INF431

La transformée de Fourier rapide
CORRIGÉ
Version: 2063:2113M

Introduction

La transformée de Fourier discrète joue un rôle central en traitement du signal et notamment dans les mécanismes de compression avec perte en MP3 et en JPEG. Dans une version simplifiée, un signal (par exemple du son) est d'abord discrétisé en temps ; chaque (petite) tranche de temps fournit n valeurs $(a_0, \ldots, a_{n-1})$ qui sont alors converties en n coefficients $(b_0, \ldots, b_{n-1})$ exprimant le même signal discret, mais dans le domaine fréquentiel. La compression est obtenue en enlevant les coefficients correspondant aux fréquences moins audibles (trop hautes ou trop basses).

Les coefficients $(b_0, \ldots, b_{n-1})$ sont obtenus par l'application :

$$\mathrm{DFT} : (a_0, \ldots, a_{n-1}) \mapsto (A(1), A(\omega), \ldots, A(\omega^{n-1})), \quad \text{où} \quad A(X) = a_0 + a_1 X + \cdots + a_{n-1} X^{n-1},$$

et $\omega \in \mathbb{C}$ est une racine $n$ième de l'unité ($\omega^n = 1$), et de surcroît primitive ($\omega^\ell \neq 1$ pour $1 \leq \ell < n$). Le calcul rapide de la transformée de Fourier discrète (DFT) fournit une illustration des techniques de diviser-pour-régner qui mène à l'algorithme de transformée de Fourier rapide (souvent abrégé FFT pour *fast Fourier transform*) [1].

Cet algorithme est par ailleurs au cœur de l'algorithmique rapide du calcul sur les entiers, les polynômes et les séries et nous en fournirons quelques exemples (multiplication et, si le temps le permet, division et applications). Dans tout ce sujet, il sera commode de considérer le n-uplet $(a_0, \ldots, a_{n-1})$ comme une structure de données pour représenter le polynôme $A(X)$ et calculer un polynôme signifie en calculer les coefficients.

1 L'algorithme de FFT

Question 1 Écrire le pseudo-code d'un algorithme naïf de calcul de DFT et estimer sa complexité. ◇

Solution. Il suffit d'une double boucle

procédure NAIVEDFT$((a_0, \ldots, a_{n-1}), \omega)$
$r := 1$;
pour $i = 0, \ldots, n-1$
faire $\begin{cases} b_i := a_{n-1}; \\ \textbf{pour } j = n-2, \ldots, 0 \\ \quad \textbf{faire } b_i := b_i * r + a_j; \\ r := \omega * r; \end{cases}$
renvoyer $(b_0, \ldots, b_{n-1})$

La complexité est clairement en $O(n^2)$ opérations du fait des deux boucles imbriquées au cours de chacune desquelles est effectué un nombre constant d'opérations. □

L'observation suivante fournit le point de départ de la récursion qui mène à l'algorithme rapide.

1. Cet algorithme, connu de Gauss et redécouvert par Cooley & Tuckey en 1965, figure dans la liste des *«Top 10 Algorithms of the 20th century»* à l'url http://www.siam.org/pdf/news/637.pdf

Question 2 Montrer que si $n = 2m$, alors ω^k est une racine du polynôme $X^m + 1$ lorsque k est impair et du polynôme $X^m - 1$ sinon. ◇

Solution. Si $k = 2p$, alors $(\omega^k)^m = \omega^{2mp} = (\omega^n)^p = 1$ et donc ω^k est racine de $X^m - 1$. Sinon, $k = 2p + 1$ et $(\omega^k)^m = (\omega^{2p})^m \omega^m = \omega^m$. Comme ω est racine primitive de l'unité et $0 < m < n$, $\omega^m - 1$ est différent de 0. L'identité

$$0 = \omega^{2m} - 1 = (\omega^m - 1)(\omega^m + 1)$$

montre donc que $\omega^m = -1$, ce qui conclut. □

Pour un polynôme $A(X) = a_0 + a_1 X + \cdots$ de degré inférieur à $2m$, on définit alors des polynômes $R_0(X)$ et $R_1(X)$ de degrés inférieurs à m par divisions euclidiennes :

$$A(X) = Q_0(X)(X^m - 1) + R_0(X), \quad A(X) = Q_1(X)(X^m + 1) + R_1(X).$$

Question 3 Montrer que si k est pair, alors $A(\omega^k) = R_0(\omega^k)$ et que si k est impair, $A(\omega^k) = R_1(\omega^k)$.◇

Solution. Il suffit d'évaluer les équations à l'aide de la question précédente. □

Question 4 Montrer que les polynômes R_0 et R_1 sont donnés par les formules

$$R_0 = (a_0 + a_m) + (a_1 + a_{m+1})X + \cdots, \quad R_1 = (a_0 - a_m) + (a_1 - a_{m+1})X + \cdots.$$

En déduire que le calcul du polynôme R_0 s'effectue en au plus m opérations, et que, les puissances de ω étant données, celui du polynôme $\overline{R}_1(X) = R_1(\omega X)$ s'effectue en au plus $2m$ opérations. ◇

Solution. La division euclidienne par $X^m \pm 1$ s'obtient facilement par l'écriture :

$$A(X) = a_0 + a_1 X + \cdots + a_{m-1}X^{m-1} + (X^m - 1 + 1)(a_m + a_{m+1}X + \cdots + a_{2m-1}X^{m-1}).$$

Les polynômes R_0 et R_1 s'obtiennent donc en au plus m additions ou soustractions de coefficients. Ensuite, le polynôme $\overline{R}_1$ s'obtient en multipliant le coefficient de X^i dans R_1 par ω^i, pour $i = 1, \ldots, m-1$. Au total, on a donc bien $2m$ opérations. □

Question 5 En déduire le pseudo-code d'un algorithme de calcul récursif de la transformée de Fourier discrète dans le cas où n est une puissance de 2. Estimer sa complexité. ◇

Solution. On applique la récursivité jusqu'au cas du degré 0 :

procédure DISCRETEFOURIERTRANSFORM$(P, [1, \omega, \ldots, \omega^{n-1}])$
si $n = 1$
alors renvoyer P
sinon
- $m := n/2$;
- **pour** $i = 0, \ldots, m-1$
  - **faire**
    - $R_0[i] := p[i] + p[m+i]$;
    - $\overline{R}_1[i] := (p[i] - p[m-i])\omega^i$;
- $valR_0 :=$ DISCRETEFOURIERTRANSFORM$(R_0, [1, \omega^2, \ldots, \omega^{n-2}])$;
- $\overline{valR}_1 :=$ DISCRETEFOURIERTRANSFORM$(\overline{R}_1, [1, \omega^2, \ldots, \omega^{n-2}])$;
- **pour** $i = 0, \ldots, m-1$
  - **faire**
    - $R[2 * i] = valR_0[i]$;
    - $R[2 * i + 1] = valR_1[i]$;
- **renvoyer** R

Les polynômes peuvent être stockés dans des tableaux de coefficients, et les opérations calculant R_0 et R_1 s'écrivent à l'aide d'une simple boucle. La complexité $C(n)$ vérifie

$$C(n) = \frac{3n}{2} + 2C(n/2) = \frac{3n}{2} + 2(\frac{3n}{4} + 2C(n/2)) = \cdots = \frac{3n}{2}\log_2 n.$$ □

Un aspect important que nous ne développons pas ici mais qui explique le succès de la FFT est qu'outre son efficacité, elle présente une excellente stabilité numérique : un calcul avec des coefficients flottants ne développe pas d'erreurs excessives.

2 Transformée de Fourier inverse et produit de polynômes

Question 6 Montrer que l'application qui associe au polynôme $A(X) = a_0 + a_1 X + \cdots + a_{n-1}X^{n-1}$ les valeurs $(A(1), \ldots, A(\omega^{n-1}))$ est linéaire et indiquer sa matrice V_ω dans la base $\{1, X, \ldots, X^{n-1}\}$.◇

Solution. La linéarité est claire. La matrice a donc pour colonnes les puissances des ω^i (elle est connue sous le nom de matrice de Vandermonde) :

$$V_\omega = \begin{bmatrix} 1 & 1 & \cdots & 1 \\ 1 & \omega & \cdots & \omega^{n-1} \\ \vdots & & & \vdots \\ 1 & \omega^{n-1} & \cdots & \omega^{(n-1)^2} \end{bmatrix}.$$

□

Question 7 Montrer que lorsque ω est une racine de l'unité, alors $V_{\omega^{-1}} V_\omega = nI_n$. ◇

Solution. Le coefficient d'indice (i, j) du produit $V_{\omega^{-1}} V_\omega$ vaut

$$\sum_{k=0}^{n-1} \omega^{-(i-1)k} \omega^{(j-1)k} = \sum_{k=0}^{n-1} \omega^{(j-i)k}.$$

Lorsque $i = j$, chaque sommant vaut 1, sinon on a affaire à la somme d'une série géométrique dont la nullité est claire, d'où le résultat. □

Question 8 En déduire que l'on peut reconstruire un polynôme de degré inférieur à n à partir de ses valeurs sur $1, \ldots, \omega^{n-1}$ (cette opération s'appelle une *interpolation*) et que ce calcul peut être effectué en $O(n \log n)$ opérations. ◇

Solution. La matrice étant inversible, l'interpolation est un produit par la matrice $V_{\omega^{-1}}$ suivi par une division par n. Ce produit s'obtient donc par une transformée de Fourier discrète utilisant les puissances de ω^{-1} (qui est aussi une racine primitive $n$ième de l'unité) à la place de celles de ω. La complexité découle de la question 5. □

Le principe du produit rapide de polynômes est alors de calculer d'abord l'évaluation des deux polynômes, puis de multiplier les valeurs deux à deux, et enfin d'interpoler le résultat.

Question 9 Écrire le pseudo-code d'un algorithme calculant le produit de deux polynômes de $\mathbb{C}[X]$ de somme des degrés au plus n en $O(n \log n)$ opérations dans $\mathbb{C}$. ◇

Solution. L'algorithme procède par évaluation-interpolation sur les puissances d'une racine de l'unité.

procédure FFTMULTIPLICATION(A, B)
$N := 1$;
tant que $N \leq n$
  faire $N := 2 * N$; — N est une puissance de 2 plus grande que n
$\omega := \exp(2i\pi/N)$;
pour $i = 2, \ldots, N-1$
  faire Calculer $\omega^i = \omega \times \omega^{i-1}$;
$ValA :=$ DISCRETEFOURIERTRANSFORM$(A, [1, \omega, \ldots, \omega^{N-1}])$;
$ValB :=$ DISCRETEFOURIERTRANSFORM$(B, [1, \omega, \ldots, \omega^{N-1}])$;
pour $i = 1, \ldots, N$
  faire $ValAB[i] := ValA[i] \times ValB[i]$;
renvoyer $\frac{1}{N}$DISCRETEFOURIERTRANSFORM$(ValAB, [1, \omega^{N-1}, \omega^{N-2}, \ldots, \omega])$

Le résultat de complexité provient des trois appels à la transformée de Fourier discrète, qui donne une complexité en $O(N \log N)$ et l'on conclut en observant que l'inégalité $N \leq 2n$ entraine $N \log N = O(n \log n)$. □

Cet algorithme forme la base de toute l'algorithmique rapide du calcul formel, avec des applications importantes en cryptographie ou en théorie des codes. En pratique, dans les bonnes implantations, plusieurs algorithmes de multiplication sont utilisés, avec des seuils automatiques choisissant l'algorithme le plus adapté en fonction du degré et du processeur. Pour les petits degrés, on utilise la multiplication naïve (parfois codée en assembleur), puis l'algorithme de Karatsuba et enfin la FFT. Les degrés typiques (par exemple pour la bibliothèque NTL) où l'algorithme de Karatsuba commence à être utilisé sont de l'ordre de la vingtaine, alors que la FFT est l'algorithme standard pour des polynômes dont le degré dépasse la centaine. De tels degrés apparaissent très couramment en cryptographie.

Les mêmes idées (Karatsuba, FFT) s'appliquent aussi au produit d'entiers, mais avec des complications pour la gestion des retenues.

3 Compléments : division et applications

Nous avons déjà obtenu que la multiplication n'est pas beaucoup plus chère que l'addition, et nous allons voir qu'il en va de même pour la division. Là encore, les idées sont à peu près les mêmes pour les entiers et les polynômes, mais plus faciles à décrire sur les polynômes, et encore plus sur les séries tronquées, sur lesquelles nous allons nous concentrer. Le n-uplet $(a_0, \dots, a_{n-1})$ représente donc maintenant la série

$$A(X) = a_0 + a_1 X + \cdots + a_{n-1} X^{n-1} + O(X^n),$$

et pour $k \leq n$, on notera $A(X) \bmod X^k$ le polynôme $a_0 + a_1 X + \cdots + a_{k-1} X^{k-1}$ obtenu en tronquant la série. Les opérations d'addition et de multiplication s'obtiennent comme auparavant (pour la multiplication, on doit tronquer le produit des polynômes $A(X) \bmod X^n$ et $B(X) \bmod X^n$).

La multiplication par la série tronquée $A(X)$ est une application linéaire, qui est inversible si $a_0 \neq 0$. L'inverse de $A(X)$ est alors l'unique série tronquée $B(X) = b_0 + \cdots + b_{n-1} X^{n-1} + O(X^n)$ telle que $A(X)B(X) = 1 + O(X^n)$. Multiplier une série par l'inverse d'une autre fournit une division que l'on appelle *division selon les puissances croissantes* (qui est utilisée pour le calcul de la décomposition en éléments simples d'une fraction rationnelle).

3.1 Approche classique

Question 10 Écrire le pseudo-code de la méthode classique pour effectuer la division selon les puissances croissantes «à la main», et estimer son nombre d'opérations arithmétiques dans $\mathbb{C}$ en fonction de n. ◇

Solution. L'algorithme s'écrit comme la division Euclidienne, mais «à l'envers» :

procédure NAIVESERIESDIVISION(B, A, n)
— Par hypothèse, $a_0 \neq 0$
$R := B; Q := 0;$
pour $i = 0$ **à** $n - 1$
faire
$c := \text{coeff}(R, i)/a_0;$ — coeff extrait le coefficient de tête
$Q := Q + cX^i;$
$R := R - cX^i A \bmod X^n;$
renvoyer Q

Il y a n itérations. Lors de ces itérations, l'extraction de coefficient ne coûte pas d'opération arithmétique, l'addition dans Q non plus puisqu'elle concerne un monôme qui n'y est pas encore présent. Tout le coût est donc concentré dans la mise à jour de R qui consiste en une multiplication par c et une soustraction pour chaque coefficient de R, donc $2(n-i)$ opérations au plus. Au total, le nombre d'opérations est donc en $O(n^2)$. □

3.2 Itération de Newton et diviser-pour-régner

Le principe de la division rapide est de calculer un développement tronqué de l'inverse du dénominateur et de le multiplier par le numérateur. Nous commençons par étudier le calcul d'inverse rapide.

Question 11 Montrer que le polynôme obtenu par l'itération de Newton [2]

$$Y_0 = 1/a_0, \qquad Y_{k+1} = Y_k + Y_k(1 - AY_k) \mod X^{2^{k+1}} \quad (k > 0)$$

vérifie $A(X)Y_k(X) = 1 + O(X^{\min(2^k,n)})$. ◇

Solution. La preuve est une récurrence sur k. La propriété est vraie pour $k = 0$. Ensuite, en multipliant par A et en soustrayant 1, on obtient les égalités suivantes :

$$\begin{aligned} AY_{k+1} - 1 + O(X^n) &= AY_k - 1 + AY_k(1 - AY_k) + O(X^n) + O(X^{2^{k+1}}) \\ &= (AY_k - 1)(1 - AY_k) = O(X^{\min(2^{k+1},n)}). \end{aligned}$$

Autrement dit, l'itération de Newton se traduit naturellement en un diviser-pour-régner, où il suffit de résoudre le problème à précision moitié, une seule fois.

Question 12 Écrire le pseudo-code d'une procédure prenant en entrée une série tronquée de terme constant non-nul et renvoyant son inverse. Estimer sa complexité. ◇

Solution. C'est une simple procédure récursive qui calcule d'abord à la précision moitié :

procédure NEWTONINVERSE(n, A)
si $n = 1$
alors renvoyer $1/u_0$
sinon
- $k := \lceil n/2 \rceil$;
- $Y :=$ NEWTONINVERSE(k, A);
- — On renvoie $Y + Y \times (1 - A \times Y)$ en tronquant les polynômes
- — chaque fois que c'est possible

renvoyer $\mathrm{rem}(Y + \mathrm{rem}(Y \times \mathrm{rem}((1 - \mathrm{rem}(A, X^n) \times Y), X^n), X^n), X^n)$

où $\mathrm{rem}(A, X^n)$ désigne l'opération de tronquer le polynôme au degré n.

Si $C(n)$ désigne la complexité en fonction de n, on a $C(1) = 1$ et pour $n > 1$,

$$C(n) \leq C(\lceil n/2 \rceil) + \kappa n \log n + cn,$$

où cn désigne le coût des additions.

Pour raisonner sur une telle inégalité, on commence par traiter le cas où n est une puissance de 2, où l'expression se simplifie et donne

$$\begin{aligned} C(n) &\leq \kappa n \log n + cn + C(n/2) \\ &\leq (\kappa n \log n + \kappa \frac{n}{2} \log \frac{n}{2}) + cn + cn/2 + C(n/4) \\ &\leq (1 + 1/2 + 1/4 + \dots)\kappa n \log n + 2cn \\ &\leq 2\kappa n \log n + O(n). \end{aligned}$$

Lorsque n n'est pas une puissance de 2, dans la somme ci-dessus, toutes les divisions par 2 sont remplacées par l'opération $x \mapsto \lceil x/2 \rceil$. Chacun de ces termes est majoré par un $\kappa(N/2^i)\log(N/2^i)$, où N est la puissance de 2 immédiatement supérieure à n. Leur somme est donc bornée par $O(n \log n)$.

Au final, on a donc dans tous les cas $C(n) = O(n \log n)$ [3]. □

2. Pour calculer l'inverse d'un nombre a, l'itération de Newton menant à la solution de l'équation $a - 1/y = 0$ s'écrit $y_{n+1} = y_n + y_n(1 - ay_n)$. La même itération est appliquée ici à des polynômes et mène à une bonne complexité.

3. On peut même être plus précis : Y n'a que degré $n/2$ et donc le produit AY coûte deux multiplications en degré $n/2$ en coupant A en 2. Ensuite, par hypothèse, $AY = 1 + O(X^k)$, de sorte que $1 - AY = X^k V$ avec V de degré $n/2$. Seuls les $n/2$ premiers coefficients du produit YV sont utiles ; ils ne coûtent donc que le coût du produit en précision $n/2$. Il en découle une complexité dominée par 3 produits en degré n, plus $O(n)$ opérations.

Pour parvenir à la division euclidienne des polynômes, il suffit d'effectuer ce calcul «à l'infini», c'est-à-dire en changeant X en $1/X$ et en faisant attention aux degrés. C'est facile, mais technique, et nous ne le traiterons pas. Au final, on peut effectuer la division euclidienne pour le prix d'environ deux multiplications seulement.

3.3 Logarithme et exponentielle

Avec cette division, il devient facile de calculer le développement en série d'un logarithme et d'une exponentielle. On rappelle les formules suivantes, où maintenant $A(X)$ désigne une série tronquée sans terme constant $a_1X + \cdots + a_{n-1}X^{n-1} + O(X^n)$ (c'est-à-dire que $a_0 = 0$ dans le n-uplet de coefficients) :

$$\log(1 + A(X)) = A(X) - \frac{1}{2}A(X)^2 + \frac{1}{3}A(X)^3 + \cdots + O(X^n),$$
$$\exp(A(X)) = 1 + A(X) + \frac{1}{2!}A(X)^2 + \frac{1}{3!}A(X)^3 + \cdots + O(X^n).$$

Question 13 Estimer la complexité d'une utilisation directe de ces formules. ◇

Solution. Pour $i > 1$, la puissance $A(X)^i$ a $n - i$ coefficients non-nuls, qui doivent être additionnés au résultat, ce qui entraîne déjà une complexité en $O(n^2)$. Le calcul de cette puissance s'obtient en multipliant la puissance précédente A^{i-1} par $A \bmod X^{n-i}$, en $O((n-i)\log(n-i))$ opérations, et ces produits dominent donc le calcul dont la complexité s'élève à $O(n^2 \log n)$ opérations arithmétiques. □

Question 14 Montrer que l'écriture $\log(1 + A) = \int A'/(1 + A)$ permet un calcul du logarithme en seulement $O(n \log n)$ opérations. ◇

Solution. Dériver la série A demande un nombre linéaire d'opérations (le coefficient de X^i est $(i + 1)a_{i+1}$) ; inverser $1 + A$ demande $O(n \log n)$ opérations d'après la question 12 ; le produit de A' par cet inverse demande encore $O(n \log n)$ opérations d'après la question 9 ; enfin intégrer la série demande à nouveau un nombre d'opérations linéaires. Au total, on a donc bien le résultat en $O(n \log n)$ opérations. □

Le calcul de l'exponentielle peut alors s'appuyer sur celui du logarithme, grâce à la relation facile $\log(\exp(A(X))) = A(X) + O(X^n)$. Il suffit, à nouveau, d'utiliser une itération de Newton pour résoudre l'équation $\log(Y) - A = 0$, ce qui mène à l'itération

$$Y_{k+1} = Y_k - Y_k(\log Y_k - A) \bmod X^{\min(2^{k+1}, n)}, \qquad Y_0 = 1.$$

Question 15 En admettant d'abord que $Y_k(X) - \exp(A(X)) = O(X^{\min(2^k, n)})$, montrer que l'exponentielle d'une série tronquée peut se calculer en seulement $O(n \log n)$ opérations, puis, s'il reste du temps, prouver cette relation. ◇

Solution. L'algorithme fonctionne comme celui de l'inverse en calculant d'abord Y à précision $n/2$ récursivement, puis en effectuant un calcul de logarithme, une multiplication, et quelques additions. La complexité $C(n)$ s'écrit donc à nouveau

$$C(n) \leq C(\lceil n/2 \rceil) + \kappa n \log n + cn,$$

ce qui mène au même résultat qu'à la question 12.

La convergence se prouve à nouveau par récurrence sur k : pour $k = 0$ la propriété est vraie. Ensuite, on conclut par la suite d'identités

$$\begin{aligned}
\log Y_{k+1} &= \log \Big(Y_k - Y_k(\log Y_k - A) + O(X^{\min(2^{k+1},n)}) \Big) \\
&= \log Y_k + \log(1 - (\log Y_k - A + O(X^{\min(2^{k+1},n)}))) \\
&= \log Y_k - (\log Y_k - A + O(X^{\min(2^{k+1},n)})) + O(X^{\min(2^{k+1},n)}) \\
&= A + O(X^{\min(2^{k+1},n)}) \\
Y_{k+1} &= \exp(A)(1 + O(X^{\min(2^{k+1},n)})) = \exp(A) + O(X^{\min(2^{k+1},n)}),
\end{aligned}$$

où l'hypothèse de récurrence est utilisée à la troisième ligne pour pouvoir développer le logarithme. □

Références

[1] Joachim von zur Gathen et Jürgen Gerhard. *Modern computer algebra*. Cambridge University Press, New York, 2nd édition, 2003.

[2] Jon Kleinberg et Éva Tardos. *Algorithm Design*. Addison Wesley, 2005.