Transformée de Fourier rapide

1. Introduction

La transformée de Fourier discrète (TFD) d'un signal discret comportant N échantillons est :

$$S_n = \frac{1}{N}\sum_{k=0}^{N-1} u_k \exp\left(-j2\pi n\frac{k}{N}\right) \tag{1}$$

Pour l'application de cette transformation au cas des fonctions périodiques, voir Transformée de Fourier discrète: série de Fourier. L'application à un signal quelconque est expliquée dans Transformée de Fourier discrète : transformée de Fourier.

Ce document présente l'algorithme de transformée de Fourier discrète rapide (FFT) puis un prototype d'implémentation en python. Enfin une implémentation pour la FFT bidimensionnelle est proposée pour processeur graphique, avec l'interface de programmation OpenGL.

2. Algorithme FFT

Il s'agit de calculer les N sommes suivantes (n variant de 0 à $N-1$) :

$$F_n = \sum_{k=0}^{N-1} u_k \exp\left(-j2\pi n\frac{k}{N}\right) = \sum_{k=0}^{N-1} u_k W_N^{nk} \tag{2}$$

Les N échantillons u_k sont complexes et bien sûr la somme est aussi complexe. Le calcul direct de ces sommes a une complexité en N^2, très pénalisante lorsque N est grand. L'algorithme de transformée de Fourier discrète rapide ([1]) repose sur la décomposition de la somme précédente obtenue en regroupant les termes u_k pairs et les termes impairs. Il nécessite un nombre d'échantillons puissance de deux $N=2^q$. Voici la décomposition :

$$F_n = \sum_{k=0}^{\frac{N}{2}-1} u_{2k} \exp\left(-j2\pi n\frac{2k}{N}\right) + \exp\left(-j2\pi\frac{n}{N}\right)\sum_{k=0}^{\frac{N}{2}-1} u_{2k+1} \exp\left(-j2\pi n\frac{2k}{N}\right) \tag{3}$$

$$= F_n^p + W_N^n F_n^i \tag{4}$$

F_n^p est la TFD des $N/2$ termes pairs. Sa période est $N/2$. De même, F_n^i est la TFD des termes impairs. Comme $\frac{N}{2}=2^{q-1}$ est divisible par deux, chacune de ces TFD est à son tour décomposée en TFD des termes pairs et impairs :

$$F_n^p = F_n^{pp} + W_{N/2}^n F_n^{pi} \tag{5}$$

$$F_n^i = F_n^{ip} + W_{N/2}^n F_n^{ii} \tag{6}$$

Cette décomposition est conduite récursivement jusqu'à obtenir des TFD à un terme. La TFD de u_k est simplement u_k. On aura, par exemple pour $q=4$ et pour la succession *ipii* :

$$F_n^{ipii} = u_m \quad (7)$$

où m est un indice compris entre 0 et $N-1$. Pour déterminer la valeur de cet indice, on remarque que la séparation en termes pairs et impairs dans l'équation (4) revient à tester le bit de poids faible (bit 0) de k. Si ce bit vaut 0, on est dans le groupe pair. Dans la deuxième décomposition, le tri se fait en fonction du bit 1 de k, et ainsi de suite. Par exemple, pour la séquence $ipii$ ces bits sont 1011. En inversant l'ordre des bits, on obtient donc la représentation binaire de m, soit ici 1101.

Pour calculer la TFD de manière récursive (de haut en bas, en partant de la TFD à N termes), il faudrait à chaque étape réorganiser les termes pairs et impairs en tableaux distincts, avant de leur appliquer la TFD de rang inférieur. Il est bien plus efficace de commencer par les TFD à un terme. Il faut pour cela commencer par ranger les échantillons u_k par ordre de leur indice obtenu en inversant l'ordre des bits. Par exemple, le terme 5 de représentation binaire 0101 sera rangé au rang de représentation binaire 1010, c'est-à-dire 10.

Notons x_h la liste des échantillons ainsi rangés. L'indice h a pour représentation binaire la succession $ipii$ (non inversée).

La première étape consiste à obtenir les TFD à deux termes. Pour reprendre l'exemple précédent, on doit calculer pour l'indice m les termes F_n^{ipi} avec $n=0$ et $n=1$ (la TFD à deux termes est de période 2). L'indice dans la liste initiale a pour représentation binaire $ipii$ (après le tri avec inversion de l'ordre des bits). L'indice h est donc impair. La TFD a deux termes s'obtient donc par :

$$F_n^{ipi} = F_n^{ipip} + W_2^n F_n^{ipii} = x_{h-1} + W_2^n x_h \quad (8)$$

Il y a deux valeurs de n à appliquer (0 et 1). De manière plus générale, pour chaque indice h pair, soit $[h/2]$ la représentation binaire de $h/2$ (obtenue en enlevant le dernier bit). On calcule :

$$F_n^{[h/2]} = x_h + W_2^n x_{h+1} \quad (n = 0,1) \quad (9)$$

Pour cette première étape, les valeurs de la liste sont donc modifiées de la manière suivante :

$$x_h \leftarrow x_h + W_2^0 x_{h+1} \quad (10)$$

$$x_{h+1} \leftarrow x_h + W_2^1 x_{h+1} \quad (11)$$

La deuxième étape consiste à calculer les TFD à 4 termes. Pour cela, on applique la transformation suivante à chaque paquet de 4 termes consécutifs :

$$x_h \leftarrow x_h + W_4^0 x_{h+2} \quad (12)$$

$$x_{h+1} \leftarrow x_{h+1} + W_4^1 x_{h+3} \quad (13)$$

$$x_{h+2} \leftarrow x_h + W_4^2 x_{h+2} \quad (14)$$

$$x_{h+3} \leftarrow x_{h+1} + W_4^3 x_{h+3} \quad (15)$$

On remarque que cette transformation nécessite un tableau de stockage intermédiaire.

On procède ensuite au calcul des TFD à $2^3 = 8$ termes, et ainsi de suite jusqu'à la TFD à N termes.

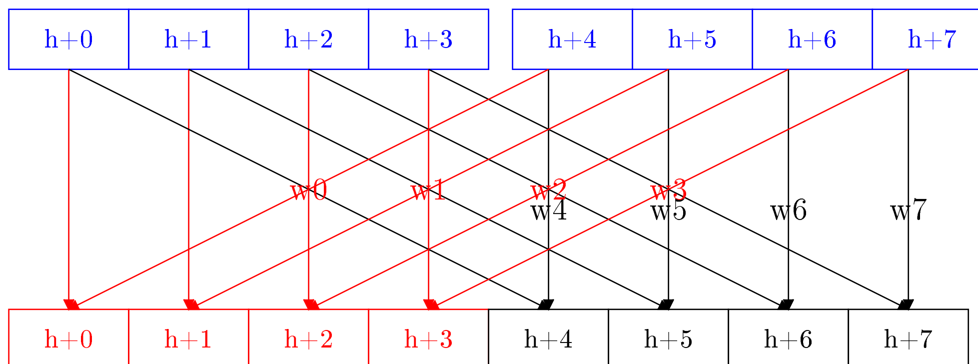
Voyons pour finir comment se fait l'étape e, qui consiste à passer des TFD à 2^{e-1} termes aux TFD à 2e termes. La première étape est $e = 1$, la dernière $e = q$ ($N = 2^q$). Posons $n(e) = 2^e$ et $n(e-1) = 2^{e-1}$. Chaque paquet de $n(e)$ termes consécutifs de la liste x_k doit subir le même calcul. Pour la première moitié de chaque paquet, la transformation est :

$$x_{h+k} \leftarrow x_{h+k} + W^k_{n(e)} x_{h+k+n(e-1)} \quad (k = 0 \cdots n(e-1) - 1) \tag{16}$$

Pour la seconde moitié, la transformation est :

$$x_{h+k} \leftarrow x_{h+k-n(e-1)} + W^k_{n(e)} x_{h+k} \quad (k = n(e-1) \cdots n(e) - 1) \tag{17}$$

La figure suivante montre comment les indices sont combinés dans le cas $n(e) = 8$. La ligne du haut représente deux TFD à 4 termes, la ligne du bas une TFD à 8 termes. L'indice h est un multiple de $n(e)$.



3. Implémentation python

Les différentes étapes de la FFT sont plus simples à implémenter si on dispose de deux tableaux, un pour l'entrée, un pour la sortie. Pour l'implémentation sur GPU (voir plus loin), cette condition est impérative. Par convention, on notera x le tableau d'entrée et y le tableau de sortie.

```
import math
import numpy
```

La première étape consiste à ranger les éléments en inversant l'ordre des bits de l'indice. La fonction suivante réalise l'inversion de bits pour un indice. Le nombre de bits q doit être aussi fourni.

```
def inversionBits(q,k):
    m = 0
    i = k
    for b in range(q):
        m = m|(i&1)
        m = m<<1
        i = i>>1
    m = m>>1
    return m
```

Exemple :

```
print(inversionBits(4,5))
--> 10
```

```
def rangerInversionBits(q,x,y):
    N = 2**q
    for k in range(N):
        m = inversionBits(q,k)
        y[m] = x[k]
```

```
x = numpy.array([0,1,2,3])
y = numpy.array([0,0,0,0])
rangerInversionBits(2,x,y)
```

Pour appliquer l'étape suivante, il faut échanger les références des tableaux :

```
z = x
x = y
y = z
```

```
print(x[0])
--> 0

print(x[1])
--> 2

print(x[2])
--> 1

print(x[3])
--> 3
```

Voyons l'écriture de la fonction qui réalise l'étape e de la FFT. Il faut calculer les $n(e)$ coefficients suivants :

$$W_{n(e)}^k = \exp\left(-j2\pi\frac{k}{n(e)}\right) = \cos\left(2\pi\frac{k}{n(e)}\right) + j\sin\left(2\pi\frac{k}{n(e)}\right) \quad (18)$$

Il s'agit de calculer un cosinus et un sinus pour k variant de 0 à $n(e)-1$. On remarque toutefois que :

$$W_{n(e)}^0 = 1 \quad (19)$$

$$W_{n(e)}^{n(e)/2} = -1 \quad (20)$$

$$W_{n(e)}^{n(e)-k} = \cos\left(2\pi\frac{k}{n(e)}\right) - j\sin\left(2\pi\frac{k}{n(e)}\right) \quad (21)$$

Il y a donc en fait $n(e)/2-1$ cosinus et sinus à calculer.

La fonction suivante réalise l'étape e. La boucle principale se fait sur l'indice k. Les cas $k=0$ et $k=n(e-1)$ sont traités à part.

```
def etapeFFT(x,y,q,e):
    ne = 2**e
    nem1 = ne/2
    for k in range(1,nem1):
        phi = 2*math.pi*k/ne
        W = math.cos(phi)+math.sin(phi)*1j
        for i in range(2**(q-e)): # boucle sur les paquets de ne termes
            h = i*ne # premier indice du paquet
            y[h+k] = x[h+k]+W*x[h+k+nem1]
            y[h+ne-k] = x[h+ne-k-nem1]+W.conjugate()*x[h+ne-k]
    for i in range(2**(q-e)):
        h = i*ne
        y[h] = x[h]+x[h+nem1]
        y[h+nem1] = x[h]-x[h+nem1]
```

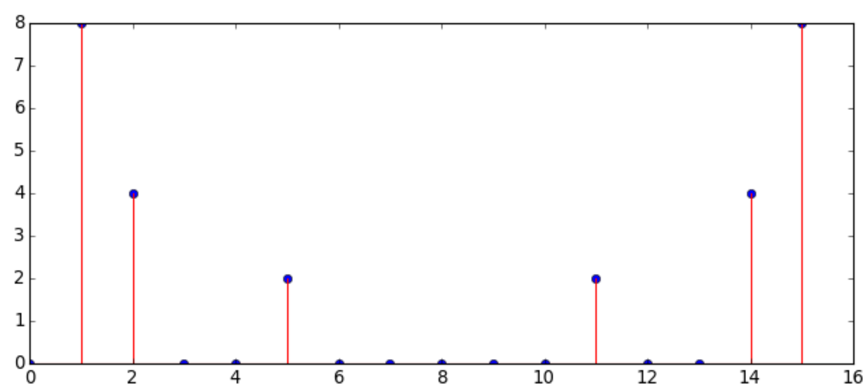
La fonction suivante calcule la transformée de Fourier discrète.

```
def fft1(u,q):
    x = numpy.array(u,dtype=complex)
    y = numpy.zeros(2**q,dtype=complex)
    rangerInversionBits(q,x,y)
    z = x
    x = y
    y = z
    for e in range(1,q+1):
        etapeFFT(x,y,q,e)
        z = x
        x = y
        y = z
    return x
```

Exemple : fonction trigonométrique échantillonnée sur une période.

```
q = 4
N = 2**q
u = numpy.zeros(N,dtype=complex)
i = numpy.zeros(N,dtype=int)
for k in range(N):
    i[k] = k
    u[k] = math.sin(2*math.pi*k/N)+0.5*math.sin(4*math.pi*k/N)+0.25*math.cos(10*math.
tfd = fft1(u,q)
```

```
from pylab import *
spectre = numpy.absolute(tfd)
figure(figsize=(10,4))
stem(i,spectre,'r')
```



4. FFT bidimensionnelle pour processeur graphique

4.a. Prototype

La FFT bibimensionnelle permet de calculer la transformée de Fourier d'une image. La TFD d'une image est obtenue en effectuant d'abord la TFD sur ses lignes, puis en appliquant la TFD sur les colonnes. Une implémentation de la FFT bidimensionnelle sur processeur graphique (GPU) permet d'effectuer la transformée de Fourier d'images très rapidement.

Le processeur graphique effectue des calculs parallèles sur des unités de calcul. Dans la mesure du possible, les calculs sur différentes unités doivent être indépendants les uns des autres. On choisit une répartition des tâches consistant à attribuer un indice du tableau à traiter (à chaque étape) par unité de calcul. Ce n'est pas la solution optimale, mais elle a l'avantage de pouvoir s'implémenter facilement avec l'interface de programmation OpenGL. Pour une répartition plus efficace des tâches, il faut utiliser OpenCL ou CUDA.

Voyons tout d'abord un prototype en python de la fonction qui sera executée par chaque unité de calcul. Chaque unité de calcul doit calculer le terme trigonométrique W correspondant à son indice.

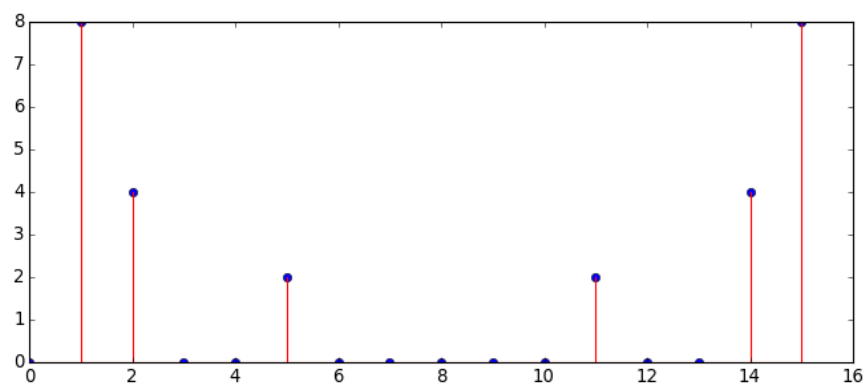
```
def etapeFFT_unite(x,y,q,e,indice):
    ne = 2**e
    nem1 = ne/2
    k = indice%ne # indice local dans le paquet de ne termes
    phi = 2*math.pi*k/ne
    W = math.cos(phi)+math.sin(phi)*1j
    a = k/nem1 # premiere ou seconde moitie du paquet
    if a==0:
        y[indice] = x[indice]+W*x[indice+nem1]
    else:
        y[indice] = x[indice-nem1]+W*x[indice]
```

Voici la fonction fft qui utilise cette méthode :

```
def fft2(u,q):
    N = 2**q
    x = numpy.array(u,dtype=complex)
    y = numpy.zeros(2**q,dtype=complex)
    rangerInversionBits(q,x,y)
    z = x
    x = y
    y = z
    for e in range(1,q+1):
        for indice in range(N):
            etapeFFT_unite(x,y,q,e,indice)
        z = x
        x = y
        y = z
    return x
```

Exemple :

```
tfd = fft2(u,q)
spectre = numpy.absolute(tfd)
figure(figsize=(10,4))
stem(i,spectre,'r')
```



4.b. Shaders openGL ES

Avec l'interface de programmation OpenGL, le processeur graphique est programmé au moyen de Shaders. On considère ici le langage OpenGL ES Shading language (utilisé dans WebGL et sur les appareils mobiles).

La FFT s'effectue sur une texture en virgule flottante, en plusieurs étapes. Chaque étape consiste à modifier en parallèle tous les points de la texture. La texture doit être carrée, avec une taille puissance de 2.

Le Vertex Shader se charge de transmettre les positions des vertex. On effectue simplement le tracé d'un carré constitué de deux triangles.

```
attribute vec2 aVertexPosition;
void main(void){
    gl_Position = vec4(aVertexPosition,0.0,1.0);
}
```

La première étape consiste à ne garder que la couche 0 de la texture (la couche rouge) et à annuler la couche 1 (couche verte). La première servira à stocker la partie réelle, la seconde à stocker la partie imaginaire. Voici le Fragment Shader qui réalise cette opération.

```
precision mediump float;
uniform sampler2D uSampler;
uniform float uRowSize;
void main() {
  float x = gl_FragCoord.x/uRowSize;
  float y = gl_FragCoord.y/uRowSize;,
  vec4 v = texture2D(uSampler,vec2(x,y));
  gl_FragColor = vec4(v.r,0.0,0.0,1.0);
}
```

La seconde étape consiste à trier les lignes en inversant les bits des indices. Le nombre de bits de la taille de la texture doit être codé en dur (9 dans l'exemple ci-dessous, pour une texture de taille 512).

```
        precision mediump float;
         uniform sampler2D uSampler;
         uniform float uRowSize;
         void main() {
           float index=0.0;
           float i = gl_FragCoord.x;
           float j =  gl_FragCoord.y;
           float i2,b;
           float s = uRowSize/2.0;
           for (int k=0; k<9; k++) { // nombre de bits à coder en dur
               i2 = floor(i/2.0);
               b = i-i2*2.0;
               if (b>0.0) index += s;
```

```
        s /= 2.0;
        i = i2;
    }
    float x = index/uRowSize;
    float y = j/uRowSize;
    vec4 v = texture2D(uSampler,vec2(x,y));
    gl_FragColor = vec4(v.r,v.g,0.0,1.0);
  }
```

Une étape de la FFT sur les lignes est réalisée par le Fragment Shader suivant. Il faut transmettre au shader le nombre $n(e)$ et le nombre $n(e-1)$.

```
precision mediump float;
 uniform sampler2D uSampler;
 uniform float uRowSize;
 uniform float uNe;
 uniform float uNem1;
 uniform float uF;
 uniform float uNorm;
 float i = gl_FragCoord.x;
 float j =  gl_FragCoord.y;
 void main(){
   float k = mod(i,float(uNe));
   float phi = uF*k;
   float cos = cos(phi);
   float sin = sin(phi);
   int a = int(k)/int(uNem1);
   vec4 v;
   float re1,im1,re2,im2;
   if (a==0) {
       v = texture2D(uSampler,vec2(i/uRowSize,j/uRowSize));
       re1 = v.r;
       im1 = v.g;
       v = texture2D(uSampler,vec2((i+uNem1)/uRowSize,j/uRowSize));
       re2 = v.r;
       im2 = v.g;
       gl_FragColor =
       vec4((re1+re2*cos-im2*sin)*uNorm,(re2*sin+im2*cos+im1)*uNorm,0.0,1.0);
   },
 else {,
       v = texture2D(uSampler,vec2((i-uNem1)/uRowSize,j/uRowSize));
       re1 = v.r;
       im1 = v.g;
       v = texture2D(uSampler,vec2(i/uRowSize,j/uRowSize));
       re2 = v.r;
       im2 = v.g;
       gl_FragColor =
       vec4((re1+re2*cos-im2*sin)*uNorm,(re2*sin+im2*cos+im1)*uNorm,0.0,1.0);
```

```
    }
  }
```

Pour une texture de 512 pixels de large, il faut appliquer les étapes avec $n(e)$ variant de 1 à 9. Une fois la FFT effectuée sur les lignes, il faut trier les colonnes en inversant les bits des indices puis appliquer les étapes de la FFT sur les colonnes. Les shaders correspondants sont analogues aux précédents.

Pour afficher le spectre, il peut être nécessaire d'ordonner la TFD de manière à placer la fréquence nulle au centre de l'image. Cela est réalisé par le shader suivant, auquel ont doit fournir la taille de la texture et sa moitiée :

```
precision mediump float;
 uniform sampler2D uSampler;
 uniform float uRowSize;
 uniform float uRowSized2;
 void main(){
   float i = gl_FragCoord.x-0.5;
   float j =  gl_FragCoord.y-0.5;
   if (i<=uRowSized2) i = uRowSized2-1.0+i;
   else i = -uRowSized2-1.0+i;
   if (j<=uRowSized2) j = uRowSized2-1.0+j;
   else j = -uRowSized2-1.0+j;
   gl_FragColor =
   vec4(texture2D(uSampler,vec2((i+0.5)/uRowSize,(j+0.5)/uRowSize)));
 }
```

Pour effectuer la transformation inverse :

```
precision mediump float;
 uniform sampler2D uSampler;
 uniform float uRowSize;
 uniform float uRowSized2;
 void main(){
   float i = gl_FragCoord.x-0.5;
   float j =  gl_FragCoord.y-0.5;
   if (i<uRowSized2-1.0) i = uRowSized2+1.0+i;
   else i = -uRowSized2+1.0+i;
   if (j<uRowSized2-1.0) j = uRowSized2+1.0+j;
   else j = -uRowSized2+1.0+j;
   gl_FragColor =
   vec4(texture2D(uSampler,vec2((i+0.5)/uRowSize,(j+0.5)/uRowSize)));
 }
```

Voici un exemple d'utilisation sur WebGL : Diffraction par une ou plusieurs ouvertures.

Références

[1] W.H. Press, S.A. Teukolsky, W.T. Vetterling, B.P. Flannery, *Numerical recipes in Fortran*, (Cambridge University Press, 1992)