Autour de la transformation de Fourier rapide

Samuel Gallay[1] et Ludovic Arnaud[2]

[1]samuel.gallay@ens-rennes.fr
[2]ludovic.arnaud@ens-rennes.fr

26 mai 2022

Rapport de lectures dirigées de L3
sous la supervision de Stéphane Balac

Table des matières

1 Introduction

Dans ce rapport, nous étudions l'article Bailey et Swarztrauber (1994) qui présente un algorithme pour évaluer numériquement efficacement une transformée de Fourier continue. Pour fixer les notations, posons $f : \mathbb{R} \to \mathbb{C}$ une fonction continue et intégrable sur $\mathbb{R}$ et définissons sa transformée de Fourier $\hat{f} : \mathbb{R} \to \mathbb{C}$.

$$\hat{f}(x) = \int_{-\infty}^{+\infty} f(t)e^{-itx}dt$$

Le calcul de $\hat{f}(x)$ demande d'approcher numériquement une intégrale. Ceci se fait par l'utilisation d'une méthode de quadrature, comme la méthode des rectangles ou la méthode de Simpson. Si de plus on suppose f nulle en dehors de l'intervalle $[-\frac{a}{2}, \frac{a}{2}]$, on a le résultat suivant :

$$\lim_{n\to\infty} \frac{a}{n} \sum_{j=0}^{n-1} f(-\frac{a}{2} + j\frac{a}{n})e^{-i(-\frac{a}{2}+j\frac{a}{n})x} = \int_{-\frac{a}{2}}^{+\frac{a}{2}} f(t)e^{-itx}dt = \hat{f}(x)$$

En pratique, on ne connaît pas forcément f sur $\mathbb{R}$ tout entier, mais seulement en n points de mesure $(t_j)_{1\leq j\leq n}$, que nous supposerons ici uniformément répartis sur l'intervalle $[-\frac{a}{2}, \frac{a}{2}]$, i.e.

$$t_j = -\frac{a}{2} + j\frac{a}{n}$$

En supposant f négligeable hors de $[-\frac{a}{2}, \frac{a}{2}]$, nous arrivons à cette expression approchée de $\hat{f}(x)$ que nous cherchons à évaluer rapidement à l'aide d'un ordinateur :

$$\hat{f}(x) \approx \frac{a}{n} \sum_{j=0}^{n-1} f(t_j)e^{-it_j x} \tag{1}$$

Si l'on souhaite évaluer $\hat{f}$ en un seul point x, l'évaluation de la somme se fait en $O(n)$. On se demande si étant donné un ensemble de m points $(x_k)_{1\leq k\leq m}$, on est capable d'évaluer les $(\hat{f}(x_k))_{1\leq k\leq m}$ avec une complexité meilleure que $O(n \times m)$. Nous ne nous intéresserons qu'au cas très particulier où $m = n$ et où les x_k sont répartis uniformément sur l'intervalle $[-\frac{b}{2}, \frac{b}{2}]$, i.e.

$$x_k = -\frac{b}{2} + k\frac{b}{n}$$

Avec ces paramètres, l'algorithme naïf calculant les $\hat{f}(x_k)$ avec la formule (1) fonctionne en $O(n^2)$:

```
import numpy as np

def cft(f, a, b, n):
    result = np.zeros(n, dtype=complex)
    for k in range(n):
        xk = -b/2 + k*b/n
        for j in range(n):
            tj = -a/2 + j*a/n
            result[k] += f(tj) * np.exp(-1j*tj*xk)
    return a/n * result
```

Toujours avec ces paramètres, il existe un algorithme classique de complexité $O(n\log(n))$ utilisant la transformation de Fourier rapide, que nous étudierons dans un premier temps. Le problème de cet algorithme est qu'il impose une relation de dépendance entre n, a, et b.

$$ab = 2\pi n \tag{2}$$

Ainsi, si n et a sont fixés par l'expérience par exemple, la valeur de b est imposée par l'algorithme. L'article Bailey et Swarztrauber (1994) que nous étudions propose une méthode qui enlève cette contrainte. Pour ce

faire, ils utilisent la transformation de Fourier fractionnaire qu'ils introduisent dans un article précédent Bailey et Swarztrauber (1991).

Dans une première partie, nous allons d'abord étudier l'algorithme de la transformation de Fourier rapide, et nous l'utiliserons pour évaluer les $\hat{f}(x_k)$ sous la contrainte (2). Ensuite, nous étudierons la transformation de Fourier fractionnaire de Bailey et Swarztrauber (1991) et enfin nous l'utilisons comme dans Bailey et Swarztrauber (1994) pour calculer les $\hat{f}(x_k)$ sans la contrainte (2). Dans une seconde partie, nous nous intéresserons à la simulation d'équations aux dérivées partielles avec la transformation de Fourier discrète, et à l'utilisation dans ce cadre de la transformation de Fourier discrète fractionnaire.

2 La transformation de Fourier discrète

2.1 Présentation

Commencons par définir la transformation de Fourier discrète. Celle-ci associe à une suite n-périodique $(z_j)_{j\in\mathbb{Z}}$ de nombres complexes sa transformée, une nouvelle suite $(D_k(z))_{k\in\mathbb{Z}}$ définie par :

$$D_k(z) = \sum_{j=0}^{n-1} z_j e^{-\frac{2i\pi jk}{n}} \tag{3}$$

On constate immédiatement que cette nouvelle suite est elle aussi n-périodique. Ainsi, un ordinateur peut donc se restreindre au calcul d'une période. Puisque nous pouvons identifier les suites n-périodiques et les séquences à n éléments, nous pouvons considérer la transformation de Fourier discrète comme un algorithme agissant sur une des séquences de n termes. À $(z_j)_{0\leq j\leq n-1}$, l'algorithme associe $(D_k(z))_{0\leq k\leq n-1}$. Cette transformation est en de nombreux points un analogue discret de la transformation de Fourier des fonctions intégrables de $\mathbb{R}$ dans $\mathbb{R}$.

Intéressons nous d'abord au calcul numérique d'une transformée à n éléments. Bien évidemment, il est possible de calculer $(D_k(z))_{0\leq k\leq n-1}$ en utilisant directement la formule (3), comme le fait l'algorithme ci-dessous.

```
def dft(Z):
    n = len(Z)
    L = np.zeros_like(Z, dtype=complex)
    for k in range(n):
        for j in range(n):
            L[k] += Z[j]*np.exp(-2*1j*np.pi*j*k/n)
    return L
```

Cet algorithme possède un coût de calcul en $O(n^2)$.

2.2 L'algorithme de la transformation de Fourier rapide

Ce qui a rendu la transformation de Fourier discrète si populaire en informatique, est l'existence d'algorithmes en $O(n\log(n))$ pour la calculer. Ces algorithmes portent le nom de *transformation de Fourier rapide*, abrégé FFT, le sigle venant de l'anglais *Fast Fourier Transform*. Le premier algorithme de FFT a été présenté par . Celui-ci ne fonctionne que dans le cas où n est une puissance de 2, mais nous verrons plus loin comment celui-ci peut être étendu en un algorithme de même complexité asymptotique pour les autres valeurs possibles de n. L'algorithme de Cooley et Tukey (1965) se base sur le calcul suivant :

$$
\begin{aligned}
D_k(z) &= \sum_{j=0}^{n-1} z_j e^{-\frac{2i\pi jk}{n}} \\
&= \sum_{j=0}^{\frac{n}{2}-1} z_{2j} e^{-\frac{2i\pi (2j)k}{n}} + \sum_{j=0}^{\frac{n}{2}-1} z_{2j+1} e^{-\frac{2i\pi (2j+1)k}{n}} \\
&= \sum_{j=0}^{\frac{n}{2}-1} z_{2j} e^{-\frac{2i\pi jk}{\frac{n}{2}}} + e^{\frac{-2ik\pi}{n}} \sum_{j=0}^{\frac{n}{2}-1} z_{2j+1} e^{-\frac{2i\pi jk}{\frac{n}{2}}}
\end{aligned}
$$

La deuxième égalité ci-dessous a été obtenue en séparant les termes pairs et les termes impairs de la somme. On reconnait ensuite les transformées de Fourier des termes pairs et des termes impairs :

$$
D_k(z) = D_k\left((z_{2j})_{0\leq j\leq \frac{n}{2}-1}\right) + e^{\frac{-2ik\pi}{n}} D_k\left((z_{2j+1})_{0\leq j\leq \frac{n}{2}-1}\right)
$$

Cette égalité est vraie pour tout $k \in \mathbb{Z}$ si l'on considère les deux sous transformées de Fourier comme des suites $\frac{n}{2}$-périodiques. On peut arranger l'équation pour n'évaluer les transformées qu'à l'intérieur d'une période :

$$
\begin{aligned}
\text{Pour } 0 \leq k \leq \frac{n}{2} - 1, & \\
D_k(z) &= D_k\left((z_{2j})_{0\leq j\leq \frac{n}{2}-1}\right) + e^{\frac{-2ik\pi}{n}} D_k\left((z_{2j+1})_{0\leq j\leq \frac{n}{2}-1}\right) \\
D_{\frac{n}{2}+k}(z) &= D_k\left((z_{2j})_{0\leq j\leq \frac{n}{2}-1}\right) - e^{\frac{-2ik\pi}{n}} D_k\left((z_{2j+1})_{0\leq j\leq \frac{n}{2}-1}\right)
\end{aligned} \tag{4}
$$

Cette relation de récurrence suggère un algorithme de type *diviser pour régner* pour calculer $(D_k(z))_{0\leq k\leq n-1}$. C'est ce que fait la fonction `fft` ci-dessous.

```python
def fft(Z):
    n = len(Z)
    if n <= 1:
        return Z.copy()
    else:
        assert(n % 2 == 0)
        even = fft(Z[::2])
        odd = fft(Z[1::2])
        result = np.zeros_like(Z, dtype=complex)
        factors = np.exp(-2*1j*np.pi*np.arange(n//2)/n)
        result[:n//2] = even + factors * odd
        result[n//2:] = even - factors * odd
        return result
```

Intéressons nous maintenant à la complexité asymptotique de l'algorithme ci-dessus. En notant $C(n)$ la complexité du calcul d'une transformée de taille n, la relation de récurrence (4) et l'algorithme donnent :

$$
C(n) = 2C\left(\frac{n}{2}\right) + \Theta(n) \tag{5}
$$

Le premier terme vient du calcul des 2 transformées de taille $\frac{n}{2}$, et le second vient des copies et de la multiplication par le préfacteur exponentiel, qui se font en $\Theta(n)$ opération. Cet algorithme est un cas d'école d'application du théorème général d'analyse de la complexité des algorithmes de type diviser pour régner. Celui-ci montre que $C(n) = \Theta(n\log(n))$.

L'algorithme de Cooley et Tukey (1965) est le premier permettant de calculer une transformation de Fourier discrète en un temps $O(n\log(n))$. C'est la meilleure complexité asymptotique connue à ce jour, mais on ne sait pas, à ma connaissance, si elle est optimale. L'implémentation en Python proposée ci-dessus est une implémentation jouet. Elle n'est pas *en place* et requiert de nombreuses copies.

Ce type de programmes est généralement écrit dans des langages de plus bas niveau, comme le C ou le Fortran, où l'utilisateur a un contrôle plus direct sur la mémoire de l'ordinateur. Un programme populaire optimisé pour le calcul de la FFT est FFTW, dont le design de la troisième version est présenté dans Frigo et Johnson (2005). Une des particulatité de ce programme est qu'il s'agit d'un compilateur écrit dans un langage de haut niveau, OCaml, pour générer un programme de bas niveau, en C, optimisé pour le calcul de transformations de Fourier discrètes d'une taille fixée. Cette approche innovante décrite dans Frigo (1999) pose de nombreux problèmes intéressants en informatique. Une telle implémentation est typiquement 1000 fois plus rapide que mon implémentation en Python.

2.3 La transformation inverse

Comme la transformation de Fourier continue, la transformation de Fourier discrète possède une formule d'inversion. Définissons la transformation de Fourier discrète inverse d'une séquence z par la formule suivante :

$$D_k^{-1}(z) = \frac{1}{n}\sum_{j=0}^{n-1} z_j e^{\frac{2i\pi jk}{n}}$$

Montrons maintenant qu'avec cette définition, D_k^{-1} joue le rôle de l'inverse de D_k :

$$\begin{aligned} D_k^{-1}\left((D_j(z))_{0\leq n-1}\right) &= \frac{1}{n}\sum_{j=0}^{n-1} D_j(z) e^{\frac{2i\pi jk}{n}} = \frac{1}{n}\sum_{j=0}^{n-1}\sum_{m=0}^{n-1} z_m e^{-\frac{2i\pi mj}{n}} e^{\frac{2i\pi jk}{n}} \\ &= \frac{1}{n}\sum_{j=0}^{n-1}\sum_{m=0}^{n-1} z_m e^{\frac{2i\pi(k-m)}{n}j} = \sum_{m=0}^{n-1} z_m \frac{1}{n}\sum_{j=0}^{n-1} e^{\frac{2i\pi(k-m)}{n}j} \end{aligned}$$

Or

$$\sum_{j=0}^{n-1} e^{\frac{2i\pi(k-m)}{n}j} = \begin{cases} \dfrac{1-e^{\frac{2i\pi(k-m)}{n}n}}{1-e^{\frac{2i\pi(k-m)}{n}}} = 0 \text{ si } k \neq m \\ n \text{ si } k = m \end{cases}$$

Ainsi,

$$D_k^{-1}\left((D_j(z))_{0\leq n-1}\right) = \sum_{m=0}^{n-1} z_m \delta_{k=m} = z_k$$

Nous avons donc maintenant la possibilité d'inverser la transformation de Fourier discrète. Comme pour la transformation dirècte, il est possible d'écrire l'algorithme naïvement en $O(n^2)$:

```
def idft(Z):
    n = len(Z)
    L = np.zeros_like(Z, dtype=complex)
    for k in range(n):
        for j in range(n):
            L[k] += (1/n) * Z[j] * np.exp(2*1j*np.pi*j*k/n)
    return L
```

Il est aussi possible d'adapter l'algorithme de la transformation de Fourier rapide, pour calculer la transformation inverse en $O(n\log(n))$ opérations :

```
def ifft(Z):
    n = len(Z)
    if n <= 1:
        return Z.copy()
    else:
```

```
    assert(n % 2 == 0)
    even = ifft(Z[::2])
    odd = ifft(Z[1::2])
    result = np.zeros_like(Z, dtype=complex)
    factors = np.exp(2*1j*np.pi*np.arange(n//2)/n)
    result[:n//2] = even + factors * odd
    result[n//2:] = even - factors * odd
    return (1/2) * result
```

2.4 Théorème de convolution

D'une manière analogue au cadre de la transformée de Fourier continue, ou au cadre du développement en séries de Fourier, il existe un lien fort entre le produit de convolution de deux suites n-périodiques et la transformation de Fourier discrète. Définissons d'abord le produit de convolution $x * y$ de deux suites n-périodiques x et y par :

$$(x * y)_l = \sum_{j=0}^{n-1} x_j y_{l-j}$$

Notons tout de suite que la suite $x * y$ est elle aussi n-périodique. Ainsi, numériquement nous pouvons nous restreindre à stocker une seule période de la suite dans un tableau à n éléments. Lorsque x et y sont des suites finies de taille n, on appelle alors *convolution circulaire* des séquences x et y la séquence $((x * y)_l)_{0 \leq l \leq n-1}$ où x et y ont été n-périodisées. Voici d'abord un algorithme naïf en $O(n^2)$:

```
def conv(x, y):
    assert(len(x) == len(y))
    n = len(x)
    L = np.zeros_like(x)
    for k in range(n):
        for j in range(n):
            L[k] += x[j] * y[(k-j)%n]
    return L
```

Le résultat fondamental ci-dessous est parfois appelé *théorème de convolution.*

$$D_k(x * y) = D_k(x) \cdot D_k(y) \tag{6}$$

Ce résultat analogue à celui des séries de Fourier a une démonstration élémentaire puisque toutes les sommes manipulées sont finies. Il s'agit donc uniquement d'un réarrangemet astucieux des termes de la double somme de sorte à faire apparaître les transformées de x et de y.

$$\begin{aligned} D_k(x * y) &= \sum_{l=0}^{n-1} (x * y)_l e^{-\frac{2i\pi}{n} kl} = \sum_{l=0}^{n-1} \sum_{j=0}^{n-1} x_j y_{l-j} e^{-\frac{2i\pi}{n} kl} \\ &= \sum_{j=0}^{n-1} x_j \sum_{l=0}^{n-1} y_{l-j} e^{-\frac{2i\pi}{n} kl} = \sum_{j=0}^{n-1} x_j e^{-\frac{2i\pi}{n} jk} \sum_{l=0}^{n-1} y_{l-j} e^{-\frac{2i\pi}{n} k(l-j)} \\ &= \sum_{j=0}^{n-1} x_j e^{-\frac{2i\pi}{n} jk} \sum_{l=0}^{n-1} y_l e^{-\frac{2i\pi}{n} kl} = D_k(x) \cdot D_k(y) \end{aligned}$$

Ce résultat nous permet d'implémenter un algorithme rapide pour calculer la convolution circulaire de deux séquences. En appliquant la transformation de Fourier discrète inverse à l'équation (6), on obtient :

$$(x * y)_l = D_l^{-1}((D_k(x) \cdot D_k(y))_{1 \leq k \leq n})$$

À l'aide de l'algorithme de transformée de Fourier rapide, la convolution circulaire de x et y peut donc se calculer en $O(n \log n)$ opérations :

```python
def conv_fft(x, y):
    assert(len(x) == len(y))
    return ifft(fft(x) * fft(y))
```

3 Transformation de Fourier fractionnaire

3.1 Définition

Nous avons maintenant tous les outils pour comprendre la transformation de Fourier fractionnaire telle qu'elle est définie dans Bailey et Swarztrauber (1991). Pour α réel et x une suite n-périodique, ou de manière équivalente x une séquence de n nombres complexes, la transformée de Fourier fractionnaire de x est la séquence $(G_k(x,\alpha))_{0\leq k\leq n-1}$ définie par :

$$G_k(x,\alpha) = \sum_{j=0}^{n-1} x_j e^{-2i\pi jk\alpha}$$

Voici un programme qui calcule en temps $O(n^2)$ la transformée de Fourier fractionnaire d'une séquence x de longueur n.

```python
def frac_dft(X, a):
    n = len(X)
    L = np.zeros_like(X, dtype=complex)
    for k in range(n):
        for j in range(n):
            L[k] += X[j]*np.exp(-2*1j*np.pi*j*k*a)
    return L
```

Cependant, il est possible de calculer cette transformée en un temps $O(n\log n)$, en utilisant la transformation de Fourier rapide.

3.2 Calcul par la transformation de Fourier rapide

Bailey et Schwarztrauber utilisent une technique décrite par Bluestein dans l'article Bluestein (1970) pour calculer la transformée fractionnaire. Par une identité remarquable, on réécrit $-2jk = -k^2 - j^2 + (k-j)^2$. Ainsi,

$$\begin{aligned} G_k(x,\alpha) &= \sum_{j=0}^{n-1} x_j e^{-2i\pi jk\alpha} \\ &= \sum_{j=0}^{n-1} x_j e^{-i\pi\alpha k^2} e^{-i\pi\alpha j^2} e^{i\pi\alpha(k-j)^2} \\ &= e^{-i\pi\alpha k^2} \sum_{j=0}^{n-1} y_j z_{k-j} \end{aligned}$$

En posant $y_j = x_j e^{-i\pi\alpha j^2}, z_j = e^{i\pi\alpha j^2}$. Attention, bien que l'expression à évaluer ressemble fortement à une convolution circulaire, ce n'en est pas une. Puisque elle est évaluée pour $0 \leq j \leq n-1$, la séquence y est bien de taille n. Par contre, la séquence z est évaluée pour des indices $-(n-1) \leq j \leq n-1$ et $z_j \neq z_{n+j}$. On ne peut donc pas utiliser directement une convolution discrète circulaire.

Pour palier à ce problème Bluestein propose de d'allonger la séquence y avec des 0 en une suite de longueur au moins $2n-1$ et d'adapter z pour pouvoir effectivement appliquer une transformation de Fourier discrète circulaire. Dans notre cas particulier, nous allons transformer y et z en deux séquences Y et Z de lonqueur $2n$ de la sorte :

$$Y_j = \begin{cases} x_j e^{-i\pi\alpha j^2} & \text{si } 0 \leq j \leq n-1 \\ 0 & \text{si } n \leq j \leq 2n-1 \end{cases} \text{ et } Zj = \begin{cases} e^{-i\pi\alpha j^2} & \text{si } 0 \leq j \leq n-1 \\ 0 & \text{si } j = n \\ e^{i\pi\alpha(2n-j)^2} & \text{si } n+1 \leq j \leq 2n-1 \end{cases} \tag{7}$$

Avec ces définitions, pour $0 \leq k \leq n-1$:

$$\begin{aligned}
(Y * Z)_k &= \sum_{j=0}^{2n-1} Y_j Z_{k-j} = \sum_{j=0}^{n-1} x_j e^{-i\pi\alpha j^2} Z_{k-j} \\
&= \sum_{j=0}^{k} x_j e^{-i\pi\alpha j^2} Z_{k-j} + \sum_{j=k+1}^{n-1} x_j e^{-i\pi\alpha j^2} Z_{k-j} \\
&= \sum_{j=0}^{k} x_j e^{-i\pi\alpha j^2} Z_{k-j} + \sum_{j=k+1}^{n-1} x_j e^{-i\pi\alpha j^2} Z_{2n+k-j} \\
&= \sum_{j=0}^{k} x_j e^{-i\pi\alpha j^2} e^{-i\pi\alpha (k-j)^2} + \sum_{j=k+1}^{n-1} x_j e^{-i\pi\alpha j^2} e^{-i\pi\alpha (k-j)^2} \\
&= \sum_{j=0}^{n-1} x_j e^{-i\pi\alpha j^2} e^{-i\pi\alpha (k-j)^2}
\end{aligned}$$

Ainsi,

$$G_k(x, \alpha) = e^{-i\pi\alpha k^2} \cdot (Y * Z)_k \tag{8}$$

Finalement, l'équation (8) nous permet d'écrire un algorithme en temps $O(n \log n)$ pour calculer la transformée de Fourier discrète fractionnaire d'une séquence x de longueur n, en utilisant la convolution circulaire basée sur la transformation de Fourier rapide.

```
def frac_fft(X, a):
    n = len(X)
    j = np.arange(n)
    y = X * np.exp(-1j*np.pi*j*j*a)
    z = np.exp(1j*np.pi*j*j*a)

    Y = np.zeros(2*n, dtype=complex)
    Z = np.zeros(2*n, dtype=complex)
    Y[:n] = y
    Z[:n] = z
    Z[n+1:] = z[:0:-1]
    return np.exp(-1j*np.pi*j*j*a) * conv_fft(Y, Z)[:n]
```

Si l'on s'intéresse de plus près au coût de calcul d'une transformée de Fourier fractionnaire, on remarque que les seules opérations en $O(n \log(n))$ sont celles qui apparaissent dans `conv_fft(Y, Z)`, qui est définie comme `ifft(fft(Y) * fft(Z))`. Or par (7), seule Y dépend effectivement de l'entrée x : Z_j ne dépend que de α et de n. Cela siginifie que si l'on a besoin de calculer plusieurs transformées de Fourier fractionnaire pour les mêmes valeurs de n et α, on peut précalculer `fft(Z)`. Ainsi, le coût d'une transformée fractionnaire est le coût de deux transformées de Fourier rapides de taille $2n$, soit environ le coût de 4 transformées de Fourier rapides de taille n.

3.3 Calcul de transformations de Fourier de tailles premières

Dans notre présentation de la transformation de Fourier rapide 2.2, nous n'avons écrit d'algorithme que dans le cas où la longueur n de la séquence était une puissance de 2. Une généralisation de l'algorithme Cooley et Tukey (1965) permet de calculer les FFT de séquence de taille $n = n_1 n_2$ en calculant d'abord n_1 FFT de taille n_2 puis en multipliant par le facteur de rotation, et enfin en calculant n_2 FFT de taille n_1. Cette approche permet de décomposer efficacement le calcul d'une FFT de taille n en le calcul de FFT de taille p, où p est un nombre premier.

La technique de Bluestein (1970) permet de ramener le calcul d'une transformation de z de taille n quelconque au calcul de transformations de taille m où $m \geq 2n$. L'idée est exactement comme dans la section 3.2 d'exprimer

$D_k(z)$ comme un produit de convolution, puis d'augmenter la taille des séquences jusqu'à une taille m de sorte à rendre le produit de convolution circulaire. Puisque le choix de $m \geq 2n$ est arbitraire, on peut par exemple choisir la plus petite puissance de 2 supérieure à $2n$. On construit ainsi un algorithme en $O(n\log(n))$ pour calculer des transformations de Fourier discrètes de toutes tailles.

4 Transformation de Fourier continue

4.1 Méthode classique par la FFT

Essayons de calculer les $\hat{f}(x_k)$ à partir des $f(t_j)$ en utilisant directement l'agorithme de la transformation de Fourier rapide.

$$\begin{aligned}
\hat{f}(x_k) &\approx \frac{a}{n}\sum_{j=0}^{n-1} f(t_j)e^{-it_jx_k} \\
&= \frac{a}{n}\sum_{j=0}^{n-1} f(t_j)e^{-i(-\frac{a}{2}+j\frac{a}{n})(-\frac{b}{2}+k\frac{b}{n})} \\
&= \frac{a}{n}\sum_{j=0}^{n-1} f(t_j)e^{-i\frac{ab}{4}}e^{i\frac{abk}{2n}}e^{i\frac{abj}{2n}}e^{-i\frac{abkj}{n^2}} \\
&= \frac{a}{n}e^{-i\frac{ab}{4}}e^{i\frac{abk}{2n}}\sum_{j=0}^{n-1} f(t_j)e^{i\frac{abj}{2n}}e^{-i\frac{abkj}{n^2}} \\
\hat{f}(x_k) &\approx \frac{a}{n}e^{i\frac{a}{2}x_k}\sum_{j=0}^{n-1} f(t_j)e^{i\frac{ab}{2n}j}e^{-i\frac{ab}{n^2}kj} \qquad (9)
\end{aligned}$$

Mise sous cette forme, l'expression de $\hat{f}(x_k)$ ressemble fortement à celle d'une transformée de Fourier discrète, avec $-i\frac{ab}{n^2}kj = -\frac{2i\pi jk}{n}$, c'est à dire $\frac{ab}{n^2} = \frac{2\pi}{n}$, soit $ab = 2\pi n$. Nous avons retrouvé l'origine de la contrainte (2). Ainsi, sous cette contrainte, $\hat{f}(x_k)$ se réécrit :

$$\hat{f}(x_k) \approx \frac{a}{n}e^{i\frac{a}{2}x_k}D_k\left(\left(f(t_j)e^{i\frac{abj}{2n}}\right)_{0\leq j\leq n-1}\right)$$

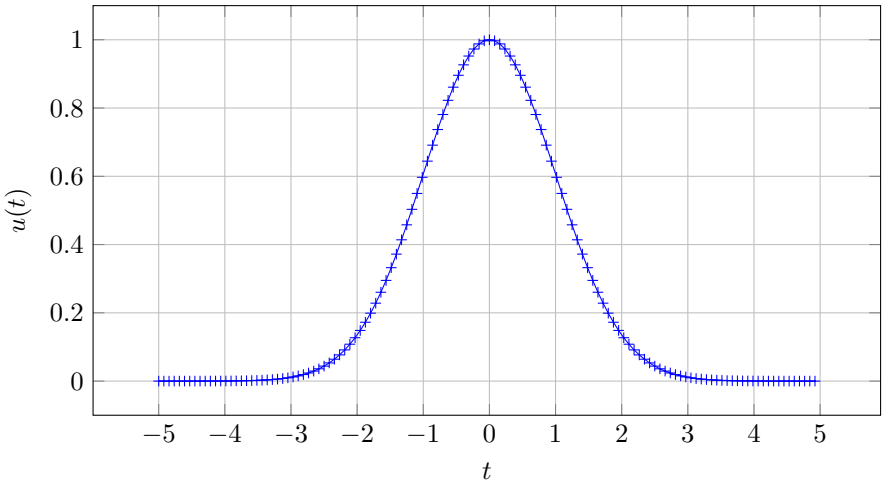
Et après simplification en utilisant la contrainte (2) :

$$\hat{f}(x_k) \approx \frac{a}{n}(-i)^n(-1)^kD_k\left(\left((-1)^jf(t_j)\right)_{0\leq j\leq n-1}\right) \qquad (10)$$

4.2 Problèmes

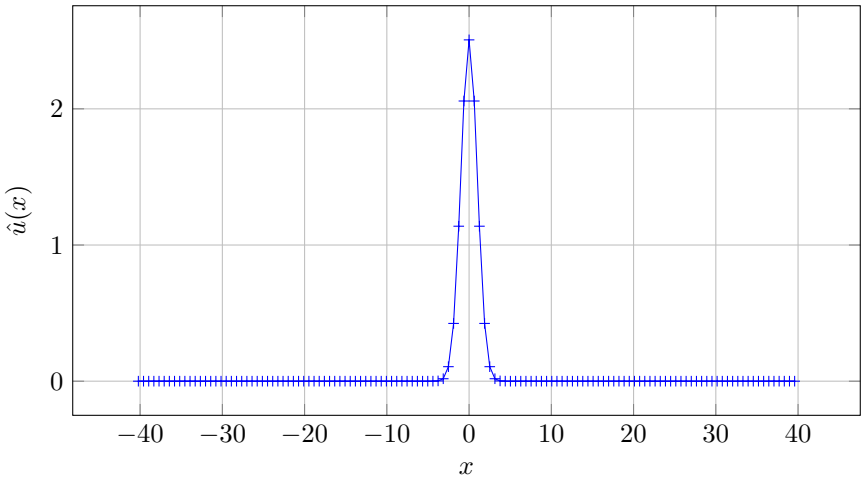
L'équation (10) permet de calculer explicitement les $\hat{f}(x_k)$ sous l'hypothèse de la contrainte (2). Essayons de comprendre pourquoi cette contrainte (2) peut être gênante en pratique. Considérons par exemple comme signal d'entrée u une gaussienne définie par $u(t) = e^{-\frac{t^2}{2}}$, échantillonnée en $n = 128$ points de l'intervalle $[-5, 5]$, c'est à dire $a = 10$ avec les notations précédentes. Voici l'allure du signal d'entrée :

Signal d'entrée



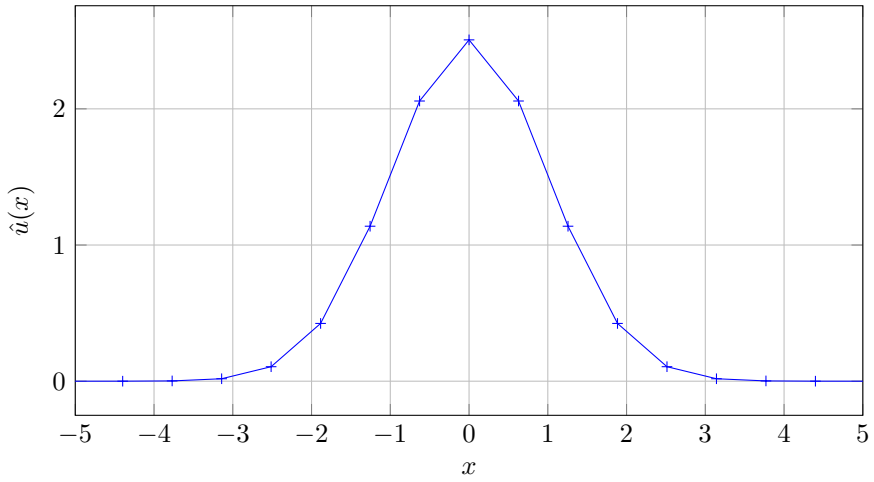
Utilisons maintenant l'équation (10) pour calculer $\hat{u}(x)$. La contrainte (2) impose $b = \frac{2\pi n}{a} \approx 80$ comme largeur de fenêtre. Le résultat est le suivant :

Transformée de Fourier du signal



On remarque qu'il n'y a que très peu de points dans le support significatif de $\hat{u}$, c'est à dire grossièrement dans l'intervalle $[-5, 5]$. Un zoom montre que la courbe ne contient qu'une petite vingtaine de points entre -5 et 5 :

Transformée de Fourier du signal (zoom entre -5 et 5)



On peut se demander comment évolue avec n le nombre de points de u de l'intervalle $[-5, 5]$ calculés par la FFT. Ce nombre de points est proportionnel à $\frac{n}{b}$, le nombre de points divisé par la taille totale de la fenêtre en fréquence. Or par (2), $\frac{n}{b} = \frac{a}{2\pi}$. Ainsi, le nombre de points qui nous intéressent ne dépend pas de n. Jouer sur le nombre de points n'améliorera pas l'allure de la courbe obtenue. La solution standard consiste à jouer sur le paramètre a en augmentant la taille de la fenêtre initiale. Puisque la gaussienne $u(t)$ est de l'ordre de la précision machine dès que $|t| \geq 10$, cela revient à rallonger artificiellement le signal d'entrée avec des zéros avant de l'envoyer à la FFT. Cette méthode implique de nombreux calculs inutiles, et la transformation de Fourier fractionnaire donne une méthode élégante de les éviter.

4.3 Par la transformation de Fourier fractionnaire

Revenons maintenant au calcul de $\hat{f}(x_k)$. L'équation (9) donnait l'expression :

$$\hat{f}(x_k) \approx \frac{a}{n} e^{i\frac{a}{2}x_k} \sum_{j=0}^{n-1} f(t_j) e^{i\frac{ab}{2n}j} e^{-i\frac{ab}{n^2}jk}$$

On reconnait une transformée fractionnaire avec $-i\frac{ab}{n^2}jk = -2i\pi jk\alpha$ soit $\frac{ab}{n^2} = 2\pi\alpha$ soit $\alpha = \frac{ab}{2\pi n^2}$.

$$\hat{f}(x_k) \approx \frac{a}{n} e^{i\frac{a}{2}x_k} \cdot G_k\left(\left(f(t_j)e^{i\frac{ab}{2n}j}\right)_{0\leq j<n}, \frac{ab}{2\pi n^2}\right) \tag{11}$$

Cette expression permet d'évaluer les $(\hat{f}(x_k))_{0\leq k\leq n-1}$ en un temps $O(n\log(n))$, et ceci sans la contrainte (2) :

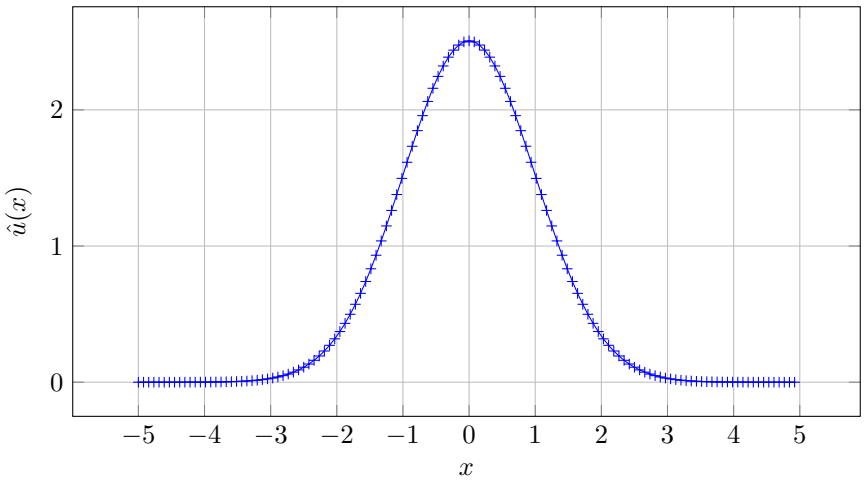
```
def cfft(u, a, b, n):
    k = np.arange(n)
    xk = -b/2 + (b/n)*k

    prefactor = (a/n) * np.exp(1j*(a/2)*xk)
    y = u * np.exp(1j*a*b/(2*n)*k)
    alpha = a*b / (2*np.pi*n*n)

    return prefactor * frac_fft(y, alpha)
```

Revenons une seconde sur la remarque du coût exact de l'évaluation de la transformée de Fourier fractionnaire de la section 3.2. On en déduit que si l'on souhaite évaluer à a, b, et n fixés pour plusieurs valeurs de x la séquence $(\hat{f}(x_k))_{0\leq k\leq n-1}$, le coût revient à 4 fois celui d'une transformée de Fourier rapide pour chaque entrée x.

4.4 Retour sur l'exemple

Utilisons maintenant l'algorithme ci-dessus sur le même exemple que celui de la section 4.2. En choisissant comme paramètre $b = 10$, on obtient un résultat beaucoup plus satisfaisant. Le nombre de points dans la fenêtre d'intérêt est directement proportionnel à n.

Transformée de Fourier du signal



Le code python utilisé pour la création de cette figure ainsi que celles présentes à la section 4.2 est disponnible en annexe A.

5 Simulation numérique d'équations aux dérivées partielles

Cette partie du rapport étudie l'utilisation de la transformation de Fourier fractionnaire pour la simulation d'équations aux dérivées partielles. Le commencement de cette étude est une discussion entre Stéphane Balac [1] notre encadrant pour ce rapport, et des physiciens du Laboratoire d'analyse et d'architecture des systèmes de Toulouse. Ces derniers ont rencontré une situation où les techniques développées dans Bailey et Swarztrauber (1991) et Bailey et Swarztrauber (1994) pouvaient être mises à profit : la simulation de l'équation de Schrödinger non-linéaire décrivant le comportement d'une fibre optique. Nous commençons par étudier comment la transformation de Fourier discrète peut être utilisée pour dériver numériquement des fonctions.

5.1 Inversion de Fourier

La formule d'inversion de Fourier permet de reconstruire f à partir de $\hat{f}$ par la formule :

$$f(t) = \frac{1}{2\pi} \int_{-\infty}^{+\infty} \hat{f}(x) e^{itx} dx$$

En repernant le raisonnement depuis le début du rapport, les équations (1), (9) et (11) deviennent respectivement :

$$\begin{aligned} f(t_j) &\approx \frac{b}{2\pi n} \sum_{k=0}^{n-1} \hat{f}(x_k) e^{i x_k t_j} \\ &\approx \frac{b}{2\pi n} e^{-i\frac{b}{2}t_j} \sum_{k=0}^{n-1} \hat{f}(x_k) e^{-i\frac{ab}{2n}k} e^{i\frac{ab}{n^2}jk} \\ &\approx \frac{b}{2\pi n} e^{-i\frac{b}{2}t_j} \cdot G_j \left(\left(\hat{f}(x_k) e^{-i\frac{ab}{2n}k} \right)_{0 \leq k < n}, -\frac{ab}{2\pi n^2} \right) \end{aligned}$$

1. https://perso.univ-rennes1.fr/stephane.balac/

Ainsi, on peut écrire un algorithme reconstruisant f à partir de $\hat{f}$ en temps $O(n\log(n))$:

```
def icfft(u, a, b, n):
    j = np.arange(n)
    tj = -a/2 + (a/n)*j

    prefactor = (b/2/np.pi/n) * np.exp(-1j*(b/2)*tj)
    y = u * np.exp(-1j*a*b/(2*n)*j)
    alpha = a*b / (2*np.pi*n*n)

    return prefactor * frac_fft(y, -alpha)
```

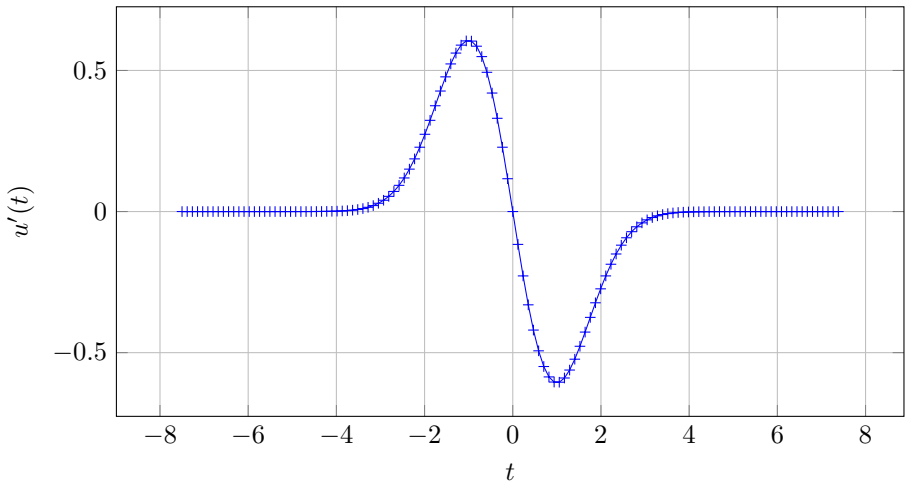
5.2 Dérivation numérique via la FFT

La dérivation se comporte très bien avec la transformée de Fourier. En effet si f est intégrable et continûment dérivable, on a la formule $\hat{f}'(x) = ix\hat{f}(x)$. Couplée à l'inversion de Fourier, cette formule donne un algorithme pour dériver numériquement une fonction.

```
def deriv(u, a, b):
    n = len(u)
    u_hat = cfft(u, a, b, n)
    xk = -b/2 + (b/n)*np.arange(n)
    u_hat *= 1j * xk
    return icfft(u_hat, a, b, n)
```

On peut utiliser cet algorithme pour calculer numériquement la dérivée de notre gaussienne favorite.

Dérivée numérique de $u : t \mapsto e^{-\frac{t^2}{2}}$



Avec $n = 128$ points, l'erreur par rapport à la dérivée explicite est de l'ordre de 10^{-12}. Le code utilisé est fourni ci-dessous :

```
a = 15
n = 128
t = a/n*np.arange(n) - a/2
u = np.exp(-t**2/2)
up = deriv(u, a, a).real
save_to_file(t, up, "gaussienne_derivee.txt")
error(up, -t*np.exp(-t**2/2))
plt.plot(t, up)
plt.show()
```

Notons que la dérivation numérique via la transformée de Fourier n'est pas liée à la transformée de Fourier fractionnaire. Celle-ci peut parfaitement être réalisée sous la contrainte (2) avec l'algorithme de FFT standard. Je conseille l'article se Johnson (2011) qui étudie la dérivation via la FFT sous le regard de l'interpolation trigonométrique. Je trouve que cet article est particulièrement éclairant dans le traitement de la fréquence de Nyquist lors de la dérivation.

5.3 L'équation de Schrödinger non-linéaire

Pour le lecteur hypothétique qui souhaiterait comprendre plus en détail l'origine et les phénomènes liés à cette équation, nous conseillons le livre *Nonlinear Fiber Optics* Agrawal (2013), qui semble être une bonne référence. Après changements de variables et renormalisations permettant de faire abstraction de nombreux paramètres physiques, l'équation peut être mise sous cette forme plus propice à la simulation :

$$\begin{cases} \dfrac{\partial u}{\partial z}(z,t) = -i\dfrac{\operatorname{sign}(\beta_2)}{2}\dfrac{\partial^2 u}{\partial t^2}(z,t) + iN^2 u(z,t)|u(z,t)|^2 \\ u(0,t) = u_0(t) \end{cases} \tag{12}$$

La partie linéaire ressemble à l'équation de Schrödinger fondement de la physique quantique :

$$i\hbar\dot{\Psi} = \frac{-\hbar^2}{2m}\Delta\Psi + V\Psi$$

dans laquelle on aurait échangé les variables spatiales et temporelles. L'équation (12) possède en plus un terme non linéaire qui rend sa simulation plus délicate. La méthode utilisée pour la simulation est une méthode pseudo-spectrale en deux étapes (de l'anglais *split-step*), qui consiste à simuler alternativement la partie linéaire et la partie non-linéaire de l'équation.

5.4 La méthode *split-step*

La simulation se fait suivant le paramètre z croissant. On suppose que le pas de la simulation $h = z_{k+1} - z_k$ est fixé à l'avance. Cette hypothèse peut aisément être enlevée si l'on souhaite implémenter une simulation à pas variable. Pour effectuer une étape de la simulation, c'est à dire calculer $u(z_{k+1}, \cdot)$ en fonction de $u(z_k, \cdot)$ par l'équation (12), on résout les trois sous-problèmes suivants :

$$\begin{cases} \dfrac{\partial u_1}{\partial z}(z,t) = iN^2 u_1(z,t)|u_1(z,t)|^2 \\ u_1(0,t) = u(z_k,t) \end{cases} \tag{13}$$

$$\begin{cases} \dfrac{\partial u_2}{\partial z}(z,t) = -i\dfrac{\operatorname{sign}(\beta_2)}{2}\dfrac{\partial^2 u_2}{\partial t^2}(z,t) \\ u_2(z=0,t) = u_1(\dfrac{h}{2},t) \end{cases} \tag{14}$$

$$\begin{cases} \dfrac{\partial u_3}{\partial z}(z,t) = iN^2 u_3(z,t)|u_3(z,t)|^2 \\ u_3(0,t) = u_2(h,t) \end{cases} \tag{15}$$

Et Finalement

$$u(z_{k+1},t) = u(z_k + h, t) = u_3(\frac{h}{2},t)$$

Ce découpage alternant la simulation des phénomènes linéaires et non-linéaires crée une erreur à chaque étape. Si l'on avait seulement alterné entre deux sous-problèmes, un linéaire et un non-linéaire, l'erreur aurait été de l'ordre de h^2. L'avantage de cette méthode symétrisée, non-linéaire, linéaire, puis à nouveau non-linéaire est que l'erreur est de l'ordre de h^3.

Il faut maintenant être capable de résoudre efficacement les trois sous-problèmes numériquement. En intégrant l'équation (13) on obtient :

$$u_1(z,t) = u(z_k,t) \cdot \exp\left(\int_0^z iN^2|u_1(z,t)|^2 dz\right) \tag{16}$$

On suppose maintenant que le terme sous l'intégrale est proche de $iN^2|u_1(0,t)| = iN^2|u_(z_k,t)|$, pour pouvoir exprimer $u_1(\frac{h}{2},t)$. L'erreur commise ici est de l'ordre de h. Une amélioration possible serait d'utiliser une méthode de Runge-Kutta de même ordre que le schéma split-step, pour rester cohérent.

Le traitement la partie linéaire (14) se fait en prenant la transformée de Fourier de u_2 par rapport à t. L'équation devient :

$$\frac{\partial \hat{u_2}}{\partial z}(z,\nu) = -i\frac{\text{sign}(\beta_2)}{2}(i\nu)^2\hat{u_2}(z,\nu)$$

Le calcul explicite d'une solution est alors possible :

$$\hat{u_2}(h,\nu) = \hat{u_2}(z_k,\nu) \cdot e^{i\frac{\text{sign}(\beta_2)}{2}\nu^2 h} \tag{17}$$

Dans la partie linéaire du schéma, la seule erreur provient de la transformation de Fourier et ne dépend pas de h. L'erreur dépend de la méthode de quadrature de l'intégrale utilisée pour la transformation de Fourier (qui pourrait aisément être améliorée d'ailleurs) et du nombre de points utilisés dans les transformations de Fourier.

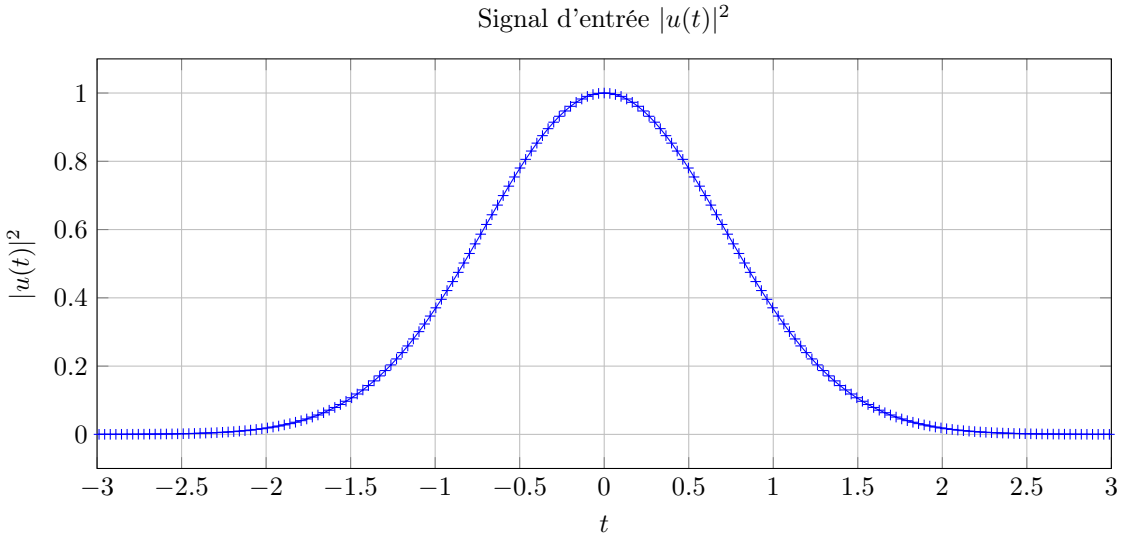
On peut donc écrire un algorithme simulant l'équation de Schrödinger non-linéaire. Le code PYTHON est disponnible en annexe B.

5.5 Résultats

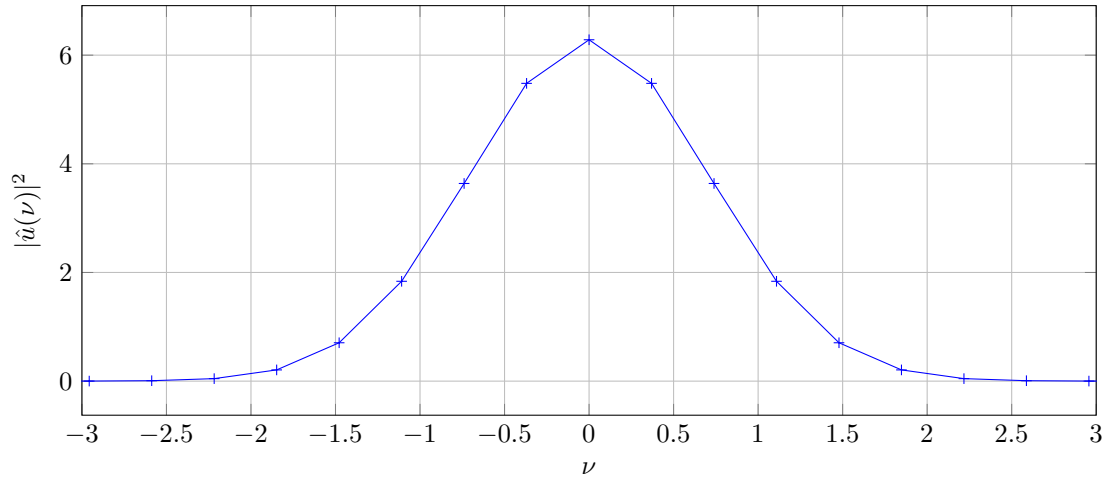
Pour la simulation, les paramètres ont été choisis comme dans le rapport *Numerical computation of the Fourier Integral Transform revisited* de STÉPHANE BALAC pour pouvoir comparer les résultats. Les Les paramètres exacts sont disponnibles dans l'annexe B.1. À l'oeil, les résultats semblent identiques à ceux de monsieur BALAC. N'ayant pas l'accès à une machine MATLAB, je n'ai pas eu l'occasion d'exécuter directement le code qu'il m'a transmis sur différents paramètres d'entrée.

Dans la version utilisant uniquement la FFT, on remarque que le nombre de points du signal spectral laisse à désirer. C'est le problème qui a été rencontré par les physiciens du LAAS de Toulouse. Avec la version utilisant la transformée fractionnaire, on peut augmenter le nombre de points dans les fréquences intéressantes (ici le nombre de points à été augmenté d'un facteur 2).

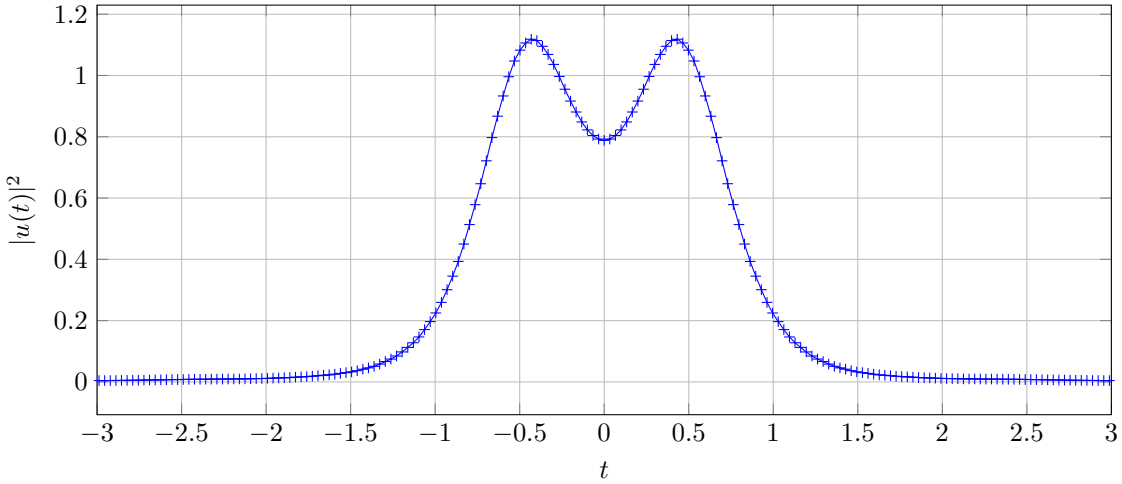
5.5.1 En utilisant seulement la FFT



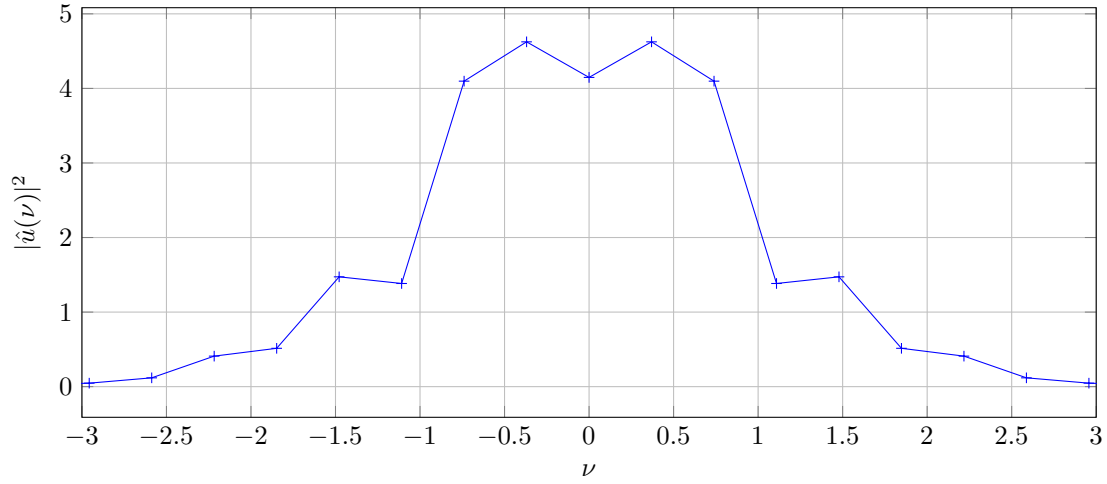
Fréquence du signal d'entrée



Signal de sortie $|u(t)|^2$

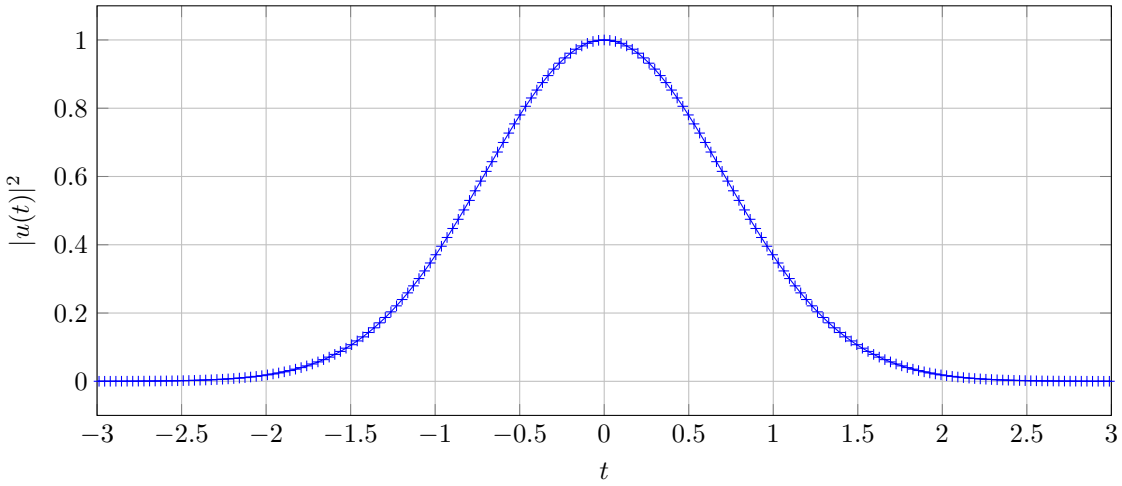


Fréquence du signal de sortie

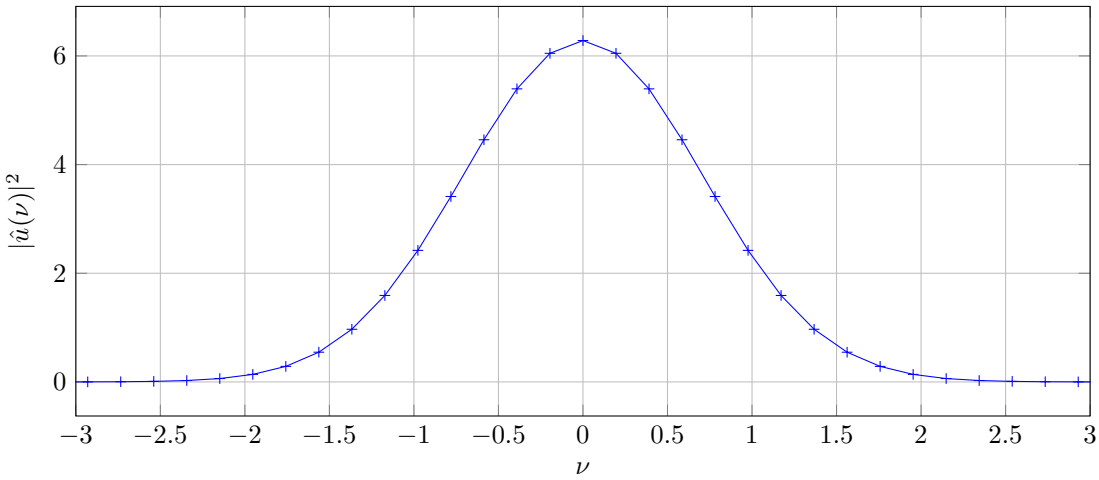


5.5.2 En utilisant la transformée de Fourier fractionnaire discrète

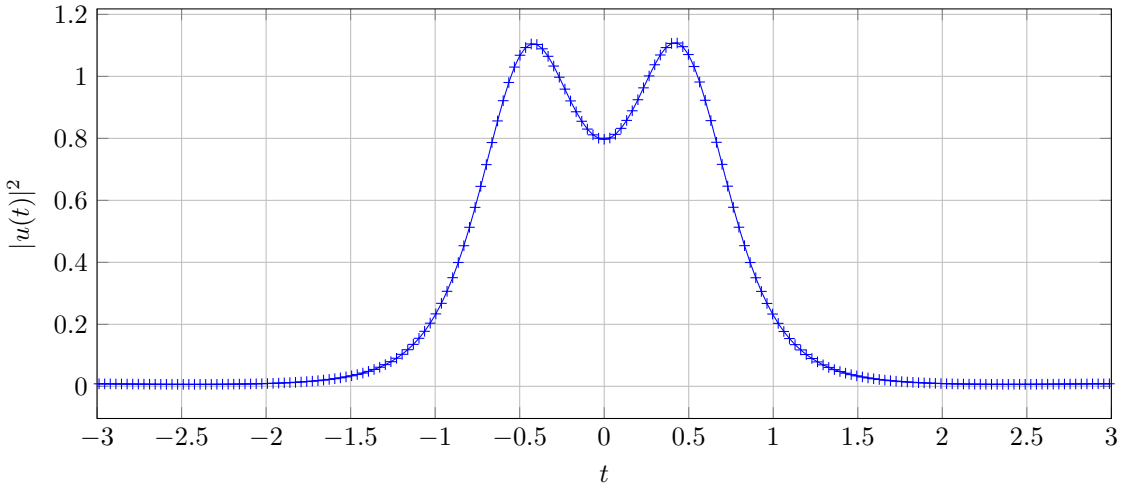
Signal d'entrée $|u(t)|^2$



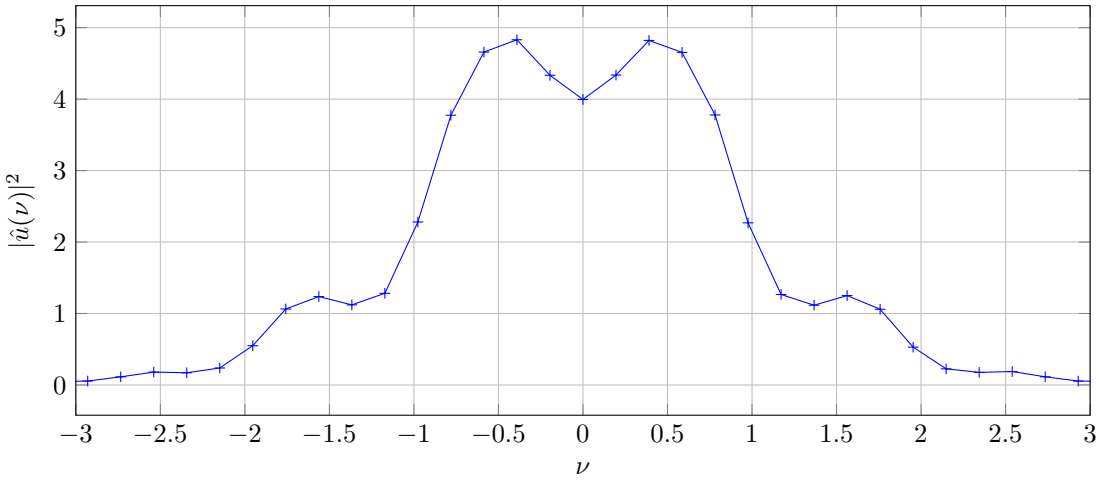
Fréquence du signal d'entrée



Signal de sortie $|u(t)|^2$



Fréquence du signal de sortie



6 Conclusion

Nous avons dans se rapport rappelés des résultats généraux sur la transformation de Fourier discrète, puis nous avons implémenté la transformation de Fourier discrète fractionnaire comme dans l'article Bailey et Swarztrauber (1991). Ensuite nous avons utilisé cette transformation fractionnaire pour calculer numériquement des transformations de Fourier continues de façon plus flexible qu'avec la transformation de Fourier rapide, comme dans l'article Bailey et Swarztrauber (1994). Enfin, nous avons utilisé la transformation de Fourier discrète pour simuler une équation aux dérivées paritelles non linéaire. Finalement, même si les applications de la transformation de Fourier discrète fractionnaire sont relativement peu fréquentes, il existe des situations comme en simulation d'équations aux dérivées partielles où elle s'avère très utile.

Références

Govind P. Agrawal : *Nonlinear fiber optics*. Elsevier Science, 5ème édition, 2013. ISBN 978-0-12-397023-7.

David H. Bailey et Paul N. Swarztrauber : The fractional Fourier transform and applications. *SIAM Rev.*, 33(3):389–404, 1991. ISSN 0036-1445. URL https://doi.org/10.1137/1033097.

David H. Bailey et Paul N. Swarztrauber : A fast method for the numerical evaluation of continuous Fourier and Laplace transforms. *SIAM J. Sci. Comput.*, 15(5):1105–1110, 1994. ISSN 1064-8275. URL https://doi.org/10.1137/0915067.

L. Bluestein : A linear filtering approach to the computation of discrete fourier transform. *IEEE Transactions on Audio and Electroacoustics*, 18(4):451–455, 1970. URL https://doi.org/10.1109/TAU.1970.1162132.

James W. Cooley et John W. Tukey : An algorithm for the machine calculation of complex Fourier series. *Math. Comp.*, 19:297–301, 1965. ISSN 0025-5718. URL https://doi.org/10.2307/2003354.

Matteo Frigo : A fast Fourier transform compiler. *In Proc. 1999 ACM SIGPLAN Conf. on Programming Language Design and Implementation*, volume 34, pages 169–180. ACM, May 1999.

Matteo Frigo et Steven G. Johnson : The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005. Special issue on "Program Generation, Optimization, and Platform Adaptation".

Steven G Johnson : Notes on FFT-based differentiation. URL https://math.mit.edu/~stevenj/fft-deriv.pdf. non publié, 2011.

A Code des exemples

```python
def save_to_file(t, u, name):
    n = len(t)
    with open(name, 'w') as f:
        f.write("X Y\n")
        f.writelines([str(t[i]) + " " + str(u[i]) + "\n" for i in range(n)])

a = 10
n = 128

# u est une gaussienne
t = a/n*np.arange(n) - a/2
u = np.exp(-t**2/2)
save_to_file(t, u.real, "gaussienne_entree.txt")
plt.plot(t, u, '-+')
plt.show()

# Calcul de la transformée de u
b = 2*np.pi*n/a
print(b)
x = b/n*np.arange(n) - b/2
SHIFT = (-1)**np.arange(n)
u_hat = a/n * (-1j)**n * SHIFT * fft(SHIFT * u)
save_to_file(x, u_hat.real, "gaussienne_sortie.txt")
plt.plot(x, u_hat.real, '-+')
plt.show()

# Centré
plt.plot(x, u_hat.real, '-+')
plt.xlim(-5, 5)
plt.show()

# CFFT
u_hat_bis = cfft(u, a, 10, n)
save_to_file(t, u_hat_bis.real, "gaussienne_sortie_bis.txt")
plt.plot(t, u_hat_bis.real, '-+')
plt.show()
```

B Code de la simulation

B.1 Paramètres de la simulation

```
N = 3 # Paramètre de la non-linéarité
sign_beta = -1 # Normal (1) or anomalous (-1) dispersion

T = 17 # Amplitude temporelle
Nt = 512 # Nombre de points de la FFT

Nz = 1000 # Nombre total d'itérations
L = 5 # Longueur de la fibre

t = np.arange(Nt)*T/Nt - T/2 # Grille temporelle
h = L / Nz # Pas spatial

# Signal en entrée de la fibre
u0 = np.exp(-t*t/2)

display_freq = 20 # [-s/2, s/2], Utilisé pour l'affichage
display_time = 6
```

B.2 Avec la FFT

```
# Détermine les fréquences induites par la FFT
FreqAmp = 2*np.pi*Nt/T
print("FreqAmp : ", FreqAmp)
nu = np.arange(Nt)*FreqAmp/Nt - FreqAmp/2

# Opérateurs à l'exécution
SHIFT = (-1)**np.arange(Nt)
CONST = T/Nt * (-1j)**Nt
DIFFUSE = np.exp(1j * sign_beta / 2 * h * nu**2)

# Affichage initial
u0_hat = CONST * SHIFT * fft(SHIFT*u0)
plot_time_and_freq(u0, t, u0_hat, nu, freqsize = display_freq, timesize = display_time)
save_to_file(t, np.abs(u0)**2, "FFT_u_entree.txt")
save_to_file(nu, np.abs(u0_hat)**2, "FFT_uhat_entree.txt")

u = u0
for i in range(Nz):
    u = u * np.exp(1j * N*N * h/2  * np.abs(u)**2) # Demi itération non linéaire
    u_hat = CONST * SHIFT * fft(SHIFT*u) # Passage en Fourier
    u_hat = DIFFUSE * u_hat # Diffusion
    u = ifft(u_hat / CONST * SHIFT) * SHIFT # Retour en temporel
    u = u * np.exp(1j * N*N * h/2  * np.abs(u)**2) # Demi itération non linéaire

# Affichage final
u_hat = CONST * SHIFT * fft(SHIFT*u)
plot_time_and_freq(u, t, u_hat, nu, freqsize = display_freq, timesize = display_time)
save_to_file(t, np.abs(u)**2, "FFT_u_sortie.txt")
save_to_file(nu, np.abs(u_hat)**2, "FFT_uhat_sortie.txt")
```

B.3 Avec la transformée fractionnaire

```
# Détermine les fréquences induites par la FFT
FreqAmp = 100
print("FreqAmp : ", FreqAmp)
nu = np.arange(Nt)*FreqAmp/Nt - FreqAmp/2

# Opérateurs à l'exécution
DIFFUSE = np.exp(1j * sign_beta / 2 * h * nu**2)

# Affichage initial
u0_hat = cfft(u0, T, FreqAmp, Nt)
plot_time_and_freq(u0, t, u0_hat, nu, freqsize = display_freq, timesize = display_time)
save_to_file(t, np.abs(u0)**2, "FrFFT_u_entree.txt")
save_to_file(nu, np.abs(u0_hat)**2, "FrFFT_uhat_entree.txt")

u = u0
for i in range(Nz):
    u = u * np.exp(1j * N*N * h/2  * np.abs(u)**2) # Demi itération non linéaire
    u_hat = cfft(u, T, FreqAmp, Nt)
    u_hat = DIFFUSE * u_hat # Diffusion
    u = icfft(u_hat, T, FreqAmp, Nt)
    u = u * np.exp(1j * N*N * h/2  * np.abs(u)**2) # Demi itération non linéaire

# Affichage final
u_hat = cfft(u, T, FreqAmp, Nt)
plot_time_and_freq(u, t, u_hat, nu, freqsize = display_freq, timesize = display_time)
save_to_file(t, np.abs(u)**2, "FrFFT_u_sortie.txt")
save_to_file(nu, np.abs(u_hat)**2, "FrFFT_uhat_sortie.txt")
```