INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES ROUEN NORMANDIE

Projet de Physique P6
STPI/P6/2019 - 42

Programmation informatique embarquée : Initiation au DSP

Étudiants :
Mehdi Helal
Jade Freneau
Badr Yacoubi
Louis Dispa
Laura-Fayna Rodriguez
Clément Blanco-volle

Enseignant-responsable du projet :
Richard GRISEL

Date de remise du rapport : 17/06/2019

Référence du projet : STPI/P6/2019 – 42

Intitulé du projet : programmation informatique embarquée : Initiation au DSP

Type de projet : Informatique et électronique

Objectifs du projet :
— S'initier au fonctionnement des microcontrôleurs
— Apprendre à utiliser les fonctionalités de l'IDE MPLAB
— Filtrage d'un signal analogique

Mots-clés du projet : Informatique embarquée / Traitement d'un signal

Institut National des Sciences Appliquées de Rouen
Département Sciences et Techniques Pour l'Ingénieur
685 Avenue de l'Université BP 08 - 76801 Saint-Étienne-du-Rouvray
Tél : +33 2 32 95 66 21 - Fax : +33 2 32 95 66 31

Table des matières

Notations et Acronymes

DSP : Digital Signal Processor
CAN : Convertisseur Analogique-Numérique
CNA : Convertisseur Numérique-Analogique
SAR : Successive Approximation Register
IDE : Integrated Development Environment
DMCI : Data Monitor and Control Interface
GBF : Générateur Basses Fréquences
SPI : Serial Peripheral Interface

Introduction

Le but de ce projet a été de nous initier au fonctionnement d'un élément électronique complexe, conçu et optimisé pour le traitement de signaux numériques : "le processeur de signal numérique" ou "Digital Signal Processor (DSP)". Le DSP possède des applications toutes trouvées notamment dans le domaine des télécommunications, du traitement audio et vidéo, de la navigation, etc... Ce processeur peut aujourd'hui être trouvé dans les modems, les téléphones portables, et de manière plus générale, dans l'électronique grand public.

Le DSP permet donc d'appliquer des opérations mathématiques sur des signaux numériques représentés sous forme de séquences d'échantillons (provenant d'un signal analogique par exemple). Il est adapté au traitement d'informations en temps réel grâce à un débit mémoire important (notamment pour l'accès à des tableaux) et a été conçu pour garder une précision numérique satisfaisante pour le grand nombre de calculs effectués. Dans ce projet nous nous sommes intéressé au processus de filtrage d'un signal à l'aide du DSP.

Notre objectif au terme de cette initiation a donc été de filtrer un signal analogique (préalablement converti en un signal numérique). Pour cela nous disposions d'un accès aux salle de TP situées dans le bâtiment Dumont-Durville pendant une heure et demi chaque semaine. Des ordinateurs munis de l'IDE MPLAB ainsi que des microcontrôleurs nous étaient également fournis pour que nous puissions expérimenter.

Chapitre 1

Methodologie et organisation du travail

1.1 Le Projet

Nous avons réalisé ce dossier dans le cadre du projet P6, obligatoire pour la validation du quatrième semestre. Pour réaliser ce projet, nous avions un créneau réservé le lundi de 16h45 à 18h15 sous la tutelle de M. Grisel. Le sujet du projet traitait de nombreux éléments que nous n'avions jamais rencontrés au cours de nos études. Notre premier objectif a donc été d'appréhender les notions sur lesquelles nous allions travailler par la suite. Le projet étant considérablement dense, le temps de recherche que nous avons consacré au cours de cette première étape de travail a été important. En effet, c'est ce qui nous a occupé lors des deux premiers cours. Cette première phase nous a permis de comprendre le fonctionnement des Microprocesseurs en général, et plus particulièrement du « Digital Signal Processor (DSP) ».

1.2 Outils et premiers pas

Durant ce projet, nous avons pu apprendre comment utiliser l'environnement de développement intégré « IDE », MPLAB. MPLAB sert notamment à coder et à simuler l'exécution d'un programme basique sur un microcontrôleur.

Ce projet étant une réalisation en équipe, nous avons dû élaborer des stratégies et des méthodes de travail afin de faciliter sa réalisation. Comme le projet fait intervenir plusieurs domaine de l'informatique, notre première approche fut de diviser le projet en quatre sous-projets. Dans un premier temps, nous avons appris à utiliser des sorties (LATA), à écrire dans un registre et à utiliser des Watchs pour surveiller le contenu des variables utilisées. Pour réaliser tout cela, nous avons interagi avec des LED du DSP.

Ensuite, nous avons manipulé les entrées analogiques en lisant une température. Puis, nous avons appris à lire un stimulus virtuel lors d'une simulation, et, enfin, nous avons mis en place un filtre.

Pour chaque sous-projet, la méthodologie a été la même. Avant de débuter la première séance consacrée à un nouveau sous-projet, notre enseignant nous communiquait des documents à étudier. Ces derniers nous permettaient de mieux comprendre les nouveaux concepts auxquels nous allions être confrontés avant de débuter notre travail.

1.3 Déroulement du projet

Ensuite, au cours des séances, notre tuteur nous expliquait les notions et informations importantes à maîtriser pour notre projet. Il nous indiquait également les expériences et les manipulations que nous devions effectuer. Enfin, après avoir assimilé et compris toutes les notions nécessaires à la réalisation des expériences, nous passions à la pratique.

Pour mener à bien ce projet, nous avions besoin de nous communiquer les différentes informations de recherche sur lesquelles chacun d'entre nous travaillait. Nous avons donc choisi d'utiliser Messenger et Google Drive comme moyen de communication car nous maîtrisons toutes ces plateformes. Nous y avons créé un groupe de travail, et, ainsi, nous avons pu avoir accès au travail de chacun et partager notre avancement au reste du groupe.

Afin d'être efficace et d'apprendre à travailler en équipe, nous nous sommes réparti les différentes tâches à accomplir. Dans un premier temps, nous avons travaillé tous ensemble durant les créneaux horaires dédiés

au projet P6. Cela nous a permis de mieux comprendre les notions de base du sujet avant de nous plonger pleinement dans le projet.

Par la suite, nous avons décidé de séparer le travail en deux : un groupe s'occupant de la recherche, prenant des notes lors des explications du professeur afin d'être capable de refaire les manipulations, et un autre groupe se chargeant de faire les manipulations en question.

Le premier groupe est composé de Jade Freneau, Laura-Fayna Rodriguez et Mehdi Helal et le second groupe est composé de Badr Yacoubi, Louis Dispa, et Clément Blanco-Volle. Au terme du projet, nous avons mis en commun tous les travaux effectués. Enfin, nous avons consacré du temps tous ensemble en dehors du créneau de projet du Lundi après-midi pour faire un bilan, vérifier l'ensemble de nos manipulations, mais également retravailler de notre côté.

Nous avons donc favorisé un travail de groupe plutôt qu'individuel, nous permettant ainsi de mieux avancer dans notre projet.

Chapitre 2

Travail réalisé

2.1 Découverte

2.1.1 Le DSP

Préambule

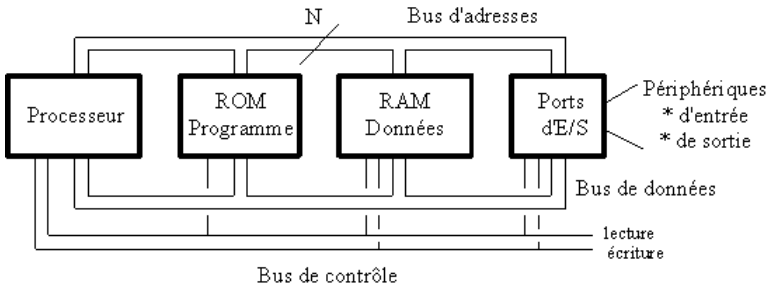
Un DSP (Digital Signal Processor ou processeur de signal numérique) est un microprocesseur chargé d'effectuer des applications de traitement numérique du signal. Ils sont ainsi utilisés dans les modems, les smartphones, GPS...

Pour notre projet P6, nous avons étudié le DSPIC33FJ256GP710A, un microcontrôleur conçu pour réaliser des filtres numériques et des boucles de commande numériques de précision à haute vitesse. Il contient un petit système informatique.

On retrouve un processeur qui centralise et effectue les tâches, des ports d'entrée-sortie permettant de piloter de systèmes et de relever des valeurs (comme la température), différentes unités périphériques permettant d'interagir avec les autres éléments d'un ordinateur et un oscillateur interne pour cadencer le rythme d'exécution des instructions.

D'autre part, il dispose d'une mémoire ROM (read only memory) pour stocker son programme et des instructions fixés lors de sa conception ainsi que d'une mémoire RAM (random access memory) permettant de stocker les variables au cours de l'exécution d'un programme et d'effectuer des calculs. Cette dernière évolue au cours du temps et peut être modifié.

FIGURE 2.1 – Schéma de l'architecture d'un microcontrôleur



source : https://fr.wikipedia.org/wiki/Microcontr%C3%B4leur

Notons que tous ces éléments sont reliés par des « bus ». Parmi eux, le bus d'adresse permettant au DSP d'accéder à un emplacement mémoire pour y lire ou écrire une information. Encore, le bus de données est chargé du transfert des informations.

Cependant, un microcontrôleur est inutilisable tant qu'il n'a pas été programmé! Nous devons donc écrire un programme à l'aide d'un environnement de développement et d'un langage de programmation. Pour notre part, nous utilisons MPLAB et codons en C. Il sera ensuite traduit en langage binaire par le compilateur afin d'être compréhensible par le microcontrôleur. Il pourra alors le lire ligne par ligne.

Caractéristique du DSP utilisé pour le projet

La documentation fourni par Microchip concernant le dsPIC33FJ256GP710A nous indique les caractéristiques de ce dernier que nous avons résumé dans un tableau :

TABLE 2.1 – Caractéristique du DSP

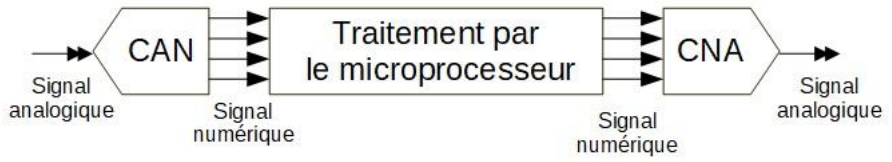
Architecture	16-bits
Fréquence du processeur	40MHz
RAM	256 KB
CAN	12 bits
Intervalle de température	$-40 \rightarrow 150°C$

2.1.2 Les convertisseurs

Généralités

L'électronique est divisée en deux sous-domaines, le domaine du numérique et le domaine de l'analogique. Les microprocesseurs ainsi que tous les autres appareils électroniques ne traitent que les données numériques. Il est donc primordiale que les données en entrée soient sous forme numérique (codées en binaire). Cependant, la plupart des signaux (une tension, un signal sonore, etc) sont des données analogiques. Il faut donc convertir ces données. Pour cela, on utilise les convertisseurs. Il en existe deux familles. Les CAN (convertisseurs analogique-numérique) et les CNA (convertisseurs numérique-analogique).

FIGURE 2.2 – schéma de l'utilisation des convertisseurs

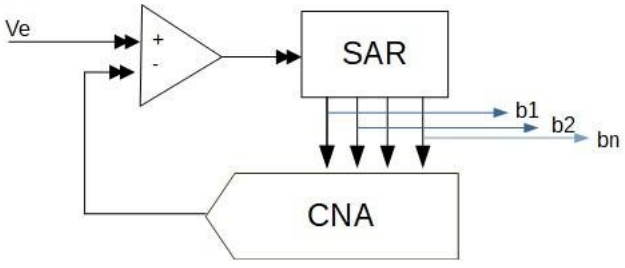


source : https://www.emse.fr/~dutertre/documents/cours_convertisseurs.pdf

Le convertisseur analogique-numérique (CAN)

Nous savons que les ordinateurs ne peuvent pas gérer les signaux analogiques tels quels, ils utilisent donc pour cela le CAN. Cet élément consiste en un montage électronique qui convertit les signaux analogiques en signaux numériques codés sur plusieurs bits. Il existe plusieurs façon de convertir ces signaux. Le CAN que nous avons utilisé est un convertisseur à approximations successives ou en anglais Successive Approximation Register (SAR). Il peut être schématisé comme ci dessous. On remarque qu'il est couplé à un convertisseur numérique-analogique (CNA).

FIGURE 2.3 – schéma du CAN



source : http://courelectr.free.fr/CONV/CONV.HTM

Le principe du codage de ce convertisseur est de déterminer de façon successive tous les bits qui constituent le nombre numérique correspondant à la tension d'entrée V_e. Pour cela on réalise une dichotomie en partant

du bit de poids le plus fort (MSB). A chaque coup d'horloge, si la tension en sortie du CNA est inférieure à la tension en entrée du CAN (V_e) on divise l'intervalle en deux, et on garde le premier bit à 1. La comparaison est effectuée grâce à un comparateur. On réalise cette opération successivement jusqu'à atteindre le bit de poids faible (LSB). On obtiendra finalement le nombre numérique décrivant le signal analogique.

Le convertisseur numérique-analogique

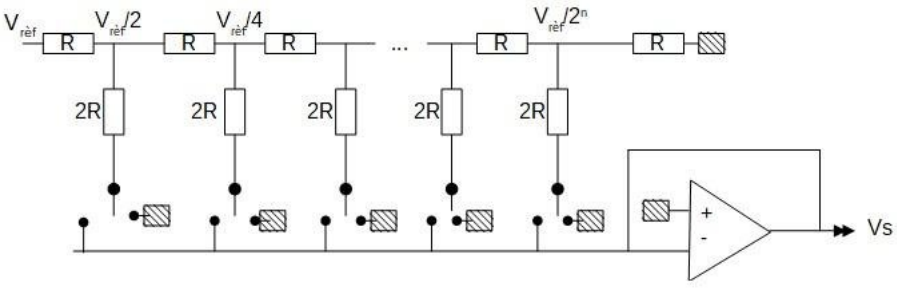
L'objectif du CNA est l'inverse de celui du CAN. En effet, ici nous disposons de valeurs numériques qu'il est nécessaire de reconvertir en données analogiques. Il est important de signaler que les données en sortie du microprocesseur n'ont pas forcement besoin d'être reconverties, tout dépend de l'utilisation que l'on souhaite en faire. Le CNA que nous avons utilisé est un convertisseur à réseau résistif (à échelle $R-2R$).

FIGURE 2.4 – schéma général du CNA

source : http://courelectr.free.fr/CONV/CONV.HTM

Le principe de ce codage est de fermer des interrupteur suivant la valeur du bit. Pour un bit à 0 on se connecte à la masse, pour un bit à 1 on se connecte à l'amplificateur et la valeur du courant est de $\frac{V_{ref}}{2n}$ avec n qui correspond au bit de l'interrupteur. Grâce à cet amplificateur on peut transformer le courant Itotal en tension de sortie Vs, car l'amplificateur additionne chaque courant. On remarque que le premier interrupteur (à gauche) est utilisé pour MSB et que les bits décroissent vers la droite jusqu'à arriver au LSB.

FIGURE 2.5 – schéma détaillé du CNA



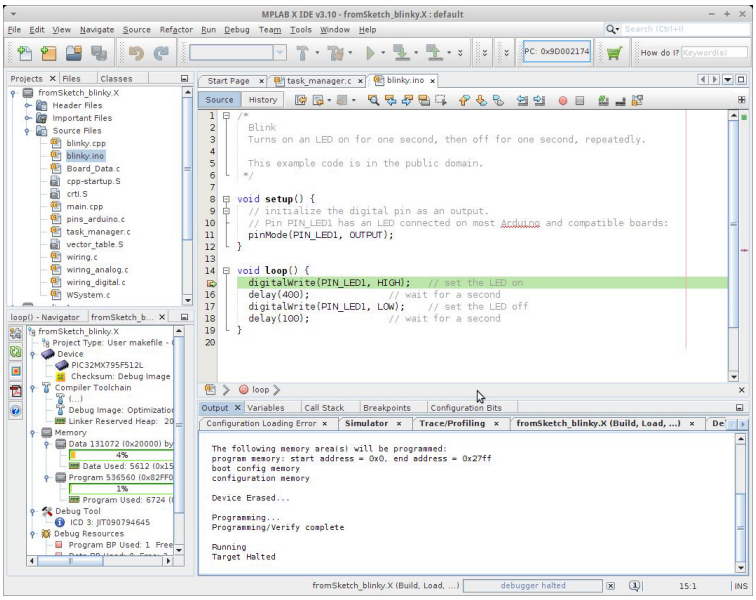
source : http://courelectr.free.fr/CONV/CONV.HTM

2.2 MPLAB

2.2.1 Description de l'IDE

L'environnement de développement intégré que nous avons utilisé afin de réaliser notre filtre se nomme MPLab, développé par Microchip Technology et basé sur l'IDE NetBeans. Il s'agit d'un logiciel offrant un éditeur de texte, un gestionnaire de projet, un compilateur, des fonctions de débogage, et bien plus. Notre microprocesseur possède une architecture 16-bit : nous utilisons donc le compilateur XC16, adapté à la programmation embarquée.

FIGURE 2.6 – Capture d'écran de l'interface



L'une des fonctionnalités les plus utiles d'MPLab est son simulateur, qui nous permet de tester notre code avant de le téléverser sur notre microprocesseur. Ce mode nous offre la possibilité de simuler des entrées analogiques, d'observer l'état de nos variables, et bien d'autres options facilitant le développement.

Il faut cependant faire attention à certains paramètres ne pouvant complètement être simulés : la vitesse d'horloge, par exemple, diffère entre le simulateur et notre microcontrôleur réel, ce qui peut entraîner des bogues lors de l'acquisition des valeurs sur nos entrées par exemple.

2.2.2 Utilisation

Afin de nous initier à la programmation sur DSP, nous avons tout d'abord réalisé quelques programmes utilisant les fonctionnalités de base de cette carte. Ainsi, nous avons créé un programme faisant osciller les huit LED de l'appareil. Cela nous a permis de mettre en oeuvre plusieurs concepts liés à la programmation embarquée :

- Les différents paramètres de configuration sont gérés par des flags, variables contenues dictant une grande partie du comportement de notre carte, allant de la vitesse d'horloge à l'activation de l'acquisition. Il est généralement plus simple de regrouper tous ces éléments dans une fonction de configuration.
- L'écriture de valeurs à travers nos entrées analogiques passe simplement par l'insertion d'un ou plusieurs octets dans des registres, comme le registre LATA, affichant une représentation binaire de l'octet contenu sur les LEDs de l'appareil lorsqu'on y écrit une certaine valeur.
- La gestion de l'horloge et des délais est importante lorsque nous travaillons avec une interface utilisateur : sans interruption momentanée, par exemple, la vitesse de clignotement des LEDs est bien trop importante pour voir un changement à l'oeil nu.

Par la suite, nous avons implémenté une fonction, analogRead(), nous permettant de lire une valeur appartenant à l'un des registres mentionnés précédemment. Celle-ci est un peu plus compliquée, puisque l'échantillonnage de valeurs se déroule en plusieurs étapes :

```
int  analogRead(char analogPIN)
{
        // AD1CHS<4:0> controls which analog pin goes to the ADC
        AD1CHS0bits.CH0SA = analogPIN;
        AD1CON1bits.SAMP = 1; // Begin sampling auto
        Delayc(2000); // Appel de la fonction de délai
        AD1CON1bits.SAMP = 0; // Fin échantillonnage et début de la conversion
        while( ! AD1CON1bits.DONE ); // wait until conversion done
        AD1CON1bits.DONE = 0; // RAZ bit DONE
        AD1CON1bits.SAMP = 1;
```

```
        return ADC1BUF0; // result stored in ADC1BUF0
}
```

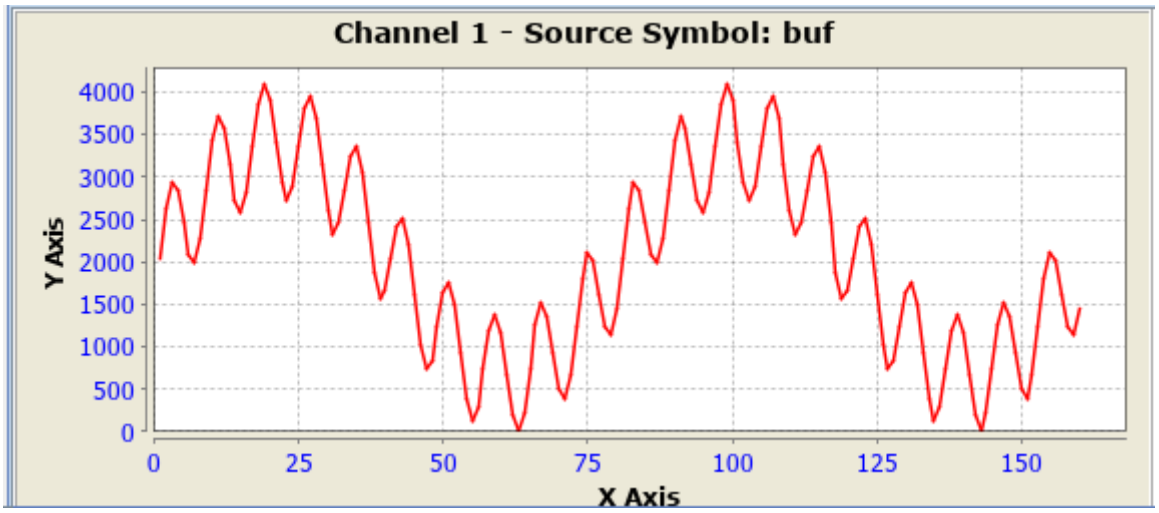
Notre fonction prend en entrée un indice de PIN, désignant la valeur que nous souhaitons lire (température, tension, ...). Nous démarrons ensuite l'acquisition à l'aide du flag AD1CON1bits.SAMP. Nous utilisons une fonction de délai afin d'être sûr que l'acquisition se soit terminée, puis nous arrêtons l'échantillonnage avant de retourner la valeur lue. Ceci nous a ainsi permis d'écrire un second programme permettant de lire la température au sein de la pièce.

2.3 Mise en place d'un filtre numérique

2.3.1 Manipulation des entrées analogiques

Avant de nous lancer dans le traitement d'un signal réel avec un DSP physique nous avons d'abord dû simuler le processus d'échantillonnage et de traitement du signal dans MPLAB. Notre tuteur nous a donc fourni des fichiers .txt contenant les valeurs à apposer en entrée de la simulation (dans MPLAB nous utilisons la fonctionalité « stimulus »). A l'aide de la fonction *analogRead* nous récupérons les valeurs du stimulus. Nous stockons les valeurs échantillonnées dans un tableau (buf) puis nous visualisons les valeurs contenue dans ce tableau grâce à l'extension DMCI de MPLAB.

FIGURE 2.7 – Capture d'écran DMCI



2.3.2 Filtrage

Avantages et inconvénients :

Le filtrage numérique comporte plusieurs avantages par rapport au filtrage analogique :

— Un filtre numérique est plus souple puisque la manière dont il va se comporter dépend entièrement de plusieurs coefficients aisément interchangeables dans l'algorithme. Ce type de filtre est entièrement programmable et donc par conséquent il n'est pas nécessaire de changer le circuit pour modifier ses caractéristiques.
— Les simulations avant l'application réelle sont très facile à mettre en oeuvre sur un ordinateur.
— La mise en série de filtres pour créer des systèmes complexes est facilitée.
— Le filtre ne sera pas altéré par le temps ou la température ou par d'autres facteurs liés à l'environnement.

Principe du filtrage :

FIGURE 2.8 – Diagramme

Ici $y(n) = h(n) \times y(n)$ avec $h(n)$ que l'on appelle la réponse impulsionnelle du filtre. Dans le cas du filtrage numérique $h(n)$ peut parfois être infini et poser des problèmes au niveau du calcul. Cependant, dans le cadre de notre projet nous nous sommes intéressés au filtre à réponse impulsionnelle finie.

Le filtre à réponse impulsionnelle finie (RIF) :

C'est un filtre dit « causal », c'est à dire que le signal de sortie de ce filtre ne dépend que des valeurs précédentes du signal d'entrée mais pas des valeurs futures de ce dernier. Ce type de filtre est utilisé dans le cas d'une limitation du nombre de valeurs échantillonnée et est toujours stable :

$$\lim_{n \to \infty} h(n) = 0$$

Pour la mise en place d'un RIF de longueur N. Nous utiliserons la relation suivante :

$$y(n) = \sum_{k=0}^{N-1} x(n-k)\, h(k)$$

Pour mettre en oeuvre un RIF il nous suffit donc de fixer les coefficients $h(k)$ et d'appliquer cette relation à $y(n)$.

Le programme :

```
// On initialise les paramètres du filtre
    C[0] = 0.071563720703125;
    C[1] = 0.09909057617875;
    C[2] = 0.12176513671875;
    C[3] = 0.13665771484375;
    C[4] = 0.141845703125;
    C[5] = 0.13665771484375;
    C[6] = 0.12176513671875;
    C[7] = 0.09909057617875;
    C[8] = 0.071563720703125;
```

Initialisation des paramètres du filtre ($h(k)$) : Ces valeurs de coefficients fournies préalablement par notre tuteur correspondent à un filtre passe bas. Nous nous serviront de cette considération pour tirer des conclusions sur le fonctionnement de notre filtre après expérimentation.

```
i = 0;
    while(1)
    {
        //échantillonage et conversion (buf contient le signal en entrée)
        buf[i] = analogRead(1);
        //y est notre tableau contenant le signal de sortie
        y[i] = 0;
        //calcul de la valeur y [i] à partir des valeurs précédentes de buf
        for (k = 0; k<9; k++)
        {
            if (i-k < 0)
            {
                y[i] += C[k] * buf[i-k+160];
            }
            else
            {
                y[i] += C[k] * buf[i-k];
            }
```

```
            //on appose ensuite la valeur de y en sortie
            SPI2BUF = y[i];
        }
        i++;
        if (i == 160)
        {
            i = 0;
        }
    }
```

Construction de la boucle de traitement :

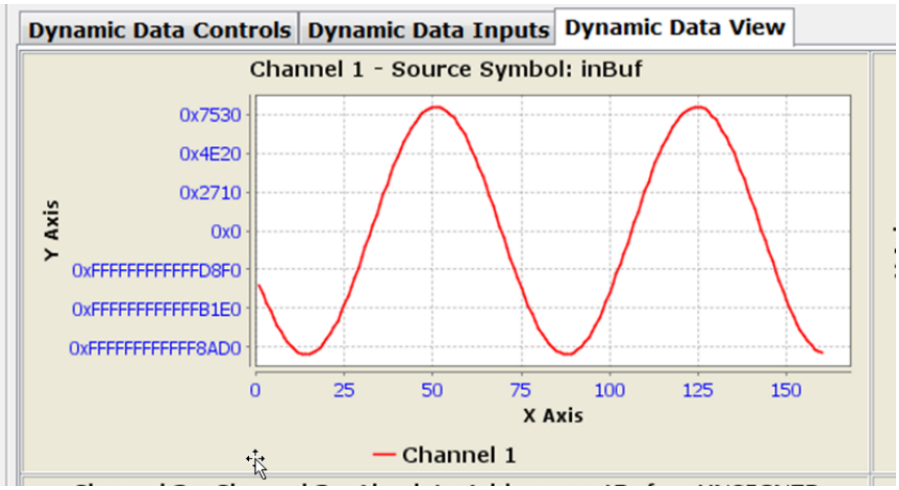
Dans cette boucle on échantillonne d'abord une valeur puis on la convertit grâce à la fonction analogRead. Ensuite nous calculons la valeur du signal de sortie (dans le tableau buf) à partir des valeurs précédentes du signal en entrée (dans le tableau y). Puis on appose la valeur calculée en sortie de la carte.

2.3.3 Traitement d'un signal réel

2.3.3.1 Acquisition du signal

Pour étudier le filtrage dans un cas pratique, nous avons utilisé un générateur de basses fréquences (GBF) pour donner un signal au microcontrôleur. Nous avons connecté le GBF à un pin analogique de la carte. Le microcontrôleur, ayant un convertisseur analogique-numérique, convertit ce signal et le stocke dans le tableau du programme. À l'aide du DMCI nous avons pu observer l'acquisition du signal et ainsi calculer la fréquence d'échantillonnage du microprocesseur. Pour une période d'une onde de 100 Hz le microcontrôleur fais 75 échantillons donc la fréquence d'échantillonnage est 7500 Hz.

FIGURE 2.9 – Capture d'écran du DMCI

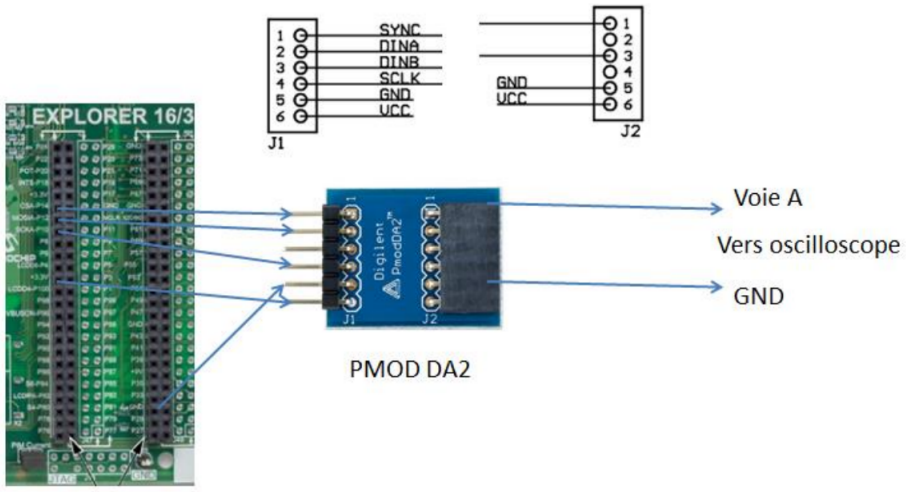


Nous avons pu calculer la fréquence de coupure qui correspond à la fréquence d'échantillonnage divisée par 40 pour le filtre passe-bas que nous avons programmé donc $\frac{7500}{40} = 187,5$ Hz . Ainsi, nous avons commencé les mesures par une fréquence basse, 100 Hz. Puis en augmentant la fréquence nous avons observé une diminution de l'amplitude du signal après le filtrage jusqu'à la coupure.

2.3.3.2 Restitution du signal

Pour restituer le signal une fois filtré il faut un convertisseur numérique-analogique or le microcontrôleur n'en possède pas. Nous avons donc dû relier un convertisseur externe (PMOD DA2) au microcontrôleur par le bus SPI2 (Serial Peripheral Interface).

FIGURE 2.10 – Convertisseur externe relié à la carte [10]

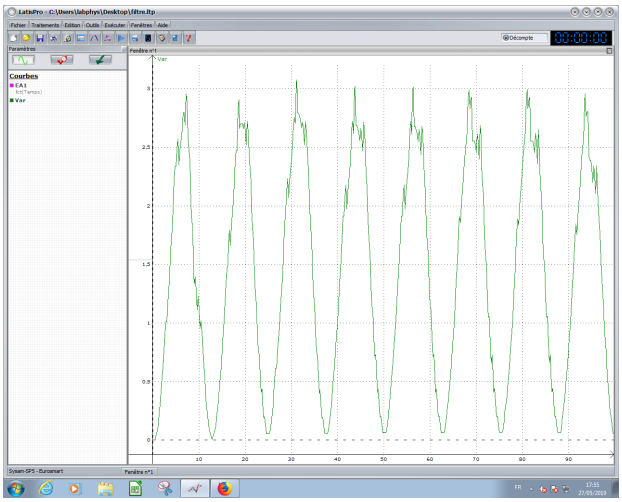


Une fois tous les paramètres du bus SPI2 fixés dans le programme, il suffit d'écrire la valeur après-filtrage dans SPI2BUF.

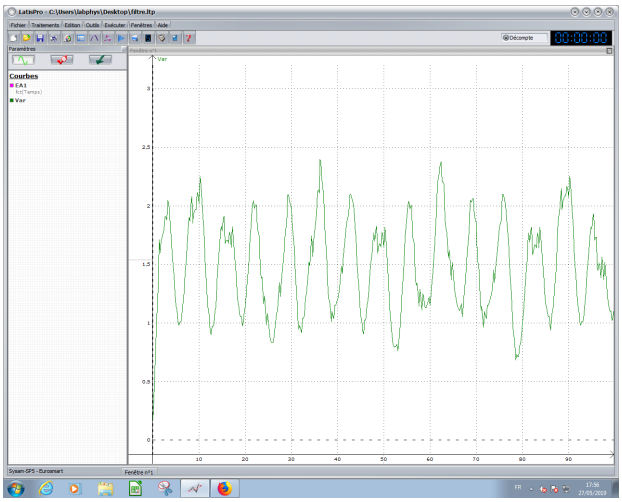
Nous avons donc effectué tous les branchements et compilé le code final sur la carte. Puis, pour mesurer la sortie analogique du filtre nous avons utilisé Latis pro. On observe bien que pour le GBF programmé à une fréquence de 80 Hz, il n'y a pas de diminution d'amplitude. Par contre pour 150 Hz l'amplitude a beaucoup diminué. Nous pouvons donc en déduire que notre filtre est opérationnel.

FIGURE 2.11 – Visualisation du signal en sortie

(a) 80 Hz



(b) 150 Hz



Chapitre 3

Conclusion et perspectives

Ce projet nous aura permis de nous familiariser avec les microcontrôleurs et la notion de filtrage numérique. Nous avons réussi à programmer et à mettre en place un filtre numérique (RIF) que nous avons testé puis appliqué à un signal analogique réel (généré par un oscilloscope). Puisque la majorité des membres du groupe souhaitent accéder au département ASI, l'accomplissement de ce projet nous aura conforté dans notre choix tout en nous permettant de mieux comprendre comment utiliser un microcontrôleur. Si nous disposions de plus de temps pour compléter ce projet nous aurions pu mettre au point une démonstration pratique de notre filtre :

— un accordeur de guitare
— application du filtre à un signal sonore (avec microphone)

Bibliographie

[1] Cours de pic. http://bravo.univ-tln.fr/pic/Cours%20de%20PIC%20G%E9n%E9ralit%E9s.pdf, Date de consultation : juin 2019.

[2] Les microcontrôleurs pic. https://www.zonetronik.com/les-microcontroleurs-pic/, Date de consultation : juin 2019.

[3] Conception avancées des circuits intégrés analogiques.convertisseurs a/n et n/a. https://www.emse.fr/~dutertre/documents/cours_convertisseurs.pdf, Date de consultation : Mai 2019.

[4] Microprocesseur. https://www.supinfo.com/articles/single/1136-microprocesseur, Date de consultation : Mai 2019.

[5] Microchip documentation dspic. https://www.microchip.com/, Date de consultation : Mars 2019.

[6] Dominique Chevallier. Convertisseur n/a et a/n. http://courelectr.free.fr/CONV/CONV.HTM, Date de consultation : Mai 2019.

[7] R. Fontenay. *Convertisseurs : numérique – analogique, analogique – numérique.* éditions Radio, 1979.

[8] Pierre-Emmanuel Hladik. Comprenez les spécificités d'une architecture microcontrôleur. https://openclassrooms.com/fr/courses/4117396-developpez-en-c-pour-lembarque/4633171-comprenez-les-specificites-d-une-architecture-microcontroleur, Date de consultation : Mai 2019.

[9] Haggège Joseph. *Initiation aux microprocesseurs et aux microcontrôleurs.* éditions Ellipses, 2018.

[10] Wikipédia. Microcontrôleur. https://fr.wikipedia.org/wiki/Microcontr%C3%B4leur_PIC, Date de consultation : Avril 2019. 2.10

[11] Wikipédia. Filtre (électronique). https://fr.wikipedia.org/wiki/Filtre_(%C3%A9lectronique), Date de consultation : Mai 2019.

Annexes

3.1 Codes

3.1.1 Delay

```
unsigned int temp_count;
void Delay( unsigned int delay_count )
 {
        temp_count = delay_count +1;
        asm volatile("outer: dec _temp_count");
                asm volatile("cp0 _temp_count");
        asm volatile("bra z, done");
        asm volatile("do #3200, inner" );
                asm volatile("nop");
        asm volatile("inner: nop");
        asm volatile("bra outer");
        asm volatile("done:");
}
// je passe ? #100 au lieu de 1500 donc le delai est
// temp_count*[(100*2))+4] en cycles soit temp_count * 204
// Je corrige dans delay.h
// j'ai besoin de pas plus de 1 us pour delai minimum
// Le decompte ci dessous donne
// dec 1 cycle
// bra nz 2 cycle
// nop : 1 cycle
// donc dur?e = (valdelay * 3 + 2 cycles)* 25 ns
// si je veux 1 us je fais valdelay = 13
// limite=450 ns soit valdelay=6
void Delay_Us( unsigned int valdelay )
{
        temp_count = valdelay;
        asm volatile("outer1: dec _temp_count");
        asm volatile("bra nz, outer1");
        asm volatile("nop"); }
```

3.1.2 Main

```
// via config
// Window target memory view cingiguration bits
// DSPIC33FJ256GP710A Configuration Bit Settings
// DSPIC33FJ256GP710A Configuration Bit Settings
// 'C' source line config statements
// FBS #pragma config BWRP = WRPROTECT_OFF
// Boot Segment Write Protect (Boot Segment may be written)
```

```
#pragma config BSS = NO_FLASH
// Boot Segment Program Flash Code Protection (No Boot program Flash segment)
#pragma config RBS = NO_RAM
// Boot Segment RAM Protection (No Boot RAM)
// FSS
#pragma config SWRP = WRPROTECT_OFF
// Secure Segment Program Write Protect (Secure Segment may be written)
#pragma config SSS = NO_FLASH
// Secure Segment Program Flash Code Protection (No Secure Segment)
#pragma config RSS = NO_RAM
// Secure Segment Data RAM Protection (No Secure RAM)
// FGS
#pragma config GWRP = OFF
// General Code Segment Write Protect (User program memory is not write-protected)
#pragma config GSS = OFF
// General Segment Code Protection (User program memory is not code-protected)
// FOSCSEL
#pragma config FNOSC = FRC
// Oscillator Mode (Internal Fast RC (FRC)) at POW
#pragma config IESO = ON
// Two-speed Oscillator Start-Up Enable (Start up with FRC, then switch)
// FOSC
#pragma config POSCMD = XT
// Primary Oscillator Source (XT Oscillator Mode)
#pragma config OSCIOFNC = OFF
// OSC2 Pin Function (OSC2 pin has clock out function)
#pragma config FCKSM = CSECMD
// Clock Switching and Monitor (Clock switching is enabled, Fail-Safe Clock Monitor is
↪  disabled)
// FWDT
#pragma config WDTPOST = PS32768
// Watchdog Timer Postscaler (1:32,768)
#pragma config WDTPRE = PR128
// WDT Prescaler (1:128)
#pragma config PLLKEN = ON
// PLL Lock Enable bit (Clock switch to PLL source will wait until the PLL lock signal is
↪  valid.)
#pragma config WINDIS = OFF
// Watchdog Timer Window (Watchdog Timer in Non-Window mode)
#pragma config FWDTEN = OFF
// Watchdog Timer Enable (Watchdog timer enabled/disabled by user software)
// FPOR
#pragma config FPWRT = PWR128
// POR Timer Value (128ms)
// FICD
#pragma config ICS = PGD1
// Comm Channel Select (Communicate on PGC1/EMUC1 and PGD1/EMUD1)
#pragma config JTAGEN = OFF
// JTAG Port Enable (JTAG is Disabled)
// #pragma config statements should precede project file includes.
// Use project enums instead of #define for ON and OFF.
#include <xc.h>
void initIO(void)
{
/* set LEDs (D3-D10/RA0-RA7) drive state low */
        LATA = 0xFF00;
/* set LED pins (D3-D10/RA0-RA7) as outputs */
        TRISA = 0xFF00;
```

```
        AD1PCFGL=0xFFFF;
               // Configure all ANx pins as Digital  }
//------------------------------------------------------
int main(void)
{
// on programme pour 40 Mhz sur le mod?le de Thermal all coefficients
// Configure Oscillator to operate the device at 40Mhz
// Fosc = Fin*M/(N1*N2), Fcy = Fosc/2
// Explorer 16 Fxt= 8 Mhz
// Fosc = 8*(38+2)/(2*2) = 80 MHz, Fcy = 40 MIPS Tcy=(1/40)us = 25 ns = 0,025 us
↪          PLLFBD=38;
// M=40 40 Mhz
//PLLFBD = 18;
//CLKDIVbits.PLLPOST=0;
// N1=2
        CLKDIVbits.PLLPRE=0;
// N2=2
// Disable Watch Dog Timer
        RCONbits.SWDTEN=0;
// Clock switch to incorporate PLL
// ATTENTION AUX MESSAGES AVEC LE DEBUGGER VOIR STEP AU DEBUT OU PAS
__builtin_write_OSCCONH(0x03);
// Initiate Clock Switch to XT  with PLL
        __builtin_write_OSCCONL(0x01);
// Start clock switching
while (OSCCONbits.COSC != 0b011);
        //ASO Wait for Clock switch to occur pour PLL et XT
// Wait for PLL to lock
while(OSCCONbits.LOCK!=1);
/* set LEDs (D3-D10/RA0-RA7) drive state low */
LATA = 0xFF00;
/* set LED pins (D3-D10/RA0-RA7) as outputs */
TRISA = 0xFF00;
        AD1PCFGL=0xFFFF;
// Configure all ANx pins as Digital
LATA = 0x01;
// leds 4 bas ? 1
while(1)
// loop forever waiting for ADC interrupts
{
LATA = LATA << 0x01;
int j = 2;
while(j--)
{
int i = 65000;
while(i--);
};
};
}
```

3.1.3 MainetADC

```
// via config
// Window target memory view cingiguration bits
// DSPIC33FJ256GP710A Configuration Bit Settings
// DSPIC33FJ256GP710A Configuration Bit Settings
```

```
// 'C' source line config statements
// FBS #pragma config BWRP = WRPROTECT_OFF
// Boot Segment Write Protect (Boot Segment may be written)
#pragma config BSS = NO_FLASH
// Boot Segment Program Flash Code Protection (No Boot program Flash segment)
#pragma config RBS = NO_RAM
// Boot Segment RAM Protection (No Boot RAM)
// FSS
#pragma config SWRP = WRPROTECT_OFF
// Secure Segment Program Write Protect (Secure Segment may be written)
#pragma config SSS = NO_FLASH
// Secure Segment Program Flash Code Protection (No Secure Segment)
#pragma config RSS = NO_RAM
// Secure Segment Data RAM Protection (No Secure RAM)
// FGS
#pragma config GWRP = OFF
// General Code Segment Write Protect (User program memory is not write-protected)
#pragma config GSS = OFF
// General Segment Code Protection (User program memory is not code-protected)
// FOSCSEL
#pragma config FNOSC = FRC
// Oscillator Mode (Internal Fast RC (FRC)) at POW
#pragma config IESO = ON
// Two-speed Oscillator Start-Up Enable (Start up with FRC, then switch)
// FOSC
#pragma config POSCMD = XT
// Primary Oscillator Source (XT Oscillator Mode)
#pragma config OSCIOFNC = OFF
// OSC2 Pin Function (OSC2 pin has clock out function)
#pragma config FCKSM = CSECMD
// Clock Switching and Monitor (Clock switching is enabled, Fail-Safe Clock Monitor is
↪  disabled)
// FWDT
#pragma config WDTPOST = PS32768
// Watchdog Timer Postscaler (1:32,768)
#pragma config WDTPRE = PR128
// WDT Prescaler (1:128)
#pragma config PLLKEN = ON
// PLL Lock Enable bit (Clock switch to PLL source will wait until the PLL lock signal is
↪  valid.)
#pragma config WINDIS = OFF
// Watchdog Timer Window (Watchdog Timer in Non-Window mode)
#pragma config FWDTEN = OFF
// Watchdog Timer Enable (Watchdog timer enabled/disabled by user software)
// FPOR
#pragma config FPWRT = PWR128
// POR Timer Value (128ms)
// FICD
#pragma config ICS = PGD1
// Comm Channel Select (Communicate on PGC1/EMUC1 and PGD1/EMUD1)
#pragma config JTAGEN = OFF
// JTAG Port Enable (JTAG is Disabled)
// #pragma config statements should precede project file includes.
// Use project enums instead of #define for ON and OFF.
#include <xc.h>
void initIO(void)
{
/* set LEDs (D3-D10/RA0-RA7) drive state low */
```

```
        LATA = 0xFF00;
/* set LED pins (D3-D10/RA0-RA7) as outputs */
        TRISA = 0xFF00;
        AD1PCFGL=0xFFFF;
               // Configure all ANx pins as Digital  }
//-----------------------------------------------------
int main(void)
{
// On initialise les paramètres du filtre
    C[0] = 0.071563720703125;
    C[1] = 0.09909057617875;
    C[2] = 0.12176513671875;
    C[3] = 0.13665771484375;
    C[4] = 0.141845703125;
    C[5] = 0.13665771484375;
    C[6] = 0.12176513671875;
    C[7] = 0.09909057617875;
    C[8] = 0.071563720703125;
// on programme pour 40 Mhz sur le mod?le de Thermal all coefficients
// Configure Oscillator to operate the device at 40Mhz
// Fosc = Fin*M/(N1*N2), Fcy = Fosc/2
// Explorer 16 Fxt= 8 Mhz
// Fosc = 8*(38+2)/(2*2) = 80 MHz, Fcy = 40 MIPS Tcy=(1/40)us = 25 ns = 0,025 us
↪          PLLFBD=38;
// M=40 40 Mhz
//PLLFBD = 18;
//CLKDIVbits.PLLPOST=0;
// N1=2
        CLKDIVbits.PLLPRE=0;
// N2=2
// Disable Watch Dog Timer
        RCONbits.SWDTEN=0;
// Clock switch to incorporate PLL
// ATTENTION AUX MESSAGES AVEC LE DEBUGGER VOIR STEP AU DEBUT OU PAS
__builtin_write_OSCCONH(0x03);
// Initiate Clock Switch to XT  with PLL
        __builtin_write_OSCCONL(0x01);
// Start clock switching
while (OSCCONbits.COSC != 0b011);
        //ASO Wait for Clock switch to occur pour PLL et XT
// Wait for PLL to lock
while(OSCCONbits.LOCK!=1);
/* set LEDs (D3-D10/RA0-RA7) drive state low */
LATA = 0xFF00;
/* set LED pins (D3-D10/RA0-RA7) as outputs */
TRISA = 0xFF00;
        LATA = 0x0F; // leds 4 bas à 1
pasquantif = 3.3/4096;
adcConfigureManual();
//initAdc1();
i = 0;
while(1)
{
buf[i] = analogRead(1);
// TC1047
y[i] = 0;
for (k = 0; k<9; k++)
{
```

```
   if (i-k < 0)
{
                y[i] += C[k] * buf[i-k+160];
}
else
{
               y[i] += C[k] * buf[i-k];
}
      SPI2BUF = y[i];
        }
i++;
if (i == 160)
{
i = 0;
}
}
}
```

3.2 Filtrage

Conception du filtre : FIR

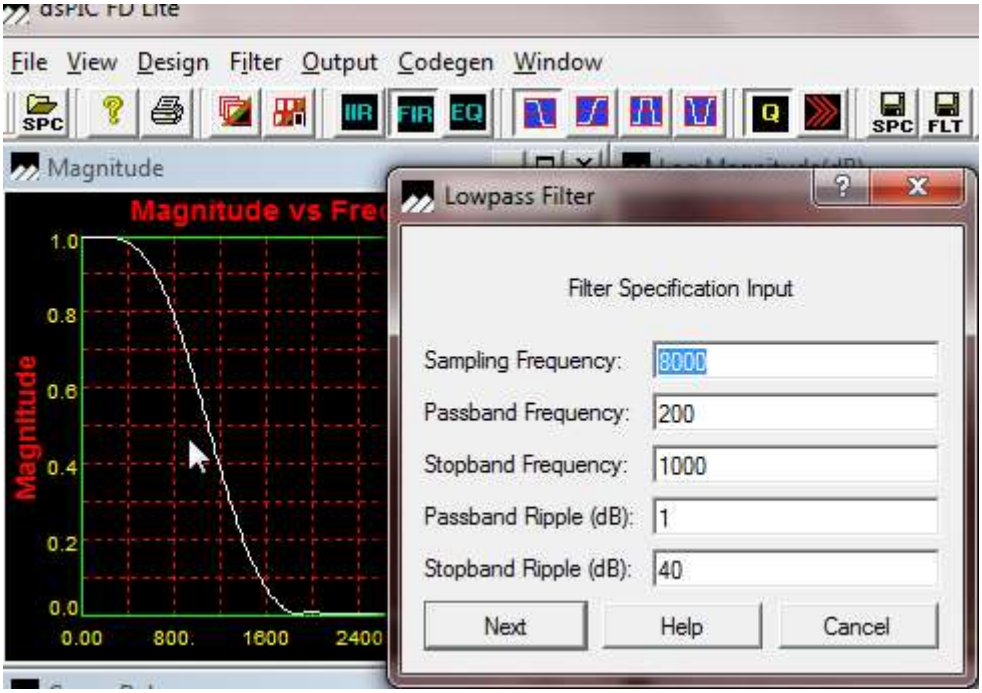
Théorie : voir chapitre 19 (fichier joint)

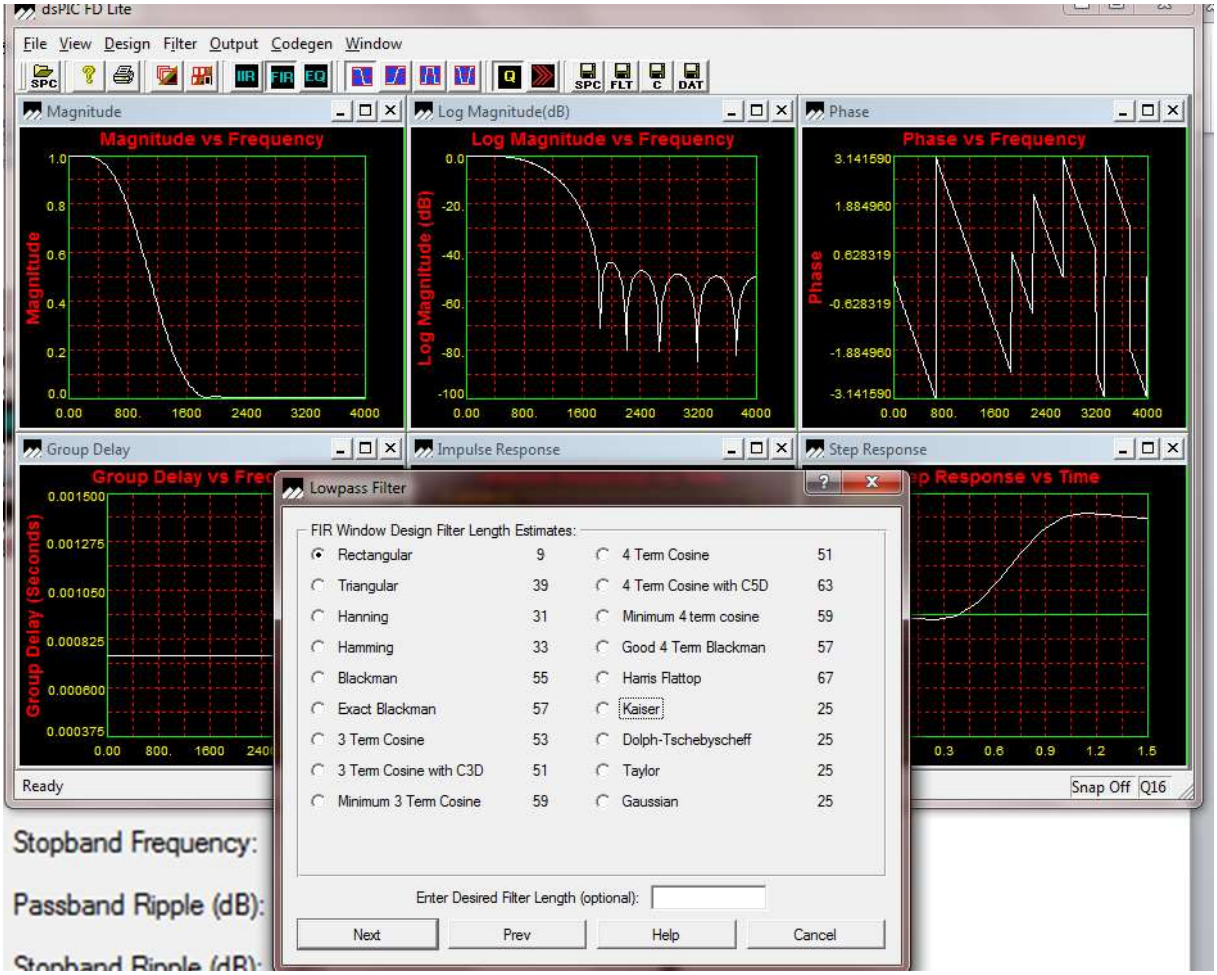
La fonction est du type :

The most obvious source of information is the *input signal*, that is, the values: $x[n]$, $x[n-1]$, $x[n-2]$, ···. The professor could have been multiplying each point in the input signal by a coefficient, and adding the products together:

$$y[n] = a_0 x[n] + a_1 x[n-1] + a_2 x[n-2] + a_3 x[n-3] + \cdots$$

Un programme tel que dsPIC FD ou dsPIC FD Lite avec les paramètres suivants permet d'avoir les paramètres des coefficients du filtre:





Le fichier .flt (ici FiltrePB8k200Hz.flt) donne 9 coefficients

FiltrePB8k200Hz.flt - Bloc-notes

Fichier Edition Format Affichage ?

```
FILTER COEFFICIENT FILE
FIR DESIGN
SAMPLING FREQUENCY              0.800000E+04 HERTZ
   9                    /* number of taps in decimal     */
   9                    /* number of taps in hexadecimal */
16                      /* number of bits in quantized coefficients (dec) */
  10                    /* number of bits in quantized coefficients (hex) */
 0                      /* shift count in decimal  */
   0  0.100000000E+01 /* shift count (hex), gain multiplier */
       2345       929  /* coefficient of tap     0 */
       3247       CAF  /* coefficient of tap     1 */
       3990       F96  /* coefficient of tap     2 */
       4478      117E  /* coefficient of tap     3 */
       4648      1228  /* coefficient of tap     4 */
       4478      117E  /* coefficient of tap     5 */
       3990       F96  /* coefficient of tap     6 */
       3247       CAF  /* coefficient of tap     7 */
       2345       929  /* coefficient of tap     8 */
  0.71563720703125OOE-01 3FB2520000000000/* coefficient of tap     0 */
  0.99090576171875OOE-01 3FB95E0000000000/* coefficient of tap     1 */
  0.12176513671875OOE+00 3FBF2C0000000000/* coefficient of tap     2 */
  0.13665771484375OOE+00 3FC17E0000000000/* coefficient of tap     3 */
  0.14184570312500OE+00 3FC2280000000000/* coefficient of tap     4 */
  0.13665771484375OOE+00 3FC17E0000000000/* coefficient of tap     5 */
  0.12176513671875OOE+00 3FBF2C0000000000/* coefficient of tap     6 */
  0.99090576171875OOE-01 3FB95E0000000000/* coefficient of tap     7 */
  0.71563720703125OOE-01 3FB2520000000000/* coefficient of tap     8 */
```

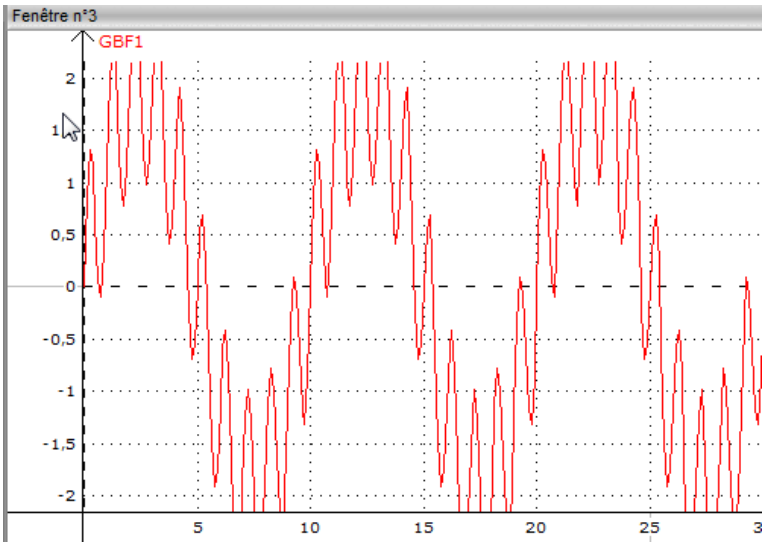
Ce type de filtre (et d'équation) est programmable dans Latis pro.

Il faut avoir préalablement fait une acquisition avec un signal de 100hz plus un 1kHz qui correspondent à des acquisitions ou on recrée un signal avec une équation une fois une plage d'acquisition effectuée.

Pour l'équation on peut écrire par exemple :

```
GBF1=2*sin(200*PI*temps)+1*sin(2000*Pi*Temps)
```

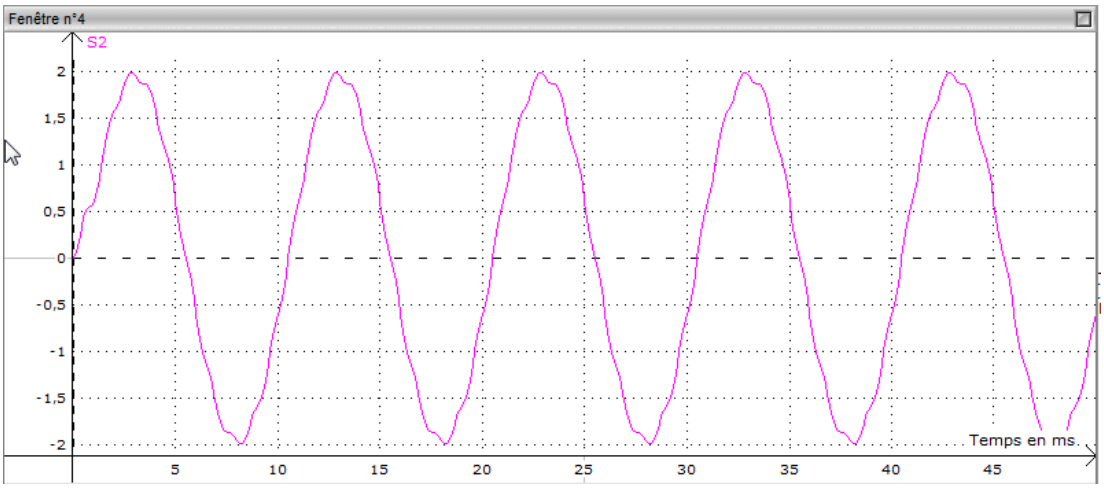
Soit le signal (sinus à 100hz et « bruit » à 1khz):

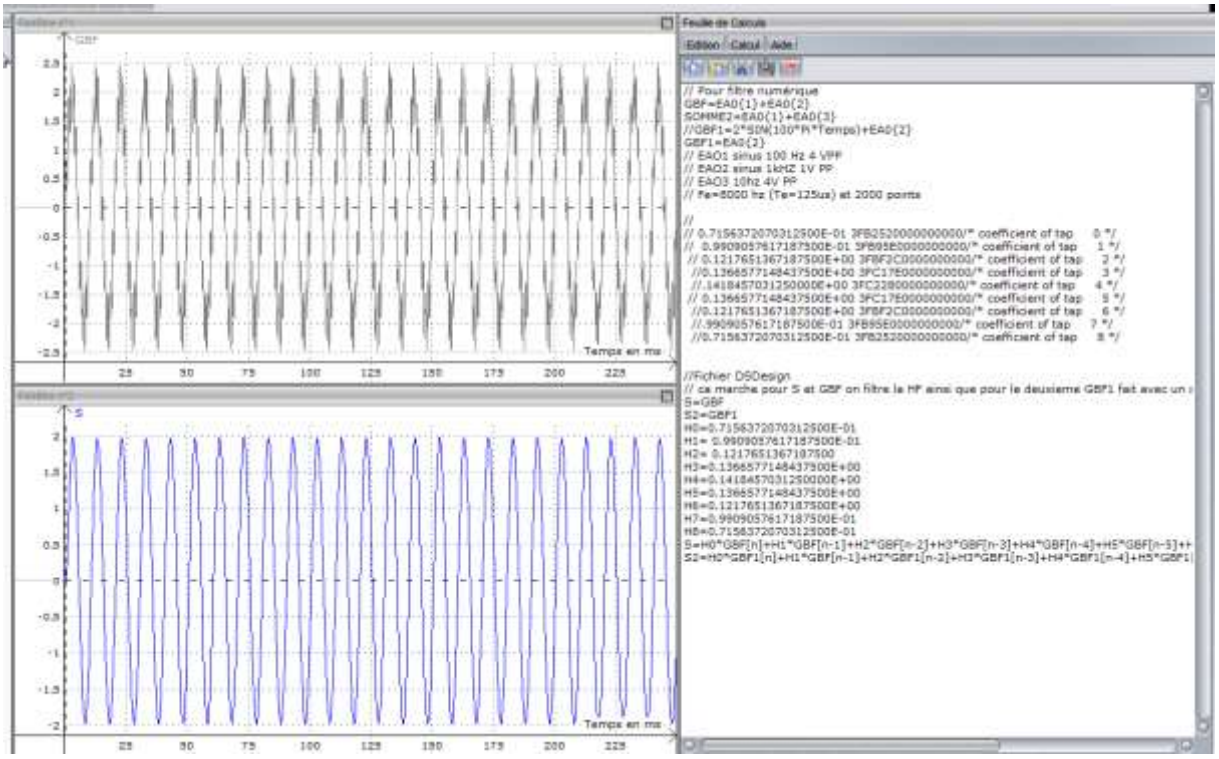


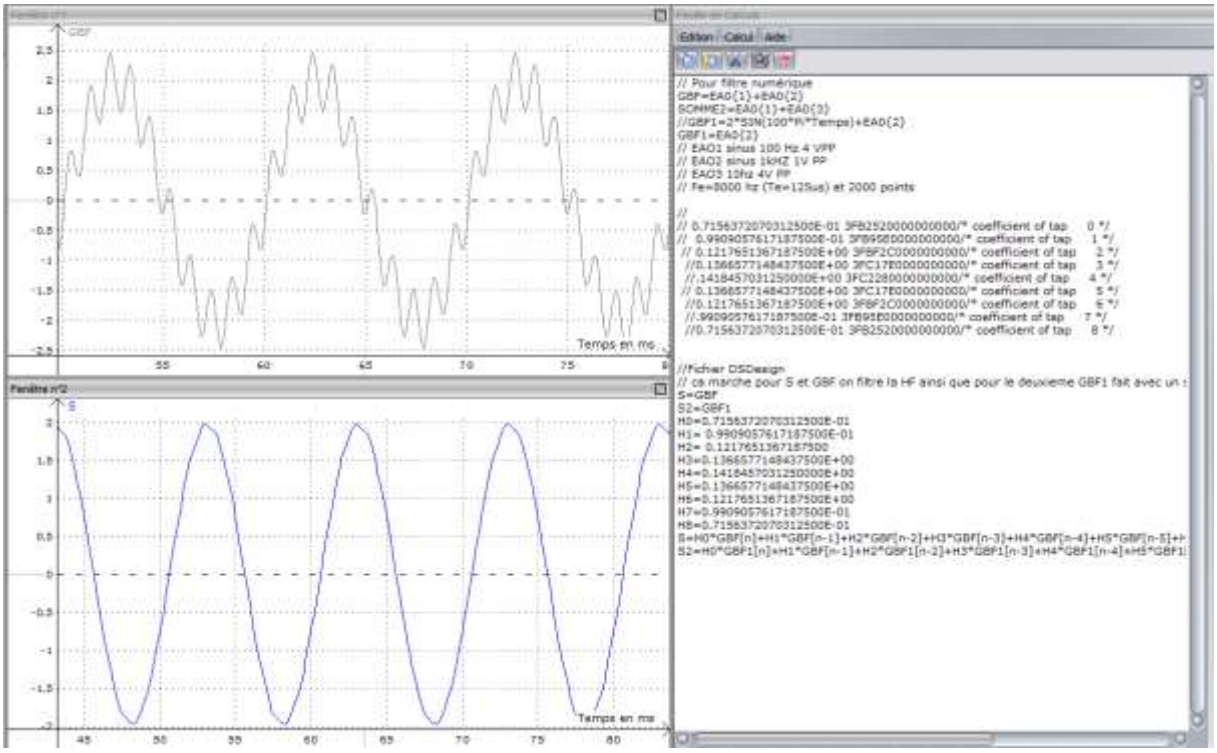
La fonction de filtrage est faite avec les coefficients définis en constante (Traitement Calculs) le signal de sortie du filtre étant ici S2

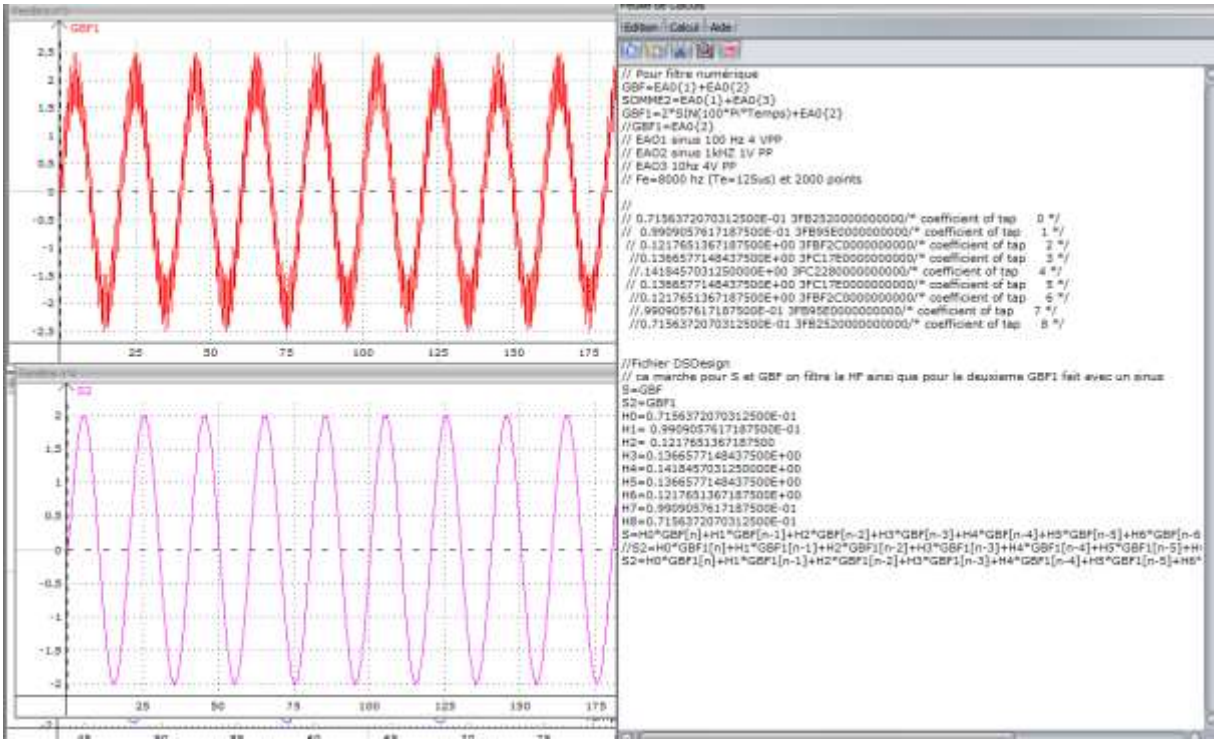
```
S2=GBF1
H0=0.7156372070312500E-01
H1= 0.9909057617187500E-01
H2= 0.1217651367187500
H3=0.1366577148437500E+00
H4=0.1418457031250000E+00
H5=0.1366577148437500E+00
H6=0.1217651367187500E+00
H7=0.9909057617187500E-01
H8=0.7156372070312500E-01
S2=H0*GBF1[n]+H1*GBF1[n-1]+H2*GBF1[n-2]+H3*GBF1[n-3]+H4*GBF1[n-4]+H5*GBF1[n-5]+H6*GBF1[n-6]+H7*GBF1[n-7]+H8*GBF1[n-8]
```

Le signal filtré indique bien la suppression du signal à 1kHz

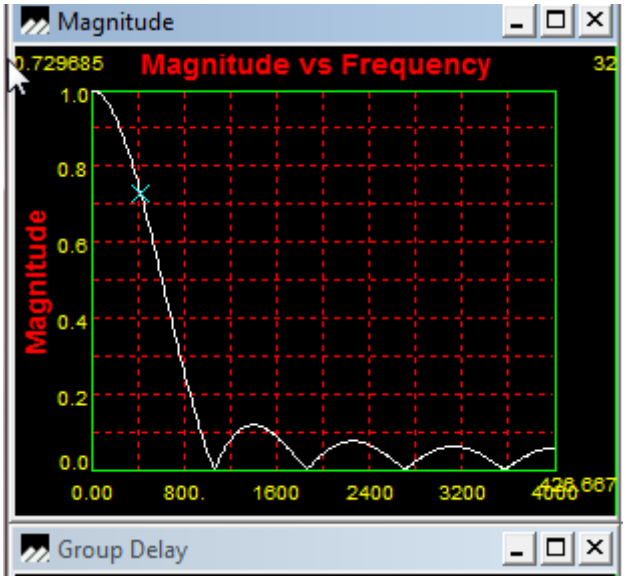


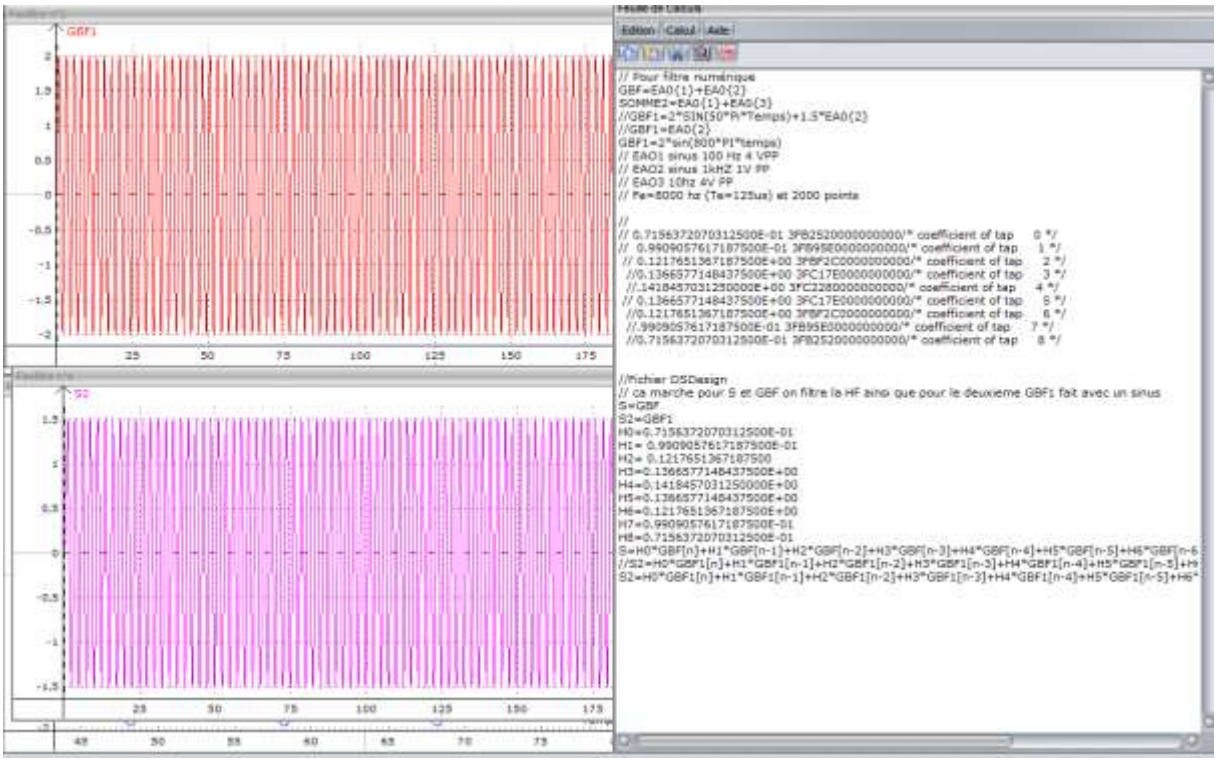




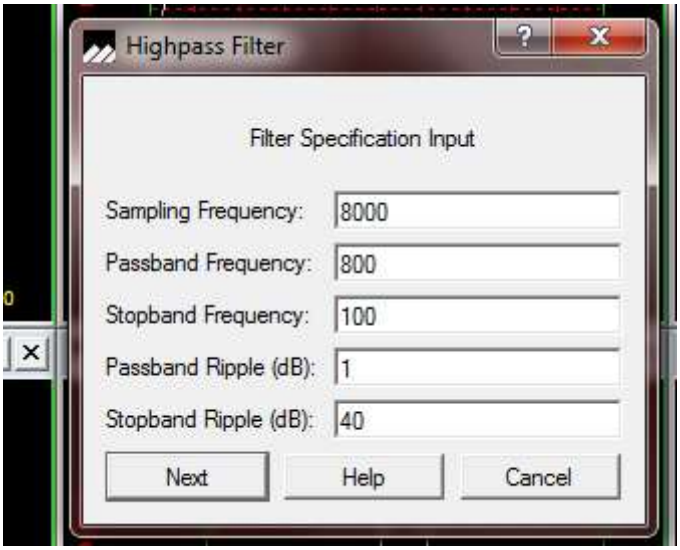


On vérifie de plus a 400hz l'atténuation de 0,73 (0,73*2 donne environ 1,46)

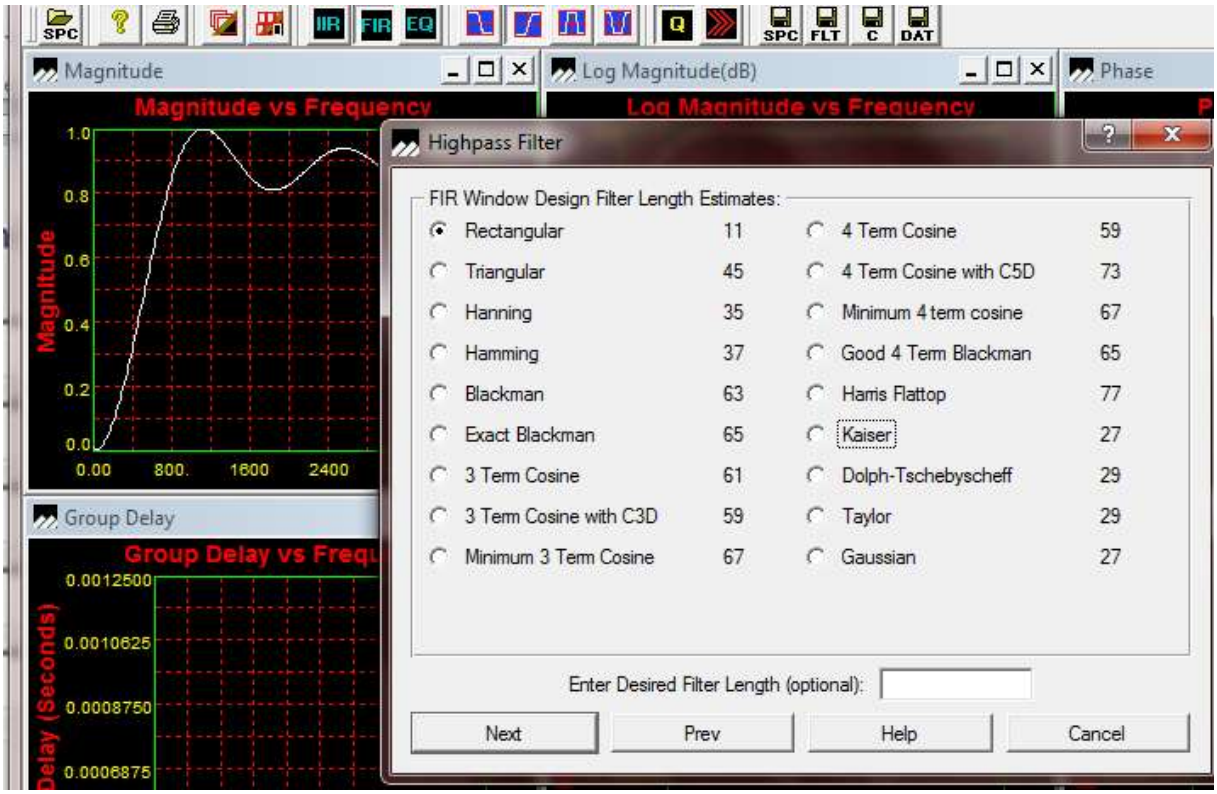


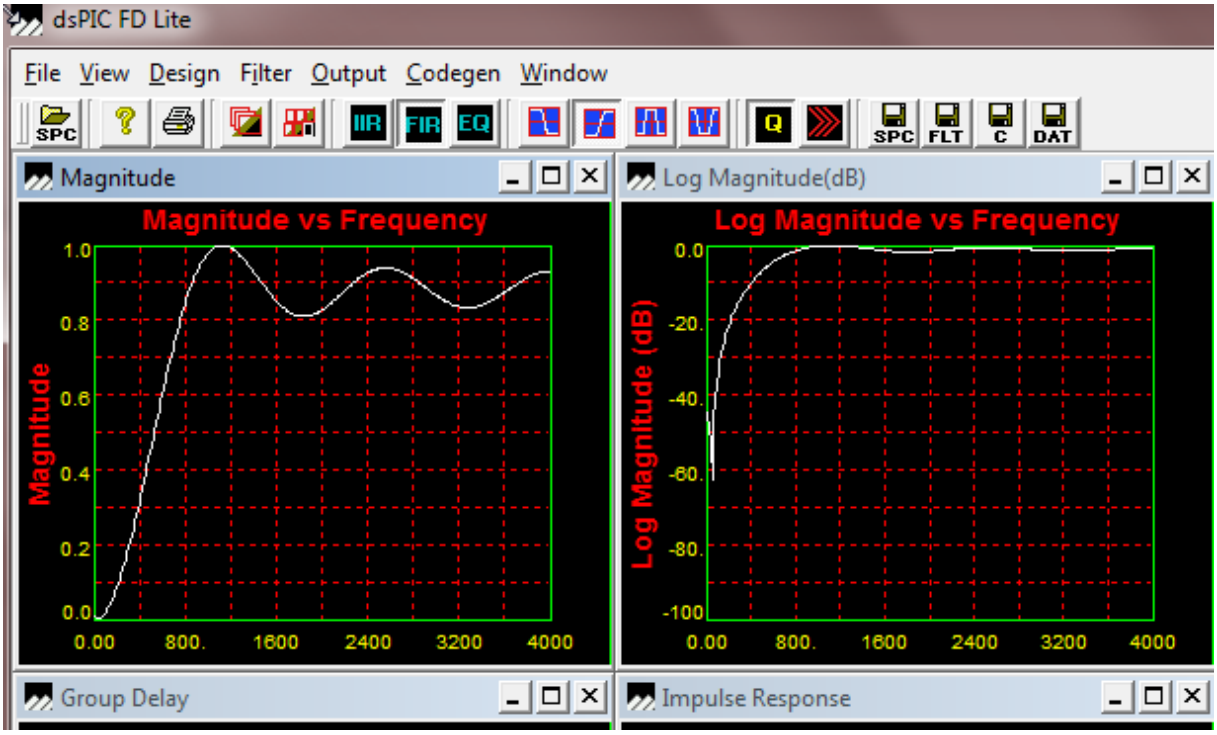


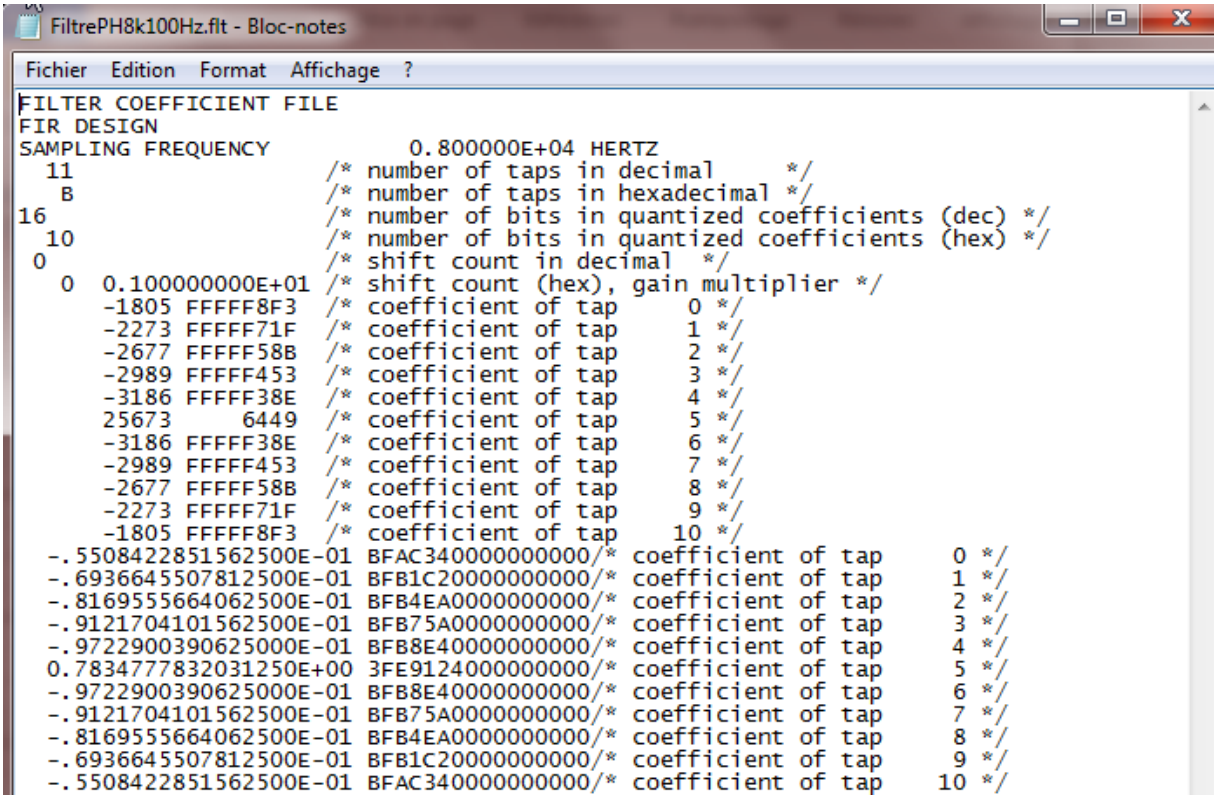
On peut utiliser la même méthode pour un filtre passe haut :



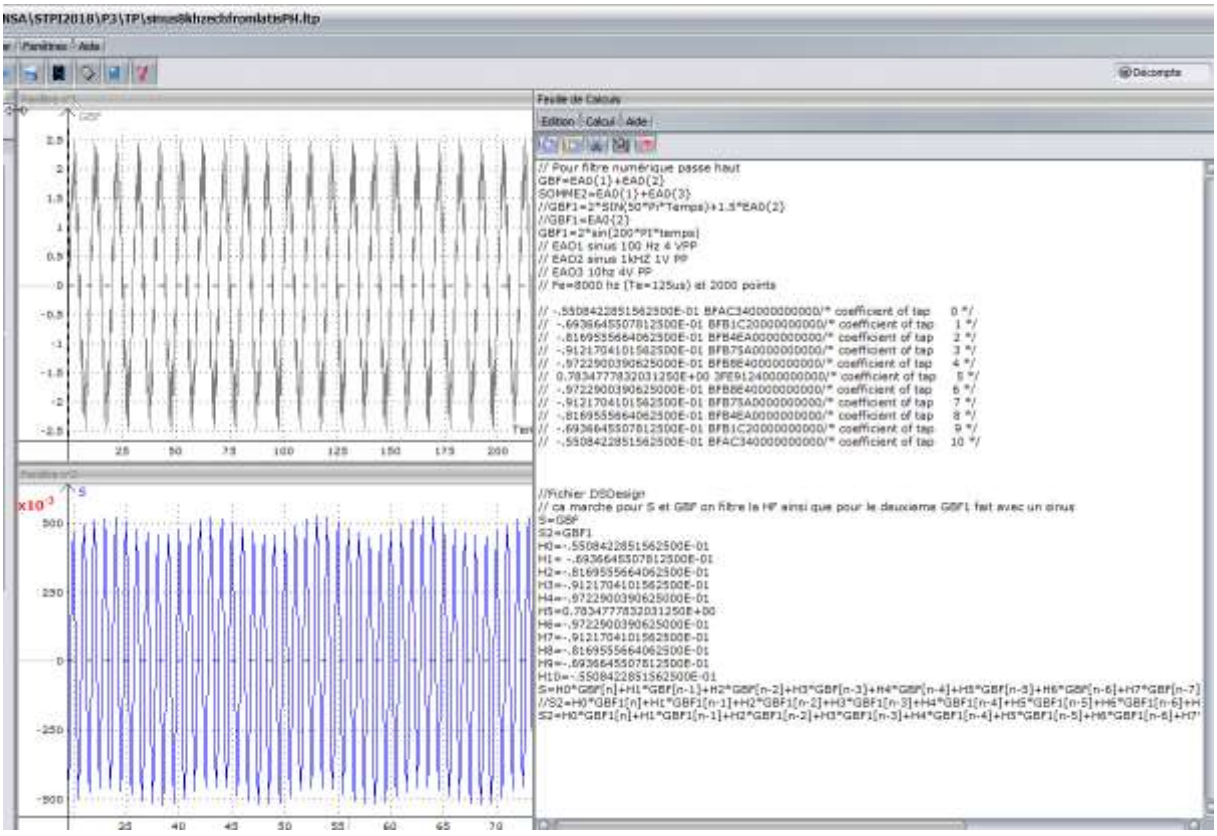
Cela donne 11 coefficients



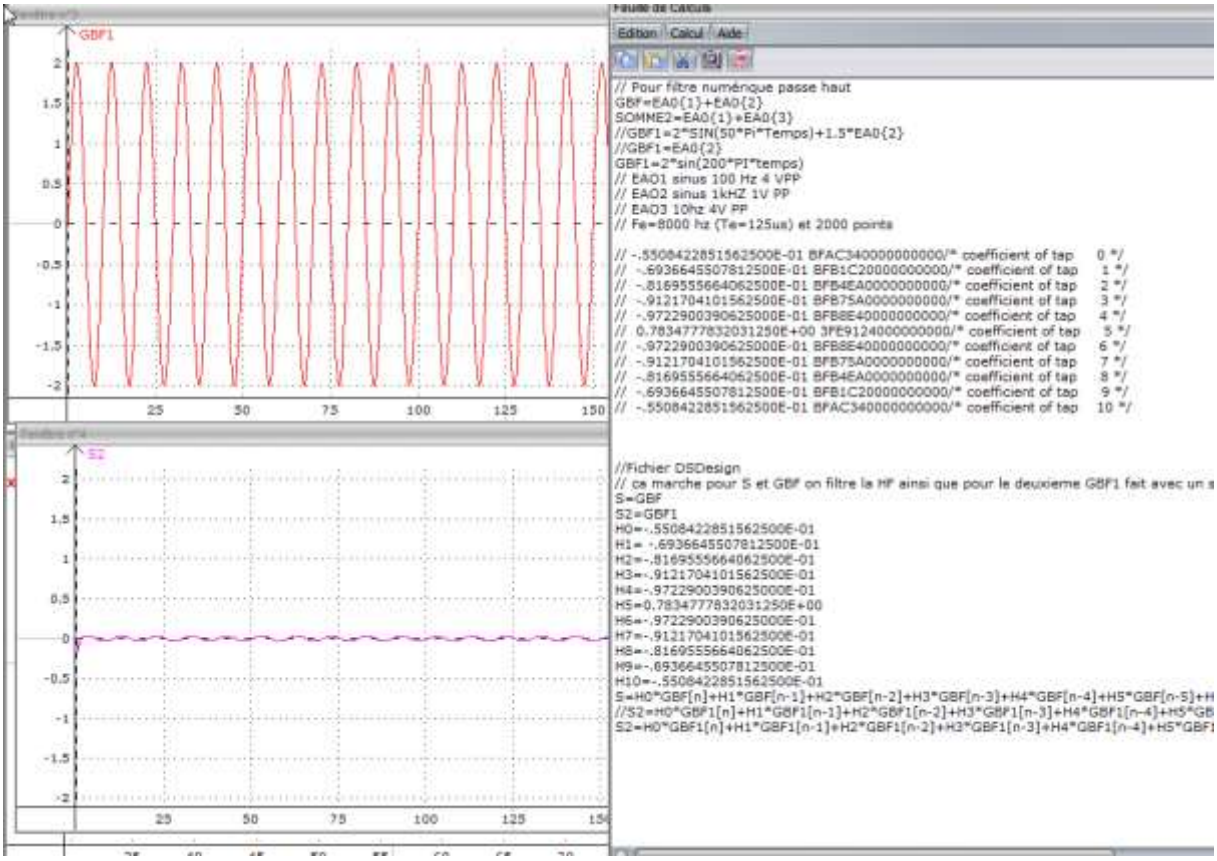




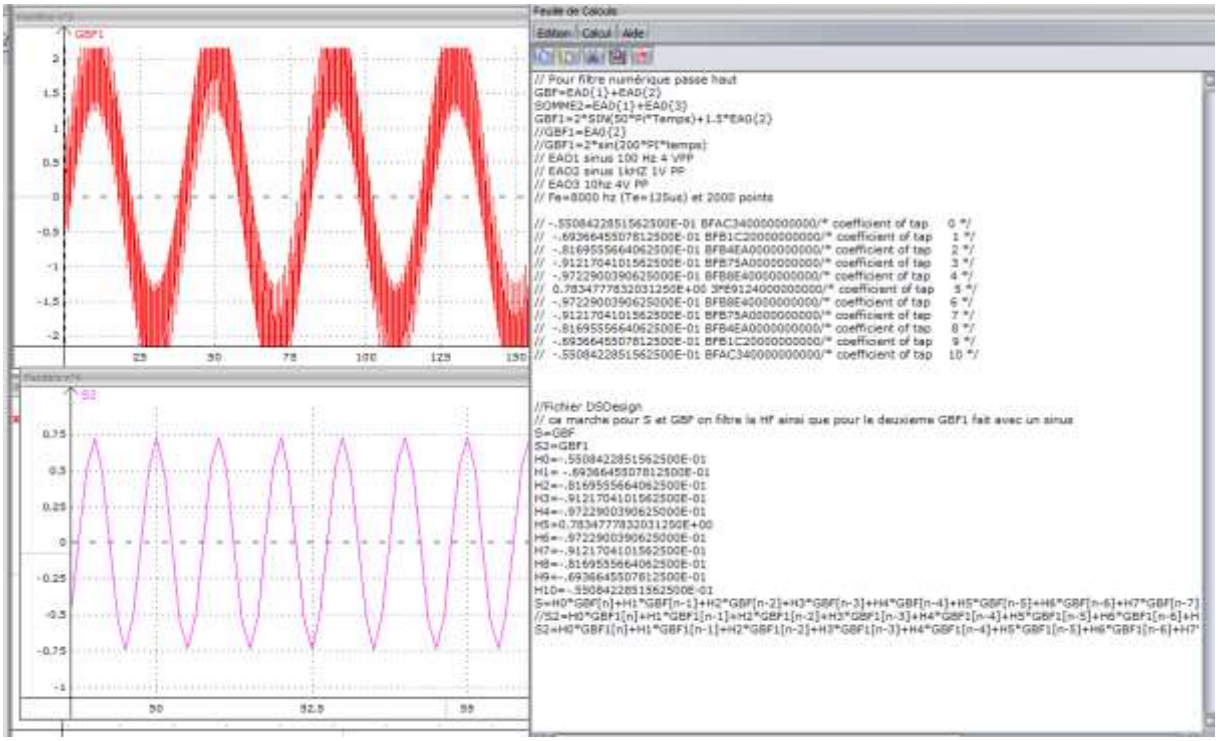
On programme les coefficients dans Latis pro, on vérifie cette fois-ci le filtrage des BF



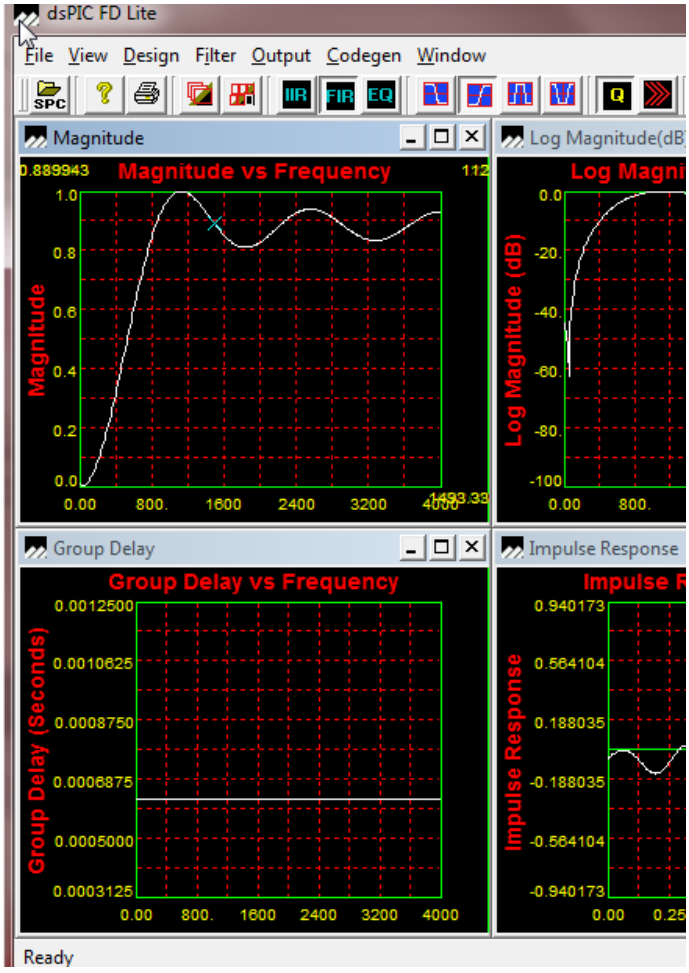
Avec une petite ondulation résiduelle

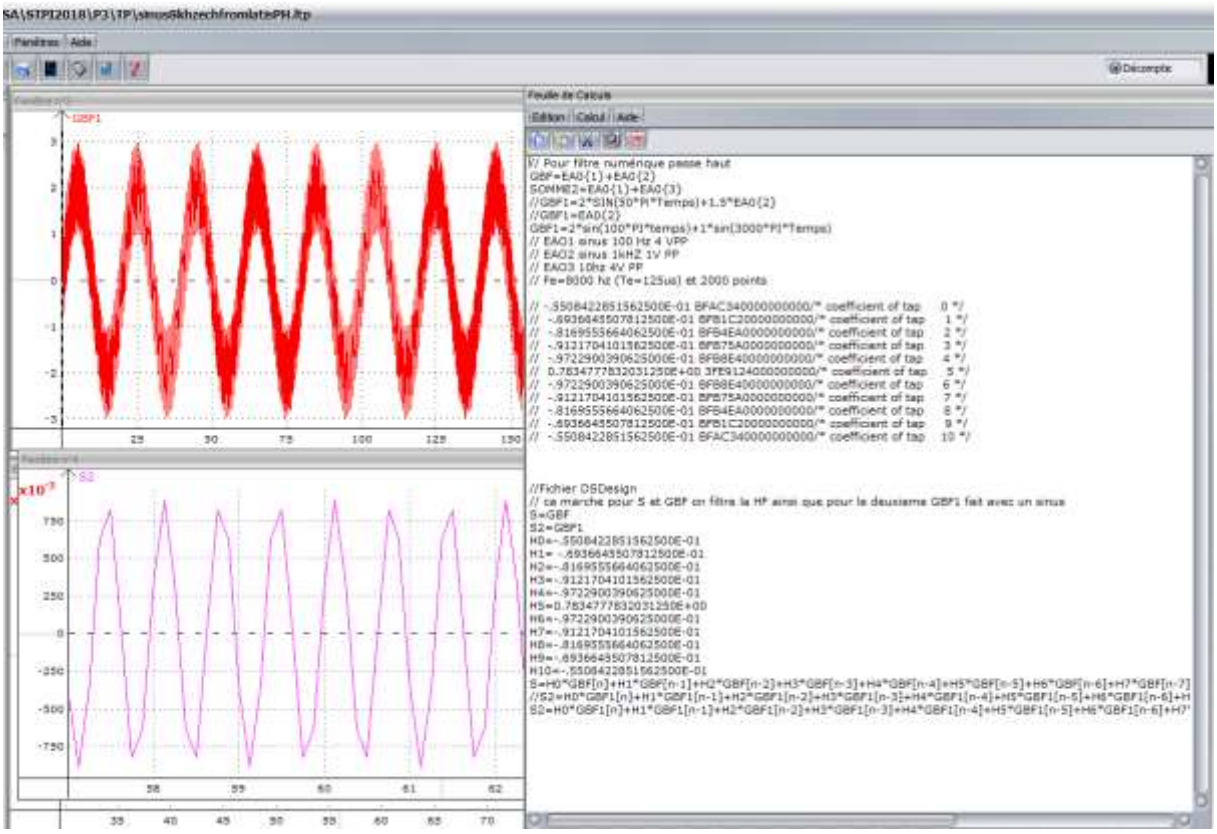


Idem pour 50hz plus sinus 1khz 1,5VPP

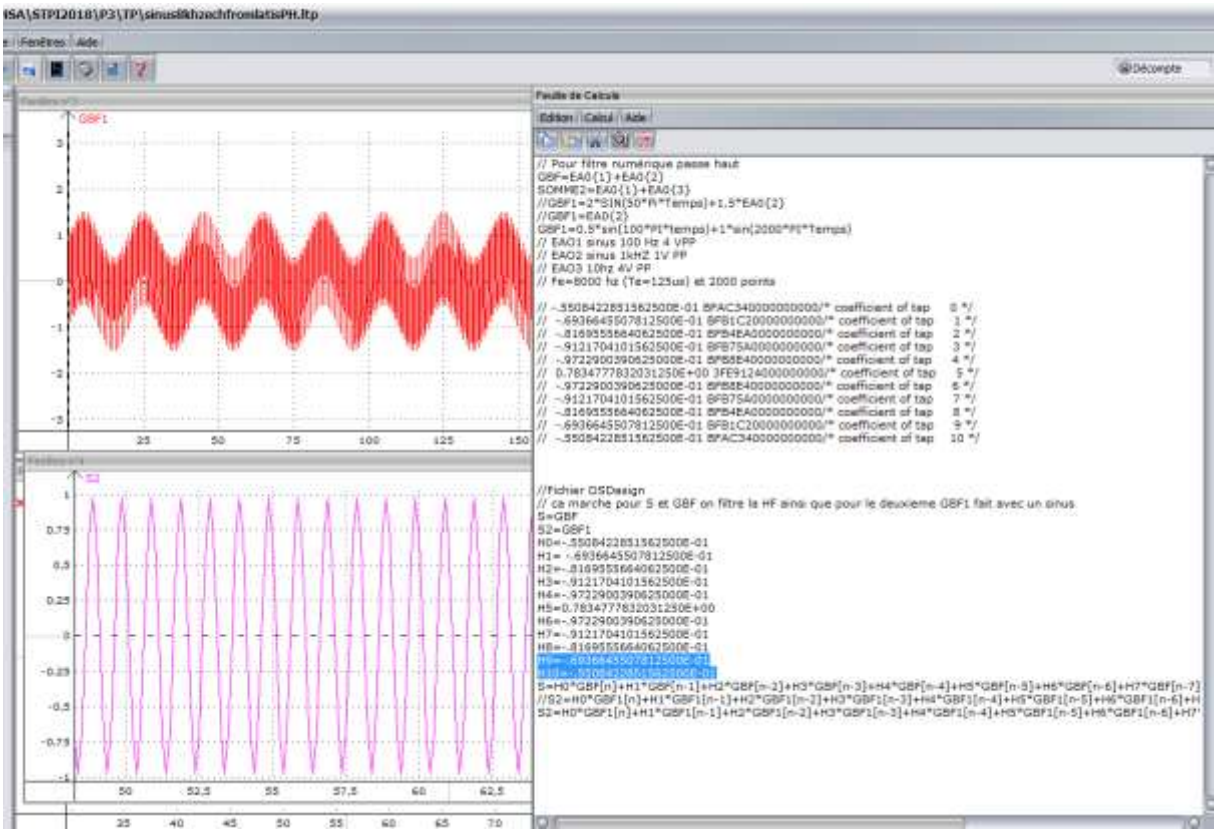


Idem avec un 50hz et 1,5khz 2VPP (on a une atténuation)





Efficace pour le filtrage à 50HZ



CHAPTER

19

Recursive Filters

Recursive filters are an efficient way of achieving a long impulse response, without having to perform a long convolution. They execute very rapidly, but have less performance and flexibility than other digital filters. Recursive filters are also called *Infinite Impulse Response* (IIR) filters, since their impulse responses are composed of decaying exponentials. This distinguishes them from digital filters carried out by convolution, called *Finite Impulse Response* (FIR) filters. This chapter is an introduction to how recursive filters operate, and how simple members of the family can be designed. Chapters 20, 26 and 33 present more sophisticated design methods.

The Recursive Method

To start the discussion of recursive filters, imagine that you need to extract information from some signal, $x[\ ]$. Your need is so great that you hire an old mathematics professor to process the data for you. The professor's task is to filter $x[\ ]$ to produce $y[\ ]$, which hopefully contains the information you are interested in. The professor begins his work of calculating each point in $y[\ ]$ according to some algorithm that is locked tightly in his over-developed brain. Part way through the task, a most unfortunate event occurs. The professor begins to babble about analytic singularities and fractional transforms, and other demons from a mathematician's nightmare. It is clear that the professor has lost his mind. You watch with anxiety as the professor, and your algorithm, are taken away by several men in white coats.

You frantically review the professor's notes to find the algorithm he was using. You find that he had completed the calculation of points $y[0]$ through $y[27]$, and was about to start on point $y[28]$. As shown in Fig. 19-1, we will let the variable, n, represent the point that is currently being calculated. This means that $y[n]$ is sample 28 in the output signal, $y[n-1]$ is sample 27, $y[n-2]$ is sample 26, etc. Likewise, $x[n]$ is point 28 in the input signal,

$x[n-1]$ is point 27, etc. To understand the algorithm being used, we ask ourselves: "What information was available to the professor to calculate $y[n]$, the sample currently being worked on?"

The most obvious source of information is the *input signal*, that is, the values: $x[n]$, $x[n-1]$, $x[n-2]$,⋯. The professor could have been multiplying each point in the input signal by a coefficient, and adding the products together:

$$y[n] = a_0 x[n] + a_1 x[n-1] + a_2 x[n-2] + a_3 x[n-3] + \cdots$$

You should recognize that this is nothing more than simple convolution, with the coefficients: a_0, a_1, a_2,⋯, forming the convolution kernel. If this was all the professor was doing, there wouldn't be much need for this story, or this chapter. However, there is another source of information that the professor had access to: the *previously* calculated values of the output signal, held in: $y[n-1]$, $y[n-2]$, $y[n-3]$,⋯. Using this additional information, the algorithm would be in the form:

$$\begin{aligned} y[n] = {} & a_0 x[n] + a_1 x[n-1] + a_2 x[n-2] + a_3 x[n-3] + \cdots \\ & + b_1 y[n-1] + b_2 y[n-2] + b_3 y[n-3] + \cdots \end{aligned}$$

EQUATION 19-1
The recursion equation. In this equation, $x[\ ]$ is the input signal, $y[\ ]$ is the output signal, and the a's and b's are coefficients.

In words, each point in the output signal is found by multiplying the values from the input signal by the "a" coefficients, multiplying the previously calculated values from the output signal by the "b" coefficients, and adding the products together. Notice that there isn't a value for b_0, because this corresponds to the sample being calculated. Equation 19-1 is called the **recursion equation**, and filters that use it are called **recursive filters**. The "a" and "b" values that define the filter are called the **recursion coefficients**. In actual practice, no more than about a dozen recursion coefficients can be used or the filter becomes unstable (i.e., the output continually increases or oscillates). Table 19-1 shows an example recursive filter program.

Recursive filters are useful because they *bypass* a longer convolution. For instance, consider what happens when a delta function is passed through a recursive filter. The output is the filter's *impulse response*, and will typically be a sinusoidal oscillation that exponentially decays. Since this impulse response in infinitely long, recursive filters are often called *infinite impulse response* (IIR) filters. In effect, recursive filters *convolve* the input signal with a very long filter kernel, although only a few coefficients are involved.

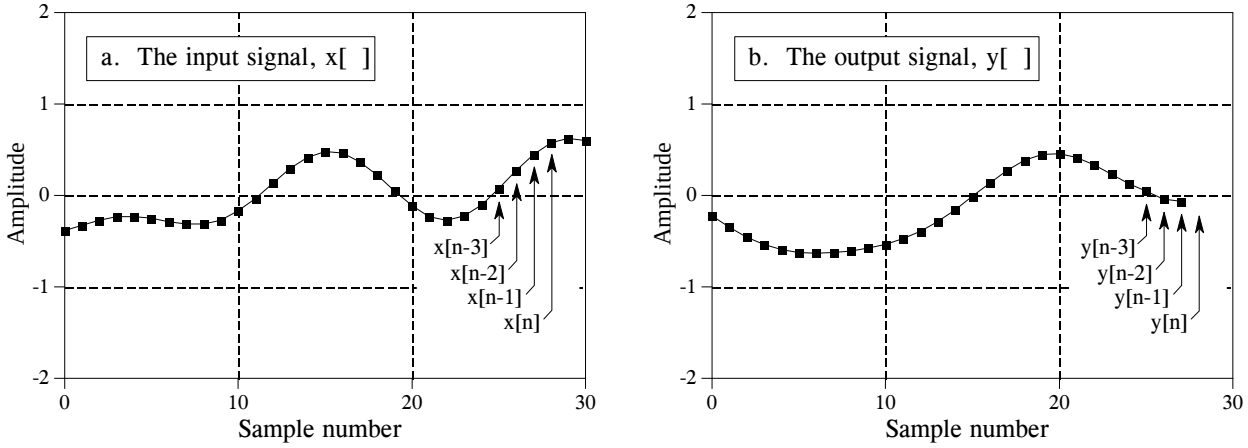


FIGURE 19-1
Recursive filter notation. The output sample being calculated, $y[n]$, is determined by the values from the input signal, $x[n]$, $x[n-1]$, $x[n-2]$, ···, as well as the *previously* calculated values in the output signal, $y[n-1]$, $y[n-2]$, $y[n-3]$, ···. These figures are shown for $n = 28$.

The relationship between the recursion coefficients and the filter's response is given by a mathematical technique called the **z-transform**, the topic of Chapter 33. For example, the z-transform can be used for such tasks as: converting between the recursion coefficients and the frequency response, combining cascaded and parallel stages into a single filter, designing recursive systems that mimic analog filters, etc. Unfortunately, the z-transform is very mathematical, and more complicated than most DSP *users* are willing to deal with. This is the realm of those that specialize in DSP.

There are three ways to find the recursion coefficients without having to understand the z-transform. First, this chapter provides design equations for several types of simple recursive filters. Second, Chapter 20 provides a "cookbook" computer program for designing the more sophisticated *Chebyshev* low-pass and high-pass filters. Third, Chapter 26 describes an iterative method for designing recursive filters with an *arbitrary* frequency response.

```
100 'RECURSIVE FILTER
110 '
120 DIM X[499]                  'holds the input signal
130 DIM Y[499]                  'holds the filtered output signal
140 '
150 GOSUB XXXX                  'mythical subroutine to calculate the recursion
160 '                           'coefficients: A0, A1, A2, B1, B2
170 '
180 GOSUB XXXX                  'mythical subroutine to load X[ ] with the input data
190 '
200 FOR I% = 2 TO 499
210    Y[I%] = A0*X[I%] + A1*X[I%-1] + A2*X[I%-2] + B1*Y[I%-1] + B2*Y[I%-2]
220 NEXT I%
230 '
240 END
```

TABLE 19-1

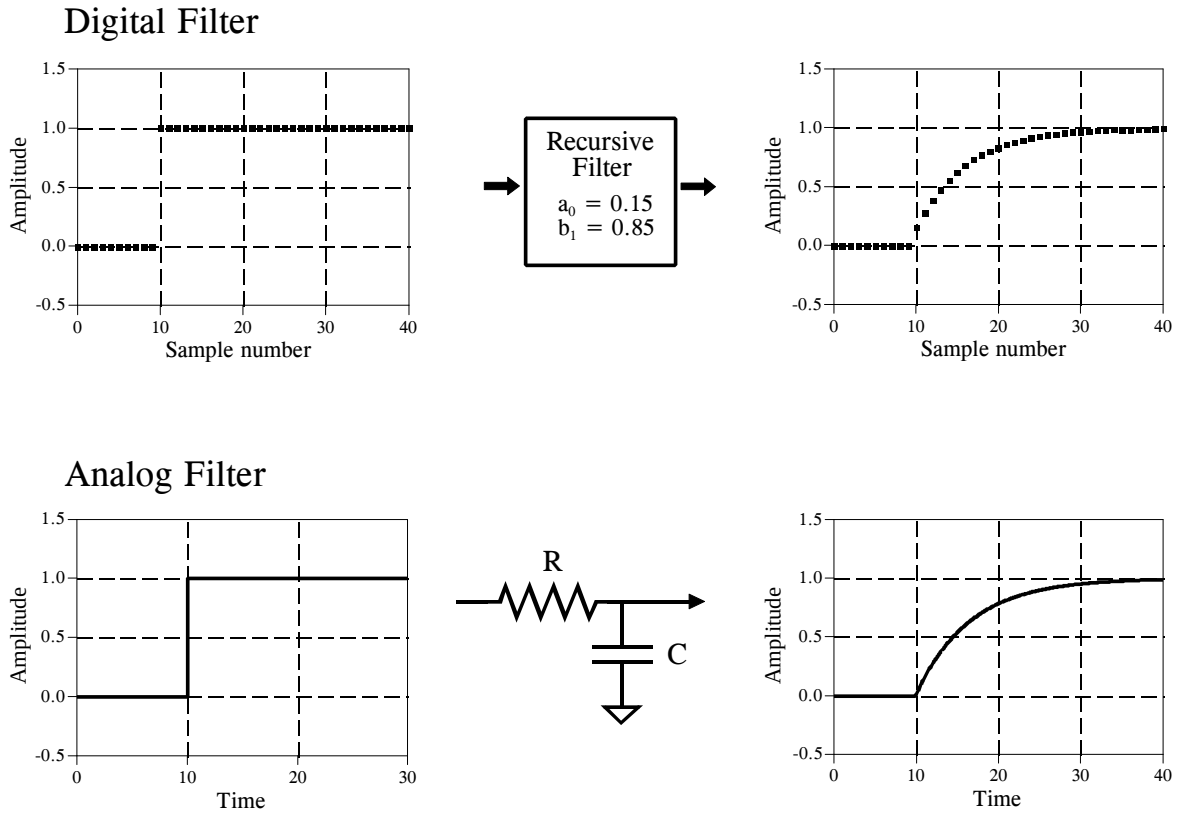


FIGURE 19-2
Single pole low-pass filter. Digital recursive filters can mimic analog filters composed of resistors and capacitors. As shown in this example, a single pole low-pass recursive filter smoothes the edge of a step input, just as an electronic RC filter.

Single Pole Recursive Filters

Figure 19-2 shows an example of what is called a **single pole** low-pass filter. This recursive filter uses just two coefficients, $a_0 = 0.15$ and $b_1 = 0.85$. For this example, the input signal is a step function. As you should expect for a low-pass filter, the output is a smooth rise to the steady state level. This figure also shows something that ties into your knowledge of electronics. This low-pass recursive filter is completely analogous to an electronic low-pass filter composed of a single resistor and capacitor.

The beauty of the recursive method is in its ability to create a wide variety of responses by changing only a few parameters. For example, Fig. 19-3 shows a filter with three coefficients: $a_0 = 0.93$, $a_1 = -0.93$ and $b_1 = 0.86$. As shown by the similar step responses, this digital filter mimics an electronic RC high-pass filter.

These single pole recursive filters are definitely something you want to keep in your DSP toolbox. You can use them to process digital signals just as you would use RC networks to process analog electronic signals. This includes everything you would expect: DC removal, high-frequency noise suppression, wave shaping, smoothing, etc. They are easy to program, fast

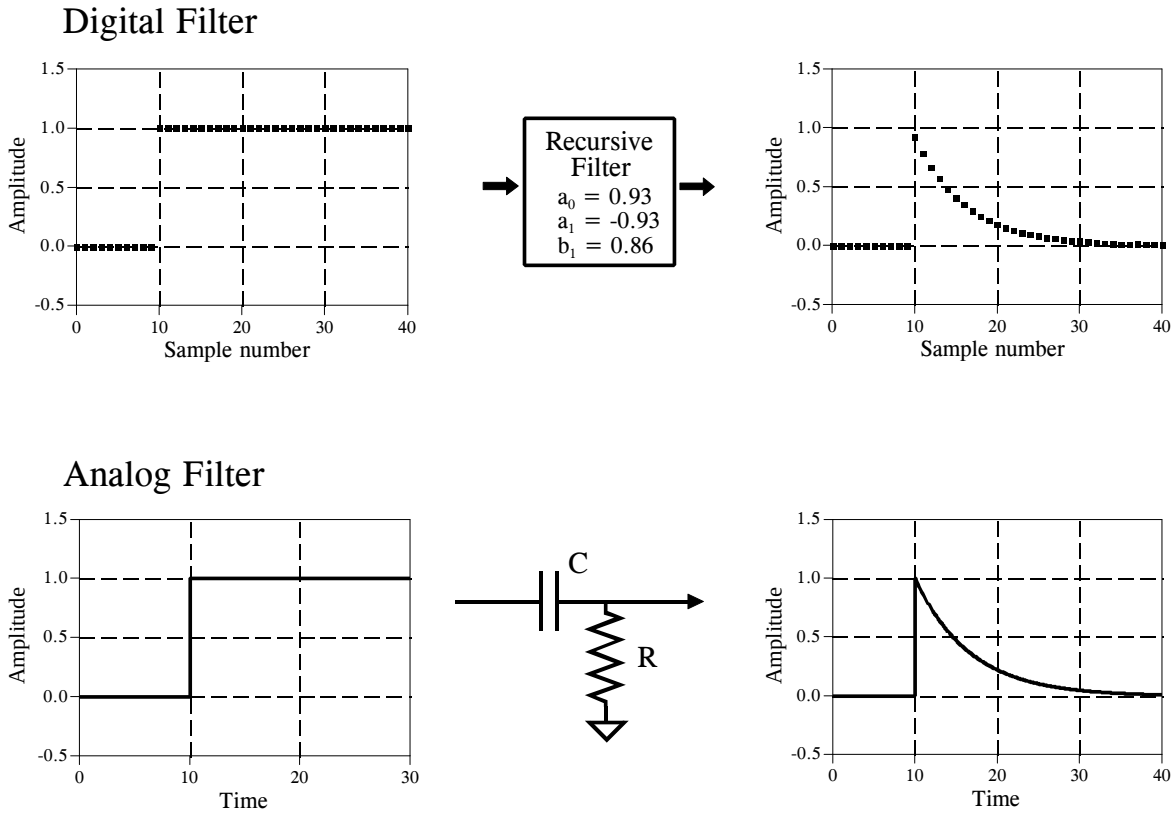


FIGURE 19-3
Single pole high-pass filter. Proper coefficient selection can also make the recursive filter mimic an electronic RC high-pass filter. These single pole recursive filters can be used in DSP just as you would use RC circuits in analog electronics.

to execute, and produce few surprises. The coefficients are found from these simple equations:

EQUATION 19-2
Single pole low-pass filter. The filter's response is controlled by the parameter, *x*, a value between zero and one.

$$a_0 = 1 - x$$
$$b_1 = x$$

EQUATION 19-3
Single pole high-pass filter.

$$a_0 = (1 + x)/2$$
$$a_1 = -(1 + x)/2$$
$$b_1 = x$$

The characteristics of these filters are controlled by the parameter, *x*, a value between zero and one. Physically, *x* is the amount of *decay* between adjacent samples. For instance, *x* is 0.86 in Fig. 19-3, meaning that the value of each sample in the output signal is 0.86 the value of the sample before it. The higher the value of *x*, the slower the decay. Notice that the

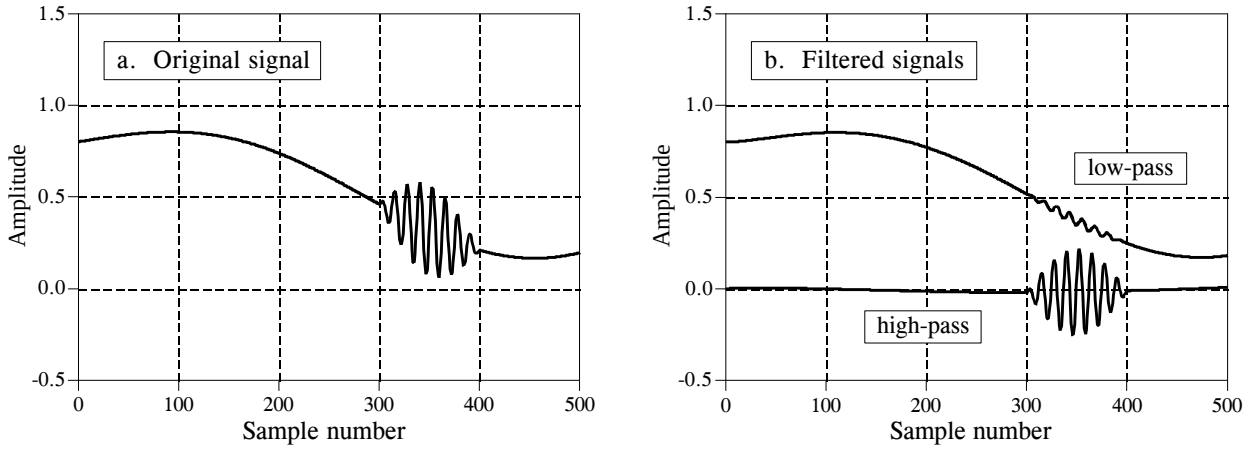


FIGURE 19-4
Example of single pole recursive filters. In (a), a high frequency burst rides on a slowly varying signal. In (b), single pole low-pass and high-pass filters are used to separate the two components. The low-pass filter uses $x = 0.95$, while the high-pass filter is for $x = 0.86$.

filter becomes *unstable* if x is made greater than one. That is, any nonzero value on the input will make the output increase until an overflow occurs.

The value for x can be directly specified, or found from the desired *time constant* of the filter. Just as $R \times C$ is the number of seconds it takes an RC circuit to decay to 36.8% of its final value, d is the number of samples it takes for a recursive filter to decay to this same level:

EQUATION 19-4
Time constant of single pole filters. This equation relates the amount of decay between samples, x, with the filter's time constant, d, the number of samples for the filter to decay to 36.8%.

$$x = e^{-1/d}$$

For instance, a sample-to-sample decay of $x = 0.86$ corresponds to a time constant of $d = 6.63$ samples (as shown in Fig 19-3). There is also a fixed relationship between x and the -3dB *cutoff frequency*, f_C, of the digital filter:

EQUATION 19-5
Cutoff frequency of single pole filters. The amount of decay between samples, x, is related to the cutoff frequency of the filter, f_C, a value between 0 and 0.5.

$$x = e^{-2\pi f_C}$$

This provides three ways to find the "a" and "b" coefficients, starting with the time constant, the cutoff frequency, or just directly picking x.

Figure 19-4 shows an example of using single pole recursive filters. In (a), the original signal is a smooth curve, except a burst of a high frequency sine wave. Figure (b) shows the signal after passing through low-pass and high-pass filters. The signals have been separated fairly well, but not perfectly, just as if simple RC circuits were used on an analog signal.

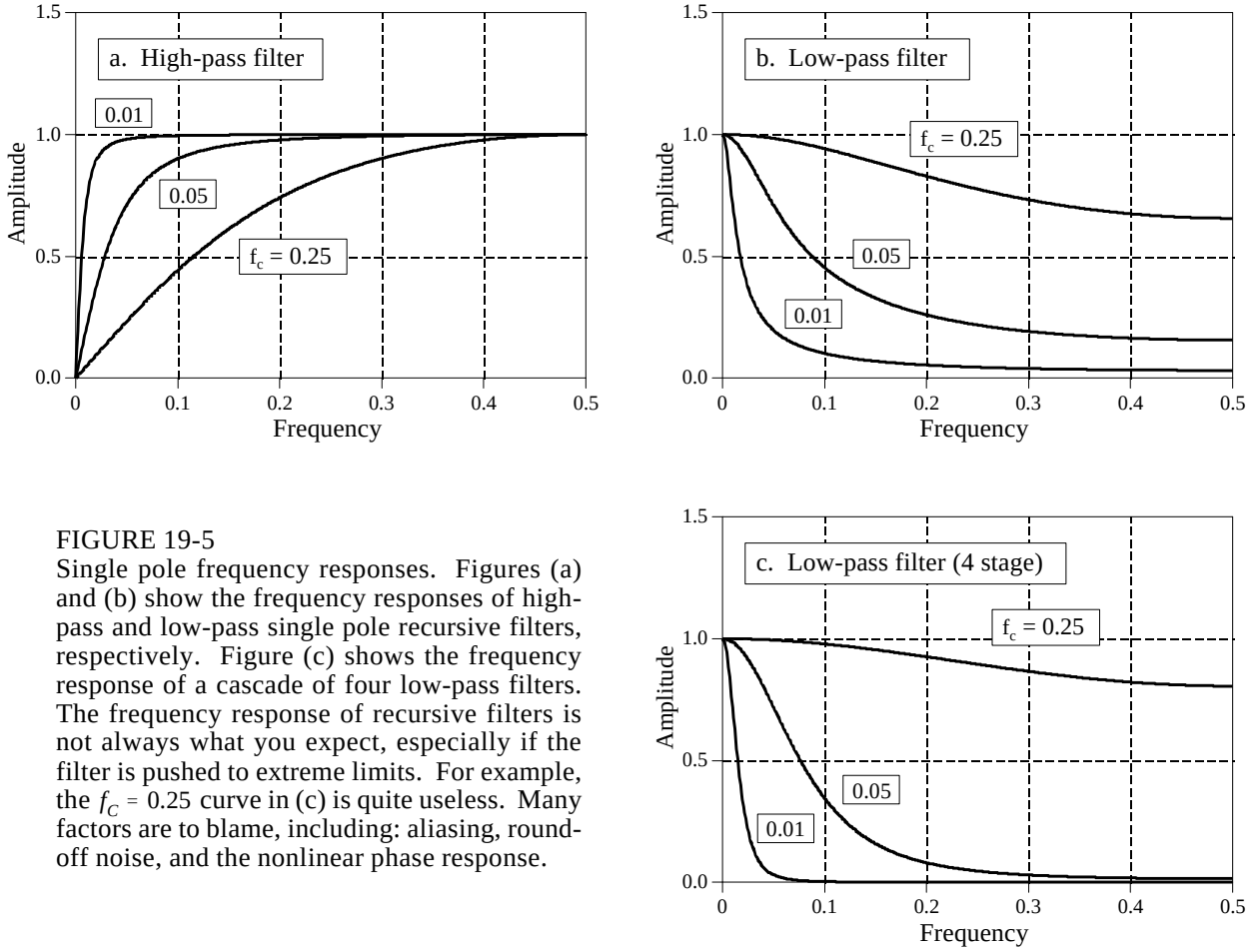


FIGURE 19-5
Single pole frequency responses. Figures (a) and (b) show the frequency responses of high-pass and low-pass single pole recursive filters, respectively. Figure (c) shows the frequency response of a cascade of four low-pass filters. The frequency response of recursive filters is not always what you expect, especially if the filter is pushed to extreme limits. For example, the f_C = 0.25 curve in (c) is quite useless. Many factors are to blame, including: aliasing, round-off noise, and the nonlinear phase response.

Figure 19-5 shows the frequency responses of various single pole recursive filters. These curves are obtained by passing a delta function through the filter to find the filter's impulse response. The FFT is then used to convert the impulse response into the frequency response. In principle, the impulse response is infinitely long; however, it decays below the single precision round-off noise after about 15 to 20 time constants. For example, when the time constant of the filter is d = 6.63 samples, the impulse response can be contained in about 128 samples.

The key feature in Fig. 19-5 is that single pole recursive filters have little ability to separate one band of frequencies from another. In other words, they perform well in the time domain, and poorly in the frequency domain. The frequency response can be improved slightly by cascading several stages. This can be accomplished in two ways. First, the signal can be passed through the filter several times. Second, the z-transform can be used to find the recursion coefficients that combine the cascade into a single stage. Both ways work and are commonly used. Figure (c) shows the frequency response of a cascade of four low-pass filters. Although the stopband attenuation is significantly improved, the roll-off is still terrible. If you need better performance in the frequency domain, look at the Chebyshev filters of the next chapter.

The four stage low-pass filter is comparable to the Blackman and Gaussian filters (relatives of the moving average, Chapter 15), but with a much faster execution speed. The design equations for a four stage low-pass filter are:

EQUATION 19-6
Four stage low-pass filter. These equations provide the "a" and "b" coefficients for a cascade of four single pole low-pass filters. The relationship between x and the cutoff frequency of this filter is given by Eq. 19-5, with the 2π replaced by 14.445.

$$\begin{aligned} a_0 &= (1-x)^4 \\ b_1 &= 4x \\ b_2 &= -6x^2 \\ b_3 &= 4x^3 \\ b_4 &= -x^4 \end{aligned}$$

Narrow-band Filters

A common need in electronics and DSP is to isolate a narrow band of frequencies from a wider bandwidth signal. For example, you may want to eliminate 60 hertz interference in an instrumentation system, or isolate the signaling tones in a telephone network. Two types of frequency responses are available: the *band-pass* and the *band-reject* (also called a **notch filter**). Figure 19-6 shows the frequency response of these filters, with the recursion coefficients provided by the following equations:

EQUATION 19-7
Band-pass filter. An example frequency response is shown in Fig. 19-6a. To use these equations, first select the center frequency, f, and the bandwidth, BW. Both of these are expressed as a fraction of the sampling rate, and therefore in the range of 0 to 0.5. Next, calculate R, and then K, and then the recursion coefficients.

$$\begin{aligned} a_0 &= 1-K \\ a_1 &= 2(K-R)\cos(2\pi f) \\ a_2 &= R^2-K \\ b_1 &= 2R\cos(2\pi f) \\ b_2 &= -R^2 \end{aligned}$$

EQUATION 19-8
Band-reject filter. This filter is commonly called a notch filter. Example frequency responses are shown in Fig. 19-6b.

$$\begin{aligned} a_0 &= K \\ a_1 &= -2K\cos(2\pi f) \\ a_2 &= K \\ b_1 &= 2R\cos(2\pi f) \\ b_2 &= -R^2 \end{aligned}$$

where:

$$K = \frac{1 - 2R\cos(2\pi f) + R^2}{2 - 2\cos(2\pi f)}$$

$$R = 1 - 3BW$$

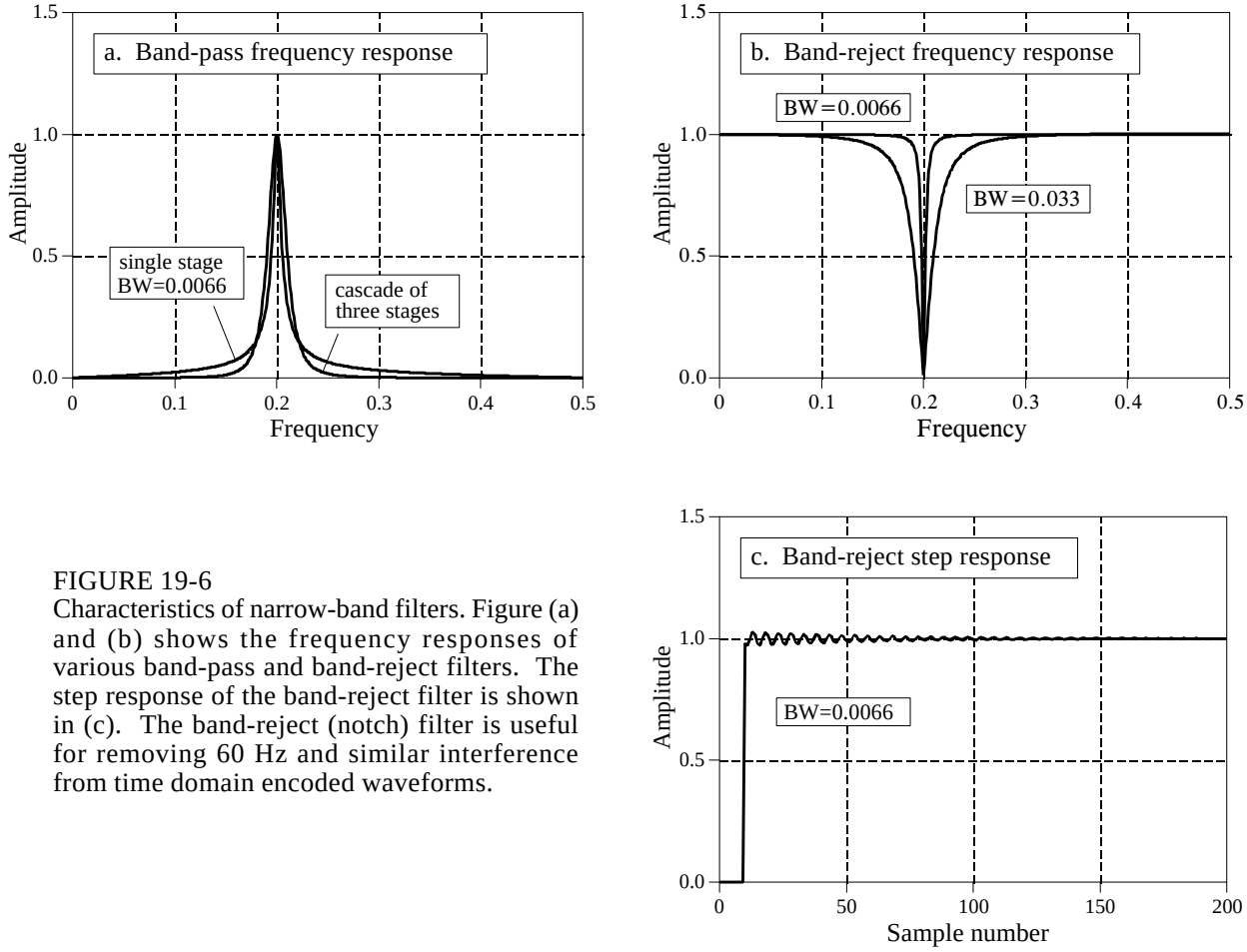


FIGURE 19-6
Characteristics of narrow-band filters. Figure (a) and (b) shows the frequency responses of various band-pass and band-reject filters. The step response of the band-reject filter is shown in (c). The band-reject (notch) filter is useful for removing 60 Hz and similar interference from time domain encoded waveforms.

Two parameters must be selected before using these equations: *f*, the center frequency, and *BW*, the bandwidth (measured at an amplitude of 0.707). Both of these are expressed as a fraction of the sampling frequency, and therefore must be between 0 and 0.5. From these two specified values, calculate the intermediate variables: *R* and *K*, and then the recursion coefficients.

As shown in (a), the band-pass filter has relatively large *tails* extending from the main peak. This can be improved by cascading several stages. Since the design equations are quite long, it is simpler to implement this cascade by filtering the signal several times, rather than trying to find the coefficients needed for a single filter.

Figure (b) shows examples of the band-reject filter. The narrowest bandwidth that can be obtain with single precision is about 0.0003 of the sampling frequency. When pushed beyond this limit, the attenuation of the notch will degrade. Figure (c) shows the step response of the band-reject filter. There is noticeable overshoot and ringing, but its amplitude is quite small. This allows the filter to remove narrowband interference (60 Hz and the like) with only a minor distortion to the time domain waveform.

Phase Response

There are three types of *phase response* that a filter can have: **zero phase**, **linear phase**, and **nonlinear phase**. An example of each of these is shown in Figure 19-7. As shown in (a), the *zero phase* filter is characterized by an impulse response that is symmetrical around sample zero. The actual shape doesn't matter, only that the negative numbered samples are a mirror image of the positive numbered samples. When the Fourier transform is taken of this symmetrical waveform, the phase will be entirely zero, as shown in (b).

The disadvantage of the zero phase filter is that it requires the use of negative indexes, which can be inconvenient to work with. The linear phase filter is a way around this. The impulse response in (d) is identical to that shown in (a), except it has been shifted to use only positive numbered samples. The impulse response is still symmetrical between the left and right; however, the location of symmetry has been shifted from zero. This shift results in the phase, (e), being a *straight line*, accounting for the name: *linear phase*. The slope of this straight line is directly proportional to the amount of the shift. Since the shift in the impulse response does nothing but produce an identical shift in the output signal, the linear phase filter is equivalent to the zero phase filter for most purposes.

Figure (g) shows an impulse response that is *not* symmetrical between the left and right. Correspondingly, the phase, (h), is *not* a straight line. In other words, it has a *nonlinear phase*. Don't confuse the terms: *nonlinear and linear phase* with the concept of *system linearity* discussed in Chapter 5. Although both use the word *linear*, they are not related.

Why does anyone care if the phase is linear or not? Figures (c), (f), and (i) show the answer. These are the **pulse responses** of each of the three filters. The pulse response is nothing more than a positive going step response followed by a negative going step response. The pulse response is used here because it displays what happens to both the rising and falling edges in a signal. Here is the important part: zero and linear phase filters have left and right edges that look the *same*, while nonlinear phase filters have left and right edges that look *different*. Many applications cannot tolerate the left and right edges looking different. One example is the display of an oscilloscope, where this difference could be misinterpreted as a feature of the signal being measured. Another example is in video processing. Can you imagine turning on your TV to find the left ear of your favorite actor looking different from his right ear?

It is easy to make an FIR (finite impulse response) filter have a linear phase. This is because the impulse response (filter kernel) is directly *specified* in the design process. Making the filter kernel have left-right symmetry is all that is required. This is not the case with IIR (recursive) filters, since the recursion coefficients are what is specified, not the impulse response. The impulse response of a recursive filter is *not* symmetrical between the left and right, and therefore has a *nonlinear* phase.

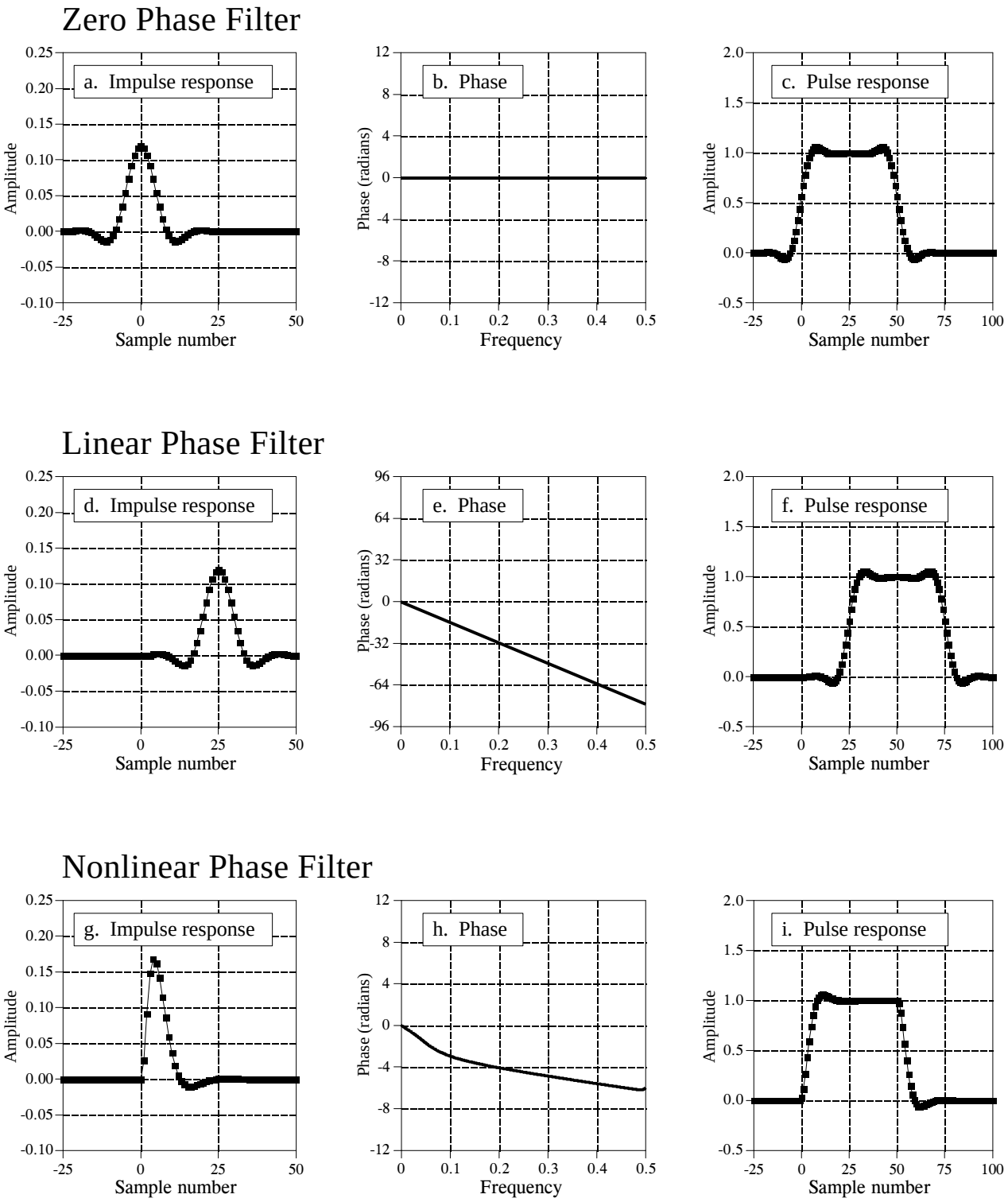


FIGURE 19-7
Zero, linear, and nonlinear phase filters. A *zero phase* filter has an impulse response that has left-right symmetry around sample number zero, as in (a). This results in a frequency response that has a phase composed entirely of zeros, as in (b). Zero phase impulse responses are desirable because their step responses are symmetrical between the top and bottom, making the left and right edges of pulses look the same, as is shown in (c). *Linear phase* filters have left-right symmetry, but not around sample zero, as illustrated in (d). This results in a phase that is linear, that is, a straight line, as shown in (e). The linear phase pulse response, shown in (f), has all the advantages of the zero phase pulse response. In comparison, the impulse responses of *nonlinear phase* filters are not symmetrical between the left and right, as in (g), and the phases are not a straight line, as in (h). The worst part is that the left and right edges of the pulse response are not the same, as shown in (i).

Analog electronic circuits have this same problem with the phase response. Imagine a circuit composed of resistors and capacitors sitting on your desk. If the input has always been zero, the output will also have always been zero. When an impulse is applied to the input, the capacitors quickly charge to some value and then begin to exponentially decay through the resistors. The impulse response (i.e., the output signal) is a combination of these various decaying exponentials. The impulse response *cannot* be symmetrical, because the output was zero before the impulse, and the exponential decay never quite reaches a value of zero again. Analog filter designers attack this problem with the **Bessel filter**, presented in Chapter 3. The Bessel filter is designed to have as linear phase as possible; however, it is far below the performance of digital filters. The ability to provide an *exact* linear phase is a clear advantage of digital filters.

Fortunately, there is a simple way to modify recursive filters to obtain a *zero phase*. Figure 19-8 shows an example of how this works. The input signal to be filtered is shown in (a). Figure (b) shows the signal after it has been filtered by a single pole low-pass filter. Since this is a nonlinear phase filter, the left and right edges do not look the same; they are inverted versions of each other. As previously described, this recursive filter is implemented by starting at sample 0 and working toward sample 150, calculating each sample along the way.

Now, suppose that instead of moving from sample 0 toward sample 150, we start at sample 150 and move toward sample 0. In other words, each sample in the output signal is calculated from input and output samples to the *right* of the sample being worked on. This means that the recursion equation, Eq. 19-1, is changed to:

$$y[n] = a_0 x[n] + a_1 x[n+1] + a_2 x[n+2] + a_3 x[n+3] + \cdots + b_1 y[n+1] + b_2 y[n+2] + b_3 y[n+3] + \cdots$$

EQUATION 19-9
The *reverse* recursion equation. This is the same as Eq. 19-1, except the signal is filtered from left-to-right, instead of right-to-left.

Figure (c) shows the result of this **reverse filtering**. This is analogous to passing an analog signal through an electronic RC circuit while running time *backwards*. !esrevinu eht pu-wercs nac lasrever emit -noituaC

Filtering in the reverse direction does not produce any benefit in itself; the filtered signal still has left and right edges that do not look alike. The magic happens when forward and reverse filtering are *combined*. Figure (d) results from filtering the signal in the forward direction and then filtering again in the reverse direction. Voila! This produces a *zero phase* recursive filter. In fact, *any* recursive filter can be converted to zero phase with this **bidirectional filtering** technique. The only penalty for this improved performance is a factor of two in execution time and program complexity.

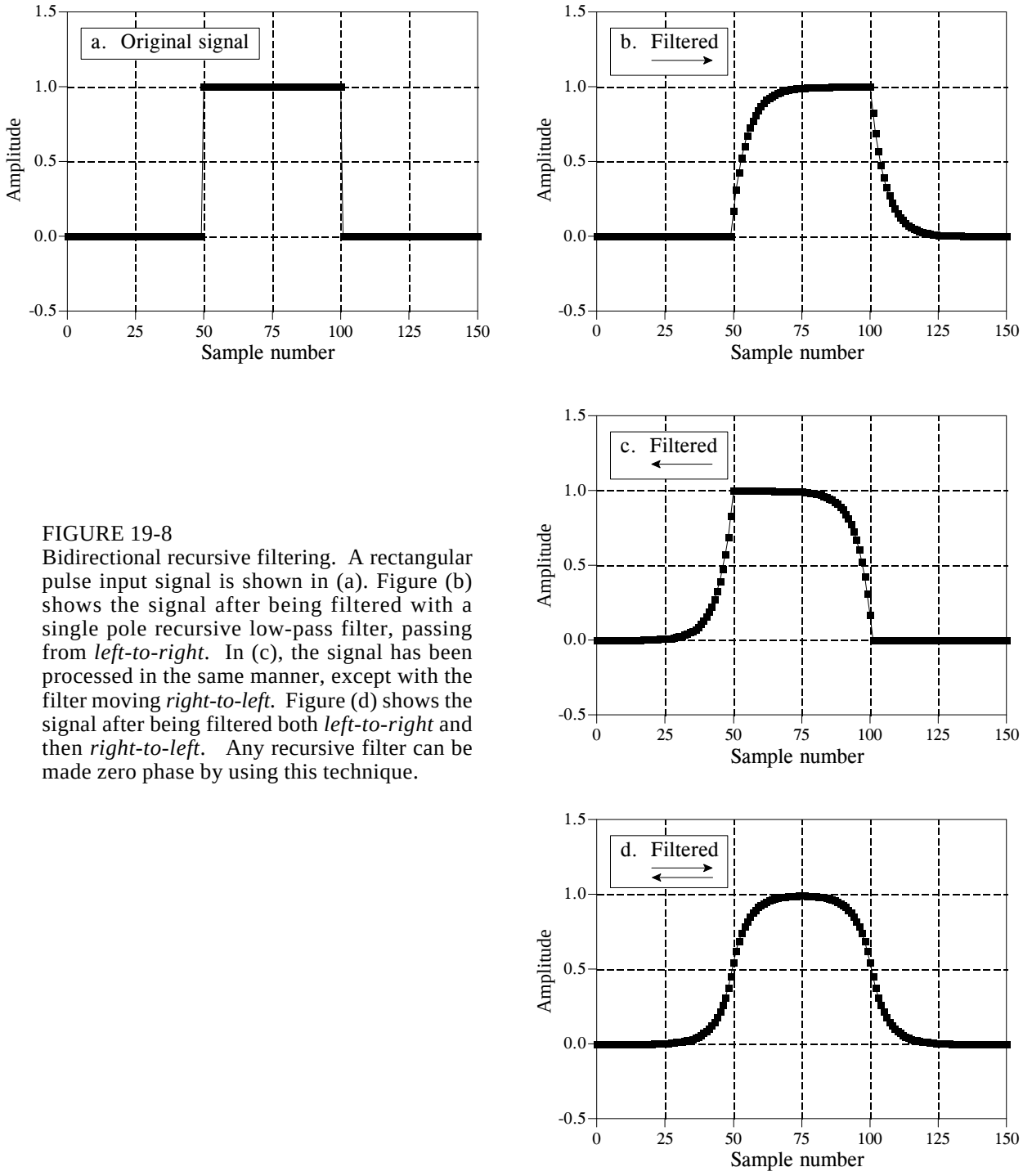


FIGURE 19-8
Bidirectional recursive filtering. A rectangular pulse input signal is shown in (a). Figure (b) shows the signal after being filtered with a single pole recursive low-pass filter, passing from *left-to-right*. In (c), the signal has been processed in the same manner, except with the filter moving *right-to-left*. Figure (d) shows the signal after being filtered both *left-to-right* and then *right-to-left*. Any recursive filter can be made zero phase by using this technique.

How do you find the impulse and frequency responses of the overall filter? The magnitude of the frequency response is the same for each direction, while the phases are opposite in sign. When the two directions are combined, the magnitude becomes *squared*, while the phase cancels to *zero*. In the time domain, this corresponds to convolving the original impulse response with a left-for-right flipped version of itself. For instance, the impulse response of a

single pole low-pass filter is a one-sided exponential. The impulse response of the corresponding bidirectional filter is a one-sided exponential that decays to the right, convolved with a one-sided exponential that decays to the left. Going through the mathematics, this turns out to be a double-sided exponential that decays both to the left and right, with the same decay constant as the original filter.

Some applications only have a portion of the signal in the computer at a particular time, such as systems that alternately input and output data on a continuing basis. Bidirectional filtering can be used in these cases by combining it with the overlap-add method described in the last chapter. When you come to the question of how long the impulse response is, don't say "infinite." If you do, you will need to pad each signal segment with an *infinite* number of zeros. Remember, the impulse response can be truncated when it has decayed below the round-off noise level, i.e., about 15 to 20 time constants. Each segment will need to be padded with zeros on both the left and right to allow for the expansion during the bidirectional filtering.

Using Integers

Single precision floating point is ideal to implement these simple recursive filters. The use of integers is possible, but it is much more difficult. There are two main problems. First, the round-off error from the limited number of bits can degrade the response of the filter, or even make it unstable. Second, the fractional values of the recursion coefficients must be handled with integer math. One way to attack this problem is to express each coefficient as a fraction. For example, 0.15 becomes 19/128. Instead of multiplying by 0.15, you first multiply by 19 and then divide by 128. Another way is to replace the multiplications with look-up tables. For example, a 12 bit ADC produces samples with a value between 0 and 4095. Instead of multiplying each sample by 0.15, you pass the samples through a look-up table that is 4096 entries long. The value obtained from the look-up table is equal to 0.15 times the value entering the look-up table. This method is very fast, but it does require extra memory; a separate look-up table is needed for each coefficient. Before you try either of these integer methods, make sure the recursive algorithm for the moving average filter will not suit your needs. It *loves* integers.