01 - AB1

1. Considere a sequência  $x[n] = [6 \ 8 \ 5 \ 4 \ 5 \ 6]$ . Implemente o algoritmo da **Transformada de Fourier Discreta (DFT)**, para 6, 8 e 32 pontos e analise o espectro frequencial desse sinal, validando os resultados com uma função `fft` já implementada. Implemente também a **Transformada Discreta Inversa de Fourier (IDFT)** para restaurar a sequência original.
2. Implemente o algoritmo de **raiz de 2 (Radix-2)** e de **raiz de 3 (Radix-3)**, com decimação no tempo, da **Transformada Rápida de Fourier (FFT)** para analisar o espectro frequencial do sinal da Atividade 1. Valide os resultados com uma função `fft` já implementada.

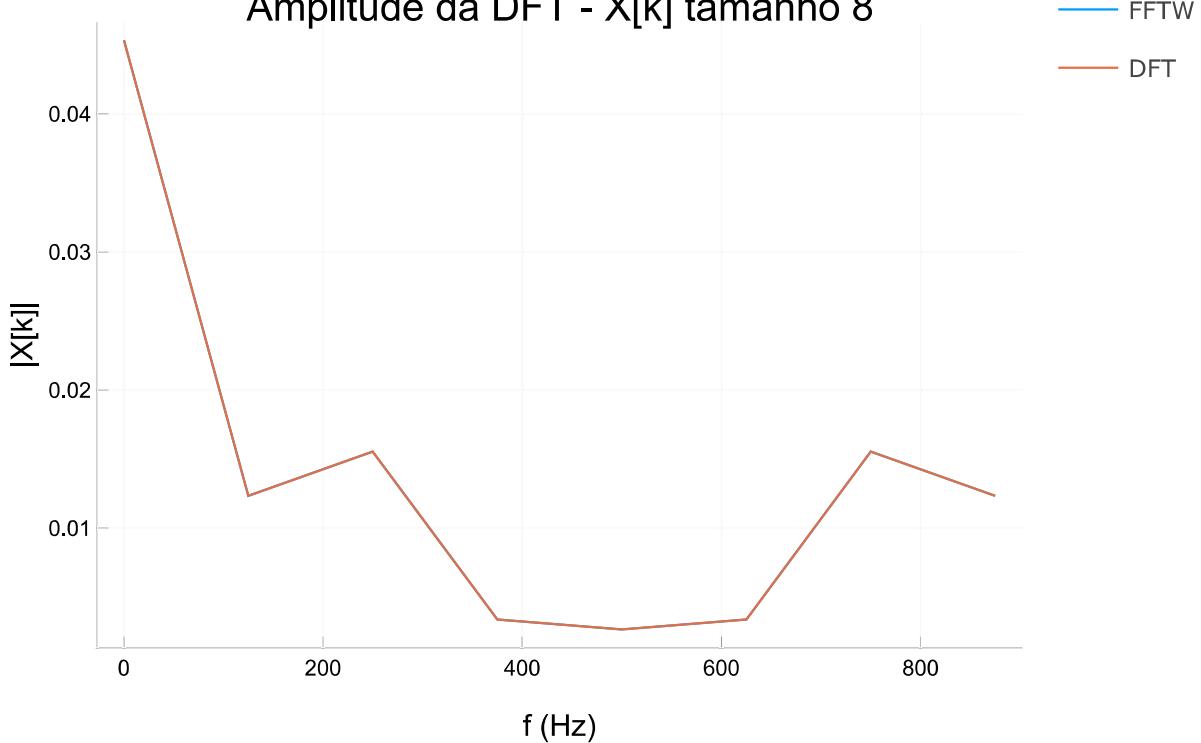
```
x = [6, 8, 5, 4, 5, 6]
· x = [6, 8, 5, 4, 5, 6]
```



```
begin
·   xL = length(x)
·   x_fft = dft_amp(x, fft)
·   x_dft = dft_amp(x, dft)

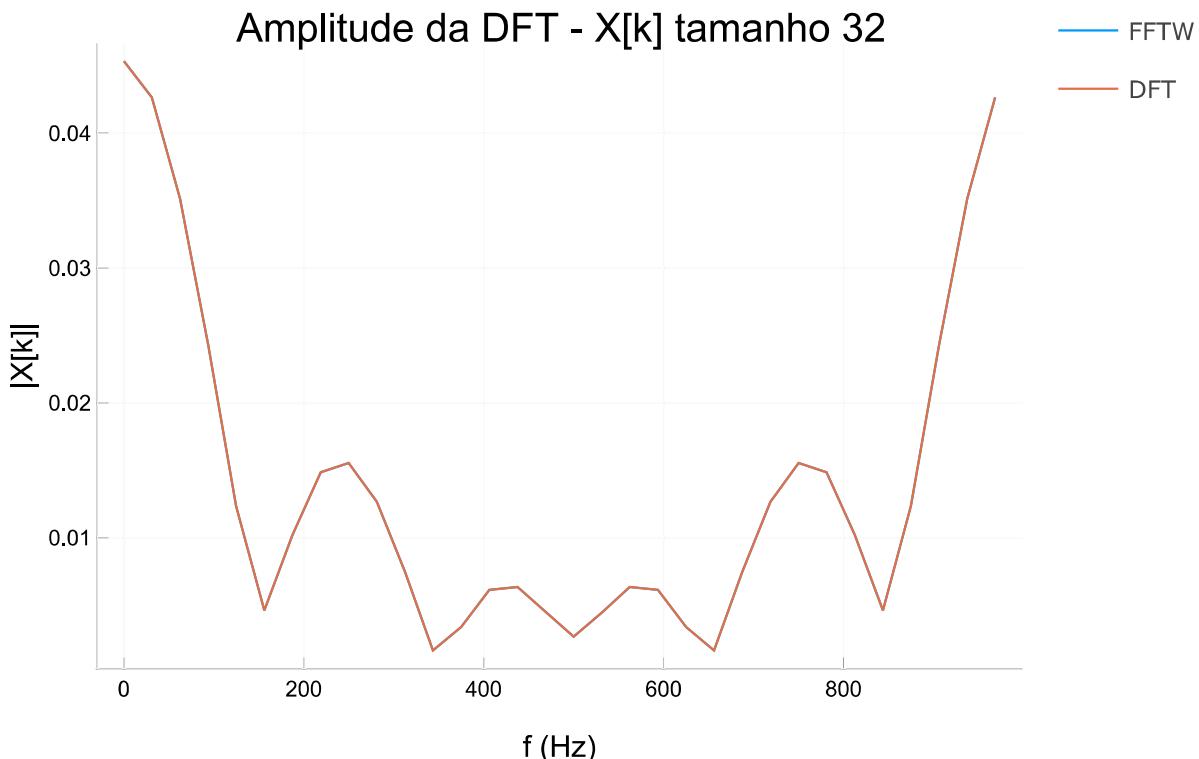
·
·   xf = Fs*(0:xL-1)/xL;
·   plot(xf, x_fft, label="FFTW")
·   plot!(xf, x_dft, label="DFT")
·   title!("Amplitude da DFT - X[k] tamanho 6")
·   xlabel!("f (Hz)")
·   ylabel!("|X[k]|")
· end
```

## Amplitude da DFT - X[k] tamanho 8



```
begin
    x8 = pad_zeros(x)
    x8L = length(x8)
    x8_fft = dft_amp(x8, fft)
    x8_dft = dft_amp(x8, dft)

    x8f = Fs*(0:x8L-1)/x8L;
    plot(x8f, x8_fft, label="FFTW")
    plot!(x8f, x8_dft, label="DFT")
    title!("Amplitude da DFT - X[k] tamanho 8")
    xlabel!("f (Hz)")
    ylabel!("|X[k]|")
end
```



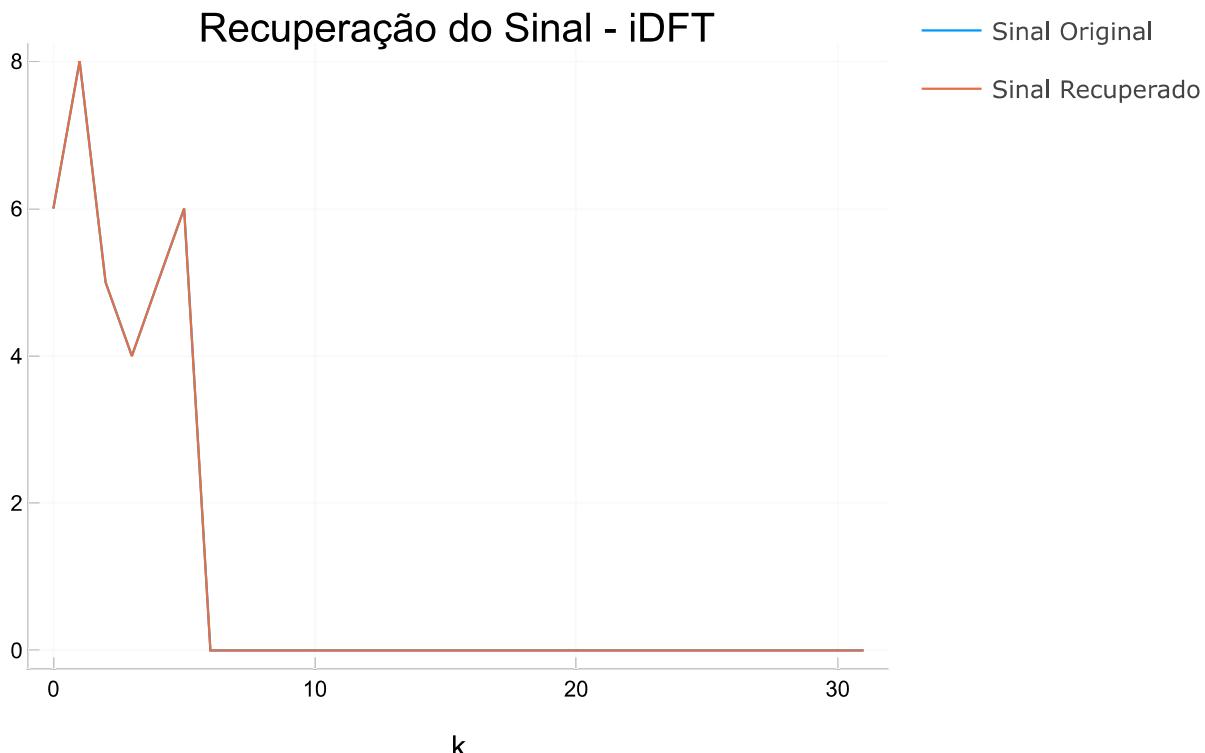
```

begin
    x16 = pad_zeros(vcat(x8, [0]))
    x32 = pad_zeros(vcat(x16, [0]))
    x32L = length(x32)
    x32_fft = dft_amp(x32, fft)
    x32_dft = dft_amp(x32, dft)

    x32f = Fs*(0:x32L-1)/x32L;
    plot(x32f, x32_fft, label="FFTW")
    plot!(x32f, x32_dft, label="DFT")
    title!("Amplitude da DFT - X[k] tamanho 32")
    xlabel!("f (Hz)")
    ylabel!("|X[k]|")
end

```

Recuperando o sinal original:



```

begin
    y32 = dft(x32)
    _x32 = idft(y32)

    plot(0:length(x32)-1, x32, label="Sinal Original")
    plot!(0:length(x32)-1, real.(_x32), label="Sinal Recuperado")
    title!("Recuperação do Sinal - iDFT")
    xlabel!("k")
end

```

## Atividade 2 - AB1

1. Considere as amostras do sinal  $x[n]$  descritas no arquivo texto `xn.txt`. Considerando que uma frequência de amostragem de 1000 Hz foi utilizada para gerar esta sequência, implemente o algoritmo **FFT-Radix2** e analise o espectro frequencial deste sinal.
2. Utilize um filtro ideal do tipo passa baixa e com frequência de corte  $\omega_c = 18$  Hz para eliminar componentes freqüenciais indesejadas de  $x[n]$ .
3. Implemente a **IFFT** para restaurar o sinal filtrado,  $x_f[n]$ , e compare com o sinal original.

Lendo o arquivo de texto e convertendo para representação em ponto flutuante:

```

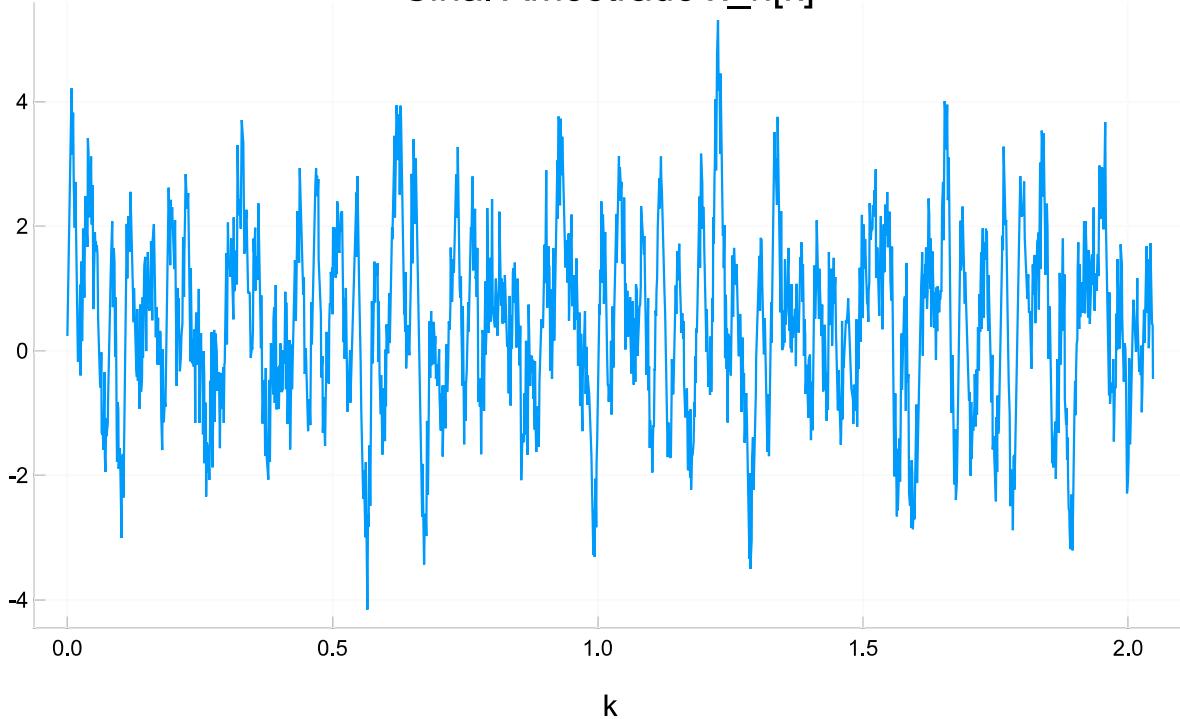
xn_txt =
" 2.3421087e-01  7.2789166e-01  1.5545091e+00  2.0164834e+00  2.2026883e+00  3.09e
<   xn_txt = open(f->read(f, String), "./xn.txt")

```

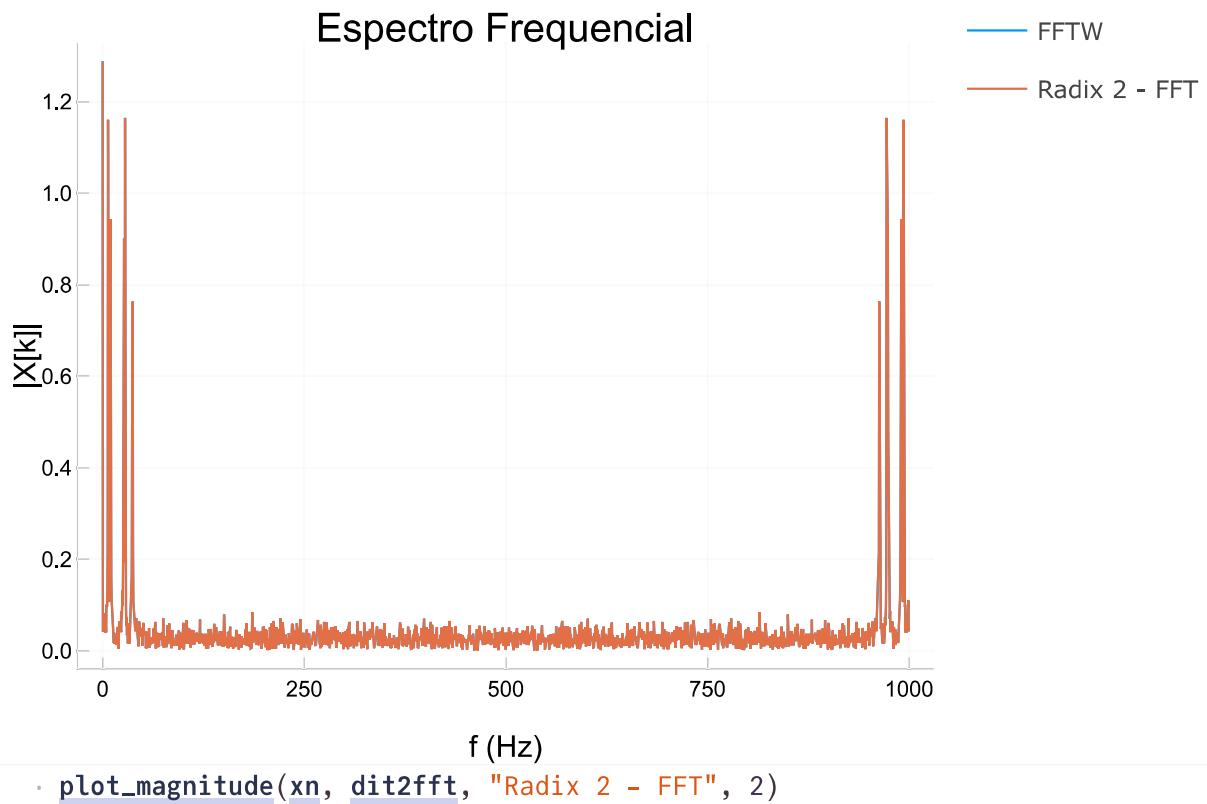
```
process_txt (generic function with 1 method)
· function process_txt(txt)
·     txt = replace(txt, r"\r\n" => "")
·     txt = replace(txt, r"\t" => " ")
·     return split(lstrip(txt), " ")
· end
```

`xn =``[0.234211, 0.727892, 1.55451, 2.01648, 2.20269, 3.09807, 3.44443, 3.41787, 4.21191, 3.14...``· xn = parse.(Float64, process_txt(xn_txt))``t_xn = 0.0:0.001:2.047``· t_xn=0:T:(length(xn)-1)*T # Vetor de tempo`

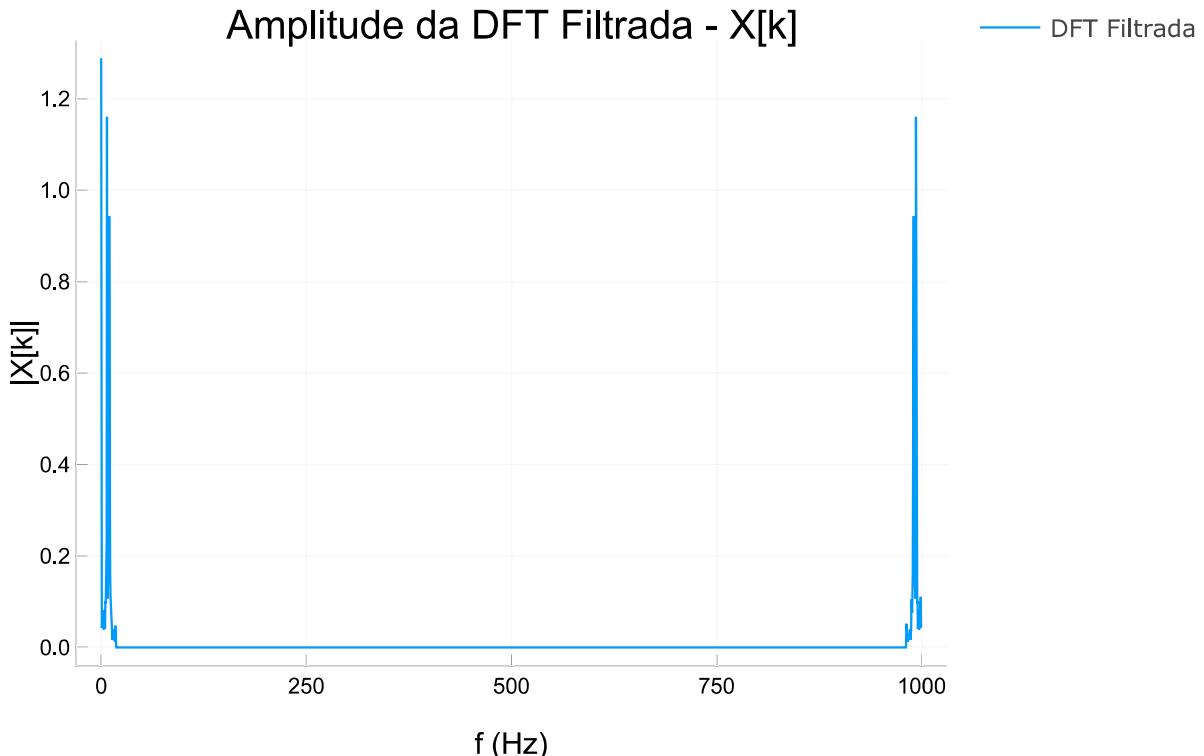
Sinal Amostrado  $x_n[k]$



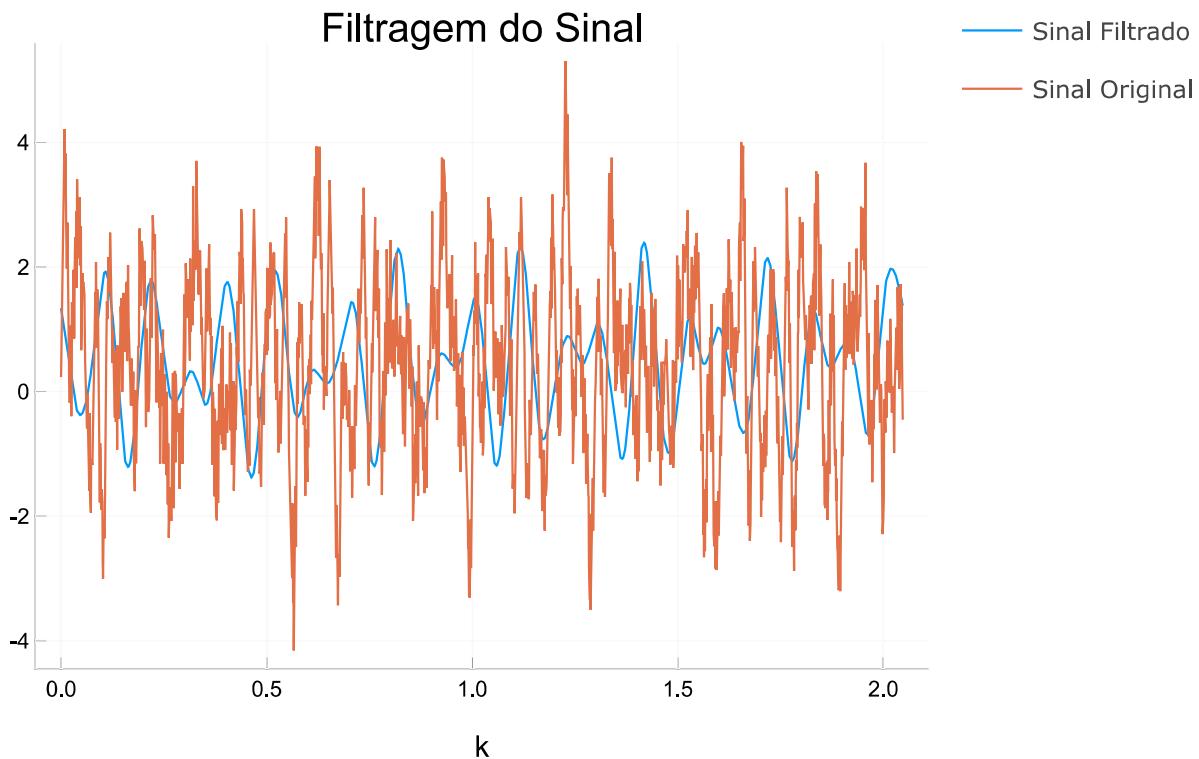
```
begin
·     plot(t_xn, xn, label=nothing)
·     title!("Sinal Amostrado  $x_n[k]$ ")
·     xlabel!("k")
· end
```



Filtrando o sinal com  $\omega_c = 18\text{Hz}$



```
begin
    L_xn = length(xn)
    f_xn = Fs*(0:L_xn-1)/L_xn
    cut_freq = findfirst(value->value >= 18 - ε, f_xn)
    mask_xn = ones(L_xn)
    mask_xn[cut_freq:L_xn-cut_freq] .= 0.
    plot(f_xn, dft_amp(xn, dit2fft) .* mask_xn, label="DFT Filtrada")
    title!("Amplitude da DFT Filtrada - X[k]")
    xlabel!("f (Hz)")
    ylabel!("|X[k]|")
end
```



```

begin
    plot(t_xn, real.(idft(dit2fft(xn) .* mask_xn)), label="Sinal Filtrado")
    plot!(t_xn, xn, label="Sinal Original")
    title!("Filtragem do Sinal")
    xlabel!("k")
end

```

## Referências

Kutz, J. N., Brunton, S. L. (2019). Data-Driven Science and Engineering: Machine Learning, Dynamical Systems, and Control. Singapore: Cambridge University Press.

Rao, K., Kim, D., & Hwang, J. (2011). Fast Fourier Transform - Algorithms and Applications. Springer Netherlands.

## Links

- [Fast Fourier transform](#)
- [Cooley–Tukey FFT algorithm](#)
- [Exemplo FFT - MATLAB](#)
- [FFTW](#)

