Transformée de Fourier discrète rapide (FFT)

S.Gibet

Année 2021

Nous présentons ici une application du précepte "diviser pour résoudre" qui a révolutionné le monde du traitement du signal : la transformée de Fourier rapide discrète, ou FFT (Fast Fourier Transform). Cette technique a rendu possible en pratique l'utilisation des techniques de filtrage par transformée de Fourier, en faisant passer d'un coût quadratique à un coût en *nlog(n)*.

1 Transformée de Fourier et transformée discrète

Définitions

- La **transformée de Fourier F** d'une fonction f est définie par :
  $\hat{s}(x) = S(x) = \int s(t)e^{-2\pi ixt}dt$
- Le **produit de convolution** f*g de deux fonctions est :

  $(f * g)(x) = \int f(x-u)g(u)du$

  Propriété : Transformée de Fourier et produit de convolution sont reliés par la relation :
  $\widehat{f * g} = F.G$

Définition : Soit $x = (x_0...x_{n-1})$ un vecteur dont les coefficients sont des nombres complexes. Sa transformée de Fourier discrète à l'ordre n est un vecteur complexe noté
$X = (c_k)_{k=0...n-1}$, défini par :

$c_k = \sum_{j=0}^{n-1} x_j \omega^{kj}$

où $\omega = e^{-2i\pi/n}$ est une racine n^{ieme} de l'unité. En notation matricielle, on peut écrire :

$$X = \begin{pmatrix} 1 & ... & 1 & ... & 1 \\ . & & . & & . \\ . & & . & & . \\ . & & . & & . \\ 1 & ... & \omega^{kj} & ... & \omega^{k(n-1)} \\ . & & . & & . \\ . & & . & & . \\ . & & . & & . \\ 1 & ... & \omega^{(n-1)j} & ... & \omega^{(n-1)^2} \end{pmatrix} = \Omega x$$

Remarque : transformée inverse :
$\left(\Omega^{-1}\right)_{jk} = \frac{1}{n} w^{-jk}$

Le calcul direct d'une transformée de Fourier en utilisant cette définition est en $O(n^2)$ opérations (n additions et multiplications pour chacun des n termes de X. Pour des problèmes réels (signal de parole, image, etc.), n est autour de valeurs comme 256 ou 1024, voire 512 x 512. Il est donc important de réduire le coût de cette transformation.

2 Algorithme FFT : diviser pour résoudre

Supposons pour simplifier que n est une puissance de 2 ($n = 2^N$), et posons $n = 2m$. Coupons chacun des termes c_k à calculer en deux moitiés :

$c_k = \sum_{j=0}^{m-1} x_j \omega^{kj} + \sum_{j=m}^{n-1} x_j \omega^{kj}$

En posant $j' = j - m$, on a :

$c_k = \sum_{j=0}^{m-1} x_j \omega^{kj} + \sum_{j'=0}^{n-m-1} x_{j'+m} \omega^{k(j'+m)} = \sum_{j=0}^{m-1} x_j \omega^{kj} + \sum_{j'=0}^{n-m-1} x_{j'+m} \omega^{kj'} \omega^{km}$

Or, on peut remarquer que $\omega^m = e^{-j\frac{2\pi m}{n}} = e^{-j\pi} = -1$

- Si k est pair, $k = 2r$, et $\omega^{km} = (-1)^{2r} = 1$. D'où :

  $c_{2r} = \sum_{j=0}^{m-1} (x_j + x_{j+m}) \omega^{2rj}$

  Or, $\omega^2 = e^{-j\frac{2\pi i}{m}}$, et les nombres c_{2r} constituent la transformée de Fourier à l'ordre $m = n/2$ du vecteur de composantes :
  $\alpha_j = x_j + x_{j+m}$ avec $(0 \leq j \leq m-1)$

- Si k est impair, $k = 2r + 1$, et $\omega^{km} = -1$. Et donc :
  $\omega^{(2r+1)(j+m)} = \omega^{(2r+1)j} \omega^{km} = -\omega^{(2r+1)j}$.

  D'où $c_{2r+1} = \sum_{j=0}^{m-1} (x_j - x_{j+m}) \omega^j \omega^{2rj}$

On reconnait la transformeée de Fourier l'ordre $m = n/2$ du vecteur de composantes : $\beta_j = (x_j - x_{j+m})\omega^j$ avec $(0 \leq j \leq m-1)$

Algorithme Ce qui précède conduit à l'algorithme :
Si la taille de X est supérieure à 1 (sinon X est son propre transformé)

- calculer α et β, vecteurs de taille $m = n/2$
- appeler récursivement le calcul de FFT sur α et β
- entrelacer les termes des résultats pour former la FFT de X

Complexité en temps On s'est ramenés au calcul d'une transformée de Fourier d'ordre n à deux calculs de transformées de Fourier d'ordre $n/2$, dont il suffit d'entrelacer les termes (ce qui se fait en $O(n)$) pour obtenir le résultat. On a $T(n) = nlog(n)$.

3 Vers une bonne implantation

Si on implante l'algorithme précédent directement, on risque de se heurter à des problèmes de *place mémoire*, du fait du stockage de vecteurs intermédiaires α et β. Notez que tous les calculs sont faits avant les appels récursifs et que donc la place des paramètres d'entrée est libre après ces calculs. En pratique, on utilise directement l'espace de X pour stocker les valeurs calculées en les rangeant dans les deux moitiés inférieures et supérieures de X, et le même espace servira pour y construire le résultat. D'où le programme :

Algorithm 1: FFT(X:Vecteur):Vecteur

Data:
$X1$: Vecteur

```
1    FFT1(k, q : entiers) // transforme la partie X1(k..q − 1)
2        taille := q - k;
3        si taille > 1 alors
4            omega := racine_primitive(1,taille);
5            m := taille div 2;
6            pour i de k à  k+m-1 faire
7                a := X1(i); b := X1(i+m);
8                X1(i) := a+b; X1(i+m) := (a − b) ∗ omega ∧ (i − k);
9            fin pour;
10           FFT1(k, k+m-1);
11           FFT1(k+m, q);
12           entrelacer(k, q); // entrelace les deux moitiés de X1(k..q)
13       fin si;
14    fin FFT1;
15 debut
16       X1 := X;
17       FFT1(0, taille(X));
18       return X1;
19 fin;
```

Entrelacer les deux moitiés du vecteur X consiste à effectuer la permutation p définie par :

- si $i < m$, $p(i) = 2i$
- si $i >= m$, $p(i) = 2(i - m) + 1$

Ce qui se regroupe en : $p(i) = 2(i \bmod m) + i \text{ div } m$. Or, nous avons supposé que n est la puissance de deux $n = 2^N$. Alors $m = 2^{N-1}$. Si on peut manipuler la représentation binaire de i, notée $i_N i_{N-1}...i_0$, alors $i \text{ div } m = i_N$ et $i \bmod m = (i_{N-1}...i_0)$. Par conséquent, $p(i)$, représenté par $(i_{N-1}...i_0 i_N)$ est obtenu par décalage circulaire. Au cours de l'algorithme, cette opération n'est plus effectuée sur X complet mais sur une partie du vecteur.

Si on analyse de manière globale le comportement de la fonction récursive FFT1, on voit qu'elle effectue une phase de descente (appels récursifs sur des tableaux de plus en plus petits), lors de laquelle les valeurs de X sont modifiées, puis une phase de remontée, avec entrelacements successifs des résultats partiels.

En fait, comme chaque entrelacement est une permutation, il est possible, et considérablement plus efficace, d'effectuer directement la permutation composée, plutôt que chacune successivement. Or, cette permutation composée est tout simplement constituée de $m = n/2$ échanges,

d'un indice à son image miroir en binaire. En séparant la partie récursive de la FFT et la phase d'entrelacements (itérative), on obtient :

Algorithm 2: FFT(X:Vecteur):Vecteur // indice allant de 0 à n-1

```
   Data:
   X1: Vecteur
   n : taille(X)
   omega_puissance(0..n − 1) de complexes
   // Ce tableau stocke les puissances successives de la racine primitive de l'unité
1    FFT2(k, q : entiers) // comme FFT1 sauf les entrelacements et les puissances de
2                         // omega précalculées
3        taille := q - k;
4        pas_omega := n/taille; // progression dans le tableau des puissances
5        si taille > 1 alors
6            m := taille div 2;
7            pour i de k à k+m-1 faire
8                a := X1(i); b := X1(i+m);
9                X1(i) := a+b; X1(i+m) :=
     (a − b) ∗ omega_puissance((i − k) ∗ pas_omega);
10           fin pour;
11           FFT2(k, k+m-1);
12           FFT2(k+m, q);
13       fin si;
14    fin FFT2;
15 debut
16       X1 := X;
17 //Précalcul des puissances de la racine primitive
18       omega := racine_primitive(1,n);
19       omega_puissance(0):=1;
20       pour i de 1 à n-1 répéter
21           omega_puissance(i) := omega_puissance(i-1)* omega;
22       fin pour;
23       FFT2(0, n);
24       pour j de 1 à taille(X) - 1// identité pour les deux extrêmes
25           j1 := miroir(j); // image miroir en notation binaire
26           si j1 > j alors X1(j) <-> X1(j1)/ sinon échange déjà fait
27       fin pour;
28       return X1;
29 fin;
```

4 Version itérative

Si on dessine l'arbre des appels de FFT, on se retrouve dans la même situation que dans celle du tri fusion, à la différence que tous les calculs sont faits avant les appels récursifs ici, alors que dans le cas du tri, ils sont faits après (la fusion).

La même technique de prise en compte des calculs peut être appliquée à cette modification près : on traite d'abord tout le tableau des données, puis les deux moitiés, puis les quatre quarts,

etc. Lorsque la taille des blocs atteint 1 tout est terminé, il ne reste plus que l'interclassent de l'algorithme précédent à effectuer.

On aboutit alors à l'algorithme suivant :

Algorithm 3: FFT(X: in out Vecteur) // X vecteur d'indice allant de 0 à n-1

Data:
n : taille(X)
$omega\_puissance(0..n-1)$ de complexes

```
1  debut
2  //Précalcul des puissances de la racine primitive
3        omega := racine_primitive(1,n);
4        omega_puissance(0):=1;
5        pour i de 1 à n-1 répéter
6              omega_puissance(i) := omega_puissance(i - 1)* omega;
7        fin pour;
8  // itération sur les niveaux
9         taille := n; pas_omega := 1;
10       tant que taille > 1 répéter
11           m := taille div 2;
12           k := 0;
13 // itération sur chaque bloc de grandeur taille
14           tant que k < n répéter
15               pour i de k à k+m-1 faire
16                   a := X(i); b := X(i+m);
17                   X(i) := a+b; X(i+m) :=
     (a − b) ∗ omega_puissance((i − k) ∗ pas_omega);
18               fin pour;
19               k := k+taille;
20           fin tant que;
21           taille := m; pas_omega := pas_omega*2;
22       fin tant que;
23 // Il ne reste plus qu'à interclasser comme dans l'algorithme précédent
24 fin;
```