Transformée de Fourier discrète

On dispose en mathématiques de quatre opérations dites "élémentaires" : l'addition, la soustraction, la division et donc la multiplication. On sait tous multiplier deux entiers en base 10 *: il suffit de faire la multiplication de chaque chiffre du multiplicateur par chaque chiffre du multiplicande, puis d'additionner le tout. Pour deux nombres de taille n, cela donne un algorithme de complexité $O(n^2)$. Mais dès que l'on veut multiplier de très grands chiffres (en informatique par exemple), cet algorithme montre très vite ses limites. Nous allons étudier ici le cas des polynômes en donnant un algorithme de multiplication utilisant la transformée de Fourier rapide.*

I - Transformée de Fourier discrète sur $\mathbb{C}$

1. Définitions

L'idée va être d'identifier les polynômes de degré inférieur à $n-1$ de la forme $\sum_{k=0}^{n-1} a_k x^k$ au vecteur de $\mathbb{C}^n$ $(a_0, \dots, a_{n-1})$. On fixe, pour toute la suite, ω une racine primitive n-ième de l'unité.

Définition 1. On appelle **transformée de Fourier discrète** l'application

$$\begin{array}{rccc} \mathrm{DFT}_\omega : & \mathbb{C}^n & \to & \mathbb{C}^n \\ & (f_0, \dots, f_{n-1}) & \mapsto & (\sum_{k=0}^{n-1} f_k, \sum_{k=0}^{n-1} f_k \omega^k, \dots, \sum_{k=0}^{n-1} f_k \omega^{(n-1)k}) \end{array}$$

Remarque 2. Si F est le polynôme associé au vecteur $f = (f_0, \dots, f_{n-1})$, on a

$$\mathrm{DFT}_\omega(f) = (F(1), \dots, F(\omega^{n-1}))$$

fait que nous utiliserons plusieurs fois dans la suite.

Dans un premier temps, on peut écrire l'algorithme de calcul suivant.

Algorithme 3.

```
# Exponentiation rapide. Calcul x^n.
def power(x, n):
    if n == 0:
        return 1
    if n == 1:
        return x
    if n % 2 == 0:
        return power(x*x, n//2)
    return x * power(x*x, (n-1)//2)

# Algorithme naïf pour calculer la transformée de Fourier rapide de f.
def naive_dft(f, omega):
    n = len(f)
    return [sum(f[k] * power(omega, i*k) for k in range(n)) for i in range
```

```
    (n)]
```

Définition 4. Le **produit de convolution** de deux vecteurs $f = (f_0, \dots, f_{n-1})$ et $g = (g_0, \dots, g_{n-1})$ de $\mathbb{C}^n$ est le vecteur de $\mathbb{C}^n$ noté $f *_n g$ défini par

$$f *_n g = \left(\sum_{i+j \equiv k \mod n} f_i g_j \right)_{k \in [\![0, n-1]\!]}$$

Exemple 5. $\mathrm{DFT}_{-1}(1,3) = (4,-2)$ et $(1,3) *_2 (1,3) = (1+9, 3+3) = (10,6)$.

2. Propriétés

Proposition 6. Soient $f = (f_0, \dots, f_{n-1})$, $g = (g_0, \dots, g_{n-1})$ deux vecteurs de $\mathbb{C}^n$. On pose $h = f *_n g = (h_0, \dots, h_{n-1})$ ainsi que F, G et H les polynômes associés respectivement à f, g et h. Modulo $X^n - 1$, on a :

$$\overline{H} = \overline{FG}$$

Démonstration. Écrivons :

$$H = \sum_{k=0}^{n-1} \left(\sum_{i+j \equiv k \mod n} f_i g_j \right) X^k$$

Comme H est au plus de degré $n-1$, on a $H = \overline{H}$ (par abus de notation). Maintenant avec $F = \sum_{k=0}^{n-1} f_k X^k$ et $G = \sum_{k=0}^{n-1} g_k X^k$, on a :

$$\begin{aligned} FG &= \sum_{k=0}^{2n-2} \left(\sum_{i+j=k} f_i g_{k-i} \right) X^k \\ &= \sum_{k=0}^{n-1} \left(\sum_{i+j=k} f_i g_{k-i} \right) X^k + \sum_{k=n}^{2n-2} \left(\sum_{i+j=k} f_i g_{k-i} \right) X^k \\ &= \sum_{k=0}^{n-1} \left(\sum_{i+j=k} f_i g_{k-i} \right) X^k + X^n \left(\sum_{k=0}^{n-2} \left(\sum_{i+j=n+k} f_i g_{k-i} \right) X^k \right) \end{aligned}$$

En passant modulo $X^n - 1$,

$$\begin{aligned}
\overline{FG} &= \overline{F}\,\overline{G} \\
&= \sum_{k=0}^{n-1}\left(\sum_{i=0}^{k} f_i g_{k-i}\right)\overline{X^k} + \sum_{k=n}^{2n-2}\left(\sum_{i=0}^{k} f_i g_{k-i}\right)\overline{X^k} \\
&\overset{\overline{X^n}=\overline{1}}{=} \sum_{k=0}^{n-1}\left(\sum_{i=0}^{k} f_i g_{k-i}\right)\overline{X^k} + \sum_{k=0}^{n-2}\left(\sum_{i+j=n+k} f_i g_{k-i}\right)\overline{X^k} \\
&= \sum_{k=0}^{n-1}\left(\sum_{i+j\equiv k \bmod n} f_i g_j\right)\overline{X^k} \\
&= \overline{H}
\end{aligned}$$

□

Théorème 7. DFT_ω est un isomorphisme d'algèbres entre $(\mathbb{C}^n, +, *_n)$ et $(\mathbb{C}^n, +, \cdot)$ dont la matrice dans la base canonique est la matrice de Vandermonde :

$$V_\omega = \begin{pmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & \omega & \omega^2 & \dots & \omega^{n-1} \\ 1 & \omega^2 & \omega^4 & \dots & \omega^{2(n-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^{n-1} & \omega^{2(n-1)} & \dots & \omega^{(n-1)^2} \end{pmatrix}$$

(où $\cdot$ est le produit sur $\mathbb{C}^n$ effectué composante par composante.)

Démonstration. Soient $f = (f_0, \dots, f_{n-1})$ et $g = (g_0, \dots, g_{n-1})$ deux vecteurs de $\mathbb{C}^n$.

— $\forall \lambda \in \mathbb{C}$,

$$\begin{aligned}
\mathrm{DFT}_\omega(\lambda f + g) &= \left(\sum_{k=0}^{n-1} \lambda f_k + g_k, \dots, \sum_{k=0}^{n-1} (\lambda f_k + g_k)\omega^{(n-1)k}\right) \\
&= \lambda\left(\sum_{k=0}^{n-1} f_k, \dots, \sum_{k=0}^{n-1} f_k \omega^{(n-1)k}\right) + \left(\sum_{k=0}^{n-1} g_k, \dots, \sum_{k=0}^{n-1} g_k \omega^{(n-1)k}\right) \\
&= \lambda\,\mathrm{DFT}_\omega(f) + \mathrm{DFT}_\omega(g)
\end{aligned}$$

DFT_ω est bien une application linéaire.

— Soit $h = f *_n g$. On note F, G et H les polynômes respectivement associés à f, g et h.
Soit $i \in [\![0, n-1]\!]$. Par la Proposition 6, on a $H = FG + Q(X^n - 1)$. Ainsi,

$$H(\omega^i) = (FG)(\omega^i) + Q(\omega^i)((\omega^i)^n - 1) = (FG)(\omega^i)$$

Or, le $(i+1)$-ième coefficient de $\mathrm{DFT}_\omega(f) \cdot \mathrm{DFT}_\omega(g)$ est

$$F(\omega^i)G(\omega^i) = (FG)(\omega^i) = H(\omega^i)$$

Donc $\mathrm{DFT}_\omega(f) \cdot \mathrm{DFT}_\omega(g) = \mathrm{DFT}_\omega(h)$, et ainsi, DFT_ω est bien un morphisme d'algèbres.

— Clairement,
$$\mathrm{DFT}_\omega(f_0,\dots,f_{n-1}) = V_\omega\,{}^t(f_0,\dots,f_{n-1})$$
donc la matrice dans la base canonique de DFT_ω est V_ω. Or, les ω^k sont distincts deux-à-deux, donc V_ω est inversible (car de déterminant non nul, en vertu de la formule de Vandermonde).

□

Proposition 8. L'inverse de DFT_ω est donné par $\frac{1}{n}\mathrm{DFT}_{\omega^{-1}}$.

Démonstration. Déjà, remarquons que si ω est une racine primitive n-ième de l'unité, alors ω^{-1} aussi :

— $(\omega^{-1})^n = (\omega^n)^{-1} = 1$.

— Et si on a $m < n$ tel que $(\omega^{-1})^m = (\omega^m)^{-1} = 1$, alors $\omega^m = 1$, ce qui est absurde.

Il suffit donc de montrer que $V_\omega V_{\omega^{-1}} = nI_n$. Soient $i,j \in [\![1,n]\!]$. En notant $c_{i,j}$ le coefficient à la i-ième ligne et à la j-ième colonne de $V_\omega V_{\omega^{-1}}$, on a :
$$c_{i,j} = \sum_{k=1}^{n} \omega^{(i-1)(k-1)}\omega^{(1-j)(k-1)} = \sum_{k=0}^{n-1} \omega^{(i-j)k} = \begin{cases} n & \text{si } i=j \\ \frac{1-\omega^{(i-j)n}}{1-\omega^{i-j}} \overset{\omega^n=1}{=} 0 & \text{sinon} \end{cases}$$

Ce qu'on voulait. □

3. Application à la multiplication de polynômes

Notre but ici va être de trouver un moyen de calculer la transformée de Fourier discrète d'un polynôme de manière efficace, puis d'en déduire un algorithme de multiplication de deux polynômes.

Proposition 9. Soit $n = 2^k$, soit ω une racine n-ième de l'unité et soit $F \in \mathbb{C}[x]$ de degré inférieur ou égal à n. On suppose qu'il existe F_0 et F_1 tels que $F = F_1X^{\frac{n}{2}} + F_0$ et on pose $R_0 = F_0 + F_1$ et $R_1 = F_0 - F_1$. Alors
$$\forall k \in [\![0, \frac{n}{2}]\!],\ F(\omega^{2k}) = R_0(\omega^{2k}) \text{ et } F(\omega^{2k+1}) = R_1(\omega^{2k+1})$$

Démonstration. Écrivons $F = F_1X^{\frac{n}{2}} + F_0$. On a donc
$$F - F_0 - F_1 = F_1(x^{\frac{n}{2}} - 1) \iff F = F_1(x^{\frac{n}{2}} - 1) + R_0$$

Soit $k \in [\![0, \frac{n}{2}]\!]$, on évalue en ω^{2k} :
$$F(\omega^{2k}) = F_1(\omega^{2k})(\omega^{nk} - 1) + R_0(\omega^{2k}) = R_0(\omega^{2k})$$

et la deuxième égalité s'obtient par un calcul similaire (en utilisant le fait que $\omega^{\frac{n}{2}} = -1$). □

Proposition 10. Si ω est une racine primitive n-ième de l'unité, alors ω^2 est une racine primitive $\frac{n}{2}$-ième de l'unité.

Démonstration. Clairement, ω^2 est une racine $\frac{n}{2}$-ième de l'unité. Maintenant, si $d \mid \frac{n}{2}$, alors

$$(\omega^2)^d = 1 \iff \omega^{2d} = 1 \iff 2d = n \iff d = \frac{n}{2}$$

□

On déduit de la Proposition 9 et de la Proposition 10 un algorithme récursif de calcul de DFT_ω qui a une complexité de $O(n\ln(n))$.

Algorithme 11.

```
# Renvoie les indices impairs d'une liste.
def odd_indices(l):
  return [l[2*k+1] for k in range(len(l)//2)]

# Fusionne deux listes de longueur égale en alternant les termes.
def alternate_merge(l1, l2):
  return [val for pair in zip(l1, l2) for val in pair]

# Calcule et renvoie les puissances successives de la racine primitive n-
    ième omega.
def primitive_root_powers(omega, n):
  return [power(omega, k) for k in range(n)]

# Cette fonction renvoie la transformée de Fourier discrète de F en omega.
    La liste l demandée est la liste des puissances de omega.
def fft(F, m, l):
  n = m + 1
  if n == 1:
    return [0] if F == 0 else [F.list()[0]]
  (F1, F0) = F.quo_rem(X^(n/2))
  R0 = F0+F1
  R1 = F0-F1
  l2 = odd_indices(l)
  return alternate_merge(fft(R0, n/2-1, l2), fft(R1.substitute(X=l[1]*X),
    n/2-1, l2))
```

Théorème 12. Soient F et G deux polynômes de degré strictement inférieur à $\frac{n}{2}$ dont on note f et g les vecteurs de $\mathbb{C}^n$ associés. Alors

$$FG = H$$

où H est le polynôme associé au vecteur $\frac{1}{n}\mathrm{DFT}_{\omega^{-1}}(\mathrm{DFT}_\omega(f)\cdot\mathrm{DFT}_\omega(g))$.

Démonstration. Comme $\deg(FG) < n$, on a $FG = H = \sum_{i=1}^{n} h_k X^k$ où $h = (h_0, \dots, h_{n-1}) = f *_n g$ par la Proposition 6. Par le Théorème 7, on a

$$\mathrm{DFT}_\omega(h) = \mathrm{DFT}_\omega(f *_n g) = \mathrm{DFT}_\omega(f) \cdot \mathrm{DFT}_\omega(g)$$

Et $\mathrm{DFT}_\omega^{-1} = \frac{1}{n}\mathrm{DFT}_{\omega^{-1}}$ par la Proposition 8. On obtient le résultat voulu. □

En utilisant l'algorithme écrit précédemment, on peut donc écrire un nouvel algorithme permettant de calculer le produit de deux polynômes de degré strictement inférieur à $\frac{n}{2}$ en $O(n\ln(n))$.

Algorithme 13.

```
# Multiplie deux polynômes de degré n.
def fast_polynomial_multiply(F, G, n, omega):
    l1 = primitive_root_powers(omega, n)
    l2 = primitive_root_powers(power(omega, n-1), n)
    prod = [fft(F, n-1, l1)[i] * fft(G, n-1, l1)[i] for i in range(n)]
    h = fft(sum(prod[k] * X^k for k in range(len(prod))), n-1, l2)
    return sum(1/n * h[k] * X^k for k in range(len(h)))
```

Remarque 14. On pourrait imaginer un algorithme calculant le produit de deux polynômes de degrés quelconques n et m sur le même modèle en considérant F et G comme des polynômes de degré 2^k où k est tel que $2^{(k-1)} \leq \max(n, m) \leq 2^k$. Il suffit ensuite de choisir ω racine primitive 2^k-ième de l'unité.