Compte rendu – Projet L3 Outils maths

Transformée de Fourier

Tom BARBIER – Brandon PERE
Année 2022 - 2023

Table des matières

I. INTRODUCTION

Nous allons voir comment et ce à quoi sert la transformé de Fourier, ainsi que la transformé de Fourier rapide. Nous allons expliquer pourquoi nous évoquons les deux. Il faut savoir que la transformé de Fourier est très utile, cependant, elle est difficilement utilisable lorsque nous l'implémentons, pourquoi ? En raison de sa complexité. En effet, la complexité de la transformé de Fourier est de $O(N^2)$, c'est pourquoi il existe, la transformé de Fourier rapide, qui a une complexité simplifiée. En effet, sa complexité est de $O(N * \log_2 N)$ pour des tableaux de tailles de taille 2^N. Dans un premier temps nous construirons la fonction pour des tableaux à 1 dimension. Puis nous généraliserons à deux dimensions en ce reposant sur la 1D.

II. TRANSFORMEE DE FOURIER DIRECTE

$$\hat{g}(u) = F\big(g(x)\big) = \sum_{x=0}^{N-1} g(x) e^{\left(\frac{-2*i*\pi*u*x}{N}\right)}$$

u index du tableau

N taille du tableau

g tableau source

ĝ tableau de destination

On remarque donc que pour chaque élément nous devons parcourir l'entièreté du tableau source ce qui explique sa complexité de $O(N^2)$. En l'implémentant en python cela nous donne :

```python
def TF1D(Matrice1D):
    N=len(Matrice1D)
    # Creation d'une matrice de la taille de l'image1D
    MatriceRes = np.zeros(N, dtype=complex)
    # On parcours notre matrice initial
    for u in range(N):
        sum = 0
        # On applique la formule
        for x in range(N):
            sum += Matrice1D[x]*cmath.exp((-2j * cmath.pi * u * x) / N)
        # On met la valeur dans la matrice resultat + on arrondi les valeurs
        MatriceRes[u]=round(sum.real, 8)+round(sum.imag, 8)*1j
    return MatriceRes
```

On peut comparer les résultats avec la fft de numpy. (bibliothèque que nous utilisons pour simplifier le code)

```
Matrice en entrée
[1 2 3 4]
-------------------------------------------------
Résultat de notre algorithme pour la TF1D
[10.+0.j -2.+2.j -2.+0.j -2.-2.j]
-------------------------------------------------



-------------------------------------------------
Résultat de l'algorithme de numpy pour la TF1
[10.+0.j -2.+2.j -2.+0.j -2.-2.j]
```

III. OPTIMISATION

1. TRANSFORMEE 1D

Nous pouvons séparer la somme en deux sommes afin d'avoir les éléments pairs d'un coté et impairs de l'autre côté ce qui donne :

$$\hat{g}(u) = F\big(g(x)\big) = \sum_{x=0}^{N-1} g(x) e^{\left(\frac{-2*i*\pi*u*x}{N}\right)}$$

$$\hat{g}(u) = \sum_{x=0}^{\frac{N}{2}-1} g(2x) * e^{-2i*\pi*2x*\left(\frac{u}{N}\right)} + \sum_{x=0}^{\frac{N}{2}-1} g(2x+1) * e^{-2i*\pi*(2x+1)*\left(\frac{u}{N}\right)}$$

$$\hat{g}(u) = \sum_{x=0}^{\frac{N}{2}-1} g(2x) * e^{-2i*\pi x*\left(\frac{2u}{N}\right)} + \sum_{x=0}^{\frac{N}{2}-1} g(2x+1) * e^{-2i*\pi x*\left(\frac{2u}{N}\right)} * e^{-2i*\pi*\left(\frac{u}{N}\right)}$$

$$car\ e^{(a+b)} = e^{a} * e^{b}$$

$$\hat{g}(u) = \sum_{x=0}^{\frac{N}{2}-1} g(2x) * e^{-2i*\pi x*\left(\frac{u}{N/2}\right)} + \sum_{x=0}^{\frac{N}{2}-1} g(2x+1) * e^{-2i*\pi x*\left(\frac{u}{N/2}\right)} * e^{-2i*\pi*\left(\frac{u}{N}\right)}$$

Le terme $e^{-2i*\pi*(\frac{u}{N})}$ ne dépend pas de la variable x de la somme il est donc possible de la sortir de la somme :

$$\hat{g}(u) = \sum_{x=0}^{\frac{N}{2}-1} g(2x) * e^{-2i*\pi x*\left(\frac{u}{\frac{N}{2}}\right)} + e^{-2i*\pi*\left(\frac{u}{N}\right)} * \sum_{x=0}^{\frac{N}{2}-1} g(2x+1) * e^{-2i*\pi x*\left(\frac{u}{N/2}\right)}$$

Nous avons donc en bleu à gauche les éléments pairs et en vert à droite les éléments impairs. On a donc notre tableau, mais celui-ci n'est pas complet car le tableau est de taille $N/2$.

Nous devons donc calculer la seconde moitié soit :

$$\hat{g}\left(u+\frac{N}{2}\right) = \sum_{x=0}^{\frac{N}{2}-1} g(2x) * e^{-2i*\pi*x*(\frac{u+\frac{N}{2}}{\frac{N}{2}})} + e^{-2i*\pi*(\frac{u+\frac{N}{2}}{N})} * \sum_{x=0}^{\frac{N}{2}-1} g(2x+1) * e^{-2*i*\pi*x*(\frac{u+\frac{N}{2}}{\frac{N}{2}})}$$

Simplifions cette expression :

$$\hat{g}\left(u+\frac{N}{2}\right) = \sum_{x=0}^{\frac{N}{2}-1} g(2x) * e^{-2i*\pi*x*(\frac{2u}{N}+1)} + e^{-2i*\pi*(\frac{1}{2}+\frac{u}{N})} * \sum_{x=0}^{\frac{N}{2}-1} g(2x+1) * e^{-2*i*\pi*x*(\frac{2u}{N}+1)}$$

$$\hat{g}\left(u+\frac{N}{2}\right) = \sum_{x=0}^{\frac{N}{2}-1} g(2x) * e^{-2i*\pi*x*(\frac{2u}{N})} * e^{-2i*\pi*x} + e^{-2i*\pi*(\frac{1}{2}+\frac{u}{N})} * \sum_{x=0}^{\frac{N}{2}-1} g(2x+1) * e^{-2*i*\pi*x*(\frac{2u}{N})} * e^{-2i*\pi*x}$$

$$e^{-2i*\pi*x} = 1 \text{ avec } x \in N$$

$$\hat{g}\left(u+\frac{N}{2}\right) = \sum_{x=0}^{\frac{N}{2}-1} g(2x) * e^{-2i*\pi*x*(\frac{2u}{N})} + e^{-2i*\pi*(\frac{u}{N})} * e^{-i\pi} * \sum_{x=0}^{\frac{N}{2}-1} g(2x+1) * e^{-2*i*\pi*x*(\frac{2u}{N})}$$

$$e^{-i\pi} = -1$$

$$\hat{g}\left(u+\frac{N}{2}\right) = \sum_{x=0}^{\frac{N}{2}-1} g(2x) * e^{-2i*\pi*x*(\frac{2u}{N})} - e^{-2i*\pi*(\frac{u}{N})} * \sum_{x=0}^{\frac{N}{2}-1} g(2x+1) * e^{-2*i*\pi*x*(\frac{2u}{N})}$$

Maintenant que nous avons simplifier le problème nous pouvons passer à l'implémentation. L'avantage de la représentation mathématique que nous avons est qu'elle laisse bien voir un algorithme récursif.

```python
def TF1R(Matrice1D):
    N=len(Matrice1D)

    # Si le tableau n'a qu'une seul valeur on retourne la valeur
    if N<=1:
        return Matrice1D
    else :
        # Récursivité sur les index pair et impair
        pair = TF1R(Matrice1D[0::2])
        impair = TF1R(Matrice1D[1::2])
        # On crée un nouveau tableau
        MatriceRes = np.zeros(N).astype(np.complex64)
        # On reconstitue petit à petit le tableau 1D avec la formule
        for i in range(0, N//2):
            MatriceRes[i] = pair[i]+cmath.exp(-2j*cmath.pi*i/N)*impair[i]
            MatriceRes[i+N//2] = pair[i]-cmath.exp(-2j*cmath.pi*i/N)*impair[i]
        return MatriceRes
```

On peut comparer les résultats avec la fft de numpy.

```
Matrice en entrée
[1 2 3 4]
--------------------------------------------------



--------------------------------------------------
Résultat de notre algorithme pour la TF1R
[10.+0.j -2.+2.j -2.+0.j -2.-2.j]
--------------------------------------------------
Résultat de l'algorithme de numpy pour la TF1
[10.+0.j -2.+2.j -2.+0.j -2.-2.j]
```

2. INVERSE

Pour ce qui est de l'inverse nous allons juste faire des changements de signe comme décrit par la formule et supprimer le coefficient pour chaque valeur retournée.

```python
def TFI1R_m(Matrice1D):
    N=len(Matrice1D)

    # Si le tableau n'a qu'une seul valeur on retourne la valeur
    if N<=1:
        return Matrice1D
    else :
        # Récursivité sur les index pair et impair
        pair = TFI1R_m(Matrice1D[0::2])
        impair = TFI1R_m(Matrice1D[1::2])
        # On crée un nouveau tableau
        MatriceRes = np.zeros(N).astype(np.complex64)
        # On reconstitue petit à petit le tableau 1D avec la formule et les changements de signes
        for i in range(0, N//2):
            MatriceRes[i] = pair[i]+cmath.exp(2j*cmath.pi*i/N)*impair[i]
            MatriceRes[i+N//2] = pair[i]-cmath.exp(2j*cmath.pi*i/N)*impair[i]
        return MatriceRes

# On supprime le coefficiant de chaque valeur
def TFI1R(tab):
    return [x/len(tab) for x in TFI1R_m(tab)]
```

3. TRANSFORMEE 2D

Nous avons donc notre transformée de Fourier rapide 1D. Pour passer en 2D il suffit d'appliquer la même formule sur toutes les lignes et sur toutes les colonnes. Pour cela on fait appel à la transposé car cela revient à appliquer la transformée de Fourier rapide sur chaque ligne puis à refaire la même opération sur la transposé du résultat.

```python
def TF2R(Matrice2D):
    # Creation d'une matrice de taille ligne*colonne de l'image2D
    MatriceRes = np.zeros((Matrice2D.shape[0], Matrice2D.shape[1]), dtype=complex)

    # On applique la TF1R sur toute les lignes de notre image2D
    for i in range(Matrice2D.shape[0]):
        MatriceRes[i] = TF1R(Matrice2D[i])
    # On inverse les lignes et les colonnes
    MatriceRes_trans = MatriceRes.transpose()
    # On applique à nouveau la TF1R sur cette matrice (donc sur les colonnes de notre image2D)
    for i in range(MatriceRes_trans.shape[0]):
        MatriceRes_trans[i] = TF1R(MatriceRes_trans[i])
    # On remet les lignes à la place des colonnes
    MatriceRes = MatriceRes_trans.transpose()
    #On créé une matrice de uint8 afin de pas avoir de problème de compatibilité lors de la création de l'image finale
    Matrice = np.zeros((MatriceRes.shape[0], MatriceRes.shape[1]), dtype=np.uint8)
    for i in range (MatriceRes.shape[0]):
        for j in range (MatriceRes.shape[1]):
            Matrice[i][j]=np.abs(MatriceRes[i][j])
    np.savetxt("TF2R/matrice_TF2R.txt", MatriceRes, fmt="%.2e")
    img = Image.fromarray(Matrice)
    img.save("TF2R/ImageTF2R.jpg")
    return MatriceRes
```

Pour calculer son inverse on fait exactement la même implémentation sauf que l'on utilise la transformée de Fourier rapide Inverse 1D sur chaque ligne et sur chaque colonne.

IV. COMPLEXITE

1. TRANSFORMEE DE FOURIER DIRECTE

On fait deux boucles, une qui parcourt le tableau et se place sur la valeur à calculer, et l'autre qui une fois que l'on est sur cette valeurs doit de nouveau parcourir toute la liste pour faire la somme.

Soit sur un tableau de taille N on a : f(N) = N * N

Donc une complexité de $O(N^2)$

2. TRANSFORMEE DE FOURIER RAPIDE

On a vu que l'on avait 2 formules similaires pour une taille divisé par 2.

Soit un tableau de taille N on a : $f(N) = 2 * f(\frac{N}{2})$

Or nous devons reconstituer le tableau donc on a une boucle qui parcourt ce tableau.

$f(N) = 2 * f(\frac{N}{2}) + N$

Or cette formule est utilisée récursivement c'est-à-dire :

$$\left.\begin{aligned} f(N) &= 2 * f\left(\frac{N}{2}\right) + N \\ 2 * f\left(\frac{N}{2}\right) &= 4 * f\left(\frac{N}{4}\right) + N \\ 4 * f\left(\frac{N}{4}\right) &= 8 * f\left(\frac{N}{8}\right) + N \\ 8 * f\left(\frac{N}{8}\right) &= 16 * f\left(\frac{N}{16}\right) + N \\ &\dots \\ \frac{N}{2} * f\left(\frac{N}{N/2}\right) &= N * f\left(\frac{N}{N}\right) + N \end{aligned}\right\} \text{n lignes}$$

avec $N = 2^n$

On remarque donc qu'a chaque fois les éléments s'annule dans les implications qui en découle :

$$f(N) = 2 * f\left(\frac{N}{2}\right) + N \rightarrow 2 * f\left(\frac{N}{2}\right) = 4 * f\left(\frac{N}{4}\right) + N \rightarrow 4 * f\left(\frac{N}{4}\right) = 8 * f\left(\frac{N}{8}\right) + N \rightarrow \cdots$$

La récursivité se fera n fois jusqu'à ce que le dénominateur soit égale à N.

Donc on a :

$f(N) = n * N + N = N(1 + n) \approx N * \log_2 N \ll N^2$

V. PERFORMANCE

Maintenant que nous avons une implémentation fonctionnelle de la transformée de Fourier rapide nous pouvons comparer avec la transformée de Fourier directe le temps que l'algorithme va mettre pour s'exécuter sur une image. Pour cela nous utilisons la bibliothèque « time ».

```
Temps d'éxecution Direct :  8.101766109466553
Temps d'éxecution Rapide :  0.8754146099090576
```

Nous avons testé les deux algorithmes sur une image de taille 128 * 128. On voit donc bien que la transformée de Fourier rapide est nettement plus rapide que la directe. Cela montre donc que la complexité de la FFT est bien inférieure à la directe.