Université de Strasbourg

UFR Mathématique Informatique Strasbourg

Thuet
Quentin

Rapport

Projet Informatique S6

Traitement du signal

Sommaire :

Descriptif détaillé du code et résultats des tests unitaires

Exemple de descriptif d'une fonction

- Nom de la fonction

`Signature`

Brève description de la fonction.

1. Description des différentes parties du code, repérées par les balises correspondantes (Ex : on retrouvera ici le code correspondant entre la balise ouvrante *//1* et la balise fermante *//*).
2. ..

Test (*testu_1() - //3*) : Description du test réalisé. On retrouvera le code correspondant à l'endroit indiqué entre parenthèse (Ex : on retrouvera ici le code correspondant dans la fonction *testu_1()* de la classe, entre la balise ouvrante *//3* et la balise fermante *//*).

Remarques

- Quand je parle de « i^{ème} valeur », il s'agit bien du i^{ème} rang du tableau à partir de 0.

Signal discret

Constructeurs

- Constructeur par défaut

`signal_discret()`

Constructeur par défaut en initialisant la taille à 0 et en faisant pointer le tableau vers NULL.
Test (*testu_1() - //2*) : On crée simplement le signal discret `sd2` défaut. On peut vérifier que le constructeur est bien exécuté en affichant *SD/ constructeur défaut* dans le code.

- Constructeur par taille

`signal_discret(int)`

Constructeur qui prend en paramètre la taille du signal souhaité et qui remplit le tableau avec des 0.
Test (*testu_1() - //1*) : On crée le signal discret `sd1` de taille 5 que l'on utilisera dans les tests suivants. On peut vérifier que le constructeur est bien exécuté en affichant *SD/ constructeur taille* dans le code.

- Constructeur par copie

`signal_discret(const signal_discret &)`

Constructeur par copie qui affecte la même taille que l'objet copié à l'objet créé, et copie les valeurs du signal une à une. On différencie le cas où le tableau copié est un tableau vide.
Test (*testu_1() - //3*) : On crée les signaux discrets `sd3` et `sd4`, respectivement par copie d'un signal non vide `sd1` et par copie d'un signal vide `sd2`. On peut vérifier que le constructeur est bien exécuté en affichant *SD/ constructeur copie* dans le code.

Destructeur

- Destructeur

`~signal_discret()`

Détruit le tableau de complexe.
Test : On peut vérifier que le destructeur est bien exécuté en affichant *SD/ destructeur* dans le code.

Accesseurs

- Accesseur de taille

`int get_size()`

Accesseur qui renvoie la taille du signal.
Test (*testu_1() - //4*) : On vérifie que la valeur de la taille du signal `sd1` précédemment créé est bien 5.

- Accesseur de valeur

`int get_value(int)`

Accesseur qui renvoie la i^{ème} valeur du signal, i étant entré en paramètre. La fonction renvoie 0 si la valeur demandée est en dehors du signal.
Test (*testu_1() - //5* et *//6*) : On vérifie que la 3ème valeur de `sd1` renvoyée est la bonne, avant et après modification.

Mutateur

- Mutateur de valeur

`void set_value(int, complexe)`

Mutateur qui remplace la i^{ème} valeur du signal, *i* étant entré en paramètre, par un complexe *c* également entré en paramètre. La fonction rallonge le tableau jusqu'à la valeur i demandé si cette valeur est plus grande que la taille du tableau.

1. Si *i* est plus petit que la taille du tableau, on remplace simplement la i^{ème} valeur du tableau par c.
2. Sinon, on récupère les valeurs du signal dans un tableau de complexe temporaire.
3. On détruit le tableau de complexes du signal pour le réallouer à la bonne taille.
4. On remplit le nouveau tableau de valeurs du signal par les valeurs précédentes grâce au tableau temporaire, puis on remplit le tableau de 0 jusqu'à atteindre la i^{ème} valeur, à laquelle on affecte *c*.
5. On met à jour la taille du signal.

Test (*testu_1() - //6*) : On vérifie que la 3ème valeur de `sd1` renvoyée est la bonne, avant et après modification, grâce à l'accesseur précédemment introduit. On vérifie également que la taille du tableau est bien modifiée si on modifie une valeur qui n'était pas dans le tableau.

Méthodes élémentaires

- Conjugué

`signal_discret conjugue()`

Renvoie le signal conjugué de celui à partir duquel est appelé la méthode. La méthode stocke le conjugué de chaque valeur du signal dans une variable res qui est finalement renvoyée.
Test (*testu_2() - //5*) : On stocke le conjugué de `sd1` dans `sd3` et on vérifie que la valeur 5+6i a bien été changée en 5-6i.

```
sd1 = {0,0,0,5+6i,0}
sd3 = {0,0,0,5-6i,0}
```

- Energie

`double energie()`

Renvoie l'énergie du signal.
Test (*testu_2() - //5*) : On calcule l'énergie de sd1 et on vérifie qu'elle vaut bien 87 (aux erreurs machines près).

```
sd1 = {0,1-5i,0,5+6i,0}
```

Méthodes avancées

- Transformée de Fourier

`complexe * tfd()`

Renvoie la transformée de Fourier discrète du signal, sous forme d'un tableau de complexe de même taille que le signal.

1. On fait une première boucle ($k = 0, \dots, N-1$, N étant la taille du signal x) qui nous permet de calculer chaque terme $X(k)$ un à un selon la formule $X(k) = \sum_{n=0}^{N-1} x(k)\, e^{-2i\pi n\frac{k}{N}}$. On commence par initialiser ce terme à 0.
2. On fait ensuite une seconde boucle ($n = 0, \dots, N-1$) pour calculer le terme. A chaque itération, on commence par initialiser un complexe qui correspond au $e^{-2i\pi n\frac{k}{N}}$, puis on ajoute son produit avec $x(k)$ à la précédente itération.

Test (*testu_3() - //1*) : On calcule la transformée de Fourier du signal sd1 et on vérifie qu'elle correspond bien au résultat attendu.

```
sd1 = {1-2i,-3+4i,5-6i,-7+8i}
tfd = {-4i+4,-8,16-20i,8i}
```

- Convolution

`signal_discret convolution(const signal_discret &)`

Calcul du produit de convolution entre le signal discret courant et un second entré comme paramètre.

1. On commence par récupérer la taille du plus grand des deux tableaux.
2. On crée le signal contenant le résultat, qui sera de même taille que le plus grand des deux en entrée.
3. On va calculer successivement les termes du signal résultat en appliquant la formule : $z(n) = \sum_{k=0}^{N-1} x(k)y(n-k)$ (où N est la taille du plus grand signal). On commence donc une boucle sur n et une sur k.
4. On calcule la somme en ajoutant $x(k)y(n-k)$ aux k premiers termes. On prend garde de ne pas calculer des termes qui sortent du signal, c'est-à-dire qu'on ne calcule pas les termes pour lesquels k est plus grand que la taille de x, $n-k$ plus grand que la taille de y, ainsi que ceux pour lesquels $n-k$ est négatif.

Test (*testu_1() - //3*) : On calcule le produit de convolution des deux signaux sda et sdb.

```
sda = {1+2i,3+4i}
sdb = {5+6i,7+8i}
sdc = {-7+16i,-16+60i}
```

- Translation

`signal_discret translation(const int)`

Renvoi le signal translaté selon le décalage entré en paramètre. La fonction renvoie un signal de même taille que celui depuis laquelle elle est appelée, donc tous les termes décalés vers un indice inférieur à 0 ou supérieur à la taille du signal sont considérés comme nuls.

1. Remplit le nouveau signal avec les valeurs translatées du signal de départ. Si l'on cherche à remplir le nouveau signal à partir d'un indice ne figurant pas dans le signal de départ, on le remplit par un 0.

Test (*testu_1() - //4*) : On calcule les translations de valeur 2 et de valeur -1 du signal sd6.

```
sd6 = {1+2i,3+4i,5+6i,7+8i}
sd7 = {0,0,1+2i,3+4i}
sd8 = {3+4i,5+6i,7+8i,0}
```

- Modulation

`signal_discret modulation(const double)`

Renvoie le signal modulé par le double entré en paramètre.

1. On calcule la modulation termes à termes, en multipliant chaque terme par le complexe temporaire $\cos(k_0) + i\ sin(k_0)$.

Test (*testu_1() - //5*) : On calcule le signal modulé issu de sd6.

Fonction externe à la classe

- Transformée de Fourier inverse

`signal_discret tfd_inverse(complexe *, const int)`

Calcul la transformée inverse d'une suite de complexes à partir d'un tableau de complexes et de sa taille.

1. On crée provisoirement un tableau de complexes pour stocker les valeurs du signal. Il est de même taille que le tableau en entrée.
2. On fait une première boucle ($k = 0, \dots, N-1$, N étant la taille du tableau en entrée X) qui nous permet de calculer chaque valeur du signal $x(k)$ un à un selon la formule $x(k) = \frac{1}{N}\sum_{k=0}^{N-1} X(k)\ e^{2i\pi n\frac{k}{N}}$. On commence par initialiser $x(k)$ à 0.
3. On fait ensuite une seconde boucle ($n = 0, \dots, N-1$) pour calculer le terme. A chaque itération, on commence par initialiser un complexe qui correspond au $e^{2i\pi n\frac{k}{N}}$, puis on ajoute son produit avec $x(k)$ à la précédente itération.
4. On multiplie finalement la valeur calculée par le quotient $\frac{1}{N}$.
5. Enfin, on recopie les valeurs du tableau provisoire dans le signal de taille N que l'on va retourner. On n'oublie pas de supprimer le tableau provisoire.

Test (*testu_3() - //2*) : On tente de retrouver le signal sd1 à partir de sa transformée de Fourier calculée précédemment.

```
sd1 = {1-2i,-3+4i,5-6i,-7+8i}
tfd = {-4i+4,-8,16-20i,8i}
```

Opérateurs

- Opérateur d'affectation

`signal_discret & operator=(const signal_discret &)`

Opérateur d'affectation, qui remplace la taille et les valeurs du signal de gauche par le signal de droite.

6. On test l'autoaffectation.
7. On supprime les précédentes valeurs du signal de gauche.
8. On recopie la taille et les valeurs du tableau de droites dans le tableau de gauche terme à termes.

Test (*testu_2() - //1*) : On crée un signal discret `sd1` de taille 5, dont la 3ème valeur est 5 + 6i. On crée un second signal discret `sd2` par défaut puis on lui affecte le premier via l'opérateur. On vérifie que la 3ème valeur est bien 5 + 6i.

```
sd1 = {0,0,0,5+6i,0}
sd2 = {0,0,0,5+6i,0}
```

- Opérateur d'addition

`signal_discret operator+(const signal_discret &)`

Opérateur d'addition qui somme deux signaux discrets en additionnant les tableaux de complexes termes à termes.

1. On stocke le min et le max des tailles pour savoir à partir de quel rang on considérera les coefficients du plus petit tableau comme nuls.
2. On additionne termes à termes les tableaux jusqu'à atteindre la taille du plus petit des deux.
3. Après avoir vérifié quel était le plus petit des deux tableaux, on remplit le tableau du résultat par les valeurs du plus grand.

Test (*testu_2 () - //2*) : On crée un signal discret `sd3` que l'on définit comme la somme de `sd1` et `sd2`, et on vérifie que la 3ème valeur de la somme de `sd1` et `sd2` vaut bien 10 + 12i.

```
sd1 = {0,0,0,5+6i,0}
sd2 = {0,0,0,5+6i,0}
sd3 = {0,0,0,10+12i,0}
```

- Opérateur de multiplication

`signal_discret operator*(const signal_discret &)`

Opérateur de multiplication similaire à celui d'addition, excepté que le nouveau signal est de même taille que le plus petit des deux multipliés (les coefficients suivants seraient nuls).

Test (*testu_2() - //3*) : On crée un signal discret `sd3` que l'on définit comme le produit de `sd1` et `sd2`, et on vérifie que la 3ème valeur du produit de `sd1` et `sd2` vaut bien -11 + 60i.

```
sd1 = {0,0,0,5+6i,0}
sd2 = {0,0,0,5+6i,0}
sd3 = {0,0,0,-11+60i,0}
```

- Opérateur de soustraction

`signal_discret operator-(const signal_discret &)`

Opérateur de soustraction strictement similaire à l'opérateur d'addition.

Test (*testu_2() - //4*) : On crée un signal discret `sd3` que l'on définit comme la différence de `sd1` et `sd2`, et on vérifie que la 3ème valeur de la différence de `sd1` et `sd2` vaut bien 0.

```
sd1 = {0,0,0,5+6i,0}
sd2 = {0,0,0,5+6i,0}
sd3 = {0,0,0,0,0}
```

- Opérateur de produit scalaire

`complexe operator,(const signal_discret &)`

Calcul du produit scalaire entre deux signaux complexes.

1. On récupère la taille minimale entre les deux tableaux.
2. On effectue le produit scalaire entre les deux tableaux. On arrête le calcul quand la taille minimale précédemment calculée est atteinte car les termes suivants de la somme seront nuls.

Test (*testu_2() - //7*) : On fait le produit scalaire entre `sd1` et `sd2` et on vérifie qu'il vaut bien 93 – 4i.

```
sd1 = {0,1-5i,0,5+6i,0}
sd2 = {0,2-6i,0,5+6i,0}
```

Testeurs

- Test unitaire 1

`int testu_1()`

Teste les constructeurs, le destructeur, les accesseurs et le mutateur, et renvoie 1 si les tests sont réussis, 0 sinon.

- Test unitaire 2

`int testu_2()`

Teste les opérateurs et les méthodes élémentaires, et renvoie 1 si les tests sont réussis, 0 sinon.

- Test unitaire 1

`int testu_3()`

Teste les méthodes avancées, et renvoie 1 si les tests sont réussis, 0 sinon.

- Testeur

`void all_testu()`

Affiche le résultat des trois tests unitaires.

Signal discret STL

Seules les fonctions présentant des modifications majeures par rapport à la classe précédente sont expliquées. Les tests sont rigoureusement les mêmes. La plupart des modifications consistent juste à récuperer la taille de `M_values` à l'aide de la méthode `size` plutôt que d'avoir la taille en attribut.

Constructeurs

- Constructeur par taille

`signal_discret_stl(int)`

On utilise la méthode `assign` qui nous permet de remplir directement `M_values` avec des 0.

Destructeur

- Destructeur

`~signal_discret_stl()`

On utilise la méthode `clear`.

Mutateur

- Mutateur de valeur

`void set_value(int, complexe)`

On ne réalloue pas la mémoire pour agrandir le tableau si besoin, mais on utilise la méthode `push_back`.

Fonction externe à la classe

- Transformée de Fourier inverse

`signal_discret_stl tfd_inverse_stl(complexe *, const int)`

On n'évite d'utiliser un tableau de complexes en accédant directement aux valeurs et en déclarant la fonction amie de la classe. En réalité on aurait aussi pu faire cette méthode avec la classe signal discret de base.

Opérateurs

- Opérateur d'affectation

`signal_discret_stl & operator=(const signal_discret_stl &)`
Ici on fait confiance à l'opérateur = de la classe `vector`.

- Opérateur d'affectation signal discret – signal discret STL

`signal_discret_stl & operator=(const signal_discret &)`
Il affecte un signal discret classique a un signal discret STL. Cela est nécessaire car la fonction échantillonnage renvoie un signal discret classique.

- Opérateur d'addition

`signal_discret_stl operator+(const signal_discret_stl &)`
On utilise à nouveau le `push_back` jusqu'à ce que le signal résultant soit aussi grand que le plus petit des deux signaux, puis on ajoute les valeurs restantes du plus grand des deux. Le fonctionnement est similaire pour l'opérateur – et pour l'opérateur *, mise à part que pour ce dernier, on s'arrête quand le signal résultant est aussi grand que le plus petit des deux signaux en entrée.

- Opérateur de produit scalaire

`complexe operator,(const signal_discret_stl &)`
Pour éviter une modification potentielle d'un argument constant, j'ai du stocké M_values dans une variable temporaire.

<u>Signal continu</u>

Constructeurs

- Constructeur par défaut

`signal_continu()`
Constructeur par défaut en initialisant l'intervalle de temps à $[0,1]$, le paramètre de variance de bruit $\varepsilon = 0,$ le parmètre de fenêtrage $s = 0$ et en faisant pointé le pointeur de fonction vers $NULL$.
Test (*testu_1() - //1*) : On crée simplement le signal continu `sc1` défaut. On peut vérifier que le constructeur est bien exécuté en affichant *SC/ constructeur défaut* dans le code.

- Constructeur par copie

`signal_continu(const signal_continu &)`
Constructeur par copie qui copie un à un les paramètres du signal entré en argument, et qui pointe vers la même fonction.
Test (*testu_1() - //2*) : On crée simplement le signal continu `sc2` par copie de `sc1`. On peut vérifier que le constructeur est bien exécuté en affichant *SC/ constructeur copie* dans le code.

- Constructeur par pointeur de fonction

`signal_continu(complexe (*)(double))`
Constructeur qui en argument un pointeur de fonction qui sera la base du signal continu.
Test (*testu_1() - //3*) : On crée le signal continu `sc3` à partir d'un pointeur sur une fonction entrée en argument du test unitaire. On peut vérifier que le constructeur est bien exécuté en affichant *SC/ constructeur pointeur sur fonction* dans le code. On vérifie aussi que la fonction stockée dans le signal continu est la même que celle entrée en argument du constructeur en comparant les évaluations en deux points.

Destructeur

- Destructeur

`~signal_continu()`
Il ne fait rien ici.
Test : On peut vérifier que le destructeur est bien exécuté en affichant *SC/ destructeur* dans le code.

Accesseurs

- Accesseurs des paramètres $t0, t1, \varepsilon, s$.

`double get_t0(), double get_t1(), double get_eps(), double get_s()`
Retournent respectivement les paramètres $t0, t1, \varepsilon, s$.
Test (*testu_1() - //4)* : On vérifie que les valeurs retournées sont bien celles qui ont été modifiées précédemment via les mutateurs.

Mutateurs

- Mutateurs des paramètres $t0, t1, \varepsilon, s, f$.

`void set_temps(double,double), void set_eps(double), void set_s(double), void set_f(complexe (*)(double))`
Permettent de modifier respectivement les paramètres $t0, t1, \varepsilon, s, f$.
Test (*testu_1() - //4)* : On vérifie que les valeurs retournées sont bien celles qui ont été modifiées via les mutateurs.

Méthode élémentaire

- Evaluation de la fonction en attribut

`complexe eval_f(double)`
Renvoie l'évaluation de la fonction en attribut, ce qui permet de tester si on a bien la fonction voulue en attribut d'un objet.
Test (*testu_1() - //3*) : On vérifie que la fonction marche bien en même temps que l'on vérifie la bonne affectation de la fonction `f_test` à notre objet.

Méthode avancée

- Echantillonnage

`signal_discret echantillonnage(int,bool)`
La fonction échantillonne le signal continu depuis laquelle elle est appelée. Le booléen en paramètre permet de choisir si l'on souhaite appliquer un fenêtrage.

1. On crée une variable r pour modifier plus facilement ce paramètre dans le fenêtrage.
2. On crée les variables qui nous seront utiles plus tard : le résultat de taille n entré en paramètre qui est le nombre d'échantillons, le temps courant à chaque itération, la valeur de l'échantillon à chaque itération et la valeur de la fonction de fenêtrage à chaque itération.
3. On initialise notre générateur de valeurs aléatoires.
4. On calcule le temps courant à chaque itération.
5. On initialise une distribution normale pour notre bruit, d'espérance la valeur de la fonction au temps courant et de variance le paramètre ε.
6. On calcule une valeur aléatoire à partir de notre distribution normale pour obtenir une valeur de notre signal bruitée.
7. Si un fenêtrage est souhaité, on calcule la valeur de la fonction fenêtrage au temps courant et on multiplie cette valeur à notre valeur du signal calculée précédemment.
8. On ajoute la valeur du signal bruité et éventuellement fenêtré au signal que l'on retournera.

Test (*testu_2()*) : Le résultat est présenté dans la partie « Résultats des tests de l'échantillonnage et des filtres ».

Testeurs

- Test unitaire 1

`int testu_1(complexe (*)(double))`
Tests des constructeurs, destructeur, accesseurs, mutateurs et de la méthode élémentaire.

- Test unitaire 2

`void testu_2(complexe (*)(double))`
Test de l'échantillonnage.

- Testeur

`void all_testu(complexe (*)(double))`
Affiche le résultat des deux tests unitaires.

Filtre

Toutes les classes sont des `template <class U>`.

Classe mère filtre

Etant donné que la classe est abstraite, on ne la testera pas directement, mais elle sera testée avec les tests des classes filles. Pour vérifier le bon appel des constructeurs et destructeurs, on peut afficher F/ constructeur … *ou* F/ destructeur *dans le code. Les accesseurs et mutateurs ne sont pas testés mais sont triviaux.*

Constructeurs

- Constructeur par défaut

`filtre()`
Constructeur par défaut qui met la durée du signal à 1 et la fréquence de coupure à 0.

- Constructeur par signal filtré, fréquence de coupure et durée

`filtre(U,double,double)`
Constructeur qui prend en paramètre le signal que l'on va filtrer, sa durée et la fréquence de coupure souhaitée.

- Constructeur par copie

`filtre(const filtre &)`
Constructeur par copie qui copie le signal que l'on va filtrer, le signal filtre, la durée du signal à filtrer et la fréquence de coupure.

Destructeur

- Destructeur

`~filtre()`
Il ne fait rien ici.

Accesseurs

- Accesseur de durée, de signal filtré, de signal filtre et de fréquence de coupure

`double get_duree(), U get_signal_apply(), U get_signal_filtre(), double get_freq_coupe()`

Renvoient respectivement la durée du signal à filtrer, le signal à filtrer, le signal filtre et la fréquence de coupure.

Mutateur

- Mutateur de durée, de signal filtré, de signal filtre et de fréquence de coupure

`void set_duree(double), void set_signal_apply(U), void set_signal_filtre(U), void set_freq_coupe(double)`

Permettent de modifier respectivement la durée du signal à filtrer, le signal à filtrer, le signal filtre et la fréquence de coupure.

Méthodes

- Application du filtre sur le signal filtré

`void apply()`

Fonction qui applique le filtre sur le signal à filtrer, en le modifiant directement.

1. On commence par créer toutes les variables nécessaires dans la suite. Tout d'abord on a besoin de récupérer la taille du signal à filtrer, que l'on stocke dans `size`. On a ensuite un tableau de complexes `tfd`, et un signal discret `spectre` dans lesquels on va stocker la transformée de Fourier discrète du signal à filtrer. De même, on a un tableau de complexes `values_spectre_filtre` et un signal discret `spectre_filtre` dans lesquels on stockera le spectre du signal après filtrage.
2. On commence par calculer le spectre du signal à filtrer. On le stocke dans un signal discret pour pouvoir appliquer le filtre plus facilement.
3. On applique le filtre en multipliant le spectre avec le signal filtre. On obtient un signal discret contenant le spectre du signal filtré.
4. On stocke ce spectre dans un tableau de complexes pour pouvoir appliquer la transformée de Fourier inverse.
5. On applique la transformée de Fourier inverse pour obtenir le signal filtré.

- Constructeur du filtre (méthode virtuelle pure)

`virtual void construct() = 0 ;`

Elle sera définie dans les classes filles.

Classe fille passe-bas idéal

Constructeurs

- Constructeurs

`passe_bas_ideal(), passe_bas_ideal(U,double,double), passe_bas_ideal(const passe_bas_ideal &)`

Ils dérivent directement de la classe mère.

Test (*test()*) : On peut vérifier qu'ils sont bien appelés en affichant *F/PBI/ constructeur …* dans le code.

Destructeur

- Destructeur

`~passe_bas_ideal()`
Il ne fait rien ici.
Test (*test()*) : On peut vérifier qu'il est bien appelé en affichant *F/PBI/ destructeur* dans le code.

Méthode

- Constructeur du filtre

`void construct()`
Construit le signal discret `signal_filtre` qui sera utiliser pour filtrer le signal à filtrer via la méthodes `apply`.

1. On commence par récupérer la taille du signal à filtrer.
2. On calcule le pas de fréquences. La variable `freq_p` à l'itération i contiendra ainsi la valeur de la fréquence à laquelle correspond la $i^{ème}$ valeur du signal filtre.
3. On itère sur la moitié du signal filtre uniquement, par symétrie du spectre que l'on va filtrer. A chaque itération, on regarde si la fréquence à laquelle correspond la $i^{ème}$ valeur du signal filtre est avant ou après la fréquence de coupure. Si elle est avant, on met 1 dans le signal filtre à ce rang, pour qu'il laisse cette fréquence du signal à filtrer intacte. Sinon, on met 0.0001 pour qu'il l'atténue fortement. On ne met pas 0 pour que les amplitudes logarithmiques du spectre soient définies. Par symétrie du spectre, on peut mettre la même valeur 0 ou 0.0001 à la valeur *taille – 1 – i* du signal filtre. On n'oublie pas d'incrémenter `freq_p`.

Test (*test()*) : Les résultats du tests sont présentés dans la partie « Résultats des tests de l'échantillonnage et des filtres ».

Test

- Test

`void test(complexe (*)(double))`
Test du filtrage passe-bas idéal.

Classe fille passe-haut idéal

Constructeurs

- Constructeurs

`passe_haut_ideal(), passe_haut_ideal(U,double,double), passe_haut_ideal(const passe_haut_ideal &)`
Ils dérivent directement de la classe mère.
Test (*test()*) : On peut vérifier qu'ils sont bien appelés en affichant *F/PHI/ constructeur …* dans le code.

Destructeur

- Destructeur

`~passe_haut_ideal()`
Il ne fait rien ici.
Test (*test()*) : On peut vérifier qu'il est bien appelé en affichant *F/PHI/ destructeur* dans le code.

Méthode

- Constructeur du filtre

`void construct()`

La méthode est similaire à celle du passe-bas idéal, sauf que les fréquences atténuées sont celles inférieures à la fréquence de coupure.

Test

- Test

`void test(complexe (*)(double))`

Test du filtrage passe-haut idéal.

Classe fille passe-bas 1er ordre

Constructeurs

- Constructeurs

`passe_bas_ordre_1(), passe_bas_ordre_1(U,double,double), passe_bas_ordre_1(const passe_bas_ordre_1 &)`

Ils dérivent directement de la classe mère.

Test (*test()*) : On peut vérifier qu'ils sont bien appelés en affichant *F/PBO1/ constructeur …* dans le code.

Destructeur

- Destructeur

`~passe_bas_ordre_1()`

Il ne fait rien ici.

Test (*test()*) : On peut vérifier qu'il est bien appelé en affichant *F/PBO1/ destructeur* dans le code.

Méthode

- Constructeur du filtre

`void construct()`

Construit le signal discret `signal_filtre` qui sera utiliser pour filtrer le signal à filtrer via la méthode `apply`.

1. On commence par récupérer la taille du signal à filtrer.
2. On calcule le pas de fréquences. La variable `freq_p` à l'itération i contiendra ainsi la valeur de la fréquence à laquelle correspond la i^{ème} valeur du signal filtre.
3. On calcule le module de la fonction de transfert d'un filtre passe-bas du premier ordre et on le stocke dans la variable complexe `temp` que l'on place ensuite dans le signal filtre. On a la fonction de transfert $H(f) = \frac{1}{1+i\frac{f}{f_c}}$ et son module $|H(f)| = \frac{1}{\sqrt{1+(\frac{f}{f_c})^2}}$, où f est la fréquence `freq_p` et f_c est la fréquence de coupure. De la même manière que pour le passe-bas idéal, on utilise la symétrie du spectre pour itérer sur la moitié de la taille du signal à filtrer uniquement.

Test (*test()*) : Les résultats du tests sont présentés dans la partie « Résultats des tests de l'échantillonnage et des filtres ».

Test

- Test

`void test(complexe (*)(double))`
Test du filtrage passe-bas 1er ordre.

Classe fille passe-haut 1er ordre

Constructeurs

- Constructeurs

`passe_haut_ordre_1(), passe_haut_ordre_1(U,double,double), passe_haut_ordre_1(const passe_haut_ordre_1 &)`
Ils dérivent directement de la classe mère.
Test (*test()*) : On peut vérifier qu'ils sont bien appelés en affichant *F/PHO1/ constructeur …* dans le code.

Destructeur

- Destructeur

`~passe_haut_ordre_1()`
Il ne fait rien ici.
Test (*test()*) : On peut vérifier qu'il est bien appelé en affichant *F/PHO1/ destructeur* dans le code.

Méthode

- Constructeur du filtre

`void construct()`
La méthode est similaire à celle du passe-bas 1er ordre, sauf que la fonction de transfert est $H(f) = \frac{i\frac{f}{f_c}}{1+i\frac{f}{f_c}}$ et son module $|H(f)| = \frac{\frac{f}{f_c}}{\sqrt{1+(\frac{f}{f_c})^2}}$.

Test

- Test

`void test(complexe (*)(double))`
Test du filtrage passe-haut 1er ordre.

Classe fille passe-bas 2ème ordre

Constructeurs

- Constructeurs

`passe_bas_ordre_2(), passe_bas_ordre_2(U,double,double), passe_bas_ordre_2(const passe_bas_ordre_2 &)`
Ils dérivent directement de la classe mère.
Test (*test()*) : On peut vérifier qu'ils sont bien appelés en affichant *F/PBO2/ constructeur …* dans le code.

Destructeur

- Destructeur

`~passe_bas_ordre_2()`
Il ne fait rien ici.
Test (*test()*) : On peut vérifier qu'il est bien appelé en affichant *F/PBO2/ destructeur* dans le code.

Méthode

- Constructeur du filtre

`void construct()`
Construit le signal discret `signal_filtre` qui sera utiliser pour filtrer le signal à filtrer via la méthode `apply`.

4. On commence par récupérer la taille du signal à filtrer.
5. On calcule le pas de fréquences. La variable `freq_p` à l'itération i contiendra ainsi la valeur de la fréquence à laquelle correspond la i^{ème} valeur du signal filtre.
6. On calcule le module de la fonction de transfert d'un filtre passe-bas du premier ordre et on le stocke dans la variable complexe `temp` que l'on place ensuite dans le signal filtre. On a le module de la fonction de transfert $|H(f)| = \frac{1}{\sqrt{(1-\left(\frac{f}{fc}\right)^2)^2+(\frac{\left(\frac{f}{fc}\right)}{Q})^2}}$, où f est la fréquence `freq_p`, f_c est la fréquence de coupure et Q le facteur de qualité, que l'on fixe à 0,1. De la même manière que pour le passe-bas idéal, on utilise la symétrie du spectre pour itérer sur la moitié de la taille du signal à filtrer uniquement.

Test (*test()*) : Les résultats du tests sont présentés dans la partie « Résultats des tests de l'échantillonnage et des filtres ».

Test

- Test

`void test(complexe (*)(double))`
Test du filtrage passe-bas 2er ordre.

Classe fille passe-haut 2ème ordre

Constructeurs

- Constructeurs

`passe_haut_ordre_2(), passe_haut_ordre_2(U,double,double), passe_haut_ordre_2(const passe_haut_ordre_2 &)`
Ils dérivent directement de la classe mère.
Test (*test()*) : On peut vérifier qu'ils sont bien appelés en affichant *F/PHO2/ constructeur …* dans le code.

Destructeur

- Destructeur

`~passe_haut_ordre_2()`
Il ne fait rien ici.
Test (*test()*) : On peut vérifier qu'il est bien appelé en affichant *F/PHO2/ destructeur* dans le code.

Méthode

- Constructeur du filtre

`void construct()`

Construit le signal discret `signal_filtre` qui sera utiliser pour filtrer le signal à filtrer via la méthode `apply`.

7. On commence par récupérer la taille du signal à filtrer.
8. On calcule le pas de fréquences. La variable `freq_p` à l'itération i contiendra ainsi la valeur de la fréquence à laquelle correspond la i^{ème} valeur du signal filtre.
9. On calcule le module de la fonction de transfert d'un filtre passe-haut du premier ordre et on le stocke dans la variable complexe `temp` que l'on place ensuite dans le signal filtre. On a le module de la fonction de transfert $|H(f)| = \frac{\left(\frac{f}{f_c}\right)^2}{\sqrt{(1-\left(\frac{f}{f_c}\right)^2)^2+(\frac{\left(\frac{f}{f_c}\right)}{Q})^2}}$, où f est la fréquence `freq_p`, f_c est la fréquence de coupure et Q le facteur de qualité, que l'on fixe à 0,1. De la même manière que pour le passe-haut idéal, on utilise la symétrie du spectre pour itérer sur la moitié de la taille du signal à filtrer uniquement.

Test (*test()*) : Les résultats du tests sont présentés dans la partie « Résultats des tests de l'échantillonnage et des filtres ».

Test

- Test

`void test(complexe (*)(double))`

Test du filtrage passe-haut 2er ordre.

Classe fille coupe-bande idéal

Constructeurs

- Constructeurs

`coupe_bande_ideal(),coupe_bande_ideal(U,double,double,double),`
`coupe_bande_ideal(const coupe_bande_ideal &)`

Ils dérivent directement des classes passe-bas idéal et passe-haut idéal.

Test (*test()*) : On peut vérifier qu'ils sont bien appelés en affichant *F/CBI/ constructeur …* dans le code.

Destructeur

- Destructeur

`~coupe_bande_ideal()`

Il ne fait rien ici.

Test (*test()*) : On peut vérifier qu'il est bien appelé en affichant *F/CBI/ destructeur* dans le code.

Méthode

- Constructeur du filtre

`void construct()`

On met 0 dans `signal_filtre` entre la fréquence de coupure héritée du passe-bas et la fréquence de coupure héritée du passe-haut, de la même manière qu'on la fait pour ces derniers.

Test (*test()*) : Les résultats du tests sont présentés dans la partie « Résultats des tests de l'échantillonnage et des filtres ».

Test

- Test

`void test(complexe (*)(double))`

Test du filtrage coupe-bande idéal.

Résultats des tests de l'échantillonnage et des filtres

On réalise l'échantillonnage de la fonction $f: x \to 5 + \frac{1}{2}\cos(150.2\pi.x) + \frac{1}{2}\cos(300.2\pi.x + \frac{\pi}{2}) + \frac{1}{2}\cos(450.2\pi.x + \frac{\pi}{4}) + \frac{1}{20}\cos(3000.2\pi.x)$ en 300 points de sa période (0,020). On n'utilise pas de fenêtrage, la fonction étant périodique. On va ensuite débruiter ce signal (en considérant l'harmonique de fréquence 1500 comme faisant partie du bruit) à l'aide de différents filtres passe-bas avec une fréquence de coupure de 600 Hz. Ensuite, on essaiera au contraire d'isoler les hautes fréquences de bruit à l'aide de différents filtres coupe-bande avec une fréquence de coupure de 2000 Hz.

Pour les spectres, l'ordonnée est le logarithme de l'amplitude. Cela permet de mieux visualiser les amplitudes du bruit, qui sont d'amplitude inférieure aux fréquences du signal pur.

Echantillonnage

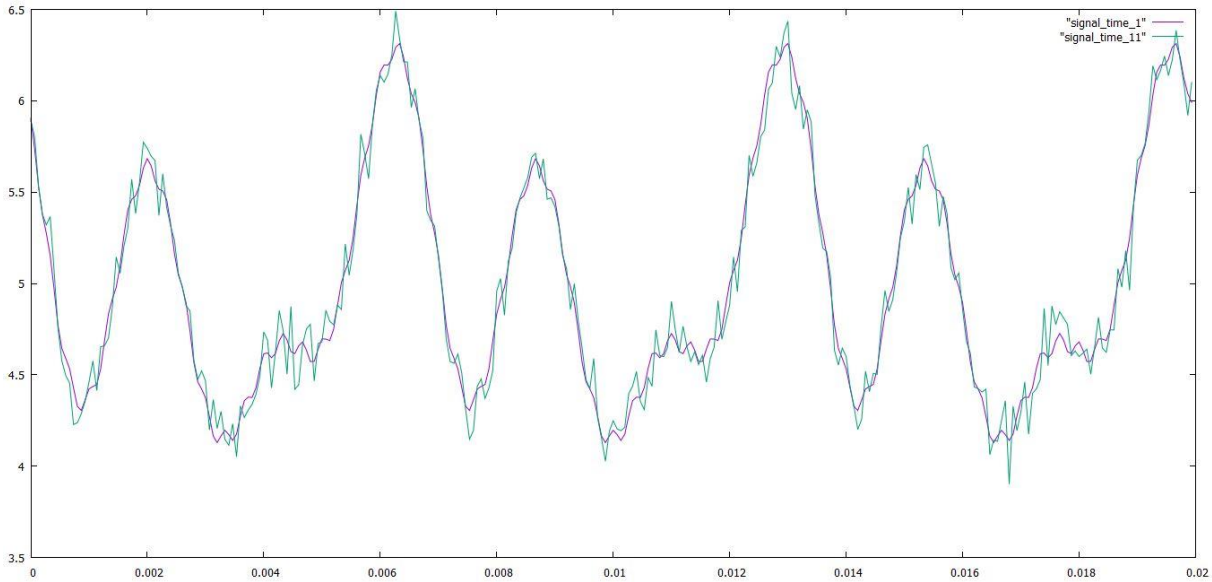


Figure 1. Signal échantillonné avec (en vert) et sans (en violet) bruit.

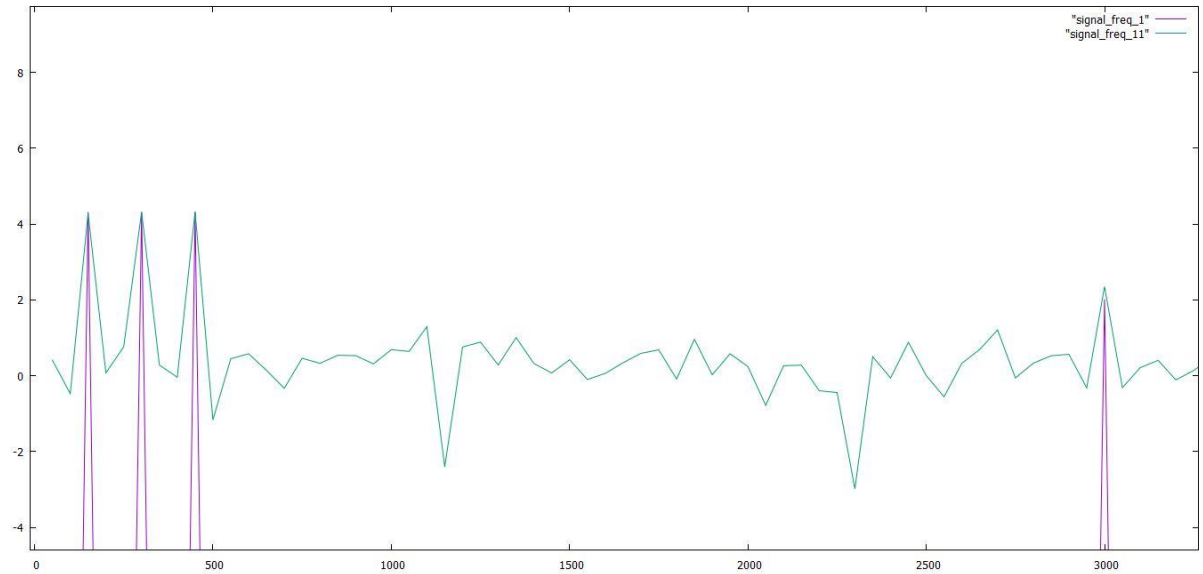


Figure 2. Spectre du signal échantillonné avec (en vert) et sans bruit (en violet).

Tout d'abord, l'échantillonnage est bon. On peut comparer en traçant la fonction f. L'axe des abscisses semble aussi bon, on voit bien que la fonction est périodique de période 0,020. On peut également identifier la valeur des harmoniques, à 150, 300, 450 et 3000 Hz. Quant à l'introduction d'un bruit, on remarque les variations autour de la vraie valeur de la fonction que cela provoque. Le résultat est très visible sur le spectre. On voit bien l'introduction de nouvelles fréquences, en plus des quatre fréquences harmoniques.

Filtrage passe-bas

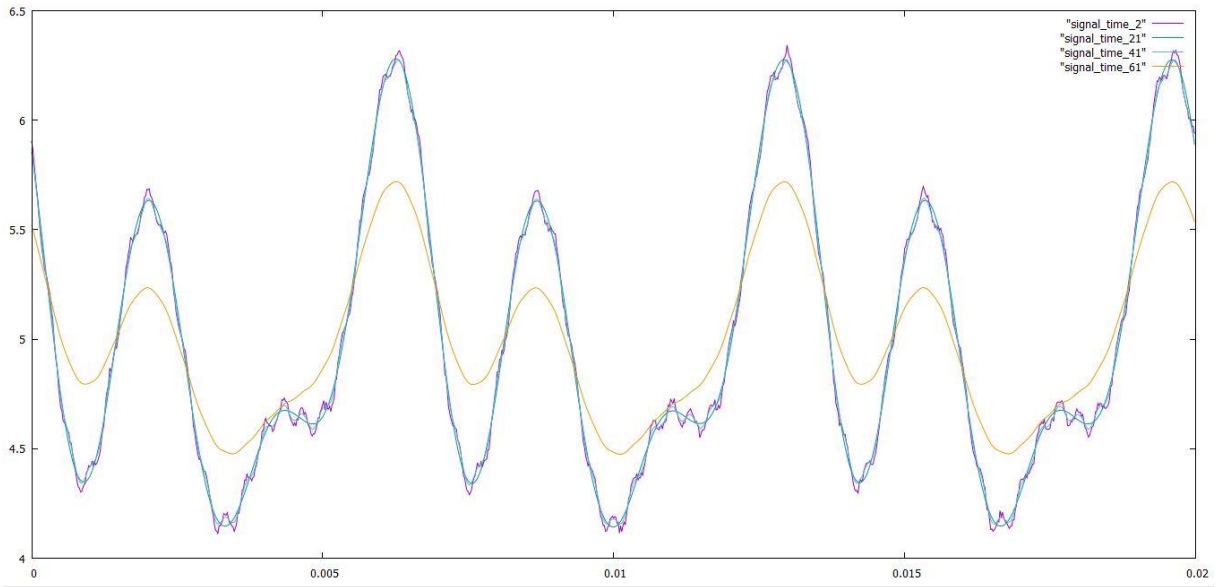


Figure 3. Signal échantillonné brut (en violet), filtré avec un filtre passe-bas idéal à 1000 Hz (en vert), filtré avec un filtre passe-bas d'ordre 1 à 1000 Hz (en bleu) et filtré avec un filtre passe-bas d'ordre 2 à 1000 Hz (en jaune).

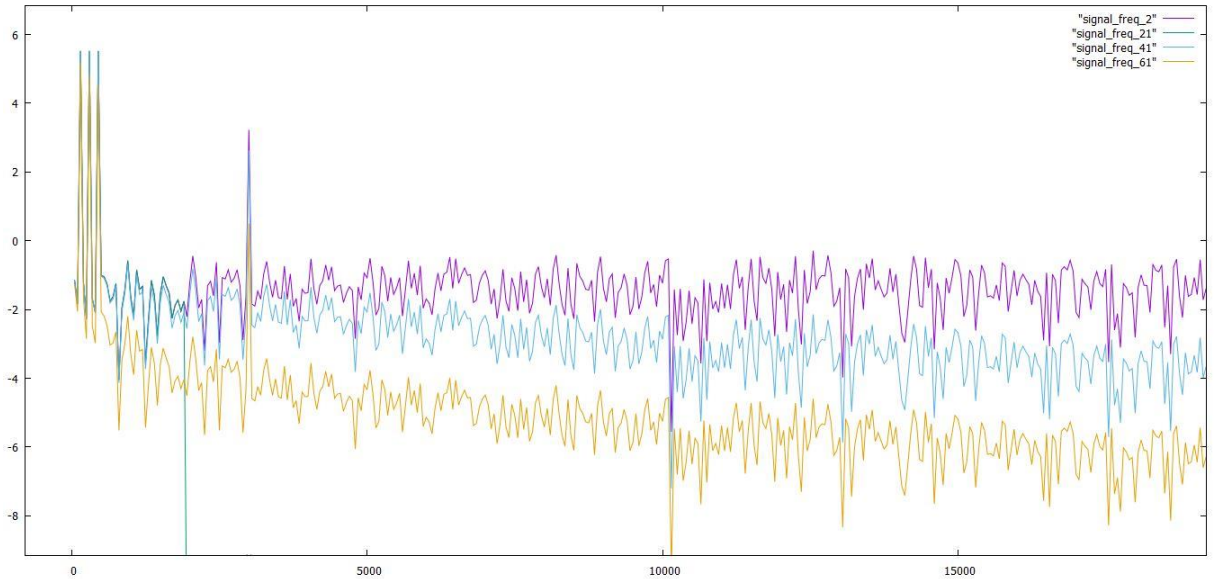


Figure 4. Spectre du signal échantillonné brut (en violet), filtré avec un filtre passe-bas idéal à 1000Hz (en vert), filtré avec un filtre passe-bas d'ordre 1 à 1000 Hz (en bleu) et filtré avec un filtre passe-bas d'ordre 2 à 1000 Hz (en jaune)

Sur la figure 3, on voit bien l'action optimale du filtre idéal. Le signal est lissé tout en gardant l'amplitude et la forme globale du signal bruité. Le spectre confirme ce résultat, on voit bien que les fréquences situées après la fréquence de coupure disparaissent complètement, y compris l'harmonique à 3000 Hz. On peut bien identifier la fréquence de coupure à 600 Hz environ. Le résultat du filtre du premier ordre est moins flagrant. On peut remarquer que le signal n'est pas parfaitement lisse, avec des résidus de la fréquence harmonique à 3000 Hz. Le vrai problème est que l'amplitude du signal est modifiée. Je pense que cela est dû à l'action du filtre passe-bas premier ordre sur les fréquences avant la fréquence de coupure. Le spectre semble confirmer cela, car on voit que les trois premières fréquences harmoniques sont un peu atténuées, et qu'il reste un résidu de celle à 3000 Hz. Il est compliqué d'identifier clairement la fréquence de coupure sur le passe-bas premier ordre. C'est encore plus compliqué sur le passe-bas deuxième ordre, qui atténue beaucoup les fréquences situées avant la fréquence de coupure. De plus, ce filtre atténue globalement le signal. Cela peut être dû au choix du facteur de qualité du filtre, qui est ici de 0,1. Bien que ce filtre soit plus précis, il est plus efficace sur les fréquences en dehors de la bande passante que le filtre du premier ordre.

Filtre passe-haut

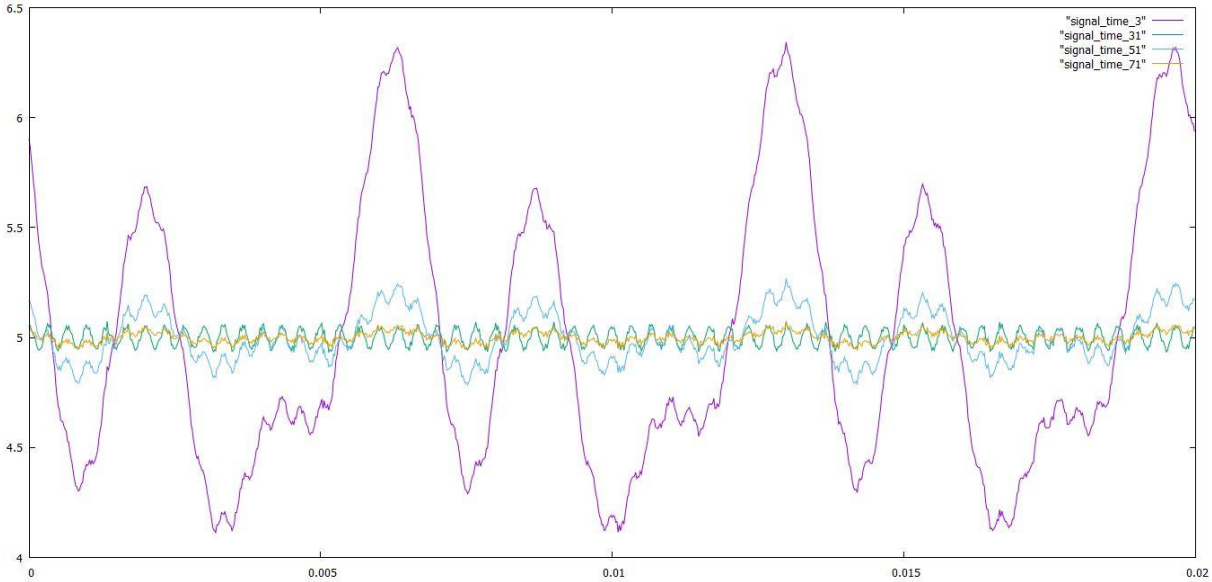


Figure 5. Signal échantillonné brut (en violet), filtré avec un filtre passe-haut idéal à 2000 Hz (en vert), filtré avec un filtre passe-haut d'ordre 1 à 2000 Hz (en bleu) filtré avec un filtre passe-haut d'ordre 2 à 2000 Hz (en jaune).

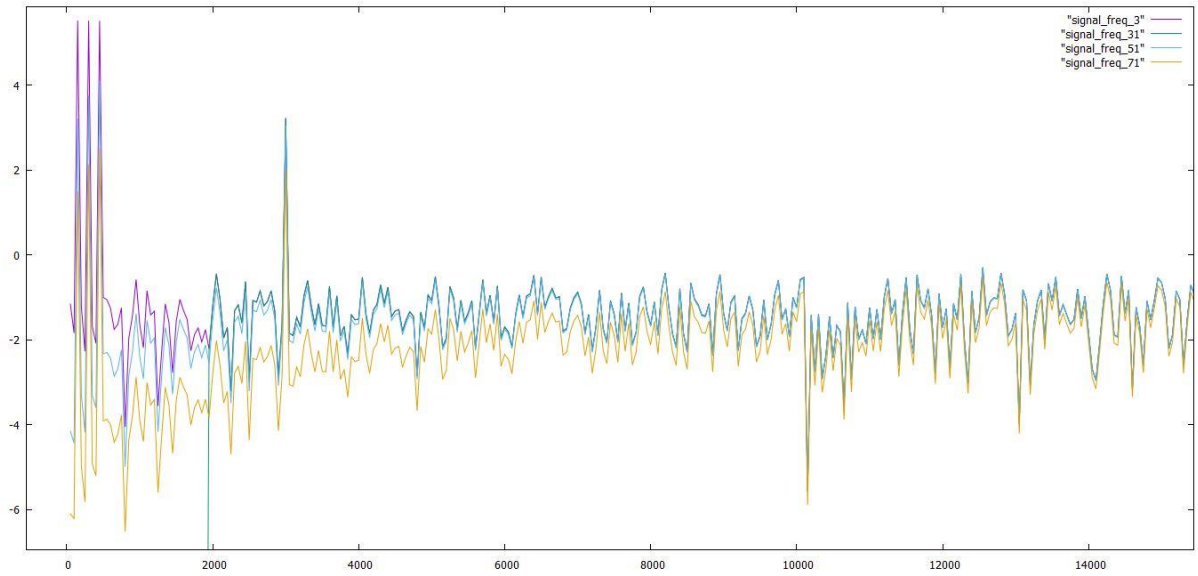


Figure 6. Spectre du signal échantillonné brut (en violet), filtré avec un filtre coupe-bande idéal à 2000 Hz (en vert), filtré avec un filtre coupe-bande d'ordre 1 à 2000 Hz (en bleu) et filtré avec un filtre passe-bas d'ordre 1 à 2000 Hz (en bleu)

L'application de ce filtre à 2000 Hz est supposée conserver une grande partie du bruit et la fréquence harmonique à 3000 Hz. Le filtre idéal réalise parfaitement cela, on voit que le résultat est encore fortement bruité, et que la forme du signal est très fortement influencée par l'harmonique à 3000 Hz. On voit d'ailleurs que celle-ci est parfait conservée sur le spectre. On peut compter le nombre de périodes et confirmer cela. Le spectre nous permet aussi d'identifier la fréquence de coupure qui se situe bien à 2000 Hz. Comme pour le passe-bas, le coupe-bande premier ordre n'atténue pas complètement les fréquences situées avant la fréquence de coupure. On voit très bien les résidus des trois premières fréquences fondamentales sur le signal et sur son spectre, l'amplitude de ces résidus étant aussi important que l'amplitude de l'harmonique à 3000 Hz. Il est à nouveau compliqué d'identifier la fréquence de coupure. Les conclusions pour le filtre du deuxième ordre sont les mêmes que pour le filtre passe-bas.

Filtre coupe-bande

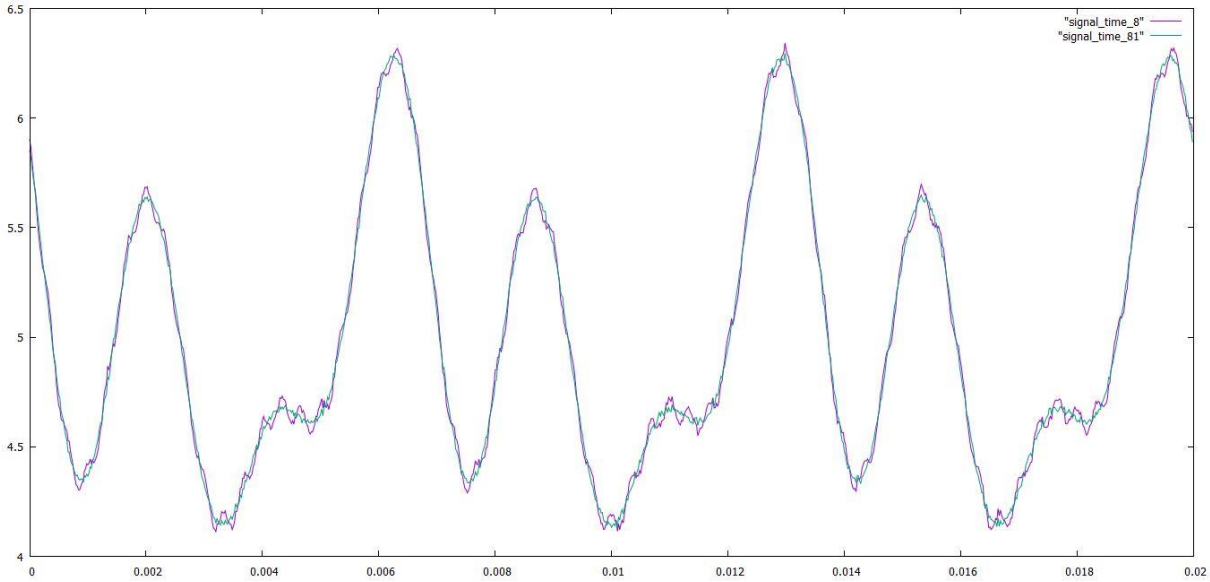


Figure 7. Signal échantillonné brut (en violet) et filtré avec un filtre coupe-bande idéal entre 2900 Hz et 3100 Hz (en bleu).

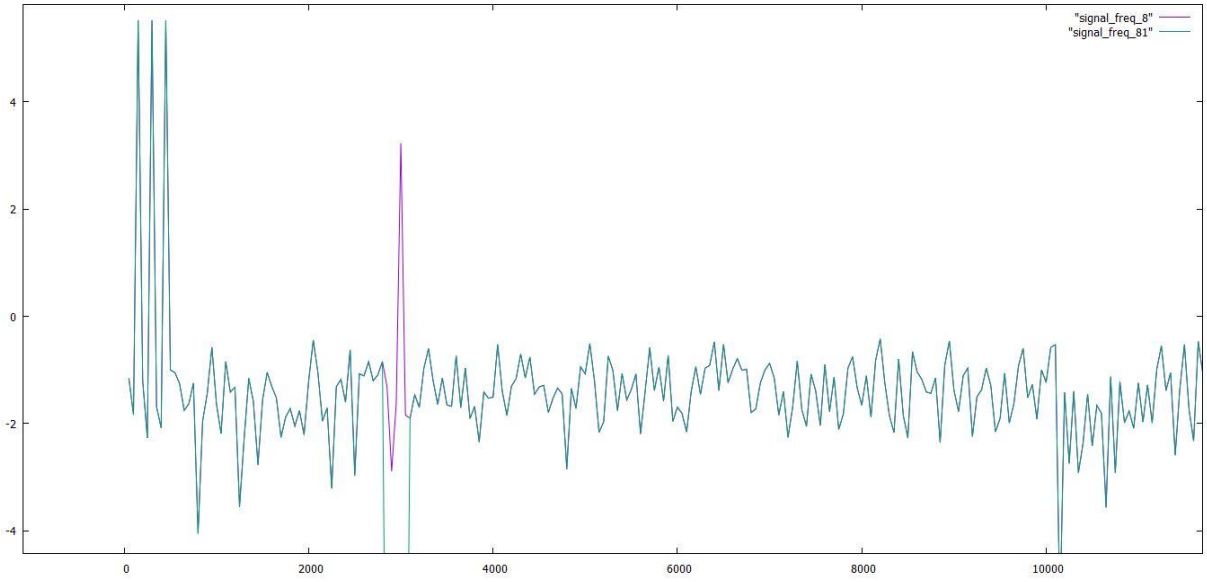


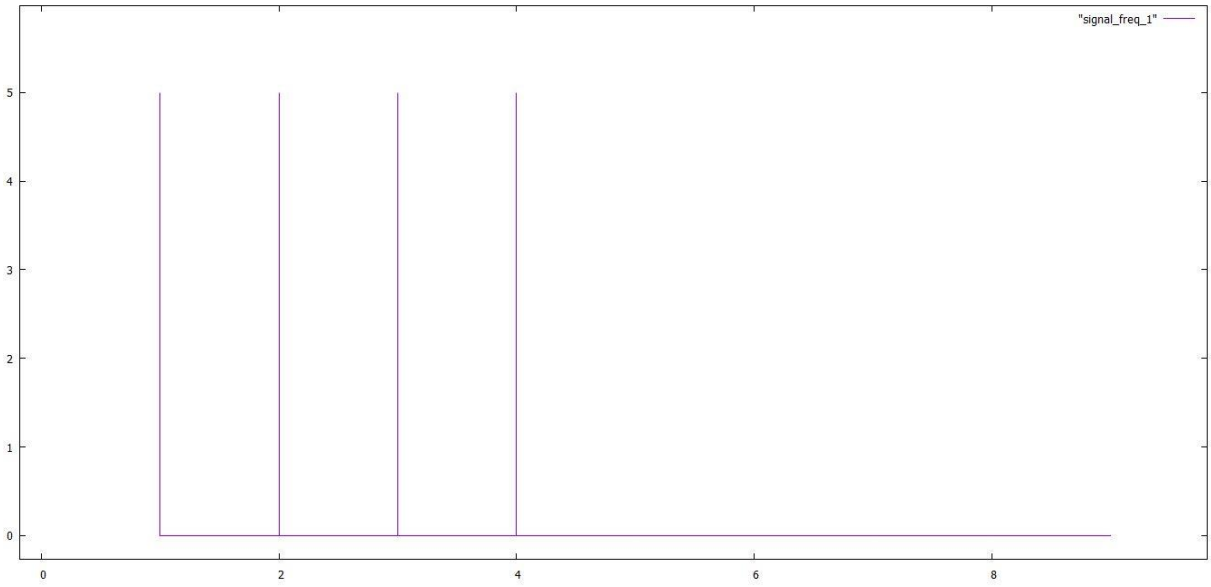
Figure 8. Spectre du signal échantillonné brut (en violet) et filtré avec un filtre coupe-bande idéal entre 2900 Hz et 3100 Hz (en bleu).

Le filtre coupe-bande idéal retire bien la fréquence harmonique à 3000 Hz. On voit la conséquence de cela sur le signal, puisque les petites ondelettes que cette fréquence provoquait ont disparus. J'ai essayé de faire des filtres coupe-bande du premier et deuxième ordre, par application successive du filtre passe-bas et du filtre coupe-bande, mais les résultats n'étaient pas satisfaisants. Le signal filtré était très atténué, et très imprécis, les fréquences les plus atténuées n'étant pas celles voulues.

Problème de sous-sampling

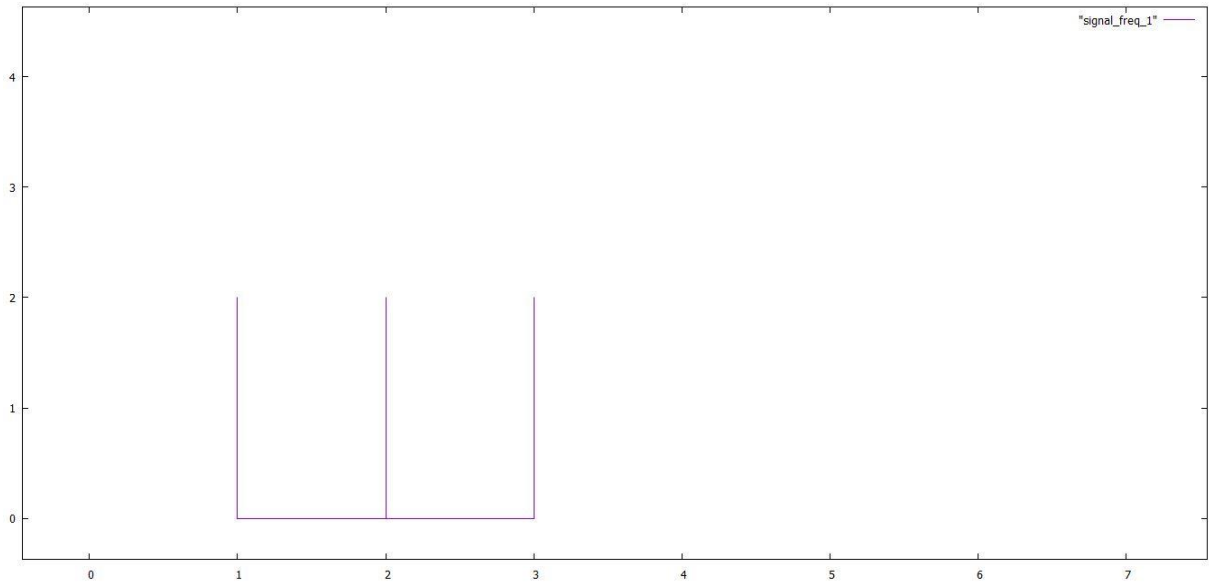
Etudions le problème de sous-sampling sur une sinusoïde : $x \to 5 + 0.5\sin(2\pi x) + 0.5\sin\left(4\pi x + \frac{\pi}{2}\right) + 0.5\sin(6\pi x + \frac{\pi}{4}) + 0.5\sin(8\pi x)$. Ce signal comporte 4 fréquences : 1 Hz, 2 Hz, 3 Hz et 4 Hz.

Voici son spectre, en prenant 20 échantillons sur une période, soit une période d'échantillonnage de $\frac{1}{20}$ et une fréquence d'échantillonnage de 20 Hz.



On voit bien les 4 fréquences du signal.

Maintenant, prenons 8 échantillons seulement, toujours sur une période, soit une période d'échantillonnage de $\frac{1}{8}$ et une fréquence d'échantillonnage de 8 Hz :



On voit que l'on a perdu de l'information. En effet, la quatrième fréquence n'apparait plus. Cela a comme conséquence que l'on ne peut plus retrouver le signal de départ à partir de ce spectre.

On constate donc que si la fréquence d'échantillonnage n'est pas assez importante, le spectre ne contient plus toutes les fréquences. Le théorème de Nyquist-Shannon précise cela : *La représentation discrète d'un signal exige des échantillons régulièrement espacés à une fréquence d'échantillonnage supérieure (strictement) au double de la fréquence maximale présente dans ce signal*.

Efficacité en temps de calcul avec la STL

L'implémentation du signal discret avec la STL et du template de la classe filtre étaient assez facile, mais les résultats sont décevants.

Voici le temps d'exécution sans la STL :

```
--- signal_discret ---
F/PBI/test effectue, signal_time_21 de spectre signal_freq_21 est le resultat du filtrage passe-bas de signal_time_2 de spectre signal_freq_2 a 600Hz
F/PHI/test effectue, signal_time_31 de spectre signal_freq_31 est le resultat du filtrage passe-haut de signal_time_3 de spectre signal_freq_3 a 2000Hz
F/PBO1/test effectue, signal_time_41 de spectre signal_freq_41 est le resultat du filtrage passe-bas de signal_time_4 de spectre signal_freq_4 a 600Hz
F/PHO1/test effectue, signal_time_51 de spectre signal_freq_51 est le resultat du filtrage passe-haut de signal_time_5 de spectre signal_freq_5 a 2000Hz
F/PBO2/test effectue, signal_time_61 de spectre signal_freq_61 est le resultat du filtrage passe-bas de signal_time_6 de spectre signal_freq_6 a 2000Hz
F/PHO2/test effectue, signal_time_71 de spectre signal_freq_71 est le resultat du filtrage passe-haut de signal_time_7 de spectre signal_freq_7 a 1000Hz
F/CBI/test effectue, signal_time_81 de spectre signal_freq_81 est le resultat du filtrage coupe-bande ideal de signal_time_8 de spectre signal_freq_8 entre 2900 et 3100 Hz

real    0m6.499s
user    0m6.266s
sys     0m0.078s
```

Et avec la STL :

```
--- signal_discret_stl ---
F/PBI/test effectue, signal_time_21 de spectre signal_freq_21 est le resultat du filtrage passe-bas de signal_time_2 de spectre signal_freq_2 a 600Hz
F/PHI/test effectue, signal_time_31 de spectre signal_freq_31 est le resultat du filtrage passe-haut de signal_time_3 de spectre signal_freq_3 a 2000Hz
F/PBO1/test effectue, signal_time_41 de spectre signal_freq_41 est le resultat du filtrage passe-bas de signal_time_4 de spectre signal_freq_4 a 600Hz
F/PHO1/test effectue, signal_time_51 de spectre signal_freq_51 est le resultat du filtrage passe-haut de signal_time_5 de spectre signal_freq_5 a 2000Hz
F/PBO2/test effectue, signal_time_61 de spectre signal_freq_61 est le resultat du filtrage passe-bas de signal_time_6 de spectre signal_freq_6 a 2000Hz
F/PHO2/test effectue, signal_time_71 de spectre signal_freq_71 est le resultat du filtrage passe-haut de signal_time_7 de spectre signal_freq_7 a 1000Hz
F/CBI/test effectue, signal_time_81 de spectre signal_freq_81 est le resultat du filtrage coupe-bande ideal de signal_time_8 de spectre signal_freq_8 entre 2900 et 3100 Hz

real    0m6.554s
user    0m6.203s
sys     0m0.109s
```

Malgré plusieurs exécutions, en moyenne, je n'ai observé aucune différence significative des temps de calculs. Je pense que je n'exploite pas assez les capacités d'optimisation de l'usage de la STL. En effet, mise à part le fait de ne pas stocké la taille, et de ne pas faire d'allocation dynamique, les fonctions utilisant la STL ne diffèrent pas beaucoup de celles avec l'allocation dynamique.

Discussion et bilan

Toutes les fonctions demandées, exceptée le coupe-bande ordre 1 ont été codées. Les classes de signaux discrets et continus ont été bien testées, mais je ne voyais pas trop comment tester les classes filtres et l'échantillonnage de manière efficace, sans avoir à afficher les résultats. En effet, cette méthode visuelle n'est pas pratique à tester à chaque fois que l'on fait une modification mineure du programme.

Il y a des choix que j'ai fait pour des fonctions qui ne sont pas optimaux à mon avis. Par exemple, il s'est avéré inutile de renvoyer des tableaux de complexes avec la transformée de Fourier, car je devais toujours transformer ces tableaux en signaux discrets au final. De plus, ma gestion de l'héritage de la classe coupe-bande ne me satisfait pas beaucoup, car on stocke deux signaux à filtrer et deux signaux filtre (un dans chaque classe mère), et on ne travaille que sur un des deux au final.

Le programme ne comporte pas de fuites de mémoire. Le résultat de la commande `valgrind ./tests` :

```
HEAP SUMMARY:
    in use at exit: 0 bytes in 0 blocks
  total heap usage: 690 allocs, 690 frees, 4,912,216 bytes allocated

All heap blocks were freed -- no leaks are possible
```

Je n'ai pas utilisé la fonction de fenêtrage car je ne lui trouvais pas d'utilité avec mes exemples périodiques.

Mes fichiers sont plutôt longs, notamment le fichier filtre.hpp, qui fait plus de 600 lignes, puisqu'il contient toutes les fonctions de 8 classes, en raison de l'usage de templates de classes.

Une bonne voie d'amélioration du programme serait de gérer et tester le cas de signaux complexes. J'ai fait le choix de ne travailler qu'avec des signaux réels car cela me permettait de bien comprendre ce que je faisais, mais le fait que la classe signal discret soit composée d'un tableau de complexes était du coup assez contraignant.

Finalement, ce projet aura surtout demandé beaucoup de travail de recherche sur les aspects mathématiques du sujet. Une fois que j'avais bien compris le but du projet, ce travail s'est avéré très intéressant.