Traitement du Signal :
algorithmique et programmation

- Résumé -

Dernière mise à jour : 1/09/2005

Sommaire

Introduction

Le thème des études effectuées est le **traitement de signal numérique temps-réel et embarqué.**

La notion de temps réel est relative à l'application. Dans le cas où le signal est un son, il va être relatif à la perception que l'on en a, par exemple le décalage entre le son de départ et le son traité. Dans le cas de la détection d'un événement dans un signal, elle sera considérée comme effectuée en temps réel si elle s'effectue dans un temps suffisamment court que le décalage par rapport à la détection instantanée ne soit pas sensible pour l'application. Par contre, dans le cas du traitement d'un son mémorisé dans un fichier, avec génération d'un autre fichier, le signal est complètement connu et défini avant le traitement. Le traitement est en temps différé.

Aujourd'hui on peut réaliser un traitement du signal en temps réel sur un processeur spécialisé de type **DSP**, sur un **microcontrôleur** ou sur un simple **ordinateur** (par exemple muni d'une carte-son, si l'on veut traiter des sons, ou d'une carte possédant des entrées-sorties analogiques avec une bande passante plus grande que la bande audio). Dans les deux premiers cas on pourra parler de vrai temps-réel, dans le deuxième plutôt de temps pseudo-réel.

I) Notions de traitements en temps réel

I.1) Principe

Pour qu'un traitement s'effectue vraiment en temps réel, il faut que le traitement d'un échantillon se fasse en une durée inférieure à $1/f_e$, où f_e est la fréquence d'échantillonnage. Par exemple, pour une fréquence d'échantillonnage de 44100Hz, cela représente une durée de 23µs environ. Mais ceci n'est valable que sur un système de traitement spécialisé, comme par exemple une carte d'évaluation à DSP, sur laquelle on contrôle complètement les processus de traitement.

Sur un ordinateur, on ne contrôle pas complètement son fonctionnement : le système d'exploitation exécute régulièrement ses propres tâches. Dans ce cas, il est préférable de traiter des blocs d'échantillons. Un bloc d'échantillons ne sera restitué qu'une fois complètement traité. Il y a un décalage temporel égal à la durée correspondant au tampon mémoire (buffer) utilisé pour ce bloc, entre le son entrant (son brut) et le son sortant (son traité). On peut alors parler de pseudo-temps réel. Par exemple, avec un buffer de 1024 octets, le décalage est de 24ms environ, une durée qui est décelable par l'oreille.

I.2) Pseudo-temps réel sur PC

Cas d'un traitement sur PC avec entrée/sortie audio par carte son

L'algorithme suivant illustre le principe de traitement par blocs à utiliser : il consiste à acquérir un son à partir de l'entrée audio de la carte son, le traiter, puis l'envoyer dans la sortie de la carte.

Algorithme

- **Initialisation** de la carte son avec les paramètres adéquats : fréquence d'échantillonnage, taille des échantillons en nombre de bits, mode mono ou stéréo, etc
- **Allocation mémoire** pour buffer audio (taille buffer paramétrable)
- **Boucle** infinie :
    - chargement du buffer à partir de l'entrée audio
    - traitement
    - écriture du buffer dans la sortie de la carte (avec attente disponibilité)

Dans le cas où l'entrée est un fichier son, la boucle s'arrête lorsque tous les échantillons du fichier ont été traités.

Dans le cas où le son n'est pas un son réel mais par exemple un signal généré par l'ordinateur, ou un son chargé à partir d'un fichier, cette génération peut-être beaucoup plus rapide que sa durée. On pourrait croire qu'il y a un risque d'encombrement en sortie et que le son ne soit pas restitué correctement ; mais en fait il n'en est rien, la carte son attend la fin de la sortie d'un bloc avant de commencer la sortie du bloc suivant. Le décalage n'est pas perceptible sauf dans certains cas comme les applications musicales où la génération de signal est déclenchée par un clavier.

Cas de la coopération entre un PC et une carte à DSP

On peut par exemple vouloir utiliser la carte son d'un PC pour les entrées-sorties, et la carte à DSP pour les traitements. Dans ce cas on dispose du port parallèle pour les échanges de données entre les 2 systèmes, qui permettra d'obtenir du temps réel.

On peut envisager d'autres configurations. Par exemple, acquisition et 1er traitement sur le PC, puis transfert au DSP, 2^{e} traitement et sortie du son résultant sur DSP.

I.3) "Vrai" temps-réel sur DSP

Sur une carte à DSP, on peut se permettre de réaliser du "vrai" temps-réel, car on peut contrôler complètement le système.

Sur la carte d'évaluation utilisée, le traitement peut être synchronisé sur l'acquisition de chaque échantillon, qui elle est synchronisée sur la fréquence d'échantillonnage du CODEC.

L'exemple de programme le plus simple pour comprendre ce principe est programmé est "pass.asm", qui ne fait que transmettre le signal de l'entrée analogique à la sortie analogique, permettant ainsi de tester le CODEC. Il fonctionne par scrutation (mode polling), et non par interruption.

Son principe est l'acquisition des échantillons les uns après les autres, alternée avec l'envoi en sortie de ces mêmes échantillons.

L'algorithme général pouvant être utilisé pour tous les programmes de traitement est le suivant :

- Initialisation du DSP (fréquence d'horloge, etc)
- Initialisation du CODEC (fréquence d'échantillonnage, etc)
- Initialisation du traitement
- Répéter indéfiniment :
    - Attente d'un échantillon du CODEC
    - Acquisition des échantillons droite et gauche
    - Saut à la routine de traitement (génération échantillons droite et gauche traités)
    - Envoi des échantillons résultats en sorties du CODEC

C'est celui du programme "pass.asm", excepté le traitement des échantillons.

L'acquisition est synchronisée sur la fréquence d'échantillonnage : une instruction boucle sur elle-même tant qu'un bit d'un registre système reste à 0 :

```
jset    #2,x:SSISR0,*       ; wait for frame sync to pass (saut sur place si bit 2 du registre de statut
                            ; de l'interface série synchrone est à 1)
jclr    #2,x:SSISR0,*       ; wait for frame sync (idem ci-dessus mais bit à 0)
```

II) Filtrage

Dans ce chapitre nous allons étudier les différentes manières de programmer des filtrages, d'abord en langage évolué (C/C++) puis en assembleur DSP56303.

II.1) Algorithmique et programmation en langage évolué (C/C++)

II.1.1) Introduction

On part de l'équation de récurrence, la relation donnant l'échantillon de sortie s(n) en fonction des échantillons précédents d'entrée e(n-i) et de sortie s(n-i) :

$$s(n)=a_0e(n)+a_1e(n-1)+\ldots+a_Le(n-L)+$$
$$b_1s(n-1)+b_2s(n-2)+...+b_Ks(n-K)$$

Dans cet exemple il y a L+1 échantillons (l'échantillon présent et L échantillons passés) d'entrée et K échantillons de sortie utilisés dans le calcul du nouvel échantillon.

L'implémentation temps-réel de cette équation nécessite de mémoriser L échantillons d'entrée et K échantillons de sortie. L'échantillon présent e(n) n'a pas besoin d'être mémorisé pour le calcul de s(n). Par contre il le sera àprès le calcul de s(n), en prévision du calcul de s(n+1) à la prochaine itération de l'algorithme.

Cette opération simple permet d'implémenter toutes sortes de filtres : passe-bas, passe-haut ou passe-bande, de tous ordres, modélisant des filtres analogiques ou non. De plus, on peut se limiter à l'étude de l'ordre 2 (K=2 et/ou L=2) : tout filtre d'ordre supérieur peut être décomposé en un certain nombre de cellules élémentaires d'ordre 1 ou 2 (mises en cascade).

L'opération de base est appelée MAC pour "multiplication et accumulation". Les DSP sont optimisés pour cette opération.

Le programme d'utilisation doit comporter les différentes étapes suivantes :

- allocation mémoire pour les coefficients a_i et b_i, les échantillons passés de e et de s (ou déclaration de tableaux statiques, respectivement a[NB_A], b[NB_B], em[NB_A-1], sm[NB_B]), et le bloc d'échantillons, où NB_A et NB_B sont les nombres de coefficients a_i et b_i (resp.)
- boucle infinie
  - lecture d'un bloc d'échantillons d'entrée pointé par e
  - pour chaque échantillon i de ce bloc
    s[i]=iir(e[i])
  - écriture du bloc d'échantillons pointé par s

Les entrées et les sorties peuvent être soit des fichiers son ou la carte son. Dans le cas de l'utilisation d'une carte son, le séquencement physique des différentes opérations pour qu'il n'y ait pas de discontinuités dans l'acquisition et la sortie des sons, est géré par les drivers de cette carte (interfaces entre le langage de programmation et l'électronique de la carte).

Dans les implémentations en C décrites ici, on utilise des variables globales. En général, pour améliorer la clarté du code, on évite au maximum de le faire. On le fait ici pour rendre le programme plus lisible.

II.1.2) Implémentation directe

La fonction C donnant cet échantillon de sortie pourrait s'écrire de la façon suivante :

```
void
iir(float *e, float *s)
{
    int     i;

    (*s)=0.;
    for(i=0 ; i<na ; i++)
        (*s)+=*(a+i)**(e-i);
    for(i=0 ; i<nb ; i++)
        (*s)+=*(b+i)**(s-i-1);
}
```

na et nb sont respectivement le nombre de coefficients a_i et le nombre de coefficients b_i.

Comme précisé ci-dessus, le passage de variables par paramètres de le fonction serait plus rigoureuse :

```
void iir(float *e, float *s, float *a, int na, float *b, int nb)
...
```

Attention, les coefficients b_i sont différents de ceux de l'équation : par commodité de programmation leur indice commence ici à 0.

Dans le cas d'un filtre du 2e ordre, on aurait L=2 et K=2

$$s(n)=a_0e(n)+a_1e(n-1)+a_2e(n-2)+b_1s(n-1)+b_2s(n-2)$$

avec

$$na=L+1$$
$$nb=L$$

L'utilisation de la fonction de filtrage pourrait s'effectuer de la manière suivante :

```
//variables globales (à éviter !) :
float       a[3], b[2], *e, *s;
int         na, nb ;

// ….
// allocation mémoire pour buffers son : e pour l'entrée et s pour la sortie
// ….
// Initialisation des coefficients du filtre a(i) et b(i), nombres na et nb
// ….
// chargement du son complet dans le buffer d'entrée
// ….
// initialisations :
em[0]=*(e+0);
em[1]=0;
sm[0]=sm[1]=0.;

for(i=0 ; i<2 ; i++)                // d'abord gestion des effets de bord : les tous premiers échantillons à 0
    *(s+i)=0.;
for(i=2 ; i<nb_ech ; i++)
    iir(e+i, s+i);
```

Cette méthode est la plus simple : le lien entre l'équation et l'algorithme est direct. Son inconvénient est qu'elle n'est pas temps-réel : le signal à traiter doit être complètement stocké en mémoire avant d'être traité. Pour que le traitement s'effectue en temps-réel, il faut qu'un échantillon de sortie soit généré avant que le prochain échantillon d'entrée soit disponible. Il faut mémoriser les échantillons précédents nécessaires (figurant dans l'équation de récurrence). Par exemple, 2 échantillons de sortie et 2 échantillons d'entrée, dans le cas d'un filtre du 2e ordre.

II.1.3) Buffers locaux avec décalages

Le principe de cette méthode est de ne mémoriser que les échantillons d'entrée et de sortie précédents nécessaires. Cette méthode est beaucoup moins coûteuse en mémoire.

On n'a plus besoin de passer l'échantillon d'entrée par pointeur, puisque les échantillons passés nécessaires au calcul du nouvel échantillon de sortie sont mémorisés dans les tampons-mémoire (buffers) locaux. La fonction de filtrage pourrait devenir :

```
float
iir(float e)
{
    int i;
    float s;

    s=a[0]*e;
    for(i=0 ; i <na-1 ; i++)
        s+=a[i+1]*em[i];
    for(i=0 ; i<nb ; i++)
        s+=b[i]*sm[i];
    for(i=na ; i>0 ; i--)
        em[i]=em[i-1];        // décalage vers la droite des em[i]
    em[0]=e;                  // mémorisation de l'échantillon d'entrée courant
    sm[1]=sm[0];              // idem pour sm[i] (1 seule valeur à décaler)
    sm[0]=s;                  // mémorisation de l'échantillon de sortie courant

    return s;
}
```

Un inconvénient de cette méthode est que le décalage des échantillons passés dans leur tableau est nécessaire, ce qui peut devenir coûteux pour les filtres d'ordre important (même si en pratique on utilisera des filtres d'ordre 1 ou 2 mis en cascade pour obtenir des filtres d'ordre supérieur). Une solution consiste à utiliser des buffers circulaires. Le principe de base est que chaque nouvel échantillon vient écraser le plus ancien. En fait concrètement, on utilise 2 pointeurs modulo, qui vont parcourir les tableaux d'échantillons du début à la fin, et re-passer à la première case après la dernière. Exemple de variable modulo 3 :

$$i\%3$$

L'incrémentation de cette variable peut s'effectuer de la manière suivante :

$$i{+}{+}\ ;\ i=i\%3\ ;\ \leftrightarrow\ i=(++i)\%3\ ;$$

$$i=i\%3\ ;\ i{+}{+}\ ;\ \leftrightarrow\ i=(i++)\%3\ ;$$

Résultat : 1er cas : i=0,1,2,0,1,2,0,… ; 2^{e} cas : 1,2,3,1,2,3,1,…

Utilisation :

```
//variables globales (à éviter !) :
float       em[2], sm[2], a[3], b[2], *e, *s;
int         ia=0, ib=0, na, nb;

// ….
// allocation mémoire pour buffers son : e pour l'entrée et s pour la sortie
// ….
// initialisation des coefficients du filtre a(i) et b(i), nombres na et nb
// ….
// initialisation des mémoires d'échantillons :
em[0]=*(e+0);
em[1]=0;
sm[0]=sm[1]=0.;
...                                  // allocation mémoire pour les blocs d'entrée *e et de sortie *s, calcul des coef. a_i et b_i
while(1)                             // boucle infinie
{
  ...                                // acquisition du bloc de signal
  for(i=0 ; i<nb_ech ; i++)          // traitement du bloc de signal
      *(s+i)=iir(*(e+i));
  ....                               // envoi du bloc de signal en sortie
}
```

II.1.4) Buffers circulaires

Dans la mesure où on utilise une mémoire spécifique pour les échantillons d'entrée passés nécessaires au calcul de la récurrence, l'échantillon d'entrée présent n'a plus à être passé par paramètre.

Dans le cas du 2nd degré, la version de la fonction de filtrage avec pointeurs modulo pourrait être par exemple :

```
float
iir(float e)
{
    int i;
    float s;

    s=a[0]*e;
    for(i=0 ; i <2 ; i++)
        s+=a[i+1]*em[(2-i+i_a)%2];
    for(i=0 ; i<2 ; i++)
        s+=b[i]*sm[(1-i+i_b)%2];
    em[i_a]=e;
    sm[i_b]=s;
    i_a=(++i_a)%2;
    i_b=(++i_b)%2;
    return s;
}
```

Avec des variables passées par paramètres à la fonction, les variables d'indice des buffers doivent être passées par pointeurs : on remplace i_a par *i_a et i_b par *i_b. Le début de définition de la fonction serait :

```
float       iir(float e, float *a, float *b, int *ia, int *ib, float *em, float *sm)
...
```

L'utilisation de cette dernière fonction pourrait se faire de la manière suivante :

```
  ...                                       // acquisition du bloc de signal
  for(i=0 ; i<nb_ech ; i++)                 // traitement du bloc de signal
      *(s+i)=iir(*(e+i), a, b, &ia, &ib, em, sm);
  ....                                      // envoi du bloc de signal en sortie
```

Dans certains cas il sera préférable de mettre le traitement du bloc dans une fonction :

```
void
filtrage_buf_2nd_deg(float e, int nb_ech, float *a, int na, float b, int nb, int *ia, int *ib, float *em, float *sm, float *s)
{
        int         i ;

        for(i=0 ; i<nb_ech ; i++)
           *(s+i)=iir(*(e+i), a, na, b, nb, ia, ib, em, sm);
}
```

Remarques :

- on ne retourne pas le buffer par retour de la fonction pour éviter d'avoir à faire une allocation mémoire à l'intérieur de celle-ci ;
- pour les indices de buffer *ia* et *ib*, on utiliserait plutôt un test *if* qu'une opération modulo, moins coûteuse que cette dernière.

Dans le cas général, la fonction deviendrait, avec des modulo :

```
float
iir(float e, float *a, int na, float *b, int nb, int *ia, int *ib, float *em, float *sm)
{
    int i, na2, nb2;
    float s;

    na2=na-1;       // variables intermédiaires pour éviter d'avoir à réaliser plusieurs fois ces soustractions
    nb2=nb-1;
    s=a[0]*e;
    for(i=0 ; i<na2 ; i++)
        s+=a[i+1]*em[(na2-i+*ia)%na2];
    for(i=0 ; i<nb ; i++)
        s+=b[i]*sm[(nb2-i+*ib)%nb];
    em[*ia]=e;
    sm[*ib]=s;
    if(++(*ia)>=na2)
        *ia=0;
    if(++(*ib)>=nb)
        *ib=0;
    return s;
}
```

II.1.5) Forme canonique

La manipulation des schémas-blocs, ou la décomposition de la fonction de transfert entrée/sortie en z peut nous amener à une autre représentation de la fonction de filtrage, dans laquelle on ne mémorise pas des échantillons passés d'entrée et de sortie, mais d'une variable intermédiaire que l'on notera w.

On décompose la fonction de transfert en 2 : l'une purement récursive et l'autre transversale.

$$H(z)=\frac{S(z)}{E(z)}=\frac{S(z)}{W(z)}\times\frac{W(z)}{E(z)}$$

avec

$$\frac{S(z)}{W(z)}=a_0+a_1z^{-1}+a_2z^{-2} \qquad \text{et} \qquad \frac{W(z)}{E(z)}=\frac{1}{1+b_1z^{-1}+b_2z^{-2}}$$

$$\rightarrow \qquad s(n)=a_0w(n)+a_1w(n-1)+a_2w(n-2)$$

$$w(n)=e(n)-b_1w(n-1)-b_2w(n-2)$$

L'avantage de cette forme est qu'il n'y a plus besoin que d'un seul buffer (pour la variable intermédiaire w) au lieu des 2 buffers d'entrée et de sortie des implémentations précédentes.

Dans le cas général on aurait :

$$s(n)=a_0w(n)+a_1w(n-1)+...+a_Lw(n-L)$$

$$w(n)=e(n)-b_1w(n-1)-...-b_Kw(n-K)$$

L'algorithme pourrait être :

- allocations mémoire et initialisations
- boucle infinie
  - lecture de l'échantillon d'entrée e(n)
  - calcul de $w(n)=f_1(e(n), w(n-i)), i=1,\ldots,L$
  - calcul de $s(n)=f_2(w(n), w(n-i)), i=1,\ldots,K$
  - décalage des w(i) : w(n-i)->w(n-i-1), i=1,…,ordre du filtre (=max(K,L))

Exemple de programmation en C (toujours avec des variables globales) :

```
float
iir_can(float e)
{
        int     n, i;
        float   s=0.;

        w[0]=e;
        for(i=0 ; i<nb ; i++)
            w[0]+=b[i]*w[i+1];
        for(i=0 ; i<=na ; i++)
            s+=a[i]*w[i];
        n=(na>=nb)?na:nb;           // n=max(na, nb)
        for(i=n ; i>=1 ; i--)
            w[i]=w[i-1];            // décalage des w(i)
        return s;
}
```

Cette implémentation réalise un décalage des éléments du tableau des valeurs passées de la variable w, et n'utilise pas de buffer circulaire (ce qui ne pose pas vraiment un problème de coût supplémentaire, vu le nombre réduit de décalages réalisés en pratique). Mais on pourrait très bien le faire. On verra que c'est ce qui se passe dans le cas du 56300.

L'utilisation de cette fonction est la même que pour la version directe.

II.1.6) Mise en cascade de filtres

Dans la programmation de filtres d'ordre important avec les DSP à virgule fixe, les erreurs d'arrondis peuvent entraîner une modification des caractéristiques des filtres, voire une instabilité.

Pour cette raison, on évite d'utiliser la forme générale de la fonction de transfert : on la décompose en n fonctions (ou cellules) élémentaires du 2^e et 1^{er} ordre, par une structure en cascade :

$$H(z)=H_1(z)\times H_2(z)\times \ldots\times H_n(z)$$

Par exemple, l'ordre 5 sera obtenu par 2 filtres d'ordre 2 et un filtre d'ordre 1. L'avantage de cette décomposition est d'avoir des valeurs de coefficients élevées donc peu sensibles à la quantification.

Un filtre d'ordre quelconque est stable si toutes les cellules qui le composent le sont. Donc la stabilité du filtre se ramène à l'étude de la stabilité des cellules élémentaires.

Au niveau des schémas-blocs, la mise en cascade consiste à connecter ces modules numériques entre eux comme des modules électroniques, la sortie d'un bloc constituant l'entrée de l'autre. Au niveau du programme, le principe est le même : l'échantillon de sortie d'une fonction constitue l'échantillon d'entrée de la fonction suivante.

```
float     em[3], sm[2], em2[3], sm2[2], em3[3], sm3[2], a[3], b[2], *e, *s;
int       ia, ib, ia2, ib2, ia3, ib3;

...                                          //allocations mémoire et initialisation coefficients et tampons mémoire entrée et sortie
while(1)                                     //boucle infinie
{
  ...                                        //acquisition du bloc de signal
  for(i=0 ; i<nb_ech ; i++)                  //traitement du bloc de signal
  {
      *(s+i)=iir(*(e+i), a, 3, b, 2, &ia, &ib, em, sm);
      *(s+i)=iir(*(s+i), a, 3, b, 2, &ia2, &ib2,  em2, sm2);
      *(s+i)=iir(*(s+i), a, 3, b, 2, &ia3, &ib3, em3, sm3);
  }
  ....                                       //envoi du bloc de signal en sortie
}
```

On peut rechercher une programmation plus compacte (avec des matrices pour les coefficients et pour les échantillons passés d'entrée et de sortie, dans le cas de l'implémentation directe), et en mettant les différents appels à la fonction iir dans une boucle. Les tampons mémoire d'échantillons et les indices de pointage dans ces tampons doivent alors être constitués par des tableaux à 2 dimensions.

Dans le cas d'une allocation dynamique (taille des tableaux définies en cours d'exécution des programmes), les variables à utiliser sont des pointeurs doubles (correspondant à des adresses de tableaux à 2 dimensions), on peut utiliser l'instruction "calloc", qui possède la propriété d'initialiser les tableaux à 0 :

Avec des pointeurs

Déclarations :

```
float **em, **sm, **ia, **ib, **a, **b ;
```

Allocations mémoire :

```
ia=(int*)calloc(nb_fltr, sizeof(int));          // tableau de 'nb_fltr' éléments
ib=(int*)calloc(nb_fltr, sizeof(int));
em=(float**)calloc(nb_fltr, sizeof(float*));// tableau à 2 dimensions : 'nb_fltr' colonnes (et 3 lignes : voir plus bas)
sm=(float**)calloc(nb_fltr, sizeof(float*));
a=(float**)calloc(nb_fltr, sizeof(float*));
b=(float**)calloc(nb_fltr, sizeof(float*));

//allocation des lignes
for(i=0 ; i<nb_fltr ; i++)
```

```
{
        *(em+i)=(float)calloc(2, sizeof(float));
        *(sm+i)=(float)calloc(2, sizeof(float));
        *(a+i)=(float)calloc(3, sizeof(float));
        *(b+i)=(float)calloc(2, sizeof(float));
}
```

Appel de la fonction :

```
for(i=0 ; i<nb_ech ; i++)
{
    *(s+i)= *(e+i);
    for(j=0 ; j<nb_fltr ; j++)
        *(s+i)=iir(*(s+i), a, 3, b, 2, *(ia+j), *(ib+j), *(em+j), *(sm+j));
}
```

Avec des tableaux

On peut utiliser des tableaux à la place des pointeurs. L'intérêt est qu'il n'y a pas d'allocation mémoire à faire. L'inconvénient est que les dimensions sont figées une fois pour toutes dans l'exécution du programme.

Allocation mémoire :

```
int ia[NB_FLTR], int ib[NB_FLTR];
float a[NB_FLTR][3], float b[NB_FLTR][2];
float em[NB_FLTR][2], float sm[NB_FLTR][2];
```

Déclaration de la fonction :

```
float iir(float e, float a[], int na, float b[], int nb, int ia[], int ib[], float em[], float sm[]);
```

Appel de la fonction :

```
for(i=0 ; i<nb_ech ; i++)
{
    s[i]=e[i];
    for(j=0 ; j<nb_flt ; j++)
        s[i]=iir(s[i], a[][i], na, b[][i], nb, ia[][i], ib[][i], em[][i], sm[][i]);
}
```

Avec la forme canonique, le principe est le même. La zone mémoire w remplace les 2 zones mémoire em et sm. Les éléments de cette zone mémoire étant tous décalés à chaque échantillon, les indices de pointage dans les tableaux de coefficents ia et ib ne sont plus nécessaires :

```
w=(float**)calloc(nb_fltr, sizeof(float*));
for(i=0 ; i<nb_fltr ; i++)
        *(w+i)=(float)calloc(3, sizeof(float));

for(i=0 ; i<nb_ech ; i++)
{
    *(s+i)= *(e+i);
    for(j=0 ; j<nb_flt ; j++)
        *(s+i)=iir_can(*(s+i), a, na, b, nb, *(w+j));
}
```

II.1.7) Programmes C/C++

Ce paragraphe donne un exemple d'implémentation d'un filtrage passe-bas du 2nd ordre en C, puis l'implémentation équivalente en C++.

Exemple de programme C

En langage C, pour éviter d'avoir à passer tous les paramètres du filtre à la fonction de filtrage IIR, il est préférable d'utiliser des structures de donnée. Mais pour éviter d'avoir des variables globales, il est quand même nécessaire de passer un pointeur vers une variable dont le type est cette structure, en paramètre à la fonction de filtrage. Tous les membres de la structure ne sont pas utilisés dans cet exemple.

```
typedef struct {
    // paramètres du filtre analogique
        FLOAT   xi;                     // coefficient de surtension
        FLOAT   fcoup;                  // fréquence de coupure
        UBYTE   type;                   // type : passe-bas, passe-haut, etc
        UBYTE   pente;                  // pente : -6dB/octave, -12, etc
    // paramètre du filtre numérique
        FLOAT   *a, *b;                 // buffers pour coefficients du filtre (respectivement numérateur et dénominateur)
        UWORD   na, nb;                 // nombre de coefficients du filtre (resp. a(i) et b(i))
        UBYTE   ia, ib;                 // indice de pointage dans le buffer des coefficients a et b resp.
        FLOAT   *em, *sm;               // buffers pour mémorisation des échantillons passés d'entrée et de sortie resp.
} Filtre;

FLOAT
iir(FLOAT e, Filtre *f)
{
        ULONG   i, na2, nb2;
        FLOAT   s;

        s=f->a[0]*e;
        na2=f->na-1;
        nb2= f->nb-1;
        for(i=0 ; i<na2 ; i++)
                s+= f->a[i+1]* f->em[(f->na2-i+ f->ia)%na2];
        for(i=0 ; i< f->nb ; i++)
                s+= f->b[i]* f->sm[(nb2-i+ f->ib)% f->nb];
        f->em[f->ia]=e;
        f->sm[f->ib]=s;
        if(++(f->ia)>=na2)
                f->ia=0;
        if(++(f->ib)>= f->nb)
                f->ib=0;
        return s;
}

// Filtrage PASSE-BAS 2e degré, pour 1 échantillon
FLOAT
passebas_2deg_iir(FLOAT e, FLOAT fe, Filtre *f)
{
        FLOAT   k, k1, k1c, k2, a[3], b[2];

        k1= fe/M_PI/ f->fc;
        k1c=k1*k1;
        k2=2*xi*k1;
        k=1+k2+k1c;
        f->a[0]=1./k;
        f->a[1]=2.* f->a[0];
        f->a[2]= f->a[0];
        f->b[0]=-2.*(1.-k1c)/k;
        f->b[1]=-(1-k2+k1c)/k;
        return iir(e, f->a, 3, f->b, 2, f->ia, f->ib, f->em, f->sm);
}
```

Exemple de programme C++

Le langage C++ apporte l'avantage de pouvoir accéder aux variables du filtre sans avoir à les passer en paramètre à la fonction, et sans même passer un pointeur de filtre en paramètre : la fonction est elle-même programmée comme appartenant au filtre (encapsulée). Le filtre est un objet du type Filtre, où Filtre est une classe. L'implémentation équivalente à la version C donnée ci-dessus pourrait être :

```
class Filtre {
   public:
            Filtre ();                          // constructeur (allocations mémoire)
            ~Filtre ();                         // destructeur (désallocations)
            FLOAT   iir(FLOAT e);
            FLOAT   passebas_2deg_iir(FLOAT e, FLOAT fe);
            VOID    filtrage_buf_mod_fc(FLOAT *buf, ULONG nb_ech, FLOAT fe, FLOAT amfc, FLOAT fmfc, ULONG *ptr);

  //variables publiques, pour pouvoir être utilisées en dehors des fonctions ci-dessus
            FLOAT   xi;
            FLOAT   fc;

  //variables privées
   private:
            UBYTE   type;
            UBYTE   pente;
            FLOAT   *a, *b;                     // buffers pour coefficients du filtre (respectivement numérateur et dénominateur)
            UWORD   na, nb;                     // nombre de coefficients du filtre (resp. a(i) et b(i))
            UBYTE   ia, ib;                     // indice de pointage dans le buffer des coefficients a et b resp.
            FLOAT   *em, *sm;                   // buffers pour mémorisation des échantillons passés d'entrée et de sortie resp.
};

// Constructeur
Filtre::Filtre ()
{
            na=3;
            nb=2;
            a=new FLOAT[na];
            b=new FLOAT[nb];
            em=new FLOAT[na-1];
            sm=new FLOAT[nb];
            ia=ib=0;
}
// Destructeur
Filtre::~Filtre ()
{
            delete a;
            delete b;
            delete em;
            delete sm;
}
// Filtrage IIR d'un échantillon
//  -forme directe
//  -échantillon d'entrée courant utilisé avant mémorisation
FLOAT
Filtre::iir(FLOAT e)
{
            ULONG   i;
            UBYTE   na2, nb2;           // variables locales pour réduire le nombre d'opérations
            FLOAT   s;

            s=a[0]*e;
            na2=na-1;
            nb2=nb-1;
            for(i=0 ; i<na2 ; i++)
                        s+=a[i+1]*em[(na2-i+ia)%na2];
            for(i=0 ; i<nb ; i++)
                        s+=b[i]*sm[(nb2-i+ib)%nb];
            em[ia]=e;
            sm[ib]=s;
            if(++ia>=na2)
                        ia=0;
            if(++ib>=nb)
                        ib=0;
            return s;
}
```

```
// Filtrage PASSE-BAS 2e degré, pour 1 échantillon
FLOAT
Filtre::passebas_2deg_iir(FLOAT e, FLOAT fe)
{
        FLOAT   k, k1, k1c, k2;                 // variables locales pour réduire le nombre d'opérations

        k1=fe/M_PI/fc;
        k1c=k1*k1;
        k2=2*xi*k1;
        k=1+k2+k1c;
        a[0]=1./k;
        a[1]=2.*a[0];
        a[2]=a[0];
        b[0]=-2.*(1.-k1c)/k;
        b[1]=-(1-k2+k1c)/k;
        return iir(e);
}
```

II.2) Synthèse de filtres

La synthèse d'un filtre numérique consiste à déterminer ses coefficients, en fonction de la fonction qu'il doit réaliser. Il existe de nombreuses méthodes pour déterminer ces coefficients.

Dans le cas de coefficients qui permettent de réaliser des filtres numériques ayant des caractéristiques communes avec des filtres analogiques, on parle de modélisation.

Pour modéliser des filtres analogiques, on utilise la transformée en Z et la transformée bilinéaire. Nous nous limiterons dans ce paragraphe à la synthèse de filtres analogiques.

II.2.1) Lien entre fonction de transfert en z et équation de récurrence

La forme générale de la fonction de transfert en z d'un filtre numérique est :

$$H(z)=\frac{S(z)}{E(z)}=\frac{\sum_{l=0}^{L} a_l z^{-l}}{1+\sum_{k=1}^{K} b_k . z^{-k}} \qquad (1)$$

E(z) et S(z) représentent respectivement les transformées en z des échantillons d'entrée e(n), avec $n=kT_e$, et de sortie s(n) courants :

$$E(z)=Z\{e(n)\}$$
$$S(z)=Z\{s(n)\}$$

Les propriétés de la transformée en z utilisées pour passer de la fonction de transfert en z à l'équation de récurrence (et réciproquement), sont celle du **retard temporel** et celle de la **linéarité**.

Si L est l'ordre du numérateur, K l'ordre du dénominateur, l'ordre du filtre est max(L,K).

$$(1) \leftrightarrow S(z).\left[1+b_1 z^{-1}+b_2 z^{-2}+...+b_k z^{-k}\right]=E(z).\left[a_0+a_1 z^{-1}+a_2 z^{-2}+...+a_L z^{-L}\right]$$

$$s(n)+b_1 s(n-1)+b_2 s(n-2)+...+b_K s(n-K)=a_0 e(n)+a_1 e(n-1)+a_2 e(n-2)+...+a_L e(n-L)$$
$$s(n)=a_0 e(n)+a_1 e(n-1)+a_2 e(n-2)+...+a_L e(n-L)-b_1 s(n-1)-b_2 s(n-2)-...-b_K s(n-K)$$

II.2.2) Forme générale de la fonction de transfert de filtre du 1er et 2nd ordres

1er ordre

$$H(z)=\frac{a_0+a_1 z^{-1}}{1+b z^{-1}}$$

ou

$$H(z) = a'_0 \frac{1 + a'_1 z^{-1}}{1 + bz^{-1}} \text{ avec } a'_0 = a_0 \text{ et } a'_1 = \frac{a_1}{a_0}$$

2nd ordre

$$H(z) = \frac{a_0 + a_1 z^{-1} + a_2 z^{-2}}{1 + b_1 z^{-1} + b_2 z^{-2}}$$

ou

$$H(z) = a'_0 \frac{1 + a'_1 z^{-1} + a'_2 z^{-2}}{1 + b_1 z^{-1} + b_2 z^{-2}} \text{ avec } a'_0 = a_0 \text{ ; } a'_1 = \frac{a_1}{a_0} \text{ ; } a'_2 = \frac{a_2}{a_0}$$

Stabilité

Un filtre numérique est stable si les pôles de sa fonction de transfert (racines, réelles ou complexes, du dénominateur) sont à l'intérieur du cercle unité, c'est à dire ont un module inférieur à 1. Ceci mène aux conditions suivantes sur les coefficients des filtres :

- dans le cas du 1er ordre :

$$|b| < 1$$

- dans la fonction de transfert du 2e ordre :

$$0 \le b_2 < 1$$
$$|b_1| < 1 + b_2$$

II.2.3) Transformée bilinéaire

Elle est utilisée dans la synthèse de filtre analogique. Elle permet d'obtenir le filtre numérique correspondant à un filtre analogique, en passant de la variable de Laplace p (domaine analogique) à la variable d'un système échantillonné z

Transformée simplifiée

Dans le cas où $f << f_e$, la transformation bilinéaire s'exprime par :

$$p = \frac{2}{T_e} . \frac{z - 1}{z + 1}$$

ou, ce qui est équivalent :

$$p = \frac{2}{T_e} . \frac{1 - z^{-1}}{1 + z^{-1}}$$

T_e est la période d'échantillonnage.

Pour synthétiser un filtre analogique, on part donc de la fonction de transfert complexe $H(j\omega)$, à la fonction de transfert de Laplace $H(p)$ (variable $p=j\omega$), puis à la fonction de transfert échantillonnée $H(z)$ (la variable z étant obtenue à l'aide de la transformée bilinéaire).

"Vraie" transformée

La transformée bilinéaire définie ci-dessus s'accompagne d'une distorsion de la réponse fréquentielle du filtre numérique obtenue, pour les fréquences proches de f_e. Une correction est donc nécessaire pour que le filtre numérique modélise le filtre analogique pour toutes les fréquences. L'expression de la véritable transformée est :

$$p = \frac{\omega}{\mathrm{tg}\frac{\omega T_e}{2}} . \frac{z-1}{z+1}$$

Pour déterminer cette relation, on part de la relation de départ entre p et z :

$$p = K . \frac{z-1}{z+1} \quad (1)$$

Pour déterminer K, on remplace z par son expression en fonction de la période d'échantillonnage $z = e^{j\omega_n T_e}$ où ω_n est la pulsation numérique, puis p par $j\omega_a$ où ω_a est la pulsation analogique. On simplifie l'expression, puis on impose que la pulsation de coupure analogique corresponde à la pulsation de coupure numérique, en posant :

$$\omega_a = \omega_N = \omega$$

On obtient alors la vraie relation entre p et z (donnée ci-dessus).

Le cas étudié dans l'approche grossière est un cas particulier de ce cas général (on peut le vérifier en remarquant que $\mathrm{tg}(x) \approx x$ si $x << \pi/2$).

II.2.4) Synthèse de filtres analogique

Pour passer de la fonction de transfert analogique à la fonction de transfert numérique, on remplace jω par p et on effectue le changement de variable directement issu de la transformée bilinéaire (simplifiée) :

$$\frac{p}{\omega_c} = \frac{f_e}{\pi f_c} . \frac{z-1}{z+1}$$

On identifie la fonction de transfert en z obtenue avec la forme générale des fonctions de transfert en z (voir paragraphe sur la forme générale des fonctions de transfert en z, plus haut), pour obtenir les coefficients du filtre analogique en fonction des paramètres du filtre analogique.

Voici les relations obtenues par ces calculs, dans le cas de filtres du 1er et du 2nd ordre :

Filtre passe-bas du 1er ordre

$$H(j\omega) = \frac{1}{1 + j\frac{\omega}{\omega_c}} \quad \rightarrow \quad a_0 = \frac{1}{1+k} \; ; \; a_1 = a_0 \; ; \; b = \frac{1-k}{1+k} \text{ avec } k = \frac{f_e}{\pi f_c}$$

Filtre passe-haut du 1er ordre

$$H(j\omega) = \frac{j\frac{\omega}{\omega_c}}{1 + j\frac{\omega}{\omega_c}} \quad \rightarrow \quad a_0 = \frac{k}{1+k} \; ; \; a_1 = -a_0 \; ; \; b = \frac{1-k}{1+k} \text{ avec } k = \frac{f_e}{\pi f_c}$$

Filtre passe-bas du 2e ordre

$$T(j\omega) = \frac{1}{1 + 2\xi j\frac{\omega}{\omega_0} + \left(j\frac{\omega}{\omega_0}\right)^2} \quad \rightarrow \quad a_0 = \frac{1}{k_1} \; ; \; a_1 = \frac{2}{k_1} = 2a_0 \; ; \; a_2 = \frac{1}{k_1} = a_0 \; ;$$

$$b_1 = \frac{2}{k_1}\left(1-k^2\right)\ ;\ b_2 = \frac{1}{k_1}\left(1-2\xi k+k^2\right) \text{ avec } k_1 = 1+2\xi k+k^2 \text{ et } k = \frac{f_e}{\pi f_c}$$

Filtre passe-bande du 2e ordre

$$T(j\omega) = \frac{2\xi j\frac{\omega}{\omega_0}}{1+2\xi j\frac{\omega}{\omega_0}+\left(j\frac{\omega}{\omega_0}\right)^2} \qquad \rightarrow \qquad a_0 = \frac{2}{k_1}\xi k\ ;\ a_1 = 0\ ;\ a_2 = -a_0\ ;$$

k, k_1, b_1 et b_2 sont identiques au cas du passe-bas.

Filtre passe-haut du 2e ordre

$$H(j\omega) = \frac{\left(j\frac{\omega}{\omega_c}\right)^2}{1+2\xi j\frac{\omega}{\omega_c}+\left(j\frac{\omega}{\omega_c}\right)^2} \quad \rightarrow \quad a_0 = \frac{k^2}{k_1}\ ;\ a_1 = -2a_0\ ;\ a_2 = a_0\ ;\ b_1 = \frac{2}{k_1}\left(1-k^2\right)$$

k, k_1, b_1 et b_2 sont identiques au cas du passe-bas.

Exemple : application numérique dans le cas du passe-bas du 2e ordre

f_c=500Hz ; ξ=0,1 ; f_e=44100Hz

$$k = 1+\frac{2\xi f_e}{\pi f_c}+\left(\frac{f_e}{\pi f_c}\right)^2 = 1+\frac{2\times 0{,}1\times 44100}{\pi\times 500}+\left(\frac{44100}{\pi\times 500}\right)^2 = 1+5{,}6115+788{,}2 = 765{,}815$$

a_0=0,00126 ; a_1=2 ; a_2=1

$$b_1 = \frac{2}{k}\left(1-\left(\frac{f_e}{\pi f_c}\right)^2\right) = \frac{2}{k}\left(1-\left(\frac{44100}{\pi\times 500}\right)^2\right) = -1{,}9808428$$

$$b_2 = \frac{1}{k}\left(1-2\times 0{,}1\times\frac{44100}{\pi\times 500}+\left(\frac{44100}{\pi\times 500}\right)^2\right) = \frac{1}{k}\left(1-5{,}615+788{,}2\right) = \frac{1}{k}\left(783{,}585\right) = 0{,}985871$$

III) Transformée de Fourier Rapide

La Transformée de Fourier Rapide (TFR ou FFT pour Fast Fourier Transform) est un algorithme rapide de calcul de la transformée de Fourier discrète (TFD). Il est basé sur une simplification des calculs à effectuer, permise par les propriétés de l'exponentielle. Il permet de calculer la TFD en temps réel, c'est à dire dans une durée inférieure à la période d'échantillonnage.

Ce paragraphe décrit d'abord les principales propriétés de la TFD et de la FFT, d'abord vues comme des "boîtes noires", c'est à dire du point de vue de leurs entrées/sorties, puis leurs principales propriétés internes. Ensuite sont étudiées une implémentation en C et une en assembleur DSP56303.

III.1) Principe

III.1.1) Approche "boîte noire"

En continu, la TF est définie par :

$$X(f) = \int_{-\infty}^{\infty} x(t) e^{-j2\pi ft} dt$$

x(t) est un signal temporel, X(f) une fonction de la fréquence.

La transformée de Fourier discrète (TFD) est définie par :

$$X(k) = \frac{1}{N} \sum_{n=0}^{N-1} w_N^{kn} x(n) \ ; k=0, 1, \ldots, N-1$$

avec

$$W_N = \exp\left(-j\frac{2\pi}{N} \right)$$

où x(n) sont les échantillons du signal à traiter (n représente un indice temporel, on pourrait écrire $x(nT_e)$), N est le nombre d'échantillons du signal. W_N est appelé facteur de phase (twiddle factor).

Principales propriétés de la TFD vue comme une boîte noire

- Le nombre d'entrées et le nombre de sorties sont les mêmes.
- Les entrées sont des échantillons de signal ; les indices des entrées représentent donc du temps discret. Les indices de sortie représentent des fréquences.
- Pour N entrées la TFD donne N sorties, dont seules les N/2 premières peuvent être conservées dans la suite du traitement puisque les N/2 suivantes représentent une symétrie par rapport au point d'indice N/2-1 (ils représentent des fréquences négatives).
- Chaque sortie représente une fréquence ; l'ensemble des sorties constitue un spectre de fréquence (amplitude et phase).
- La fréquence de la sortie i est égale à :

  $$f_i = i \times \frac{f_e}{N} \qquad i=0,\ldots,N-1 \qquad (1)$$

  où f_e est la fréquence d'échantillonnage.
- Dans le cas où les entrées de la TFD ne sont pas constituées par les échantillons d'un signal réel mais les éléments d'un tableau définis de manière artificielle, la sortie d'indice i+1 représente le nombre de périodes présentes dans les N échantillons du signal d'entrée. En effet, soit f_e la fréquence d'échantillonnage ; elle représente le nombre d'échantillons de signal prélevés en 1 seconde. Supposons que l'on ait $N=f_e$ et 1 période de sinus dans ces N points ; cela représente une fréquence du signal de 1 Hz, donc seule la 2e sortie de la FFT est non-nulle. Considérons les autres exemples suivants :
  - f_e=512, N=512 et 2 périodes de sinus dans ces N points : la fréquence du signal est 2 Hz donc seule la 3e sortie de la FFT est non nulle ;
  - f_e=512, N=256 et 1 période de sinus dans ces N points : la fréquence du signal est 2 Hz donc seule la 2e sortie de la FFT est non nulle, car d'après la relation (1) :

    $$i = f_i \times \frac{N}{f_e} = 2 \times \frac{256}{512} = 1$$
- Les entrées et les sorties de la TFD sont des nombres complexes. Concrètement :
  - le signal d'entrée constitue la partie réelle du complexe, et la partie imaginaire est prise égale à 0 ;
  - on ne s'intéresse souvent qu'au module de la sortie.

Principales propriétés de la FFT vue comme une boîte noire

- La FFT ne s'applique qu'à des nombres N d'échantillons qui sont des puissances de 2 (64, 128, 256, 512, 1024, 2048,…).
- Les sorties de la FFT sont cryptées, c'est à dire pas dans l'ordre naturel (par rapport à l'ordre des sorties de la TFD). Dans le cas de l'utilisation du DSP 56303, il existe le mode d'adressage en bits inversés, qui permet de retrouver l'ordre naturel.

III.1.2) Approche détaillée

Le principe de la FFT est de décomposer la TFD à effectuer en TFD élémentaires successives et en parallèle.

Principe de calcul

On peut démontrer que de nombreuses simplification de calcul peuvent être apportées à l'expression de la TFD.

Rappel :

$$X(k) = \frac{1}{N}\sum_{n=0}^{N-1} w_N^{kn} x(n) \ ; k=0, 1, \ldots, N-1 \quad \text{avec } W_N^{nk} = \exp\left(-j\frac{2\pi nk}{N}\right)$$

On développe cette expression, en ommettant le facteur 1/N non-obligatoire, pour simplifier ; d'abord selon n :

$$X(k) = w_N^{0\times k}x(0) + w_N^{1\times k}x(1) + \ldots + w_N^{(N-1)\times k}x(N-1), \ k=0, 1, \ldots, N-1$$

puis selon k :

$$X(0) = w_N^{0\times 0}x(0) + w_N^{1\times 0}x(1) + \ldots + w_N^{(N-1)\times 0}x(N-1)$$
$$X(1) = w_N^{0\times 1}x(0) + w_N^{1\times 1}x(1) + \ldots + w_N^{(N-1)\times 1}x(N-1)$$
$$\ldots\ldots\ldots\ldots$$
$$X(N-1) = w_N^{0\times (N-1)}x(0) + w_N^{1\times (N-1)}x(1) + \ldots + w_N^{(N-1)\times (N-1)}x(N-1)$$

et en notation matricielle :

$$[X] = [W][X]$$

$[X]$ étant de dimension N×1 et $[W]$ N×N

Le calcul de cette équation nécessite donc N^2 multiplications complexes (et N^2-N additions). On peut démontrer que l'on a les propriétés suivantes :

$$w_N^{2k} = w_{N/2}^{k}, \quad W_{\frac{N}{2}}^{\frac{N}{2}n} = 1, \quad W_N^{k+N/2} = -W_N^k$$

utilisant la définition de l'exponentielle complexe :

$$\exp(j\alpha) = \cos(\alpha) + j\sin(\alpha) \quad \text{et} \quad \exp(-j\alpha) = \cos(\alpha) - j\sin(\alpha)$$

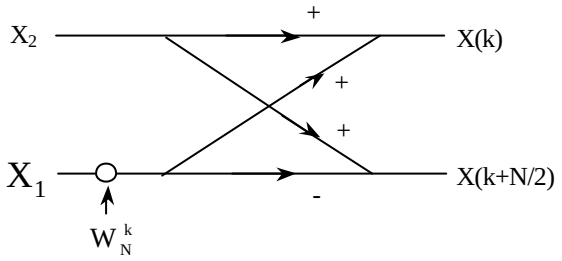
Ces propriétés permettent d'aboutir à :

$$X(k) = X_1(k) + W_N^k X_2(k)$$

et

$$X(k+\frac{N}{2}) = X_1(k) - W_N^k X_2(k)$$

avec k=0, 1, …, N/2-1, où $X_1(k)$ est la TFD des N/2 points d'indice pair et $X_2(k)$ la TFD des N/2 points d'indice impair. On peut représenter ce calcul par une structure dite "en papillon" ("butterfly") :



$X_1(k)$ et $W_N^k X_2(k)$ sont utilisés par X(k) et X(k+N/2), mais sont calculés une seule fois.

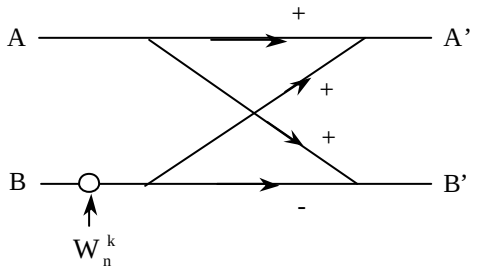
Ainsi, si le nombre d'échantillons est une puissance de 2 : $N=2^P$, on peut réitérer P fois cette décomposition en temps sur chacune des transformées précédentes jusqu'à arriver à des décompositions portant sur 2 éléments.

Le nombre de multiplications nécessaires passe de N^2 à $\log_2(N)$ (la puissance de 2 nécessaire pour obtenir N), et le nombre d'additions de N^2-N à $N\times\log_2(N)$. Par exemple, pour une TFD de 1024 points, on passe respectivement de $1{,}05\times10^6$ (environ) à 20480 et de $1{,}05\times10^6$ à 30720.

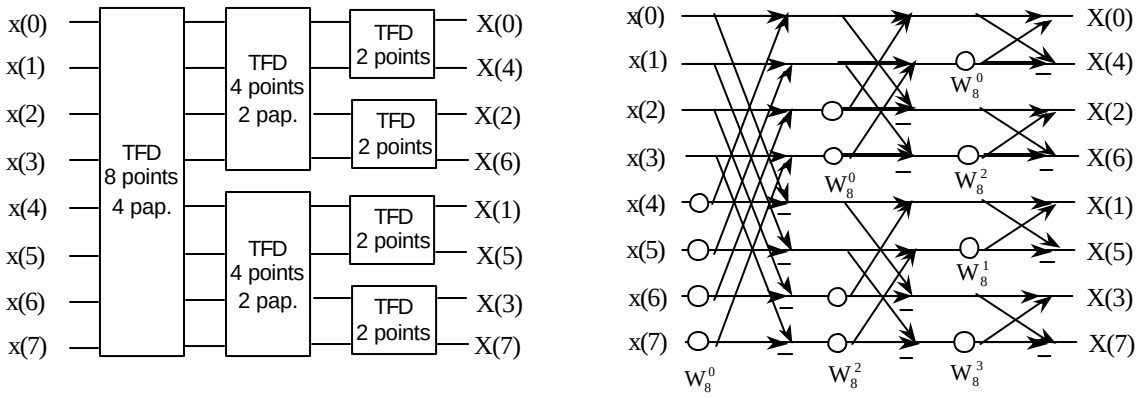
Cette version de l'algorithme est qualifiée de "Décimation en Temps" (ou DIT pour Decimation In Time) car la décomposition entre indices pairs et impairs a porté sur les indices temporels. On aurait pu également réaliser une décimation en fréquence.

Schémas fonctionnel et structurel de la FFT

Reprenons la structure de base de la FFT, le papillon, qui permet d'obtenir 2 éléments de sortie en fonction de 2 éléments d'entrée, en n'utilisant qu'un seul facteur de phase.



La structure de l'algorithme de FFT complet pour un cas simple de N=8 échantillons (et donc 8 points de sortie) est représentée ci-dessous, sous forme de schéma-blocs et avec mise en évidence des papillons.



Remarque : Cette forme est dite avec ré-arrangement des sorties. On aurait pu inverser la structure en shémas-blocs, en changeant l'ordre des échantillons d'entrée pour tomber sur des points de sortie dans l'ordre naturel.

Soit N le nombre de points sur lesquels la FFT est calculée, l'algorithme de calcul de la FFT est composé de 3 boucles imbriquées :

- La 1ère sur les passes (étapes de FFT intermédiaires)
  Le nombre de passes est $P=\log_2(N)/\log_2(2)$ car P est la puissance de 2 du nombre de points (exemple : P=3 pour N=8)
- La 2e sur les groupes
  1 groupe pour la 1ère passe, multiplié par 2 à chaque passe ; soit p le numéro de la passe courante, le nombre de groupes est 2^{p-1}
- La 3e sur les papillons
  N/2 papillons pour la 1ère passe, division par 2 à chaque passe ; le nombre de papillons est $N/2^p$

Exemple pour N=8 :

Passe 1 : 1 groupe de 4 papillons
Passe 2 : 2 groupes de 2 papillons
Passe 3 : 4 groupes de 1 papillon

Algorithme

La partie de l'algorithme concernant le calcul des sorties est la suivante :

```
Initialisation des variables :
   - Nombre de papillons par groupe : ppg=N/2
   - Nombre de groupes par passe : gpp=1
Boucle sur les passes : P itérations (P=log(N)/log(2))
      Boucle sur les groupes : gpp itérations
            Boucle sur les papillons : ppg itérations
                  Calcul du papillon courant
            Fin boucle papillons
      Fin boucle groupes
      ppg=ppg/2
      gpp=gpp*2
Fin boucle passes
```

Comme indiqué précédemment, les sorties de la FFT sont dans un ordre dit "crypté". L'algorithme complet comporte donc 2 parties :

- Calcul des sorties de la FFT
- Remise des sorties dans l'ordre naturel

Décryptage des sorties

Les sorties de la FFT sont cryptées, c'est à dire qu'elles ne sont pas dans le "bon" ordre, l'ordre naturel (1, 2, 3, etc). Il faut donc les remettre dans le bon ordre avant leur utilisation dans la suite du traitement.

Voici, pour quelques valeurs de N, l'ordre des sorties obtenu :

- 4 points : 0, 2, 1, 3
- 8 points : 0, 4, 2, 6, 1, 5, 3, 7
- 16 points : 0, 8, 2, 14, 4, 10, 6, 8, 1, 15

Si l'on écrit les indices (= les adresses) des points codés en binaire, on peut remarquer une certaine logique de codage, par exemple dans le cas N=8 (codage sur 3 bits donc) :

000, 100, 010, 110, 001, 101, 011, 111

et ce que l'on souhaiterait avoir (ordre naturel) :

000, 001, 010, 011, 100, 101, 110, 111

On s'aperçoit qu'il suffit de "renverser" les bits, c'est à dire de leur appliquer une symétrie centrale.

On peut également d'abord crypter les entrées (c'est à dire les mettre dans l'ordre en bits inversés, puis appliquer l'algorithme ; les sorties sont alors dans l'ordre correct). Dans le DSP56303, l'adressage en bit inversés (bit reverse) est implémenté.

III.2) Implémentation en C

Exemple d'algorithme de décryptage

Un nombre à n bits peut être exprimé par :

$$n = \sum_{m=0}^{b-1} b_m 2^m$$

Exemple :

$$4=b_0 2^0 + b_1 2^1 + b_0 2^2 = 0\times 2^0 + 0\times 2^1 + 1\times 2^2$$

Nombre correspondant en codage en bits inversés :

$$n' = \sum_{m=0}^{b-1} b_m 2^{b-1-m}$$

Exemple :

$$\begin{aligned} & b_0 2^{2-0} + b_1 2^{2-1} + b_0 2^{2-2} \\ &= b_0 2^2 + b_1 2^1 + b_0 2^0 \\ &= 0\times 2^0 + 0\times 2^1 + 1\times 2^2 \\ &= 4 \end{aligned}$$

L'algorithme pourrait alors s'écrire :

r=0 (r : nombre converti en bits inversés)
pour chaque bit b_m
    on détermine s'il est égal à 1 et on additionne 2^{b-1-m} à r

Et de manière plus détaillée :

```
r=0
pour m=b-1 à 0 par pas de –1
    si n>>m=1 (*)
        r=r+2^(b-1-m)
        n=n-2^m
```

(*) on décale les bits m fois vers la droite (résultat de l'opérateur >> : décalage d'un bit vers la droite).

Un exemple d'implémentation en C pourrait donc être la fonction 'bitrev()' suivante, qui retourne le nombre r, en bits inversés, d'un nombre n. Arguments : n=nombre à traiter ; b : nombre de bits sur lesquels n est codé.

```
int bitrev(int n, int b)
{
        int m, r ;
        r=0 ;

        for(m=b-1 ; m>=0 ; m--)
                if((n>>m)==1)
                {
                        r+=two(b-1-m) ;
                        n-=two(m) ;
                }
        return r ;
}
```

avec

```
#define two(x)   (1<<(x))        //puissance de 2 de x
```

Exemple d'exécution :

n=6=110			
b-1=3-1=2			
m=2	n>>2=1	r=0+2^0=1	n=2
m=1	n>>1=1	r=1+2^1=3	n=0
m=0	n>>0=0	r=3	n=0

Pour le reste de l'implémentation, on pourra analyser le programme donné en exemple sur le site www-dsp.efrei.fr ("fourier.c" et "test_fourier.c").

IV) Analyse spectrale

L'analyse spectrale est la principale application de la transformée de Fourier (TF). Elle consiste à appliquer la TF de manière continue, à un signal arrivant en flot.

Elle nécessite que la condition de temps-réel soit respectée, à savoir

$$t_t < T_e$$

où t_t est le temps de traitement d'un échantillon et T_e la période d'échantillonnage.

Il existe 2 grandes approches : une approche séquentielle et une approche parallèle.

IV.1) Approche séquentielle

Cette approche est l'approche classique. Elle consiste à réaliser les différentes opérations de manière séquentielle :

- acquisition d'un bloc de signal
- application de la TFR
- sortie du résultat

L'inconvénient de cette méthode est qu'elle entraîne une perte d'échantillons pendant le traitement.

IV.2) Approche parallèle

L'avantage de cette approche est de ne pas comporter de perte d'échantillons. Elle nécessite d'utiliser 2 zones de stockage de données M1 et M2 : pendant que la TFR est appliquée aux

données de l'une des 2 zones, les échantillons s'accumulent dans l'autre (au rythme de la période d'échantillonnage T_e).

Cette approche nécessite de gérer les interruptions déclenchées lorsqu'un nouvelle échantillon arrive (c'est à dire tous les T_e). Cette interruption interrompt le programme principal qui réalise le traitement.

IV.3) Transformée de Fourier glissante

L'inconvénient de la TFR est de n'être applicable qu'au cas des nombres d'échantillons égaux à des puissances de 2.

L'analyse spectrale est possible sans utiliser la TFR, car on peut démontrer que la TFD peut s'exprimer de manière récursive, c'est à dire que la TFD de N points du signal arrivant en flot peut s'exprimer en fonction de la TFD calculée sur les N points précédents, décalés d'un seul échantillon. La relation de récurrence est :

$$X_p(k) = \left(\frac{x(p) - x(p-N)}{N} \right) + w_N^k X_{p-1}(k)$$

où p est l'indice du nouvel échantillon et $X_{p-1}(k)$ la TFD calculée sur les N échantillons jusqu'au $(p-1)^e$.

Annexe 1 : Programme de FFT en C

```
/*-------------------------------------------------------------------------------------------
            analyse.c : Analyse frequentielle
                        B. Decoux (decoux@efrei.fr), nov. 2003
-------------------------------------------------------------------------------------------
Programme réalisé à partir de :

    Program: REALFFT.C
     Author: Philip VanBaren
       Date: 2 September 1993

 Description: These routines perform an FFT on real data.

  Note: Output is BIT-REVERSED! so you must use the BitReversed to
      get legible output, (i.e. Real_i = buffer[ BitReversed[i] ]
                               Imag_i = buffer[ BitReversed[i]+1 ] )
      Input is in normal order.

  Copyright (C) 1995  Philip VanBaren
 */

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "define.h"
#include "fourier.h"

#define SWAP(a, b) tempr=(a); (a)=(b); (b)=tempr;

UWORD  nb_pts, nb_car, nb_vec;
FLOAT  **car;
int    *BitReversed;
short  *SinTable;
int    Points = 0;

/*
 *  Initialize the Sine table and Twiddle pointers (bit-reversed pointers)
 *  for the FFT routine.
 */
/*
 *  Le fait d'utiliser des tableaux de sinus et de cosinus est plus rapide que
 *  d'utiliser les fonctions mathématiques 'sin' et 'cos'
 */
void
InitializeFFT(int fftlen)
{
  int i;
  int temp;
  int mask;

  /*
   *  FFT size is only half the number of data points
   *  The full FFT output can be reconstructed from this FFT's output.
   *  (This optimization can be made since the data is real.)
   */
  Points = fftlen;

  if((SinTable=(short *)malloc(Points*sizeof(short)))==NULL)
  {
    puts("Error allocating memory for Sine table.");
    exit(1);
  }
  if((BitReversed=(int *)malloc(Points/2*sizeof(int)))==NULL)
  {
    puts("Error allocating memory for BitReversed.");
    exit(1);
  }
 /*calcul des pointeurs pour adresser les tableaux de sinus et cosinus, en mode 'bits inversés'*/
  for(i=0;i<Points/2;i++)
  {
    temp=0;
    for(mask=Points/4;mask>0;mask >>= 1)
      temp=(temp >> 1) + (i&mask ? Points/2 : 0);
    BitReversed[i]=temp;
  }
 /*Calcul des tableaux de sinus et de cosinus*/
  for(i=0;i<Points/2;i++)
  {
    register double s,c;
    s=floor(-32768.0*sin(2*M_PI*i/(Points))+0.5);  /*arrondi à la partie entière*/
    c=floor(-32768.0*cos(2*M_PI*i/(Points))+0.5);
    if(s>32767.5) s=32767;
    if(c>32767.5) c=32767;
    SinTable[BitReversed[i]  ]=(short)s;
    SinTable[BitReversed[i]+1]=(short)c;
  }
}
/*
```

```
 *  Free up the memory allotted for Sin table and Twiddle Pointers
 */
void EndFFT()
{
        free(BitReversed);
        free(SinTable);
        Points=0;
}
/*
 *  Actual FFT routine.  Must call InitializeFFT(fftlen) first!
 */
void RealFFT(short *buffer)
{
  short *A,*B;
  short *sptr;
  short *endptr1,*endptr2;
  int *br1,*br2;
  long HRplus,HRminus,HIplus,HIminus;
  int ButterfliesPerGroup=Points/4;

  endptr1=buffer+Points;

  /*
   *  Butterfly:
   *     Ain-----Aout
   *        \ /
   *        / \
   *     Bin-----Bout
   */

  while(ButterfliesPerGroup>0)
  {
    A=buffer;
    B=buffer+ButterfliesPerGroup*2;
    sptr=SinTable;

    while(A<endptr1)
    {
      register short sin=*sptr;
      register short cos=*(sptr+1);
      endptr2=B;
      while(A<endptr2)
      {
        long v1=((long)*B*cos + (long)*(B+1)*sin) >> 15;
        long v2=((long)*B*sin - (long)*(B+1)*cos) >> 15;
            *B=(*A+v1)>>1;
        *(A++)=*(B++)-v1;
            *B=(*A-v2)>>1;
        *(A++)=*(B++)+v2;
      }
      A=B;
      B+=ButterfliesPerGroup*2;
      sptr+=2;
    }
    ButterfliesPerGroup >>= 1;
  }
  /*
   *  Massage output to get the output for a real input sequence.
   */
  br1=BitReversed+1;
  br2=BitReversed+Points/2-1;

  while(br1<=br2)
  {
    register long temp1,temp2;
    short sin=SinTable[*br1];
    short cos=SinTable[*br1+1];
    A=buffer+*br1;
    B=buffer+*br2;
    HRplus = (HRminus = *A     - *B    ) + (*B << 1);
    HIplus = (HIminus = *(A+1) - *(B+1)) + (*(B+1) << 1);
    temp1  = ((long)sin*HRminus - (long)cos*HIplus) >> 15;
    temp2  = ((long)cos*HRminus + (long)sin*HIplus) >> 15;
    *B     = (*A     = (HRplus  + temp1) >> 1) - temp1;
    *(B+1) = (*(A+1) = (HIminus + temp2) >> 1) - HIminus;

    br1++;
    br2--;
  }
  /*
   *  Handle DC bin separately
   */
  buffer[0]+=buffer[1];
  buffer[1]=0;
}


/*
        signal:     buffer contenant le signal à traiter
        coef :              coefficients de Fourier retournés par la transformée
        nb_pts :    nombre de points de la TF
```

```
*/
VOID
fft(FLOAT *signal, FLOAT *coef, UWORD nb_pts)
{
	ULONG	i;
	LONG	re, im;
	int		*bri;			/*bit reverse index*/
	WORD	*bufW;

	bufW=(WORD*)calloc(nb_pts, sizeof(WORD));
	for(i=0 ; i<nb_pts ; i++)
		*(bufW+i)=(WORD)(*(signal+i));

	InitializeFFT(nb_pts);
	RealFFT(bufW);
	bri=BitReversed;	/* 'bri' pointe sur le tableau des adresses en bits inversés */
	for(i=0 ; i<nb_pts/2 ; i++)
	{
		re=*(bufW+(*bri));
		im=*(bufW+(*bri)+1);
		*(coef+i)=sqrt(re*re+im*im);	/*module*/
		bri++;
	}
	free(bufW);
}

/*-------------------------------------------------------------------------------------
	test_fourier.c

	Génération d'un signal de test (ex : somme de sinus)
	sur 'nb_pts' et FFT de ce signal
-------------------------------------------------------------------------------------*/
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <fcntl.h>
#include "define.h"
#include "fourier.h"

VOID
main()
{
	FLOAT	*buf1, *buf2;
	int		nb_pts, i;

	nb_pts=64;
	buf1=(FLOAT*)calloc(nb_pts, sizeof(FLOAT));	/*buffer float pour le signal*/
	buf2=(FLOAT*)calloc(nb_pts, sizeof(FLOAT));	/*buffer float pour la TF*/

	for(i=0 ; i<nb_pts ; i++)
		*(buf1+i)=10000.*(sin(i*2.*M_PI/nb_pts*8)+0.5*sin(i*2.*M_PI/nb_pts*20));
	for(i=0 ; i<nb_pts ; i++)
		printf("%.2f ", *(buf1+i));
	printf("\n");

	fourier(bufF, (ULONG)nb_pts, 1);
	fft(buf1, buf2, nb_pts);
	for(i=0 ; i<nb_pts ; i++)
		printf("%.2f ", *(buf2+i));
	printf("\n");
	free(buf1);
	free(buf2);
}
```