Transformée de Fourier

Audric Schiltknecht

Année 2007

Table des matières

1 Introduction

Dans le cours de mathématiques, nous avons définit la série de Fourier d'une fonction continue par morceaux, 2π-périodique. Lors du cours de physique, nous avons étendu cette notion aux cas de fonctions T-périodiques. Mais qu'en est-il pour des fonctions quelconques, cas le plus fréquent ?

L'objet de ce document est l'étude de la *Transformée de Fourier* d'une fonction. Nous partirons de la définition formelle de cette transformation pour en arriver à une expression *approchée* beaucoup plus légère et plus facilement utilisable. Nous étudierons ensuite divers algorithmes permettant le calcul de cette transformation, et comparerons leur rapidité d'exécution. Enfin, nous appliquerons ces résultats à un exemple concret.

En annexe se trouvent les preuves des propriétés utilisées dans ce document, les *listings* des algorithmes, ainsi que les interprétations graphiques des divers résultats obtenus.

2 Définitions formelles

2.1 Notion de signal

Un signal x est une fonction du temps, bornée sur $\mathbb{R}$. Un signal est le support d'une information : cela peut être la tension aux bornes d'un générateur, l'intensité, ou l'enregistrement d'un son. On supposera dans toute la suite que le *support temporel*[1] du signal est une partie finie de $\mathbb{R}$ (on dit aussi que x est à *support compact*). Il faudra donc prendre en compte pour la suite que les symboles $\int$ et $\sum$ représentent en fait des sommes finies.

2.2 Transformée de Fourier, représentation fréquentielle d'un signal

On appelle (sous réserve d'existence)[2] *Transformée de Fourier* l'application $\mathfrak{F}$ (aussi notée $\widehat{.}$) définie *formellement* par :

$$\mathfrak{F}(x) : \nu \longmapsto \int_{-\infty}^{\infty} x(t) * e^{-2i\pi\nu t}\, dt \tag{2.1}$$

[1] Le support temporel du signal correspond à sa durée

[2] On supposera que ces conditions sont remplies, notament $x \in \mathcal{L}(\mathbb{R})$

Ainsi, on associe à un signal temporel x sa *représentation fréquentielle* (ν correspond à la *fréquence* du signal).

2.3 Limitations

On remarque que cette définition entraîne de grosses contraîntes sur la fonction x considérée.
Même si pour certaines fonctions vérifiant ces conditions d'existence, il est possible d'effectuer un calcul formel, cela n'est pas vraiment pratique pour le traitement de données expérimentales. Pour palier à cet inconvenient, on utilisera alors la *Transformée Discrète de Fourier*. Cette méthode nous permet alors de calculer la valeur de $\mathfrak{F}(x)(\nu)$ pour plusieurs valeurs (discrètes) de ν.

3 Notions d'échantillonage

Dans les technologies de télécommunications, les signaux sont de plus en plus souvent transmis sous forme numérique. Pour des raisons de rapidité et d'utilisation des moyens de transmissions, on n'enverra pas le signal de façon continue. Le principe de *l'échantillonage* est de remplacer le signal continu x par une suite de nombres $(x_n)_{n\in\mathbb{N}}$. Concrètement, cela correspond à prendre pour les x_n les valeurs du signal continu en certaines valeurs de t : $x_n = x(t_n), t_n \in \mathbb{R}$
Voyons comment cela peut affecter la *Transformée de Fourier* du signal.

3.1 Impulsion de Dirac

Définition Nous allons définir la suite de fonction $(x_n)_{n\in\mathbb{N}-\{0\}}$ par :
$\forall n \in \mathbb{N} - \{0\}$:

$$x_n(t) = \begin{cases} n & \text{si } t \in [-\frac{1}{2n}, \frac{1}{2n}] \\ 0 & \text{sinon} \end{cases}$$

Exemple On a représenté la fonction x_2 sur la figure 1.

Propriété On montre que :

$$\forall n \in \mathbb{N} - \{0\}, \quad \int_{-\infty}^{\infty} x_n(t)dt = 1$$

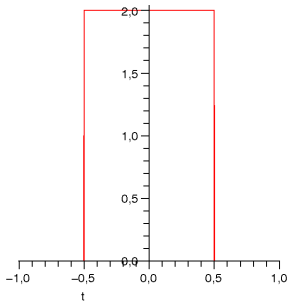

FIG. 1 – $x_2(t)$

Définition du Dirac On définit *l'impulsion de Dirac* comme étant la limite de la suite x_n. On la note δ. Une impulsion de Dirac **n'est pas** une fonction, c'est un objet mathématique que l'on nomme **mesure**.

Remarque Mathématiquement, on définit l'impulsion de Dirac comme *limite faible* de la suite (x_n), c'est-à-dire de la façon suivante :

$$\forall f \text{ continue en } 0, \lim_{n\to\infty} \int_{-\infty}^{+\infty} x_n(t)f(t) = f(0).$$

C'est aussi la forme linéaire définie par :

$$\delta : f \longrightarrow \delta(f) = f(0)$$

Propriétés

$$\delta(t) = 0 \text{ si } t \neq 0$$
$$\int_{-\infty}^{\infty} \delta(t)dt = 1$$
$$\forall g \text{ continue en } t_0, \quad \int_{-\infty}^{\infty} \delta(t - t_0) \times g(t)dt = g(t_0) \tag{3.1}$$

Bien que cela ne soit pas rigoureusement vrai, on peut "représenter" tel que sur la figure 2.

Peigne de Dirac On va définir le peigne de Dirac de période T_e de la façon suivante :

$$\delta_{T_e} \quad : \quad t \longmapsto \sum_{n\to-\infty}^{n\to\infty} \delta(t - nT_e)$$

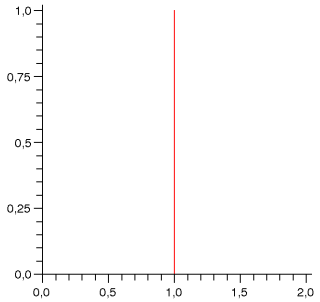

FIG. 2 – $\delta(t-1)$

3.2 Échantillonage

Soit x un signal que l'on souhaite échantillonner à la période T_e. Alors, on note x_e le signal échantillonné, défini par : $x_e(t) = x(t) \times \delta_{T_e}(t)$.

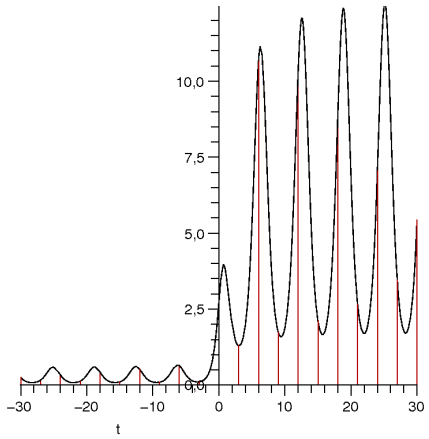

FIG. 3 – En noir le signal x, en rouge le signal "échantillonné" x_e

Quelle relation peut-il exister entre $\mathfrak{F}(x)$ et $\mathfrak{F}(x_e)$?

3.3 Expression de la Transformée de Fourier de x_e

La relation (3.1) nous donne :

$$\int_{-\infty}^{\infty} \delta(\nu - \nu_0) e^{-i2\pi\nu t} dt = e^{-i2\pi\nu_0 t}$$

Nous avons alors :

$$\begin{aligned}
\widehat{x_e}(\nu) &= \int_{\mathbb{R}} x_e(t) e^{-i2\pi\nu t} dt \\
&= \int_{\mathbb{R}} x(t) \sum_{n\in\mathbb{Z}} \delta(t - nT_e) e^{-i2\pi\nu t} dt \\
&= \sum_{n\in\mathbb{Z}} \int_{\mathbb{R}} x(t)\delta(t - nT_e) e^{-i2\pi\nu t} dt \\
\widehat{x_e}(\nu) &= \sum_{n\in\mathbb{Z}} x(nT_e) e^{-i2\pi\nu nT_e} \qquad (3.2)
\end{aligned}$$

Nous allons alors faire l'hypothèse suivante : considérons que le signal est un signal dit *à bande limitée*. Cela signifie que le *support fréquentiel*[3] est une partie finie de $\mathbb{R}$. En pratique c'est couramment le cas. En effet, les divers capteurs utilisés pour acquerir des signaux ont une plage de fréquences limitée. Voyons quelles sont les conséquences.

4 Théorème de Shannon

On admettra ici l'énoncé de ce théorème, proposé par Claude Shannon en 1948.

4.1 Énoncé

Soit x un signal analogique à support fréquentiel fini, c'est-à-dire que $\widehat{x}$ possède toutes ses fréquences dans l'intervalle $[-\nu_{max}, \nu_{max}]$[4]. Pour que sa discrétisation[5] à la fréquence $\frac{1}{T_e}$ ne lui fasse pas perdre d'information, il faut et il suffit que cette fréquence soit supérieure à deux fois la fréquence maximale contenue dans le signal, ie $\frac{1}{T_e} = \nu_e \geqslant 2 \times \nu_{max}$.

Dans ce cas, on a :

$$\forall \nu \in [-\nu_{max}, \nu_{max}], \quad \widehat{x}(\nu) = T_e \widehat{x_e}(\nu). \qquad (4.1)$$

[3] Analogue au support temporel, mais pour les fréquences

[4] Les fréquences négatives n'ont pas de sens physique, mais doivent être prises en compte dans cette transformation.

[5] L'échantillonage du signal

4.2 Exemple classique : Les CD Audios

Le spectre des fréquences audibles pour l'oreille humaine est d'environ [20 Hz, 20 kHz]. Ainsi, la valeur minimale pour la fréquence d'échantillonage est donc de 2× 20 kHz =40 kHz. En pratique, on prend 44,1 kHz.

5 Transformée de Fourier Discrète

5.1 Notations

On considère x un signal analogique. On échantillonne ce signal à la période T_e, et on prendra un nombre fini N de ces valeurs. On a alors la famille $(x_n)_{n\in[|0..N-1|]}$, avec $\forall n \in [|0..N-1|], x_n = x(nT_e)$ (on échantillonne à partir de $t_0 = 0$)
De même, on considère la famille $(\widehat{x}_n)_{n\in[|0..M-1|]}$, ensemble de M valeurs obtenues par échantillonnage du signal fréquentiel $\widehat{x}$, à la fréquence ν_f.
On notera $L = 2 \times \nu_{max}$. Nous disposons alors de $M = \frac{L}{\nu_f}$ valeurs de $\widehat{x}$.

Ainsi, pour respecter la condition de Shannon, il faut : $\frac{1}{T_e} \geqslant L$. Prenons par exemple $\frac{1}{T_e} = L$. Comme on doit pouvoir passer de la représentation temporelle à la représentation fréquentielle du signal, on doit donc prendre M=N (sans quoi, il y aurait perte d'informations). En simplifiant à l'aide de la valeur de M, il vient : $\nu_f = \frac{\nu_e}{N}$.

5.2 Expression des $\widehat{x}_n$ en fonction des x_n

On possède maintenant deux famillles finies : $(x_n)_{n\in[|0..N-1|]}$ et $(\widehat{x_n})_{n\in[|0..N-1|]}$.
Soit $k \in [|0..N-1|]$, existe-t-il un moyen de calculer $\widehat{x_k}$ en fonction des $(x_n)_{n\in[|0..N-1|]}$?
Par définition :

$$\widehat{x_k} = \widehat{x}(k \times \nu_f) \tag{5.1}$$

Or, ν_e vérifie le théorème de Shannon, donc d'après (4.1), il vient :

$$\forall \nu \in [-\nu_{max}, \nu_{max}] \quad : \quad \widehat{x}(\nu) = T_e \widehat{x_e}(\nu)$$

Comme tous les $\widehat{x_n}$ vérifient cette condition, en injectant dans (5.1), il vient :

$$\widehat{x_k} = T_e \widehat{x_e}(k \times \nu_f)$$

D'autre part, l'expression de la Transformée de Fourier (3.2) nous permet d'écrire l'égalité suivante :

$$\widehat{x_k} = T_e \sum_{\mathbb{Z}} x(nT_e) e^{-i2\pi k \nu_f n T_e} = T_e \sum_{\mathbb{Z}} x_n e^{-i2\pi n k \nu_f T_e}$$

En simplifiant grâce à l'expression de ν_f :

$$\widehat{x_k} = T_e \sum_{-\infty}^{\infty} x_n e^{-i\frac{2\pi nk}{N}}$$

Comme les x_n sont nuls pour $n \notin [|0..N-1|]$, il vient l'expression finale[6] :

$$\forall k \in [|0..N-1|], \quad \widehat{x_k} = T_e \sum_{n=0}^{N-1} x_n e^{-i\frac{2\pi nk}{N}} \tag{5.2}$$

5.3 Définition

On appellera *Transformée de Fourier Discrète* l'application qui, à une famille $(x_i)_{i\in[|0..N-1|]}$, associe la famille des $(\widehat{x_i})_{i\in[|0..N-1|]}$, définie par :

$$\forall k \in [|0..N-1|], \quad \widehat{x_k} = X_k = \sum_{j=0}^{N-1} x_j e^{-i\frac{2\pi jk}{N}} \tag{5.3}$$

Remarque Selon les ouvrages, on peut aussi la multiplier par $\frac{1}{N}$ ou $\frac{1}{\sqrt{N}}$.

5.4 Propriété

Similairement à la *Transformée de Fourier*, cette application est inversible, et on définit la *Transformée de Fourier Discrète Inverse* par :

$$\forall k \in [|0..N-1|], \quad x_k = \frac{1}{N} \sum_{j=0}^{N-1} X_j e^{i\frac{2\pi jk}{N}} \tag{5.4}$$

Elle ne diffère de la *Transformée Discrète* que d'un facteur 1/N et du signe de l'exponentielle. Il est maintenant temps de s'intéresser aux différentes méthodes de calcul de cette transformation.

6 Étude d'algorithmes

On suppose maintenant, et pour toute la suite, que l'on dispose d'une suite $(x_n)_{n\in[|1..N-1|]}$ composée de N éléments. On la stockera sous forme d'une *liste*, notée simplement x.[7] On récupèrera la Transformée de Fourier de cette liste sous la forme d'une autre liste, X.

[6] On remarque que cette expression est au plus une approximation

[7] Sous *Maple* et *Octave*, les élements d'une telle liste sont numérotés de 1 à N. En *C/C++*, c'est de 0 à $N-1$

6.1 Par une méthode "naïve"

La première méthode qui vient à l'esprit est de calculer les N sommes en utilisant la définition. Cependant, cette méthode est assez longue, puisqu'elle nécessite, pour *chaque* $\widehat{x}_k$:

- N multiplications complexes donc $4N$ multiplications réelles et $2N$ additions réelles.
- $N-1$ additions complexes donc $2(N-1)$ additions réelles.

Au total, on a donc $4N^2 + 2N(N-1) = 6N^2 - 2N$ opérations à effectuer, soit une complexité de l'ordre de $O(N^2)$. Cette procédure, bien qu'évidente à mettre en place, possède néanmoins le gros inconvénient de demander des temps de calculs assez considérables.

6.2 Par une méthode récursive

On va maintenant utiliser le fait que la Transformée de Fourier d'un élement est cet élement. Ainsi, on va chercher à décomposer notre somme de N termes en deux sommes de $N/2$ termes, que l'on décomposera à leur tour en sommes de $N/4$ termes, jusqu'à ne plus avoir que des "sommes" de 1 élement. Cette méthode nécessite donc que N soit une puissance de 2. Cet algorithme est aussi connu comme *Algorithme de Cooley-Tuckey*. Sa compléxité est de $O(N \log_2(N))$. Cependant, les nombreux appels récursifs peuvent entraîner de lourdes charges sur la mémoire, ralentissant ainsi le calcul.

6.3 Par un calcul itératif

Il nous reste à décrire un dernier algorithme. Celui-ci se base sur une manière de sommer particulière, puisque l'on va utiliser les écritures binaires des indices. Comme cet algorithme est assez complexe à mettre en application, voyons plutôt un exemple sur une Transformée de Fourier à $N = 2^3$ élements.

6.3.1 Décomposition en base 2

On partira de la définition suivante, donnée en (5.3) :

$$\forall k \in [|0..N-1|], \quad \widehat{x_k} = X_k = \sum_{j=0}^{N-1} x_j e^{-i\frac{2\pi jk}{N}}$$

Posons $\omega = \mathrm{e}^{-i\frac{2\pi}{N}} = \mathrm{e}^{-i\frac{2\pi}{8}}$. On a alors : $\forall z \in \mathbb{Z} \quad \omega^{8z} = 1$, et l'expression de

la Transformée de Fourier devient :

$$\forall k \in [|0..N-1|], \quad X_k = \sum_{j=0}^{N-1} x_j \omega^{jk}$$

On décompose les indices k et j de la façon suivante :

$$\left\{ \begin{array}{ll} j & = j_2 \times 2^2 + j_1 \times 2^1 + j_0 \times 2^0 \\ k & = k_2 \times 2^2 + k_1 \times 2^1 + k_0 \times 2^0 \end{array} \right. \quad \forall i \in \{0,1,2\},\ j_i \in \{0,1\} \text{ et } k_i \in \{0,1\}$$

Ainsi, on obtient :

$$\begin{aligned} j \times k =& (j_2 \times 4 + j_1 \times 2 + j_0 \times 1) \times (k_2 \times 4 + k_1 \times 2 + k_0 \times 1) \\ =& 16 k_2 j_2 + 8(k_1 j_2 + k_2 j_1) \\ & + 4(k_0 j_2 + k_1 j_1 + k_2 j_0) + 2(k_0 j_1 + k_1 j_0) \\ & + k_0 j_0 \end{aligned}$$

De sorte que :

$$\omega^{jk} = \omega^{4k_0 j_2} \cdot \omega^{(4k_1+2k_0)j_1} \cdot \omega^{(4k_2+2k_1+k_0)j_0}$$

On obtient alors l'expression de X_k sous la forme :

$$X_k = X(4k_2+2k_1+k_0) = \sum_{j_0=0}^{1} [\sum_{j_1=0}^{1} [\sum_{j_2=0}^{1} x(4j_2 + 2j_1 + j_0)\omega^{4k_0 j_2}]\omega^{2j_1(2k_1+k_0)}]\omega^{j_0 k}$$

On va alors poser quelques notations intermédiaires :

$$\begin{aligned} \triangleright \quad x_1(4k_0 + 2j_1 + j_0) &= \sum_{j_2=0}^{1} x(4j_2 + 2j_1 + j_0)\omega^{4k_0 j_2} \\ &= x(2j_1 + j_0) + x(4 + 2j_1 + j_0)\omega^{4k_0} \\ \triangleright \quad x_2(4k_0 + 2k_1 + j_0) &= \sum_{j_1=0}^{1} x_1(4k_0 + 2j_1 + j_0)\omega^{2j_1(2k_1+k_0)} \\ &= x_1(4k_0 + j_0) + x_1(4k_0 + 2 + j_0)\omega^{2(2k_1+k_0)} \\ \triangleright \quad x_3(4k_0 + 2k_1 + k_2) &= \sum_{j_0=0}^{1} x_2(4k_0 + 2k_1 + j_0)\omega^{j_0 k} \end{aligned}$$

On aura alors :

$$X(4k_2 + 2k_1 + k_0) = x_3(4k_0 + 2k_1 + k_2)$$

Remarque : Cette méthode est aussi appellée *renversement des bits*[8] : cela se comprend par la dernière égalité : pour obtenir le signal X_k, il faut *renverser* l'ordre des bits de k dans son écriture binaire.

6.3.2 Détail d'un calcul

Explicitons les divers calculs des signaux x_1 :
On peut obtenir le tableau suivant :

$x_1(0)$	=	$x(0)+x(4)$	$x_2(0)$	=	$x_1(0)+x_1(2)$	$x_3(0)$	=	$x_2(0)+x_2(1)$
$x_1(1)$	=	$x(1)+x(5)$	$x_2(1)$	=	$x_1(1)+x_1(3)$	$x_3(1)$	=	$x_2(0)+\omega^4 x_2(1)$
$x_1(2)$	=	$x(2)+x(6)$	$x_2(2)$	=	$x_1(0)-x_1(2)$	$x_3(2)$	=	$x_2(2)+\omega^2 x_2(3)$
$x_1(3)$	=	$x(3)+x(7)$	$x_2(3)$	=	$x_1(1)-x_1(3)$	$x_3(3)$	=	$x_2(2)+\omega^6 x_2(3)$
$x_1(4)$	=	$x(0)-x(4)$	$x_2(4)$	=	$x_1(4)+\omega^2 x_1(6)$	$x_3(4)$	=	$x_2(4)+\omega x_2(5)$
$x_1(5)$	=	$x(1)-x(5)$	$x_2(5)$	=	$x_1(5)+\omega^2 x_1(7)$	$x_3(5)$	=	$x_2(4)+\omega^5 x_2(5)$
$x_1(6)$	=	$x(2)-x(6)$	$x_2(6)$	=	$x_1(4)+\omega^6 x_1(6)$	$x_3(6)$	=	$x_2(6)+\omega^3 x_2(7)$
$x_1(7)$	=	$x(3)-x(7)$	$x_2(7)$	=	$x_1(5)+\omega^6 x_1(7)$	$x_3(7)$	=	$x_2(6)+\omega^7 x_2(7)$

Ainsi, on remarque que le calcul du premier signal revient à effectuer quelques additions et soustractions, suivant un schéma précis. Pour les autres signaux, le principe reste le même, à une multiplication par une certaine puissance de ω près. C'est dans cette suite d'opérations que réside la puissance de l'algorithme.
En effet, cet algorithme est d'une complexité de l'ordre de $O(N\log_2(N))$. On remarque que sa complexité est identique à celle de l'algorithme récursif, mais à l'inverse de ce dernier, il ne nécessite pas un usage aussi important de mémoire, ce qui le rend plus performant.

[8] ou *bit reverseal* en anglais

7 Mise en application des algorithmes

Nous avons décrit trois algorithmes permettant de calculer une Transformée de Fourier Discrète, et évoqué leurs complexités respectives. Il est maintenant temps de vérifier *expérimentalement* nos propos.

7.1 Protocole

Pour cela, nous disposons des trois algorithmes, codés sous les trois langages suivants : *Maple, Octave, C++*. Les fichiers sources concernant les algorithmes *Maple* et *Octave* sont quasiment identiques ; ceux en *C++* comportent en plus la gestion des nombres complexes.
Pour tester ces algorithmes, nous allons faire générer des listes de 2^N élements de façon *pseudo-aléatoire*. Nous en calculerons les Transformées de Fourier Discrètes par les diverses méthodes, et mesurerons le temps nécessaire à leur exécution. Ce temps sera mesuré[9] à l'aide de l'instruction *time* sous *Maple*, de l'instruction *tic-toc* sous *Octave*, et de manière plus complexe (mais néanmoins plus précise) en *C++* : on comptera le nombre de cycles nécessaires au processeur pour effectuer ces calculs, et l'on se servira de la relation existant entre temps de calcul, cycle et fréquence du processeur. Ainsi, on peut mesurer des temps avec une précision atteignant 1/fréquence, soit de l'ordre de la nanoseconde ! Nous effectuerons plusieurs itérations des calculs, puis prendrons la valeur moyenne du temps nécessaire.
Le nombre d'élements ira de 2 à une limite fixée par la mémoire disponible sur la machine, ainsi que de la gestion de la mémoire et des ressources faite par le programme. C'est pourquoi certaines mesures ne pourront pas être effectuées (notamment, celles concernant l'algorithme *Naïf*).
La machine utilisée pour les tests sera la même, dotée d'un processeur AMD Athlon 3500+ et de 1Go de mémoire vive. Cette machine tourne sous Linux.

7.2 Résultat

On présentera ici les résultats sous forme de tableaux, et l'on pourra faire références à des schémas placés en annexes (figures 4 à 6).

[9] Il est possible d'évoquer l'altération due à l'introduction de la mesure : il y a influence de l'observateur sur l'observé, mais cela ne rentrera pas en compte à notre échelle.

7.2.1 Maple :

Temps mesuré en secondes			
Puissance de 2	Naïf	Récursif	Itératif
1	$4,92 \cdot 10^{-4}$	$2,00 \cdot 10^{-5}$	$5,20 \cdot 10^{-5}$
2	$1,22 \cdot 10^{-3}$	$2,88 \cdot 10^{-4}$	$9,60 \cdot 10^{-5}$
3	$1,92 \cdot 10^{-3}$	$2,72 \cdot 10^{-4}$	$2,44 \cdot 10^{-4}$
4	$6,24 \cdot 10^{-3}$	$1,16 \cdot 10^{-3}$	$5,32 \cdot 10^{-4}$
5	$5,48 \cdot 10^{-2}$	$1,41 \cdot 10^{-3}$	$1,23 \cdot 10^{-3}$
6	$1,66 \cdot 10^{-1}$	$3,36 \cdot 10^{-3}$	$2,92 \cdot 10^{-3}$
7	$9,54 \cdot 10^{-1}$	$2,08 \cdot 10^{-2}$	$6,84 \cdot 10^{-3}$
8	$5,45$	$4,08 \cdot 10^{-2}$	$2,14 \cdot 10^{-2}$
9	$25,9$	$7,04 \cdot 10^{-2}$	$3,76 \cdot 10^{-2}$
10	161	$1,07 \cdot 10^{-1}$	$9,28 \cdot 10^{-2}$
11	$1,03 \cdot 10^{3}$	$2,45 \cdot 10^{-1}$	$2,33 \cdot 10^{-1}$
12	.	$6,24 \cdot 10^{-1}$	$6,52 \cdot 10^{-1}$
13	.	$1,43$	$2,11$
14	.	$3,94$	$6,60$
15	.	$13,7$	$25,4$
16	.	$34,1$	106
17	.	133	487

Analyse : Étonnamment, à partir de 2^{11}, l'algorithme *récursif* se montre plus véloce que l'algorithme *itératif*, bien qu'ils possèdent tous deux une complexité similaire. Mais l'on remarque la nette domination des algorithmes *récursif* et *itératif* sur l'algorithme *naïf*, et cela dès la première valeur !

7.2.2 Octave

Temps mesuré en secondes			
Puissance de 2	Naïf	Récursif	Itératif
1	$1,58 \cdot 10^{-2}$	$2,25 \cdot 10^{-2}$	$2,09 \cdot 10^{-2}$
2	$2,64 \cdot 10^{-3}$	$2,89 \cdot 10^{-3}$	$2,59 \cdot 10^{-3}$
3	$8,39 \cdot 10^{-3}$	$6,21 \cdot 10^{-3}$	$3,60 \cdot 10^{-3}$
4	$3,10 \cdot 10^{-2}$	$1,44 \cdot 10^{-2}$	$7,93 \cdot 10^{-3}$
5	$1,21 \cdot 10^{-1}$	$3,34 \cdot 10^{-2}$	$1,78 \cdot 10^{-2}$
6	$4,84 \cdot 10^{-1}$	$7,26 \cdot 10^{-2}$	$3,99 \cdot 10^{-2}$
7	$8,90 \cdot 10^{-1}$	$1,60 \cdot 10^{-1}$	$9,01 \cdot 10^{-2}$
8	$3,51$	$1,73 \cdot 10^{-1}$	$2,00 \cdot 10^{-1}$
9	$13,6$	$3,43 \cdot 10^{-1}$	$4,49 \cdot 10^{-1}$
10	$53,2$	$7,34 \cdot 10^{-1}$	$6,43 \cdot 10^{-1}$
11	206	$2,01$	$1,33$
12	821	$3,35$	$2,40$
13	$3,29 \cdot 10^{3}$	$7,07$	$5,86$
14	.	$14,9$	$15,8$
15	.	$31,4$	$46,1$
16	.	$66,4$	150
17	.	139	530
18	.	294	$1,98 \cdot 10^{3}$
19	.	611	$7,56 \cdot 10^{3}$

Analyse : Ici, la plus grande légereté du programme a permis d'effectuer quelques mesures de plus. On peut cependant faire la même remarque que précedement, concernant les temps d'exécution des deux derniers algorithmes.

7.2.3 C++

Temps mesuré en secondes			
Puissance de 2	Naïf	Récursif	Itératif
1	$1,45 \cdot 10^{-6}$	$2,18 \cdot 10^{-6}$	$1,21 \cdot 10^{-6}$
2	$7,93 \cdot 10^{-6}$	$6,85 \cdot 10^{-6}$	$2,62 \cdot 10^{-6}$
3	$2,46 \cdot 10^{-5}$	$1,80 \cdot 10^{-5}$	$5,51 \cdot 10^{-6}$
4	$1,09 \cdot 10^{-4}$	$5,55 \cdot 10^{-5}$	$1,37 \cdot 10^{-5}$
5	$4,11 \cdot 10^{-4}$	$1,08 \cdot 10^{-4}$	$3,77 \cdot 10^{-5}$
6	$1,44 \cdot 10^{-3}$	$2,42 \cdot 10^{-4}$	$7,03 \cdot 10^{-5}$
7	$3,29 \cdot 10^{-3}$	$4,92 \cdot 10^{-4}$	$1,44 \cdot 10^{-4}$
8	$1,26 \cdot 10^{-2}$	$8,12 \cdot 10^{-4}$	$3,44 \cdot 10^{-4}$
9	$4,87 \cdot 10^{-2}$	$1,64 \cdot 10^{-3}$	$6,48 \cdot 10^{-4}$
10	$1,94 \cdot 10^{-1}$	$2,88 \cdot 10^{-3}$	$1,17 \cdot 10^{-3}$
11	$7,83 \cdot 10^{-1}$	$5,58 \cdot 10^{-3}$	$2,20 \cdot 10^{-3}$
12	$3,11$	$1,21 \cdot 10^{-2}$	$3,63 \cdot 10^{-3}$
13	$12,6$	$2,48 \cdot 10^{-2}$	$7,72 \cdot 10^{-3}$
14	$51,2$	$5,10 \cdot 10^{-2}$	$1,78 \cdot 10^{-2}$
15	203	$1,11 \cdot 10^{-1}$	$3,68 \cdot 10^{-2}$
16	813	$2,82 \cdot 10^{-1}$	$7,82 \cdot 10^{-2}$
17	$3,27 \cdot 10^{3}$	$5,57 \cdot 10^{-1}$	$2,16 \cdot 10^{-1}$
18	.	$1,02$	$4,79 \cdot 10^{-1}$
19	.	$2,16$	$1,04$
20	.	$4,53$	$2,23$
21	.	$9,50$	$4,62$
22	.	$19,9$	$9,65$
23	.	$41,7$	$20,5$
24	.	$89,1$	$43,9$
25	.	233	$93,4$

Analyse : On note tout de suite que le langage *C++* permet d'effectuer des mesures sur un plus grand intervalle. Mais surtout, c'est avec ce langage que l'expérience rejoint la théorie : l'algorithme *itératif* s'exécute bien plus rapidement que l'algorithme *récursif* !

7.2.4 Causes plausibles de la divergence théorie/pratique

Comme nous avons pu le constater, les résultats obtenus pour les algorithmes *récursif* et *itératif*, qu'ils soient programmés en *Maple* ou en *Octave* ne concordent pas avec nos prévisions. En effet, nous avons démontré que

ces deux algorithmes possèdent la même complexité en temps, mais que l'algorithme récursif, de part sa plus grande utilisation de la mémoire, semblait devoir s'incliner face à l'algorithme itératif. Or, cela n'est pas le cas, sauf pour le programme en *C++*. Sans entrer dans des détails complexes et techniques, plus relatifs à l'informatique qu'à notre sujet, tentons de donner quelques idées expliquant cette différence.
Il faut tout d'abord savoir que les langages *Maple*, *Octave* et *C++* diffèrent sur un point : leur mode d'exécution. Alors que le *C++* doit être compilé, les langages *Maple* et *Octave* sont interprétés. D'après *Wikipédia* :

> "Un programme écrit en langage interprété est converti en instructions directement exécutables par la machine au moment de son exécution. Au contraire, un programme écrit en langage compilé est traduit en instructions lisibles par la machine une fois pour toutes."

Il est possible que cette interprétation soit une cause de la "lenteur" des programmes *Maple* et *Octave*, car l'on remarque aussi que les temps mesurés sont considérablement plus court en *C++* que dans les autres langages.

7.2.5 Interprétations graphiques

En annexes se trouvent les exploitations graphiques de ces données. Les courbes sont tracées de la façon suivante : en abscisses se trouve le nombre d'éléments, la graduation se faisant en fonction de la puissance de 2 choisie ; en ordonnées le temps mesuré, exprimé en secondes, sur une échelle du type *logarithmique*.
Ces courbes permettent de constater les différences entre les algorithmes, mais qu'en est-il par rapport aux prévisions théoriques ? Pour étudier cela, nous avons pris les résultats obtenus pour le programme en *C++*, car plus proches de la théorie. Ensuite, nous avons tracé les courbes représentant la complexité en fonction du nombre d'éléments. Pour pouvoir faire une comparaison, nous avons effectué une *mise à l'echelle* : les valeurs obtenues pour le plus grand nombre d'éléments seront posées comme références, et l'on étalonne nos courbes "théoriques" par rapport à ces valeurs. Le résultat est donné sur la figure 7. Nous pouvons donc constater que les estimations théoriques annoncées sont proches des résultats obtenus.

8 Multiplication de polynômes

Nous avons définit et étudié diverses méthodes de calcul de cette transformation, sans toutefois en évoquer les aspects pratiques. Il existe de nombreuses applications pour la Transformée de Fourier, notement en Traitement de l'information : analyse spectrale, effet d'un filtre, ... Cependant, nous allons en voir une utilisation mathématique, dans un domaine qui au premier abord semble n'avoir aucun lien avec une telle transformation : la multiplication. Plus précisement, nous étudierons ici la multiplication des polynômes[10].

8.1 Notations

Soient P et Q, deux polynômes de degré $n-1$, dont nous souhaitons calculer le produit. Posons $R = PQ$, on a donc $deg(R) = 2n-2$.

Notons :

$$P = \sum_{i=0}^{n-1} p_i X^i \text{ et } Q = \sum_{i=0}^{n-1} q_i X^i$$
$$\text{et } R = \sum_{i=0}^{2n-1} r_i X^i \text{ avec } r_{2n-1} = 0$$

Tout l'intêret de cette méthode consiste à calculer *astucieusement* les coefficients de R à l'aide d'un Transformée de Fourier. On va poser

$$\begin{cases} p = (p_0, p_1, \ldots, p_{n-1}, \underbrace{0, \ldots, 0}_{n \text{ termes}}) \\ q = (q_0, q_1, \ldots, q_{n-1}, \underbrace{0, \ldots, 0}_{n \text{ termes}}) \end{cases}$$

On cherche à se ramener à une Transformée de Fourier à $N = 2n$ élements[11]. On pose $\omega = \mathrm{e}^{-i\frac{2\pi}{N}}$, racine primitive *N-ième* de l'unité.

8.2 Calcul des $\widehat{p_i}$, $\widehat{q_i}$

D'après la formule (5.3), on a :

$$\triangleright \quad \forall k \in [|0 \ldots 2n-1|], \quad \widehat{p}_k = \sum_{j=0}^{2n-1} p_j \omega^{jk} = P(\omega^k)$$

[10] Pour l'anecdote : Cette méthode peut aussi s'appliquer pour la multiplication de grands entiers.

[11] Utile, car si $n = 2^r$, alors $N = 2^{r+1}$

De même, pour Q et R :

$$\triangleright \quad \forall k \in [|0 \dots 2n-1|], \quad \widehat{q}_k = \sum_{j=0}^{2n-1} q_j \omega^{jk} = Q(\omega^k)$$

$$\triangleright \quad \forall k \in [|0 \dots 2n-1|], \quad \widehat{r}_k = \sum_{j=0}^{2n-1} r_j \omega^{jk} = R(\omega^k)$$

Il vient :

$$\forall k \in [|0 \dots 2n-1|], \quad \widehat{r}_k = (P \times Q)(\omega^k) = P(\omega^k) \times Q(\omega^k) = \widehat{p}_k \times \widehat{q}_k$$

On retrouve ensuite les coefficients de R par application de la Transformée Inverse.

8.3 Utilité

Une question tout à fait légitime vient à l'esprit : pourquoi utiliser une telle méthode ? La réponse se trouve dans la complexité des calculs à effectuer. En effet, nous savons qu'évaluer le produit de deux polynômes de degré n est de l'ordre de $O(n^2)$ Ici, il nous faut, en utilisant la transformée de Fourier *itérative* :

- Calculer les valeurs de P et de Q en les racines $N = 2n$-ièmes de l'unité ;
- Calculer deux transformées de Fourier à N élements ;
- Multiplier les N coefficients obtenus entre eux ;
- Calculer une transformée inverse de Fourier à N élements.

Si N est une puissance de 2, (ce que l'on peut toujours obtenir, en complétant la liste des coefficients par des *zéros* [12]), alors il est possible d'utiliser la transformation itérative, par exemple. Alors, les deux première stransformations demandent $O(N \log_2(N))$ opérations, le produit N, la tranformée inverse $O(N \log_2(N))$, soit un total de l'ordre de $O(N \log_2(N))$[13], ce qui est bien plus avantageux !

[12] C'est le *zero padding*

[13] C'est $N + 3 \times O(N \log_2(N))$, qui est bien un $O(N \log_2(N))$

8.4 Résultat

On se contentera ici de vérifier que la multiplication par *Transformée de Fourier Discrète* donne les mêmes résultats qu'une multiplication ordinaire. Posons :

$$P = \sum_{i=0}^{7} iX^i \text{ donc } deg(P) = 7$$
$$Q = \sum_{i=0}^{7} i^i X^i \text{ donc } deg(Q) = 7$$
$$\text{et } R = P \times Q \text{ donc } deg(R) = 14$$

Alors, d'après *Maple* : $R = X + 3\,X^2 + 9\,X^3 + 42\,X^4 + 331\,X^5 + 3745\,X^6 + 53815\,X^7 + 927420\,X^8 + 1801024\,X^9 + 2674603\,X^{10} + 3547994\,X^{11} + 4419526\,X^{12} + 5267850\,X^{13} + 5764801\,X^{14}$.
Par notre algorithme : $R := [0, 1, 3, 9, 42, 331, 3745, 53815, 927420, 1801024, 2674603, 3547994, 4419526, 5267850, 5764801, 0]$.
En prenant en compte le fait que R est un polynôme de "degré 15 dont le coefficient dominant est nul"[14], nous pouvons constater que les coefficients obtenus par ces deux méthodes concordent bien.

9 Conclusion

La *Transformée de Fourier Discrète* présente l'avantage de posséder par rapport à son homologue *continue* une expression plus légère, et donc plus pratique à mettre en application, en particulier dans les problèmes physiques.

Diverses méthodes pour son calcul existent ; nous en avons étudiées trois, et vu leur efficacité et rapidité respectives. Ces tests nous ont permis de montrer une divergence théorie/pratique, et d'apporter des éléments d'explications quant aux causes de ces divergences.

Finalement, nous avons vu que cette transformation peut être mise en application aussi bien dans le domaine physique (analyse du son, filtres électroniques, ...) que dans le domaine mathématique, avec l'exemple traité de la multiplication des polynômes, où elle permet une diminution conséquente du temps de calcul.

[14] C'est donc bien un polynôme de degré 14.

A Propriétés de la suite x_n

Montrons :
$\forall n \in \mathbb{N} - \{0\}$:

$$\int_{-\infty}^{\infty} x_n(t)\mathrm{dt} = 1 \tag{A.1}$$

$$\forall f \in \mathcal{L}(\mathbb{R}), \text{si } f \text{ est continue en 0 alors } \lim_{n\to\infty} \int_{-\infty}^{\infty} x_n(t) \times f(t)\mathrm{dt} = f(0) \tag{A.2}$$

Pour (A.1) :

$$\int_{-\infty}^{\infty} x_n(t)\mathrm{dt} = \int_{\frac{1}{2n}}^{\frac{1}{2n}} n\mathrm{dt} = n \times \frac{1}{n} = 1.$$

Pour (A.2) :

$$f \text{ continue en } 0 \iff \quad \forall \varepsilon > 0,\ \exists \alpha > 0,\ \forall x \in]-\alpha, \alpha[,\ |f(x) - f(0)| < \varepsilon$$

Soit $\varepsilon > 0$, on pose α le rang correspondant. On a alors :

$$\forall n\ \frac{1}{\alpha},\ [-\frac{1}{2n}, \frac{1}{2n}] \subset]-\infty, \infty[,\ \text{donc } \forall x \in [-\frac{1}{2n}, \frac{1}{2n}],\ |f(x) - f(0)| < \varepsilon.$$

$$\text{D'où } |\int_{-\infty}^{\infty} x_n(t)f(t)\mathrm{dt} - f(0)| = |n\int_{-\frac{1}{2n}}^{\frac{1}{2n}} f(t) - f(0)\mathrm{dt}| < n\int_{\frac{1}{2n}}^{\frac{1}{2n}} \varepsilon\ \mathrm{dt}\ = \varepsilon.$$

$$\text{Alors : } |\int_{-\infty}^{\infty} x_n(t)f(t)\mathrm{dt} - f(0)| < \varepsilon$$

$$\text{D'où l'on déduit : } \lim |\int_{-\infty}^{\infty} x_n(t)f(t)\mathrm{dt} - f(0)| = 0$$

et donc la formule suivante :

$$\lim_{n\to\infty} \int_{-\infty}^{\infty} x_n(t)f(t)\mathrm{dt} = f(0) \quad \square$$

B Calcul de la Transformée Inverse

Préliminaires

En premier lieu, nous allons démontrer la formule suivante :

$$S_N = \sum_{k=0}^{N-1} \mathrm{e}^{\frac{i2\pi n\alpha}{N}} = N\delta_{\alpha,0} \tag{B.1}$$

Preuve :

- $\alpha = 0$ ou $\alpha \in N\mathbb{Z}$

$$S_N = \sum_{k=0}^{N-1} \mathrm{e}^{\frac{i2\pi k}{N}\times 0} = \sum_{k=0}^{N-1} \mathrm{e}^{0} = \sum_{k=0}^{N-1} 1 = N$$

- $\alpha \neq 0$, avec la condition $\alpha \notin N\mathbb{Z}$

$$S_N = \sum_{k=0}^{N-1} \mathrm{e}^{(\frac{i2\pi\alpha}{N})^k} = \frac{(\mathrm{e}^{\frac{i2\pi\alpha}{N}})^N - 1}{\mathrm{e}^{\frac{i2\pi\alpha}{N}} - 1}$$

Nous avons imposé la condition : $\alpha \notin N\mathbb{Z}$. Cependant, nous verrons qu'elle est toujours réalisée. Il vient alors :

$$S_N = \frac{\mathrm{e}^{i2\pi\alpha} - 1}{\mathrm{e}^{\frac{i2\pi\alpha}{N}} - 1} = \frac{1-1}{\mathrm{e}^{\frac{i2\pi\alpha}{N}} - 1} = 0.$$

Preuve de la formule d'inversion

Passons maintenant à la formule d'inversion :

$$\forall n \in [|1..N|] : \quad x_n = \frac{1}{N} \sum_{k=0}^{N-1} X_k \mathrm{e}^{i\frac{2\pi kn}{N}} \tag{B.2}$$

Preuve $\forall k \in [|1..N|]$:

$$\begin{aligned} A_N = \sum_{k=0}^{N-1} X_k \mathrm{e}^{-i\frac{2\pi nk}{N}} &= \sum_{k=0}^{N-1} [\sum_{j=0}^{N-1} x_j \mathrm{e}^{i\frac{2\pi kj}{N}}] \mathrm{e}^{-i\frac{2\pi kn}{N}} \\ &= \sum_{k=0}^{N-1} \sum_{j=0}^{N-1} x_j \mathrm{e}^{i\frac{2\pi k(j-n)}{N}} \\ &= \sum_{j=0}^{N-1} x_j \sum_{k=0}^{N-1} \mathrm{e}^{i\frac{2\pi k(j-n)}{N}} \\ &= \sum_{j=0}^{N-1} x_j \sum_{k=0}^{N-1} \mathrm{e}^{i\frac{2\pi k(j-n)}{N}} \end{aligned}$$

Ainsi, on retrouve l'expression de S_N, avec $\alpha = (j-n)$. Montrons que les hypothèses sont bien vérifiées :

$$\begin{gathered} 0 \leqslant j \leqslant N-1 \\ 0 \leqslant n \leqslant N-1 \Rightarrow 1-N \leqslant -n \leqslant 0 \end{gathered}$$

Il vient donc : $1-N \leqslant j-n \leqslant N-1 \Leftrightarrow |j-n| \leqslant N-1$:
$j-n$ ne peut donc être un multiple non-nul de N. D'où :

$$\begin{aligned} A_N &= \sum_{j=1}^{N} x_j \times N \times \delta_{j-n,0} \\ &= N \times x_n \quad \square \end{aligned}$$

C Formule Récursive

Pour calculer la Transformé de Fourier Discrète, on peut utiliser l'algorithme récursif : on va décomposer la somme par séparation des termes pairs et impairs.

Preuve :

1. Calcul de la Transformée de Fourier d'un élement
   En utilisant la définition :

$$X_k = \sum_{m=1}^{N} x_m \mathrm{e}^{-i2\frac{\pi(m-1)(k-1)}{N}} \text{ , comme } k = 1, N = 1 :$$

$$X = \sum_{m=1}^{1} x_m \mathrm{e}^{-i2\pi(m-1)\times 0} = x_1 = x$$

2. Décomposition des termes de la somme
   En supposant que N est une puissance de 2 supérieure à 1 (dans le cas contraire, c'est 1, et l'on est ramené au cas précédent), on pose $N = 2P$
   On va décomposer les termes de la manière suivante : $\forall k \in [|1..N|]$

$$\begin{aligned}
X_k &= \sum_{m=1}^{N} x_m \mathrm{e}^{-i\frac{2\pi(m-1)(k-1)}{N}} \\
&= \sum_{m=1}^{P} (x_{2m} \mathrm{e}^{-i\frac{2\pi(2m-1)(k-1)}{N}} + x_{2m-1} \mathrm{e}^{-i\frac{2\pi(2m-2)(k-1)}{N}}) \\
&= \mathrm{e}^{-i\frac{2\pi(k-1)}{N}} \sum_{m=1}^{P} x_{2m} \mathrm{e}^{-i\frac{2\pi(2m-2)(k-1)}{2P}} + \sum_{m=1}^{P} x_{2m-1} \mathrm{e}^{-i\frac{2\pi(2m-2)(k-1)}{2P}} \\
&= \mathrm{e}^{-i\frac{2\pi(k-1)}{N}} \sum_{m=1}^{P} x_{2m} \mathrm{e}^{-i\frac{2\pi(m-1)(k-1)}{P}} + \sum_{m=1}^{P} x_{2m-1} \mathrm{e}^{-i\frac{2\pi(m-1)(k-1)}{P}}
\end{aligned}$$

   On voit alors apparaître les k-ièmes termes de la Transformée de Fourier de rang P, la première étant effectuée sur les termes pairs, la secondes sur les termes impairs.
   De plus, on peut voir que :

$$X_{k+P} = \mathrm{e}^{-i\frac{2\pi(k+P-1)}{N}} \sum_{m=1}^{P} x_{2m} \mathrm{e}^{-i\frac{2\pi(m-1)(k+P-1)}{P}} + \sum_{m=1}^{P} x_{2m-1} \mathrm{e}^{-i\frac{2\pi(m-1)(k+P-1)}{P}}$$

$$\begin{aligned} \text{Or : } \mathrm{e}^{-i\frac{2\pi(k+P-1)}{N}} &= \mathrm{e}^{-i\frac{2\pi(k-1)}{N}} \mathrm{e}^{-i\frac{2\pi P}{2P}} \\ &= -\mathrm{e}^{-i\frac{2\pi(k-1)}{N}} \\ \text{et : } \mathrm{e}^{-i\frac{2\pi(m-1)(k+P-1)}{P}} &= \mathrm{e}^{-i\frac{2\pi(m-1)(k-1)}{P}} \mathrm{e}^{-i\frac{2\pi P}{P}} \\ &= \mathrm{e}^{-i\frac{2\pi(m-1)(k-1)}{P}} \end{aligned}$$

$$\text{Ainsi : } X_{k+P} = -\mathrm{e}^{-i\frac{2\pi(k-1)}{N}} \sum_{m=1}^{P} x_{2m} \mathrm{e}^{-i\frac{2\pi(m-1)(k-1)}{P}} + \sum_{m=1}^{P} x_{2m-1} \mathrm{e}^{-i\frac{2\pi(m-1)(k-1)}{P}}$$

Conclusion

Ainsi, pour calculer la Transformée de Fourier de N=2P points, il suffit de calculer celle de P points sur les termes de rang pair, et sur les termes de rang impairs. Ces deux sommes se calculent selon la même méthode, si l'on suppose P pair : on se ramène donc à une formule récursive.

Complexité

Nous allons montrer que l'utilisation de cet algorithme réduit la compléxité du calcul à $O(N \log_2(N))$.

Preuve : Soit $M(N)$ (respectivement $A(N)$) le nombre de multiplications (d'additions) à effectuer pour calculer la Transformée de Fourier d'un signal de N points. Le calcul utilise la Transformée de Fourier de $N/2$ points, ainsi

que $N/2$ multiplications *complexes* et N additions *complexes*, soit $2N$ multiplications *réelles* et $3N$ additions *réelles*[15] On a alors les égalités suivantes :

$$M(N) = 2M(N/2) + 2N$$
$$A(N) = 2A(N/2) + 3N$$

Posons les variables réduites $\bar{M}(N) = M(N)/N$ et $\bar{A}(N) = A(N)/N$, et effectuons le changement de variable[16] $r = \log_2(N)$. Le système devient :

$$\begin{cases} \bar{M}(2^r) = 2^{1-r}M(2^{r-1}) + 2 \\ \bar{M}(2^r) = 2^{1-r}A(2^{r-1}) + 3 \end{cases} \iff \begin{cases} \bar{M}(r) = \bar{M}(r-1) + 2 \\ \bar{A}(r) = \bar{A}(r-1) + 3 \end{cases}$$

Remarquons maintenant, d'après le premier élement de preuve, que la Transformée de Fourier d'un élément ne nécessite ni multiplication, ni addition (c'est l'identité). Puisque $1 = 2^0$, on a donc :

$$\begin{cases} \bar{M}(0) = 0 \\ \bar{A}(0) = 0 \end{cases}$$

Il vient, par addition membre à membre des égalités :

$$\begin{cases} \bar{M}(r) = 2r \\ \bar{A}(r) = 3r \end{cases} \iff \begin{cases} M(N) = 2N\log_2(N) \\ A(N) = 3N\log_2(N) \end{cases}$$

Au total, on a une complexité en $5N\log_2(N)$, soit en $O(N\log_2(N))$ $\square$

[15] En effet, une addition complexe compte pour deux additions réelles, et une multiplication complexe pour 4 multiplications réelles et deux additions réelles.

[16] On a $N = 2^r$

D Algorithme de calcul itératif - Cas 2^r

On pose $N = 2^r$, et $\omega = \mathrm{e}^{i\frac{2\pi}{N}}$.
On a alors : $\forall z \in \mathbb{Z} \quad \omega^{2^r z} = 1$

On décompose les indices de la même manière :

$$\left\{ \begin{array}{ll} j & = j_{r-1} \times 2^{r-1} + \cdots + j_1 \times 2^1 + j_0 \\ k & = k_{r-1} \times 2^{r-1} + \cdots + k_1 \times 2^1 + k_0 \end{array} \right. \quad \forall i \in \{0 \cdots r-1\},\ (j_i, k_i) \in \{0,1\} \times \{0,1\}$$

Alors :

$$X(2^{r-1}k_{r-1} + \cdots + 2k_1 + k_0) = \sum_{j_0=0}^{1} \cdots \sum_{j_{r-1}=0}^{1} x(2^{r-1}j_{r-1} + \cdots + 2j_1 + j_0)\omega^{jk}$$

et : $\omega^{jk} = \prod_{m=1}^{r} \omega^{2^{r-m} j_{r-m}(k_0 + 2k_1 + \cdots + 2^{m-1}k_{m-1})}$ Posons les notations intermédiaires :

$$\triangleright \quad x_1(\underbrace{2^{r-1}k_0 + 2^{r-2}j_{r-2} + \cdots + 2j_1 + j_0}_{=a_1(j_{r-2})}) = \sum_{j_{r-1}=0}^{1} x(j)\omega^{2^{r-1}j_{r-1}k_0}$$

$$\triangleright \quad x_2(\underbrace{2^{r-1}k_0 + 2^{r-2}k_1 + 2^{r-3}j_{r-3} + \cdots + 2j_1 + j_0}_{=a_2(j_{r-3})}) = \sum_{j_{r-2}=0}^{1} x_1(a_1)\omega^{2^{r-2}j_{r-2}(2k_1+k_0)}$$

$$\vdots$$

$$\triangleright \quad x_m(\underbrace{2^{r-1}k_0 + \cdots + 2^{r-m}k_{m-1} + 2^{r-(m+1)}j_{r-(m+1)} + \cdots + j_0}_{=a_m(j_{r-(m+1)})})$$

$$= \sum_{j_{r-m}=0}^{1} x_{m-1}(a_{m-1})\omega^{2^{r-m}j_{r-m}(k_0 + 2k_1 + \cdots + 2^{m-1}k_{m-1})}$$

$$\vdots$$

$$\triangleright \quad x_r(2^{r-1}k_0 + 2^{r-2}k_1 + \cdots + k_{r-1}) = \sum_{j_0=0}^{1} x_{r-1}(a_{r-1}(j_0))\omega^{kj_0}$$

On peut alors exprimer le $m^{\text{ième}}$ signal de la façon suivante :

$$x_m(a_m(j_{r-(m+1)})) = x_{m-1}(2^{r-1}k_0 + \cdots + 2^{r-(m-2)}k_{m-2} + 2^{r-(m-1)} \times 0 + 2^{r-m}j_{r-m} + \cdots + j_0)$$
$$+\omega^{2^{r-m}(k_0 + \cdots + 2^{m-1}k_{m-1})} x_{m-1}(2^{r-1}k_0 + \cdots + 2^{r-m+2}k_{m-2} + 2^{r-m+1} \cdot 1 + 2^{r-m}j_{r-m} + \cdots + j_0)$$

Au final : $X_k = X(2^{r-1}k_{r-1} + \cdots + 2k_1 + k_0) = x_r(2^{r-1}k_0 + \cdots + k_{r-1})$.

Complexité

Montrons que la complexité de cet algorithme est de l'ordre de $O(N \log_2(N))$:

Preuve : Pour chaque signal intermédiaire, on doit effectuer une addition et une multiplication *complexes*. On a donc :

- 4 additions réelles ;
- 4 multiplications réelles ;

Il faut calculer N valeurs pour chaque signal intermédiaire.
Comme l'on dispose de $\log_2(N)$ signaux (déduit de la décomposition en base 2), on doit effectuer en tout : $8N \log_2(N)$ opérations, ce qui justifie bien une complexité en $O(N \log_2(N))$ $\square$

E Algorithmes - Maple

```
> restart:
```

Calcul de la TFD par une procédure "naïve"

Cette procédure calcule les coefficients de Fourier par une méthode dite "naive".

```
> TFDSimple:=proc(x)

   local n,X:
   n:=nops(x):

   X:=[seq(sum(x[j]*exp(-2*I*(j-1)*(k-1)*Pi/n),j=1..n),k=1..n)];

   end proc:
```

Calcul de la TFD par une procédure récursive

Cette procédure prend en entrée une liste représentant les coefficients d'un polynôme dont on cherche à calculer les coefficients de Fourier.

```
> TFDRecur:=proc(x)

   local N,CoefFFT,xp,xi,u,v,omega:
   N:=nops(x):

   if N=1 then CoefFFT:=x:
   else

   xp:=[seq(x[2*i],i=1..N/2)]:
   xi:=[seq(x[2*i-1],i=1..N/2)]:
   u:=TFDRecur(xp):
   v:=TFDRecur(xi):

   omega:=exp(-2*I*Pi/N):

   CoefFFT:=[seq(omega^(k-1)*u[k]+v[k],k=1..N/2),seq(-omega^(k-1)*u
   [k]+v[k],k=1..N/2)];
   end if:
   CoefFFT;

   end proc:
```

Calcul de la TFD par itération

On va utiliser l'écriture des divers coefficients en binaire.

```
> TFDIter:=proc(x)
   local n,i,j,k,a,b,p,l,z,w,h,m,tmp:
   n:=nops(x):
   l:=Array(1..n):
   l:=Array(x):
   p:=n/2:#p: puissance de 2 dans décomposition.

   while p>=1 do
      z:=1: #première valeur de omega.
      w:=exp(-I*Pi/p): #c'est omega.

      for h from 1 to p do #Variable servant à la décomposition
         for m from 1 to n/(2*p) do
      #On va calculer le signal xm
            a:=h+2*(m-1)*p:  #Première valeur du signal x_(m-1)
            b:=a+p:          #Seconde valeur du signal x_(m-1)
            tmp:=(l[a]-l[b])*z:
            l[a]:=l[a]+l[b]: #x_m(a) = x_(m-1)(a)+x_(m-1)(b)
            l[b]:=tmp:       #x_m(b) = (x_(m-1)(a)-x_(m-1)(b))*z
         end do:
         #On passe au signal m+1 -> w<-w^(m+1)
         z:=z*w:
      end do:
      p:=p/2:
   end do:

   #On a maintenant notre liste contenant les signaux x_r
   #Il reste à remettre les signaux dans le bon ordre.
   j:=1:
   for i from 1 to n do
      if j>i then tmp:=l[j]:
         l[j]:=l[i]:
         l[i]:=tmp:
      end if:
      p:=n/2:
      while p>=2 and j>p do
         j:=j-p:
         p:=p/2:
      end do:
      j:=j+p:
   end do:
   l;
   end proc:
```

Calcul de la Transformée Inverse.

De même, il est possible d'effectuer deux méthodes pour calculer la transformée inverse de Fourier :

La méthode naïve

En utilisant le même algorithme, on obtient :

```
> TFDISimple:=proc(X)

  local n,F:
  n:=nops(X):

  F:=[seq((1/n)*sum(X[i]*exp(2*I*(j-1)*(i-1)*Pi/(n)),i=1..n),j=
  1..n)];

  end proc:
```

La méthode récursive

Attention : Pour que l'on puisse retrouver les valeurs initiales, il ne faut pas oublier de diviser par le nombre de valeurs.

```
> TFDIRecur:=proc(X)
  local Coef,N,Xi,Xp,U,V,Omega,k;
  N:=nops(X);

  if N=1 then Coef:=X:
  else
     Xi:=[seq(X[2*k-1],k=1..N/2)];
     Xp:=[seq(X[2*k],k=1..N/2)];
     U:=TFDIRecur(Xi);
     V:=TFDIRecur(Xp);
     Omega:=exp(2*I*Pi/N);
     Coef:=[seq(U[k]+Omega^(k-1)*V[k],k=1..N/2),seq(U[k]-Omega^
  (k-1)*V[k],k=1..N/2)]:
   end if;
  Coef;
  end proc:
```

La méthode itérative

```
> TFDIIter:=proc(x)
  local n,i,j,k,a,b,p,l,z,w,h,m,tmp:
  n:=nops(x):
  l:=Array(1..n):
  l:=Array(x):
  p:=n/2:

  while p>=1 do
     z:=1:
     w:=exp(I*Pi/p); #C'est la différence

     for h from 1 to p do
        for m from 1 to n/(2*p) do
           a:=h+2*(m-1)*p:
           b:=a+p:
           tmp:=(l[a]-l[b])*z:
           l[a]:=l[a]+l[b]:
           l[b]:=tmp:
        end do:
        z:=z*w:
     end do:
     p:=p/2:
  end do:

  j:=1:
  for i from 1 to n do
     if j>i then tmp:=l[j]:
        l[j]:=l[i]:
        l[i]:=tmp:
     end if:
     p:=n/2:
     while p>=2 and j>p do
        j:=j-p:
        p:=p/2:
     end do:
     j:=j+p:
  end do:
  for i from 1 to n do
     l[i]:=l[i]/n:
  end do:
  return l;
  end proc:
```

Mesure du temps de calcul

On va ici s'intéresser à la mesure du temps nécessaire pour calculer les coefficients de Fourier (Transformée Directe) pour n "grand", par exemple n=2^5, n=2^10,... .
Pour cela, on va définir la liste de nos coefficients par une méthode "pseudo-aléatoire" :

```
> with(RandomTools[MersenneTwister]):
  A:=[seq(GenerateFloat(),i=1..2^8)]:
```

Méthode naïve

On effectue le calcul pour la méthode naïve :

```
> t:=time():
  TFDSimple(A):
  Temps := time()-t;
```

$$Temps := 6.472 \qquad (5.1.1)$$

Méthode récursive

Calcul pour la méthode récursive :

```
> t:=time():
  TFDRecur(A):
  Temps := time()-t;
```

$$Temps := 0.076 \qquad (5.2.1)$$

Méthode itérative

Mesure pour la méthode itérative :

```
> t:=time():
  TFDIter(A):
  Temps:= time()-t;
```

$$Temps := 0.040 \qquad (5.3.1)$$

Multiplication de polynômes

Algorithme

On va prendre en entrée deux listes contenant les coefficients des deux polynômes dont on cherche à calculer le produit.
Il faut prendre des précautions, car la procédure ne vérifie pas si les deux polynômes sont de même degré, qui doit être une puissance de 2, si l'on souhaite utiliser la méthode récursive ou itérative.

```
> Multiplication:=proc(P,Q)

  local n,N,R,A,B,i,j,k:

  n:=nops(P):
  N:=2*n:

  #On crée la liste des coefficients étendus à 2n éléments.
  A:=[seq(P[k],k=1..n),seq(0,k=n+1..N)];
  B:=[seq(Q[k],k=1..n),seq(0,k=n+1..N)];

  #On calcule la TFD de chacune de ces listes.
  A:=TFDIter(A):
  B:=TFDIter(B):

  #On effectue les produits
  R:=[seq(A[k]*B[k],k=1..N)]:

  #On récupère les coefficients.
  TFDIIter(R);

  end proc:
```

Exemple

On va poser $P := \sum_{i=0}^{7} iX^i$ et $Q := \sum_{i=0}^{7} i^i X^i$ donc $n = 8 = 2^3$.

Algo

On va demander la liste des coefficients du polynôme R=P*Q. Pour une lecture plus facile, nous en prendrons les valeurs arrondies.

```
> P:=[seq(i,i=0..7)]:
> Q:=[seq(i^i,i=0..7)]:
> R:=evalf(Multiplication(P,Q)):
> [seq(R[j],j=1..2*nops(P))];
```

$$[0.00091 + 0.002\,i,\ 1.0024 + 0.0020\,i,\ 3.0045 + 0.005\,i,\ 9.0008 + 0.00221\,i,\ 41.999 + 0.003031250000\,i,\ 330.999 + 0.002975000000\,i,\ 3745.000 + 0.003750000000\,i,\ 53815.00035 - 0.001348605397\,i,\ 927420.0013 - 0.002\,i,\ 1.801024000\,10^6 - 0.0030\,i,\ 2.674603000\,10^6 - 0.005\,i,\ 3.547993997\,10^6 - 0.00165\,i,\ 4.419525997\,10^6 - 0.003031250000\,i,\ 5.267849999\,10^6 - 0.001775000000\,i,\ 5.764800996\,10^6 - 0.003750000000\,i,\ 0.00205 - 0.0000986053972\,i] \quad (6.2.1.1)$$

```
> R:=[seq(round(R[j]),j=1..2*nops(P))];
```

$$R := [0, 1, 3, 9, 42, 331, 3745, 53815, 927420, 1801024, 2674603, 3547994, 4419526, 5267850, 5764801, 0] \quad (6.2.1.2)$$

Maple

On va evaluer le polynôme P*Q :

```
> P:=sum(i*X^i,i=0..7):
> Q:=sum(i^i*X^i,i=0..7):
> sort(expand(P*Q),X,ascending);
```

$$X + 3X^2 + 9X^3 + 42X^4 + 331X^5 + 3745X^6 + 53815X^7 + 927420X^8 + 1801024X^9 + 2674603X^{10} + 3547994X^{11} + 4419526X^{12} + 5267850X^{13} + 5764801X^{14} \quad (6.2.2.1)$$

```
>
```

F Algorithmes - Octave

F.1 Naïf

```
function[CoefFFT]=TFD(x)
%Fonction naive
%Calcul de la FFT à partir de la définition

tic %Mise en route du chronomètre

N=length(x);
CoefFFT=zeros(1,N); %Définition de la matrice qui sera donnée en sortie

for k=1 :N
        for m=1 :N
        %On effectue le calcul
        CoefFFT(k)= CoefFFT(k)+x(m)*exp(-j*2*pi*(m-1)*(k-1)/N);
        end
end
toc %Arret du chronomètre

endfunction
```

Algorithme pour le calcul de l'inverse

```
function[CoefFFT]=TFDI(x)
%Fonction naive
%Calcul de la FFT à partir de la définition

tic %Mise en route du chronomètre

N=length(x);
CoefFFT=zeros(1,N); %Définition de la matrice qui sera donnée en sortie

for k=1 :N
        for m=1 :N
        %On effectue le calcul
        CoefFFT(k)= CoefFFT(k)+x(m)*exp(j*2*pi*(m-1)*(k-1)/N);
        end
end
```

```
toc %Arret du chronomètre

endfunction
```

F.2 Récursif

```
function[CoefFFT]=TFDR(x)
%Pas de tic-toc dans cette fonction, car appel récursif.
N=length(x);
CoefFFT=zeros(1,N);

if N==1
        CoefFFT=x;
        return;
else
        xi=zeros(1,N/2);
        xp=zeros(1,N/2);
        u=zeros(1,N/2);
        v=zeros(1,N/2);
        for a=1:N/2
                xp(a)=x(2*a);
                xi(a)=x(2*a-1);
        end
        u=TFDR(xp);
        v=TFDR(xi);
        omega=exp(-2*j*pi/N);
        for k=1:N/2
                CoefFFT(k)=omega^(k-1)*u(k)+v(k);
                CoefFFT(k+N/2)=-omega^(k-1)*u(k)+v(k);
        end
end

endfunction
```

Algorithme pour le calcul de l'inverse

```
function[CoefFFT]=TFDIR(x)
%Pas de tic-toc dans cette fonction, car appel récursif.
N=length(x);
CoefFFT=zeros(1,N);
```

```
if N==1
        CoefFFT=x;
        return;
else
        xi=zeros(1,N/2);
        xp=zeros(1,N/2);
        u=zeros(1,N/2);
        v=zeros(1,N/2);
        for a=1:N/2
                xp(a)=x(2*a);
                xi(a)=x(2*a-1);
        end
        u=TFDIR(xp);
        v=TFDIR(xi);
        omega=exp(2*j*pi/N);
        for k=1:N/2
                CoefFFT(k)=omega^(k-1)*u(k)+v(k);
                CoefFFT(k+N/2)=-omega^(k-1)*u(k)+v(k);
        end
end

endfunction
```

F.3 Itératif

```
function[l]=TFDB(x)
tic;
N=length(x);
l=x;
p=N/2; %p =puissance de 2 dans la décomposition
tmp=0;

while p>=1
        z=1; %w^0
        w=exp(-j*pi/p);
        for h=1:p %Sert pour décomposer
                for m=1:N/(2*p)
                %On calcule le signal x_m
                        a=h+2*(m-1)*p; %Premier indice du signal x_(m-1)
                        b=a+p;         %Second indice du signal x_(m-1)
```

```
                tmp=(l(a)-l(b))*z;
                l(a)=l(a)+l(b);     %x_m(a) = x_(m-1)(a)+x_(m-1)(b)
                l(b)=tmp;           %x_m(b) = (x_(m-1)(a)-x_(m-1)(b))
            end
        %On passe au signal m+1 -> w<-w^(m+1)=w*w
            z=z*w;
        end
        p=p/2;
end
b=1;

%l contient maintenant la liste des signaux x_r
%On la remet dans le bon ordre

for a=1:N
        if b>a
            tmp=l(b);
            l(b)=l(a);
            l(a)=tmp;
        end
        m=N/2;
        while m>=2 & b>m
            b=b-m;
            m=m/2;
        end
        b=b+m;
end
toc %On affiche le temps écoulé
endfunction
```

Algorithme pour le calcul de l'inverse

```
function[l]=TFDIB(x)
tic;
N=length(x);
l=x;
p=N/2; %p =puissance de 2 dans la décomposition
tmp=0;

while p>=1
%On fait les signaux en 2^p
```

```
        z=1 ; %w^0
        w=exp(j*pi/p) ; %C'est la transformée inverse
        for h=1 :p %Sert pour décomposer
                for m=1 :N/(2*p)
                %On calcule le signal x_m
                        a=h+2*(m-1)*p ;
                        b=a+p ;
                        tmp=(l(a)-l(b))*z ;
                        l(a)=l(a)+l(b) ;
                        l(b)=tmp ;
                end
        %On passe au siganl m+1 -> w<-w^(m+1)=w*w
                z=z*w ;
        end
        p=p/2 ; %Signaux en 2^(p-1)
end
b=1 ;

%l contient maintenant la liste des signaux x_r
%On la remet dans le bon ordre

for a=1 :N
        if b>a
                tmp=l(b) ;
                l(b)=l(a) ;
                l(a)=tmp ;
        end
        m=N/2 ;
        while m>=2 & b>m
                b=b-m ;
                m=m/2 ;
        end
        b=b+m ;
end
%On divise les coefficients par N
for i=1 :N
        l(i)=l(i)/N ;
end
toc %On affiche le temps écoulé
endfunction
```

F.4 Multiplication de polynômes

```
function[R]=Multiplication(P,Q)

n=length(P) ;
N=2*n ;
A=zeros(1,N) ;
B=zeros(1,N) ;
R=zeros(1,N) ;

for i=1 :n
        A(i)=P(i) ;
        B(i)=Q(i) ;
end

%On calcule les transformées de chaque élément.
A=TFDB(A) ;
B=TFDB(B) ;
%On effectue les produits
for i=1 :N
        R(i)=A(i)*B(i) ;
end
R=TFDIB(R) ;
endfunction
```

G Schémas

Ci-dessous sont représentées les interprétations graphiques des résultats obtenus.

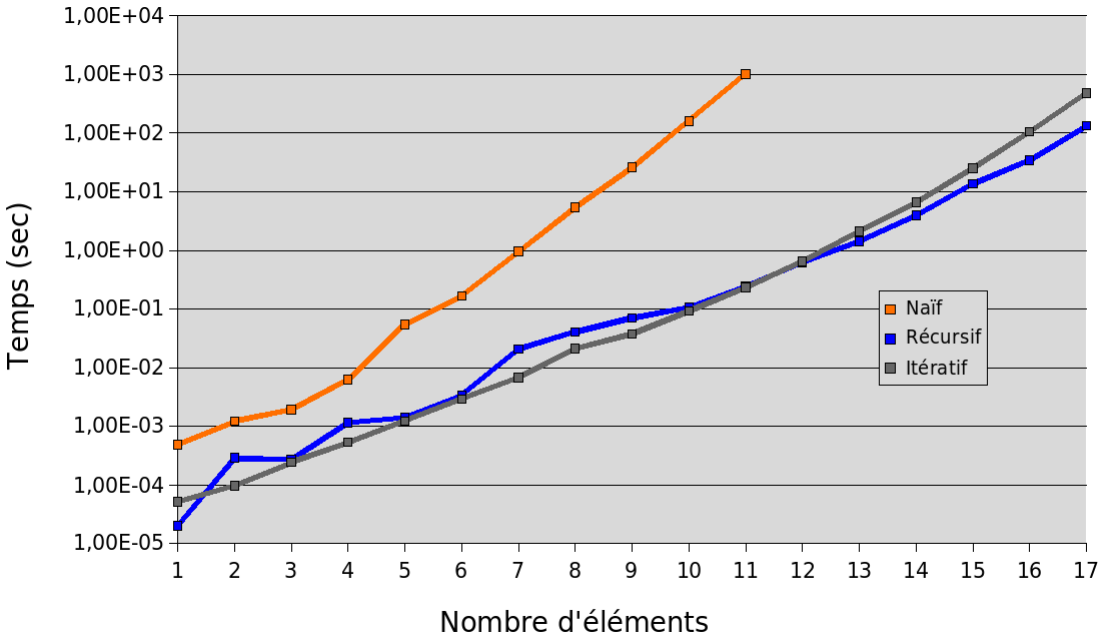

FIG. 4 – Maple

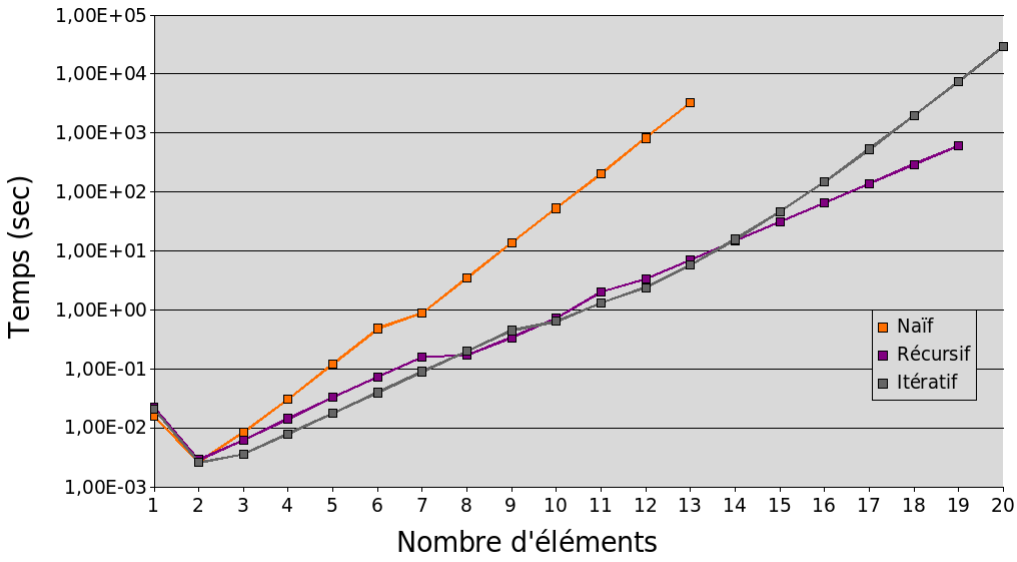

FIG. 5 – Octave

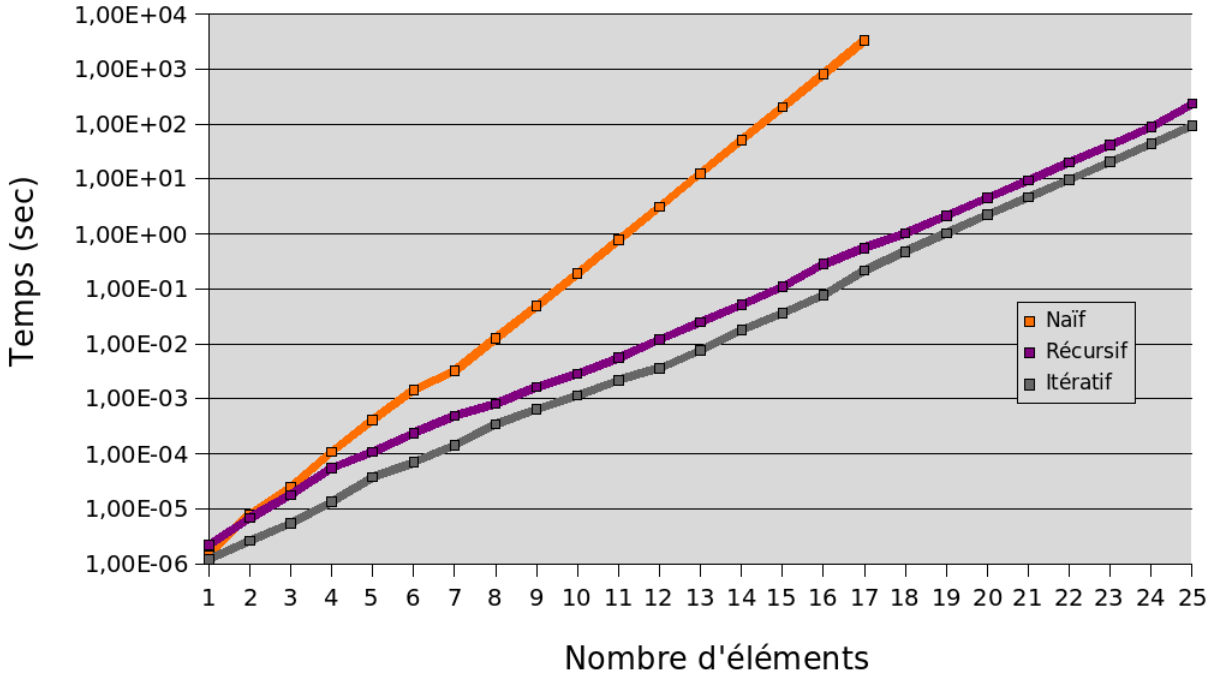

FIG. 6 – C++

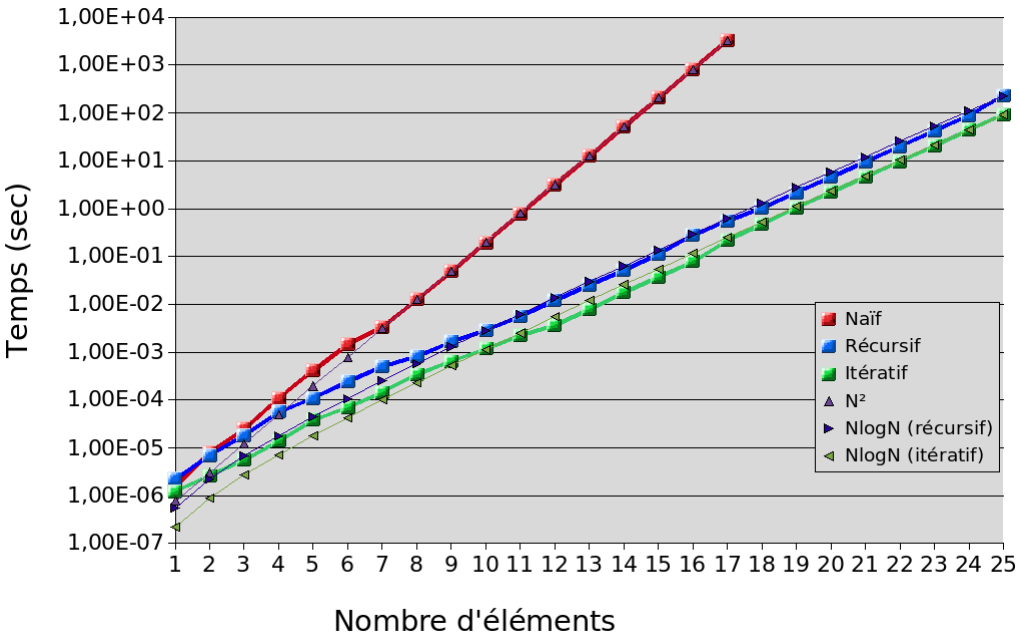

FIG. 7 – C++ - Comparaison avec les estimations théoriques