Université Pierre et Marie Curie **Master M2 "IFMA"**
Programmation en C++ **D. Simon**

Projet : traitement du signal et puissance des puissances de deux

Tous les fichiers sont à envoyer à damien.simon@upmc.fr **avant le 31 janvier 2014** (dernier délai). Vous êtes invités à envoyer tout votre code et vos fichiers et également un fichier auxiliaire (en .pdf) avec vos commentaires et les réponses aux questions qui ne sont pas directement du code informatique.

Table des matières

1 Buts et outils

1.1 Le cadre de travail

1.1.1 Contenu du projet

Les programmes informatiques permettent de traiter d'énormes quantités données lorsque ces dernières ont une structure qui se prête facilement à un traitement algorithmique. Des exemples classiques sont les transformées de Fourier pour réaliser une analyse en fréquence d'un signal ou les transformées en ondelettes pour compresser des images (formats JPEG2000 par exemple). Étant données la résolution et la précision de la plupart des enregistreurs de signaux actuels, il est essentiel que de tels algorithmes puissent être programmés aussi efficacement que possible. Pour cela, il faut utiliser de nombreuses astuces mathématiques.

La transformée de Fourier peut être énormément accélérée lorsque le nombre de points du signal est une puissance de deux, comme nous allons le voir ci-dessous ! Ainsi, pour analyser en fréquence un signal de 1890 points, la solution la plus simple est de le compléter (intelligemment !) à $2048 = 2^1 1$ (ou de le tronquer à 1024) points puis de ne faire qu'ensuite la transformée de Fourier.

Le but de ce projet est non seulement de programmer les transformées de Fourier et en ondelettes de Haar mais également de bâtir l'environnement nécessaire pour lire et écrire les données dans un format donné.

Le projet s'articule de la manière suivante :

— élaboration de classes génériques qui permettent de coder de différentes manières un signal, de le lire et de l'écrire dans un fichier suivant un format donné,

— programmation des transformations mathématiques en prenant soin d'être optimal à la fois en temps et en précision numérique, sensibilisation à la notion de complexité,
— test de performance des programmes sur des données préfabriquées.

1.1.2 Complexité d'un algorithme et temps de calcul

Une même tâche peut souvent être réalisée par différents algorithmes sans que ceux-ci ne soient équivalents en termes de temps de calcul. Il est évident que l'addition prend moins de temps qu'une multiplication qui prend elle-même moins de temps qu'une racine carrée ou une fonction trigonométrique. La marge de manœuvre du programmeur reste réduite puisque le choix des opérations est en général imposé par la nature mathématique du calcul à réaliser ; néanmoins, lorsque l'on se rend compte que la même quantité est calculée à de nombreuses reprises, il vaut mieux la calculer une fois pour toutes et la stocker dans une variable.

La complexité d'un algorithme est le nombre d'opérations à réaliser pour traiter des données de taille N (nombre de points, taille d'un tableau, d'un vecteur, d'une matrice) lorsque N est grand. Ce qui est important ici est moins la nature des opérations à réaliser pour chaque point que le nombre de fois où il faut parcourir les données. Par exemple, calculer la somme des éléments d'un tableau de taille N a une complexité $O(N)$, calculer la somme des carrés de ces éléments à également une complexité $O(N)$. En revanche, trier un tel tableau peut être fait en $O(N^2)$ (trouver le minimum et l'isoler, trouver ensuite le minimum des restants et l'isoler, etc) mais peut être amélioré en $O(N \log N)$! Le comportement asymptotique en $O(N^\alpha)$ est vraiment l'information importante : entre deux programmes qui se comportent comme $10^6 N$ et $4N^2$, c'est toujours le premier qui gagne dès que N est assez grand car, en pratique, les N considérés sont en général énormes.

Ainsi, pour bien coder un algorithme, on commencera toujours par évaluer la complexité sans se soucier des détails : pour cela, il suffit en général de compter le nombre d'imbrications de boucles `for` et de multiplier les nombres d'itérations correspondantes ! Il faut toujours réduire ce nombre le plus possible : c'est en général un question d'astuces mathématiques. Ce n'est qu'ensuite que l'on travaille boucle par boucle en essayant de refaire le moins possible les mêmes calculs.

1.1.3 Lecture et écriture de données

La lecture et l'écriture des données est toujours une question sensible. En effet, il faut toujours penser que c'est un programme informatique (un autre ou le même) qui a produit ou va ensuite utiliser les données qu'on manipule. Il faut donc être très rigoureux dans le codage des entrées-sorties : choix de points, virgules ou points-virgules, insertion d'espaces ou passage à la ligne, choix de parenthèses ou de crochets, etc.

Le choix des formats de données est rarement laissé au programmeur puisqu'en général on utilise des standards déjà imposés. Ainsi, dans ce projet, les fichiers de données à lire sont déjà fournis dans un format décrit ci-dessous et *il vous incombe de vous y conformer.*

1.2 Transformations de fonctions

1.2.1 Théorie générale et transformation de Fourier usuelle

Les transformations de fonctions usuelles de fonctions sont avant tout des formules de changements de base. Un signal sera pour nous une fonction $f : \{0, 1, \ldots, N-1\} \to \mathbb{R}$ ou $\mathbb{C}$. La base naturelle est celle des fonctions δ_k définies par $\delta_k(l) = 1$ si $l = k$ et 0 sinon. On a ainsi la représentation usuelle

$$f = \sum_{k=0}^{N-1} f(k)\delta_k \tag{1}$$

L'espace des fonctions $\mathbb{C}^{\{0,\ldots,N-1\}}$ est de dimension N et est muni du produit hermitien $\langle f, g\rangle = \sum_k \overline{f(k)}g(k)$. Soit $(u_k)_{0\leq k\leq N-1}$ une autre famille orthonormée ; on a ainsi $f = \sum_l f'_l u_l$ avec des coefficients f'_l définis par

$$f'_l = \sum_{k=0}^{N-1} f_k \langle u_l, e_k\rangle \tag{2}$$

Dans la suite, nous considérons la transformée de Fourier qui correspond à la famille de fonctions orthogonales $e_k(l) = \exp(2i\pi kl/N)/\sqrt{N}$ et la transformation en ondelettes de Haar définies ci-dessous.

D'un point de vue informatique, nous avons besoin d'une classe `SignalComplexe` qui code la description d'une fonction dans l'une des bases et d'une classe `Transformation` d'objets fonctionnels qui calcule les nouveaux coefficients dans une autre base et renvoie ainsi un autre objet de type `SignalComplexe`. En termes de complexité, une telle transformation est *a priori* de complexité $O(N^2)$ puisqu'il faut calculer N coefficients f'_l et chaque coefficient requiert une somme de N termes sur les coefficients initiaux.

Nous allons voir ci-dessous qu'il existe deux cas où ces calculs peuvent être grandement accélérés en utilisant astucieusement la base 2.

1.2.2 La transformée de Fourier rapide

Nous réservons la notation $\widehat{f}(k)$ pour les coefficients de Fourier d'une fonction et nous choisirons la normalisation suivante :

$$\widehat{f}(k) = \sum_{l=0}^{N-1} f(l) \exp\left(\frac{-2i\pi kl}{N}\right) \tag{3a}$$

$$f(l) = \frac{1}{N} \sum_{k=0}^{N-1} \widehat{f}(k) \exp\left(\frac{2i\pi kl}{N}\right) \tag{3b}$$

Supposons que $N = 2^P$ avec $P \in \mathbb{N}$ et décomposons la somme de l'équation (3) selon les indices pairs $l = 2l_1$ et impairs $l = 2l_1 + 1$. Il vient alors :

$$\begin{aligned}\widehat{f}(k) &= \left(\sum_{l_1=0}^{2^{P-1}} f(2l_1) \exp\left(2i\pi \frac{kl_1}{2^{P-1}}\right)\right) + \left(\sum_{l_1=0}^{2^{P-1}} f(2l_1+1) \exp\left(2i\pi \frac{kl_1}{2^{P-1}}\right)\right) \exp\left(2i\pi \frac{k}{2^P}\right) \\ &= A_{P-1}(k) + B_{P-1}(k) \exp\left(2i\pi \frac{k}{2^P}\right)\end{aligned}$$

Faisons alors deux remarques-clefs :

1. périodicité : chaque terme $A_{P-1}(k)$ apparaît dans deux $\widehat{f}(k)$ puisque $A_{P-1}(k) = A_{P-1}(k + 2^{P-1})$ pour $0 \leq k < 2^{P-1}$. Il n'y ainsi que 2^P coefficients indépendants et non 2^{P+1}.
2. récurrence : chaque terme $A_{P-1}(k)$ ou $B_{P-1}(k)$ est la transformée de Fourier d'un signal de taille moitié $N/2 = 2^{P-1}$;

Un rapide calcul montre que cette récurrence permet un algorithme en complexité $O(NP) = O(N \log N)$ puisque la formule précédente ne nécessite qu'une addition à partir des coefficients précédents et on obtient ainsi un gain énorme.

Prenons l'exemple $N = 8$ et poursuivons la récurrence. Notons $\omega_n = \exp(2i\pi/n)$ la n-ème racine de l'unité. On a ainsi

$$\begin{aligned}\widehat{f}(k) = &\left(\left(f(0) + f(4)\omega_2^k\right) + \left(f(2) + f(6)\omega_2^k\right)\omega_4^k\right) \\ &+ \left(\left(f(1) + f(5)\omega_2^k\right) + \left(f(3) + f(7)\omega_2^k\right)\omega_4^k\right)\omega_8^k\end{aligned}$$

On voit ainsi que le calcul s'organise bien à condition de réordonner les $f(l)$ initiaux.

Exercice 1. *Écrire en base* 2 *les nombres* 0 *à* 7 *et comprendre quel est l'ordre caché derrière la suite* $(0, 4, 2, 6, 1, 5, 3, 7)$.

En fait, $e^{2i\pi k/2}$ ne prend que deux valeurs possibles selon la parité de k, $e^{2i\pi k/4}$ ne prend que 4 valeurs possibles, etc. Chaque terme $e^{2i\pi k/2^p}$ ne dépend donc que du début de l'écriture de k en base 2 ! Il s'agit donc de calculer une seule fois chaque terme pour chaque sous-partie du tableau.

1.2.3 La transformation en ondelettes de Haar

On définit la fonction réelle $\psi : \mathbb{R} \to \mathbb{R}$ par :

$$\psi(x) = \begin{cases} 1 & \text{si } 0 \leq x < 1 \\ -1 & \text{si } 1 \leq x < 2 \\ 0 & \text{sinon} \end{cases} \tag{4}$$

On considère les dilatées et translatées

$$\psi_{n,k}(x) = 2^{n/2}\psi(2^n(x+k))$$

de la fonction ψ. Alors la théorie des ondelettes dit que cette famille est une base de Hilbert de l'espace $L^2(\mathbb{R})$. Un signal dont le support est contenu $[0, 2^N - 1]$ et qui est constant sur chaque $[k, k+1[$ est caractérisé soit par :

— soit par les 2^N valeurs sur chaque intervalle $[k, k+1[$ (codage `'p'` défini plus bas)
— soit par ses coefficients dans la base des $\psi_{n,k}$ dont on peut montrer qu'ils sont tous nuls sauf précisément 2^N d'entre eux.

On a ainsi un changement de base orthonormée qui définit la transformation en ondelettes de Haar.

1.2.4 La transformation binomiale

Soit une suite $(a_n)_{0\leq n\leq N}$ (signal de taille $N+1$). La transformation binomiale de cette suite est la suite $s = T(a)$ définie, pour tout $n \in \{0, 1, \ldots, N\}$ par

$$s_n = \sum_{k=0}^{n} (-1)^k C_n^k a_k$$

Exercice 2. *Montrer que cette tranformation est une involution.*

2 Codage informatique

2.1 Structure générale et `Makefile`

Nous allons écrire les fichiers suivants :
- des fichiers `signal.hpp` et `signal.cpp` pour gérer stockage, lecture et écriture des signaux
- des fichiers `transformation.hpp` et `transformation.cpp` qui dépendent de `signal.hpp` pour coder les transformations précédentes.
- un fichier `test1.cpp` qui dépend des précédents et permet de les tester sur des fichiers de données prédéfinis. L'exécutable correspondant sera `test1.exe`.
- un fichier `signauxvaries.hpp` et un fichier `signauxvaries.cpp` qui dépendent de `signal.hpp`.
- un fichier `test2.cpp` qui dépend de tous les précédents et réalise les tests de la dernière section dans un exécutable appelé `test2.exe`

Exercice 3 (facultatif)**.** *Écrire le* `Makefile` *correspondant.*

2.2 Gestion des signaux

On souhaite introduire deux classes `SignalReel` et `SignalComplexe` qui permettent de stocker et manipuler un signal. On se référera à la documentation pour savoir quelles sont les opérations licites sur les nombres complexes codés en précision `double`.

Il est interdit de passer à la section suivante tant que les tests de vos classes `SignalReel` et `SignalComplexe` ne reproduisent pas *exactement* les fichiers de données qui vous sont demandés.

```
#ifndef SIGNAUX
#define SIGNAUX
class SignalReel {
        private:
                unsigned int N;
                std::vector<double> t;
                char codage;
        public:
                unsigned int lecture(std::istream & i);
                void ecriture(std::ostream & o) const;
                double & operator[](unsigned int k) ;
                double operator[](unsigned int k) const;
};

class SignalComplexe {
        private:
                unsigned int N;
                std::vector<std::complex<double> > t;
                char codage;
        public:
                SignalComplexe(const SignalReel &);
                unsigned int lecture(std::istream & i);
                void ecriture(std::ostream & o) const;
                std::complex<double> & operator[](unsigned int k) ;
                std::complex<double> operator[](unsigned int k) const;
};
#endif
```

(nom du fichier : `projet-codes/signal.hpp`*)*

Le constructeur `SignalComplexe(const SignalReel &)` bâtit canoniquement un signal complexe à partir d'un signal réel en mettant les parties imaginaires à zéro.

Le codage pourra prendre ensuite les valeurs `'p'` pour "points", `'f'` pour "Fourier", `'h'` pour "Haar", `'b'` pour "Binomiale", qui indique dans quelle base les coefficients devront être interpréter, ou encore tout autre lettre pour les codages que nous souhaiterons ajouter par la suite.

Exercice 4. *Inclure les bibliothèques nécessaires en début du fichier* `.hpp`.
Écrire directement les mutateurs et accesseurs dans les classes.
Ajouter dans ces classes les constructeurs et destructeur nécessaires et écrire leurs codes dans les fichiers `.cpp`

Le format de lecture/écriture dans un fichier devra absolument suivre le schéma suivant pour les signaux réels

```
#   la_valeur_de_c   le_nombre_de_points
t[0]
t[1]
t[2]
...
t[N-1]
```

(nom du fichier : `projet-codes/schema-donnees.txt`*)*

et, pour les signaux complexes,

```
#   la_valeur_de_c   le_nombre_de_points
Re(t[0])         Im(t[0])
Re(t[1])         Im(t[1])
Re(t[2])         Im(t[2])
...
Re(t[N-1])       Im(t[N-1])
```

(nom du fichier : `projet-codes/schema-donnees-c.txt`*)*

On remarquera l'absence de virgules comme séparateur ainsi que la présence du croisillon (#) sur la première ligne.

Exercice 5. *Écrire les fonctions de lecture et d'écriture et les tester sur le fichier*[1] `donnees.dat` *en le lisant puis l'écrivant sur un autre fichier puis en comparant que les deux sont bien identiques.*

2.3 Transformations

Nous souhaitons ici écrire une classe générale qui permettent de changer de représentations pour un signal complexe donné. Nous proposons la structure mère suivante :

```
class Transformation {
   public:
      virtual void DirectTransform(const SignalComplexe & Input,
                                   SignalComplexe & Output) = 0;
      virtual void DirectTransformInplace(SignalComplexe & IOput) =0;
      virtual void InverseTransform(const SignalComplexe & Input,
                                   SignalComplexe & Output) =0;
      virtual void InverseTransformInplace(SignalComplexe & IOput) =0;
};
```

(nom du fichier : `projet-codes/principe-transfo.hpp`*)*

Les versions `Inplace` prennent un signal, le transforme et remplace le signal d'origine alors que les autres versions transforment le signal `Input` et mettent le résultat dans le signal `Output`.

2.3.1 Transformation de Fourier

Exercice 6. *Écrire une classe* `Fourier` *qui hérite de* `Transformation` *et réalise la transformation de Fourier usuelle (pas la rapide !).*

1. disponible sur ma page web.

Exercice 7. *La tester sur le signal écrit dans le fichier* `donnees.dat` *et vérifier que les résultats coïncident avec les résultats du fichier* `donnees-TF.dat`.

On souhaite à présent réaliser la transformation de Fourier rapide. **Attention : ce n'est pas si facile et on essaiera d'abord de bien comprendre ce que doit faire le programme à partir de la récurrence présentée dans l'introduction.**

Exercice 8. *(plus difficile) Écrire une classe* `FFT` *qui hérite de* `Transformation` *et réalise la transformation de Fourier rapide lorsque la taille du signal d'origine est une puissance de deux (si ce n'en est pas une, elle doit faire utiliser la classe précédente).*

On pourra également se référer à l'appendice qui explique comment manipuler les nombres entiers en écriture binaire directement et commencer par faire l'exercice 19 puis ajouter une méthode privée dans `FFT` *qui réordonne un tableau selon le bon ordre binaire expliqué dans l'introduction.*

Exercice 9. *La tester également sur le signal écrit dans le fichier* `donnees.dat` *et vérifier que les résultats coïncident avec les résultats du fichier* `donnees-TF.dat`.

Exercice 10. *Comparer les durées mises par la transformée usuelle et la transformée rapide sur le fichier* `donnees-gros.dat`. *Commenter.*

2.3.2 Transformation de Haar (facultatif)

Exercice 11. *(facultatif) Écrire une classe* `Haar` *qui hérite de* `Transformation` *et réalise la transformation en ondelettes de Haar lorsque la taille du signal d'origine est une puissance de deux.*

Essayer de le faire avec une complexité $O(N \log N)$.

Tester le programme sur des données de test.

2.3.3 Transformation binomiale

Exercice 12. *Écrire une classe* `Binomiale` *qui hérite de* `Binomiale` *et réalise la transformation binomiale.*

Afin de limiter le temps de calcul et les erreurs numériques, il faut ne calculer aucune *factorielle et utiliser en revanche la structure récursive du triangle de Pascal pour le calcul des coefficients binomiaux.*

Tester le programme sur des données de test.

3 Production

3.1 Interface

Exercice 13. *Écrire une fonction* `main` *dans le fichier* `test1.cpp` *qui*

- *demande le nom du fichier initial à charger*
- *demande le nouveau codage du signal (en le proposant parmi les possibles)*
- *demande le nom du fichier de sortie*
- *détecte le codage d'origine, réalise la conversion du signal (si nécessaire) par le moyen le plus rapide possible et l'écrive dans le fichier de sortie.*
- *annonce dans le terminal combien de temps a été mis pour la conversion*
- *propose de revenir au point de départ ou de quitter le programme.*

3.2 Test sur des signaux aléatoires

Pour éviter d'avoir à créer des signaux à la main et avoir une batterie de tests plus riche, on souhaite créer des signaux de deux types, aléatoires et déterministes. Toutes les questions suivantes, sauf spécification contraire, seront réalisées dans les fichiers `signauxvaries.hpp` et `signauxvaries.cpp`.

Exercice 14. *Créer une classe* `SignalReelDeterministe` *qui hérite de* `SignalReel` *et qui contient deux constructeurs qui prennent comme arguments un entier et une fonction qui transforme des* `double`/`unsigned` *en* `double`. *Faire de même pour* `SignalComplexe` mutatis mutandis. *Dans les deux cas, les valeurs du signal seront les valeurs* $f(0)$, $f(1)$, *etc.*

Exercice 15. *Calculer les transformées de Fourier de la fonction* sin *sur* $[0, 1]$ *(discrétisée en* 4096 *points) ainsi que de* δ_0 *et vérifier qu'on obtient bien les résultats escomptés.*

On souhaite à présent générer des signaux aléatoires : il faut donc ajouter une méthode de prototype `void generate(void)` qui permet de régénérer le signal en gardant la taille et le codage constants.

Exercice 16. *Créer une classe* `SignalReelRandom` *qui hérite de* `SignalReel` *qui contienne une méthode virtuelle pure* `generate()` *comme expliquée précédemment.*

Exercice 17. *Créer une classe* `MvtBrownien` *qui hérite de* `SignalReelRandom` *et permettent de générer un mouvement brownien échantilloné sur N points avec un pas de temps de δ. Créer une classe* `PontBrownien` *qui hérite elle aussi de* `SignalReelRandom` *et génère un pont brownien de longueur fixée. Calculer la transformée de Fourier de ce dernier.*

Exercice 18. *Créer un fichier* `test2.cpp` *qui permette de tester ces différentes situations et écrivent les données dans des fichiers que nous pourrons ensuite visualiser.*

A Codage binaire des entiers et opérateurs logiques

Un entier `unsigned int` correspond à un entier avec des valeurs entre 0 et $2^K - 1$ où K est une précision qui dépend de la machine, du compilateur, etc. On peut accéder à sa valeur grâce à la bibliothèque `limits` et la commande

```
std::numeric_limits<unsigned int>::max()
```

.

Son écriture en base deux est donc de la forme $\sum_{k=0}^{K-1} b_k 2^k$ avec $b_k \in \{0,1\}$ que l'on écrit plus communémemnt $b_K b_{K-1} \dots b_1 b_0$ ou même $b_{k_0} \dots b_1 b_0$ où k_0 est le premier bit qui vaut 1 (i.e. on n'écrit pas les premiers zéros). Une fois ce codage compris, on remarque que l'on peut percevoir cette suite de bits comme une suite de booléens auxquels on a envie d'appliquer les opérateurs logiques usuels. On peut ainsi, pour des variables `n` et `m` de type `unsigned int` pour une machine où $K = 8$ qui vaudrait `n=45=00101101` et `m=75=11001011`, utiliser les opérateurs suivants :

- négation : `~n=11010010`$= 210 = (2^8 - 1) - n$`=11111111-n`,
- "et" logique : `n& m = 00001001`$= 9$,
- "ou" logique : `n|m = 11101111`$= 239$,
- "ou" exclusif logique : `n^m = 11100110`$= 230$,
- décalage vers la gauche `n << 1=01011010`$= 2n$ et `n<< 2=10110100`$= 4n$ et plus généralement `n <<p`$= 2^p n$ (modulo la valeur maximale d'un entier),
- décalage vers la droite `n << 1=01011010`$= n/2$ (division euclidienne).

Exercice 19. *En utilisant ces opérateurs binaires, écrire une fonction*

```
unsigned int conjugate(unsigned poweroftwo,unsigned int n)
```

telle que si `poweroftwo` *est une puissance de deux (disons* 2^M*) et si* `n` *est strictement plus petit que* `poweroftwo` *qui s'écrit* $b_{M-1} \dots b_1 b_0$ *alors* `conjugate(n)` *est l'entier qui s'écrit* $b_0 b_1 \dots b_{M-1}$.

B Utilisation de `gnuplot` pour visualiser les résultats

Pour visualiser les signaux écrits dans un fichier, on pourra utiliser le logiciel `gnuplot` en écrivant dans le terminal

```
gnuplot
```

puis, à l'intérieur de ce logiciel, on peut représenter la k-ème colonne du fichier `donnees.dat` avec la commande

```
plot 'donnees.dat' using k with lines
```