Domaine fréquentiel et 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

0. Signal sonore

Un son pur : le cosinus

Avant de parler de fréquences, fixons la brique de base. Le son le plus simple qui soit — une note pure qui s'écrit :

$$x(t) = \cos(2\pi f t)$$

où f est la **fréquence** en Hertz (Hz $= s^{-1}$) et t le **temps** en secondes. La fréquence est le **nombre d'oscillations par seconde**.

D'où sort ce 2π ? Le cosinus accomplit un tour complet quand son argument avance de 2π : c'est sa période naturelle. En plaçant $2\pi f$ devant le t, on règle la vitesse à laquelle cet argument défile.

Vérifions sur un exemple. Pour $f = 5$ Hz :

$$x(t) = \cos(2\pi \cdot 5 \cdot t)$$

Évaluons l'argument en $t = 1/5$ de seconde :

$$2\pi \cdot 5 \cdot \tfrac{1}{5} = 2\pi$$

L'argument vaut exactement 2π : **une oscillation complète** vient de se produire, en un cinquième de seconde. Il y en aura donc 5 en une seconde — et c'est précisément ce que veut dire « 5 Hz ».

Deux quantités en découlent :

- la **période** $T = \dfrac{1}{f}$, la durée d'une oscillation — ici $T = 0.2$ s ;
- la **pulsation** $\omega = 2\pi f$, en radians par seconde, qui permet l'écriture compacte

$$x(t) = \cos(\omega t) = \cos(10\pi t) \qquad \text{pour } f = 5 \text{ Hz}$$

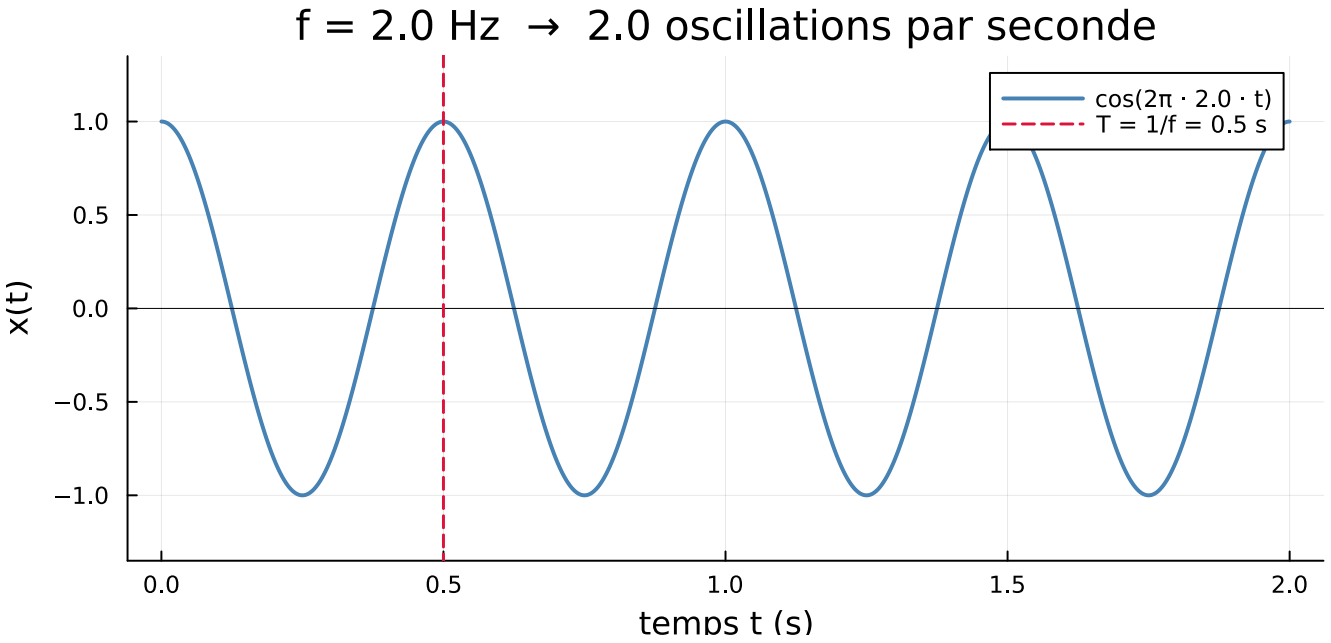
À retenir

f compte les **oscillations par seconde**, ω compte les **radians par seconde**, et $T = 1/f$ est la **durée d'une oscillation**. Les trois disent la même chose sous trois unités différentes.

Manipulez le curseur ci-dessous pour changer la fréquence f, et observez deux choses :

- plus f augmente, plus les oscillations se **resserrent** ;
- le trait rouge marque la fin de la **première** oscillation, en $t = T = 1/f$. Il se rapproche de zéro exactement à la même vitesse.

f = 2.0 Hz




```
let
    durée = 2.0                                    # secondes affichées
    t = range(0, stop = durée, length = 2000)
    x = cos.(2π * f_cos .* t)

    T = 1 / f_cos                                  # la période

    plot(t, x,
        label = "cos(2π · $(f_cos) · t)", color = :steelblue, linewidth = 2,
        xlabel = "temps t (s)", ylabel = "x(t)",
        ylim = (-1.35, 1.35), legend = :topright, size = (680, 320))

    hline!([0], color = :black, linewidth = 0.5, label = nothing)
    vline!([T], color = :crimson, linestyle = :dash, linewidth = 1.5,
        label = "T = 1/f = $(round(T, digits = 3)) s")

    title!("f = $(f_cos) Hz  →  $(f_cos) oscillations par seconde")
end
```

Une précaution avant d'écouter : l'échantillonnage

Un son réel est une onde **continue** : la pression de l'air varie à chaque instant, sans interruption. Mais un ordinateur ne sait pas stocker une infinité de valeurs.

La solution s'appelle l'**échantillonnage** : on enregistre la valeur du signal à intervalles réguliers. Le nombre de relevés par seconde est la **fréquence d'échantillonnage** f_e, exprimée en Hertz (unités $\frac{1}{s}$).

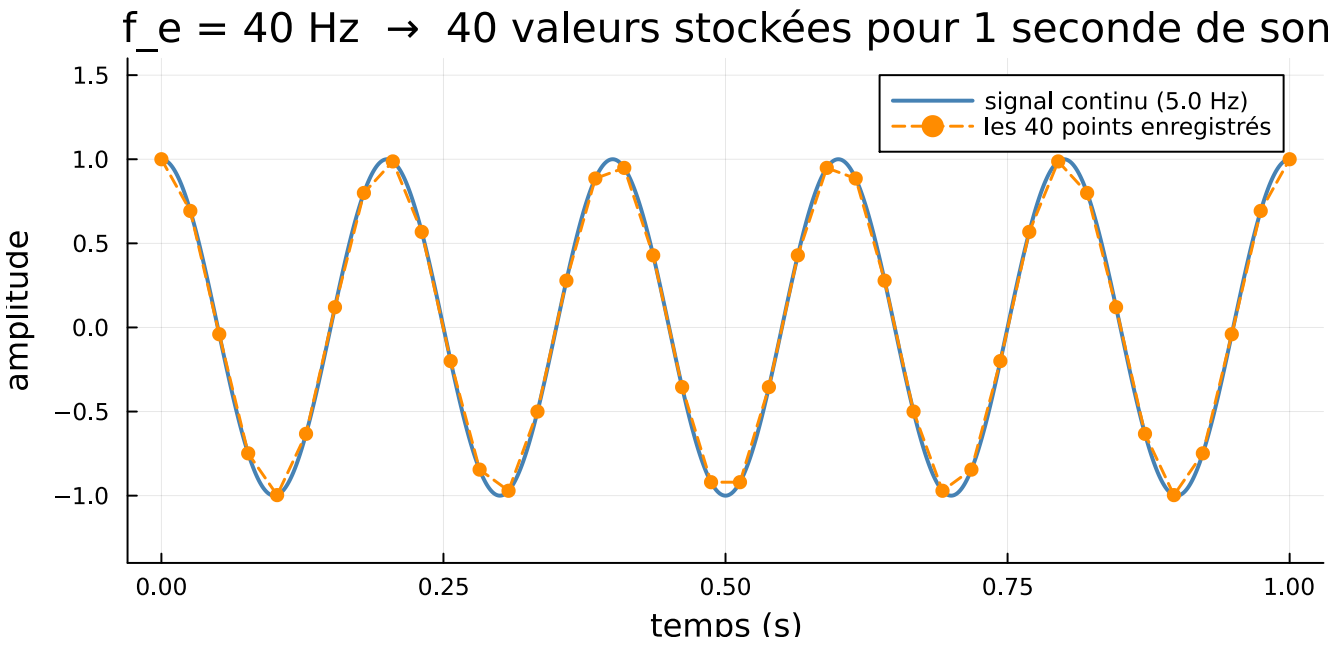
Un CD audio, par exemple, prend **44 100 points par seconde**.

Un son stocké dans un ordinateur, c'est donc simplement **une liste finie de nombres**, accompagnés de la convention qui dit à quelle cadence ils ont été relevés.

Le graphe ci-dessous montre un cosinus à **5 Hz** (en bleu, la « vraie » onde continue) et les points effectivement retenus par l'échantillonnage (en orange). Le curseur règle f_e.

À essayer : faites varier f_e et observez le compromis. Plus f_e est grand, plus les points collent à la courbe — mais plus il y a de nombres à stocker. Plus f_e est petit, plus le fichier est léger — mais moins il reste de détail.

f_e = 40 Hz



▶ **Le trait orange relie les points par des segments droits. Est-ce vraiment ça, le signal ?**

Fixons maintenant les réglages qui serviront à fabriquer deux notes. Les deux curseurs ci-dessous définissent la fréquence d'échantillonnage et le nombre de points ; tout le reste — la durée du signal, l'axe des temps — en découle.

Signal temporel

Fréquence d'échantillonnage = 8192

```
md"Fréquence d'échantillonnage = $(@bind échantillonnage Slider((2).^(4:13), default=8192, show_value = true))"
```

```
fe = 8192
```
```
1  fe = échantillonnage
```

Ça correspond à un écart entre deux échantillons de: (mesuré en secondes)

```
Δt = 0.0001220703125
```
```
1  Δt = 1 / fe
```

nombre_échantillons = 1024

```
temps_total = 0.125
```
```
1  temps_total = nombre_échantillons * Δt
```

En prenant 1024 échantillons, le signal dure 0.125 secondes. Comme ces nombres sont équidistants (distance de Δt secondes entre eux), on peut représenter cette suite de façon compact comme suit:

```
temps = 0.0001220703125:0.0001220703125:0.125
```
```
1  temps = range(Δt, stop=temps_total, length=nombre_échantillons)
```

Avec ces réglages, construisons les deux notes. Le *la* et le *ré* sont de simples cosinus, chacun à sa fréquence.

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

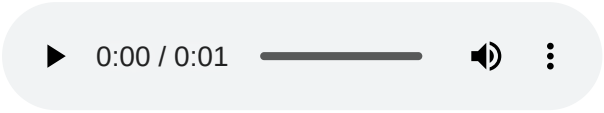
```
la =
▸ [0.943593+0.331106im, 0.780737+0.624859im, 0.529804+0.84812im, 0.219101+0.975702im, -0.1
```
```
1  la = cispi.(2*440*temps)
2  #More accurate method for cis(pi*x)
```

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

```
ré =
▸ [0.974735+0.223365im, 0.900216+0.435443im, 0.78021+0.625518im, 0.620779+0.783986im, 0.42
```
```
1  ré = cispi.(2*293.7*temps)
```

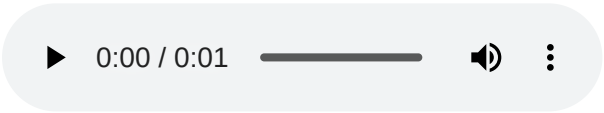
Écoutez ces trois extraits dans l'ordre. Le troisième est **littéralement la somme des deux premiers** : on additionne les deux signaux, échantillon par échantillon.

la




```
let
    # Ces trois cellules audio sont volontairement **indépendantes** des curseurs
    # ci-dessus : elles utilisent des réglages fixes, choisis pour que le son soit
    # audible. D'où les noms locaux `fs_audio` / `t_audio`.
    fs_audio = 2^13          # 8192 Hz
    durée = 2                # secondes
    t_audio = range(start=1/fs_audio, stop=2, length=fs_audio)
    son_la = cispi.(2*440*t_audio)
    md"""
    #### la
    $(play_sound(son_la, fs_audio))
    """
end
```

ré




```
let
    fs_audio = 2^13
    durée = 2
    t_audio = range(1/fs_audio, stop=2, length=fs_audio)
    son_ré = cispi.(2*293.7*t_audio)
    md"""
    #### ré
    $(play_sound(son_ré, fs_audio))
    """
end
```

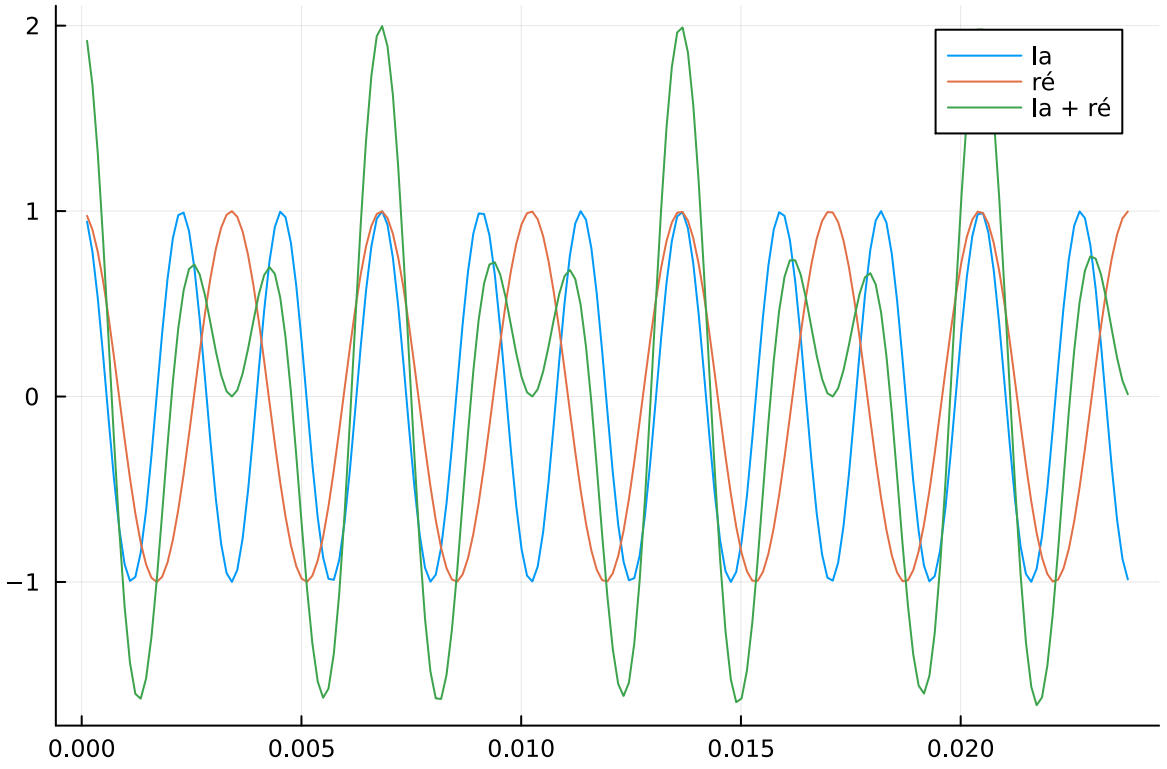
la + ré

0:00 / 0:01

```
let
    fs_audio = 2^13
    durée = 2
    t_audio = range(1/fs_audio, stop=2, length=fs_audio)
    son_la = cispi.(2*440*t_audio)
    son_ré = cispi.(2*293.7*t_audio)
    md"""
    #### la + ré
    $(play_sound(son_la + son_ré, fs_audio))
    """
end
```

Voyons maintenant le signal en temporel:

temps_zoom = 0.023724955897177418




```
begin
    index_max = something(findfirst(t -> t >= temps_zoom, temps), length(temps))
    plot(temps[1:index_max], real.(la[1:index_max]), label = "la")
    plot!(temps[1:index_max], real.(ré[1:index_max]), label = "ré")
    plot!(temps[1:index_max], real.(la[1:index_max] + ré[1:index_max]), label = "la
    + ré")
end
```

▶ **En regardant uniquement la courbe rouge (la + ré), sauriez-vous dire de quelles notes elle est faite ?**

Le fil conducteur de tout le cours

Ce cours construit l'outil mathématique qui permet de **séparer les fréquences**, et qui s'appelle la **transformée de Fourier**.

Le plan tient en trois mots — **aller, travailler, revenir** :

1. **Aller** (§1) — passer du temps domaine temporel vers les fréquences. On va construire la transformation.
2. **Travailler** (§2) — dans le monde des fréquences, des problèmes durs deviennent faciles : nettoyer un son, flouter une image, multiplier des polynômes.

3. **Revenir** (§3) — retourner vers un signal temporel sans rien perdre… et comprendre les pièges quand on échantillonne mal.

Et enfin (§4) : **combien ça coûte ?** C'est l'algorithme *FFT*, l'un des plus importants du 20e siècle.

Une remarque technique avant de continuer

Vous allez voir du code qui écrit les notes avec `cispi`, c'est-à-dire une **exponentielle complexe** $e^{i2\pi ft}$ plutôt qu'un simple cosinus.

$$e^{i2\pi ft} = \cos(2\pi ft) + i\,\sin(2\pi ft)$$

Le son que vous entendez est la **partie réelle**, donc bel et bien un cosinus. Le nombre complexe n'est pas là pour faire joli : il transporte *en même temps* le cosinus et le sinus, et on verra en §1.1 que c'est précisément ce qui rend les calculs simples. Pour l'instant, retenez juste : **exponentielle complexe = une oscillation pure à une fréquence donnée**.

1. Aller : construire la transformée

On cherche une machine qui, nourrie avec le signal `la + ré`, réponde : « *il y a du 440 Hz, et il y a du 293.7 Hz* ».

Plutôt que de sortir la formule d'un chapeau, on va la **fabriquer**. La question à se poser est modeste :

> Étant donné un signal et une fréquence f qui m'intéresse, comment mesurer **la quantité de f présente dans ce signal** ?

1.1 L'idée : mesurer une ressemblance

Voici la recette, en trois gestes. Elle est plus simple qu'elle n'en a l'air.

1. Je fabrique une **onde de référence** parfaitement pure à la fréquence f que je teste.
2. Je **multiplie** mon signal par cette onde de référence, point par point.
3. J'**additionne** tous les produits obtenus. Ce nombre unique est ma mesure.

Pourquoi est-ce que ça marche ? Regardons les deux cas possibles.

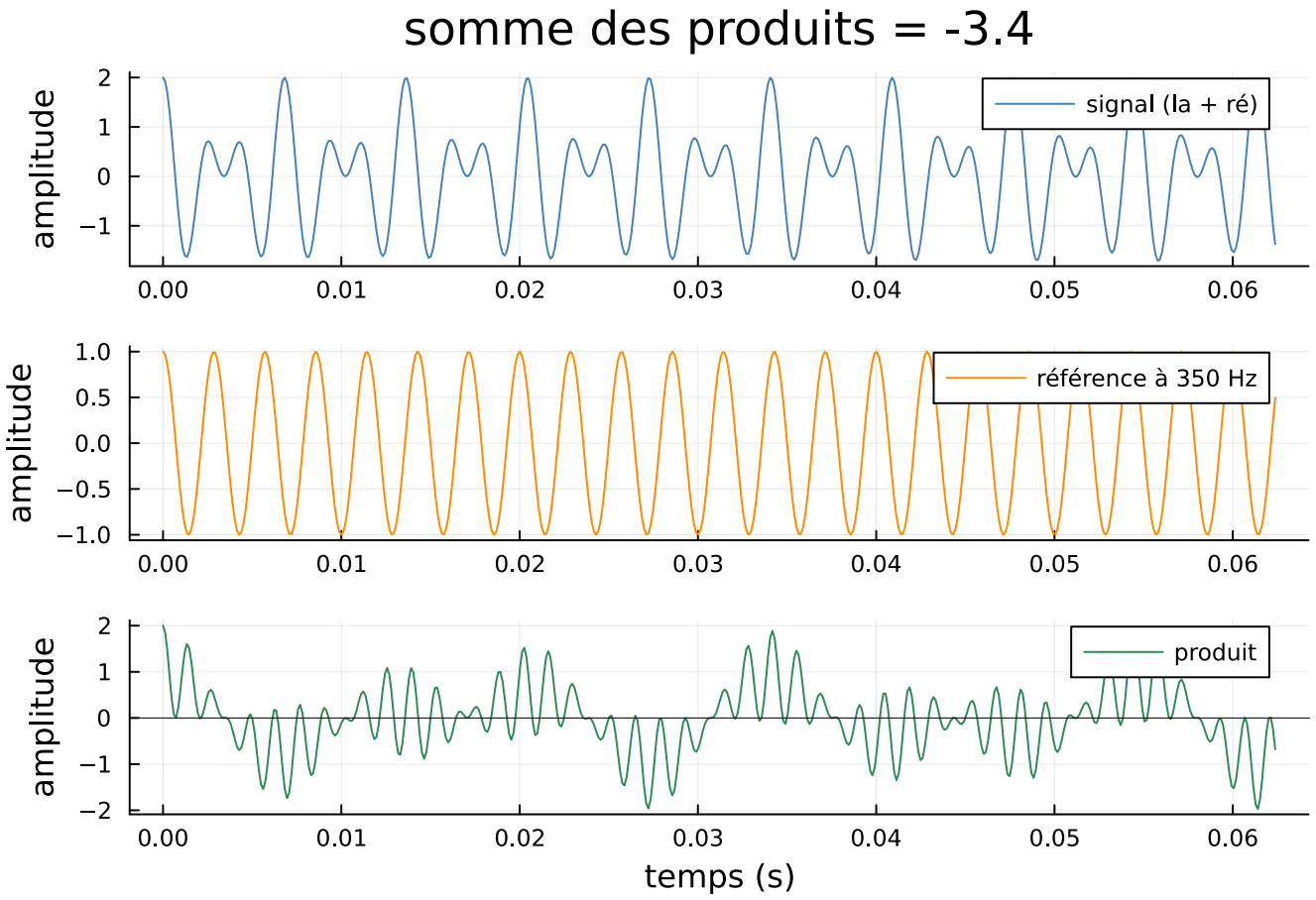
- **Si la fréquence f est présente dans le signal**, les deux courbes montent et descendent *ensemble*. Quand l'une est positive, l'autre l'est aussi ; quand l'une est négative, l'autre aussi. Le produit de deux nombres de même signe est **positif**. Les produits s'accumulent donc tous dans le même sens et la somme devient **grande**.
- **Si la fréquence f est absente**, les deux courbes se décalent sans cesse : parfois en phase, parfois en opposition. Les produits sont tantôt positifs, tantôt négatifs, et ils **se compensent**. La somme retombe à $\mathbf{0}$.

C'est tout. La transformée de Fourier n'est rien d'autre que cette mesure, répétée pour toutes les fréquences.

Testons-le sur notre signal `la + ré`. Déplacez le curseur pour choisir la fréquence testée f et observez les trois graphes ainsi que la somme affichée en dessous.

À essayer : amenez le curseur sur **440** puis sur **294**, et comparez avec n'importe quelle valeur intermédiaire, par exemple **350**.

fréquence testée `f_test` = 350 Hz




```
let
    # Un signal court et lisible : la + ré, sur 40 ms
    fs = 8192
    N = 512
    t = (0:N-1) ./ fs
    signal = cos.(2π * 440 .* t) .+ cos.(2π * 293.7 .* t)

    # L'onde de référence à la fréquence que l'on teste
    référence = cos.(2π * f_test .* t)

    # Le produit des deux, point par point
    produit = signal .* référence

    # La mesure : la somme de tous les produits
    mesure = sum(produit)

    p1 = plot(t, signal, label = "signal (la + ré)", color = :steelblue,
        legend = :topright, ylabel = "amplitude")
    p2 = plot(t, référence, label = "référence à $(f_test) Hz", color = :darkorange,
        legend = :topright, ylabel = "amplitude")
    p3 = plot(t, produit, label = "produit", color = :seagreen,
        legend = :topright, xlabel = "temps (s)", ylabel = "amplitude")
    hline!(p3, [0], color = :black, linewidth = 0.5, label = nothing)

```

```
    plot(p1, p2, p3, layout = (3, 1), size = (680, 460),
        plot_title = "somme des produits = $(round(mesure, digits = 1))")
end
```

▶ **Qu'observez-vous en balayant le curseur ?**

Pourquoi le nombre complexe, enfin

Il reste un défaut à la recette ci-dessus. Notre référence est un **cosinus**, qui vaut 1 en $t = 0$. Si le signal à détecter est un **sinus** — la même fréquence, mais décalée d'un quart de période — il est en permanence en décalage avec la référence, les produits se compensent, et la mesure retombe à zéro. On raterait une fréquence pourtant bien présente, uniquement à cause de son **déphasage**.

La parade est élégante : on teste le cosinus **et** le sinus en même temps, en utilisant comme référence l'exponentielle complexe

$$e^{-i2\pi ft} = \cos(2\pi ft) - i\sin(2\pi ft)$$

La **partie réelle** du résultat mesure la ressemblance avec le cosinus, la **partie imaginaire** celle avec le sinus. Le **module** $|X|$ combine les deux et donne l'amplitude de la fréquence *quel que soit son déphasage*. C'est pour cela qu'on trace presque toujours $|X|$ — et c'est aussi pour cela que les nombres complexes du cours n°1 reviennent ici.

1.2 La définition

On vient de construire la formule à la main. Il ne reste qu'à l'écrire proprement.

Notre signal n'est pas une fonction continue : c'est une **liste de N nombres $x_0, x_1, \ldots, x_{N-1}$**, les échantillons. « Multiplier puis additionner » s'écrit donc avec un $\sum$, et on obtient la **transformée de Fourier discrète** (*Discrete Fourier Transform*, DFT) — celle que l'ordinateur calcule réellement.

La transformée de Fourier discrète et son inverse :

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

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

Lisons la première formule **en la comparant à ce qu'on vient de faire au §1.1** :

Dans la formule	Ce qu'on a fait à la main
x_n	`le signal`
$e^{-i2\pi\frac{k}{N}n}$	`l'onde de référence`
`le produit` $x_n \cdot e^{\cdots}$	`« je multiplie point par point »`
`le` $\sum_{n=0}^{N-1}$	`« j'additionne tout »`
`l'indice` k	`« la fréquence que je teste »`

C'est la même chose. La formule ne dit rien de plus que le slider que vous venez de manipuler : pour chaque fréquence k, elle mesure la ressemblance entre le signal et une oscillation pure.

La deuxième ligne (l'inverse) est quasiment identique — seuls le signe de l'exposant et le facteur $1/N$ changent. C'est ce qui nous permettra de **revenir** au §3.

Et si le signal était une vraie fonction continue $f(t)$ plutôt qu'une liste de points ? La somme devient une intégrale, et on obtient la **transformée de Fourier continue** — c'est celle que vous manipulerez à la main aux séances d'exercices :

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

$\sum$ devient $\int$, x_n devient $f(t)$: **c'est le même objet, la même idée**. Une version compte des échantillons, l'autre balaie un continuum.

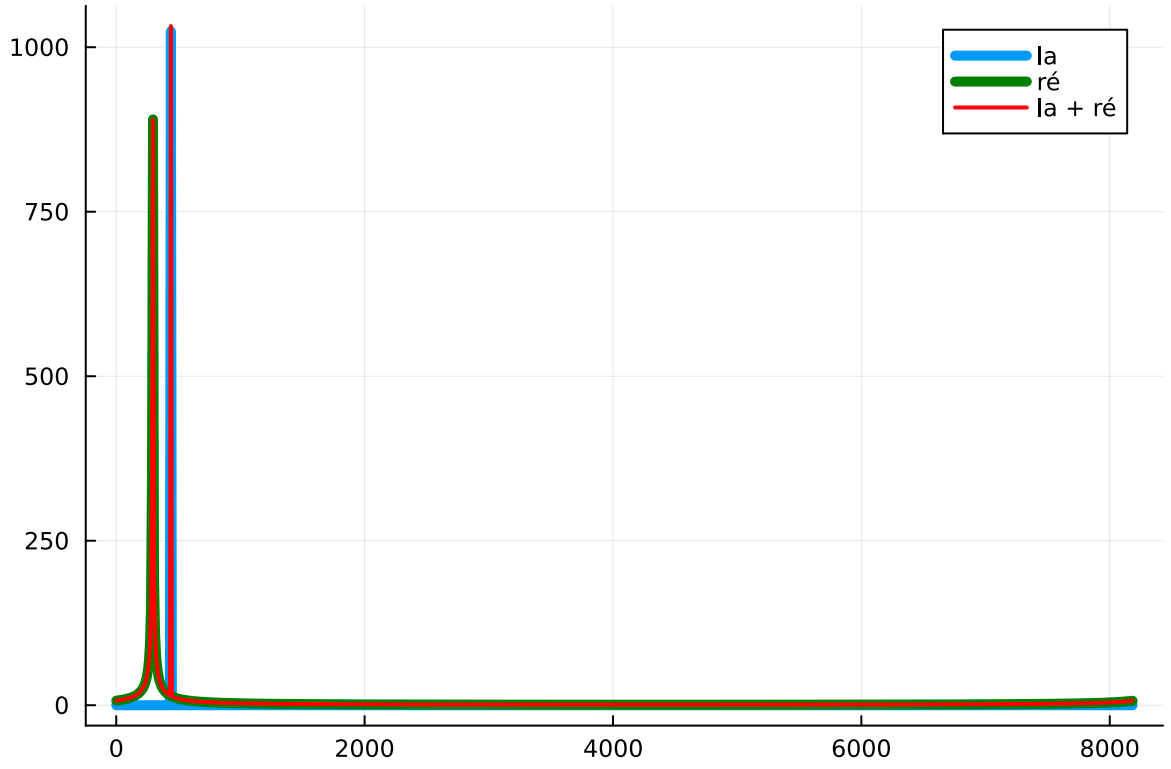
1.3 Lire un spectre

Appliquons `fft` à nos trois signaux. Le résultat s'appelle le **spectre** : en abscisse les fréquences, en ordonnée « combien il y en a ».

Le menu déroulant permet de basculer entre le module, la partie réelle et la partie imaginaire — c'est le module (`abs`) qui est le plus parlant, pour la raison expliquée à la fin du §1.1.

Signal fréquentiel

abs




```
let
    f = (0:length(la)-1) .* (échantillonnage / length(la))
    sel = Dict(
        "abs" => abs,
        "real" => real,
        "imag" => imag,
    )[freq_sel]
    plot(f, sel.(fft(la)), label = "la", linewidth=5)
    plot!(f, sel.(fft(ré)), label = "ré", linewidth=5, color = :green)
    plot!(f, sel.(fft(la + ré)), label = "la + ré", linewidth=2, color = :red)
end
```

Voilà le résultat qu'on cherchait depuis le début du cours. Le signal `la + ré`, illisible dans le temps, devient ici **deux pics nets**, aux bons endroits. Et remarquez que la courbe rouge (`la + ré`) est exactement la superposition des deux autres : **la transformée est linéaire**, la somme des signaux donne la somme des spectres.

Ce que l'œil ne savait pas lire sur la courbe temporelle se lit maintenant d'un coup d'œil.

Intuition Le signal pur "la" est $f_{\text{la}}(t) = \exp(2\pi 440 t i) = \cos(2\pi 440 t) + i\sin(2\pi 440 t)$. 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.

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

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

1.4 Le pont avec les séances d'exercices

Aux **séances d'exercices**, vous calculez des transformées **à la main**, sur des signaux idéaux : l'échelon $u(t)$, l'impulsion de Dirac $\delta(t)$, la fonction $\mathbf{sinc}$. Vous intégrez et vous obtenez une formule close.

Ici, on ne calcule aucune intégrale : on donne N nombres à l'ordinateur et il rend N nombres.

Ce sont les **deux faces du même objet** :

Aux exercices — continu, à la main	Ici — discret, à la machine
$F(\xi) = \int f(t)\, e^{-i2\pi\xi t}\, dt$	$X_k = \sum_n x_n e^{-i2\pi kn/N}$
$f(t)$ définie pour **tout** t	N échantillons $x_0, \dots, x_{N-1}$
ξ parcourt un **continuum**	N fréquences discrètes k
un Dirac $\delta(\xi - f_0)$	**un pic** sur une case du spectre
durée infinie	durée finie $N\Delta t$

Les exercices servent à **comprendre le mécanisme** sur des cas simples ; le notebook sert à **l'appliquer** à de vrais signaux. Quand vous calculerez la transformée de $e^{-at}u(t)$ à la main, vous ferez exactement ce que `fft` fait ici numériquement.

Reste une question très concrète : `fft` rend une liste de nombres, **sans unité**. Comment savoir à quelle fréquence en Hz correspond la case k ? Comprendre cela est indispensables pour lire correctement un spectre.

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

Les deux vecteurs ont la même dimension

C'est le point le plus important, et le plus facile à perdre de vue : `fft` **prend N nombres et rend N nombres**.

$$\underbrace{(x_0, \dots, x_{N-1})}_{\text{temporel, } N \text{ valeurs}} \quad \longleftrightarrow \quad \underbrace{(X_0, \dots, X_{N-1})}_{\text{fréquentiel, } N \text{ valeurs}}$$

Aucune information n'est créée ni perdue : on change de **base** dans un espace de dimension N. Ni la fonction `fft`, ni les vecteurs eux-mêmes ne portent d'unité — *c'est à nous de savoir à quelle fréquence correspond chaque case*. Deux nombres suffisent à tout reconstituer : Δt (ou f_e) et N.

	Temporel	Fréquentiel
`Nombre de valeurs`	N	N
`Pas`	$\Delta t = 1/f_e$ [s]	$\Delta\xi = f_e/N = 1/(N\Delta t)$ [Hz]
`Échantillons`	$t_n = n\,\Delta t,\ n = 0, \dots, N-1$	$\xi_k = k\,\Delta\xi,\ k = 0, \dots, N-1$
`Intervalle couvert`	$[0,\ N\Delta t)$ [s]	$[0,\ f_e)$ [Hz]
`Largeur`	$N\Delta t$ `(la durée)`	$N\Delta\xi = f_e$

Les lignes « pas » et « largeur » sont **croisées** :

$$\underbrace{\Delta t}_{\text{pas en temps}} \longrightarrow \underbrace{f_e = 1/\Delta t}_{\textbf{largeur en fréquence}}, \qquad \underbrace{N\Delta t}_{\text{largeur en temps}} \longrightarrow \underbrace{\Delta\xi = 1/(N\Delta t)}_{\textbf{pas en fréquence}}$$

En une phrase : **la finesse d'un domaine fixe l'étendue de l'autre.** Échantillonner plus finement (petit Δt) élargit la bande de fréquences accessible ; enregistrer plus longtemps (grand $N\Delta t$) affine la résolution. Ce sont deux réglages **indépendants**.

A retenir

Avec f_e échantillons par seconde en temporel, on peut mesurer des fréquences entre 0 et f_e. Donc, pour mesurer des hautes fréquences dans le signal (des oscillations rapides), il faut augmenter la fréquence d'échantillonage (logique). Autrement dit, **l'inverse du pas en temporel donne la largeur du domaine en fréquences**.

Cette observation est **symmétrique**. L'écart entre deux échantillons en fréquences est $\Delta_\xi = \frac{1}{N\Delta t}$, dont l'inverse $N\Delta t$ donne la durée du signal en temporel. Donc, pour augmenter la précision pour détecter des fréquences proches, il faut augmenter la durée du signal.

Exemple chiffré, à manipuler

Les deux curseurs ci-dessous sont ceux du §0. Changez-les et regardez **quelle colonne bouge** : c'est le meilleur moyen d'ancrer la règle croisée.

f_e = 1000 Hz — N = 8 échantillons

	Temporel	Fréquentiel
Nombre de valeurs	8	8
Pas	Δt = 1.0 ms	Δξ = 125.0 Hz
Intervalle	[0, 8.0) ms	[0, 1000) Hz
Lecture signée	–	[-500, 500) Hz
Nyquist $f_e/2$	–	500.0 Hz

Voici, pour ces réglages, la fréquence associée à **chaque case k** du spectre. La deuxième ligne est la lecture brute $k\,\Delta\xi$; la troisième est la **lecture signée**, celle qu'il faut avoir en tête.

k	0	1	2	3	4	5	6	7
$k\,\Delta\xi$ (Hz)	0.0	125.0	250.0	375.0	500.0	625.0	750.0	875.0
lecture signée (Hz)	0.0	125.0	250.0	375.0	500.0	-375.0	-250.0	-125.0

▸ **Pourquoi la seconde moitié du tableau est-elle lue en négatif ?**

Vérification. Avec $f_e = 1000$ Hz et $N = 8$, la résolution vaut $\Delta\xi = 125$ Hz. Le signal $\cos(2\pi \cdot 125\,t)$ tombe donc *exactement* sur la case $k = 1$. Son spectre doit être nul partout ailleurs — sauf en $k = 7$, le jumeau symétrique à -125 Hz.

k	0	1	2	3	4	5	6	7
$\lvert X_k \rvert$	0.0	4.0	0.0	0.0	0.0	0.0	0.0	4.0

```
let
    fe = 1000
    N  = 8
    Δt = 1 / fe
    t  = (0:N-1) .* Δt
    x  = cos.(2π * 125 .* t)

    X = round.(abs.(fft(x)), digits = 10)

    Markdown.parse(
        "| ``k`` | " * join(string.(0:N-1), " | ") * " |\n" *
        "|:--|" * repeat(":--:|", N) * "\n" *
        "| ``\\|X_k\\|`` | " * join(string.(X), " | ") * " |"
    )
end
```

Exactement comme annoncé : deux pics, en $k = 1$ et $k = 7$, et des zéros partout ailleurs. La machine confirme le calcul à la main.

2. Travailler : ce qu'on y gagne

Nous savons aller dans le domaine des fréquences. **Pourquoi y aller ?**

Parce que certaines opérations qui sont difficiles, voire impossibles, dans le domaine temporel y deviennent **triviales**. On va en voir quatre, de la plus parlante à la plus surprenante :

- **2.1** nettoyer un son de son bruit — en une ligne ;
- **2.2** la convolution, qui devient une simple multiplication ;
- **2.3** flouter et analyser une image ;
- **2.4** multiplier des polynômes (et de très grands nombres) bien plus vite.

2.1 Nettoyer un son

Voici une situation très concrète. On dispose d'un enregistrement — ici un accord de trois notes — **pollué par un sifflement aigu** : une fréquence parasite que l'on veut supprimer sans abîmer la musique.

Dans le domaine temporel, le problème paraît désespéré : le sifflement est **mélangé au signal utile à chaque instant**. Il n'y a aucun morceau de la courbe qu'on pourrait couper — le parasite est *partout*, additionné à la musique en chaque point.

Construisons le signal et écoutons le dégât.

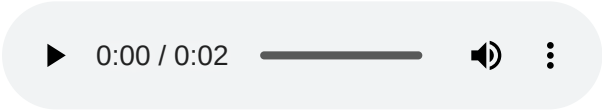
```
begin
    fs_audio = 8192              # fréquence d'échantillonnage (Hz)
    durée = 2                    # secondes
    t_audio = range(0, stop = durée, length = fs_audio * durée)

    # Un accord de do majeur : do - mi - sol
    accord = 0.5 * (
        cos.(2π * 261.6 .* t_audio) .+      # do
        cos.(2π * 329.6 .* t_audio) .+      # mi
        cos.(2π * 392.0 .* t_audio)         # sol
    )

    # Le parasite : un sifflement aigu à 3000 Hz
    sifflement = 0.6 * cos.(2π * 3000 .* t_audio)

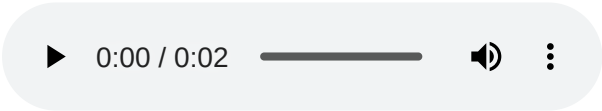
    accord_bruité = accord .+ sifflement
end;
```

L'accord propre — ce qu'on voudrait entendre :



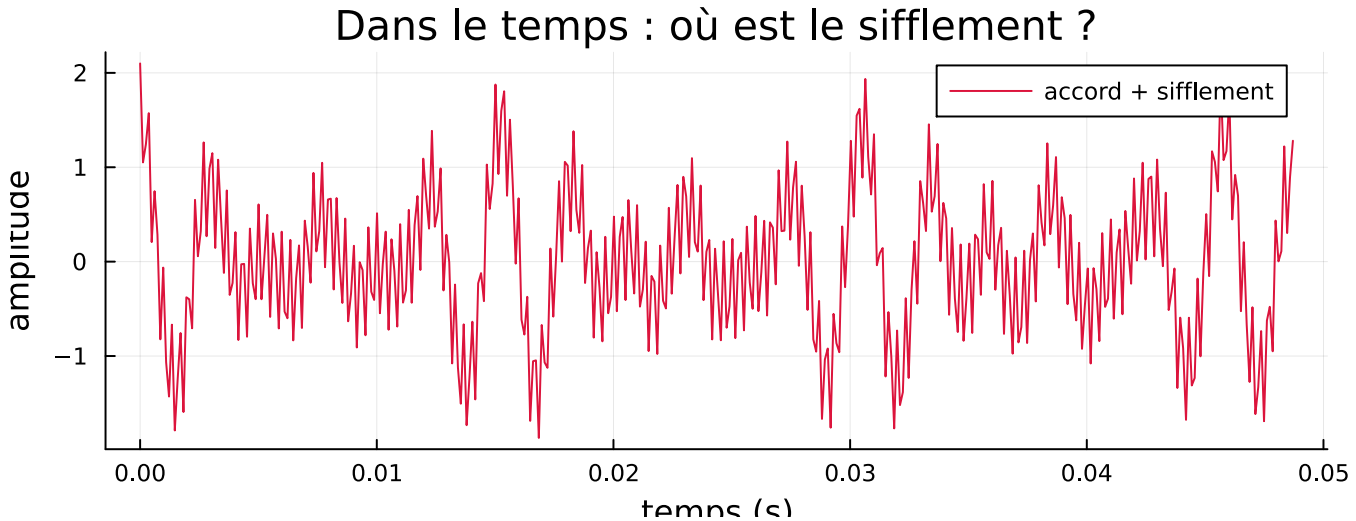

```
play_sound(accord ./ maximum(abs.(accord)), fs_audio)
```

L'accord pollué — ce qu'on a réellement enregistré. Le sifflement est désagréable :




```
play_sound(accord_bruité ./ maximum(abs.(accord_bruité)), fs_audio)
```

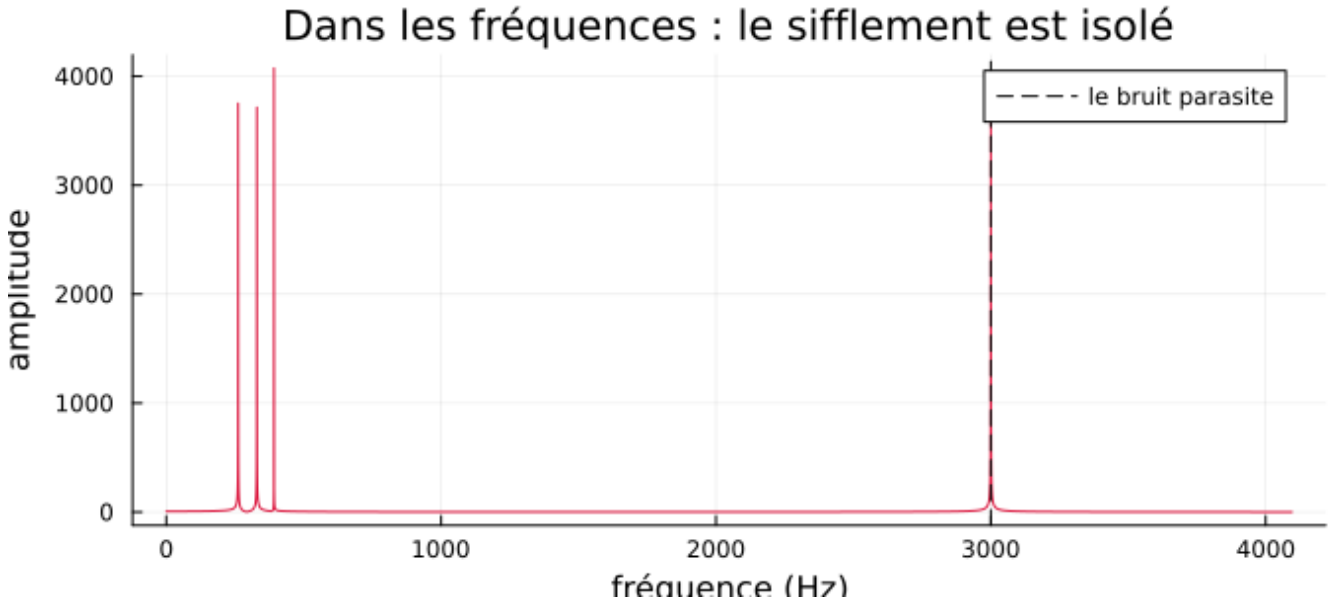
Regardons d'abord le signal **dans le temps**. Voici quelques millisecondes, zoomées :




```
let
    n = 400
    plot(t_audio[1:n], accord_bruité[1:n],
        label = "accord + sifflement", color = :crimson,
        xlabel = "temps (s)", ylabel = "amplitude",
        title = "Dans le temps : où est le sifflement ?", size = (680, 260))
end
```

Impossible de désigner le parasite sur ce graphe : il est **fondu dans la courbe**. On voit bien de petites oscillations rapides greffées sur les grandes, mais il n'existe aucun intervalle de temps qu'on pourrait supprimer sans détruire aussi la musique.

Passons maintenant dans le domaine des fréquences.




```
let
    N = length(accord_bruité)
    spectre = abs.(fft(accord_bruité))
    freqs = (0:N-1) .* (fs_audio / N)
    moitié = 1:div(N, 2)          # on n'affiche que les fréquences positives

    plot(freqs[moitié], spectre[moitié],
        label = nothing, color = :crimson,
        xlabel = "fréquence (Hz)", ylabel = "amplitude",
        title = "Dans les fréquences : le sifflement est isolé",
        size = (680, 300))
    vline!([3000], color = :black, linestyle = :dash, linewidth = 1,
        label = "le bruit parasite")
end
```

Voilà toute la magie de la transformée de Fourier.

À gauche, trois pics serrés : ce sont le *do*, le *mi* et le *sol*, la musique qu'on veut garder. Loin à droite, isolé, un quatrième pic : le sifflement à 3000 Hz.

Ce qui était **inextricablement mélangé dans le temps est parfaitement rangé dans les fréquences**.

L'opération de nettoyage du bruit tient maintenant en une ligne : *mettre à zéro tout ce qui est au-dessus d'un certain seuil de fréquence*.

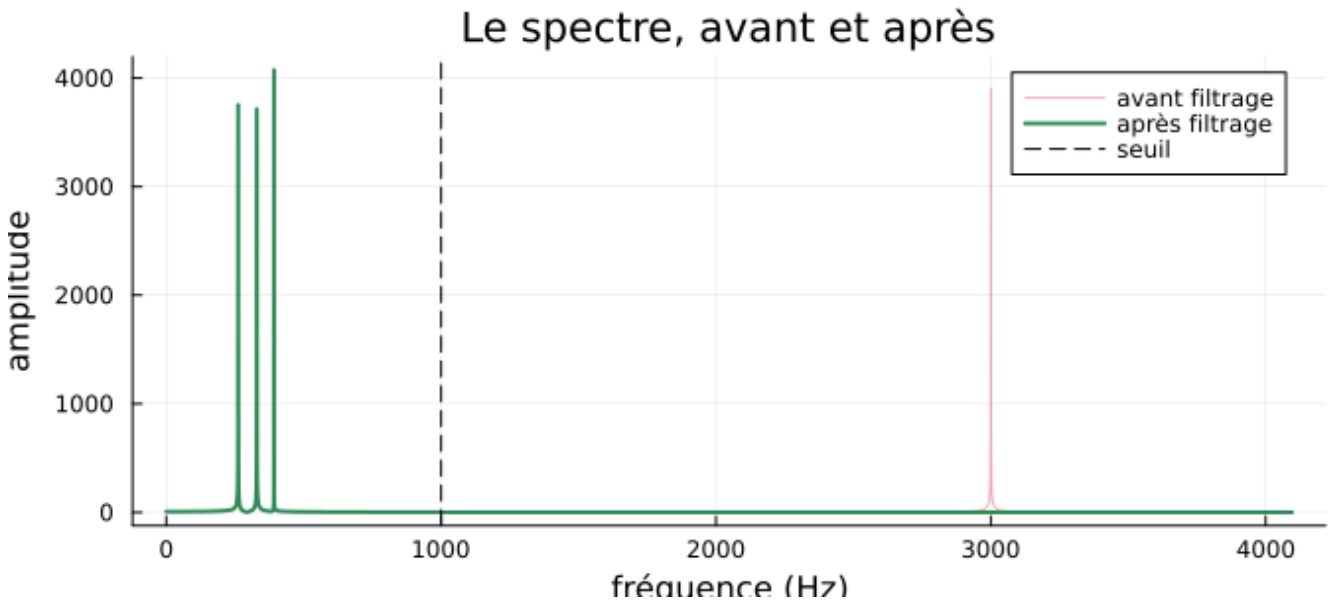
seuil de coupure = 1000 Hz

```julia
accord_filtré = let
    N = length(accord_bruité)
    freqs = (0:N-1) .* (fs_audio / N)

    # Le spectre du signal pollué
    Y = fft(accord_bruité)

    # ── LE FILTRE, EN UNE LIGNE ──
    # Toute fréquence au-dessus du seuil est mise à zéro.
    # Attention à la symétrie : une fréquence f apparaît en f ET en fs - f.
    Y[(freqs .> seuil) .& (freqs .< fs_audio - seuil)] .= 0

    # Retour dans le temps
    real.(ifft(Y))
end;
```




```julia
let
    N = length(accord_filtré)
    freqs = (0:N-1) .* (fs_audio / N)
    moitié = 1:div(N, 2)

    plot(freqs[moitié], abs.(fft(accord_bruité))[moitié],
        label = "avant filtrage", color = :crimson, alpha = 0.35)
    plot!(freqs[moitié], abs.(fft(accord_filtré))[moitié],
        label = "après filtrage", color = :seagreen, linewidth = 2,
        xlabel = "fréquence (Hz)", ylabel = "amplitude",
        title = "Le spectre, avant et après", size = (680, 300))
    vline!([seuil], color = :black, linestyle = :dash, label = "seuil")
end
```

Le moment de vérité : écoutez le résultat. Le sifflement a disparu, l'accord est intact.

0:00 / 0:02

```
play_sound(accord_filtré ./ maximum(abs.(accord_filtré)), fs_audio)
```

Jouez avec le curseur du seuil et réécoutez à chaque fois:

- au-dessus de **3000 Hz** : le sifflement revient — on ne coupe plus assez haut ;
- entre **400 et 3000 Hz** : l'accord est propre, le parasite est éliminé ;
- en dessous de **392 Hz** : on commence à manger le *sol*, puis le *mi*, puis le *do*. L'accord s'appauvrit et devient sourd.

Ce curseur, c'est exactement le **bouton des aigus** de votre chaîne hi-fi, et c'est la brique de base de tout égaliseur audio.

▸ **Pourquoi ne pouvait-on pas faire ce nettoyage directement sur le signal temporel ?**

Le schéma à retenir

Ce que nous venons de faire est le motif universel du traitement du signal :

$$\text{signal} \xrightarrow{\texttt{fft}} \text{spectre} \xrightarrow{\text{opération facile}} \text{spectre modifié} \xrightarrow{\texttt{ifft}} \text{signal traité}$$

On fait un détour par le monde des fréquences **parce que le travail y est plus simple**, puis on revient. Les sections qui suivent sont toutes des variantes de ce même aller-retour.

2.2 La convolution devient une multiplication

Le filtre du §2.1 a un nom dans le domaine temporel : c'est une **convolution**. C'est l'opération qui décrit ce que fait tout système physique à un signal — un haut-parleur, une salle qui résonne, une lentille photo légèrement floue, un circuit électronique.

Pour chaque point de sortie, on fait une **somme de produits décalés**.

Définitions.

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:

commutatif	$f * g = g * f$
associatif	$(f * g) * h = f * (g * h)$
distributif	$f * (g + h) = f * g + f * h$
conjugué	$\overline{f * g} = \overline{f} * \overline{g}$

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:

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

Cette définition a un coût bien réel. Pour chacun des n points de sortie, il faut faire une somme sur n termes : n^2 **opérations**.

Et c'est ici qu'intervient le résultat le plus utile de tout le cours — le **théorème de convolution** :

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

En clair : **convoluer dans le temps, c'est multiplier point par point dans les fréquences**. Cette double somme pénible devient une simple multiplication terme à terme.

C'est exactement ce que nous avons fait sans le dire au §2.1 : mettre des cases du spectre à zéro *est* une multiplication du spectre (par un masque de 0 et de 1), donc *est* une convolution du signal. Nous avons convolué sans jamais écrire la double somme.

Les sections **2.3** et **2.4** montrent le gain de vitesse que cela représente, sur des images puis sur des polynômes.

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

2.3 Application : les images

Une image est un signal comme un autre, simplement en **deux dimensions** : au lieu d'une valeur par instant, une valeur par pixel. Tout ce qu'on vient de voir s'y applique.

Convoluer une image avec un petit motif — qu'on appelle un **noyau** (*kernel*) — permet de la flouter, de la nettoyer, ou d'en détecter les contours. C'est l'opération de base du traitement d'images… et la brique élémentaire des réseaux de neurones convolutifs.

Illustration : convolution d'images

```
mandrill =
```



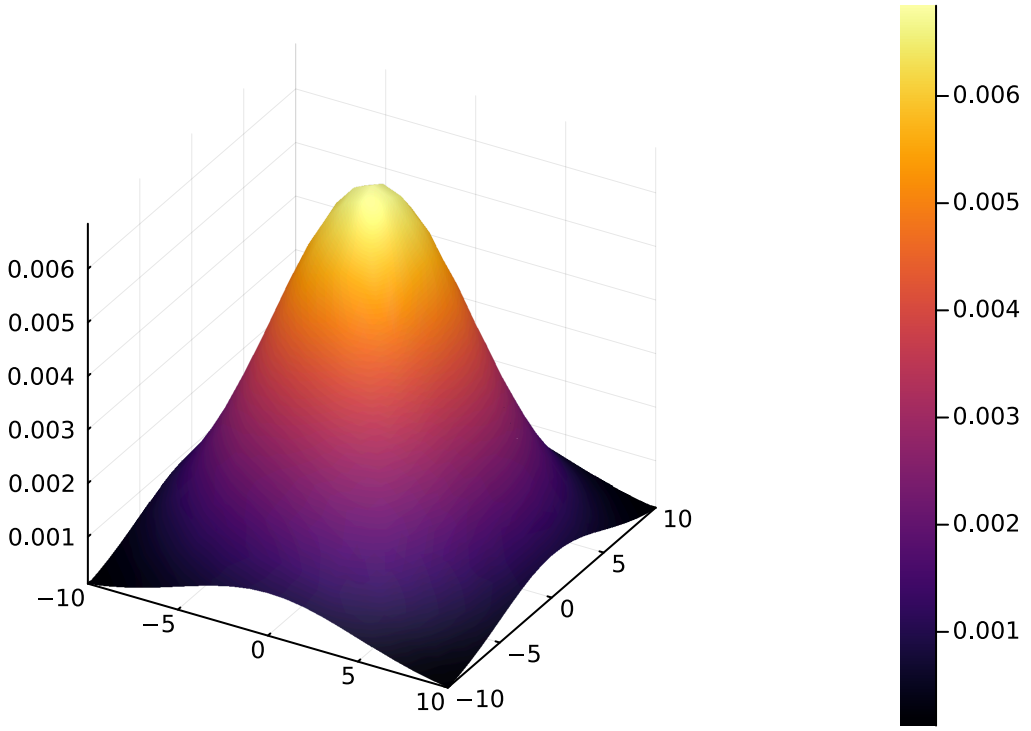
```
1 mandrill = testimage("mandrill")
```

Gaussian Kernel

Le noyau gaussien ci-dessous est une petite « bosse ». Convoluer avec lui revient à **remplacer chaque pixel par une moyenne pondérée de ses voisins** — d'où le flou. Le curseur `d` règle la largeur de la bosse, donc l'intensité du flou.

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

d = 5




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

La fonction `imfilter` calcule la cross-corrélation.



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

La convolution est obtenue avec `reflect`.



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

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

Autre kernels



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

```
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
```

```
convert(AbstractArray, 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) !

Et la vitesse ?

Comparons les deux façons de calculer cette convolution : directement (*FIR*, la double somme) ou en passant par Fourier (*FFT*).

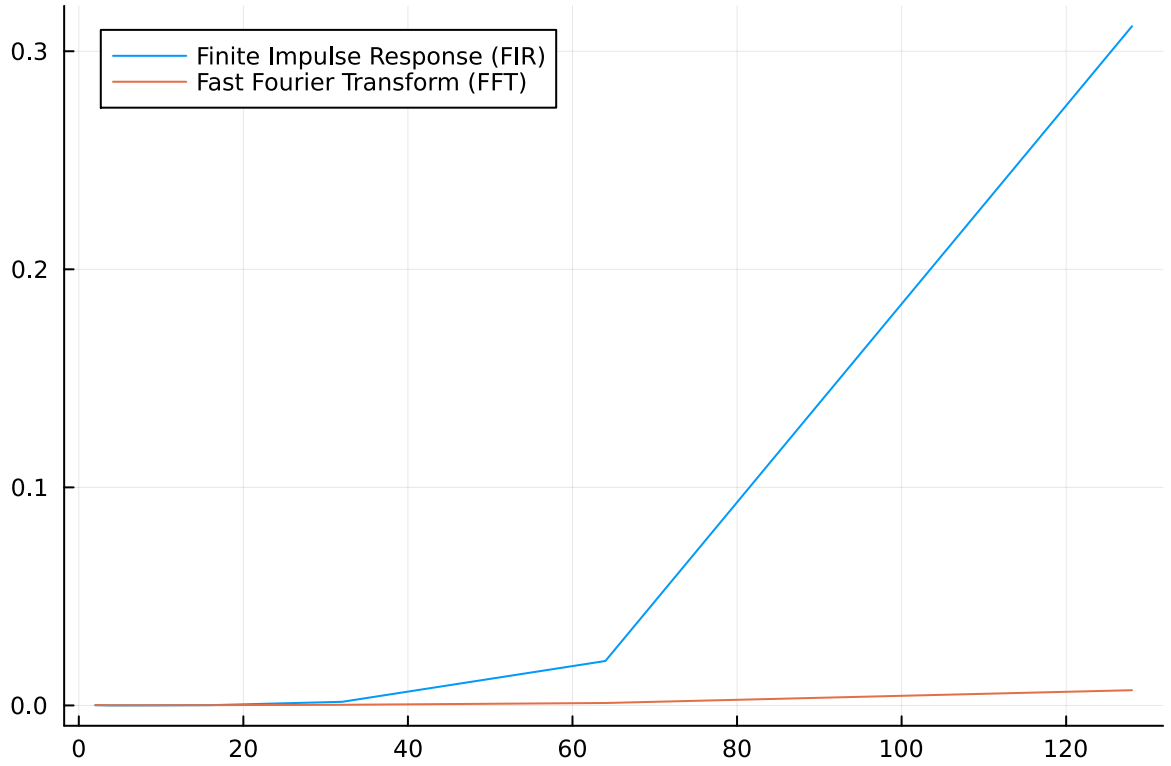
Le résultat est spectaculaire, et il justifie à lui seul l'existence de la section 4 : dès que l'image grandit un peu, la méthode directe s'effondre alors que la méthode par Fourier tient bon.

Nous verrons **pourquoi** la FFT est si rapide au §4 — c'est la dernière pièce du puzzle.

Performance

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

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




```
let
    n = 2 .^ (1:7)
    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
```

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

```
PlutoUI.details(
    md"Quelle est la complexité de FIR ? et de de FFT ?",
    md"""
**FIR**: ``(f \ast g)_{k,l} = \sum_{i=1}^n\sum_{j=1}^n f_{i,j} g_{k - i, l -j}``
Pour une pair ``k,l``, c'est ``n^2`` opérations.
Comme il y a ``n`` valeurs de ``k`` et idem pour ``l``, je dois faire cette double somme ``n^2`` fois.
Donc la complexité totale c'est ``n^4``.

**FFT**: ``\mathcal{F}^{-1}(\mathcal{F}(f) \cdot \mathcal{F}(g))``
* La complexité de ``\mathcal{F}(f)`` est ``n^2\log(n^2)``
* La complexité de ``\mathcal{F}(g)`` est ``n^2\log(n^2)``
* La complexité du produit ``\mathcal{F}(f) \cdot \mathcal{F}(g)`` est ``n^2``.
* La complexité de la FT inverse ``\mathcal{F}^{-1}(\ldots)`` est ``n^2\log(n^2)``
Au total, ``3n^2\log(n^2) + n^2``. Notons que ``\log(n^2) = 2\log(n)``.
En ignorant les constantes, on a donc ``n^2\log(n) + n^2``.
""",
)
```

2.4 Application : multiplier des polynômes

Dernier exemple, et le plus inattendu — il n'y a ni son ni image ici, juste de l'algèbre.

Quand on multiplie deux polynômes, les coefficients du résultat sont obtenus par… une **convolution** des coefficients de départ. Chaque coefficient du produit est une somme de produits croisés : exactement la formule du §2.2.

En effet, soit $p(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n$ et $q(x) = b_0 + b_1 x + b_2 x_2 + \cdots + b_n x^n$, on a $c_k = \sum_i a_i b_{k-i}$

Donc le même truc s'applique : $\mathcal{F}$, multiplication terme à terme, $\mathcal{F}^{-1}$. On passe de n^2 à $n \log n$.

Illustration : le produit de polynômes

```
deg = 4
```

```
deg = 4
```

```
p = 0.4928663846517609 + 0.500754647435169·x + 0.918826202979902·x^2 + 0.24779959901156823·x^3
```

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

q = 0.7242783369547477 + 0.14397414385941143·x + 0.0866257009847562·x^2 + 0.3903584224540142·x^3

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

0.3569724454164764 + 0.43364575903402586·x + 0.78027653195511·x^2 + 0.5475358641484979·x^3 + 0.3107444932420005·x^4 + 0.38013736107276497·x^5 + 0.09673066055489307·x^6

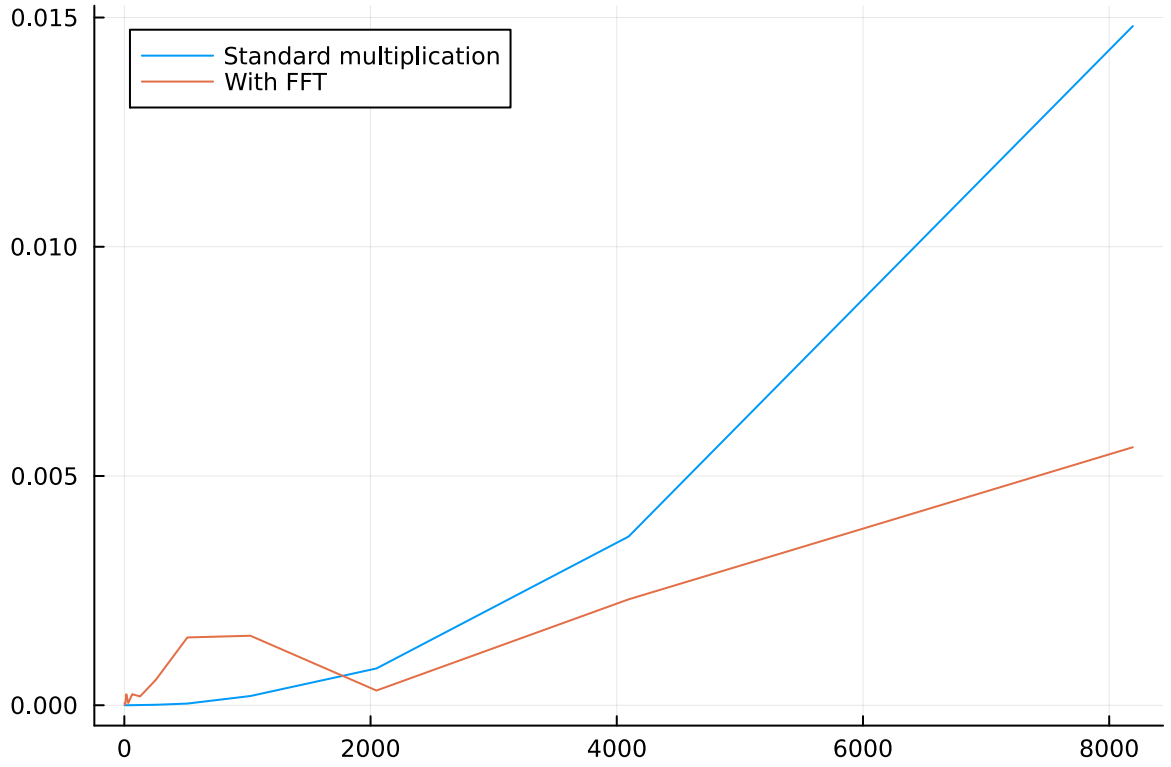
```
p * q
```

0.3569724454164764 + 0.4336457590340258·x + 0.78027653195511·x^2 + 0.5475358641484979·x^3 + 0.3107444932420005·x^4 + 0.38013736107276497·x^5 + 0.09673066055489311·x^6

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

poly_time (generic function with 1 method)

```
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 ?**

3. Retour : reconstruire, et les pièges

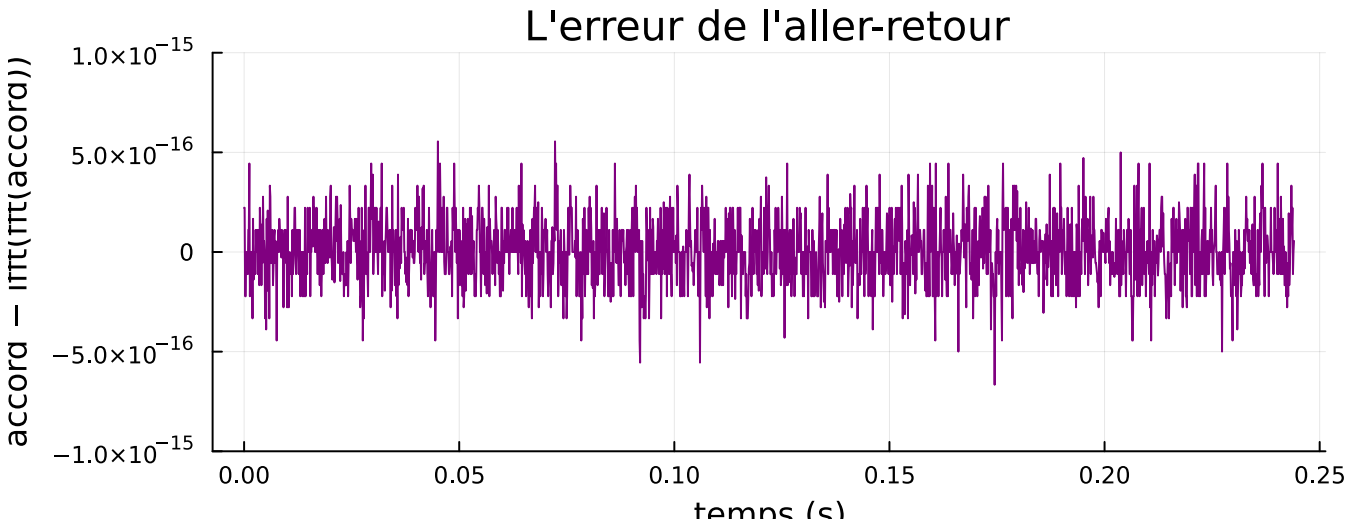
Nous savons aller dans les fréquences (§1) et y travailler (§2). Reste la troisième étape, celle qui rend tout le reste utilisable : **revenir**.

Deux questions se posent, et elles sont de nature très différente :

- **3.1** L'aller-retour est-il **exact** ? Est-ce qu'on perd de l'information en chemin ?
- **3.2–3.3** Que se passe-t-il quand on n'a **pas assez d'échantillons** ? C'est là qu'apparaît le piège le plus célèbre du traitement du signal.

3.1 L'aller-retour ne perd rien

Question légitime : en passant par les fréquences, ne perd-on pas un peu du signal en route ? Vérifions-le directement — prenons notre accord, faisons l'aller-retour `fft` puis `ifft`, et comparons au signal de départ.




```
let
    aller_retour = real.(ifft(fft(accord)))
    différence = accord .- aller_retour

    plot(t_audio[1:2000], différence[1:2000],
        label = nothing, color = :purple,
        xlabel = "temps (s)", ylabel = "accord - ifft(fft(accord))",
        title = "L'erreur de l'aller-retour",
        ylim = (-1e-15, 1e-15), size = (680, 260))
end
```

Lisez bien **l'échelle de l'axe vertical** : 10^{-15}. La différence est **nulle** — ce qu'on voit n'est que le bruit d'arrondi de l'arithmétique à virgule flottante, à la limite de la précision de la machine.

$$\mathcal{F}^{-1}\big(\mathcal{F}(x)\big) = x$$

Conclusion capitale : le spectre contient exactement la même information que le signal. Rien n'est perdu, rien n'est ajouté. Ce ne sont pas deux objets différents, mais **deux points de vue sur le même objet** — comme décrire un lieu par ses coordonnées GPS ou par son adresse postale.

C'est ce qui rend légitime tout le §2 : si l'aller-retour n'était qu'approximatif, filtrer un son abîmerait la musique à chaque passage. Ici, seul ce qu'on a **délibérément** modifié dans le spectre change.

La procédure **aller** → **travailler** → **revenir** est maintenant complète.

3.2 Le piège : le repliement spectral

Tout ce qui précède supposait une chose qu'on n'a jamais questionnée : que nos N échantillons décrivent **fidèlement** le signal continu d'origine.

Que se passe-t-il si on n'en prend pas assez ?

La réponse est le phénomène du **repliement spectral** (*aliasing*). Il faut d'abord comprendre une propriété surprenante de la DFT : elle considère **implicitement** que le signal se répète à l'infini.

```
md"""
## 3.2 Le piège : le repliement spectral

Tout ce qui précède supposait une chose qu'on n'a jamais questionnée : que nos
``N`` échantillons décrivent **fidèlement** le signal continu d'origine.

Que se passe-t-il si on n'en prend pas assez ?

La réponse est le phénomène du **repliement spectral** (*aliasing*). Il faut
d'abord comprendre une propriété surprenante de la DFT : elle considère
**implicitement** que le signal se répète à l'infini.
"""
```

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

Cette périodicité cachée a une conséquence directe. Si on réduit le nombre de points du spectre, le signal temporel devient plus court — et **ce qui dépasse ne disparaît pas : il se replie et vient s'additionner par-dessus le début**.

L'illustration ci-dessous le montre sur une somme de deux gaussiennes. Jouez avec `ratio` : c'est le facteur de sous-échantillonnage.

Le mécanisme du repliement.

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.

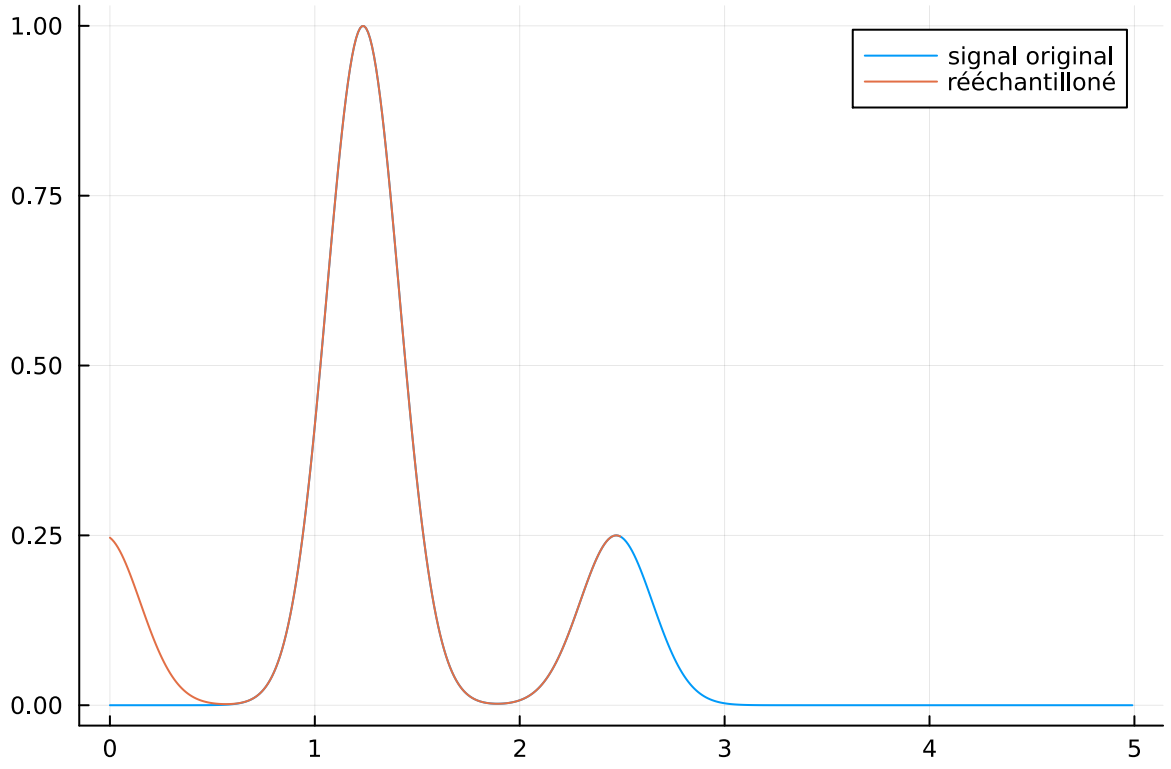
▶ **Que va-t-il advenir de la deuxième moitié du signal ?**

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




```
let
    N = 512
    d = ratio
    x = range(0, stop = t_max - t_max / N, length = N)
    gauss(x) = exp(-16(x-centre)^2)/4 + exp(-16(x-centre/2)^2)
    plot(x, gauss, label = "signal original")
    X = fft(gauss.(x))
    Xs = X[1:d:N]
    half = div(N, 2d)
    xs = real.(ifft(Xs))
    plot!(x[1:length(xs)], xs, label = "rééchantilloné")
end
```

Exemple : Deux cosinus qui se confondent

Prenons la somme de deux cosinus, l'un à 1 Hz et l'autre à 11 Hz :

$$x(t) = a\cos(2\pi\cdot 1\cdot t) + b\cos(2\pi\cdot 11\cdot t), \qquad a = 1, \quad b = 3.$$

En continu, ce sont deux composantes bien distinctes. Le spectre a quatre pics : en ± 1 Hz (hauteur $a/2$) et en ± 11 Hz (hauteur $b/2$).

Échantillonnons à $f_e = 10$ Hz, c'est-à-dire aux instants $t_n = n/10$. Pour le second cosinus,

$$\cos\!\left(2\pi\cdot 11\cdot \tfrac{n}{10}\right) = \cos\!\left(2\pi\tfrac{n}{10} + 2\pi n\right) = \cos\!\left(2\pi\tfrac{n}{10}\right),$$

car on ajoute un nombre entier de tours. Les échantillons valent donc

$$x_n = (a+b)\cos\!\left(2\pi\tfrac{n}{10}\right).$$

Ce sont ceux d'**un seul** cosinus à 1 Hz, d'amplitude $a+b = 4$. La DFT ne montre rien à 11 Hz, qui est hors de la fenêtre $[-f_e/2,\ f_e/2] = [-5,\ 5]$. Elle montre un seul pic en ± 1 Hz, de hauteur $(a+b)/2 = 2$ au lieu de $a/2 = 0.5$: la composante à 11 Hz est tombée dans la même case que la vraie composante à 1 Hz, et les deux se sont **additionnées**.

À partir des échantillons, on ne connaît plus que la somme $a+b$: impossible de distinguer $a = 1,\ b = 3$ de $a = 4,\ b = 0$. Le dégât est **irréversible**.

Faites varier f_e ci-dessous. Le graphique montre le spectre $|X_k|/N$ sur la fenêtre $[-f_e/2,\ f_e/2]$. Les croix indiquent où se trouvent les pics du spectre **continu**, quand ils tombent dans la fenêtre.

```
md"""
### Exemple : Deux cosinus qui se confondent

Prenons la somme de deux cosinus, l'un à 1 Hz et l'autre à 11 Hz :

```math
x(t) = a\cos(2\pi\cdot 1\cdot t) + b\cos(2\pi\cdot 11\cdot t), \qquad a = 1,\quad b = 3.
```

En continu, ce sont deux composantes bien distinctes. Le spectre a quatre pics : en ``\pm 1`` Hz (hauteur ``a/2``) et en ``\pm 11`` Hz (hauteur ``b/2``).

Échantillonnons à ``f_e = 10`` Hz, c'est-à-dire aux instants ``t_n = n/10``. Pour le second cosinus,

```math
\cos\!\left(2\pi\cdot 11\cdot \tfrac{n}{10}\right)
```
```

```
= \cos\!\left(2\pi \tfrac{n}{10} + 2\pi n\right)
= \cos\!\left(2\pi \tfrac{n}{10}\right),
```

car on ajoute un nombre entier de tours. Les échantillons valent donc

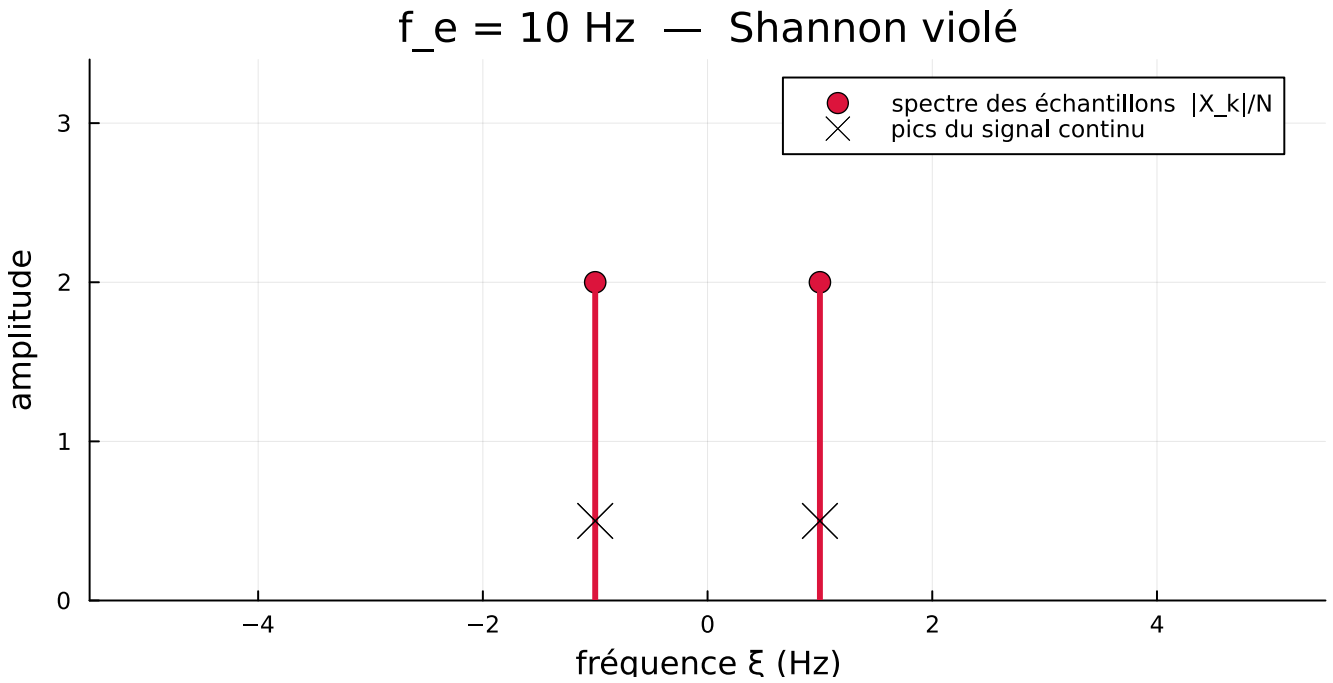
```math
x_n = (a+b)\cos\!\left(2\pi \tfrac{n}{10}\right).
```

Ce sont ceux d'**un seul** cosinus à 1 Hz, d'amplitude ``a+b = 4``. La DFT ne montre rien à 11 Hz, qui est hors de la fenêtre ``[-f_e/2,\ f_e/2] = [-5,\ 5]``. Elle montre un seul pic en ``\pm 1`` Hz, de hauteur ``(a+b)/2 = 2`` au lieu de ``a/2 = 0.5`` : la composante à 11 Hz est tombée dans la même case que la vraie composante à 1 Hz, et les deux se sont **additionnées**.

À partir des échantillons, on ne connaît plus que la somme ``a+b`` : impossible de distinguer ``a = 1,\ b = 3`` de ``a = 4,\ b = 0``. Le dégât est **irréversible**.

Faites varier ``f_e`` ci-dessous. Le graphique montre le spectre ``|X_k|/N`` sur la fenêtre ``[-f_e/2,\ f_e/2]``. Les croix indiquent où se trouvent les pics du spectre **continu**, quand ils tombent dans la fenêtre.
"""

f_e = 10 Hz




```
let
    a, b = 1, 3
    f1, f2 = 1, 11
    durée = 4                                   # secondes → résolution Δξ = 1/durée
    = 0.25 Hz
    N = durée * fe_ab
    t = (0:N-1) ./ fe_ab
    x = a .* cos.(2π * f1 .* t) .+ b .* cos.(2π * f2 .* t)

    # Spectre normalisé, réordonné sur [-f_e/2, f_e/2)
    X = fftshift(fft(x)) ./ N
    ξ = fftshift(fftfreq(N, fe_ab))
    pics = abs.(X) .> 1e-9                      # on n'affiche que les cases non
    nulles

    shannon = fe_ab > 2 * f2
    plot(ξ[pics], abs.(X[pics]),
        seriestype = :sticks, marker = :circle, markersize = 6,
        color = shannon ? :seagreen : :crimson, linewidth = 3,
        label = "spectre des échantillons  |X_k|/N",
        xlabel = "fréquence ξ (Hz)", ylabel = "amplitude",
        xlim = (-fe_ab/2 - 0.5, fe_ab/2 + 0.5), ylim = (0, 3.4),
        legend = :topright, size = (680, 340))
    hline!([0], color = :black, linewidth = 0.5, label = nothing)

    # Pics du spectre continu : ±f1 (hauteur a/2) et ±f2 (hauteur b/2)
    vraies_f = [-f2, -f1, f1, f2]
    vraies_h = [b/2, a/2, a/2, b/2]
    dedans = abs.(vraies_f) .<= fe_ab/2
    scatter!(vraies_f[dedans], vraies_h[dedans],
```

```
        marker = :xcross, markersize = 9, color = :black,
        label = "pics du signal continu")

    title!("f_e = $(fe_ab) Hz  –  Shannon " * (shannon ? "respecté" : "violé"))
```

Règle de Shannon pour ce cas. La plus haute fréquence du signal est $f_{\max} = 11$ Hz : il faut $f_e > 2f_{\max} = 22$ Hz.

❌ Avec $f_e = 10$ Hz, la règle est **violée**. D'après $f_{\text{alias}} = |f - f_e \cdot \text{round}(f/f_e)|$, la composante à 11 Hz est vue à 1.0 Hz, **exactement** comme celle à 1 Hz. Les deux tombent dans la même case et leurs amplitudes s'additionnent : une seule barre de hauteur $(a+b)/2 = 2$ au lieu de $a/2 = 0.5$.

3.3 Shannon–Nyquist : la règle à retenir

Nous venons de voir le repliement **subir**. Existe-t-il une règle pour l'**éviter** ? Oui, et c'est le théorème central de la numérisation du son et de l'image.

Théorème de Shannon–Nyquist

Si un signal ne contient **aucune fréquence au-dessus de $f_{\max}$**, alors il suffit de l'échantillonner à une fréquence

$$f_e \geq 2\, f_{\max}$$

pour pouvoir le reconstruire **parfaitement** à partir de ses échantillons.

D'où vient le facteur 2 ? De la symétrie du spectre découverte au §1.3 : un signal réel à $f_{\max}$ occupe *deux* pics, en $+f_{\max}$ et en $-f_{\max}$. La bande réellement occupée est donc large de $2f_{\max}$, et il faut au moins autant de place pour la loger sans chevauchement.

Si $f_e < 2f_{\max}$, les copies du spectre se chevauchent : les hautes fréquences se **replient** dans les basses et viennent se déguiser en fréquences qu'elles ne sont pas. Le dégât est **irréversible** — une fois les copies additionnées, aucun filtre ne peut les re-séparer.

C'est pour cela que le CD audio échantillonne à **44 100 Hz** : l'oreille humaine monte à environ 20 000 Hz, et $2 \times 20\,000 = 40\,000$, avec une marge de sécurité.

Entendons le repliement. On prend un *la* à 440 Hz et on le échantillonne à des fréquences de plus en plus basses.

Tant que $f_e > 880$ Hz, Shannon est respecté et on entend un *la*. En dessous, la note **change** — on entend une fréquence qui n'a jamais été jouée : son alias.

fréquence d'échantillonnage `fe_test` = 8192 Hz

Shannon respecté ? **oui ✓** — il faudrait $f_e \geq 880$ Hz.

Fréquence réellement entendue : **440.0 Hz** (c'est bien le *la* d'origine)

0:00 / 0:02

```
let
    f_note = 440
    durée = 2
    n = round(Int, fe_test * durée)
    t = range(0, stop = durée, length = n)
    signal = cos.(2π * f_note .* t)

    # La fréquence réellement perçue après repliement :
    # toute fréquence se replie dans la bande [0, fe/2]
    alias = abs(f_note - fe_test * round(f_note / fe_test))
    respecté = fe_test >= 2 * f_note

    md"""
    Shannon respecté ? **$(respecté ? "oui ✓" : "NON ×")** — il faudrait ``f_e \ge 880`` Hz.

    Fréquence réellement entendue : **$(round(alias, digits = 1)) Hz**
    $(respecté ? "(c'est bien le *la* d'origine)" : "(ce n'est plus le *la* ! la note s'est repliée)")

    $(play_sound(signal, fe_test))
    """
end
```

▶ **Pourquoi est-ce si grave ? Ne suffit-il pas de filtrer après coup ?**

4. Le coût : l'algorithme FFT

Il reste une dette à payer. Au §2.3, le graphe de performance montrait la méthode « Fourier » écraser la méthode directe, et nous avons dit : *on verra pourquoi plus tard*. Nous y sommes.

Le problème est le suivant. La DFT du §1.2 demande, pour **chacune** des N fréquences, une somme sur N échantillons : cela fait N^2 opérations. Pour une seconde de son à 44 100 Hz, $N^2 \approx 2 \cdot 10^9$ — beaucoup trop lent pour être utile.

La **FFT** (*Fast Fourier Transform*) ramène ce coût à $N \log N$, soit environ **700 000** opérations au lieu de deux milliards. Ce n'est pas une optimisation de détail : c'est ce qui rend possible l'audio numérique, la compression JPEG, l'imagerie médicale et la téléphonie mobile. L'algorithme figure au top 10 des algorithmes du 20e siècle.

L'idée en une phrase : la matrice de la DFT est bourrée de **redondances**, et en les exploitant on peut ramener un problème de taille N à deux problèmes de taille $N/2$. C'est un *diviser pour régner*, et c'est ce que nous allons découvrir en regardant la matrice.

4.1 L'algorithme *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. En posant $z_N = e^{-i2\pi/N}$, la DFT s'écrit $X = F_N x$ avec

$$F_N = \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & z_N & z_N^2 & \cdots & z_N^{N-1} \\ 1 & z_N^2 & z_N^4 & \cdots & z_N^{2(N-1)} \\ \vdots & & & & \vdots \\ 1 & z_N^{N-1} & z_N^{2(N-1)} & \cdots & z_N^{(N-1)^2} \end{bmatrix}, \qquad (F_N)_{k,n} = z_N^{kn}.$$

C'est aussi l'évaluation du polynôme $\sum_n x_n z^n$ aux N racines N-ièmes de l'unité.

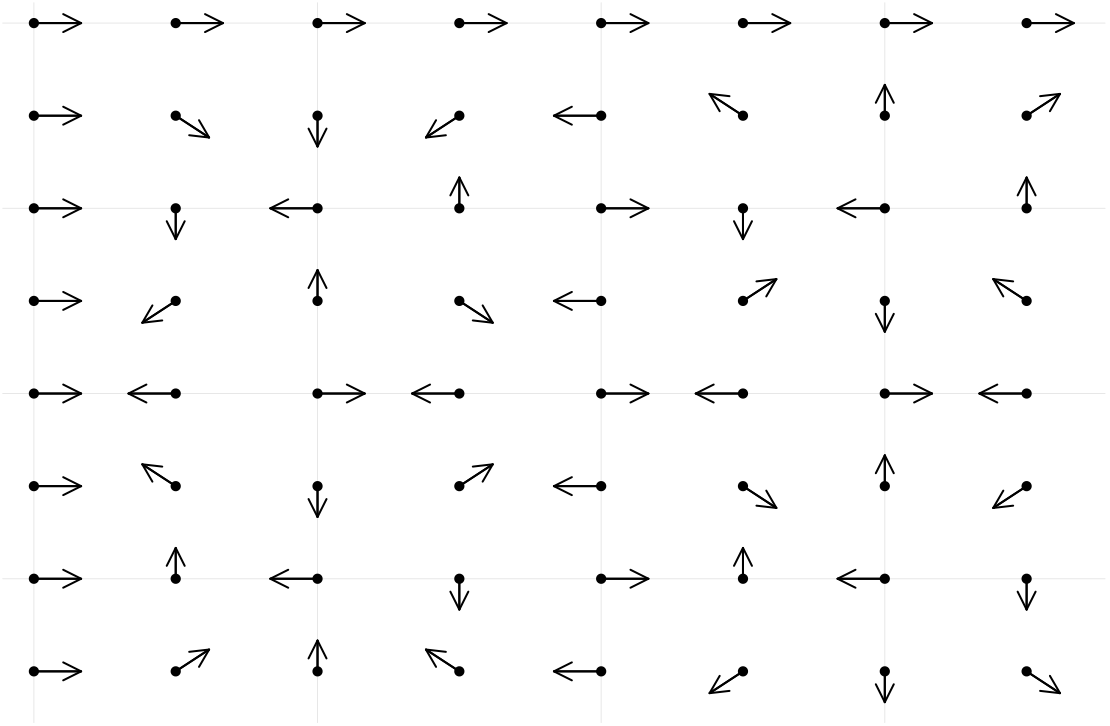
Un produit matriciel a complexité $\Omega(n^2)$. Ça ne nous donne pas une complexité de $\Omega(n \log n)$, il va falloir utiliser la structure assez spéciale de cette matrice…

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
```

▸ **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
```

```
1 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
```

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

```
true
```

```
1 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
```

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

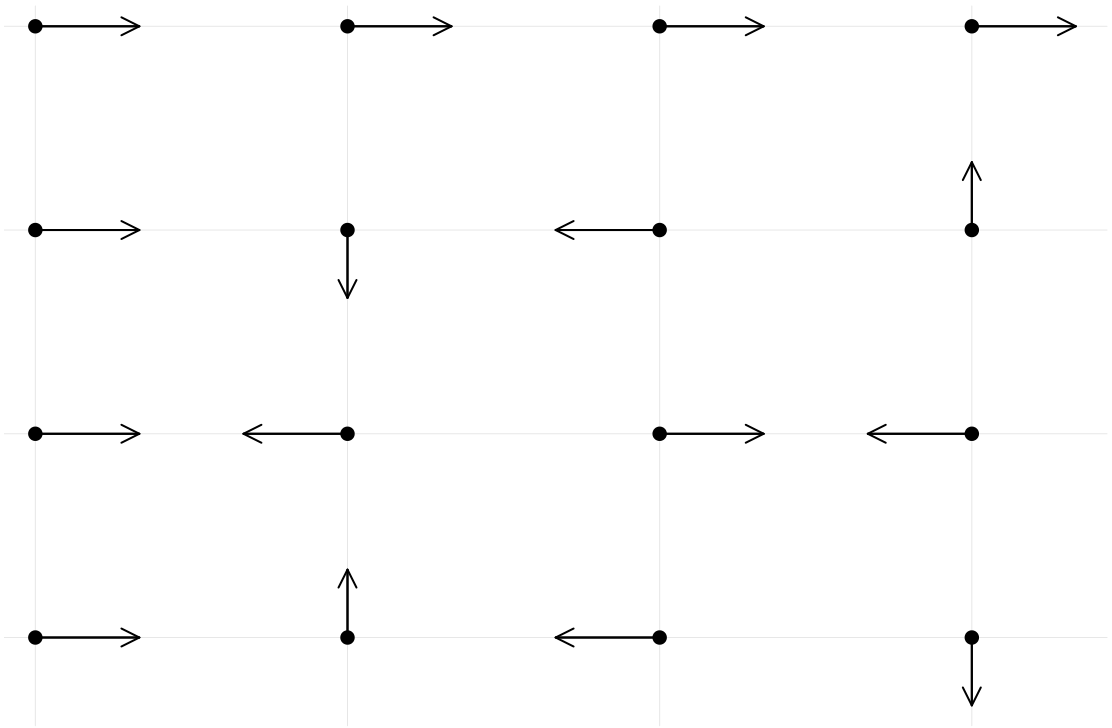
```
partie_rouge = 4×4 Matrix{ComplexF64}:
                  -1.0-0.0im            -1.0-0.0im       …        -1.0-0.0im
               -0.707107+0.707107im   0.707107+0.707107im     -0.707107-0.707107im
                     0.0+1.0im             0.0-1.0im                0.0-1.0im
                0.707107+0.707107im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

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

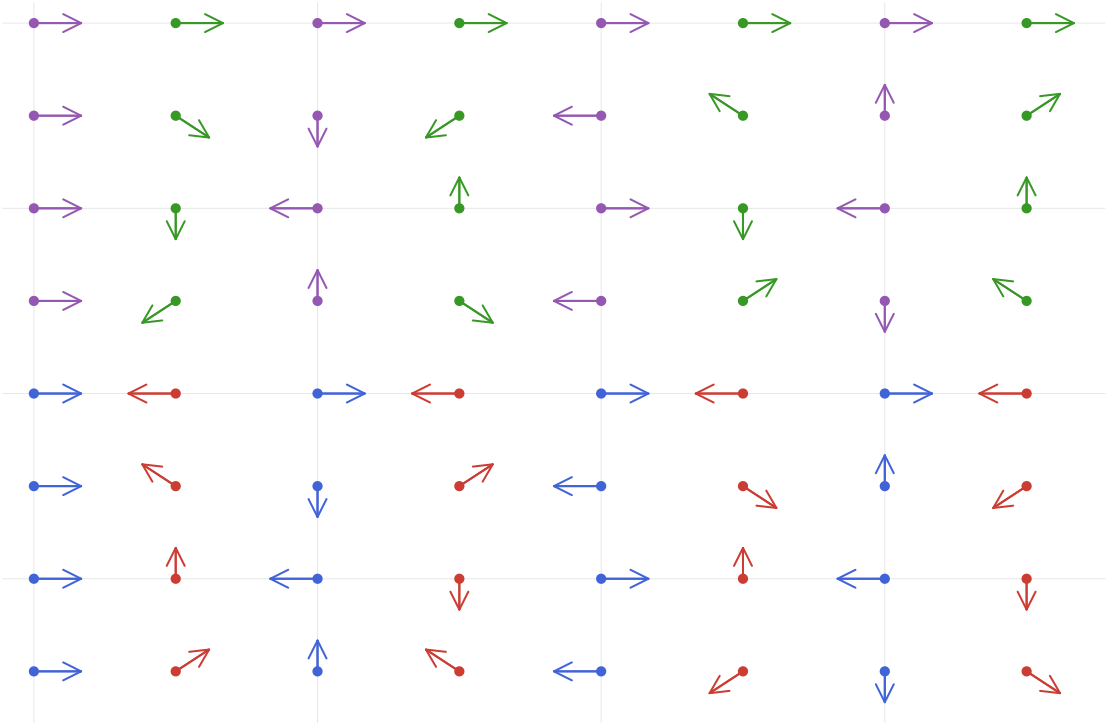
```
true
```

```
1 partie_verte == -partie_rouge
```

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



```
1 arrows(4, false)
```



```
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
```

```
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
```

```
F(4)
```

```
true
```

```
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
```

```
1 partie_verte
```

```
0.7071067811865476 - 0.7071067811865475im
```

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

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

```
1 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
```

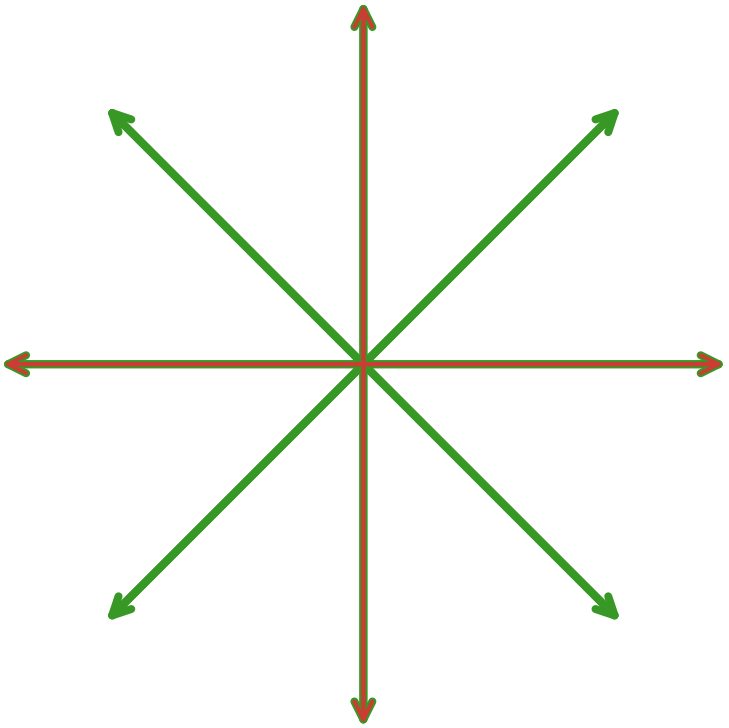
```
1 cispi(-2/8) .^ (0:3) .* F(4)
```

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

```
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



Voilà la clé. Un problème de taille N se ramène à **deux** problèmes de taille $N/2$, plus un travail linéaire de recombinaison. On recommence sur chaque moitié, et ainsi de suite jusqu'à des problèmes de taille 1.

Il y a $\log_2 N$ niveaux de découpage, et chaque niveau coûte N : d'où le $N \log N$.

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

4.2 La structure cachée, en équations

On note $z_N = e^{-i2\pi/N}$, de sorte que $X_k = \sum_{n=0}^{N-1} x_n \, z_N^{kn}$.

Séparer pairs et impairs. On coupe la somme en $n = 2m$ et $n = 2m+1$, et on sort le facteur z_N^k de la seconde :

$$X_k = \sum_{m=0}^{N/2-1} x_{2m} \, z_N^{2km} \; + \; z_N^k \sum_{m=0}^{N/2-1} x_{2m+1} \, z_N^{2km}.$$

Or $z_N^2 = z_{N/2}$: chaque somme est une **DFT de taille $N/2$**,

$$X_k = E_k + z_N^k\, O_k, \qquad E_k = \sum_{m=0}^{N/2-1} x_{2m}\, z_{N/2}^{km}, \qquad O_k = \sum_{m=0}^{N/2-1} x_{2m+1}\, z_{N/2}^{km}.$$

Obtenir la seconde moitié. Les appels récursifs ne rendent que $E_0, \dots, E_{N/2-1}$ et $O_0, \dots, O_{N/2-1}$. Deux observations suffisent :

- E et O sont $N/2$-périodiques : $E_{k+N/2} = E_k$, $O_{k+N/2} = O_k$;
- avancer de $N/2$ fait faire un demi-tour au *twiddle factor* : $z_N^{k+N/2} = -z_N^k$.

D'où le **papillon** :

$$\boxed{X_k = E_k + z_N^k\, O_k, \qquad X_{k+N/2} = E_k - z_N^k\, O_k, \qquad k = 0, \dots, N/2-1.}$$

C'est exactement ce que montraient les couleurs de la matrice : les parties mauve et bleue sont $F_{N/2}$ appliquée aux pairs (E), les parties verte et rouge sont $\pm z_N^k F_{N/2}$ appliquée aux impairs ($\pm z_N^k O$).

Un problème de taille N coûte donc deux problèmes de taille $N/2$, plus $N/2$ multiplications et N additions. On s'arrête à la taille 1, où $X_0 = x_0$.

Pseudo-code (N est une puissance de 2) :

```
FFT(x₀, …, x_{N-1}) :
    si N = 1 : renvoyer (x₀)                # DFT de taille 1
    E ← FFT(x₀, x₂, …, x_{N-2})             # indices pairs
    O ← FFT(x₁, x₃, …, x_{N-1})             # indices impairs
    pour k = 0, …, N/2 - 1 :
        t ← z_N^k · O_k                     # z_N = e^{-i2π/N}
        X_k       ← E_k + t
        X_{k+N/2} ← E_k - t                 # papillon
    renvoyer (X₀, …, X_{N-1})
```

Utilitaires

```
using Plots
```

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

```
using Polynomials, LinearAlgebra
```

```
import WAV, Base64
```

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

```
# 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)
    buffer = IOBuffer()
    WAV.wavwrite(Int.(trunc.(real.(sound*2^15))), buffer, Fs=samplerate, nbits=16)
    audio_b64 = Base64.base64encode(take!(buffer))
    HTML("""<audio controls src="data:audio/wav; base64,$(audio_b64)"></audio>""")
end
```

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

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

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

```
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
```

```
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
```