Fast Fourier Transform
Projet Mathématiques INSA STPI 2

Manon Ansart, Daming Li, Merlin Nimier-David, Pei Wang,

2012

Table des matières

1 Introduction **2**

2 Présentation mathématique **3**
2.1 Qu'est-ce qu'une transformée ? 3
2.2 Transformées de Fourier 3
2.3 Transformée de Fourier discrète 5
2.4 L'algorithme Fast Fourier Transform de Cooley-Tukey 7
2.4.1 Définition 7
2.4.2 La permutation miroir 7
2.4.3 Diviser pour régner 8
2.4.4 Expression des formules de récurrence 10
2.4.5 Permuter à nouveau 12
2.4.6 Conclusion 12
2.4.7 Exemple pour N = 4 12

3 Présentation informatique **14**
3.1 Implémentation en Pascal 14
3.1.1 Gestion des données d'entrée 14
3.1.2 Gestion des complexes 14
3.1.3 Permutation miroir 15
3.1.4 Les formules de récurrence 16
3.2 Présentation des résultats 18
3.3 Critique des résultats 18

4 Applications **20**
4.1 Analyse spectrale 20
4.2 Reconstruction d'un signal 21
4.3 Exemple : analyse et synthèse de la fonction triangle 21
4.4 Traitement d'image 26

5 Conclusion **28**

6 Bibliographie **29**

1 Introduction

La Transformée de Fourier Discrète (DFT, ou Discrete Fourier Transform en anglais) est un outils mathématique très utilisé dans le traitement du signal afin de décomposer un signal en composantes trigonométriques simples. Cette transformée possède de nombreuse applications dans des domaines variés de la physique, tel que le traitement d'images par exemple.

Dans une première partie, nous aborderons l'aspect mathématique général de la Transformée de Fourier Dicrète par un court historique et des éléments concernant les diverses Transformées de Fourier, puis plus particulièrement nous présenterons la *Fast Fourier Transform de Cooley Turkey*, un algorithme plus rapide que le calcul direct que nous avons utilisé pour notre programme.

Ce dernier sera détaillé dans une deuxième partie, au sein de laquelle nous présenterons et critiquerons également les résultats obtenus avec notre programme. Enfin, nous verrons quelques unes des applications les plus importantes de la Transformée de Fourier Discrète.

2 Présentation mathématique

2.1 Qu'est-ce qu'une transformée ?

On voit souvent en mathématiques et en physique les concepts d'*opérateur* et de *transformée*. Ils ont un point commun : ils prennent une fonction en entrée et ils produisent une fonction nouvelle en sortie. Quelques exemples d'opérateurs :

- opérateurs Nabla ∇
- intégration $\int$
- différentiation D
- transposée
- conjugaison complexe

Une transformée sort une fonction aussi, mais elle change en même temps les variables de la fonction. Un exemple que l'on connaît bien est la transformée de Legendre. En mécanique, la Lagrangienne est en fonction des coordonnées généralisées, des vitesses généralisées, et du temps. Après la transformée de Legendre, on obtient une nouvelle fonction Hamiltonienne qui est très utilisée, décrite par les coordonnées généralisées, les quantités de mouvements conjuguées, et le temps. D'autres exemples de transformées :

- transformée de Laplace
- transformée des ondettes
- transformée Z

2.2 Transformées de Fourier

En 1822, Joseph Fourier (1768-1830) publia son grand oeuvre "Théorie analytique de la chaleur" où il introduit l'utilisation de séries de Fourier pour résoudre l'équation de transfert thermique. Cette théorie connaît un grand succès parce qu'elle peut résoudre beacoup de problèmes et étudier les propriétés des fonctions. Néanmoins, au début, Fourier reçu beaucoup de critiques (surtout venant de Lagrange) qui interrogeaient sur la convergence. Pour que la théorie de Fourier soit rigoureuse, après avoir bien étudié, Dirichlet (1805-1859) donna ce que l'on appelle conditions de Dirichlet pour tester la convergence de série de Fourier.

Conditions de Dirichlet : Si une fonction $f(x)$ est périodique ($T = 2l$), bornée et absolument intégrable, qui admet un nombre fini de discontinuités et d'extrêmes, alors sa série de Fourier converge vers $f(x)$ aux points de continuités, et converge vers $\frac{f(x^+)+f(x^-)}{2}$ aux points de discontituités.

Calcul de la transformée de Fourier : La formule (forme complexe) est donnée par :

$$f(x) = \sum_{n=-\infty}^{+\infty} c_n e^{\frac{in\pi x}{l}}$$

Le $n^{\text{ème}}$ coefficient de Fourier est donné par :

$$c_n = \frac{1}{2l} \int_{-l}^{l} f(x) e^{-\frac{in\pi x}{l}} dx$$

L'obtention de ces formules fonctionne sur l'espace préhilbertien et à l'aide de l'orthogonalité des fonctions trigonométriques que l'on ne discutera pas en détails ici. On peut les retrouver dans les livres.
On observe qu'une fonction f, anciennement en fonction du temps, est transformée en une autre fonction (le coefficient) c_n, dont la variable est la fréquence. Cela indique que l'on peut étudier une fonction de deux aspects qui ont beaucoup d'intérêts physiques.

La série de Fourier peut être généralisée en l'intégrale de Fourier en laissant la période $T \to +\infty$ (dans le cas là fonction est apériodique et satisfait les conditions correspondantes). La formule est donnée par :

$$f(t) = \frac{1}{2\pi} \int_{-\infty}^{+\infty} F(\omega) e^{i\omega t} d\omega$$

avec

$$F(\omega) = \int_{-\infty}^{+\infty} f(t) e^{-i\omega t} dt$$

Ici f est une fonction du temps alors que F est une fonction de la fréquence. La deuxième formule s'appelle la transformée de Fourier et la première s'appelle la transformée de Fourier inverse.

Fourier ne s'est jamais rendu compte de l'importance de son invention. En fait, l'analyse de Fourier s'étend à une branche importante de mathématiques : l'analyse harmonique.
On classifie maintenant les fonctions en quatre catégories :

1. Continue en temps, discrète en fréquence

2. Continue en temps, continue en fréquence

3. Discrète en temps, discrète en fréquence

4. Discrète en temps, continue en fréquence

On peut voir que la première catégorie de fonctions correspond à la série de Fourier (FS) et que la deuxième correspond à l'intégrale de Fourier (FT). Ces deux sonts des transformées de Fourier énormément utilisées par les physiciens, car les fonctions rencontrées en physique sont généralement continues.
Cependant, l'ordinateur ne peut manipuler que des données discrètes. Cela nécessite la troisième catégorie de fonctions, dans laquelle on applique la DFT (Discrete Fourier Transform) : c'est là qu'intervient l'algorithme FFT. *La dernière catégorie s'appelle DTFT (Discrete Time Fourier Transform), nous ne l'aborderons pas dans le cadre de ce projet.*

2.3 Transformée de Fourier discrète

On appelle ici une fonction un signal. Soit un signal $X(k)$ de N échantillions, $0 \leq k < N$. On veut l'écrire sous le forme d'une série comme la série de Fourier. C'est-à-dire, on veut déterminer le coefficient x_n de formule :

$$X(k) = \sum_{n=0}^{N-1} x_n e^{\frac{i2\pi kn}{N}}$$

Rappelons l'algèbre linéaire sur le corps complexe, on vérifie que les vecteurs (dans notre cas, les vecteurs sont des fonctions) $u_k = [e^{\frac{2\pi ink}{N}} : n = 0, 1, \ldots, N-1]^T$ forment une base orthogonale. En effet,

$$u_k^H u_{k'} = \sum_{n=0}^{N-1} (e^{\frac{2\pi ikn}{N}})(e^{\frac{2\pi i(-k)n}{N}}) = \sum_{n=0}^{N-1} e^{\frac{2\pi i(k-k')n}{N}} = N\delta_{kk'}$$

H est ici opérateur hermitien. Grâce à cette propriété importante, par analogie avec la série de Fourier (les coefficients sont les projections du signal sur chaque vecteur de la base orthogonale), on peut ainsi obtenir facilement les coefficients x_n :

$$x_n = \frac{1}{N} \sum_{k=0}^{N-1} X(k) e^{-\frac{i2\pi kn}{N}}$$

On compare la transformée de Fourier discrète avec les séries de Fourier :

$$x_n = \frac{1}{N} \sum_{k=0}^{N-1} X(k) e^{-\frac{i2\pi kn}{N}}$$

$$c_n = \frac{1}{2l} \int_{-l}^{l} f(x) e^{-\frac{in\pi x}{l}} dx$$

On observe que :

– N correspond à la période T
– l'intégration $\int$ est remplacée par la somme $\sum$
– le signal $X(k)$ correspond à la fonction $f(x)$
– le terme exponentielle est inchangé

Il est facilement vérifié que la DFT est une application linéaire. On peut l'exprimer sous la forme d'une matrice Vandermonde :

$$F = \begin{pmatrix} \omega_N^{0\cdot 0} & \omega_N^{0\cdot 1} & \cdots & \omega_N^{0\cdot(N-1)} \\ \omega_N^{1\cdot 0} & \omega_N^{1\cdot 1} & \cdots & \omega_N^{1\cdot(N-1)} \\ \vdots & \vdots & \ddots & \vdots \\ \omega_N^{(N-1)\cdot 0} & \omega_N^{(N-1)\cdot 1} & \cdots & \omega_N^{(N-1)1\cdot(N-1)} \end{pmatrix}$$

où $\omega_N = e^{-\frac{2\pi i}{N}}$ est la $N^{\text{ième}}$ racine de l'unité.
Arpès normalisation unitaire, on obtient une matrice unitaire :

$$U = \frac{1}{\sqrt{N}} F$$
$$U^{-1} = U^H$$
$$\mid det(U) \mid = 1$$

La transformée inverse peut alors facilement s'exprimer comme :

$$F^{-1} = \frac{1}{N} F^H$$

L'orthogonalité est exprimée en orthonormalité :

$$\sum_{m=0}^{N-1} U_{km} U_{mn}^H = \delta_{kn}$$

Théorème de Parseval comme pour la série de Fourier :

$$\sum_{n=0}^{N-1} \mid x_n \mid^2 = \sum_{n=0}^{N-1} \mid X_n \mid^2$$

Ce théorème montre que la norme du vecteur reste la même lors de la transformée.

On voit maintenant que pour effectuer la DFT, il suffit de multiplier la matrice F avec le vecteur X. Pourtant ce sera un lourde tâche même pour l'ordinateur lorsque N est grand. En effet, il faudrait alors effectuer $N * N$ opérations. Pour faciliter ce travail, de nombreux algorithmes (dont la famille d'algorithme Transformée de Fourier Rapide) ont été inventés. Le plus connu est celui de Cooley-Tukey.

2.4 L'algorithme Fast Fourier Transform de Cooley-Tukey

Cet algorithme permet de calculer la Transformée de Fourier Discrète d'un vecteur en un temps record. Décrit par Cooley–Tukey en 1965, c'est l'agorithme de calcul rapide le plus répandu. Il est basé sur le principe du *divide and conquer*, qui consiste à subdiviser un problème complexe en sous-problèmes plus simples, de manière récurrente.

2.4.1 Définition

Soit g un vecteur de taille $N = 2^M$. La Transformée de Fourier Discrète est l'application linéaire définie par :

$$G : \begin{cases} \mathbb{C}^N & \longrightarrow & \mathbb{C}^N \\ g & \longmapsto & G \end{cases}$$

où G est le vecteur complexe de dimension N tel que :

$$G_j = \sum_{k=0}^{N-1} g_k * W^{j*k} \text{ avec } W = e^{\frac{2i\pi}{N}}$$

On note S la matrice de l'application :

$$S = (W_N^{jk})$$

On a donc :

$$G = Sg$$

La matrice S est de dimension $N \times N$. **Les lignes et colonnes sont numérotées de 0 à N-1**. Le calcul de G correspond donc à un produit matrice vecteur très grand et donc long à réaliser, c'est pourquoi il est préférable d'utiliser une autre méthode dont le temps de calcul est plus court.

2.4.2 La permutation miroir

La première étape de l'algorithme consiste à permuter les lignes de la matrice S selon la *transformation mirroir*. Cette transformation est définie sur l'ensemble des entiers, et agit sur la représentation binaire. On l'appliquera au numéro des lignes de la matrice S, qui est de dimension $n = 2^M$. Les entiers à transformer sont donc compris entre 0 et $(2^M - 1)$.
Tout entier n peut être décomposé en base 2 par :

$$n = \sum_{i=0}^{M-1} \lambda_i * 2^i$$

avec λ_i le $i^{\text{ième}}$ bit, i variant de 0 à $M - 1$.
On choisit ici de limiter la représentation binaire à M bits. Les entiers transformés ne dépassant pas $(2^M - 1)$, M bits suffisent à tous les représenter.

Exemple : pour $M = 4$, les numéros de ligne varient entre 0 et $2^4 - 1 = 15$, soit en binaire entre $(0000)_2$ et $(1111)_2$.

La permutation miroir échange simplement la position des bits :

$$(\lambda_{M-1}, \lambda_M, \ldots, \lambda_1, \lambda_0)_2 \longmapsto (\lambda_0, \lambda_1, \ldots, \lambda_M, \lambda_{M-1})_2$$

On la définit donc comme suit, $\forall n \in [0, 2^M - 1]$:

$$r_M \begin{cases} \mathbb{N} & \longrightarrow & \mathbb{N} \\ n & \longmapsto & r_M(n) = \sum_{i=0}^{M-1} \lambda_{M-1-i} * 2^i \end{cases}$$

On remarque que $r_M(r_M(n)) = n$.

Cette permutation, une fois appliquée à la matrice S, donnera une matrice T possédant des propriétés très utiles sur lesquelles se reposent l'algorithme.

2.4.3 Diviser pour régner

Nous allons appliquer la permutation miroir à la matrice S, c'est à dire que nous allons changer l'ordre des ses lignes : la ligne j deviendra la ligne $r_M(j)$. Ainsi, nous n'allons plus multiplier g par S, mais par PS, que l'on notera désormais T^M. T^M possède des propriétés très intéressantes, sur lesquelles repose l'algorithme FFT.
En effet, on observe que :

$$T^M = \begin{pmatrix} T^{M-1} & T^{M-1} \\ T^{M-1} * L^{M-1} & -T^{M-1} * L^{M-1} \end{pmatrix}$$

Nous allons démontrer cette égalité bloc par bloc.
On note :
- $\alpha_{j,k}$ le bloc supérieur-gauche, avec $j \in [0; 2^{M-1}[$ et $k \in [0; 2^{M-1}[$
- $\beta_{j,k}$ le bloc supérieur-droit, avec $j \in [0; 2^{M-1}[$ et $k \in [2^{M-1}; 2^M[$
- $\delta_{j,k}$ le bloc inférieur-gauche, avec $j \in [2^{M-1}; 2^M[$ et $k \in [0; 2^{M-1}[$
- $\mu_{j,k}$ le bloc inférieur-droit, avec $j \in [2^{M-1}; 2^M[$ et $k \in [2^{M-1}; 2^M[$

$$T^M = \begin{pmatrix} \alpha_{j,k} & \beta_{j,k} \\ \delta_{j,k} & \mu_{j,k} \end{pmatrix}$$

Blocs supérieurs Pour $r_M(j) = j' \in [0; 2^{M-1}[$ et $k \in [0; 2^M[$, on a :

$$S_{r_M(j),k} = W^{r_M(j)*k} = \exp(\frac{2I\pi}{2^M} \sum_{l=0}^{M-1} \lambda_{M-1-l} * 2^l * k)$$

Or pour $r_M(j) \in [0; 2^{M-1}[$, le bit de point fort de $r_M(j)$ est égal à 0 donc $\lambda_{M-1} = 0$.
On a donc :

$$S_{r_M(j),k} = \exp(\frac{2I\pi}{2^M} \sum_{l=1}^{M-1} \lambda_{M-1-l} * 2^l * k)$$

On effectue un changement d'indice en posant $n = l - 1$.

$$\begin{aligned} S_{r_M(j),k} &= \exp(\tfrac{2I\pi}{2^M} \textstyle\sum_{n=0}^{M-2} \lambda_{M-2-n} * 2^{n+1} * k) \\ &= \exp(\tfrac{2I\pi}{2^{M-1}} \textstyle\sum_{n=0}^{M-2} \lambda_{M-2-n} * 2^n * k) \end{aligned}$$

– Pour $k \in [0; 2^{M-1}[$, c'est à dire dans le bloc supérieur-gauche, on a :

$$\exp(\frac{2I\pi}{2^{M-1}} \sum_{n=0}^{M-2} \lambda_{M-2-n} * 2^n * k) = S_{r_{M-1}(j),k}$$

Donc $\boxed{\alpha_{j,k} = T^{M-1}_{j,k}}$

– Pour $k \in [2^{M-1}; 2^M[$, c'est à dire dans le bloc supérieur-droit, on a :

$$\begin{aligned} & \exp(\tfrac{2I\pi}{2^{M-1}} \textstyle\sum_{n=0}^{M-2} \lambda_{M-2-n} * 2^n * k) \\ = \quad & \exp(\tfrac{2I\pi}{2^{M-1}} \textstyle\sum_{n=0}^{M-2} \lambda_{M-2-n} * 2^n * k - 2^{M-1} + 2^{M-1}) \\ = \quad & \begin{array}{l} \exp(\tfrac{2I\pi}{2^{M-1}} \textstyle\sum_{n=0}^{M-1} \lambda_{M-2-n} * 2^n * k - 2^{M-1}) \\ * \exp(\tfrac{2I\pi}{2^{M-1}} \textstyle\sum_{n=0}^{M-2} \lambda_{M-2-n} * 2^n * 2^{M-1}) \end{array} \\ = \quad & T_{j,k-2^{M-1}} * \exp(2I\pi \textstyle\sum_{n=0}^{M-2} \lambda_{M-2-n} * 2^n) \\ = \quad & T_{j,k-2^{M-1}} \end{aligned}$$

Donc $\boxed{\beta_{j,k} = T_{j,k-2^{M-1}}}$

Blocs inférieurs Pour $r_M(j) = j' \in [2^{M-1}; 2^M[$ et $k \in [0; 2^M[$, on a :

$$S_{r_M(j),k} = W^{r_M(j)*k} = \exp(\frac{2I\pi}{2^M} \sum_{l=0}^{M-1} \lambda_{M-1-l} * 2^l * k)$$

Or pour $r_M(j) \in [2^{M-1}; 2^M[$, le bit de point fort de $r_M(j)$ est égal à 1 donc $\lambda_{M-1} = 1$.
On a donc :

$$S_{r_M(j),k} = \exp(\frac{2I\pi}{2^M} \sum_{l=1}^{M-1} \lambda_{M-1-l} * 2^l * k) * \exp(\frac{2I\pi}{2^M} * k)$$

On effectue le même changement d'indice que précédemment : $l = n - 1$.

– Pour $k \in [0; 2^{M-1}[$, c'est à dire dans le bloc inférieur-gauche, on a :

$$\begin{aligned} \exp(\tfrac{2I\pi}{2^M} \textstyle\sum_{l=1}^{M-1} \lambda_{M-1-l} * 2^l * k) &= \alpha_{j-2^{M-1},k} \\ &= T^{M-1}_{j-2^{M-1},k} \end{aligned}$$

De plus, on note L^{M-1} la matrice diagonale de dimension $(M-1)$ telle que :

$$L^{M-1}_{k,k} = \exp(\frac{2I\pi}{2^M} * k)$$

On a donc finalement, pour le bloc inférieur-droit :

$$\boxed{\delta_{j,k} = T^{M-1}_{j-2^{M-1},k} * L^{M-1}_{k,k}}$$

– Pour $k \in [2^{M-1}; 2^M[$, c'est à dire dans le bloc inférieur-droit, on a :

$$\exp(\frac{2I\pi}{2^M} \sum_{l=1}^{M-1} \lambda_{M-1-l} * 2^l * k) = \beta_{j-2^{M-1},k} = T^{M-1}_{j-2^{M-1},k}$$

De plus, on remarque que :

$$\begin{aligned}
\exp(\tfrac{2I\pi}{2^M} * k) &= \exp(\tfrac{2I\pi}{2^M} * (k - 2^{M-1} + 2^{M-1})) \\
&= \exp(\tfrac{2I\pi}{2^M}(k - 2^{M-1})) * exp(\tfrac{2I\pi}{2^M}(2^{M-1})) \\
&= \exp(\tfrac{2I\pi}{2^M}(k - 2^{M-1})) * exp(I\pi) \\
&= -\exp(\tfrac{2I\pi}{2^M}(k - 2^{M-1})) \\
\exp(\tfrac{2I\pi}{2^M} * k) &= -L^{M-1}_{k-2^{M-1},k-2^{M-1}}
\end{aligned}$$

On a donc $\boxed{\mu_{j,k} = -T_{j-2^{M-1},k} * L^{M-1}_{k-2^{M-1},k-2^{M-1}}}$

Conclusion : On a démontré bloc par bloc que :

$$T^M = \begin{pmatrix} T^{M-1} & T^{M-1} \\ T^{M-1} * L^{M-1} & -T^{M-1} * L^{M-1} \end{pmatrix}$$

2.4.4 Expression des formules de récurrence

En vue d'implémenter l'algorithme, il est judicieux d'exprimer le problème sous forme de formules de récurrence. On pose alors :

$$u_0 = g$$

$$u_1 = \begin{pmatrix} I^{M-1} & I^{M-1} \\ L^{M-1} & -L^{M-1} \end{pmatrix} * u_0$$

Le premier rang est le vecteur donné à transformer, le rang suivant s'obtient par produit matriciel. On obtient alors facilement l'expression de chaque coordonnée du vecteur u_1 :

– Pour $j \in [0, 2^{M-1}[$ c'est à dire dans la moitié supérieure du vecteur u_1 :

$$u_1[j] = u_0[j] + u_0[j + 2^{M-1}]$$

– Pour $j \in [2^{M-1}, 2^M[$ c'est à dire dans la moitié inférieure du vecteur u_1 :

$$u_1[j] = e^{\frac{2I\pi}{2^M} * j} * (u_0[j] - u_0[j + 2^{M-1}])$$

Le terme exponentielle provient de la matrice diagonale $L^{M-1} = (e^{\frac{2I\pi}{2^M} * j})_{j,j}$.

À chaque étape, on réduit davantage la taille des sous-blocs de la matrice (diviser) car on la divise en qautre blocs. Ainsi, au rang 1, chaque sous-bloc est une matrice carrée de dimension $\frac{n}{2}$. Au rang suivant, on divise chaque sous-bloc en quatre : la dimension est alors $\frac{n}{4}$. Au rank k, on a alors des sous-blocs de dimension $\frac{n}{2^k} = \frac{2^M}{2^k} = 2^{M-k}$, et il y a 2^{k-1} bloc dans la matrice.

$$u_k = \begin{pmatrix} \begin{pmatrix} I^{M-k} & I^{M-k} \\ L^{M-k} & -L^{M-k} \end{pmatrix} & 0 & 0 \\ 0 & \ddots & 0 \\ 0 & 0 & \begin{pmatrix} I^{M-k} & I^{M-k} \\ L^{M-k} & -L^{M-k} \end{pmatrix} \end{pmatrix} * u_{k-1}$$

Les sous-blocs étant tous des matrices carrées diagonales, le produit avec le vecteur u_{k-1} peut être exprimé de manière relativement compacte. On introduit un indice q permettant de sélectionner le sous-bloc et un indice j permettant de se déplacer à l'intérieur de ce sous bloc.

– Pour $q \in [0, 2^{k-1} - 1[$, c'est à dire pour chacun des 2^{k-1} sous-blocs
– et pour $j \in [0, 2^{M-k} - 1[$ c'est à dire pour chacune des 2^{M-k} lignes de ce sous-bloc, on a :

$$\begin{cases} u_k[q * 2^{M-k+1} + j] &= u_{k-1}[q * 2^{M-k+1} + j] + u_{k-1}[q * 2^{M-k+1} + j + 2^{M-k}] \\ u_k[q * 2^{M-k+1} + j + 2^{M-k}] &= e^{\frac{2I\pi}{2^{M-k+1}} * j} * (\quad u_{k-1}[q * 2^{M-k+1} + j] \\ & \qquad\qquad -u_{k-1}[q * 2^{M-k+1} + j + 2^{M-k}]) \end{cases}$$

Le résultat de la transformation, G, est obtenu après M étapes de la récurrence. On a en effet, pour $k = M$:

$$u_M = \begin{pmatrix} \begin{pmatrix} I^0 & I^0 \\ L^0 & -L^0 \end{pmatrix} & 0 & 0 \\ 0 & \ddots & 0 \\ 0 & 0 & \begin{pmatrix} I^0 & I^0 \\ L^0 & -L^0 \end{pmatrix} \end{pmatrix} * u_{M-1} = \begin{pmatrix} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} & 0 & 0 \\ 0 & \ddots & 0 \\ 0 & 0 & \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \end{pmatrix} * u_{M-1}$$

$u_M = T^M * u_0 = G^* = PG$ avec P la matrice de permutation.

2.4.5 Permuter à nouveau

Il ne faut donc pas oublier d'appliquer à nouveau la permutation miroir sur les lignes du vecteur u_M. En effet, le résultat de la récurrence est :

$$\begin{aligned} & u_M = T^M u_0 = PG \\ \Rightarrow \quad & Pu_m = PPG = G \end{aligned}$$

2.4.6 Conclusion

Avec un produit matriciel direct $G = Sg$, il aurait fallu effectuer de l'ordre de n^2 opérations (complexité $O(n^2)$). Grâce à la FFT, nous avons obtenu le vecteur transformé G après seulement $M = log_2(n)$ étapes de récurrence comprenant chacune $2^{k-1} * 2^{M-k} * 2 = n$ opérations : la complexité est $O(nlog_2(n))$. L'amélioration est de l'ordre de $\frac{n}{log_2(n)}$.
La transformée de Fourier discrète ayant un champ d'application extrêmement large, un tel algorithme de calcul est proprement remarquable.

Exemple : Pour un calcul sur 1024 échantillons, on fait le calcul $\frac{1024}{log_2(1024)} = 102.4$ fois plus rapidement avec la FFT qu'avec le calcul direct.

Remarque : La version de l'algorithme décrite ici, ainsi que son implémentation, sont limitées à l'utilisation de vecteurs de dimension $n = 2^M$. Cependant, il existe diverses méthodes, qui ne seront pas détaillées ici, permettant de travailler avec des vecteurs de dimension arbitraire.

2.4.7 Exemple pour N = 4

Pour illustrer le fonctionnement de l'algorithme, nous allons calculer la transformée d'un vecteur simple de dimension $N = 2^2$. On est donc dans un cas $M = 2$, on aura 2 étapes de récurrence à calculer. Prenons :

$$g = u_0 = (1, 2, 3, 4)$$

- Rang $k = 1$: $q = 0$ constant et j varie de 0 à 1. On calcule chaque coordonnée du vecteur u_1 à l'aide des formules de récurrence :

$$\begin{cases} u_1(0) & = u_0(0) + u_0(2) & = 1 + 3 & = 4 \\ u_1(1) & = u_0(1) + u_0(1+2) & = 2 + 4 & = 6 \\ u_1(0+2) & = \exp(\frac{2I\pi * 0}{2^{2-1+1}}) * (u_0(0) - u_0(0+2)) & = 1 * (1-3) & = -2 \\ u_1(1+2) & = \exp(\frac{2I\pi * 1}{2^{2-1+1}}) * (u_0(1) - u_0(1+2)) & = I * (2-4) & = -2I \end{cases}$$

- Rang $k = 2$: q varie de 0 à 1 et $j = 0$ constant. De même, on calcule chaque coordonnée du vecteur u_2 à l'aide des formules de récurrence :

$$\begin{cases} u_2(0) & = u_1(0) + u_1(1) & = 4 + 6 & = 10 \\ u_2(2) & = u_1(2) + u_1(2+1) & = -2 - 2I & = -2 - 2I \\ u_2(0+1) & = \exp(\frac{2I\pi * 0}{2}) * (u_1(0) - u_1(0+1)) & = 1 * (4-6) & = -2 \\ u_2(2+1) & = \exp(\frac{2I\pi * 0}{2}) * (u_1(2) - u_1(2+1)) & = 1 * (-2 - (-2I)) & = -2 + 2I \end{cases}$$

Finalement, il ne faut pas oublier de permuter à nouveau les lignes du vecteur u_2 avec la permutation miroir (ici, on inverse alors les lignes 1 et 2). On obtient ainsi le résultat de la transformée :

$$\begin{aligned} G[i] &= u_2[r_M(i)] \\ G &= \begin{pmatrix} 10 \\ -2-2I \\ -2 \\ -2+2I \end{pmatrix} = DFT(g) \end{aligned}$$

3 Présentation informatique

3.1 Implémentation en Pascal

3.1.1 Gestion des données d'entrée

Pour permettre au programme de prendre en entrée des vecteurs de grande dimension, nous avons choisi de lire directement les données dans un fichier `source.d` rempli par l'utlisateur. De même, le vecteur transformé (données de sortie) est écrit dans le fichier `result.d` pour permettre d'utiliser le résultat plus facilement.
Les données d'entrée doivent être formatées d'une manière précise afin d'être comprises par le programme :

- Une coordonnée du vecteur par ligne de fichier
- D'abord la partie réelle, puis la partie imaginaire. Elles doivent être séparées par une virgule suivie d'un espace. Si une partie est nulle, il faut écrire explicitement un 0.

Par exemple, pour utiliser le vecteur $(1 + 2i, 42)$, on écrira :

```
1  1, 2
2  42, 0
```

Après avoir lu le fichier `source.d`, le programme connaît la dimension du vecteur donné. On peut alors déterminer si cette dimension est de la forme $n = 2^m$. Si ce n'est pas le cas, le programme s'arrête car on ne peut pas appliquer l'algorithme.

3.1.2 Gestion des complexes

Afin de pouvoir gérer des vecteurs complexes dans notre programme, nous avons défini deux Types de données. Le premier, représentant un nombre complexe, est un simple tableau de deux réels (partie réelle, partie imaginaire). Le second représente un vecteur complexe et est un tableau de dimension arbitraire de complexes (chaque complexe est une des coordonnées du vecteur).

```
1  type Complex = Array [0..1] of Real;
2  type CArray = Array of Complex;
```

Ensuite, il a fallu écrire des fonctions pour les opérations de bases sur les complexes.

```
1  function multiply(c1, c2 : Complex) : Complex;
2  var prod : Complex;
3  begin
4      prod[0] := c1[0] * c2[0] - c1[1] * c2[1];
5      prod[1] := c1[0] * c2[1] + c1[1] * c2[0];
6      multiply:= prod;
7  end;
8
9  function add(c1,c2 : Complex):Complex;
10 var sum : Complex;
11 begin
12     sum[0] := c1[0] + c2[0];
13     sum[1] := c1[1] + c2[1];
14     add := sum;
15 end;
16
17 function substract(c1,c2 : Complex):Complex;
18 var sum : Complex;
19 begin
20     sum[0] := c1[0] - c2[0];
21     sum[1] := c1[1] - c2[1];
22     substract := sum;
23 end;
```

Enfin, cette fonction permet de prendre le conjugué d'un nombre complexe. Elle est utile pour le calcul de la transformée de Fourier inverse.

```
1  function conjugate(c1 : Complex):Complex;
2  var conj : Complex;
3  begin
4      conj[0] := c1[0];
5      conj[1] := - c1[1];
6      conjugate := conj;
7  end;
```

3.1.3 Permutation miroir

Pour effectuer la permutation miroir sur un entier, on fait usage de deux fonctions agissant sur la représentation binaire du nombre. `getBit` permet de connaître l'état d'un bit dans le nombre donné tandis que `setBit` permet d'affecter la valeur choisie à ce bit.

```
1  function getBit(const Val: DWord; const BitVal: Byte): Boolean;
2  begin
3      getBit := (Val and (1 shl BitVal)) <> 0;
4  end;
5
```

```
6  function enableBit(const Val: DWord; const BitVal: Byte;
7                     const SetOn: Boolean): DWord;
8  begin
9      enableBit := (Val or (1 shl BitVal))
10                  xor (Integer(not SetOn)
11                  shl BitVal);
12 end;
```

Il suffit ensuite de permuter l'état du $i$ème bit avec celui du $(m-i)$ème bit. La connaissance de m est nécessaire : en effet, on ne peut pas se contenter d'effectuer la permutation sur toute la longueur de l'entier, qui pourra être codé sur 8, 16 ou 32 bits selon le système. On ne travaille que sur les m premiers bits, et on laisse les autres dans leur état précédent (en théorie, ils restent tous à 0).

```
1  function mirrorTransform(n,m:Integer):Integer;
2  var i,p : Integer;
3  begin
4      p := 0;
5
6      for i:=0 to m-1 do
7      begin
8          p := enableBit(p, m-1-i, getBit(n, i));
9      end;
10
11     mirrorTransform:=p;
12 end;
```

3.1.4 Les formules de récurrence

Une fois ces différents outils en place, il est relativement simple d'implémenter l'algorithme FFT. En effet, il suffit de partir du vecteur u_O donné puis d'appliquer les formules de récurrence M fois. Pour finir, on réorganise le vecteur final u_M en appliquant la permutation miroir aux numéros de ses lignes.

```
1  function fft(g:CArray; order:Integer):CArray;
2  var previousRank, nextRank : CArray;
3      i : Integer;
4  begin
5      previousRank := g;
6      for i:=1 to order do
7          nextRank := doStep(i, order, previousRank);
8          previousRank := nextRank;
9      end;
10
11     nextRank := doPermutation(nextRank, order);
12     fft := nextRank;
13 end;
```

La fonction `doStep` calcule donc les coordonnées du vecteur u_k en fonction de celles du vecteur u_{k-1}. Dès le début de la fonction, on calcule 2^{m-k} et on l'enregistre dans la variable `offset`. En effet, ce terme est extrêmement utilisé dans les calculs, il est donc bon, pour les performances du programme, de conserver sa valeur dans une variable.

```
1  function doStep(k, M : Longint; prev:CArray):CArray;
2  var expTerm, substractTerm : Complex;
3      dimension, q, j, offset : Longint;
4      u : CArray;
5  begin
6      offset := system.round(intpower(2, M-k));
7
8      SetLength(u, length(prev));
9
10     for q:=0 to system.round(intpower(2, k-1) - 1) do
11         for j:=0 to (offset - 1) do
12
13             u[q*2*offset + j] := add( prev[q*2*offset + j],
14                                       prev[q*2*offset + j + offset] );
15
16             expTerm[0] := cos( (j * PI) / offset );
17             expTerm[1] := sin( (j * PI) / offset );
18             substractTerm := substract( prev[q*2*offset + j],
19                                         prev[q*2*offset + j + offset] );
20             u[q*2*offset + j + offset] := multiply(expTerm,
21                                                    substractTerm);
22         end;
23     end;
24
25     doStep := u;
26 end;
```

La boucle extérieure, d'indice `q`, permet de se déplacer de bloc en bloc tandis que la boucle intérieure, d'indice `j`, permet de se déplacer de ligne en ligne à l'intérieur du bloc en cours. Il suffit alors de calculer, comme défini par les formules de récurrence :

$$\begin{cases} u_k(q*2^{M-k+1}) &= u_{k-1}(q*2^{M-k+1}+j) + u_{k-1}(q*2^{M-k+1}+j+2^{M-k}) \\ u_k(q*2^{M-k+1}) &= u_{k-1}(q*2^{M-k+1}+j) + u_{k-1}(q*2^{M-k+1}+j+2^{M-k}) \end{cases}$$

Avec q variant de 0 à $2^{k-1}-1$ et j variant de 0 à $2^{M-k}-1$.

3.2 Présentation des résultats

Le programme, désormais fonctionnel, permet d'appliquer la transformée de Fourier à n'importe quel vecteur complexe de dimension $n = 2^M$.
Les résultats ont été tronqués en longueur pour ne pas encombrer le rapport.

$$g = \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \end{pmatrix} \longrightarrow G = \begin{pmatrix} 4 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

$$g = \begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \end{pmatrix} \longrightarrow G = \begin{pmatrix} 10 \\ -2 - 2i \\ -2 \\ -2 + 2i \end{pmatrix}$$

$$g = \begin{pmatrix} 1 + i \\ 1 - i \\ 2 \\ 2i \\ i \\ -i \\ 4 \\ 7 - 3i \end{pmatrix} \longrightarrow G = \begin{pmatrix} 1,50E+001 - 1,00E+000i \\ 3,12E+000 - 9,78E+000i \\ -4,00E+000 - 4,00E+000i \\ -8,19E+000 + 1,29E+000i \\ -1,00E+000 + 5,00E+000i \\ -1,12E+000 + 5,78E+000i \\ -6,00E+000 + 8,00E+000i \\ 1,02E+001 + 2,71E+000i \end{pmatrix}$$

Le programme a été compilé à l'aide de Free Pascal Compiler (FPC) 2.6.0 pour architecture i386 puis exécuté sous OSX 10.8.2, processeur Intel Core i7.

3.3 Critique des résultats

Pour tester la précision du programme, comparons ses résultats à ceux calculés avec Maple (Maple est une référence car ce logiciel utilise du calcul multi-précision, plus avancé que celui implémenté dans notre programme).

Vecteur initial	Notre programme	Maple
1 + i	+15,000000000 -1,000000000i	+15,000000000 -1,000000000i
1 - i	+3,1213203440 -9,778174593i	+3,1213203430 -9,778174591i
2	-4,000000000 -4,000000000i	-4,000000000 -4,000000000i
2i	-8,192388155 +1,2928932190i	-8,192388153 +1,292893219i
i	-1,000000000 +5,0000000000i	-1,000000000 +5,000000000i
- i	-1,121320344 +5,7781745930i	-1,121320343 +5,778174591i
4	-6,000000000 +8,0000000000i	-6,000000000 +8,000000000i
7 - 3i	+10,192388155 +2,7071067810i	+10,192388150 +2,707106781i

Écart relatif Δ_k	
0	%
2,07030887128991E-010	%
0	%
2,85321211555446E-010	%
0	%
3,60357577407922E-010	%
0	%
5,01736720506764E-010	%

On calcule l'écart relatif sur le module de chaque coordonnée du vecteur transformé :

$$\Delta_k = \frac{||z_k^{\text{maple}}| - |z_k^{\text{programme}}||}{|z_k^{\text{maple}}|}$$

L'écart relatif est de l'ordre de grandeur 10^{-10} pour un vecteur de dimension $n = 8$. La précision de notre programme est satisfaisante.

4 Applications

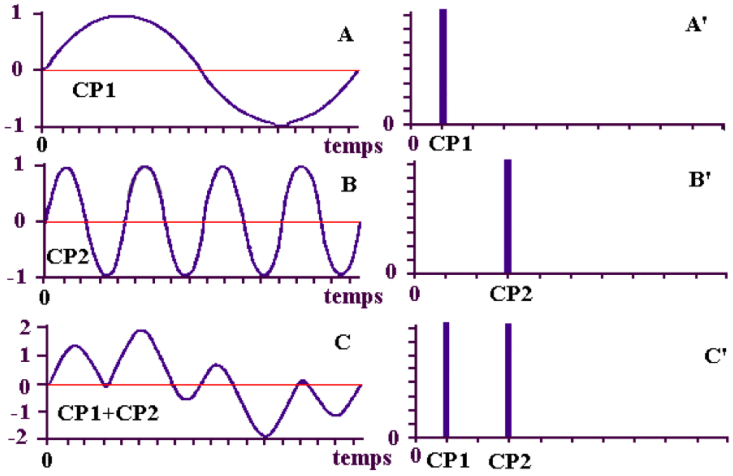
4.1 Analyse spectrale

La Transformée de Fourier Discrète (DFT) permet de mettre en évidence les composantes périodiques présentes dans un signal.

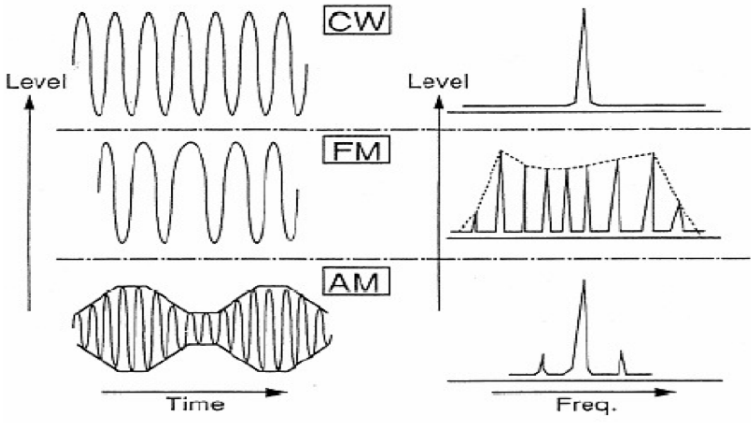
Rappel : Pour un signal discret x échantillonné dans le temps avec N échantillons, on a définit le résultat de la DFT X comme :

$$X(k) = \sum_{n=0}^{N-1} x(n) e^{-j\frac{2\pi}{N}nk} \text{ pour } k \in [0; N[$$

Exemple : Appliquons la Transformée de Fourier Discrète à divers signaux. À gauche, les courbes A et B sont caractérisées par une composante périodique de fréquence différente mais de même amplitude. La courbe C, qui résulte de l'addition de deux signaux, est caractérisée par les deux composantes périodiques. Cette décomposition, représentée sur les courbes A', B' et C', correspond aux $X(k)$ obtenus par la Transformée de Fourier Discrète.



Exemple : À gauche, plusieurs signaux modulés permettant de transporter du son via radio. À droite, on obtient le spectre des fréquences grâce à l'application de la DFT.



4.2 Reconstruction d'un signal

Une autre application très utile est de reconstituer un signal à partir de ses composantes périodiques : il s'agit de la Transformée de Fourier Discrète Inverse (IDFT). Étant donnés les $X(k)$ toutes les composantes périodiques du signal, on peut re-calculer la valeur des N échantillons $x(n)$ comme suit :

$$x(n) = \frac{1}{N} \sum_{k=0}^{N-1} X(k) e^{+j\frac{2\pi}{N}nk} \text{ pour } n \in [0; N[$$

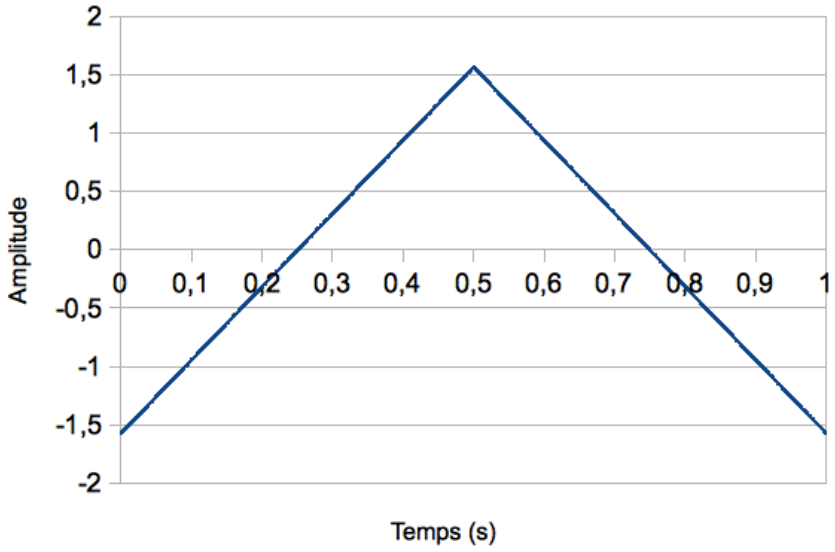
Il est ainsi possible de synthétiser un signal : c'est ce que nous ferons avec l'exemple de la fonction triangle.

4.3 Exemple : analyse et synthèse de la fonction triangle

On étudie la fonction triangle suivante, périodique de période $T = 1$s :

$$f(t) = \begin{cases} +t * 2\pi - \frac{\pi}{2} & \text{si } t \in [0; \frac{1}{2}] \\ -t * 2\pi + \frac{3\pi}{2} & \text{si } t \in]\frac{1}{2}; 1] \end{cases}$$

On échantillonne $f(t)$ à une fréquence $f = 2048$Hz. Ainsi, sur une période $T = 1$s, on obtient 2048 échantillons.



Une fois l'échantillonnage effectuée, on applique la transformée de Fourier grâce au programme réalisé. Pour comparer les résultats, on réitère l'expérience avec un échantillonnage à $16Hz$. *L'échelle verticale des graphes suivants a été tronquée.*

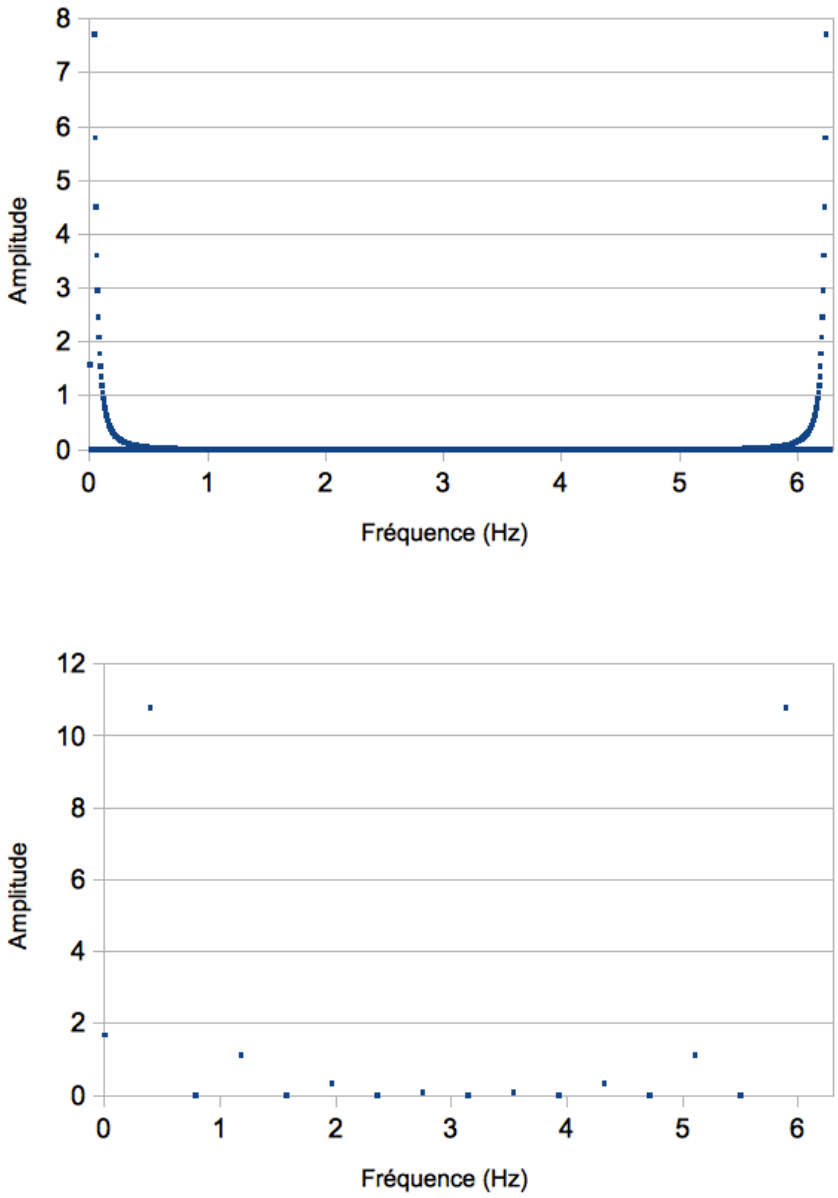


FIGURE 4.1 – Résultat de la transformée pour $n = 2048$ et $n = 16$

Nous souhaitons trouver à l'aide de Maple la série de Fourier de la fonction triangle précédemment décrite, et ainsi pouvoir l'exprimer en une somme des fonctions trigonométriques.
Comme la fonction triangle est une fonction paire, elle peut être exprimée sous la forme :

$$f(x) = \frac{a_0}{2} + \sum_{n=1}^{\infty} a_n cos(nx)$$

avec les coefficients a_n tels que :

$$a_n = \frac{1}{\pi} \int_{-\pi}^{\pi} f(x) cos(nx) dx$$

Comme il est impossible en pratique de calculer la série jusqu'à l'infini, on prend les N premiers termes de la série :

$$S_N = \frac{a_0}{2} + \sum_{n=1}^{N} a_n cos(nx)$$

avec les coefficients a_n calculés par Maple selon :

$$a_n = \frac{1}{\pi} \int_{-\pi}^{\pi} f(x) cos(nx) dx$$

On obtient ainsi une approximation d'ordre N de la fonction triangle. On peut estimer la précision de cette approximation : d'après le théorème de Pythagore, la moyenne du carré de l'écart est donnée par :

$$\Delta^2 = \frac{1}{\pi} \int_{-\pi}^{\pi} |f(x) - s_N(x)|^2 dx = \frac{1}{\pi} \int_{-\pi}^{\pi} |f(x)|^2 dx - \frac{1}{2} a_0^2 - \sum_{i=1}^{N} a_i^2$$

Pour obtenir une visualisation de cette précision, toujours à l'aide de Maple, on trace la somme partielle S_N successivement pour $N = 2$, $N = 4$, $N = 8$ et $N = 32$. Le résultat est visible sur la figure 4.2. On remarque que dès $N = 32$, le signal original est reconstitué de manière relativement précise.

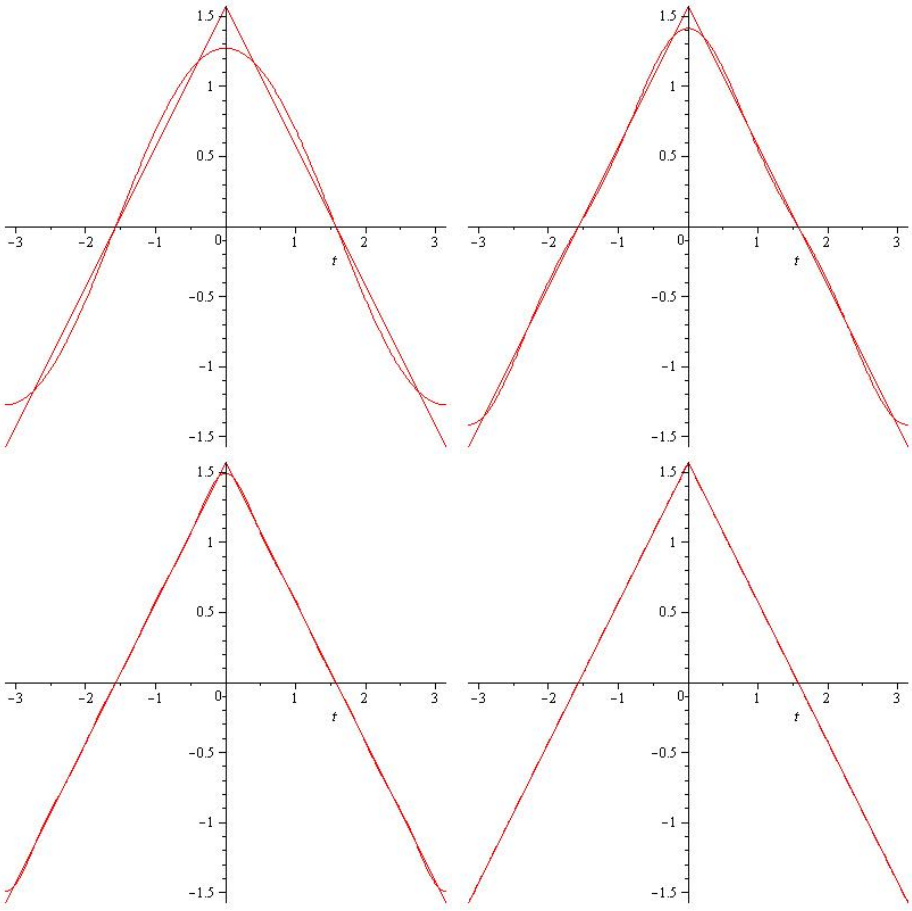


FIGURE 4.2 – Approximation du signal (fonction triangle) à partir des coefficients de Fourier sous Maple pour $N = 2$, $N = 4$, $N = 8$ et $N = 32$

4.4 Traitement d'image

Une image correspond à la répartition d'intensités lumineuses dans un plan, elle peut donc être vue comme un signal visuel en deux dimensions. De ce fait, on peut réaliser sur une image, comme sur n'importe quel autre signal, une analyse fréquentielle et lui appliquer la transformée de Fourrier. On remarquera que ces calculs nécessitent une technique légèrement plus avancée que celle qui a été explicitée dans ce rapport étant donné que ce signal est en deux dimensions, alors que ceux que nous traitons ne sont qu'en une dimension.

Une image est composée d'un signal et de bruit. Ces deux parties sont portées par des fréquences différentes : le signal correspond aux basses fréquences alors que le bruit correspond aux hautes fréquences. Il est donc facile de séparer ces deux parties grâce à un filtre et de produire ainsi un effet intéressant sur l'image.

Le filtre parfait réagit en réponse à une fréquence idéale, c'est-à-dire qu'il laisse passer toutes les fréquences inférieures ou supérieures à celle-ci, ce qui n'est possible que lorsque le signal a auparavant été décomposé en fonctions simples des fréquences. En pratique, il faut également délimiter une marge d'erreur autorisée à partir de laquelle les fréquences peuvent passer.

FIGURE 4.3 – Image originale

Prenons l'exemple d'un filtre passe-bas. Comme son nom l'indique, ce type de filtre ne laissent passer que les basses fréquences et bloque les hautes fréquences. On ne garde donc que le signal de l'image, et on supprime le bruit : on dit alors que l'image est "débruitée". La suppression du bruit provoque donne un effet de flou à l'image.

Un filtre passe-haut, au contraire, va "bruiter" l'image en laissant passer les hautes fréquentes, qui correspondent au bruit et pas les basses fréquences, correspondant au signal. Ce filtre a été appliquée à l'image originale de l'exemple du filtre passe-bas.

Une autre application importante de la transformée de Fourier discrète au traitement de l'image est la compression d'image. Le but de la compression est de représenter le signal visuel de l'image le plus efficacement possible c'est-à-dire en utilisant un petit nombre de composantes tout en gardant un maximum d'in-

Application du filtre passe-bas

Application du filtre passe-haut

Source : [http://webia.lip6.fr/~thomen/Teaching/BIMA/cours/Fourier_2011_1.pdf]

formation. Les fréquences les plus élevées sont celles qui contiennent le moins d'informations concernant l'image, c'est donc celle qui seront supprimées lors de la compressions. Il existe plusieurs types de compressions avec plus ou moins de pertes, c'est-à-dire que certains types de compressions vont supprimer plus de composantes que d'autre. Cette technique de compression est utilisée notamment pour obtenir des images de format JPEG.

5 Conclusion

Bilan du travail Ce projet nous a permis de découvrir le Transformée de Fourier Discrète, qui, comme nous avons pu le voir, possède plusieurs applications intéressantes. Nous avons également eu un aperçu des autres transformées de Fourier, bien que nous ne les ayons pas étudiées en détails.

Ce projet a été très interessant du point de vue mathématique. La compréhension et la démonstration de l'agorithme Fast Fourier Transform ont resprésentées une partie importante de notre projet, qui nous a poussée à réfléchir en groupe et à partager notre analyse. Nous avons également pu constater que le lien entre l'informatique et les mathématiques est très fort. De plus, nous avons constaté qu'il est impossible de créer un programme utilisant un algorithme que l'on ne comprend pas mathématiquement.

Nous avons également pu vérifier l'importance du temps de calcul informatique et les limites de ces calculs en comparant des résultats informatiques à des résultats obtenus manuellement ou grâce au logiciel de calcul formel Maple, ce que nous n'avions pas eu l'occasion de faire en dehors du cadre de ce projet.

Perspectives Le but de l'algorithme FFT étant d'optimiser le temps de calcul, il pourrait être interessant de ré-écrire notre programme dans un autre language. Bien que les calculs effectués en Pascal soient obtenus en un temps qui nous paraît instantanné, l'utilisation d'un langage plus bas-niveau (tel que le C, voir l'Assembleur) pourrait permettre certaines optimisations, autant au niveau du temps de calcul que sur l'utilisation des ressources.

6 Bibliographie

Cours (texte et vidéo)

- *Base du traitement des images, Transformée de Fourier*
  [http://webia.lip6.fr/~thomen/Teaching/BIMA/cours/Fourier_2011_1.pdf]
- *DFT/FFT Transforms and Applications*
  [http://abut.sdsu.edu/TE302/Chap6.pdf]
- *Fourier Analysis and Applications*
  [http://ocw.nctu.edu.tw/course_detail.php?bgid=1&gid=1&nid=13#.UMqWVjkkWQs]
- *Linear systems and optimization — The fourier transform and its applications*
  [http://see.stanford.edu/see/lecturelist.aspx?coll=84d174c2-d74f-493d-92ae-c3f45c0ee091]
- *Mathematics Of The Discrete Fourier Transform (DFT) With Audio Applications*
  [https://ccrma.stanford.edu/~jos/log/]

Documents

- *Application de la transformation rapide de Fourier au filtrage d'images*
  [http://gilemon.free.fr/site-rapport/rapportfiltrage.htm]
- *Discrete Fourier transform*
  [http://en.wikipedia.org/wiki/Discrete_Fourier_transform]
- *Document fourni par Monsieur Gleyse*
- *La transformée de Fourier et ses applications (partie 1)*
  [http://www.techniques-ingenieur.fr/base-documentaire/sciences-fondamentales-th8/applications-des-mathematiques-42102210/la-transformee-de-fourier-et-ses-applications-partie-1-af1440/echantillonnage-transformee-en-z-et-filtrage-numerique-af1440niv10002.html#2.3]