KLAFYVEL

Zeste de savoir

Jouons à implémenter une transformée de Fourier rapide !

29 juin 2021

Table des matières

Introduction

La transformée de Fourier est un outil essentiel dans de nombreux domaines, que ce soit en Physique, en traitement du signal, ou en Mathématiques. La méthode qui est probablement la plus connue pour la calculer numériquement s'appelle la **FFT** pour *Fast Fourier Transform*, ou Transformée de Fourier Rapide. Dans ce petit tutoriel, je vous propose d'essayer de comprendre et d'implémenter cet algorithme de manière efficace. J'utiliserais pour cela le langage Julia , mais il devrait vous être possible de suivre en utilisant d'autres langages tels que Python ou C. Nous comparerons les résultats obtenus avec ceux donnés par le portage en Julia de la bibliothèque FFTW .

Ce tutoriel S'adresse plutôt à des personnes ayant déjà eu l'occasion de rencontrer la transformée de Fourier, mais sans l'avoir pour autant déjà implémentée. Il se base en grande partie sur la troisième édition de Numerical Recipes [1]footnote:1, que je ne saurais que vous encourager à consulter : c'est une mine d'or.

1. [2]footnote:1 William H. Press, Saul A. Teukolsky, William T. Vetterling, & Brian P. Flannery. (2007). Numerical Recipes 3rd Edition: The Art of Scientific Computing (3e éd.). Cambridge University Press.

1. Quelques rappels sur la transformée de Fourier discrète

Introduction

La transformée de Fourier discrète est une transformation qui découle de la transformée de Fourier et est, comme son nom l'indique, adaptée pour des signaux discrets. Dans cette première partie je vous propose de découvrir comment construire la transformée de Fourier discrète puis comprendre pourquoi la transformée de Fourier rapide est utile.

1.1. La transformée de Fourier

Ce tutoriel n'a pas vocation à présenter la transformée de Fourier. Cependant, il existe plusieurs définitions de la transformée de Fourier et même au sein d'un unique domaine, il arrive que l'on en utilise plusieurs. Nous utiliserons la suivante: pour une fonction f, sa transformée de Fourier $\hat{f}$ est définie par :

$$\hat{f}(\nu) = \int_{-\infty}^{+\infty} f(x) e^{-2i\pi\nu x} \mathrm{d}x$$

1.2. De la transformée de Fourier à la transformée de Fourier discrète

Telle que défini dans la section précédente, la transformée de Fourier d'un signal est une fonction continue de la variable ν. Or, pour représenter un signal quelconque, nous ne pouvons utiliser qu'un nombre fini de valeurs. Pour cela on procède en quatre étapes:

1. On **échantillonne** (ou discrétise) le signal à analyser. Cela signifie qu'au lieu de travailler sur la fonction qui associe la valeur du signal en fonction de la variable x, on va travailler sur une suite discrète de valeurs du signal. Dans le cas de la FFT, on échantillonne avec un pas constant. Par exemple si on regarde un signal temporel comme la valeur d'une tension lue sur un voltmètre, on pourrait enregistrer la valeur à chaque *tic* d'une montre.
2. On **fenêtre** le signal discrétisé. Cela signifie que l'on garde uniquement un nombre fini de points du signal.
3. On échantillonne la transformée de Fourier du signal pour obtenir la transformée de Fourier discrète.

4. On fenêtre la transformée de Fourier discrète pour le stockage.

Je vous propose de raisonner sur un signal jouet qui aura la forme d'une Gaussienne. Cela rend le raisonnement un peu plus simple car la transformée de Fourier d'une Gaussienne réelle est elle aussi une gaussienne réelle,[1] footnote:1 ce qui simplifie les représentations graphiques.

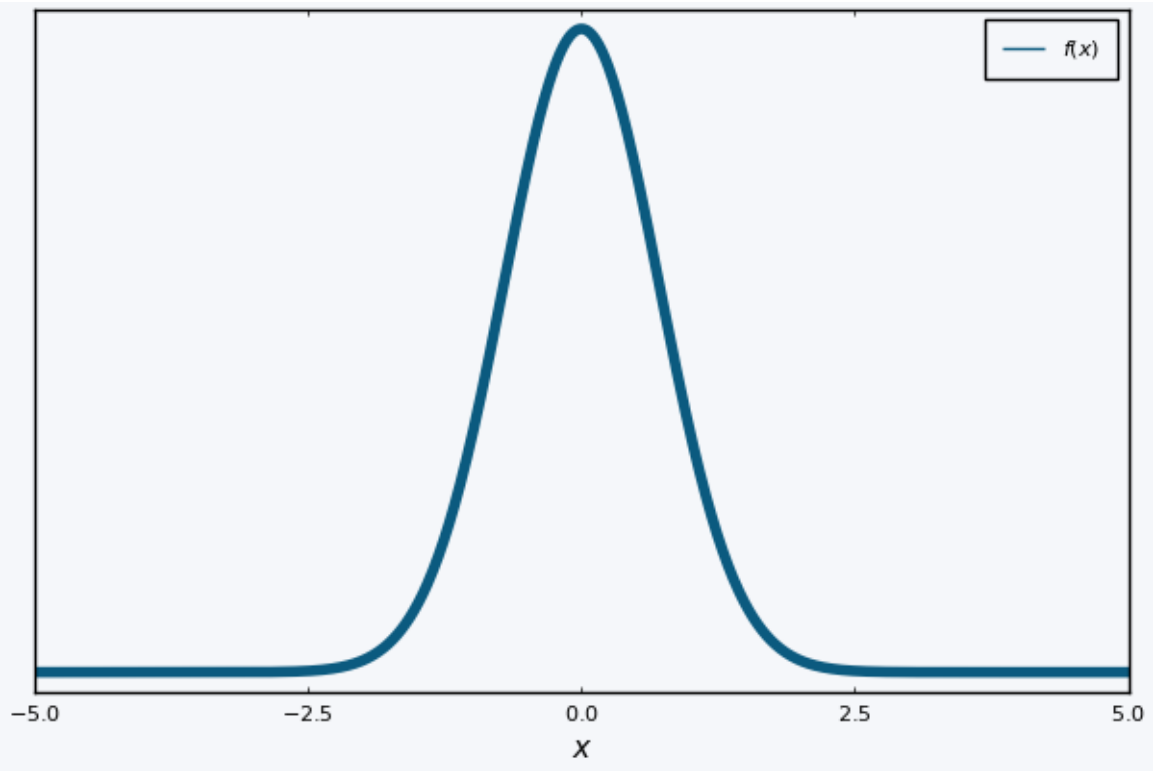

FIGURE 1.1. – Le signal qui nous servira d'exemple

Plus formellement, on a :

$$f(x) = e^{-x^2},\ \hat{f}(\nu) = \sqrt{\pi}e^{-(\pi\nu)^2}$$

Intéressons-nous tout d'abord à l'échantillonnage. Mathématiquement, on peut représenter le processus par la multiplication du signal f par un peigne de Dirac de péridode T, $_T$. Le peigne de Dirac est défini ainsi:

$$_T(x) = \sum_{k=-\infty}^{+\infty} \delta(x - kT)$$

Avec δ la distribution de Dirac . Voici le graphe que l'on peut obtenir si l'on représente f et $g = _T \times f$ ensemble:

1. [2] footnote:1 On dit que les gaussiennes sont des fonctions propres de la transformée de Fourier.

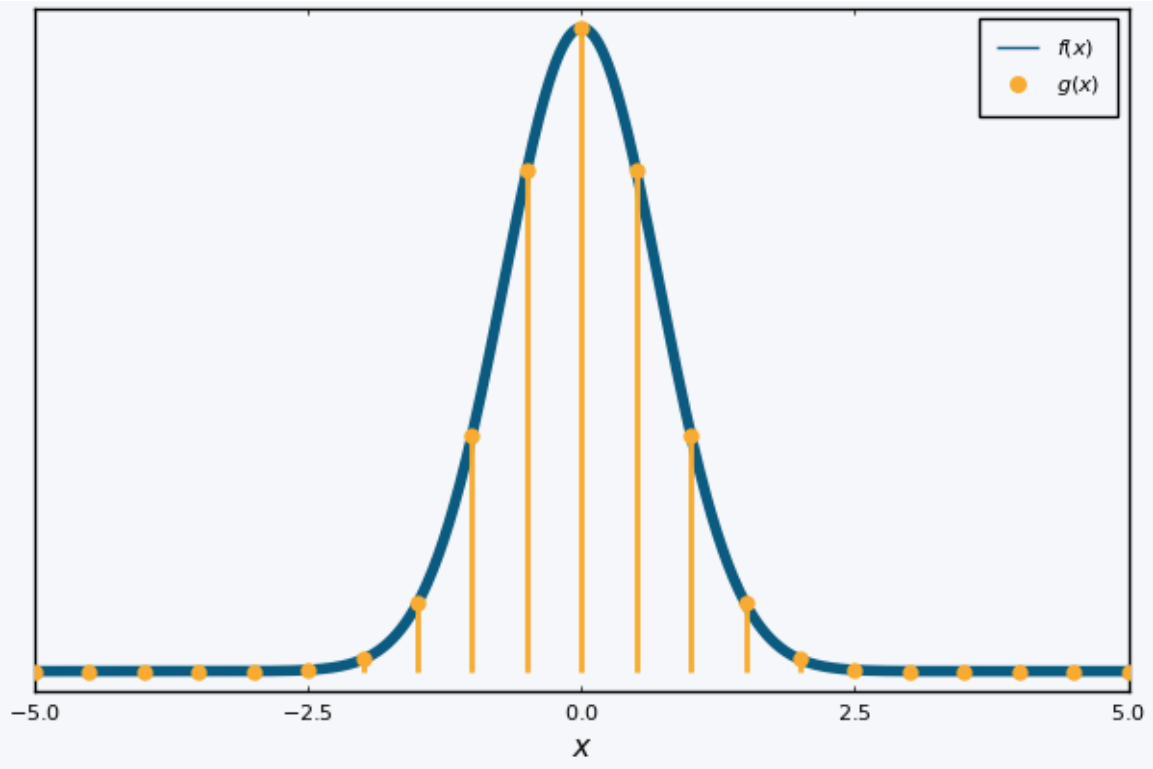

FIGURE 1.2. – Le signal et le signal échantillonné.

La transformée de Fourier de la nouvelle fonction g s'écrit [3]footnote:2:

$$\begin{aligned}\hat{g}(\nu) &= \int_{-\infty}^{+\infty} \sum_{k=-\infty}^{+\infty} \delta(x-kT) f(x) e^{-2i\pi x\nu} \mathrm{d}x \\ &= \sum_{k=-\infty}^{+\infty} \int_{-\infty}^{+\infty} \delta(x-kT) f(x) e^{-2i\pi x\nu} \mathrm{d}x \\ &= \sum_{k=-\infty}^{+\infty} f(kT) e^{-2i\pi kT\nu}\end{aligned}$$

Si on pose $f[k] = f(kT)$ le signal échantillonné et $\nu_{\text{ech}} = \frac{1}{T}$ la fréquence d'échantillonnage, on a :

$$\hat{g} = \sum_{k=-\infty}^{+\infty} f[k] e^{-2i\pi k \frac{\nu}{\nu_{\text{ech}}}}$$

Si on trace la transformée de Fourier du signal de départ $\hat{f}$ et celle du signal échantillonné $\hat{g}$, on obtient le graphe suivant:

2. [4]footnote:2 Il faudrait ici justifier que l'on peut inverser les signes somme et intégrale.

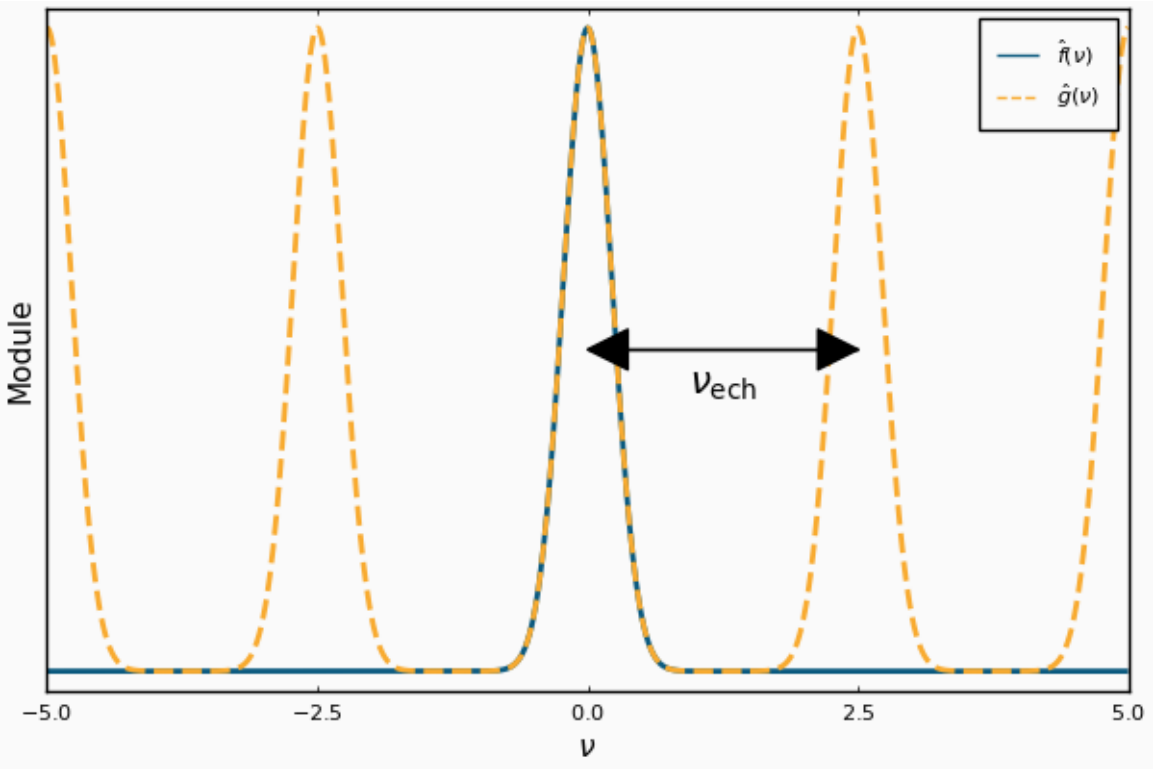

FIGURE 1.3. – Transformée de Fourier du signal et de son signal échantillonné.

i

On remarque que l'échantillonnage du signal a amené à la périodisation de sa transformée de Fourier. Cela mène à une propriété importante en traitement du signal: le critère de Nyquist-Shanon , et une de ses conséquences, le repliement de spectre, ou *aliasing*. Je vous laisse consulter l'article Wikipédia à ce propos si cela vous intéresse, mais on peut se faire une rapide idée de ce qu'il se passe si l'on trace le graphe précédent avec un échantillonnage trop large: les cloches de la transformée du signal échantillonné se superposent.

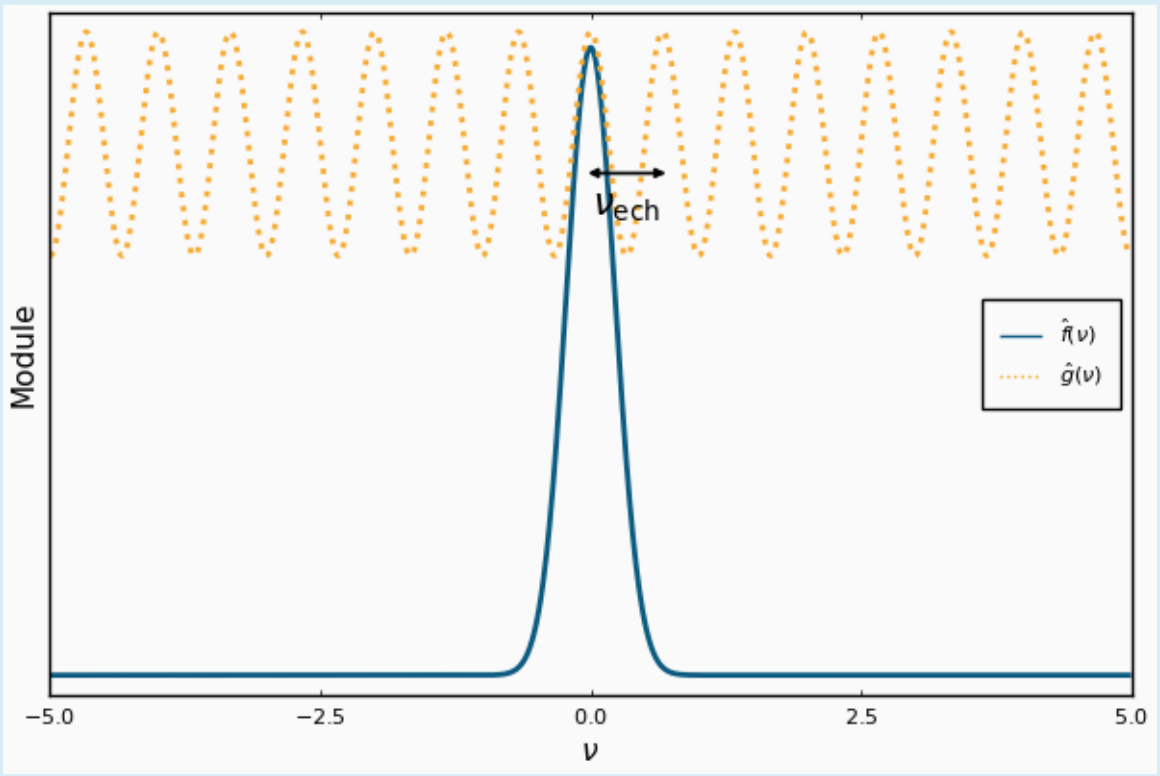

FIGURE 1.4. – Transformée de Fourier du signal et de son signal échantillonné, illustration du phénomène de repliement de spectre.

On peut ensuite regarder le processus de fenêtrage. Il existe plusieurs méthodes qui ont chacune leurs avantages, mais nous nous intéresserons ici uniquement à la fenêtre rectangulaire. Le principe est simple: on ne regarde que les valeurs de f que pour x compris entre $-x_0$ et $+x_0$. Cela signifie que l'on multiplie la fonction f par une fonction porte Π_{x_0} qui vérifie:

$$\Pi_{x_0}(x) = \begin{cases} 1 & \text{si } x \in [-x_0, x_0] \\ 0 & \text{sinon} \end{cases}$$

Graphiquement, voici comment on pourrait représenter h et f ensemble.

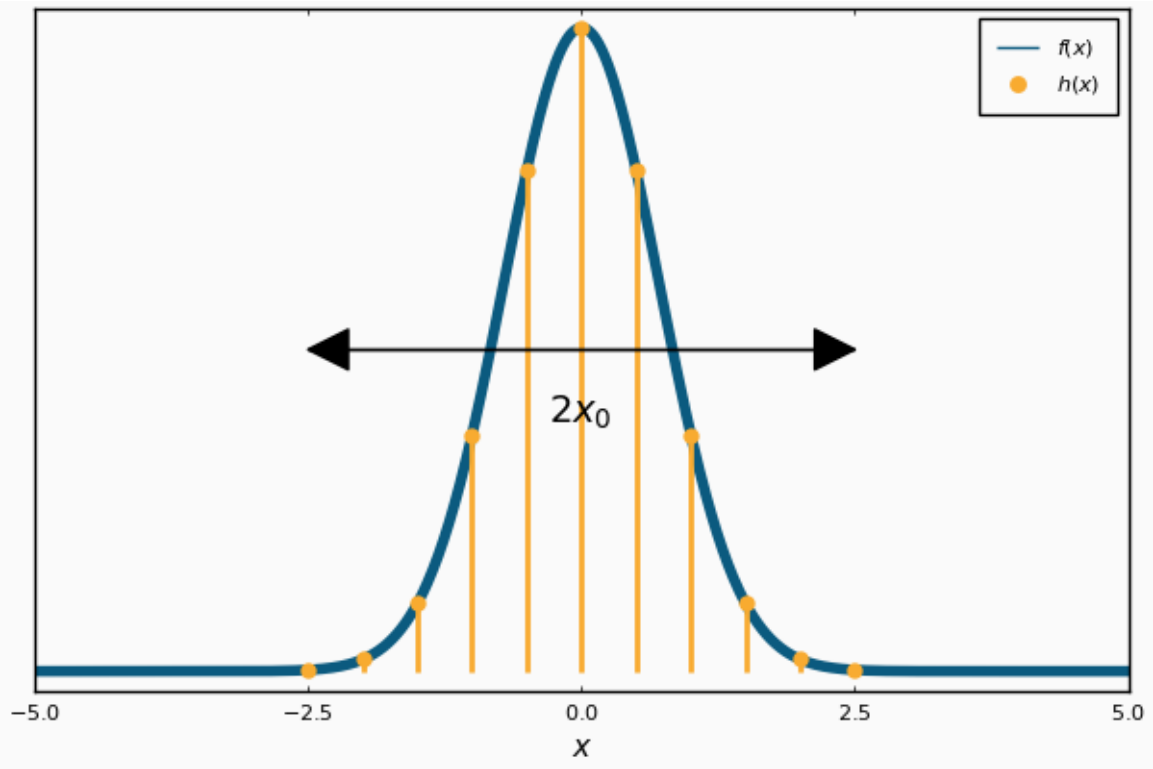

FIGURE 1.5. – Signal échantillonné et fenêtré

Concrètement, c'est équivalent à limiter la somme du peigne de Dirac à un nombre fini de termes. On peut alors écrire la transformée de Fourier de $h = \Pi_{x_0} \times_T \times f$:

$$\hat{h}(\nu) = \sum_{k=-k_0}^{+k_0} f[k] e^{-2i\pi k \frac{\nu}{\nu_{\text{ech}}}}$$

Le choix du fenêtrage n'est absolument pas anodin, et peut mener à des problèmes innatendus si on l'ignore. Là encore je vous conseille de consulter l'article Wikipédia associé au besoin.

Nous pouvons maintenant passer à la dernière étape: l'échantillonnage de la transformée de Fourier. En effet, on ne peut stocker qu'un nombre fini de valeurs sur notre ordinateur or, telle que définit, la fonction $\hat{h}$ est continue. On sait déjà qu'elle est périodique, de période ν_{ech}, on peut donc ne stocker que les valeurs entre 0 et ν_{ech}. Il reste encore à l'échantillonner, et en particulier à trouver le pas d'échantillonnage adéquat. Il est clair que l'on souhaite que l'échantillonnage soit le plus "fin" possible, pour ne manque aucun détail de la transformée de Fourier! Pour cela on peut s'inspirer de ce qu'il s'est passé lorsque l'on a échantillonné f : sa transformée de Fourier est devenue périodique, de période ν_{ech}. Or la transformée de Fourier inverse (l'opération qui permet de retrouver le signal à partir de sa transformée de Fourier) possède des propriétés similaires à la transformée de Fourier. Cela signifie que si on échantillonne $\hat{h}$ avec un pas d'échantillonnage ν_s, alors sa transformée de Fourier inverse devient périodique de période $1/\nu_s$. Cela donne une limite basse sur les valeurs que peut prendre ν_s! En effet, si

la transformée inverse a une période inférieure à la largeur de a fenêtre ($1/\nu_s < 2x_0$), alors le signal reconstruit pris entre $-x_0$ et x_0 ne correspondra pas au signal initial f!

On choisit donc $\nu_s = \frac{1}{2x_0}$ pour discrétiser $\hat{h}$. On utilise le même processus de multiplication par un peigne de Dirac pour discrétiser. De cette manière on obtient la transformée de Fourier d'une nouvelle fonction l:

$$\hat{l}(\nu) = \sum_{n=-\infty}^{+\infty} \delta(\nu - n\nu_s) \sum_{k=-k_0}^{+k_0} f[k] e^{-2i\pi k \frac{n\nu_s}{\nu_{\text{ech}}}}$$

Cette notation est un peu compliquée, et on peut s'intéresser plutôt à $\hat{l}[n] = \hat{l}(n\nu_s)$:

$$\begin{aligned} \hat{l}[n] = \hat{l}(n\nu_s) = & \sum_{k=-k_0}^{+k_0} f[k] e^{-2i\pi k \frac{n\nu_s}{\nu_{\text{ech}}}} \\ = & \sum_{k=0}^{N-1} f[k] e^{-2i\pi k \frac{n}{N}} \end{aligned}$$

Pour obtenir la dernière ligne, j'ai ré-indicé $f[k]$ de manière à commencer à 0, en notant N le nombre d'échantillons. J'ai ensuite supposé que la taille de la fenêtre correspondait à un nombre entier d'échantillons c'est-à-dire que $2x_0 = N \times T$, ce qui se réécrit $N \times \nu_s = \nu_{\text{ech}}$. Cette expression est la **transformée de Fourier discrète** du signal.

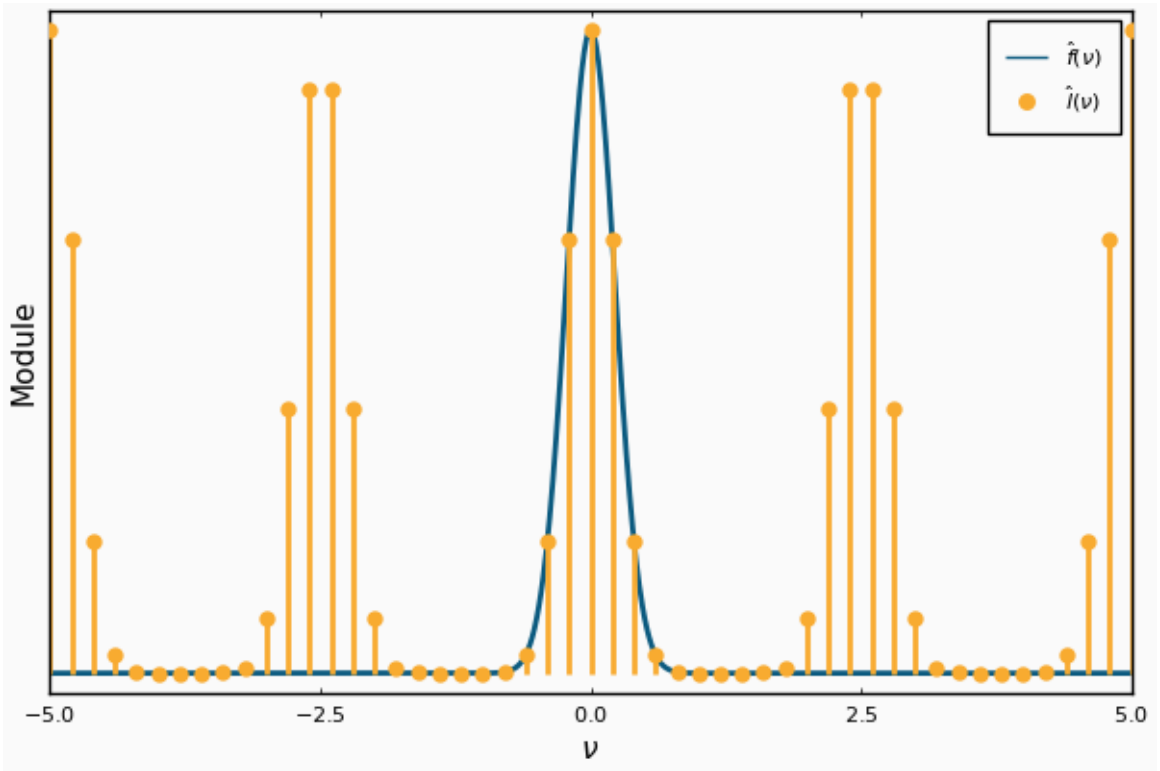

FIGURE 1.6. – Échantillonnage de la transformée de Fourier du signal échantillonné pour obtenir la transformée de Fourier discrète.

i

On voit que la fréquence d'échantillonnage n'entre pas en compte dans cette équation, et il y a de nombreuses applications où on oublie purement et simplement l'existence de cette fréquence.

?

Il reste un dernier point à éclaircir : cette transformée discrète est définie pour une infinité (discrète) de valeurs de n. Comment la stocker sur notre ordinateur?

Ce problème est résolu assez simplement par un fenêtrage de la transformée de Fourier discrète. Puisque la transformée a été périodisée par l'échantillonnage du signal de départ, il suffit de stocker une période de la transformée pour stocker l'intégralité de l'information contenue dans cette dernière. Le choix qui est généralement fait est de garder tous les points entre O et ν_{ech}. Cela permet de n'utiliser que des n positifs, et on peut reconstruire facilement le graphe de la transformée au besoin en inversant la première et la seconde moitiée de la transformée calculée. En pratique (pour l'implémentation), la transformée de Fourier discrète est donc donnée par:

$$\forall n = 0...(N-1),\ \hat{f}[n] = \sum_{k=0}^{N-1} f[k] e^{-2i\pi k \frac{n}{N}}$$

Pour conclure sur notre fonction d'exemple, on obtient le graphe suivant:

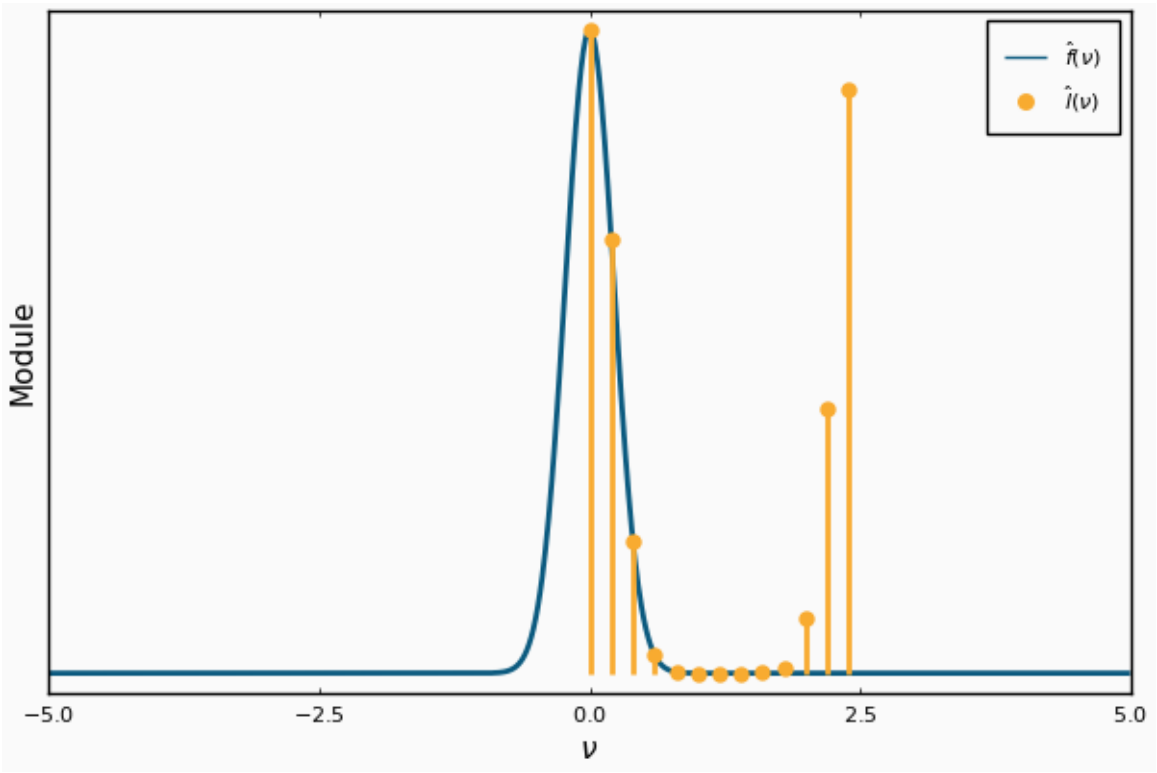

FIGURE 1.7. – Fenêtrage de la transformée de Fourier discrète pour le stockage.

1.3. Calculer la transformée de Fourier discrète

Nous avons donc à notre disposition l'expression de la transformée de Fourier discrète d'un signal f :

$$\hat{f}[n] = \sum_{k=0}^{N-1} f[k] e^{-2i\pi k \frac{n}{N}}$$

Cette expression est celle d'un produit matriciel qui ressemblerait à ceci :

$$\hat{f} = \mathbf{M} \cdot f$$

avec

$$\mathbf{M} = \begin{pmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & e^{-2i\pi 1\times 1/N} & e^{-2i\pi 2\times 1/N} & \dots & e^{-2i\pi 1\times (N-1)/N} \\ 1 & e^{-2i\pi 1\times 2\times 1/N} & e^{-2i\pi 2\times 2/N} & \ddots & \vdots \\ \vdots & \vdots & \ddots & \ddots & e^{e-2i\pi (N-2)\times (N-1)/N} \\ 1 & e^{-2i\pi (N-1)\times 1/N} & \dots & e^{e-2i\pi (N-1)\times (N-2)/N} & e^{-2i\pi (N-1)\times (N-1)/N} \end{pmatrix}$$

Les connaisseurs remarqueront qu'il s'agit ici d'une matrice de Vandermonde sur les racines de l'unité.

On peut donc implémenter relativement simplement ce calcul!

```
1 function naive_dft(x)
2         N = length(x)
3         k = reshape(0:(N-1), 1, :)
4         n = 0:(N-1)
5         M = @. exp(-2im * π * k * n / N)
6         M * x
7 end
```

i La macro `@.` ligne 5 permet de vectoriser le calcul de l'expression qu'elle englobe (`exp(-2im * * k * n / N)`). En effet la fonction `exp` et les opérateurs de division et multiplication sont définis pour des scalaires. Cette macro permet d'informer Julia qu'il doit appliquer les opérations scalaires terme à terme.

Et pour vérifier qu'il donne effectivement le bon résultat, il suffit de le comparer avec une implémentation de référence:

```
using FFTW

let
        a = rand(1024)
        b = fft(a)
        c = naive_dft(a)
        b ≈ c
end
```

Le dernier bloc s'évalue à `true`, ce qui confirme que nous ne sommes pas totalement à côté de la plaque!

> *i* J'utilise l'opérateur ≈ pour comparer plutôt que == afin d'autoriser des petites différences, notamment à cause des erreurs d'arrondies sur les flottants.

Cependant, ce code est-il efficace? On peut le vérifier en comparant l'empreinte mémoire et la vitesse d'exécution.

```
using BenchmarkTools

let
        @benchmark fft(a) setup=(a = rand(1024))
end
```

```
BenchmarkTools.Trial:
  memory estimate:  34.67 KiB
  allocs estimate:  34
  --------------
  minimum time:     20.943 µs (0.00% GC)
  median time:      23.785 µs (0.00% GC)
  mean time:        28.827 µs (1.14% GC)
  maximum time:     6.736 ms (48.98% GC)
  --------------
  samples:          10000
  evals/sample:     1
```

```
let
        @benchmark naive_dft(a) setup=(a = rand(1024))
end
```

```
BenchmarkTools.Trial:
  memory estimate:  16.03 MiB
  allocs estimate:  4
  --------------
  minimum time:     35.746 ms (0.00% GC)
  median time:      38.367 ms (0.00% GC)
  mean time:        39.493 ms (1.33% GC)
  maximum time:     51.803 ms (0.00% GC)
  --------------
  samples:          127
  evals/sample:     1
```

Comme vous avez pu le constater le temps d'exécution maximal de l'implémentation de référence est plus élevé de deux ordres de grandeur que les temps d'exécution moyen et médian. On doit cela à la compilation *Just in time* (JIT) de Julia. Si nous étions en train d'écrire une vraie bibliothèque Julia on pourrait envisager d'optimiser notre code pour qu'il compile rapidement. Nous nous contenterons d'ignorer le temps d'exécution maximum dans ce tutoriel, qui correspond uniquement au temps de compilation pour la première exécution du code. Je vous renvoie à la documentation de `BenchmarkTools.jl` pour plus d'informations.

Notre implémentation est donc *vraiment* lente (environ 10000 fois plus) et possède une empreinte mémoire très élevée (environ 500 fois) par comparaison avec l'implémentation de référence! Pour améliorer cela, nous allons implémenter la transformée de Fourier rapide.

1.4. Pourquoi un algorithme de transformée de Fourier rapide ?

Avant de replonger les mains dans le cambouis, posons-nous d'abord la question: est-il bien nécessaire de chercher à améliorer cet algorithme?

Avant de répondre directement, intéressons-nous à quelques applications de la transformée de Fourier et de la transformée de Fourier discrète.

La transformée de Fourier a d'abord une foultitude d'applications théoriques, que ce soit pour résoudre des équations différentielles, en traitement du signal ou en physique quantique. Elle possède également des applications pratiques en optique et en spectroscopie .

La transformée de Fourier discrète a également de nombreuses applications, en analyse des signaux, pour la compression de données, la multiplication de polynômes ou le calcul de produits de convolutions.

Notre implémentation naïve de la transformée de Fourier discrète a une complexité en temps et en mémoire en $\mathcal{O}(N^2)$ avec N la taille de l'échantillon d'entrée, cela est dû au stockage de la matrice et au temps de calcul du produit matriciel. Concrètement, si on souhaitait analyser un signal sonore de 3 secondes échantilloné à 44kHz avec des données stockées sur avec des flottants simple précision (4 octets), il faudrait donc environ $2 \times (44000 \times 3)^2 \times 4 \approx 100\ 000\ 000\ 000$

octets de mémoire (un nombre complexe est stocké sur 2 flottants). On peut aussi estimer le temps nécessaire à faire ce calcul. Le temps médian pour 1024 points était de 38.367 ms. Pour notre signal de 3 secondes, il faudrait environ $38.867 \times \left(\frac{44000\times3}{1024}\right)^2 \approx 637\,537$ millisecondes, soit plus de 10 minutes!

On comprend aisément l'intérêt de réduire la complexité du calcul. En particulier l'algorithme de la transformée de Fourier rapide (utilisé par l'implémentation de référence) possède une complexité en $\mathcal{O}(N \log N)$. D'après notre *benchmark*, l'algorithme traite une entrée de 1024 points en 23.785µs. Il devrait donc traiter le signal sonore en environ $23.785 \times \frac{44000\times3\times\log(44000\times3)}{1024\log 1024} \approx 5\,215$ micro-secondes, soit environ 120000 fois plus rapidement que notre algorithme. On peut vraiment dire que le *fast* de *Fast Fourier Transform* n'est pas volé!

Conclusion

Nous avons vu comment la transformée de Fourier discrète était construite, puis nous avons tenté de manière naïve de l'implémenter. Bien que cette implémentation soit relativement simple à mettre en œuvre (surtout avec un langage tel que Julia qui facilite les manipulations de matrices), nous avons également vu ses limites en termes de temps d'exécution et d'empreinte mémoire.

Il est temps de passer à la FFT proprement dite!

2. Implémentons la FFT

Introduction

Dans cette partie nous allons implémenter la FFT en partant d'une approche simple, puis en complexifiant au fur et à mesure pour essayer de calculer la transformée de Fourier d'un signal réel de la manière la plus efficace possible. Pour comparer les performances de nos implémentations, nous continuerons à comparer avec l'implémentation de FFTW.

2.1. Ma première FFT

Nous avons trouvé précédemment l'expression de la transformée de Fourier discrète:

$$\hat{f}[n] = \sum_{k=0}^{N-1} f[k] e^{-2i\pi k \frac{n}{N}}$$

L'astuce au cœur de l'algorithme de la FFT consiste à remarquer que si l'on essaie de couper cette somme en deux, en séparant les termes pairs et les termes impairs, on obtient (en supposant que N soit pair), pour $n < N/2$:

$$\begin{aligned}
\hat{f}[n] &= \sum_{k=0}^{N} f[k] e^{-2i\pi k \frac{n}{N}} \\
&= \sum_{m=0}^{N/2-1} f[2m] e^{-2i\pi 2m \frac{n}{N}} + \sum_{m=0}^{N/2-1} f[2m+1] e^{-2i\pi (2m+1) \frac{n}{N}} \\
&= \sum_{m=0}^{N/2-1} f[2m] e^{-2i\pi m \frac{n}{N/2}} + e^{-2i\pi n/N} \sum_{m=0}^{N/2-1} f[2m+1] e^{-2i\pi m \frac{n}{N/2}} \\
&= \hat{f}^{\text{pair}}[n] + e^{-2i\pi n/N} \hat{f}^{\text{impair}}[n]
\end{aligned}$$

où $\hat{f}^{\text{pair}}$ et $\hat{f}^{\text{impair}}$ sont les transformées de Fourier de la séquence des termes pairs de f et de la séquence des termes impairs de f. On peut donc calculer la première moitiée de la transformée de Fourier de f en calculant les transformées de Fourier de ces deux séquences de longueur $N/2$ et en les recombinant. De même, si on calcule $\hat{f}[n + N/2]$ on a:

$$\begin{aligned}\hat{f}[n+N/2] &= \sum_{m=0}^{N/2-1} f[2m]e^{-2i\pi m\frac{n+N/2}{N/2}} + e^{-2i\pi(n+N/2)/N}\sum_{m=0}^{N/2-1} f[2m+1]e^{-2i\pi m\frac{n+N/2}{N/2}} \\ &= \sum_{m=0}^{N/2-1} f[2m]e^{-2i\pi m\frac{n}{N/2}} - e^{-2i\pi n/N}\sum_{m=0}^{N/2-1} f[2m+1]e^{-2i\pi m\frac{n}{N/2}} \\ &= \hat{f}^{\text{pair}}[n] - e^{-2i\pi n/N}\hat{f}^{\text{impair}}[n]\end{aligned}$$

Cela signifie qu'en calculant deux transformées de Fourier de longueur $N/2$, on est capable de calculer deux éléments d'une transformée de Fourier de longueur N. En supposant pour simplifier que N soit une puissance de deux[1]footnote:1, cela dessine naturellement une implémentation récursive de la FFT. D'après le master theorem, cet algorithme sera de complexité $\mathcal{O}(N \log_2 N)$, ce qui est bien mieux que le premier algorithme naïf que nous avons implémenté, qui présente une complexité en $\mathcal{O}(N^2)$.

```
function my_fft(x)
        # Condition d'arrêt, la TF d'un tableau de taille 1 est ce même tableau.
        if length(x) <= 1
                x
        else
                N = length(x)
                # Xº contient la TF des termes impairs et Xᵉ celle des termes pairs.
                # La subtilité étant que les tablaux de Julia commencent à 1 et non à 0.
                Xº = my_fft(x[2:2:end])
                Xᵉ = my_fft(x[1:2:end])
                factors = @. exp(-2im * π * (0:(N/2 - 1)) / N)
                [Xᵉ .+ factors .* Xº; Xᵉ .- factors .* Xº]
        end
end
```

On peut vérifier comme précédemment que code donne un résultat juste, puis comparer ses qualités d'exécution avec l'implémentation de référence.

```
let
        @benchmark fft(a) setup=(a = rand(1024))
end
```

```
BenchmarkTools.Trial:
  memory estimate:  34.67 KiB
```

1. [2]footnote:1 En pratique on peut toujours se ramener à ce cas en faisant du bourrage de zéros.

```
  allocs estimate:  34
  --------------
  minimum time:     20.962 µs (0.00% GC)
  median time:      23.864 µs (0.00% GC)
  mean time:        29.704 µs (0.92% GC)
  maximum time:     5.254 ms (52.07% GC)
  --------------
  samples:          10000
  evals/sample:     1
```

```
let
        @benchmark my_fft(a) setup=(a = rand(1024))
end
```

```
BenchmarkTools.Trial:
  memory estimate:  1.20 MiB
  allocs estimate:  14322
  --------------
  minimum time:     3.684 ms (0.00% GC)
  median time:      4.423 ms (0.00% GC)
  mean time:        4.924 ms (4.04% GC)
  maximum time:     15.208 ms (54.38% GC)
  --------------
  samples:          1012
  evals/sample:     1
```

On voit que l'on a bien amélioré le temps d'exécution (d'un facteur 8) et l'empreinte mémoire de l'algorithme (d'un facteur 13), sans pour autant se rapprocher de l'implémentation de référence.

2.2. Analyse de la première implémentation

Reprenons le code précédent:

```
function my_fft(x)
        # Condition d'arrêt, la TF d'un tableau de taille 1 est ce
            même tableau.
        if length(x) <= 1
                x
        else
                N = length(x)
```

```
7                   # Xº contient la TF des termes impairs et Xᵉ celle
                        des termes pairs.
8                   # La subtilité étant que les tablaux de Julia
                        commencent à 1 et non à 0.
9                   Xº = my_fft(x[2:2:end])
10                  Xᵉ = my_fft(x[1:2:end])
11                  factors = @. exp(-2im * π * (0:(N/2 - 1)) / N)
12                  [Xᵉ .+ factors .* Xº; Xᵉ .- factors .* Xº]
13          end
14  end
```

Et essayons de suivre les allocations de mémoire. Pour simplifier, on peut supposer que l'on travaille sur un tableau de 4 éléments, `[f[0], f[1], f[2], f[3]]`. Le premier appel à `my_fft` garde en mémoire le tableau initial, puis lance la fft sur deux sous tableaux de taille 2: `[f[0], f[2]]` et `[f[1], f[3]]`, puis les appels récursifs gardent en mémoire avant de recombiner les tableaux `[f[0]]` et `[f[2]]` puis `[f[1]]` et `[f[3]]`. Au plus, on a donc $log_2(N)$ tableaux alloués avec des tailles divisées par deux à chaque fois. Non seulement ces tableaux occupent de la mémoire, mais en plus on perd du temps à les allouer!

Or si l'on observe la définition de la récurrence que l'on utilise, à chaque étape i (c'est-à-dire pour chaque taille de tableaux, $N/2^i$), la somme des tailles des tableaux intermédiaires est toujours N. Autrement dit, cela donne l'idée que l'on pourrait s'épargner toutes ces allocations de tableaux et utiliser en permanence le même tableau, à condition de réaliser toutes les associations de tableaux de même taille à la même étape.

Schématiquement on peut représenter le processus de la FFT pour un tableau à 8 éléments ainsi:

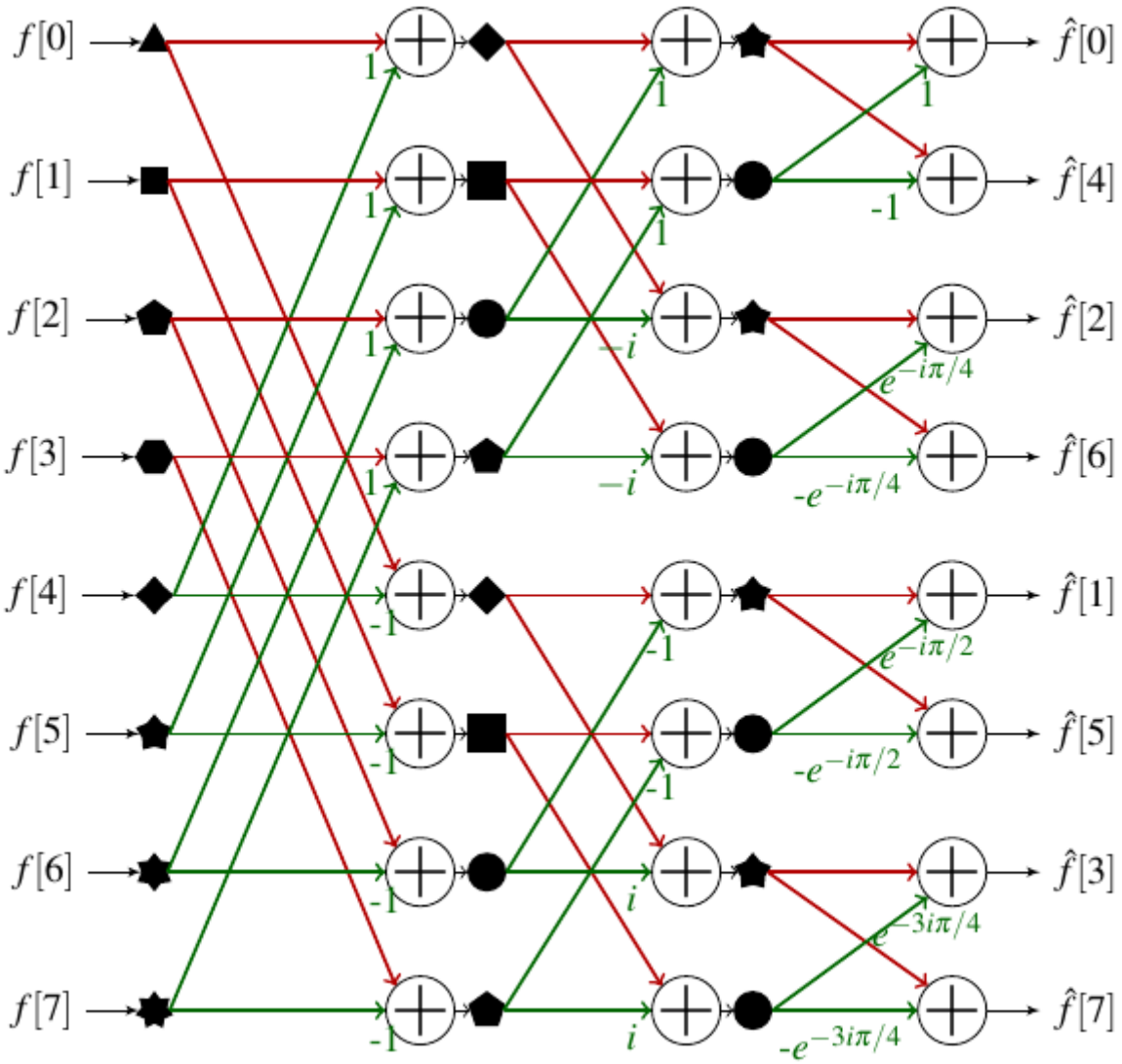

FIGURE 2.1. – Illustration du processus de la FFT. Les couleurs indiquent si un élément est traité comme un tableau pair (rouge) ou un tableau impair (vert). Les formes géométriques permettent d'associer les éléments qui sont dans le même sous-tableau. On représente également les coefficients multiplicatifs appliqués aux éléments impairs. Ce schéma, un peu compliqué, est la clé de ce qui va suivre. N'hésitez pas à passer un peu de temps pour le comprendre.

Comment lire ce schéma? Chaque colonne correspond à une profondeur de la récurrence de notre première FFT. La colonne la plus à gauche correspond à la récurrence la plus profonde: on a découpé le tableau d'entrée suffisamment pour arriver à des sous-tableaux de taille 1. Ces 8 sous-tableaux sont symbolisés par 8 formes géométriques différentes. On passe ensuite au niveau de récurrence supérieur. Chaque paire de sous-tableaux de taille 1 doit être combinée pour créer un sous-tableau de taille 2, qui sera stocké dans les mêmes cases mémoire que les deux sous-tableaux de taille 1. Par exemple, on combine le sous-tableau qui contient $f[0]$ et le sous-tableau qui contient $f[4]$ en utilisant la formule démontrée précédemment pour former le tableau $[f[0] + f[4], f[0] - f[4]]$, que j'ai appelé dans la suite , et on stocke les deux valeurs en position 0 et 4. Les couleurs des flèches permettent de distinguer celles portant un coefficient (qui correspondent au traitement que l'on fait subir au sous-tableau $\hat{f}^{\text{impair}}$ dans les formules de la section précédente). Après avoir construit les 4 sous-tableaux de taille 2, on peut passer à une

nouvelle étape de la récurrence pour calculer deux sous-tableaux de taille 4. Enfin la dernière étape de la récurrence combine les deux sous-tableaux de taille 4 pour calculer le tableau de taille 8 qui contient la transformée de Fourier.

En s'inspirant de ce schéma on peut penser à avoir une fonction dont la boucle principale calculerait successivement chaque colonne pour arriver au résultat final. De cette manière on effectue tous les calculs sur un seul et même tableau et on minimise le nombre d'allocations! Il y a cependant un problème : on voit que les $\hat{f}[k]$ ne semblent pas ordonnés à la fin du processus.

En réalité, ces $\hat{f}[k]$ sont ordonnés via une permutation inverse des bits . Cela signifie que si l'on écrit les indices k en binaire, puis que l'on inverse cette écriture (le MSB devenant le LSB[3]footnote:1), on obtient l'indice auquel $\hat{f}[k]$ se trouve après l'algorithme de FFT. Le process de permutation est décrit par le tableau suivant dans le cas d'un calcul sur 8 éléments.

k	**Représentation binaire de k**	**Représentation binaire inversée**	**Indice de $\hat{f}[k]$**
0	000	000	0
1	001	100	4
2	010	010	2
3	011	110	6
4	100	001	1
5	101	101	5
6	110	011	3
7	111	111	7

Si on sait calculer la permutation inverse des bits, on peut simplement réordonner à la fin du processus le tableau pour obtenir le bon résultat. Cependant avant de se jeter sur l'implémentation il est intéressant de regarder ce qu'il se passe si à la place on réordonne le tableau d'entrée *via* cette permutation.

1. [4]footnote:1 MSB et LSB sont les acronymes de *Most Significant Bit* et *Least Significant Bit*. Dans un nombre représenté sur n bits, le MSB est donc le bit qui porte l'information sur la puissance de 2 la plus élevée (2^{n-1}) alors que le LSB porte l'information sur la puissance de 2 la plus faible (2^0). Concrètement le MSB est le bit le plus à gauche de la représentation binaire d'un nombre, alors que le LSB est le plus à droite.

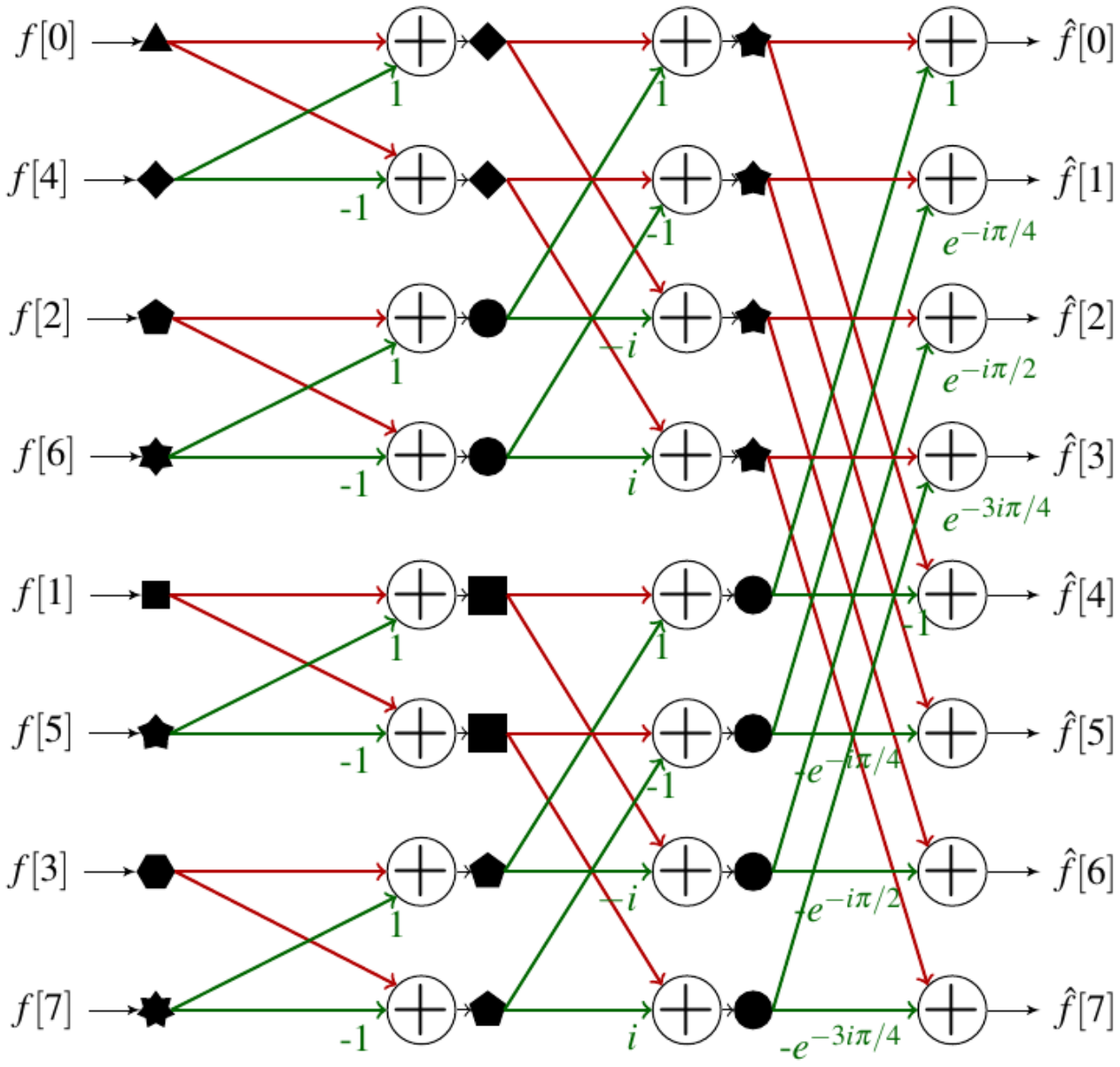

FIGURE 2.2. – Schématisation de la FFT avec une entrée permutée. Les couleurs et les symboles sont les mêmes que sur la première illustration.

On voit qu'en procédant ainsi on a un ordonnement simple des sous tableaux. Puisque de toute manière il faudra procéder à une permutation du tableau, il est intéressant de le faire avant le calcul de la FFT.

2.3. Calculer la permutation inverse des bits

Il faut donc commencer par être en mesure de calculer la permutation. Il est possible de réaliser la permutation en place simplement une fois que l'on sait quels éléments échanger. Plusieurs méthodes existent pour effectuer la permutation, et une recherche dans Google Scholar vous donnera un aperçu de la richesse des approches.

Nous pouvons utiliser une petite astuce ici : puisque nous traitons uniquement des tableaux dont la taille est une puissance de 2, on peut écrire la taille N ainsi: $N = 2^p$. Cela signifie que les indices peuvent être stockés sur p bits. On peut alors simplement calculer l'indice permuté *via* des opérations binaires. Par exemple si $p = 10$ alors l'indice 797 pourrait être représenté ainsi: `1100011101`.

On peut séparer le processus d'inversion en plusieurs étapes. Dans un premier temps on échange les 5 bits les plus significatifs et les 5 bits les moins significatifs. Ensuite sur chacun des demi-mots on inverse les deux bits les plus significatifs et les deux bits les moins significatifs (les bits centraux ne changent pas). Enfin sur les mots de deux bits que l'on vient d'échanger, on échange le bit le plus significatif et le bit le moins significatif.

Un exemple d'implémentation serait le suivant:

```
bit_reverse(::Val{10}, num) = begin
        num = ((num&0x3e0)>>5)|((num&0x01f)<<5)
        num = ((num&0x318)>>3)|(num&0x084)|((num&0x063)<<3)
        ((num&0x252)>>1)|(num&0x084)|((num&0x129)<<1)
end
```

Un algorithme équivalent peut être appliqué pour toutes les valeurs de p, il faut simplement faire attention à ne plus modifier les bits centraux quand on a un nombre impair de bits dans un demi-mot. Dans ce qui suit il y a un exemple pour plusieurs longueurs de mots.

Contenu masqué n°1

On peut ensuite faire la permutation à proprement parler. L'algorithme est relativement simple: il suffit d'itérer sur le tableau, calculer l'indice inversé de l'indice courant et réaliser l'inversion. La seule subtilité est qu'il ne faut réaliser l'inversion qu'une fois par indice du tableau, donc on discrimine en ne réalisant l'inversion que si l'indice courant est inférieur à l'indice inversé.

```
function reverse_bit_order!(X, order)
        N = length(X)
        for i in 0:(N-1)
                j = bit_reverse(order, i)
                if i<j
                        X[i+1],X[j+1]=X[j+1],X[i+1]
                end
        end
        X
end
```

2.4. Ma seconde FFT

Nous sommes maintenant suffisamment équipés pour nous lancer dans une seconde implémentation de la FFT. La première étape sera de calculer la permutation de bits inverse. Ensuite on pourra effectuer le calcul de la transformée de Fourier en suivant le schéma montré précédemment. Pour ce faire on stockera la taille n_1 des sous tableaux et le nombre de cases n_2 dans le tableau global qui séparent deux éléments de même indices dans les sous-tableaux. L'implémentation peut se faire ainsi:

```
function my_fft_2(x)
        N = length(x)
        order = Int(log2(N))
        @inbounds reverse_bit_order!(x, Val(order))
        n₁ = 0
        n₂ = 1
        for i=1:order # i done le numéro de la colonne dans lequel
            on se trouve.
                n₁ = n₂ # n₁ = 2^(i-1)
                n₂ *= 2 # n₂ = 2^i

                step_angle = -2π/n₂
                angle = 0
                for j=1:n₁ # j est l'indice dans Xᵉ et Xᵒ
                        factors = exp(im*angle) # z =
                            exp(-2im*π*(j-1)/n₂)
                        angle += step_angle # a = -2π*(j+1)/n₂

                        # On combine l'élément j de chaque groupe
                            de sous-tableaux
                        for k=j:n₂:N
                                @inbounds x[k], x[k+n₁] = x[k] +
                                    factors * x[k+n₁], x[k] -
                                    factors * x[k+n₁]
                        end
                end
        end
        x
end
```

i Il y a deux petites subtilités dues à Julia: les tableaux commencent leur numérotation à 1, et on utilise la macro `@inbounds` pour accélérer un peu le code en désactivant les vérifications de débordement de tableaux.

On peut à nouveau mesurer les performances de cette implémentation. Pour que la comparaison reste juste, il faut utiliser la fonction `fft!` et non plus `fft`, car elle travaille en place.

```
@benchmark fft!(a) setup=(a = rand(1024) |> complex)
```

```
BenchmarkTools.Trial:
  memory estimate:  2.34 KiB
  allocs estimate:  30
  --------------
  minimum time:     20.358 µs (0.00% GC)
  median time:      22.760 µs (0.00% GC)
  mean time:        29.585 µs (2.02% GC)
  maximum time:     7.197 ms (49.08% GC)
  --------------
  samples:          10000
  evals/sample:     1
```

```
@benchmark my_fft_2(a) setup=(a = rand(1024) .|> complex)
```

```
BenchmarkTools.Trial:
  memory estimate:  0 bytes
  allocs estimate:  0
  --------------
  minimum time:     51.361 µs (0.00% GC)
  median time:      52.232 µs (0.00% GC)
  mean time:        58.667 µs (0.00% GC)
  maximum time:     271.770 µs (0.00% GC)
  --------------
  samples:          10000
  evals/sample:     1
```

On a significativement amélioré notre temps d'exécution et notre empreinte mémoire. On peut voir qu'il y a zéro octet alloué (cela signifie que le compilateur n'a pas besoin de stocker les quelques variables intermédiaires en RAM), et que le temps d'exécution se rapproche de celui de l'implémentation de référence.

2.5. Le cas particulier d'un signal réel

Jusqu'ici nous avons raisonné sur des signaux complexes, qui utilisent deux flottants pour le stockage. Cependant dans de nombreuses situations on travaille avec des signaux à valeur réelle. Or dans le cas d'un signal réel, on sait que $\hat{f}$ vérifie $\hat{f}(-\nu) = \overline{\hat{f}(\nu)}$. Cela signifie que la moitié des valeurs que l'on calcule est redondante. Bien que l'on calcule la transformée de Fourier d'un signal réel, le résultat peut être un nombre complexe. Afin d'économiser de l'espace de stockage,

on peut penser à utiliser cette moitié de tableau pour stocker les nombres complexes. Pour cela deux propriétés vont nous aider.

2.5.1. Propriété 1 : Calculer la transformée de Fourier de deux fonctions réelles en même temps

Si l'on a deux signaux réels f et g, on peut définir le signal complexe $h = f + ig$. On a alors:

$$\hat{h}[k] = \sum_{n=0}^{N-1} (f[n] + ig[n]) e^{-2i\pi kn/N}$$

On peut remarquer que

$$\begin{aligned}\overline{\hat{h}[N-k]} &= \overline{\sum_{n=0}^{N-1} (f[n] + ig[n]) e^{-2i\pi (N-k)n/N}} \\ &= \sum_{n=0}^{N-1} (f[n] - ig[n]) e^{-2i\pi kn/N}\end{aligned}$$

En combinant les deux on a

$$\begin{aligned}\hat{f}[k] &= \frac{1}{2}(\hat{h}[k] + \overline{\hat{h}[N-k]}) \\ \hat{g}[k] &= -\frac{i}{2}(\hat{h}[k] - \overline{\hat{h}[N-k]})\end{aligned}$$

2.5.2. Propriété 2 : Calculer la transformée de Fourier d'une seule fonction

L'idée est d'utiliser la propriété précédente en utilisant le signal des éléments pairs et celui des éléments impairs. Autrement dit pour $k = 0...N/2 - 1$ on a $h[k] = f[2k] + if[2k+1]$.

On a alors :

$$\begin{aligned}\hat{f}^{\text{pair}}[k] &= \sum_{n=0}^{N/2-1} f[2k] e^{-2i\pi kn/(N/2)} \\ \hat{f}^{\text{impair}}[k] &= \sum_{n=0}^{N/2-1} f[2k+1] e^{-2i\pi kn/(N/2)}\end{aligned}$$

On peut recombiner les deux transformées partielles. Pour $k = 0...N/2 - 1$:

$$\begin{aligned}\hat{f}[k] &= \hat{f}^{\text{pair}}[k] + e^{-2i\pi k/N} \hat{f}^{\text{impair}}[k] \\ \hat{f}[k + N/2] &= \hat{f}^{\text{pair}}[k] - e^{-2i\pi k/N} \hat{f}^{\text{impair}}[k]\end{aligned}$$

En utilisant la première propriété, on a alors:

$$\hat{f}[k] = \frac{1}{2}(\hat{h}[k] + \overline{\hat{h}[N/2-k]}) - \frac{i}{2}(\hat{h}[k] - \overline{\hat{h}[N/2-k]})e^{-2i\pi k/N}$$
$$\hat{f}[k+N/2] = \frac{1}{2}(\hat{h}[k] + \overline{\hat{h}[N/2-k]}) + \frac{i}{2}(\hat{h}[k] - \overline{\hat{h}[N/2-k]})e^{-2i\pi k/N}$$

2.5.3. Calcul en place

Le tableau h, qui est présenté précédemment, est à valeurs complexes. Cependant le signal d'entrée est à valeurs réelles et deux fois plus long. L'astuce est donc d'utiliser deux cases du tableau initial pour stocker un élément complexe de h. Il est utile de poser les calculs avec nombres complexes avant de commencer à écrire du code. Pour le cœur de la FFT, si on note x_i le tableau à l'étape i de la boucle principale, on a:

$$\mathrm{Re}(x_{i+1}[k]) = \mathrm{Re}(x_i[k]) + \mathrm{Re}(e^{-2i\pi j/n_2})\mathrm{Re}(x_i[k+n_1]) - \mathrm{Im}(e^{-2i\pi j/n_2})\mathrm{Im}(x_i[k+n_1])$$
$$\mathrm{Re}(x_{i+1}[k]) = \mathrm{Re}(x_i[k]) + \mathrm{Re}(e^{-2i\pi j/n_2})\mathrm{Re}(x_i[k+n_1]) - \mathrm{Im}(e^{-2i\pi j/n_2})\mathrm{Im}(x_i[k+n_1])$$

$$\mathrm{Re}(x_{i+1}[k+n_1]) = \mathrm{Re}(x_i[k]) - \mathrm{Re}(e^{-2i\pi j/n_2})\mathrm{Re}(x_i[k+n_1]) + \mathrm{Im}(e^{-2i\pi j/n_2})\mathrm{Im}(x_i[k+n_1])$$
$$\mathrm{Re}(x_{i+1}[k+n_1]) = \mathrm{Re}(x_i[k]) - \mathrm{Re}(e^{-2i\pi j/n_2})\mathrm{Re}(x_i[k+n_1]) + \mathrm{Im}(e^{-2i\pi j/n_2})\mathrm{Im}(x_i[k+n_1])$$

Avec l'organisation que l'on choisit, on aura $\mathrm{Re}(x[k]) = x[2k]$ et $\mathrm{Im}(x[k]) = x[2k+1]$.

La dernière étape est le recombinaison de h pour trouver le résultat final. La formule de la propriété 2 se ré-écrit après un calcul désagréable mais peu compliqué :

$$\begin{aligned}\mathrm{Re}(\hat{x}[k]) = \; & 1/2 \times (\mathrm{Re}(h[k]) + \mathrm{Re}(h[N/2-k]) + \mathrm{Im}(h[k])\mathrm{Re}(e^{-2i\pi k/N}) + \mathrm{Re}(h[k])\mathrm{Im}(e^{-2i\pi k/N})... \\ & ... + \mathrm{Im}(h[N/2-k])\mathrm{Re}(e^{-2i\pi k/N}) - \mathrm{Re}(h[N/2-k])\mathrm{Im}(e^{-2i\pi k/N})) \\ \mathrm{Im}(\hat{x}[k]) = \; & 1/2 \times (\mathrm{Im}(h[k]) - \mathrm{Im}(h[N/2-k]) - \mathrm{Re}(h[k])\mathrm{Re}(e^{-2i\pi k/N}) + \mathrm{Im}(h[k])\mathrm{Im}(e^{-2i\pi k/N})... \\ & ... + \mathrm{Re}(h[N/2-k])\mathrm{Re}(e^{-2i\pi k/N}) + \mathrm{Im}(h[N/2-k])\mathrm{Im}(e^{-2i\pi k/N}))\end{aligned}$$

Il y a un cas particulier où cette formule ne fonctionne pas: quand $k = 0$ on sort du tableau h qui ne contient que $N/2$ éléments. Cependant on peut utiliser la symétrie de la Transformée de Fourier pour voir que $h[N/2] = h[0]$. Le cas $k = 0$ se simplifie alors énormément:

$$\mathrm{Re}(\hat{x}[0]) = \mathrm{Re}(h[0]) + \mathrm{Im}(h[0])$$
$$\mathrm{Im}(\hat{x}[0]) = 0$$

Pour effectuer le calcul en place, il est utile d'être en mesure de calculer $\hat{x}[N/2-k]$ au même moment où on calcule $\hat{x}[k]$. En réutilisant les résultats précédents et le fait que $e^{-2i\pi(N/2-k)/N} = -e^{2i\pi k/N}$, on trouve:

$$\mathrm{Re}(X(N/2-n)) = 1/2 \times (\mathrm{Re}(h[N/2-k]) + \mathrm{Re}(h[k]) - \mathrm{Im}(h[N/2-k]]\mathrm{Re}(e^{-2i\pi k/N})... \\ ... + \mathrm{Re}(h[N/2-k])\mathrm{Im}(e^{-2i\pi k/N}) - \mathrm{Im}(h[k])\mathrm{Re}(e^{-2i\pi k/N}) - \mathrm{Re}(h[k])\mathrm{Im}(e^{-2i\pi k/N}))$$

$$\mathrm{Im}(X(N/2\text{-}n)) = 1/2 \times (\mathrm{Im}(h[N/2\text{-}k]) - \mathrm{Im}(h[k]) + \mathrm{Re}(h[N/2\text{-}k])\mathrm{Re}(e^{-2i\pi k/N})...$$

Après ce petit moment désagréable, nous sommes prêts à implémenter une nouvelle version de la FFT!

2.6. Une FFT pour les réels

Puisque le calcul effectif de la FFT se fait sur un tableau qui fait la moitié de la taille du tableau d'entrée, on a besoin d'une fonction pour calculer l'indice inversé sur 9 bits pour pouvoir continuer à tester sur 1024 points.

```
bit_reverse(::Val{9}, num) = begin
        num = ((num&0x1e0)>>5)|(num&0x010)|((num&0x00f)<<5)
        num = ((num&0x18c)>>2)|(num&0x010)|((num&0x063)<<2)
        ((num&0x14a)>>1)|(num&0x010)|((num&0x0a5)<<1)
end
```

Contenu masqué n°2

Pour tenir compte des spécificités de la représentation des complexes que l'on utilise, on implémente une nouvelle version de `reverse_bit_order`.

```
function reverse_bit_order_double!(x, order)
        N = length(x)
        for i in 0:(N-1)
                j = bit_reverse(order, i)
                if i<j
                        # swap real part
                        x[2*i+1],x[2*j+1]=x[2*j+1],x[2*i+1]
                        # swap imaginary part
                        x[2*i+2],x[2*j+2]=x[2*j+2],x[2*i+2]
                end
        end
        x
end
```

Ce qui mène à la nouvelle implémentation de la FFT.

```
function my_fft_3(x)
        N = length(x) ÷ 2
        order = Int(log2(N))
        @inbounds reverse_bit_order_double!(x, Val(order))

        n₁ = 0
        n₂ = 1
        for i=1:order # i done le numéro de la colonne dans lequel
            on se trouve.
                n₁ = n₂ # n₁ = 2^(i-1)
                n₂ *= 2 # n₂ = 2^i

                step_angle = -2π/n₂
                angle = 0
                for j=1:n₁ # j est l'indice dans Xᵉ et Xᵒ
                        re_factor = cos(angle)
                        im_factor = sin(angle)
                        angle += step_angle # a = -2π*j/n₂

                        # On combine l'élement j de chaque groupe
                            de sous-tableaux
                        @inbounds for k=j:n₂:N
                                re_x = x[2*k-1]
                                im_x = x[2*k]
                                re_x = x[2*(k+n₁)-1]
                                im_x = x[2*(k+n₁)]
                                x[2*k-1]      = re_x +
                                    re_factor*re_x -
                                    im_factor*im_x
                                x[2*k]        = im_x +
                                    im_factor*re_x +
                                    re_factor*im_x
                                x[2*(k+n₁)-1] = re_x -
                                    re_factor*re_x +
                                    im_factor*im_x
                                x[2*(k+n₁)]   = im_x -
                                    im_factor*re_x - re_factor*im_
                                    x
                        end
                end
        end
        # On construit la version finale de la TF
        # N la moitié de la taille de x
        # Cas particulier n=0
        x[1] = x[1] + x[2]
        x[2] = 0

        step_angle = -π/N
        angle = step_angle
```

```
        @inbounds for n=1:(N÷2)
                re_factor = cos(angle)
                im_factor = sin(angle)
                re_h = x[2*n+1]
                im_h = x[2*n+2]
                re_h_sym = x[2*(N-n)+1]
                im_h_sym = x[2*(N-n)+2]
                x[2*n+1]     = 1/2*(re_h + re_h_sym +
                    im_h*re_factor + re_h*im_factor +
                    im_h_sym*re_factor - re_h_sym*im_factor)
                x[2*n+2]     = 1/2*(im_h - im_h_sym -
                    re_h*re_factor + im_h*im_factor +
                    re_h_sym*re_factor + im_h_sym*im_factor)
                x[2*(N-n)+1] = 1/2*(re_h_sym + re_h -
                    im_h_sym*re_factor + re_h_sym*im_factor -
                    im_h*re_factor - re_h*im_factor)
                x[2*(N-n)+2] = 1/2*(im_h_sym - im_h +
                    re_h_sym*re_factor + im_h_sym*im_factor -
                    re_h*re_factor + im_h*im_factor)
                angle += step_angle
        end
        x
end
```

On peut maintenant vérifier les performances de la nouvelle implémentation:

```
@benchmark fft!(x) setup=(x = rand(1024) .|> complex)
```

```
BenchmarkTools.Trial:
  memory estimate:  2.34 KiB
  allocs estimate:  30
  --------------
  minimum time:     21.142 µs (0.00% GC)
  median time:      23.768 µs (0.00% GC)
  mean time:        33.451 µs (5.10% GC)
  maximum time:     10.136 ms (50.07% GC)
  --------------
  samples:          10000
  evals/sample:     1
```

```
@benchmark my_fft_3(x) setup=(x = rand(1024))
```

```
BenchmarkTools.Trial:
  memory estimate:  0 bytes
  allocs estimate:  0
  --------------
  minimum time:     29.906 µs (0.00% GC)
  median time:      30.471 µs (0.00% GC)
  mean time:        34.659 µs (0.00% GC)
  maximum time:     83.834 µs (0.00% GC)
  --------------
  samples:          10000
  evals/sample:     1
```

C'est un très bon résultat!

2.7. Optimisation des fonctions trigonométriques

Si on analyse l'exécution de `my_fft_3` à l'aide du *profiler* de Julia, on s'aperçoit que la plus grosse partie du temps est passée à calculer des fonctions trigonométriques et à créer les objets `StepRange` utilisés dans les boucles `for`. Le second problème peut être contourné facilement en utilisant des boucles `while`. Pour le premier, au détour de *Numerical Recipes* on peut lire (section 5.4 "*Recurrence Relations and Clenshaw's Recurrence Formula*", page 219 de la troisième édition) :

> If your program's running time is dominated by evaluating trigonometric functions, you are probably doing something wrong. Trig functions whose arguments form a linear sequence $\theta = \theta_0 + n\delta, n = 0, 1, 2...$, are efficiently calculated by the recurrence
>
> $$\begin{aligned} \cos(\theta + \delta) &= \cos\theta - [\alpha\cos\theta + \beta\sin\theta] \\ \sin(\theta + \delta) &= \sin\theta - [\alpha\sin\theta - \beta\cos\theta] \end{aligned}$$
>
> Where α and β are the precomputed coefficients
>
> $$\alpha = 2\sin^2\left(\frac{\delta}{2}\right) \quad \beta = \sin\delta$$

Contenu masqué n°3

Cette relation présente également des intérêts en termes de stabilité numérique. On peut directement implémenter une version finale de notre FFT en utilisant ces relations.

```
function my_fft_4(x)
    N = length(x) ÷ 2
    order = Int(log2(N))
```

```
    @inbounds reverse_bit_order_double!(x, Val(order))

    n₁ = 0
    n₂ = 1

        i=1
    while i<=order # i done le numéro de la colonne dans lequel on
        se trouve.
        n₁ = n₂ # n₁ = 2^(i-1)
        n₂ *= 2 # n₂ = 2^i

        step_angle = -2π/n₂
        α = 2sin(step_angle/2)^2
        β = sin(step_angle)
        cj = 1
        sj = 0
                j = 1
        while j<=n₁ # j est l'indice dans Xᵉ et Xᵒ
            # On combine l'élement j de chaque groupe de
                sous-tableaux
                        k = j
            @inbounds while k<=N
                re_x = x[2*k-1]
                im_x = x[2*k]
                re_x = x[2*(k+n₁)-1]
                im_x = x[2*(k+n₁)]
                x[2*k-1]      = re_x + cj*re_x - sj*im_x
                x[2*k]        = im_x + sj*re_x + cj*im_x
                x[2*(k+n₁)-1] = re_x - cj*re_x + sj*im_x
                x[2*(k+n₁)]   = im_x - sj*re_x - cj*im_x

                                k += n₂
            end
            # On calcule le prochain cosinus et le prochain sinus.
            cj, sj = cj - (α*cj + β*sj), sj - (α*sj-β*cj)
                        j+=1
        end
                i += 1
    end
    # On construit la version finale de la TF
    # N la moitié de la taille de x
    # Cas particulier n=0
    x[1] = x[1] + x[2]
    x[2] = 0

    step_angle = -π/N
    α = 2sin(step_angle/2)^2
    β = sin(step_angle)
    cj = 1
    sj = 0
```

```
        j = 1
    @inbounds while j<=(N÷2)
        # On calcule les cosinus et sinus avant le calcul principal ici pour compenser la première
        # étape de la boucle que l'on a sautée.
        cj, sj = cj - (α*cj + β*sj), sj - (α*sj-β*cj)

        re_h = x[2*j+1]
        im_h = x[2*j+2]
        re_h_sym = x[2*(N-j)+1]
        im_h_sym = x[2*(N-j)+2]
        x[2*j+1]     = 1/2*(re_h + re_h_sym + im_h*cj + re_h*sj + im_h_sym*cj - re_h_sym*sj)
        x[2*j+2]     = 1/2*(im_h - im_h_sym - re_h*cj + im_h*sj + re_h_sym*cj + im_h_sym*sj)
        x[2*(N-j)+1] = 1/2*(re_h_sym + re_h - im_h_sym*cj + re_h_sym*sj - im_h*cj - re_h*sj)
        x[2*(N-j)+2] = 1/2*(im_h_sym - im_h + re_h_sym*cj + im_h_sym*sj - re_h*cj + im_h*sj)

            j += 1
    end
    x
end
```

On peut vérifier que l'on obtient toujours le bon résultat:

```
let
        a = rand(1024)
        b = fft(a)
        c = my_fft_4(a)
        real.(b[1:end÷2]) ≈ c[1:2:end] && imag.(b[1:end÷2]) ≈ c[2:2:end]
end
```

```
true
```

En termes de performances, on est enfin parvenu à dépasser l'implémentation de référence!

```
@benchmark fft!(x) setup=(x = rand(1024) .|> complex)
```

```
BenchmarkTools.Trial:
  memory estimate:  2.34 KiB
  allocs estimate:  30
  --------------
  minimum time:     20.885 µs (0.00% GC)
  median time:      23.300 µs (0.00% GC)
  mean time:        31.784 µs (4.09% GC)
  maximum time:     8.072 ms (58.49% GC)
  --------------
  samples:          10000
  evals/sample:     1
```

```
@benchmark my_fft_4(x) setup=(x = rand(1024))
```

```
BenchmarkTools.Trial:
  memory estimate:  0 bytes
  allocs estimate:  0
  --------------
  minimum time:     15.112 µs (0.00% GC)
  median time:      15.395 µs (0.00% GC)
  mean time:        17.172 µs (0.00% GC)
  maximum time:     55.256 µs (0.00% GC)
  --------------
  samples:          10000
  evals/sample:     1
```

Conclusion

Si l'on compare les différentes implémentations proposées dans ce tutoriel ainsi que les deux implémentations de référence, puis que l'on trace les valeurs médianes de temps d'exécution, d'empreinte mémoire et de nombre d'allocations, on obtient le graphe suivant :

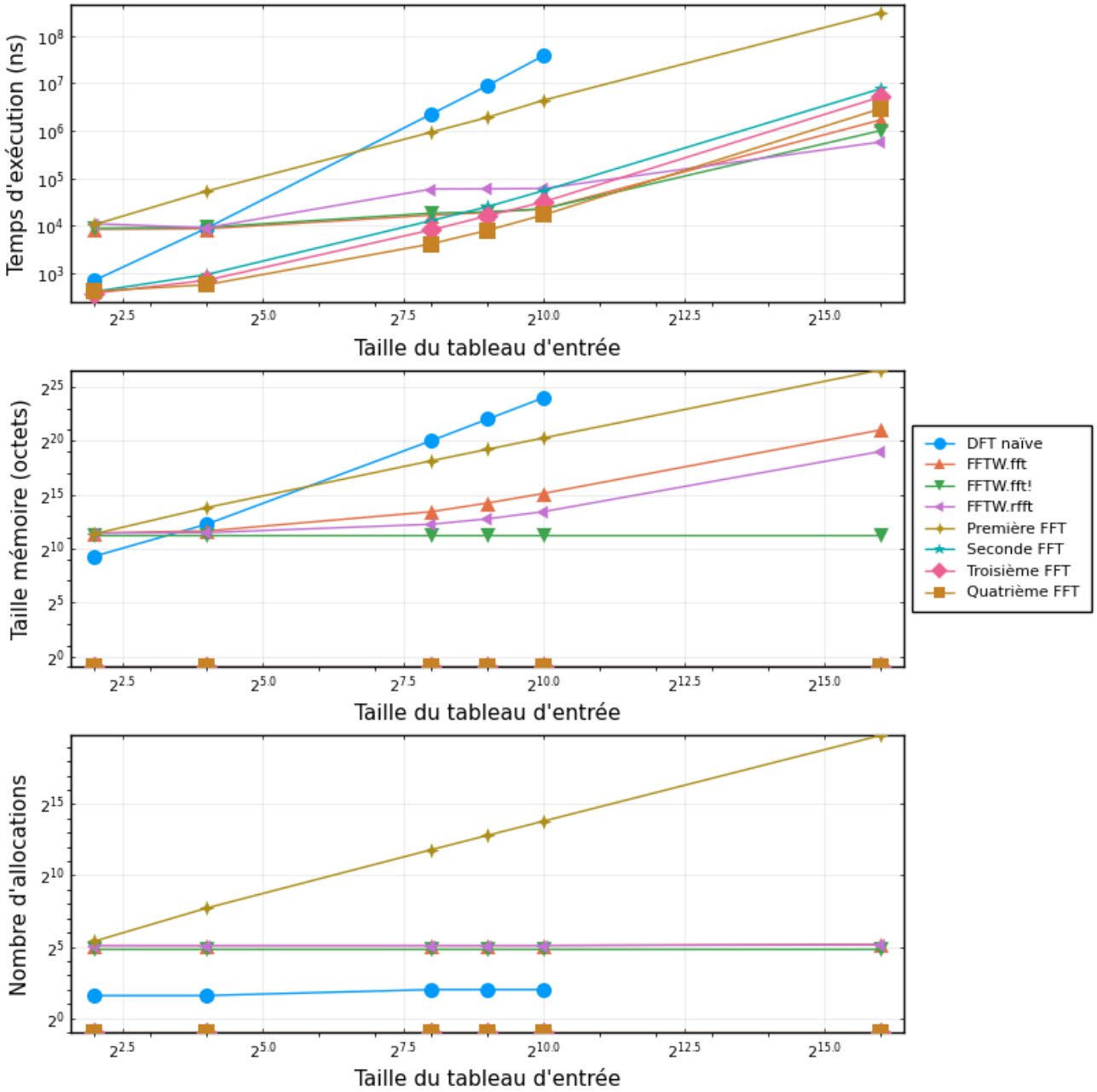

FIGURE 2.3. – Benchmark des différentes solutions: valeurs médianes.

J'ai ajouté la fonction `FFTW.rfft` qui est censée être optimisée pour les réels. On voit qu'en réalité, à moins de travailler sur de très gros tableaux, elle n'apporte pas énormément de performances.

On peut voir que les dernières versions de l'algorithme se défendent très bien en termes de nombre d'allocations et d'empreinte mémoire. Pour ce qui est du temps d'exécution, l'implémentation de référence finit par être plus rapide sur les très gros tableaux.

Comment peut-on expliquer ces différences, en particulier entre notre dernière implémentation et l'implémentation dans FFTW? Quelques éléments de réponse:

1. FFTW résout un problème bien plus large. En effet notre implémentation est "naïve" par exemple dans le sens où elle ne peut travailler que sur des tableaux d'entrée dont la

taille est une puissance de deux. Et encore, seulement ceux pour laquelle on s'est donné la peine d'implémenter une méthode de la fonction `bit_reverse`. Le problème de la permutation de bits inverse est un peu plus compliqué à résoudre dans le cas général. De plus FFTW assure une bonne exécution sur de nombreux types d'architectures, propose des transformées de Fourier discrètes en multiples dimensions etc… Si le sujet vous intéresse, je vous conseille cet article [5]footnote:1 qui présente le fonctionnement interne de FFTW.

2. La représentation des nombres complexes joue en notre faveur. En effet on évite à notre implémentation de devoir faire la moindre conversion, cela se voit en particulier dans les codes de test où l'on se charge de récupérer la partie réelle et la partie imaginaire de la transformée:

```
real.(b[1:end÷2]) ≈ c[1:2:end] && imag.(b[1:end÷2]) ≈ c[2:2:end]
```

1. Notre algorithme n'a pas été pensé avec la stabilité numérique en tête. C'est un aspect qui pourrait encore être amélioré. De même, nous ne l'avons pas testé sur autre chose que du bruit. Le bloc suivant présente cependant quelques tests qui laissent penser qu'il "se comporte bien" pour quelques fonctions de test.

Contenu masqué n°4

Ces simplifications et cas particuliers permettent à notre implémentation de gagner fortement en rapidité. Cela rend d'autant plus remarquable l'implémentation de FFTW qui s'en sort tout de même très bien!

1. [6]footnote:1 Frigo, Matteo & Johnson, S.G.. (2005). The Design and implementation of FFTW3. Proceedings of the IEEE. 93. 216 - 231. 10.1109/JPROC.2004.840301.

Conclusion

Au terme ce tutoriel j'espère vous avoir aidé à comprendre les mécanismes qui font fonctionner le calcul de la FFT, et montré comment l'implémenter efficacement, modulo quelques simplifications. Personnellement, écrire ce tutoriel m'a permis de me rendre compte des grandes qualités de FFTW, l'implémentation de référence, que j'utilise au quotidien dans mon travail!

Cela devrait vous permettre de comprendre que pour certains cas d'utilisation, il peut être intéressant d'implémenter et d'optimiser sa propre FFT. Une application qui a peu été discutée dans ce tutoriel est le calcul de produits de convolution. Une méthode efficace dans le cas où on convolue des signaux de longueur comparable est de le faire un multipliant les deux transformées de Fourier puis en prenant la transformée de Fourier inverse. Dans ce cas, puisque la multiplication se fait terme à terme, il n'est pas nécessaire que la transformée de Fourier soit ordonnée. On pourrait donc imaginer une implémentation spéciale qui sauterait la partie permutation de bits inverse.

Une autre amélioration que l'on pourrait apporter concerne le calcul de la transformée de Fourier inverse. Il s'agit d'un calcul très similaire (seuls les coefficients multiplicatifs changent), et peut constituer un bon exercice pour expérimenter avec les codes donnés dans ce tutoriel.

Je tiens enfin à remercier @Gawaboumga, @Næ, @zeqL et @luxera pour leurs retours sur la bêta de ce tutoriel, et @Gabbro pour la validation!

Contenu masqué

Contenu masqué n°1

```
1 bit_reverse(::Val{64}, num) = bit_reverse(Val(32),
      (num&0xffffffff00000000)>>32)|(bit_reverse(Val(32),
      num&0x00000000ffffffff)<<32)
2 bit_reverse(::Val{32}, num) = bit_reverse(Val(16),
      (num&0xffff0000)>>16)|(bit_reverse(Val(16),
      num&0x0000ffff)<<16)
3 bit_reverse(::Val{16}, num) = bit_reverse(Val(8),
      (num&0xff00)>>8)|(bit_reverse(Val(8), num&0x00ff)<<8)
4 bit_reverse(::Val{8}, num) = bit_reverse(Val(4),
      (num&0xf0)>>4)|(bit_reverse(Val(4), num&0x0f)<<4)
5 bit_reverse(::Val{4}, num) =bit_reverse(Val(2),
      (num&0xc)>>2)|(bit_reverse(Val(2), num&0x3)<<2)
```

```
bit_reverse(::Val{3}, num) =
    ((num&0x1)<<2)|((num&0x4)>>2)|(num&0x2)
bit_reverse(::Val{2}, num) = ((num&0x2)>>1 )|((num&0x1)<<1)
bit_reverse(::Val{1}, num) = num
```

Retourner au texte.

Contenu masqué n°2

Pour compléter les autres méthode de `bit_reverse` on peut utiliser les implémentations suivantes:

```
bit_reverse(::Val{31}, num) = begin
        bit_reverse(Val(15), num&0x7fff0000>>16)| (num&0x00008000)
            |(bit_reverse(Val(7),num&0x00007fff)<<16)
end
bit_reverse(::Val{15}, num) = bit_reverse(Val(7),
    (num&0x7f00)>>8)|
    (num&0x0080)|(bit_reverse(Val(7),num&0x007f)<<8)
bit_reverse(::Val{7}, num) = bit_reverse(Val(3), (num&0x70)>>4 )|
    (num&0x08) |(bit_reverse(Val(3), num&0x07)<<4)
```

Retourner au texte.

Contenu masqué n°3

On peut montrer cela en utilisant les identitées trigonométriques classiques:

$$\begin{aligned}
\cos(\theta+\delta) &= \cos\theta\cos\delta - \sin\theta\sin\delta \\
&= \cos\theta\left[2\cos^2\frac{\delta}{2}-1\right] - \sin\theta\sin\delta \\
&= \cos\theta\left[2(1-\sin^2\frac{\delta}{2})-1\right] - \sin\theta\sin\delta \\
&= \cos\theta - [\underbrace{\sin^2\frac{\delta}{2}}_{=\alpha}\cos\theta + \underbrace{\sin\delta}_{=\beta}\sin\theta]
\end{aligned}$$

Et avec $\sin x = \cos(x-\frac{\pi}{2})$, on a directement la seconde formule. Retourner au texte.

Contenu masqué n°4

```
function test_signal(s)
        b = fft(s)
         c = my_fft_7_5(s)
         real.(b[1:end÷2]) ≈ c[1:2:end] && imag.(b[1:end÷2]) ≈
    c[2:2:end]
end

let
         t = range(-10, 10; length=1024)
         y = @. exp(-t^2)
         noise = rand(1024)
         test_signal(y .+ noise)
 end
```

```
true
```

```
let
        t = range(-10, 10; length=1024)
        y = @. sin(t)
         noise = rand(1024)
         test_signal(y .+ noise)
end
```

```
true
```

Retourner au texte.