À la fréquence de Fourier

1.1 La Transformée de Fourier

À quoi ça sert ?

Un monde parallèle ou les choses deviennent plus simples (ou plus compliquées)

- **1.2.1 Traitement du signal** Séparation des contributions de chaque fréquence
- **1.2.2 Avantage computationel** L'opération de convolution devient une multiplication classique

1.1.1 Illustration : séparation des fréquences

Audio

la

0:00 / 0:00

ré

0:00 / 0:00

la + ré

0:00 / 0:00

Signal temporel

Un son est un signal continu mais on va l'échantillonner. C'est à dire qu'on va prendre un nombre fini de valeur par seconde à des distance égale dans le temps. Dans cet exemple, on prend une valeur toute les 1024^{e} de seconde.

Fréquence d'échantillonnage = 1024

Ça correspond à une distance en seconde entre deux échantillons de:

```
Δt = 0.0009765625
Δt = 1 / échantillonnage
```

nombre_échantillons = 1024

```
temps_total = 1.0
temps_total = nombre_échantillons * Δt
```

En prenant 1024, le signal dure 1.0 secondes. Comme ces nombres sont équidistants (distance de `Δt` secondes entre eux), on peut représenter cette suite de façon compact comme suit:

```
temps = 0.0009765625:0.0009765625:1.0
temps = range(Δt, stop=temps_total, length=nombre_échantillons)
```

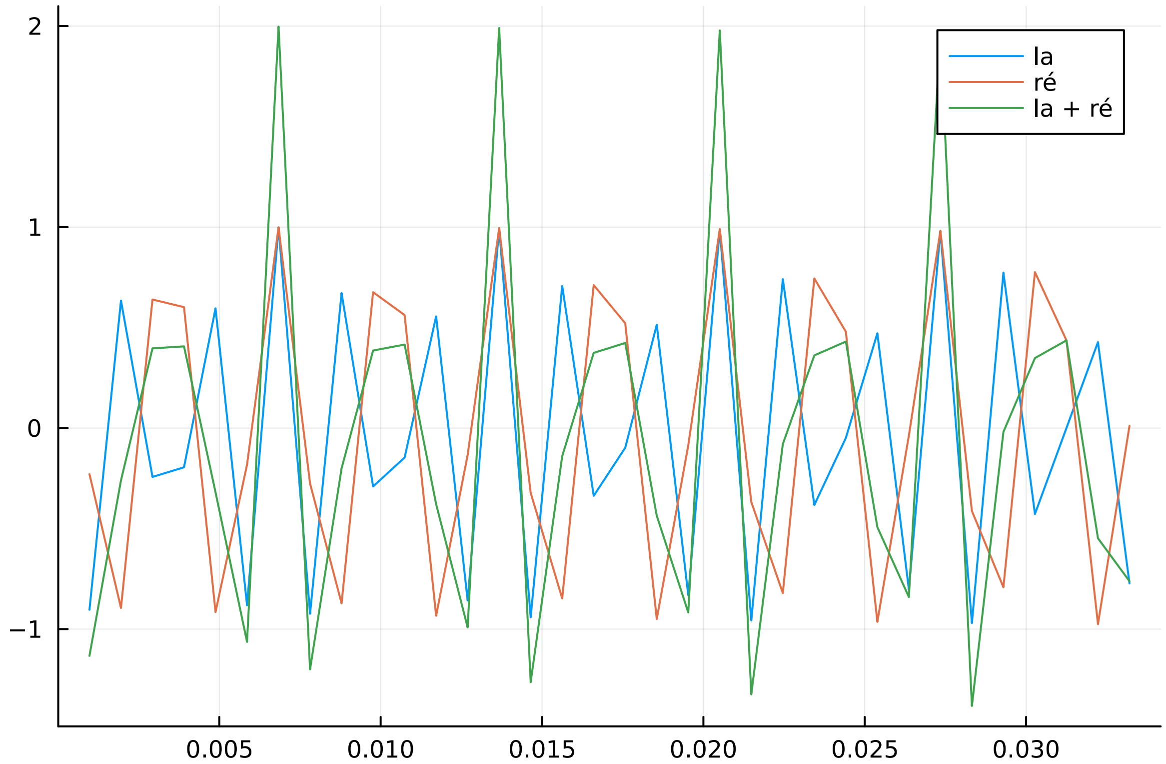
Le *la* est une note de musique de fréquence 440 Hz

```
la =
▸[-0.903989+0.427555im, 0.634393-0.77301im, -0.24298+0.970031im, -0.19509-0.980785im, 0.5
la = cispi.(2*440*temps)
```

La note de musique *ré* a une fréquence de 293.7 Hz

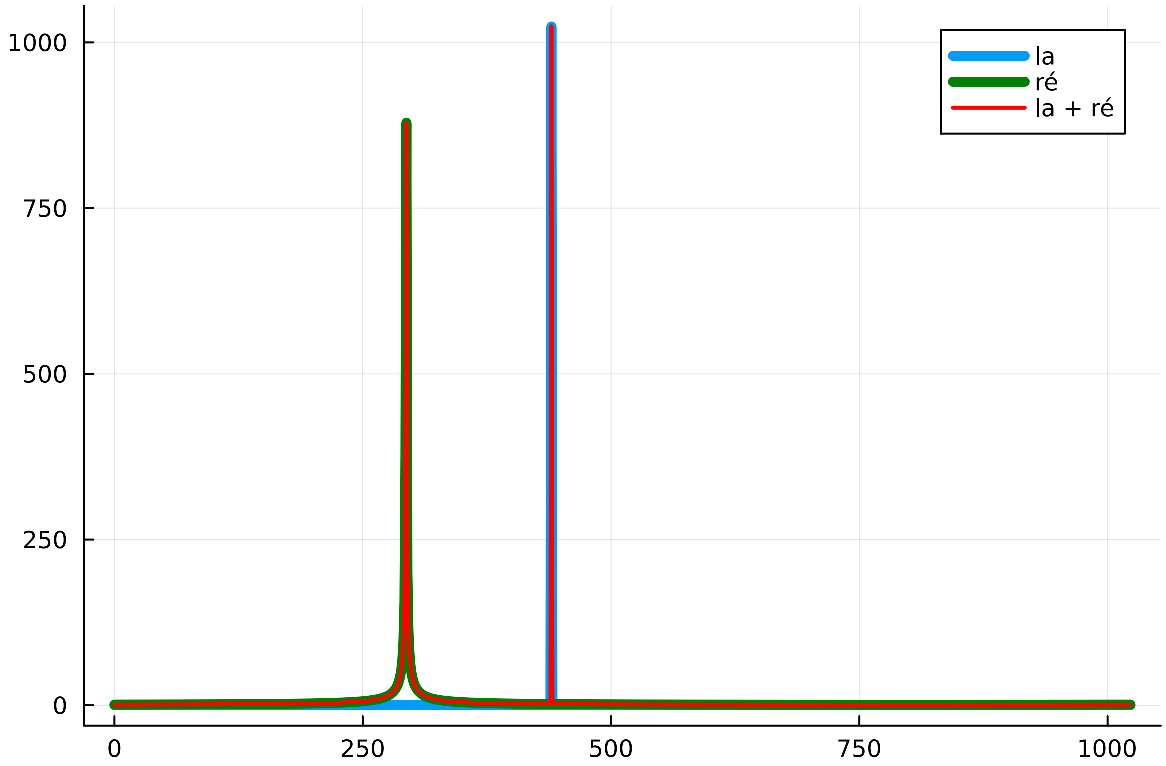
```
ré =
▸[-0.229267+0.973364im, -0.894874-0.44632im, 0.639596-0.768711im, 0.601597+0.7988im, -0.9
ré = cispi.(2*293.7*temps)
```

temps_zoom = 0.032447076612903226



Signal fréquentiel

abs



On a la valeur de la transformée de fourier, tous les 1.0 Hz

1.1.2 Calcul rapide de convolution

La convolution continue:

$$(f * g)(x) = \int_{-\infty}^{\infty} f(s) g(x - s) \mathrm{d}s$$

La convolution discrète:

$$(f * g)_n = \sum_{k=-\infty}^{\infty} f_k g_{n-k}$$

Propriétés algébriques:

$$\begin{aligned}
\text{commutatif} \qquad & f * g = g * f \\
\text{associatif} \qquad & (f * g) * h = f * (g * h) \\
\text{distributif} \qquad & f * (g + h) = f * g + f * h \\
\text{conjugué} \qquad & \overline{f * g} = \overline{f} * \overline{g}
\end{aligned}$$

La cross-corrélation:

$$(f \star g)(x) = \int_{-\infty}^{\infty} f(s)g(x+s)\mathrm{d}s \qquad (f \star g)_n = \sum_{-\infty}^{\infty} f_k g_{n+k}\mathrm{d}s$$

Convolution theorem: Notons la tranformée de Fourier de f par $\mathcal{F}(f)$. Par la transformée de Fourier, la convolution devient un produit classique:

$$f * g = \mathcal{F}^{-1}(\mathcal{F}(f) \cdot \mathcal{F}(g))$$

▶ **Quel est la complexité de la convolution vs produit classique s'ils sont discrets de longueur n ?**

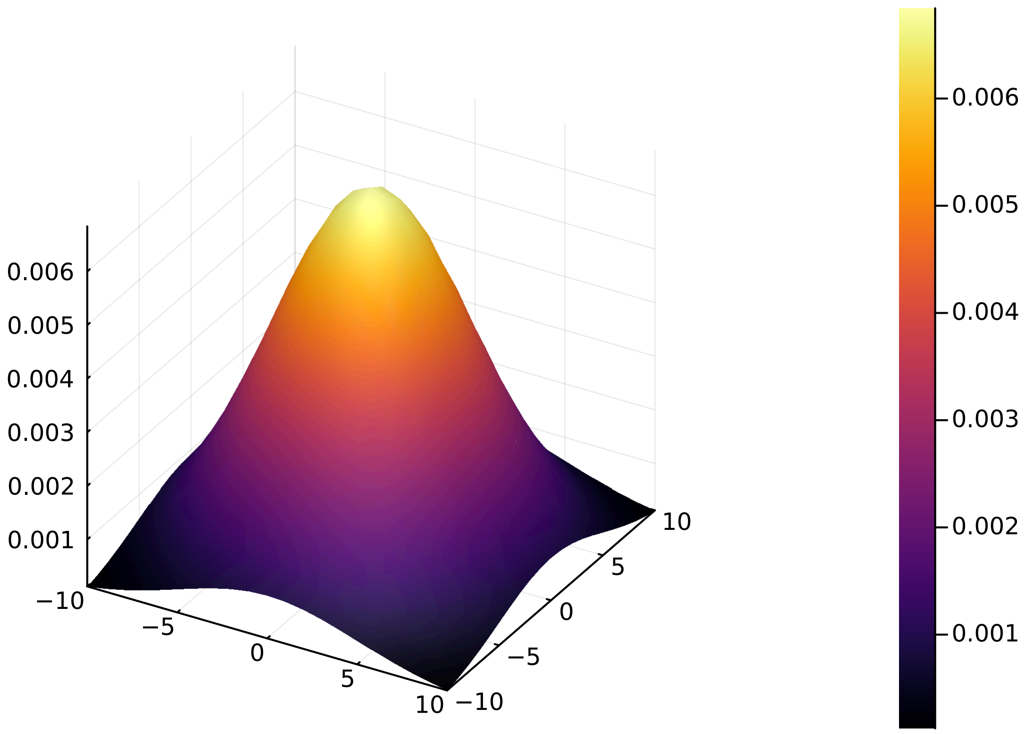
1.1.2.1 Illustration: convolution d'images

mandrill =

```
mandrill = testimage("mandrill")
```

Gaussian Kernel

La fonction `gaussian(d)` donne une matrix de taille $(2d+1) \times (2d+1)$ contenant la valeur d'une Gaussienne.




```
surface(-2d:2d, -2d:2d, Kernel.gaussian(d))
```

d = 5

La fonction `imfilter` calcule la cross-corrélation.

```
imfilter(mandrill, Kernel.gaussian(d))
```

La convolution est obtenue avec `reflect`.

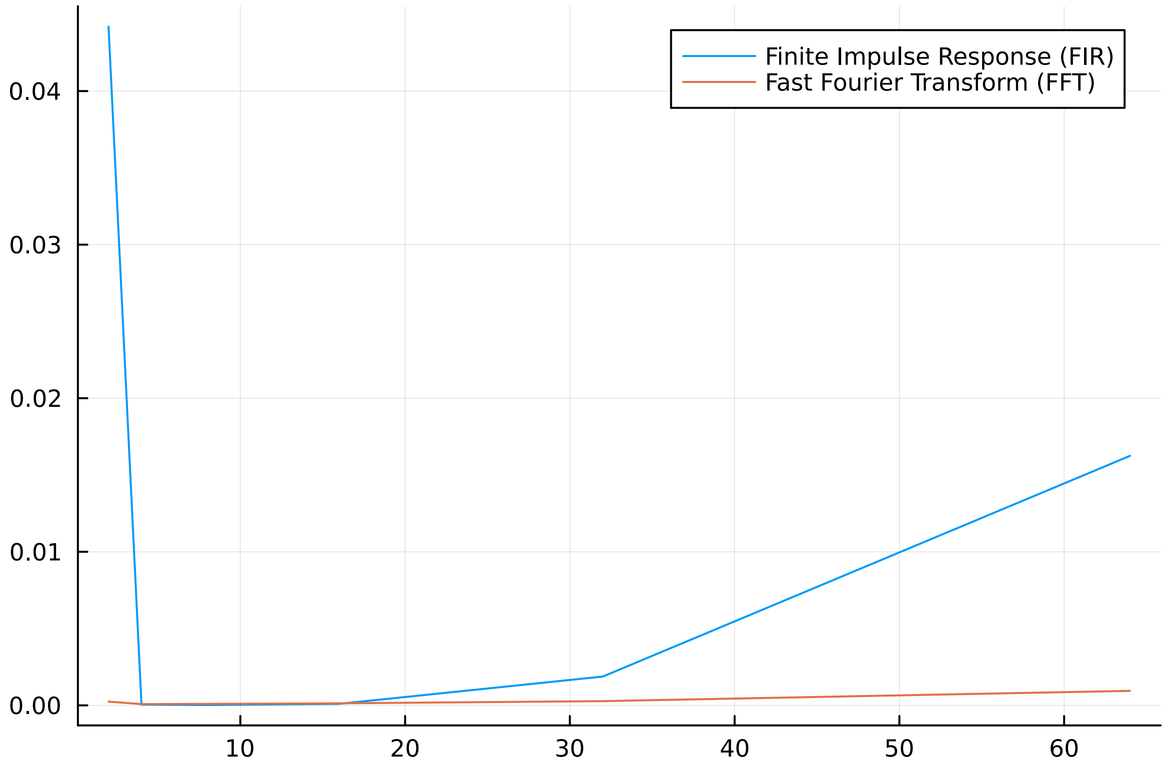
```julia
imfilter(mandrill, reflect(Kernel.gaussian(d)))
```

▶ **Pourquoi obtient-on le même résultat avec la convolution et la cross-corrélation ?**

Performance

imfilter_time (generic function with 1 method)

```julia
function imfilter_time(n, alg)
    A = rand(n, n)
    B = rand(n, n)
    return @elapsed imfilter(A, B, alg)
end
```




```julia
let
    n = 2 .^ (1:6)
    fir_time(n) = imfilter_time(n, Algorithm.FIR())
    fft_time(n) = imfilter_time(n, Algorithm.FFT())
    plot(n, fir_time ∘ Int, label = "Finite Impulse Response (FIR)")
    plot!(n, fft_time ∘ Int, label = "Fast Fourier Transform (FFT)")
end
```

```
3×3 OffsetArray(::Matrix{Int64}, -1:1, -1:1) with eltype Int64 with indices -1:1×-1:1:
 0   1  0
 1  -4  1
 0   1  0
```

```julia
convert(AbstractArray, Kernel.Laplacian())
```

▶ **Quelle est la complexité de FIR ? et de de FFT ?**

Autre kernels

```
imfilter(mandrill, Kernel.Laplacian())
```

Différents kernels permettent d'analyser des aspects différents d'une image. Le kernel à utilisé peut aussi être **appris**, c'est la base des **Convolutional Neural Networks** (CNNs) !

1.1.2.2 Illustration : Le produit de polynômes

deg = 4

```
deg = 4
```

p = $0.7796687029531827 + 0.469518663671715 \cdot x + 0.0011458814335719714 \cdot x^2 + 0.9384199938988756 \cdot x^3$

```
p = Polynomial(rand(deg))
```

q = $0.6796170322631845 + 0.32107489164797887 \cdot x + 0.06281710966206722 \cdot x^2 + 0.16803807511205748 \cdot x^3$

```
q = Polynomial(rand(deg))
```

$0.5298761300495284 + 0.5694249251187605 \cdot x + 0.2005059490178012 \cdot x^2 + 0.7986419584807358 \cdot x^3 + 0.38027209129363143 \cdot x^4 + 0.0591413833762262 \cdot x^5 + 0.1576902894214358 \cdot x^6$

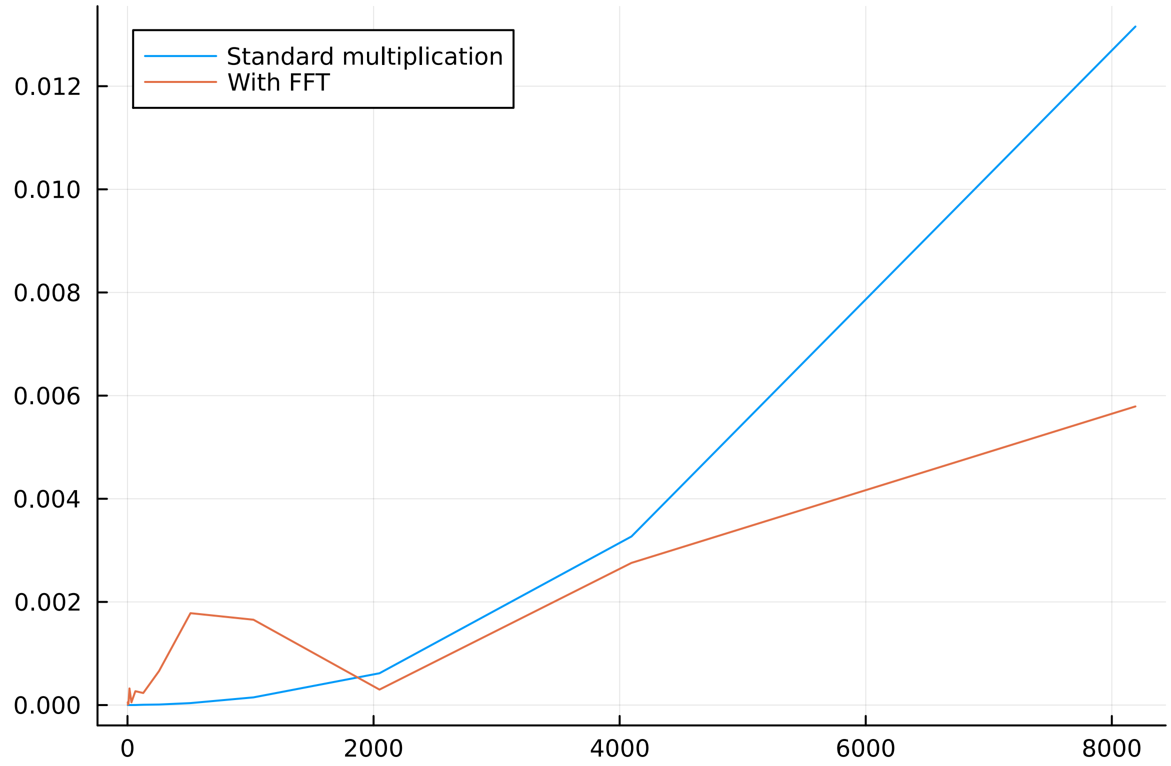
```julia
p * q
```

$0.5298761300495284 + 0.5694249251187605 \cdot x + 0.2005059490178012 \cdot x^2 + 0.7986419584807358 \cdot x^3 + 0.38027209129363154 \cdot x^4 + 0.05914138337622623 \cdot x^5 + 0.15769028942143584 \cdot x^6$

```julia
Polynomials.poly_multiplication_fft(p, q)
```

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

```julia
function poly_time(n, with_fft::Bool)
    p = Polynomial(rand(n))
    q = Polynomial(rand(n))
    @elapsed if with_fft
        Polynomials.poly_multiplication_fft(p, q)
    else
        p * q
    end
end
```




```
let
    n = 2 .^ (1:13)
    std_time(n) = poly_time(n, false)
    fft_time(n) = poly_time(n, true)
    plot(n, std_time ∘ Int, label = "Standard multiplication")
    plot!(n, fft_time ∘ Int, label = "With FFT")
end
```

▶ **Quelle est la complexité du produit de polynômes avec ou sans la FFT ?**

▶ **Comment utiliser cela pour calculer le produit de grand nombres ?**

1.1.2 Définition de la transformée de Fourier

La transformée de Fourier continue et son inverse:

$$F(\xi) = \int_{-\infty}^{\infty} f(t) e^{-i2\pi\xi t} \mathrm{d}t$$
$$f(t) = \int_{-\infty}^{\infty} F(\xi) e^{i2\pi\xi t} \mathrm{d}\xi$$

La transformée de Fourier discrète et son inverse:

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

Intuition Le signal pur "la" est $f_{\text{la}}(t) = \exp(2\pi 440ti) = \cos(2\pi 440t) + i\sin(2\pi 440t)$. Si j'évalue en $t = 1/440s$, j'obtiens la fin d'une période: $\exp(2\pi i) = \cos(2\pi) + i\sin(2\pi)$. Ces exponentielles pures sont *orthogonales*, c'est à dire que ces intégrales entre fréquences différentes sont nulles.

$$F(440) = \int_{-\infty}^{\infty} f(t)e^{-i2\pi 440t}\mathrm{d}t = \int_{-\infty}^{\infty} 1\mathrm{d}t = \infty$$
$$F(293.7) = \int_{-\infty}^{\infty} f(t)e^{-i2\pi 293.7t}\mathrm{d}t = 0$$

Cet intégrale permet donc de détecter la quantité d'une certaine fréquence dans un signal, en ignorant toutes les autres fréquences!

Rajouter linearité : TODO

▶ **Que vaut la distance entre deux valeurs successive en temporel et en fréquentiel pour les signaux discrets ?**

TODO : même dimension en temporel et fréquentiel

▶ **Que vaut le signal en dehors de ses N points ?**

1.1.3 Le replis spectral

On vient de voir que le signal est extrapolé périodiquement par la transformée de fourier discrète. Si on prend un signal de `0` à `t_max` et qu'on divise la fréquence d'échantillonnage de sa transformée de fourier par 2. Ça signifie que le signal sera 2 fois moins long.

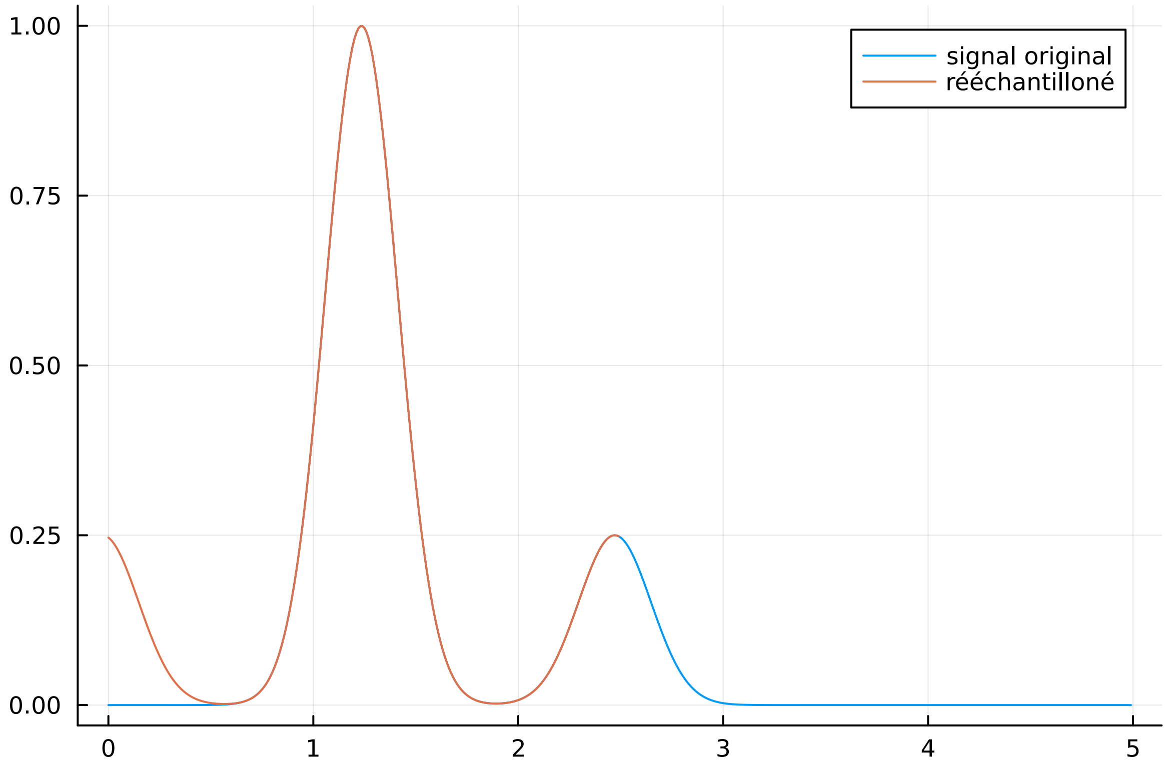
▶ **Que va-t-il advenir de la deuxième moitié du signal ?**

Illustrons cela avec la fonction `exp(-(t-centre)^2)` qui est initiallement échantillonnée avec 512 échantillons de `0` à `t_max` et qu'on ré-échantillonne ensuite à `512 / ratio` en fréquentiel, ce qui divise `t_max` par `ratio` également.

t_max = 5.0

centre = 2.470967741935484

ratio = 2



1.2 Fast Fourier Transform

Attardons-nous à présent sur l'algorithme utilisé pour calculé la DFT (transformée de fourier discrète). On a vu précédemment que cet algorithme a une complexité de $\Omega(n\log(n))$. C'est parmis le top 10 des meilleurs algorithmes du 20e siècle ! Son Implémentation la plus rapide est dans la libraire FFTW : the Fastest Fourier Transform in the West ! Comment est-elle la plus rapide ? Voir ici. L'algorithme est initialement découvert part Gauss en 1805 puis redécouvert par Cooley et Tukey en 1965.

La DFT est en fait une évaluation d'un polynômes aux différentes racines `N` ièmes de l'unité:

$$X_k = \sum_{n=0}^{N-1} x_n e^{-i2\pi \frac{k}{N} n}$$
$$X_k = \sum_{n=0}^{N-1} x_n z_{k,N}^n \qquad z_{k,N} = e^{-i2\pi \frac{k}{N}}$$

La DFT peut aussi être vue comme un produit matriciel:

$$X = \begin{bmatrix} 1 & 1 & \cdots & 1 & 1 \\ 1 & z_N & \cdots & z_N^{N-2} & z_N^{N-1} \\ 1 & z_N^2 & \cdots & z_N^{2(N-2)} & z_N^{2(N-1)} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 1 & z_N^{(N-1)2} & \cdots & z_N^{(N-1)(N-2)} & z_N^{(N-1)(N-1)} \end{bmatrix} x$$

Un produit matriciel a complexité $\Omega(n^2)$. Ça ne nous donne pas une complexité de $\Omega(n \log n)$, il va falloir utiliser la structure assez spéciale de cette matrice...

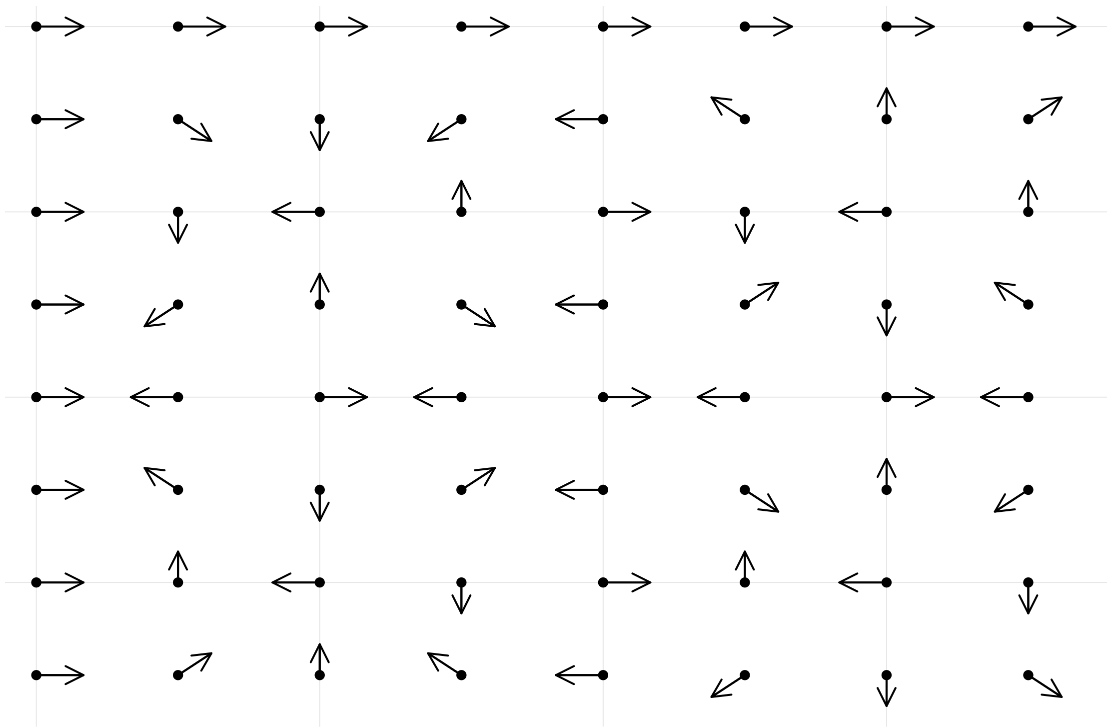
On a acquis une intuition géométrique sur les racines z_N. On va s'en servir pour visualiser cette matrice.

```
8×8 Matrix{ComplexF64}:
 1.0+0.0im        1.0+0.0im        1.0+0.0im  …   1.0+0.0im        1.0+0.0im
 1.0+0.0im   0.707107-0.707107im   0.0-1.0im      0.0+1.0im   0.707107+0.707107im
 1.0+0.0im        0.0-1.0im       -1.0-0.0im     -1.0-0.0im        0.0+1.0im
 1.0+0.0im  -0.707107-0.707107im   0.0+1.0im      0.0-1.0im  -0.707107+0.707107im
 1.0+0.0im       -1.0-0.0im        1.0-0.0im      1.0-0.0im       -1.0-0.0im
 1.0+0.0im  -0.707107+0.707107im   0.0-1.0im  …   0.0+1.0im  -0.707107-0.707107im
 1.0+0.0im        0.0+1.0im       -1.0-0.0im     -1.0-0.0im        0.0-1.0im
 1.0+0.0im   0.707107+0.707107im   0.0+1.0im      0.0-1.0im   0.707107-0.707107im
```

```
F(8)
```

▶ **Qu'observe-t-on dans la matrice ci-dessous ?**

Besoin d'un indice ? ☐



On peut le vérifier numériquement également:

```
partie_mauve = 4×4 Matrix{ComplexF64}:
               1.0+0.0im   1.0+0.0im   1.0+0.0im   1.0+0.0im
               1.0+0.0im   0.0-1.0im  -1.0-0.0im   0.0+1.0im
               1.0+0.0im  -1.0-0.0im   1.0-0.0im  -1.0-0.0im
               1.0+0.0im   0.0+1.0im  -1.0-0.0im   0.0-1.0im
```

```
partie_mauve = F(8)[1:4, 1:2:8]
```

```
partie_bleue = 4×4 Matrix{ComplexF64}:
               1.0+0.0im   1.0-0.0im   1.0-0.0im   1.0-0.0im
               1.0+0.0im   0.0-1.0im  -1.0-0.0im   0.0+1.0im
               1.0+0.0im  -1.0-0.0im   1.0-0.0im  -1.0-0.0im
               1.0+0.0im   0.0+1.0im  -1.0-0.0im   0.0-1.0im
```

```
partie_bleue = F(8)[5:8, 1:2:8]
```

```
true
```

```
partie_mauve == partie_bleue
```

```
partie_verte = 4×4 Matrix{ComplexF64}:
                     1.0+0.0im                1.0+0.0im         …         1.0+0.0im
                0.707107-0.707107im  -0.707107-0.707107im       0.707107+0.707107im
                     0.0-1.0im                0.0+1.0im                   0.0+1.0im
               -0.707107-0.707107im   0.707107-0.707107im      -0.707107+0.707107im
```

```
partie_verte = F(8)[1:4, 2:2:8]
```

```
partie_rouge = 4×4 Matrix{ComplexF64}:
              -1.0-0.0im              -1.0-0.0im        …         -1.0-0.0im
         -0.707107+0.707107im    0.707107+0.707107im     -0.707107-0.707107im
               0.0+1.0im               0.0-1.0im                 0.0-1.0im
          0.707107+0.707107im   -0.707107+0.707107im      0.707107-0.707107im
```

```julia
partie_rouge = F(8)[5:8, 2:2:8]
```

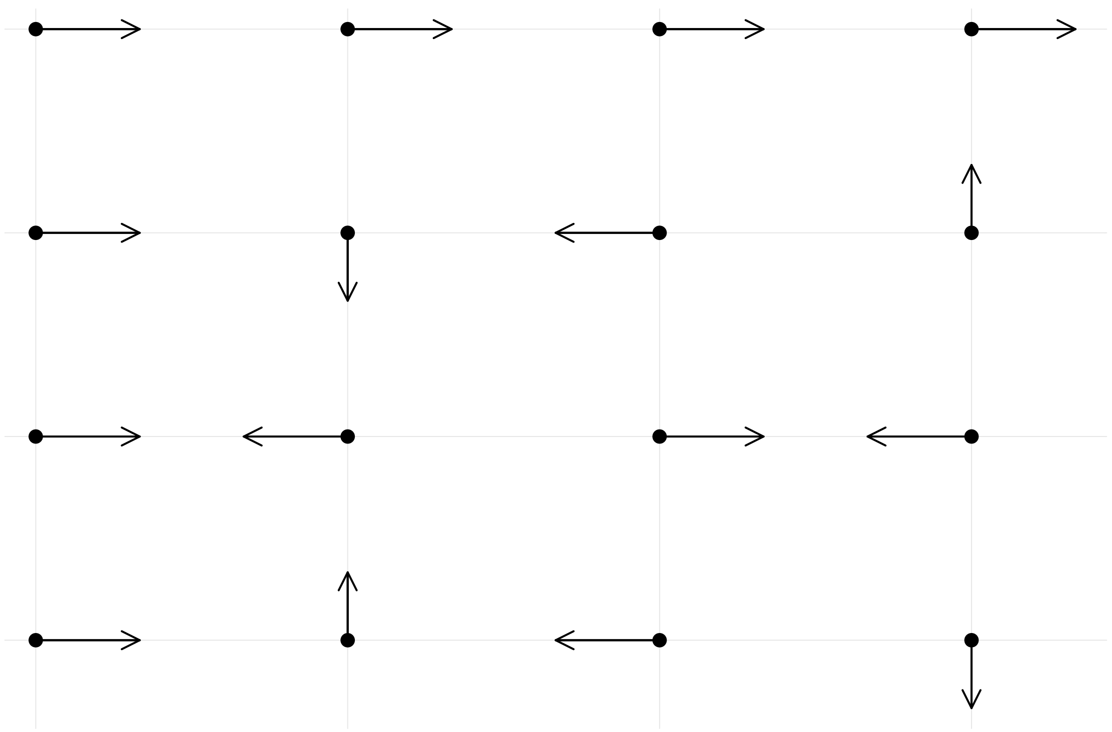
```
true
```

```julia
partie_verte == -partie_rouge
```

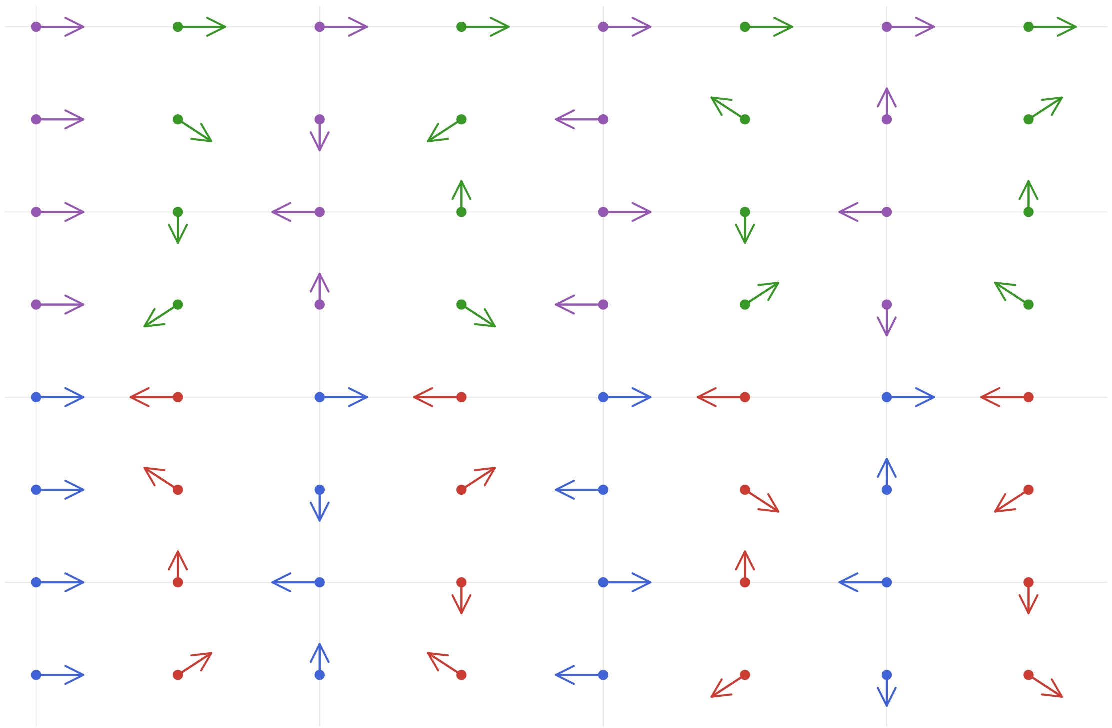
```
true
```

```julia
isapprox(0.0, 1e-300, atol = 1e-8)
```

▶ **Qu'observe-t-on en comparant cela à la matrice 2 fois plus petite ?**



```julia
arrows(4, false)
```



```
1 arrows(8, true)
```

Vérifions le à nouveau numériquement:

```
4×4 Matrix{ComplexF64}:
 1.0+0.0im   1.0-0.0im   1.0-0.0im   1.0-0.0im
 1.0+0.0im   0.0-1.0im  -1.0-0.0im   0.0+1.0im
 1.0+0.0im  -1.0-0.0im   1.0-0.0im  -1.0-0.0im
 1.0+0.0im   0.0+1.0im  -1.0-0.0im   0.0-1.0im
```

```
1 partie_bleue
```

```
4×4 Matrix{ComplexF64}:
 1.0+0.0im   1.0+0.0im   1.0+0.0im   1.0+0.0im
 1.0+0.0im   0.0-1.0im  -1.0-0.0im   0.0+1.0im
 1.0+0.0im  -1.0-0.0im   1.0-0.0im  -1.0-0.0im
 1.0+0.0im   0.0+1.0im  -1.0-0.0im   0.0-1.0im
```

```
1 F(4)
```

```
true
```

```
1 partie_bleue == F(4)
```

```
4×4 Matrix{ComplexF64}:
       1.0+0.0im            1.0+0.0im       …        1.0+0.0im
  0.707107-0.707107im  -0.707107-0.707107im      0.707107+0.707107im
       0.0-1.0im            0.0+1.0im                0.0+1.0im
 -0.707107-0.707107im   0.707107-0.707107im     -0.707107+0.707107im
```

```
partie_verte
```

```
0.7071067811865476 - 0.7071067811865475im
```

```
cispi(-2/8)
```

```
[1.0+0.0im, 0.707107-0.707107im, 2.22045e-16-1.0im, -0.707107-0.707107im]
```

```
cispi(-2/8) .^ collect(0:3)
```

```
4×4 Matrix{ComplexF64}:
         1.0+0.0im            1.0+0.0im       …        1.0+0.0im
    0.707107-0.707107im  -0.707107-0.707107im      0.707107+0.707107im
 2.22045e-16-1.0im       -2.22045e-16+1.0im       -2.22045e-16+1.0im
   -0.707107-0.707107im   0.707107-0.707107im     -0.707107+0.707107im
```

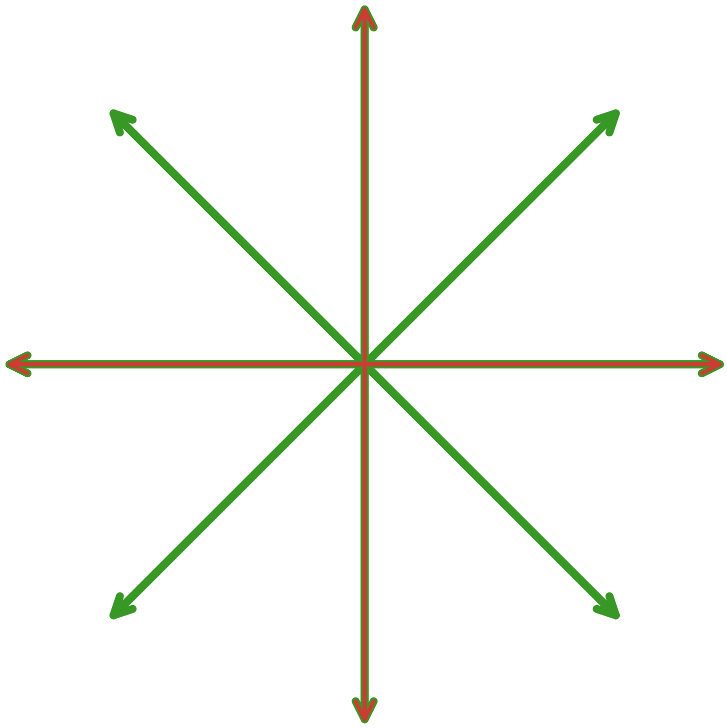
```
cispi(-2/8) .^ (0:3) .* F(4)
```

```
4×4 BitMatrix:
 1  1  1  1
 1  1  1  1
 1  1  1  1
 1  1  1  1
```

```
isapprox.(real.(partie_verte), real.(cispi(-2/8) .^ (0:3) .* F(4)), atol=1e-8)
```

En se rappelant que l'intuition géométrique des `N` racines de l'unité, on comprend pourquoi les `N/2` racines sont un sous-ensembles des `N` racines.

max_N = 8



▶ **Quel est la complexité de l'algorithme FFT.**

Utilitaires

```julia
using Plots
```

```julia
using PlutoUI, FFTW, Images, TestImages, ImageFiltering
```

```julia
using Polynomials, LinearAlgebra
```

```julia
import WAV
```

play_sound (generic function with 1 method)

```julia
# See https://discourse.julialang.org/t/how-do-i-play-sound-in-a-notebook-preferably-pluto/49403/8?u=blegat
function play_sound(sound, samplerate)
    dir = mktempdir(cleanup = false)
    file = joinpath(dir, "audio.wav")
    WAV.wavwrite(Int.(trunc.(real.(sound*2^15))), file, Fs=samplerate, nbits=16)
    md"""$(LocalResource(file))"""
end
```

```julia
begin
    struct Join
        list
        Join(a) = new(a)
        Join(a, b, args...) = Join(tuple(a, b, args...))
    end
    function Base.show(io::IO, mime::MIME"text/html", d::Join)
        for el in d.list
            show(io, mime, el)
        end
    end
end
```

```julia
begin
    struct HTMLTag
        tag::String
        parent
    end
    function Base.show(io::IO, mime::MIME"text/html", d::HTMLTag)
        write(io, "<", d.tag, ">")
        show(io, mime, d.parent)
        write(io, "</", d.tag, ">")
    end
end
```

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

```julia
f(i, j, n) = cispi(-2 * i * j / n)
```

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

```julia
function F(n)
    A = [
        f(k, m, n)
        for k in 0:(n-1), m in 0:(n-1)
    ]
end
```

```
8×8 Matrix{ComplexF64}:
 1.0+0.0im        1.0+0.0im        1.0+0.0im  …   1.0+0.0im        1.0+0.0im
 1.0+0.0im   0.707107-0.707107im   0.0-1.0im      0.0+1.0im   0.707107+0.707107im
 1.0+0.0im        0.0-1.0im       -1.0-0.0im     -1.0-0.0im        0.0+1.0im
 1.0+0.0im  -0.707107-0.707107im   0.0+1.0im      0.0-1.0im  -0.707107+0.707107im
 1.0+0.0im       -1.0-0.0im        1.0-0.0im      1.0-0.0im       -1.0-0.0im
 1.0+0.0im  -0.707107+0.707107im   0.0-1.0im  …   0.0+1.0im  -0.707107-0.707107im
 1.0+0.0im        0.0+1.0im       -1.0-0.0im     -1.0-0.0im        0.0-1.0im
 1.0+0.0im   0.707107+0.707107im   0.0+1.0im      0.0-1.0im   0.707107-0.707107im
```

```julia
F(8)
```

arrows (generic function with 2 methods)

```
function arrows(n, use_color = false)
    plot(showaxis = false)
    for i in 0:(n-1)
        for j in 0:(n-1)
            if use_color
                color = Colors.JULIA_LOGO_COLORS[1 + iseven(j) * 2 + (2i < n)]
            else
                color = :black
            end
            scatter!([j], [-i], markersize = 8 / sqrt(n), markerstrokewidth = 0,
            legend = nothing; color)
            c = f(i, j, n) / 3 # Divide by 3 so that arrow is not too long in plot
            plot!([j, j + real(c)], [-i, -i + imag(c)], arrow = true, legend =
            nothing; color)
        end
    end
    plot!()
end
```

draw_roots! (generic function with 1 method)

```
function draw_roots!(N; kws...)
    for i in 1:N
        b, a = sincospi(2 * i / N)
        plot!([0, a], [0, b]; arrow = true, label = nothing, kws...)
    end
end
```

qa (generic function with 1 method)

```
function qa(question, answer)
    return HTMLTag("details", Join(HTMLTag("summary", question), answer))
end
```