Filtrage de signaux numériques

J.-M Friedt, 9 février 2015

Objectif de ce TP :
— exploiter les fonctions de traitement du signal de GNU/Octave pour concevoir un filtre
— implémenter ce filtre dans un microcontrôleur avec des calculs sur des entiers uniquement
— valider expérimentalement le fonctionnement du filtre

1 Conception d'un filtre de réponse finie (FIR)

Le sujet de ce TP n'est pas de développer la théorie des filtres numériques mais uniquement de rappeler quelques uns des concepts utiles en pratique :

— un filtre de réponse impulsionnelle finie [1] (Finite Impulse Response, FIR) b_m est caractérisé par une sortie y_n à un instant donné n qui ne dépend que des valeurs actuelle et passées de la mesure x_m, $m \leq n$:

$$y_n = \sum_{k=0..m} b_k x_{n-k}$$

L'application du filtre s'obtient pas convolution des coefficients du filtre et des mesures.

— un filtre de réponse impulsionnelle infinie (IIR) a une sortie qui dépend non seulement des mesures mais aussi des valeurs passées du filtre.

— GNU/Octave (et Matlab) proposent des outils de synthèse des filtres FIR et IIR. Par soucis de stabilité numérique et simplicité d'implémentation, nous ne nous intéresserons qu'aux FIR. Dans ce cas, les coefficient de récursion sur les valeurs passées du filtre (nommées en général a_m dans la littérature) seront égaux à $a = 1$.

Le calcul d'un filtre dont la réponse spectrale ressemble à une forme imposée par l'utilisateur (suivant un critère des moindres carrés) est obtenu par la fonction `firls()`. Cette fonction prend en argument l'ordre du filtre (nombre de coefficients), la liste des fréquences définissant le gabarit du filtre et les magnitudes associées. Elle renvoie la liste des coefficients `b`.

L'application d'un FIR de coefficients `b` (avec `a=1`) ou de façon plus générale d'un IIR de coefficients `a` et `b` sur un vecteur de données expérimentales `x` s'obtient par `filter(b,a,x);`

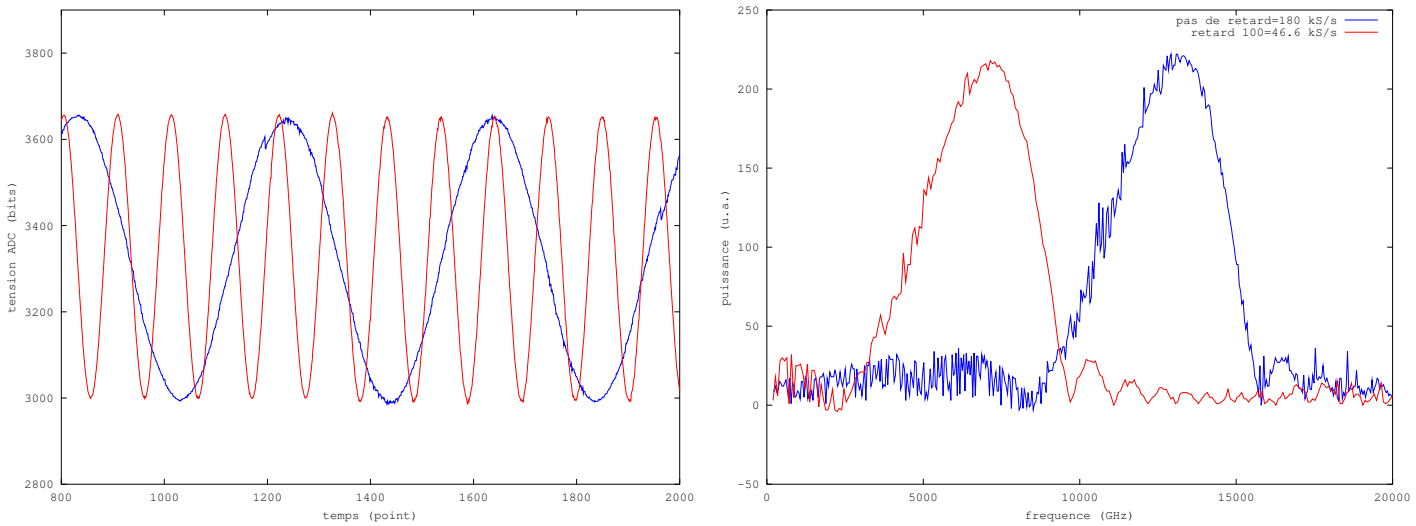


FIGURE 1 – Gauche : le même signal (sinusoïde à 440 Hz) échantilloné à deux fréquences différentes. Droite : les mêmes coefficients de filtres appliqués à un *chirp* balayant de 200 à 20000 Hz mais pour des fréquences d'échantillonage différentes.

Tout filtre se définit en terme de fréquence normalisée : la fréquence 1 correspond à la demi-fréquence d'échantillonnage (fréquence de Nyquist) lors de la mesure des données.

Exercices :

1. en supposant que nous échantillonnons les données sur un convertisseur analogique-numérique à 16000 Hz, identifier les coefficients d'un filtre passe-bande entre 700 et 900 Hz. Combien faut-il de coefficients pour obtenir un résultat satisfaisant ?
2. Quelle est la conséquence de choisir trop de coefficients ? Quelle est la conséquence de choisir trop peu de coefficients ?
3. ayant obtenu la réponse impulsionnelle du filtre, quelle est sa réponse spectrale ?
4. valider sur des données (boucle sur les fréquences) simulées la réponse spectrale du filtre

Au lieu de boucler sur des fréquences et observer la réponse du filtre appliqué à un signal monochromatique, le `chirp` est un signal dont la fréquence instantanée évolue avec le temps. Ainsi, la fonction
`chirp([0:1/16000:5],40,5,8000);`
génère un signal échantillonné à 16 kHz, pendant 5 secondes, balayant la gamme de fréquences allant de 40 Hz à 8 kHz.

Exercices :

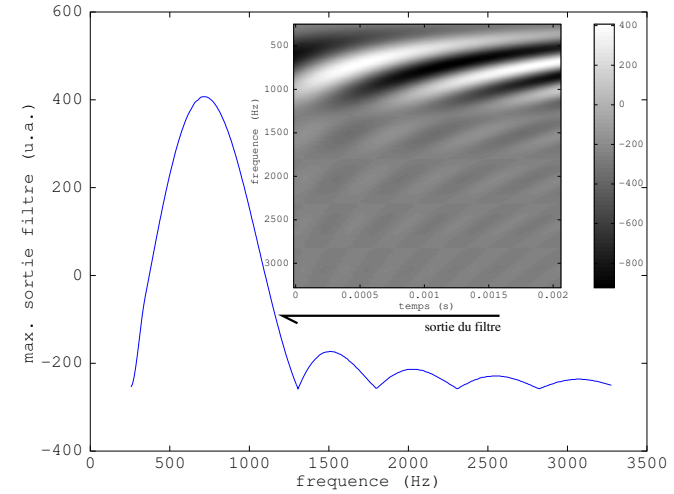
1. appliquer le filtre identifié auparavant au chirp, et visualiser le signal issu de cette transformation
2. que se passe-t-il si la fréquence de fin du chirp dépasse 8 kHz dans cet exemple ?

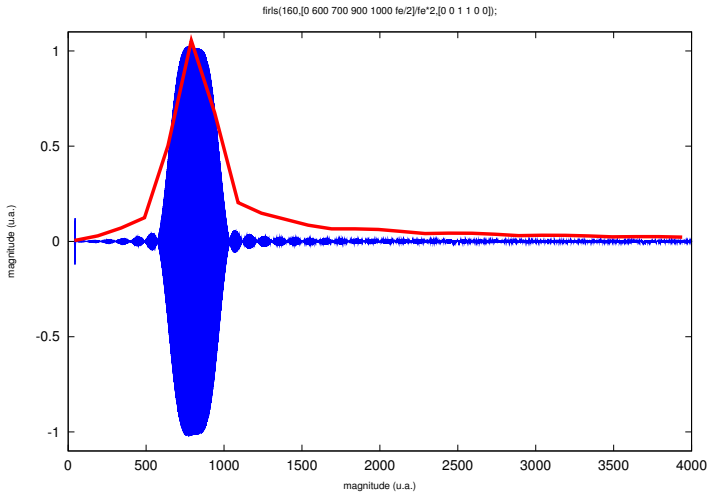
2 Application pratique du filtre

Exercices : Implémenter sur STM32

1. sachant que les acquisitions se font sur 12 bits, quelle est la plage de valeurs sur laquelle se font les mesures ?
2. comment exploiter les coefficients numériques du filtre pour un calcul sur des entiers sur 16 bits (short) ? sur 32 bits (long) ?

1. l'échantillonnage périodique de mesures analogiques acquises sur ADC1
2. l'affichage du résultat de ces mesures
3. le filtrage de ces mesures par un FIR synthétisé selon la méthode vue plus haut
4. l'affichage du résultat de ce filtrage
5. l'affichage de l'amplitude du signal résultant du filtrage
6. appliquer ce programme à une mesure sur un chirp généré par la carte son du PC au moyen de `audacity`
7. comparer l'expérience et la simulation.






```
fe=16000;
ffin=4000;
fdeb=40;

b=firls(160,[0 600 700 900 1000 fe/2]/fe*2,[0 0 1 1 0 0]);
x=chirp([0:1/fe:5],fdeb,5,ffin);
f=linspace(fdeb,ffin,length(x));
plot(f,filter(b,1,x));

freq=[fdeb:150:ffin];
k=1;
for f=freq
  x=sin(2*pi*f*[0:1/fe:1]);
  y=filter(b,1,x);
  sortie(k)=max(y);
  k=k+1;
end
hold on;plot(freq,sortie,'r')
xlabel('frequence (Hz)')
ylabel('magnitude (u.a.)')
```

En haut à gauche : mesure expérimentale de la réponse d'un FIR passe bande conçu pour être passant entre 700 et 900 Hz. La mesure est effectuée au moyen d'un chirp de 40 à 4000 Hz en 30 secondes généré par une carte son d'ordinateur. En bas à gauche : modélisation de la réponse d'un filtre passe bande entre 700 et 900 Hz, en bleu par filtrage d'un chirp, en rouge par calcul de l'effet du filtre sur quelques signaux monochromatiques. Ci-dessus : programme GNU/Octave de la simulation.

Exemple de programme et résultats :

```
// #define pc
#define debug

#ifndef pc
//#include <stm32f1/stm32f10x.h>      // definition uint32_t
#include <libopencm3/cm3/common.h> // BEGIN_DECL
```

```
#include <libopencm3/stm32/f1/rcc.h>
#include <libopencm3/stm32/f1/flash.h>
#include <libopencm3/stm32/f1/gpio.h>
#include <libopencm3/stm32/f1/adc.h>
#include <libopencm3/stm32/usart.h>
#include <libopencm3/stm32/timer.h>
#include <libopencm3/cm3/nvic.h>
#else
#include<stdio.h>
#include<stdlib.h>
#include<math.h>
#endif

#define NB 128
#define NBFIL 61

unsigned short temperature[NB] ;
volatile int jmf_index=0;

long filtre(unsigned short *,long *,long ,long ,long *);

#ifndef pc
static void usart_setup(void)
{
        rcc_peripheral_enable_clock(&RCC_APB2ENR, RCC_APB2ENR_USART1EN);

        gpio_set_mode(GPIOA, GPIO_MODE_OUTPUT_50_MHZ,
                GPIO_CNF_OUTPUT_ALTFN_PUSHPULL, GPIO_USART1_TX);
        usart_set_baudrate(USART1, 115200);
        usart_set_databits(USART1, 8);
        usart_set_stopbits(USART1, USART_STOPBITS_1);
        usart_set_mode(USART1, USART_MODE_TX_RX);
        usart_set_parity(USART1, USART_PARITY_NONE);
        usart_set_flow_control(USART1, USART_FLOWCONTROL_NONE);
        usart_enable(USART1);
}

static void gpio_setup(void)
{ rcc_peripheral_enable_clock(&RCC_APB2ENR, RCC_APB2ENR_IOPCEN);
  gpio_set_mode(GPIOA, GPIO_MODE_INPUT, GPIO_CNF_INPUT_ANALOG, GPIO0);
  gpio_set_mode(GPIOA, GPIO_MODE_INPUT, GPIO_CNF_INPUT_ANALOG, GPIO1);
  gpio_set_mode(GPIOC, GPIO_MODE_OUTPUT_2_MHZ, GPIO_CNF_OUTPUT_PUSHPULL, GPIO8|GPIO9);
}

static void timer_setup(void)
{ uint32_t timer;
  volatile uint32_t *rcc_apbenr;
  uint32_t rcc_apb;

  timer = TIM2;
  rcc_apbenr = &RCC_APB1ENR;
  rcc_apb = RCC_APB1ENR_TIM2EN;

  rcc_peripheral_enable_clock(rcc_apbenr, rcc_apb);
  timer_reset(timer);
  timer_set_mode(timer, TIM_CR1_CKD_CK_INT, TIM_CR1_CMS_EDGE, TIM_CR1_DIR_UP);
  timer_set_period(timer, 0xF*5);
  timer_set_prescaler(timer, 0x8);
  timer_set_clock_division(timer, 0x0);
  timer_set_master_mode(timer, TIM_CR2_MMS_UPDATE); // Generate TRGO on every update
  timer_enable_counter(timer);
}

static void irq_setup(void)
{ nvic_set_priority(NVIC_ADC1_2_IRQ, 0);
  nvic_enable_irq(NVIC_ADC1_2_IRQ);
}

static void adc_setup(void)
{ int i;
  rcc_peripheral_enable_clock(&RCC_APB2ENR, RCC_APB2ENR_ADC1EN);
  adc_off(ADC1);
  adc_enable_scan_mode(ADC1);
  adc_set_single_conversion_mode(ADC1);
  adc_enable_external_trigger_injected(ADC1,ADC_CR2_JEXTSEL_TIM2_TRGO); // start with TIM2 TRGO
  adc_enable_eoc_interrupt_injected(ADC1);                              // Generate the ADC1_2_IRQ
  adc_set_right_aligned(ADC1);
//adc_enable_temperature_sensor(ADC1);
```

```
  adc_set_sample_time_on_all_channels(ADC1, ADC_SMPR_SMP_28DOT5CYC);
  adc_power_on(ADC1);

  for (i = 0; i < 800000; i++) __asm__("nop");
  adc_reset_calibration(ADC1);
  while ((ADC_CR2(ADC1) & ADC_CR2_RSTCAL) != 0);
  adc_calibration(ADC1);
  while ((ADC_CR2(ADC1) & ADC_CR2_CAL) != 0);
}

void adc1_2_isr(void)
{ ADC_SR(ADC1) &= ~ADC_SR_JEOC;
  if (jmf_index<NB)
     {temperature[jmf_index]=adc_read_injected(ADC1,1);jmf_index++;}
}
#else
  #define fe 32768.
  #define USART1 0
  volatile float temps=0.;
  volatile float freq=fe/(float)NB; // garantit 1 periode

  void adc1_2_isr(void)
    {temperature[jmf_index]=(int)(1200.+1000.*cos(2*M_PI*freq*temps));jmf_index++;
     temps+=(1/fe);
    }
#endif

static void my_usart_print_int(unsigned int usart, int value)
{
#ifndef pc
  int8_t i;
  uint8_t nr_digits = 0;
  char buffer[25];
  if (value < 0) {usart_send_blocking(usart, '-');value=-value;}
  while (value > 0) {buffer[nr_digits++] = "0123456789"[value % 10];value /= 10;}
  for (i = (nr_digits - 1); i >= 0; i--) {usart_send_blocking(usart, buffer[i]);}
  usart_send_blocking(usart, '\r');usart_send_blocking(usart, '\n');
#else
  printf("%d ",value);
#endif
}

long filtre(unsigned short *in,long *fil,long nin,long nfil,long *sortie)
{unsigned short k,j;long s=0,max=-2500;
 if (nin>nfil)
  {for (k=0;k<nin-nfil;k++)
    {s=0;
     for (j=0;j<nfil;j++)
         s+=((((long)fil[j]*(long)in[nfil-1+k-j])))/256;
     s=s/256;
     sortie[k]=s;
     if (max<s) {max=s;}
    }
  }
  return((long)max);
}

int main (void)
{
 unsigned short k=0;
 long sortie[NB],max;

 unsigned char channel_array[16];

// a=(round(firls(60,[0 500 700 900 1000 32768/16]/32768*16,[0 0 1 1 0 0])*65536)')
/*long fil200[NBFIL]={-284,-343,193,700,362,-457,-665,-167,108,8,390,1092,\
     557,-1361,-2181,-294,2040,1797,-110,-628,-77,-1315,-3220,-821,\
    5539,7177,-1131,-10626,-8081,5196,12938,5196,-8081,-10626,-1131,7177,\
    5539,-821,-3220,-1315,-77,-628,-110,1797,2040,-294,-2181,-1361,\
    557,1092,390,8,108,-167,-665,-457,362,700,193,-343,-284};
*/

// a=(round(firls(60,[0 500 700 900 1000 32768/2]/32768*2,[0 0 1 1 0 0])*65536)')
long fil200[NBFIL]={-384,-543,-695,-836,-964,-1074,-1164,-1229,-1270,-1283, \
 -1268,-1224,-1153,-1054,-930,-783,-616,-433,-237,-34,173,379,578,767,941,\
 1096,1228,1334,1411,1459,1475,1459,1411,1334,1228,1096,941,767,578,379,173,\
 -34,-237,-433,-616,-783,-930,-1054,-1153,-1224,-1268,-1283,-1270,-1229,-1164,\
 -1074,-964,-836,-695,-543,-384};
```

```
#ifndef pc
  rcc_clock_setup_in_hse_8mhz_out_24mhz();
  gpio_setup();
  usart_setup();
  timer_setup();
  irq_setup();
  adc_setup();

  gpio_set(GPIOC, GPIO8|GPIO9);
  channel_array[0] = 0; // channel number for conversion
  adc_set_injected_sequence(ADC1, 1, channel_array);
#endif
  while (1) {
#ifdef pc
     adc1_2_isr();
#endif
     if (jmf_index>=NB)
        {
#ifdef debug
         for (k=0;k<NB;k++) my_usart_print_int(USART1, temperature[k]);
#endif
         max=filtre(temperature,fil200,NB,NBFIL,sortie);
#ifdef debug
         for (k=0;k<NB-NBFIL;k++) my_usart_print_int(USART1, sortie[k]);
#endif
#ifndef pc
          my_usart_print_int(USART1, max);
#else
          printf("\t%f\t%d\n", freq,max);
          temps=0.;
          freq+=10.;
          if (freq>=(fe/10.)) return(0);
#endif
          jmf_index=0;
        }
#ifndef pc
     gpio_toggle(GPIOC, GPIO8);
#endif
  }
  return 0;
}

/* freq=[50:50:3000];
   k=1;
   for f=freq
     x=sin(2*pi*f*[0:8/32768:0.1]);
     y=filter(a,1,x);
     mm(k)=max(y(100:200));k=k+1;
   end
   plot(freq,mm)
*/
```

On notera en particulier que ce programme est conçu soit pour s'exécuter sur PC (cas `#define pc`) soit sur STM32 (cas `#undef pc`) : la seule différence tient dans les fonctions d'initialisation et de communication, tandis que la partie arithmétique est commune aux deux implémentations. Il est en effet bien plus aisé de valider l'algorithme avec des valeurs synthétiques (appel explicite de la fonction de gestion d'interruption dans le `main` dans le cas de l'exécution sur PC – fonction qui renvoie dans ce cas des données idéales de sinisoïde à fréquence variable) que de déverminer le programme directement sur microcontrôleur.

3 Conception d'un filtre de réponse infinie (IIR)

Un IIR tient compte non seulement des dernières observations mais aussi des valeurs passées du filtre :

$$y_n = \sum_{k=0..m} b_k x_{n-k} - \sum_{k=1..m} a_k y_{n-k}$$

GNU/Octave propose les fonctions `cheb` et `butter` pour concevoir des filtres IIR. Une fois les coefficients b et a (cette fois a est un vecteur) identifiés, nous appliquons la relation précédente pour identifier la nouvelle valeur de la sortie filtrée. Un IIR présente moins de coefficients qu'un FIR pour une fonction de transfert donnée et par conséquent induit une lattence plus faible. Cependant, il peut être instable dans son implémentation numérique.

Exercice :
Exploiter ces fonctions pour concevoir un filtre aux performances comparables aux IIR proposés ci-dessus, et implémenter ces filtres dans le STM32 pour en valider le fonctionnement.

4 Au delà de la programmation – déverminer un programme avec gdb

Il arrive qu'un programme soit difficile à déverminer (*debug*) et nécessite de sonder l'état de la mémoire du microcontrôleur en cours d'éxécution. `gdb` (GNU Debugger) fournit une telle fonctionnalité : l'exécution du programme est interrompue afin de permettre à l'utilisateur de sonder des informations telles que valeur d'une variable, état de la pile ou point d'arrêt.

Sur le circuit qui nous intéresse ici, l'interface de programmation communique avec `gdb` au moyen de `openocd`. Cet outil, classiquement utilisé pour contrôler les sondes JTAG, supporte le protcole de communication des cartes de démonstration STM32VL-Discovery.

`gdb` nécessite d'une part une interface de communication vers le microcontrôleur capable d'exécuter le code qui lui est destiné, et d'autre part le fichier original qui a été transmis vers la cible afin d'associer les points d'arrêt (*breakpoint*) et les variables à ds symboles compréhensibles par le développeur.

Nous lançons d'une part
`openocd -s /usr/local/bin/openocd-bin/share/openocd/scripts/ -f board/stm32vldiscovery.cfg \`
`-f interface/stlink-v1.cfg` pour créer le lien de communication entre `gdb` et le microcontrôleur, et d'autre part
`arm-none-eabi-gdb programme.elf` afin d'interfacer `gdb` (pour architecture ARM) avec le microcontrôleur en chargeant le fichier et la table des symboles associée `programme.elf`. La communication à proprement dit s'initialise par `target remote localhost:3333` et le programme est chargé en mémoire du microcontrôleur par la commande `load`. Une fois le programme en cours d'éxécution, il est possible d'en interrompre la séquence d'opération, définir des points d'arrêt (`breakpoint`), afficher la liste des instructions (`list`) ou afficher l'état de la pile (`backtrace`) ou de variables (`print variable`).

Cette méthode de travail est aussi applicable sur PC si l'on prend soin d'autoriser au préalable de l'exécution du programme qui crashe la génération d'un fichier image du contenu de la mémoire au moment du crash (`core dump`). La génération de ce fichier est désactivée par défaut sous GNU/Linux à cause de l'espace important requis par ces fichiers. Pour ce faire, `ulimit -c unlimited` et on exécute le programme trivialememt erroné ci-dessous

```
#include <stdio.h>

void assigne(char* c,int i)
{c[i]=i;}

int main()
{char c[4];
 int i;
 for (i=0;i<40960;i++) {
    assigne(c,i);
    printf("%d\n",i);}
}
```

qui se traduit par une violation d'accès à un segment mémoire qui n'est pas alloué à ce processus et la génération du fichier `core.pid`. `gdb` est alors invoqué par `gdb ./programme ./core.pid`. Afin d'accéder aux symboles, on aura pris soin de compiler avec option de déverminage `-g` et la commande `print i` nous informe de l'étape à laquelle le programme a crashé.

```
Program terminated with signal 11, Segmentation fault.
#0  assigne (i=8660, c=0xbf8d9e2c "") at core.c:4
4       {c[i]=i;}
(gdb) print i
$1 = 8660
(gdb) bt
#0  assigne (i=8660, c=0xbf8d9e2c "") at core.c:4
#1  main () at core.c:9
```

Le programme peut aussi être exécuté (`run`), depuis `gdb`, éventuellement pas à pas (`start` puis `step`), avec assignation de points d'arrêt (`tb ligne` pour définir le *breakpoint*).

Références

[1] A.V. Oppenheim & R.W. Schafer, *Discrete-Time Signal Processing*, Prentice Hall (1989)

[2] W.H. Press, B.P. Flannery, S.A. Teukolsky, & W.T. Vertterling, *Numerical recipies in Pascal*, Cambridge University Press (1994), ou sur le web `http://en.wikipedia.org/wiki/Cooley%E2%80%93Tukey_FFT_algorithm`