Séance 1: programmation d'un filtre FIR en virgule fixe

Lors de cette séance, on va programmer quelques algorithmes classiques de traitement du signal, sur des nombres entiers. Le langage employé sera le langage C, et on emploiera l'environnement de développement intégré(Integrated Development Environment ou IDE) ***Netbeans*** *pour tester les différents programmes...*

1 introduction

Un filtre FIR est un filtre numérique dont la fonction de transfert en Z s'écrit : $G(z)=\sum_{k=0}^{NG-1} g_k . z^{-k}$

la sortie s_n du filtre, d'entrée e_n s'écrit : $s_n=\sum_{k=0}^{NG-1} \underbrace{g_k}_{\text{coeffs du filtre}} \cdot \underbrace{e_{n-k}}_{\text{entrées passées}}$

remarque: l'ordre du filtre est égal à $NG-1$, le nombre de coefficients g_k est égal à NG

Un filtre FIR présente les avantages suivants sur un filtre IIR {infinite response filter }:

- il est systématiquement stable {sa sortie ne peut pas diverger, puisque sa réponse impulsionnelle est de durée finie}

2- il a un temps de retard de groupe constant, et connu (si phase linéaire...). Ceci est très important pour beaucoup d'applications (chaque fois que l'on doit localiser temporellement l'entrée: écho radar, spectrométrie, traitement de son et d'images)

2 Synthèse et analyse d'un filtre fir sous scilab

2.1 mise en place des outils informatiques

- ☐ télécharger le fichier *be_systemes_embarque_eea_631.zip* depuis le centre de ressources EEA, puis l'extraire sous votre compte...
- ☐ Lancer le logiciel scilab
- ☐ ouvrir avec un éditeur de texte le fichier *fir.sce, (click droit sur l'icône, ouvrir avec kate).*
  Ce fichier est un programme scilab effectuant les tâches suivantes :
  1- calcul d'un filtre FIR passe-bas à l'aide de la fonction scilab :*wfir*
  2- quantification des coefficients g_n du filtre
  3- comparaison des réponses impulsionnelles et fréquentielles
  4- génération d'un bout de programme en langage c, contenant la déclaration des coefficients entiers, des décalages à droite, et des coefficients réels dans un fichier *toto.c*
- ☐ exécuter sous scilab le programme '*fir.sce*', et vérifier qu'il ne produit pas d'erreur.(*le code d'exécution sous scilab est écrit en remarque à la première ligne du fichier: modifier éventuellement le nom du répertoire, sélectionner le texte par glissement de la souris bouton gauche maintenu appuyé, le copier ensuite sous scilab en cliquant sur le bouton du milieu de la souris dans la fenêtre* ***scilab****, après le prompt >>)*
- ☐ éditer également le fichier *toto.c* avec l'éditeur de texte ***kate***...

2.2 calcul d'un filtre FIR depuis un gabarit

- on veut fabriquer un passe-bas laissant passer les fréquences inférieures à 100 hz, et qui travaille dans la bande de fréquence 0 à 2000hz. Le gain du filtre devra répondre au gabarit figure 1

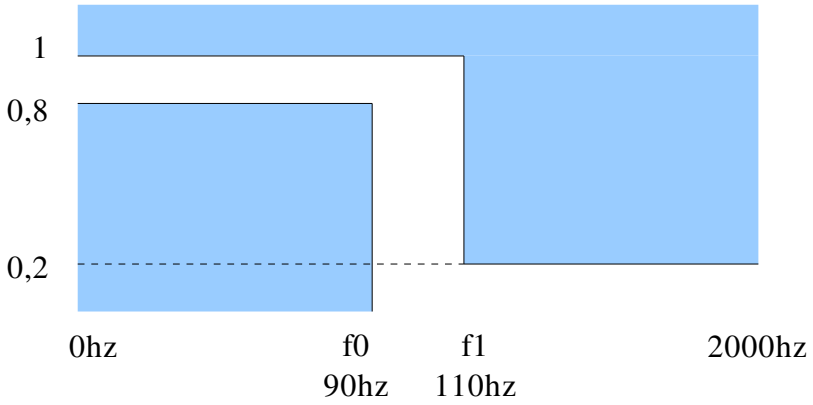

figure 1: gabarit du filtre fir

2.2.1 choix d'une fréquence d'échantillonnage(bande de fréquence traitée)

- ☐ modifier la fréquence d'échantillonnage *fe* dans le programme *fir.sce*, pour que le filtre travaille bien dans la bande de fréquence 0 à 2000 hz.

2.2.2 choix d'un nombre d'échantillons(raideur de coupure)

2.2.2.1 méthode manuelle

- ☐ exécuter le programme, et vérifier si le filtre répond au gabarit, en visualisant la réponse fréquentielle, et en visualisant les fréquences *f0* et *f1* correspondant aux gains *g0*=0.8 et *g1*=0.2.
- ☐ augmenter le nombre d'échantillons *NG* de la réponse impulsionnelle, et recommencer la mesure...

Vous avez dû vous rendre compte que la recherche graphique des fréquences f0 et f1 pour lesquelles le gain du filtre correspond aux valeurs du gabarit est très fastidieuse...

- ☐ ajouter dans la partie 8 du programme ***fir.sce*** les lignes de code déterminant automatiquement *f0* et *f1*. Il est conseillé de vous inspirer des lignes ci-dessous, qui emploient la fonction ***find*** de scilab, pour détecter automatiquement une fréquence *f0* et l'afficher à l'écran...

```
i=find(abs_Gf>=0.9); // i= indices pour lesquelles abs_Gf>=0.9
i=max(i);            // i= plus grande valeur de i
f0=fk(i);            // f0=frequence correspondante
disp(" |Gf|=0.9, lorsque f=" +string(f0)); // affichage a l'ecran
```

- ☐ - augmenter *NG* jusqu'à ce que le filtre réponde au gabarit. On s'arrêtera lorsque qu'on aura déterminé *NG* à la dizaine près...
- ☐ pour la valeur de *NG* finalement retenue, déterminer la durée réelle *TG* (en secondes) de la réponse impulsionnelle du filtre (ajouter dans la partie 8 du programme le code permettant d'évaluer *TG* en fonction de *NG* et de la période d'échantillonnage *te*, puis d'afficher *TG* à l'écran)

2.2.2.2 mise en évidence du lien entre durée réelle et raideur de coupure

- ☐ - choisir à présent une fréquence d'échantillonnage *fe*=40000hz , et modifier *NG* pour que le filtre réponde au gabarit...{ choisir directement comme valeur de *NG* le nombre d'échantillons qui correspond à la même durée réelle qu'en 2.2.2.1 }
  - quels sont les avantages et inconvénients de ce filtre par rapport à celui déterminé en 2.2.2.1

2.2.2.3 méthode plus systématique

pour déterminer approximativement le nombre d'échantillons *NG* d'un filtre *FIR*, sans tâtonnement, on procède de la façon suivante.

Phase 1: on calcule le filtre FIR pour une valeur quelconque de *NG*, pas trop petite (typiquement *NG*=100).

Phase 2: on en déduit l'(es)écart(s) en fréquences ΔF entre la réponse fréquentielle désirée et la réponse fréquentielle obtenue. Par exemple dans notre cas on évalue : $\Delta F = F_1 - F_0$, où F_1 est la fréquence pour laquelle le gain vaut *0.8*, et F_0 la fréquence pour laquelle le gain vaut *0.2*.

Phase 3: si à présent on fait varier *NG ou Te*, en employant la même méthode de calcul du filtre,alors le produit $P = \Delta F . \underbrace{NG . Te}_{\text{durée réelle } TG \text{ de la réponse impulsionnelle}}$ restera approximativement constant.

Connaissant la valeur de *P* pour les valeurs initiales de $NG, \Delta F, Te$, on peut en déduire la valeur de *NG* pour d'autres valeurs de ΔF ou de Te , et cette valeur de *P*

- **application à l'exemple considéré :**

- ☐ initialiser NG à 100 et fe à 4000 au début de votre programme.
- ☐ compléter le code de la partie 9 du programme ***fir .sce***, permettant de déterminer la valeur approximative de NG à retenir pour que le filtre réponde au gabarit..
- ☐ modifier la valeur de NG en début de programme, en fonction du résultat trouvé, et vérifier que le filtre 'colle à peu près' au gabarit

2.3 codage du filtre FIR en nombres entiers

*Dans cette partie on va s'intéresser au codage du filtre en arithmétique **16/32** bits*

2.3.1 calcul du facteur d'échelle de signal

- ☐ - modifier le programme pour ajuster la variable *NB_BITS* au nombre de bits de codage
- ☐ En appliquant la technique du travail préparatoire avec des additions-multiplications en parallèle, déterminer le plus grand facteur d'échelle de signal $\lambda_{SIGNAL} = 2^{L_{SIGNAL}}$, en puissance entière de 2 que l'on peut employer pour ce filtre FIR.

  On vous a plus ou moins maché le travail dans la partie 1 du programme: rechercher la chaîne de caractères LAMBDA_SIGNAL dans le programme, et compléter les lignes correspondant à la détermination de $\lambda_{SIGNAL} = 2^{L_{SIGNAL}}$, en appliquant la même technique que pour le travail préparatoire.
- ☐ Déterminer analytiquement, en fonction de $\lambda_{SIGNAL} = 2^{L_{SIGNAL}}$ et du nombre de coefficients NG , l'erreur maximale en sortie du filtre, due aux bruits de quantification de troncature en sortie des décalages à droite

2.3.2 analyse des erreurs induites par la quantification des coefficients

le programme propose plusieurs possibilités de quantification des coefficients gn, et stocke les résultats de la façon suivante (mêmes notations que pour le travail préparatoire):
- l'ensemble des coefficients entiers est stocké dans le tableau gn_N[1..NG]
- l'ensemble des coefficients quantifiés est stocké dans le tableau gn_q[1..NG]
- les décalages associés sont stockés dans le tableau Lg[1..NG]
le code correspondant peut être trouvé en cherchant la chaîne de caractères :QUANTIFICATION
- Par défaut, chaque coefficient gn est quantifié avec sa propre échelle...

2.3.2.1 erreur dans le cas d'un facteur d'échelle par coefficient

- ☐ -ajouter dans le programme les instructions permettant d'afficher les plus valeurs minimale et maximale des décalages dans le tableau Lg (employer les fonctions scilab ***min*** et ***max***)
- ☐ relever également l'erreur absolue maximale entre la réponse fréquentielle du filtre quantifié : $G_q(z)=\sum_{n=0}^{NG-1} g_{nq}.z^{-n}$, et la réponse du filtre idéal : $G(z)=\sum_{n=0}^{NG-1} g_n.z^{-n}$.
  Vous pouvez procéder graphiquement, mais il vaut mieux ajouter une ligne de programme qui calcule et affiche cette valeur)
- ☐ sauver le programme ***toto.c*** généré sous le nom ***toto_2321.c*** *(pour plus tard)*

2.3.2.2 cas de l'utilisation du même facteur d'échelle pour les coefficients

- ☐ -modifier la méthode de quantification, en choisissant le même facteur d'échelle *Lg* pour tous les coefficients, *Lg* étant le plus grand possible
- ☐ Afficher le tableau ***Lg*** et en déduire la valeur de ***Lg*** retenue
- ☐ relever à nouveau l'erreur absolue maximale entre la réponse fréquentielle du filtre quantifié : $G_q(z)=\sum_{n=0}^{NG-1} g_{nq}.z^{-n}$, et la réponse du filtre idéal: $G(z)=\sum_{n=0}^{NG-1} g_n.z^{-n}$
- ☐ sauver le programme ***toto.c*** généré sous le nom ***toto_2322.c*** *(pour plus tard)*

2.3.2.3 cas de l'utilisation du même facteur d'échelle pour les coefficients, égal au facteur d'échelle de signal

- ☐ - modifier la méthode de quantification, en choisissant le même facteur d'échelle *Lg=L_SIGNAL*, pour tous les coefficients.
- ☐ relever à nouveau l'erreur absolue maximale entre la réponse fréquentielle du filtre quantifié : $G_q(z)=\sum_{n=0}^{NG-1} g_{nq}.z^{-n}$, et la réponse du filtre idéal $G(z)=\sum_{n=0}^{NG-1} g_n.z^{-n}$
- ☐ sauver le programme ***toto.c*** généré sous le nom ***toto_2323.c*** *(pour plus tard)*

2.3.2.4 commentaires

- ➪ *- quelle est la méthode la plus précise...*
- ➪ *quelle est la méthode la moins coûteuse en termes de temps de calcul?*
- ➪ si je voulais imposer *L_SIGNAL = LG* trouvé en 2.3.2.2 , sur combien de bits devrais-je coder la variable interne *V* ?(cf additions/ multiplications en parallèle)
  C'est très souvent cette dernière méthode qui est retenue sur les DSP, qui disposent de registres codés sur plus de 2.NB_BITS... par exemple, l'ADSP-BF-537 sur lequel vous allez tester vos filtres dispose d'une arithmétique 16/32 bits, et de 2 registres d'accumulation A0,A1 sur 40 bits...

3 Codage en langage *C*

3.1 utilisation d'un environnement de développement intégré (IDE in english)

3.1.1 qu'est-ce qu'un IDE (Integrated Development Environment)

Un IDE est une application regroupant un ensemble d'outils informatiques permettant au développeur de gagner en efficacité. Il existe de nombreux IDE plus ou moins riches en outils (Eclipse, Visual studio, Borland C++, Kdevelop, Netbeans .) souvent multi-langages.
Il est probable que vous ayez à employer de nombreux IDE à l'avenir...L'important n'est pas d'utiliser

tel ou tel IDE, mais de savoir quelles sont les fonctionnalités principales d'un IDE (et de prendre l'habitude de les exploiter intelligemment). Typiquement un IDE contient :

1- un éditeur multi-fichiers avancé, avec coloration syntaxique, auto formatage de code, indentation automatique, création de 'squelettes type' de fichiers...
2- des fonctionnalités de visualisation des classes , variables et fonctions, de recherche -remplacement- modification avancées(indispensable)
3- un compilateur et lanceur d'application intégré, permettant de retrouver rapidement les erreurs dans les fichiers source, et générant automatiquement les MakeFile
4- un débugger intégré permettant d'exécuter le programme pas à pas, de visualiser les variables et la pile d'appel (indispensable)
5- un générateur de versions de développement CVS (Control Version System), pour garder en mémoire toutes les modifs, générer les docs des versions du logiciel, et permettre de travailler à plusieurs sur le même projet...

3.1.1 Prise en main de l'IDE netbeans

En ce qui nous concerne, nous emploierons **Netbeans**, initialement prévu pour le développement en langage java, mais dont les fonctionnalités C et C++ sont largement suffisantes. *Il est possible que ce petit guide de lancement ne soit plus a jour, ou que les menus soient en français, selon la version de l'IDE employée, essayez de vous adapter si tel est le cas...*
AVERTISSEMENT : *Ayez une lecture intelligente du travail demandé!!!, le but est de vous montrer les différentes fonctionnalités et leur intérêt, pas de vous faire exécuter bêtement une série de consignes sans aucune signification=> focalisez votre attention sur ce que signifient les différentes étapes, pas sur la façon de les exécuter*

Si vous travaillez sur votre compte LSI : NE PAS EFFECTUER LES MISES A JOUR (update) QUI VONT REMPLIR VOTRE COMPTE LSI SI VOUS LES LANCEZ!...

3.1.1.1 Création d'un nouveau projet

- ☐ lancer netbeans, en effectuant un double click sur l'icone, ou en écrivant *netbeans* sous un terminal
- ☐ choisir le menu ***file->New Project***, de type ***C/C++*** , de sous type ***C/C++ Application***
- ☐ cliquer sur Next , et choisir comme nom de projet : ***systemes_embarques*** (sans accents, sans espaces, sans majuscules svp). *En profiter pour relever le répertoire du projet, fermer la fenêtre Welcome qui s'affiche la toute première fois que vous lancez Netbeans*

3.1.1.2 personnalisation des propriétés du projet

- ☐ placer le curseur de la souris sur l'item ***systemes_embarques*** de la fenetre Projects. Si cette fenêtre n'est pas visible, la faire apparaître en sélectionnant: ***Window -> Projects***
- ☐ effectuer un click droit, ce qui active le menu contextuel lié à ce projet,sélectionner ***properties*** *(tout en bas je crois).*
- ☐ au niveau du compilateur C, nous allons demander un niveau de warning élevé=> sélectionner ***build->C compiler-> Warning Level-> More Warnings***
- ☐ au niveau du Linker, nous allons intégrer la librairie mathématique, ce qui peut s'opérer de plusieurs façons

  méthode 1- sélectionner ***build->Linker-> Additional Options,*** *cliquer sur les* ***...*** *à droite qui déclenchent l'ouverture d'une fenêtre de saisie,* et saisir : ***-lm***

  méthode 2- sélectionner ***build->Linker-> Libraries,*** *cliquer*

*sur les **...** à droite, **add Standard Library-> Mathematics***
A l'avenir, pour personnaliser un projet, vous pourrez utiliser les autres options du menu properties...

3.1.1.3 Création des fichiers du projet main.c, fir.c, fir.h

- ☐ sous la fenêtre Projects, sélectionner le champ ***systemes_embarques ->Source Files ,*** effectuer un click droit(menu contextuel) puis sélectionner ***New-> Main C File.***
  *- modifier le nom du fichier: choisir **main** à la place de **newmain**, et clicker sur **finish***
- ☐ En procédant de la même façon, créer à présent un nouveau fichier C(***Empty C File***) s'appelant ***fir.c*** *(l'extension .c est automatiquement rajoutée au nom)*
- ☐ sélectionner le champ ***systemes_embarques ->Header Files*** effectuer un click droit(menu contextuel) puis sélectionner ***New-> C Header File,*** que vous appellerez ***fir*** (l'extension *.h* sera ajoutée automatiquement au nom du fichier)**.**

3.1.1.4 Explication de la structure du projet

*Les fonctions relatives au filtres **fir** seront toutes stockées dans le fichier **fir.c***
*Le fichier **fir.h** contiendra la déclaration de toutes les fonctions du fichier **fir.c**, qui devront être visibles depuis les autres fichiers du programme, avec la directive **extern**, qui signifie que les fonctions sont déclarées 'ailleurs' et ne devront être trouvées qu'au linckage.*
UNE DIRECTIVE EXTERN VERS UNE FONCTION NON DEFINIEE NE GENERE D'ERREUR QU'AU LINCKAGE, PAS A LA COMPILATION.

Exemple :
*- dans le fichier **main.c**,*

- ☐ *ajouter la ligne suivante, juste avant la déclaration de* la *fonction main*

```
#include "fir.h"
```

- ☐ *ajouter également l'appel à une fonction **teste_fir** dans la fonction **main()***
  teste_fir();
- ☐ *dans le fichier **fir.h**, juste après les 2 lignes*
  extern "C"
  #endif
  *ajouter la ligne de déclaration de la fonction externe **teste_fir**, QUI N'EXISTE PAS*
  extern void teste_fir(void);
- ☐ sauver tous les fichiers (icones = 2 disquettes)
- ☐ lancer la compilation du fichier ***main.c*** en le sélectionnant au niveau de l'éditeur, puis en activant le menu ***build->compile main.c***
  *en bas de l'IDE, vous devriez voir apparaître la fenêtre de sortie, qui détaille les messages du compilateur, et qui finit par :**Build successful. Exit value 0.**=> aucun problème de compilation*
- ☐ à présent lancer la compilation + linckage de tout le projet => ***Build->Build Main Project***
  une erreur devrait apparaître dans la fenêtre de sortie : dans le genre
  *patati et patata, **undefined reference to teste_fir**...*
  *qui se traduit par : tu me parles d'une fonction **teste_fir** , et je ne la trouve nulle part!...*
  - clicker sur le message d'erreur, et l'IDE va vous placer sous l'éditeur de texte, à la ligne du fichier qui a généré cette erreur, qui est la ligne d'appel de la fonction ***teste_fir*** dans la fonction ***main()*** du fichier ***main.c***...

Ce type d'erreur est très fréquent, et se produira chaque fois que vous ferez appel à une fonction

contenue dans une librairie non liée au projet (le plus classique est l'emploi des fonctions cos,sin,etc.. sans lincker avec la librairie mathématique)...

- ☐ Correction de l'erreur : il suffit de rajouter la fonction ***teste_fir*** dans le fichier ***fir.c*** qui contient l'implémentation des fonctions déclarées dans ***fir.h***
  => *ajouter les lignes suivantes dans fir.c*
  void teste_fir(void){
  }
  - construire a nouveau le projet, puis l'exécuter menu **Run->run main Project**.

Il ne se passe rien (c'est normal) , mais il n'y a plus d'erreur, votre application est correctement structurée :

1- le programme principal ne connait que ce qu'il a à connaître des filtres *fir* (le lancement d'une fonction de test)
2- les fonctions de *fir.c* à connaître depuis *main.c* sont déclarées dans le header *fir.h* correspondant
3- le fichier *fir.c* contient l'implémentation des fonctions et structures propres aux filtres *fir*.

Quand nous rajouterons des filtres iir, nous procéderons de même en créant un fichier source ***iir.c*** *,un fichier header* ***iir.h*** *et ainsi de suite (une vraie application en c contient facilement plusieurs centaines de header .h et de source .c)*

3.2 codage du filtre fir idéal (coefficients en double précision)

3.2.1 création des structures et fonctions, organisation du code

à ce stade on va créer les 3 fonctions et la structure qui vont permettre de tester le filtre fir, en commençant tout d'abord par les coefficients réels...

- soit e_n l'entrée à l'instant *n* du filtre, nous allons écrire une fonction qui réalise le calcul à l'instant *n* de la nouvelle sortie s_n , qui vérifie l'équation : $s_n = \sum_{k=0}^{NG-1} e_{n-k} \cdot g_k$.

1-cette fonction réalise 'un pas de filtrage fir', on l'appellera ***one_step_fir***
2- entrées-sorties de la fonction : pour calculer s_n , il faut connaître
  l'entrée e_n
  les coefficients g_k ,
  le nombre de coefficients *NG* du filtre
  les entrées passées : e_{n-k}, $pour\ k = 1..NG$
3- cette fonction aura pour rôle
  de calculer la sortie s_n
  mais aussi de mettre à jour les entrées passées e_{n-k} , pour la prochaine fois...

Organisation du code :
1- On commence par créer un type structure ***s_fir*** , regroupant les informations utiles à la programmation du filtre, de la façon suivante. (On définit également un type ***p_fir***: pointeur sur une structure ***s_fir***)

```
typedef struct {
                int NG;         // nombre de coefficients du filtre= ordre +1
                double *gn; // coeffs du filtre;
                double *en_k;      // memoires, ou etats, du filtre
```

} s_fir;
typedef s_fir *p_fir;

2- On crée les 3 fonctions qui permettront d'exploiter notre filtre FIR

```
// constructeur
      p_fir new_fir(int NG, double *gn){
        p_fir *f;
      //1- reservation memoire pour f, employer malloc() et sizeof()
      //2- transmission de l'adresse du tableau de coeffs gn, et de NG
      //3- reserv memoire pour entrees passees en_k[0..NG-1], et init a 0
        return f; // renvoie  f, pointant sur la structure initialisee
      }
// destructeur
      void destroy_fir(p_fir f){
      //1- liberation memoire pour les entrees passees en_k, utiliser free()
      //2- liberation memoire pour f, utiliser free()
      }
// exploitation: un pas de filtrage fir
      double one_step_fir(double en,p_fir f){
        double sn;
      //1-integration de en dans les memoires en_k
      //2-sn<- somme k=0 a NG-1 (gk).en_k[k]
        return sn;
      }
```

- ☐ écrire les 3 fonctions, ainsi que la déclaration de la structure ***s_fir***, dans le programme ***fir.c***

3.2.2 création d'une fonction de test , emploi du debugger

3.2.2.1 création d'une fonction de test

Avant de tester le vrai filtre *fir*, à plus de 250 coefficients, vous allez vérifier le bon fonctionnement de vos fonctions, sur un (des) exemple(s) simple(s), en implémentant la fonction ***teste_fir*** de la façon suivante

```
// 1-creation d'un tableau de coeffs gn de test NG_test=2, gn_test=[1,2];
  double gn_test[2];
  gn_test[0]=1;gn_test[1]=2;
  int NG_test=2;// nombre de coefficients
// 2-creation du pointeur fir_test pour ce tableau,
  p_fir fir_test=new_fir(NG_test,gn_test);
// 3-verification de one_step_fir sur quelques echantillons;
  double en,sn;
  en=1;sn=one_step_fir(en,fir_test); // pas 1
  en=0;sn=one_step_fir(en,fir_test); // pas 2
  en=0;sn=one_step_fir(en,fir_test); // pas 3
//4- liberation memoire fir_test
  destroy_fir(fir_test);
}
```

- ☐ - - écrire sur une feuille de papier les valeurs attendues pour *sn*, ainsi que pour le tableau *en_k,* pour les 3 premiers pas de filtrage 1,2,3.

3.2.2.2 insertion de points d'arrêt

un point d'arrêt (break point) est un endroit où l'on stoppe l'exécution en mode débuggage, pour pouvoir observer les valeurs des variables...
pour tester le bon fonctionnement , on va insérer un point d'arrêt à la ligne d'appel de la fonction ***teste_fir****, dans le programme* ***main.c.***

- ☐ éditer le fichier ***main.c,***
- ☐ effectuer un click gauche sur le numéro de la ligne de l'instruction ***teste_fir()***,

Dans la fonction **main()**. *cette ligne devrait se colorer d'un très joli rose bonbon, ce qui*

*indique la présence d'un point d'arrêt de programme, ou **breakpoint**.*
- vérification (gestion des points d'arrêt) :

- ☐ activer ***window->debugging-> breakpoint***. Vous devriez voir une fenêtre où s'affiche le programme et le numéro de ligne du break point que vous avez inséré.

*Ceci est très utile lorsque vous avez plusieurs points d'arrêt dans plusieurs fichiers (suppression, ajout, etc). En plus chez Sun ils ils sont très sympa parce-qu'ils ont coloré les **breakpoint** avec le même très joli rose bonbon...*

3.2.2.3 Lancement en mode debuggage (penser à regarder les raccourcis clavier)

- ☐ -activer le menu ***run-> debug main project***. *Le programme va se lancer, et s'arrêter sur votre point d'arrêt.*

À ce stade vous pouvez

*1- visualiser les variables locales, par exemple **en**, **NG**, en plaçant la souris dessus*
*2- visualiser la pile d'appel, les variables locales, désassembler le source,etc... : **window->debugging->faites votre choix***
3- avancer en mode pas à pas (instruction par instruction)
***run->step into** => pas à pas en entrant dans les fonctions (rentrera dans **teste_fir()**)*
***run->step over**=> pas à pas en n'entrant pas dans les fonctions (exécutera **teste_fir()** et s'arrêtera après*
***run->step out** => pas à pas sortant de la fonction courante (exécutera jusqu'à sortir de la fonction courante **main()**)*
*4- continuer l'exécution jusqu'au prochain point d'arrêt (rose bonbon, on n'insistera jamais assez sur cet aspect fondamental de l'IDE) :**run->continue***

- ☐ avancer en mode pas à pas en entrant dans la fonction ***teste_fir()***. Vérifier en mode pas à pas que les fonctions ***new_fir,one_step_fir, destroy_fir***, fonctionnent correctement. C'est le moment de comparer les valeurs écrites sur votre feuille de papier, aux valeurs obtenues en mode débuggage...

3.3 test du filtre fir calculé avec scilab

3.3.1 insertion des constantes générées par scilab au projet, et debuggage

- ☐ - ajouter aux fichiers source du programme un nouveau fichier source c vide: **constants_fir.c**
- ☐ ajouter aux fichiers header du programme un nouveau fichier header c vide: ***constants_fir.h***
- ☐ éditer (surtout pas avec ***netbeans***!... avec un éditeur de texte quelconque) le fichier ***toto_2321.c*** généré par le programme scilab ***fir.sce.*** Copier le contenu de ce fichier dans le programme ***constants_fir.c*** (édité sous netbeans)
- ☐ dans le fichier ***constants_fir.h*** *ajouter la définition des types et variables définies dans **constants_fir.c***

```
#define int_16 short int
#define int_32 long int
extern long int NG; // nombre de coeffs du filtre
extern int_16 gn_N[]; // coefficients entiers, on précise juste qu'il s'agit d'un tableau de int_16
```

et ainsi de suite...

- ☐ dans le fichier ***fir.c***, rajouter la directive
  #include "constants_fir.h"
- ☐ compiler et exécuter votre projet pour vérifier que tout est correct
- ☐ Modifier la fonction ***teste_fir()*** de façon à ce que le filtre *fir* testé corresponde au filtre *fir* généré par scilab. Ceci revient à appeler la fonction *new_fir* avec les coefficients *gn* et le nombre de coefficients *NG* définis dans *constants_fir.h (variables globales)*

- ☐ tester le bon fonctionnement à l'aide du debugger sur quelques échantillons lorsque l'entrée est une séquence impulsion unité...

  *- on veut tester le bon fonctionnement sur **NB_ECH**=10000 échantillons lorsque l'entrée est une séquence cosinusoidale d'amplitude **amp_e**=100, de fréquence **f_hz**=90hz,puis 110hz...*

- ☐ pour cela employer une boucle ***for*** dans la fonction ***teste_fir***, et y ajouter le code de génération d'une entrée ***en*** sinusoidale de fréquence ***f_hz,*** d'amplitude ***amp_e***, connaissant la période d'échantillonnage ***te***. { *les **noms** en **gras italique** doivent être des variables locales de la fonction teste_fir, initialisées aux valeurs demandées* }
- ☐ modifier également le code de *teste_fir* pour mesurer, en régime permanent
  la valeur maximum ***max_sn*** de la sortie ***sn***
  la valeur minimum ***min_sn*** de la sortie ***sn***
  la moyenne ***moy_sn*** de la sortie ***sn***
  la puissance(moyenne du carré) ***pow_sn***, de la sortie ***sn***
  l'écart-type ***sigma_sn*** de la sortie ***sn****(on emploiera le fait que le carré de l'écart-type est égal à la puissance moins le carré de la moyenne, un gand classique des probas...)*

  - pour effectuer ces mesures uniquement en régime permanent, il faudra les lancer uniquement lorsque l'indice *n* est supérieur à un indice de mesure ***n_mesure***, fixé arbitrairement: *dans votre cas choisir **n_mesure**=2*NG...*

  - l'affichage des résultats obtenus (***min***,***max***,***moy***,***pow***,***sigma***) sera effectué à l'aide de la fonction *printf* du langage c... *{ rappels: format %i ou %d => entier, format %f ou %e ou %g => réel, **\n** => saut de ligne}*

- ⇨ comparer les résultat obtenus aux résultats théoriques (gain du filtre), pour chacune des fréquences testées...

3.3.2 optimisation du code, emploi d'un buffer tournant, et d'auto-in(dé)crémentations

*Le code de la fonction **one_step_fir()** est très probablement assez mal écrit... essayez d'avoir un regard critique, en vous demandant ce que le processeur va devoir calculer pour le réaliser, et il ne devrait plus du tout vous satisfaire...*

3.3.2.1 buffer tournant: principe

le temps que le processeur passe à mettre à jour les entrées passées *en_k* est du temps perdu. Pour éviter de décaler réellement les entrées, on emploie généralement un *buffer* tournant, structuré comme suit:

Indice du buffer	Contenu de ***en_k*** à l'instant *n*	Contenu de ***en_k*** à l'instant *n+1*
0	...	
...	...	
i-2	$e_{n-[NG-2]}$	$e_{n-[NG-2]}=e_{[n+1]-[NG-1]}$
i-1	$e_{n-[NG-1]}$	$e_{n+1}=e_{[n+1]-0}$
i	e_{n-0}	$e_{n}=e_{[n+1]-1}$
i+1	e_{n-1}	$e_{n-1}=e_{n}=e_{[n+1]-2}$
i+2	e_{n-2}	$e_{n-2}=e_{n}=e_{[n+1]-3}$
...	...	...
NG-1		...

Seul cet élément a changé, entre *n* et *n+1*

Tableau 1: fonctionnement d'un buffer tournant

L'emploi de ce buffer pour stocker les entrées passées est moins intuitif que celui du buffer précédent { pour lequel on stockait e_n dans l'élément 0 à l'échantillon *n*}.
-> pour mettre à jour les entrées passées à l'instant *n*, il faut
1- stocker e_n dans le i^{eme} élément du buffer *en_k*.
2- décrémenter *i* pour le coup d'après { échantillon *n+1*}
3- si $i=0$, le réinitialiser à $i=NG-1$
pour que ce système fonctionne, il faut de plus garder en mémoire l'indice courant *i*, et l'initialiser...

3.3.2.2 auto-incrémentation des coeffs, principe

Le travail qu'effectue le processeur pour le calcul de l'adresse de ***gn[k]*** et ***en_k[k]*** doit également être éliminé... Pour cela on emploie généralement des boucles avec auto-in(dé)crémentation de pointeurs...
par exemple, au lieu de programmer

```
for (k=0;k<NG;k++) {
 s=s+g[k]*e[k];
}
```

on aura tout intérêt à programmer(lire attentivement les remarques svp)...

```
// le même code avec auto incrementation de g et e :
   // g++ signifie: le pointeur g, qu'on incremente de 1 element
   // aussitot après y avoir accédé
   // *(g++) doit se lire
   // 1- ce qui est pointé par g : *g,
   // 2- puis on incrémente g d'un élément
   // s+=patati patata se lit : s=s+patati patata,
   // mais le calcul de l'adresse de s n'est effectué qu'une seule fois..
   for (k=0;k<NG;k++) {
     s+=(*(g++)) * (*(e++));
   }
```

3.3.2.3 intégration au filtre fir

1- intégration du buffer tournant dans le programme...

☐ - modifier la structure ***s_fir*** pour y intégrer
1-l'indice courant *i* du buffer tournant .
2-Le nombre d'échantillons *N2=NG-i* après l'échantillon *i* inclu

3-l'adresse courante ***ei*** dans laquelle on doit ranger *l'entrée courante en*

- ☐ modifier la fonction new_fir, en y ajoutant l'initialisation de *i*, *N2* et *ei*
  i sera initialisé à 0, on en déduira l'initialisation de N2 et ei
- ☐ modifier la fonction ***one_step_fir*** de façon à employer les mémoires ***en_k*** comme un buffer tournant (vous inspirer de la trame ci-dessous)

```
   //-------------------------------------------------------
//0- e et g sont des pointeurs de double
//1- e pointe sur ei, g pointe sur gn, sn est initialisee à 0
//2- on range l'entrée en dans e[0]
//3- pour k allant de 0 à N2-1, faire
//   sn=sn+ *(e++) . *(g++) , vive l'auto-incrémentation
//4- e pointe sur en_k[0], on ne modifie pas g
//5- pour k allant de 0 à i-1 faire
//   sn=sn+ *(e++) . *(g++) , et encore vive l'auto-incrémentation
//6- mise a jour de i, N2 , ei pour le coup d'apres
//    si i>0 alors
//       decremente i et le pointeur ei, incremente N2
//    sinon on reinitialise
//      i=NG-1;N2=1;ei= adresse de en_k[NG-1]
//    finsi
```

- ☐ ***vérifier*** le fonctionnement correct du buffer tournant, en pour les 2 ou 3 premiers échantillons à l'aide du debugger...

3.4 codage du filtre FIR avec des coefficients entiers (et comparaison avec le *fir* ideal)..

On va dans cette partie comparer les performances du fir idéal avec celles du fir en nombres entiers calculé avec scilab.

On a 2 choix possibles,

choix 1-coder les fonctions relatives au fir en nombres entiers dans un nouveau fichier ***fir_entier.c*** *avec un header* ***fir_entier.h****,*

choix 2-considérer que tout ce qui est relatif aux filtres fir, entiers ou réels, est codé dans le fichier ***fir.c****..*

C'est ce 2 ème choix qui a été retenu par le prof, uniquement par manque de temps, parce-qu'au fond s'il avait eu un vrai logiciel de filtrage à développer, c'est le choix 1 qu'il aurait retenu...

3.4.1 préparation de la programmation des fonctions entières, un peu de méthode

les ¾ du travail de la programmation des filtres fir ont déjà été effectués avec les réels double précision, et ce serait très dommage de ne pas en profiter... pour cette raison

- ☐ dupliquer (copier-coller) la structure ***s_fir***, et les fonctions ***new_fir***,***destroy_fir***,***one_step_fir*** dans le fichier ***fir.c*** (attention, c'est le moment dangereux, veiller à ce que les copies forment un bloc contigu...)
- ☐ sélectionner l'ensemble de ce que vous avez dupliqué...
- ☐ dans le bloc sélectionné, remplacer la chaine de caractères *fir*, par la chaine de caractères *fir_entier* (***edit->replace***: ***match case*** actif, ***whole word*** inactif, ***réduit à la sélection*** actif)
  - tant qu'on y est, (toujours uniquement sur le bloc sélectionné!...)
  - remplacer la chaine de caractères *double*, par la chaine de caracteres *int_16 (****whole word*** actif*)*
  - remplacer la chaine de caractères *gn*, par la chaine de caracteres *gn_N (****whole word*** actif*)*
  - remplacer la chaine de caractères e*n*, par la chaine de caracteres en_16 *(****whole word*** actif*)*

vous venez d'effectuer les ¾ du travail d'implémentation du filtre fir en nombres entiers...

3.4.2 modification de la structure et des fonctions

- ☐ - modifier la structure **s_fir_entier** : ajouter les champs nécessaires à la programmation du filtre fir avec des coefficients entiers et facteur d'échelle de signal...{ même analyse qu'en 3.2.1 , mais cette fois-ci débrouillez-vous...}
- ☐ modifiez la fonction **new_fir_entier**() : entrées,réservation de mémoire et initialisation des champs de la structure
- ☐ modifiez la fonction **destroy_fir_entier**() : libération de mémoire
- ☐ modifiez la fonction **one_step_fir()**: elle doit à présent correspondre à la programmation avec buffer tournant et auto-incrémentation du filtre fir avec coefficients entiers et décalages à droite. Ne pas oublier la multiplication par $2^{-LSIGNAL}$ avec quantification par arrondi, de la sortie... L'accumulateur(variable interne V) doit être un entier codé sur 32 bits (ils serait judicieux de la renommer *v_32* , ou quelque chose dans ce genre)

3.4.3 comparaison du fir en nombres entiers et du fir avec coefficients réels

- ☐ modifiez la fonction **teste_fir** de façon à y calculer simultanément la sortie **sn** du filtre fir avec coeffs réels, et la sortie **sn_16** du filtre fir avec coefficients entiers...
  - pour être équitable, on arrondira l'entrée *en* à l'entier le plus proche avant de l'envoyer à la fonction **one_step_fir**, en employant par exemple le code ci-dessous ...

```
en=floor(en+0.5); // arrondi(x)=troncature(x+0.5), entrée de one_step_fir
en_16=(int_16) en;// entree de la fonction one_step_fir_entier
```

  - l'amplitude **amp_en** sera choisie aussi grande que possible (32767)

  soit **epsn= sn-sn_16** l'erreur entre la sortie réelle et la sortie entière,
  évaluer en régime permanent (même principe qu'en 3.3.1) :
  la moyenne **moy_eps** de **epsn**
  les valeurs maximale **max_eps** et minimale **min_eps** de **epsn**
  la puissance **pow_eps** de **epsn**
  l'écart-type **sigma_eps** de **epsn**
- ☐ vérifier que le fonctionnement est satisfaisant pour les fréquences 90hz et 110hz..
  relever les valeurs obtenues dans un tableau...

3.4.4 emploi d'un seul facteur d'échelle (voir 2.3.2.3)...

- ☐ copier le contenu du fichier **toto_2323.c** dans le fichier **constants_fir.c** (en écrasant son contenu précédent).
- ☐ relever la valeur de *Lg_moins_L* , et modifier le code de la fonction **one_step_fir_entier** conformément à cette valeur (mettre en commentaires le décalage à droite qui ne sert plus à rien, ne pas l'effacer)
- ☐ effectuer les même tests qu'en 3.4.3 , et relever les valeurs dans un tableau...

⇨ les résultats sont-ils en accord avec les commentaires en 2.3.2.4 ?...