Filtrage de signaux numériques

J.-M Friedt, 2 décembre 2016

Objectif de ce TP :
— acquérir périodiquement un signal et établir la fréquence d'échantillonnage,
— 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 Premier pas sur STM32

FIGURE 1 – La carte STM32VL-discovery

La plate-forme utilisée est disponible pour un peu moins de 12 euros chez Farnell (référence 1824325) et ne nécessite aucun matériel additionnel pour fonctionner. Le microcontrôleur équipant cette carte est le bas de gamme de la série STM32 mais le cœur ARM Cortex-M3 est compatible avec toute la gamme de ce fabriquant. En particulier, les caractéristiques de cette carte sont :

— processeur STM32F100RB, cœur Cortex-M3 cadencé à 24 MHz, 128 KB flash, 8 KB RAM,
— 7 canaux DMA, 3 USARTs, 2 SPIs, 2 I2C, ADC, DAC, RTC, horloges, deux chiens de garde,
— interface ST-Link par USB pour déverminer/programmer,
— résonateur externe rapide (HSE) 8 MHz,
— résonateur externe lent (LSE) 32768 Hz.
— deux LEDs (vert, bleu) connectées au **port C** (broches **8 et 9**).

Le STM32 [1] est une classe de microcontrôleurs proposés par ST Microelectronics autour du cœur ARM Cortex-M3 [1, 2]. Ce cœur, exploité par divers fondeurs sous d'autres dénominations, se programme en C au moyen du même compilateur que vu précédemment (`gcc`), mais configuré pour générer du code à destination de la nouvelle architecture cible (ARM), différente de l'architecture hôte (Intel). Il s'agit donc de *cross*compilation. Un ouvrage conséquent d'exemples est proposé en [3]. Cependant, contrairement aux exemples de cet ouvrage, nous utiliserons une implémentation libre des bibliothèques d'accès aux périphériques qu'est `libopencm3` [2]. Le rôle de la bibliothèque est de cacher au néophyte les détails du fonctionnement du matériel pour ne donner accès qu'à des fonctions plus explicites du type `gpio_set()` ou `gpio_clear()`. L'exemple ci-dessous fait clignoter une diode :

```
#include <libopencm3/cm3/common.h> // BEGIN_DECL
#include <libopencm3/stm32/f1/memorymap.h>
#include <libopencm3/stm32/f1/rcc.h>
#include <libopencm3/stm32/f1/gpio.h>

static void gpio_setup(void)
{rcc_clock_setup_in_hse_8mhz_out_24mhz();
 rcc_peripheral_enable_clock(&RCC_APB2ENR, RCC_APB2ENR_IOPCEN);
 gpio_set_mode(GPIOC, GPIO_MODE_OUTPUT_2_MHZ, GPIO_CNF_OUTPUT_PUSHPULL, GPIO8|GPIO9);
}

int main(void)
{int i;
 gpio_setup();
 gpio_set(GPIOC,GPIO8);
 gpio_clear(GPIOC,GPIO9);

 while (1) {
   gpio_toggle(GPIOC, GPIO8);
   gpio_toggle(GPIOC, GPIO9);
   for (i=0; i<800000; i++) __asm__("nop");
 }
 return 0;
}
```

Ce premier programme se contente de faire clignoter les deux diodes mais donne la séquence de base d'initialisation du microcontrôleur :

1. activer la source d'horloge du périphérique (GPIO)
2. définir la direction d'utilisation du périphérique (entrée ou sortie)
3. définir l'état initial du périphérique
4. une boucle infinie avec changement d'état des diodes et attente.

1. `www.st.com/stm32` et en particulier la série STM32F1xx `www.st.com/web/en/resource/technical/document/reference_manual/CD00171190.pdf`
2. Dépôt git à `https://github.com/libopencm3/libopencm3`

Le programme se compile[3] en complétant la commande de `gcc` par un certain nombre d'options :

```
arm-none-eabi-gcc -I/usr/local/arm-none-eabi/include -fno-common -mthumb -mcpu=cortex-m3 -msoft-float  \
  -MD -DSTM32F1 -c usart.c
arm-none-eabi-gcc -o usart.elf usart.o -lopencm3_stm32f1 --static -Wl,--start-group -lc -lgcc \
  -lnosys -Wl,--end-group -L/usr/local/arm-none-eabi/lib -T./stm32vl-discovery.ld -nostartfiles \
  -Wl,--gc-sections -mthumb -mcpu=cortex-m3 -msoft-float -mfix-cortex-m3-ldrd
```

Noter en particulier l'appel aux bibliothèques `c` et `opencm3` par les options `-lc` et `-lopencm3_stm32f1`. Par ailleurs, `-I` renseigne la liste des répertoires contenant les fichiers d'entête (`.h`) et `-L` la liste des répertoires contenant les implémentations des bibliothèques. L'option `-T` indique la taille et l'emplacement des ressources mémoire : ces informations sont contenues dans le fichier

```
MEMORY
{ rom (rx) : ORIGIN = 0x08000000, LENGTH = 128K
  ram (rwx): ORIGIN = 0x20000000, LENGTH = 8K
}

INCLUDE libopencm3_stm32f1.ld
```

qui peut se trouver dans l'archive disponible à `http://jmfriedt.free.fr/tp_stm32.tar.gz`.
Cette commande se résume par le `Makefile` suivant :

```
NAME=usart
#NAME=adc-dac-timer
#NAME=adc-dac-printf
#NAME=filtre
#NAME=miniblink
#NAME=timer_jmf
all: $(NAME).bin

$(NAME).bin: $(NAME).elf
	arm-none-eabi-objcopy -Obinary $(NAME).elf $(NAME).bin

$(NAME).o: $(NAME).c
	arm-none-eabi-gcc -I/usr/local/arm-none-eabi/include \
	  -fno-common -mthumb -mcpu=cortex-m3 -msoft-float -MD -DSTM32F1 -c $(NAME).c

$(NAME).elf: $(NAME).o
	arm-none-eabi-gcc -o $(NAME).elf $(NAME).o -lopencm3_stm32f1 --static -Wl,--start-group \
	      -lc -lgcc -lnosys -Wl,--end-group -L/usr/local/arm-none-eabi/lib \
	      -T./stm32vl-discovery.ld -nostartfiles -Wl,--gc-sections \
	      -mthumb -mcpu=cortex-m3 -msoft-float -mfix-cortex-m3-ldrd

flash: $(NAME).bin
	st-flash write v1 $(NAME).bin 0x8000000
clean:
	rm *.elf *.o
```

Lors du passage d'une famille de STM32 à l'autre, les 3 points auxquels il est bon de veiller sont

1. la fréquence de cadencement du cœur, 24 MHz pour le processeurs bas de gamme (VL), 72 MHz sinon,
2. la taille de la mémoire renseignée dans le *linker script* appelé par l'option `-T` (ici, `-T../stm32vl-discovery.ld`)
3. penser à préciser la famille du modèle de STM32 utilisé (par exemple `-DSTM32F10X_MD` pour la famille *Medium Density* du STM32F103).

Le listing correspondant se génère par :

```
arm-none-eabi-objdump -dSt usart.elf > usart.list
```

et la liste d'instructions aux divers formats attendus par les outils de programmation (format hexadécimal Intel ou Motorola, image binaire) à partir du fichier ELF :

```
arm-none-eabi-objcopy -Obinary usart.elf usart.bin
arm-none-eabi-objcopy -Oihex usart.elf usart.hex
arm-none-eabi-objcopy -Osrec usart.elf usart.srec
```

Rappel : `gcc -S` arrête la compilation à la génération de l'assembleur (fichier `.s`), `gcc -c` arrête la compilation à la génération des objets (avant le linker).

Une fois le programme compilé, il est transféré au micocontrôleur par l'interface JTAG, ici émulée par le port série virtuel[4]. La commande pour transférer le programme `usart.bin` est :

```
st-flash write v1 usart.bin 0x8000000
```

3. la chaîne de compilation croisée proposée est issue de `summon-arm-toolchain` disponible à `https://github.com/jmfriedt/summon-arm-toolchain` – alternativement disponible sous forme de paquet binaire sous Debian/GNU Linux nommé `gcc-arm-none-eabi`.
4. Le code source du programme `st-flash` est disponible à `https://github.com/texane/stlink`

Les exemples sont inspirés de [3] et des exemples de `libopencm3`[5]. On notera par ailleurs que l'outil pour transférer un programme au format binaire depuis le PC vers un processeur STM32F1 en dehors de son installation sur une carte Discovery est disponible sous la forme de `stm32flash`.

2 Communiquer par RS232

Le STM32 est muni d'au moins deux interfaces de communication asynchrone (USART). L'interface connectée au convertisseur RS232-USB de la carte de développement est l'USART1. L'exemple qui suit aborde deux aspects :

— communication sur port asynchrone (RS232) avec initialisation des périphériques, initialisation des horloges, et initialisation du port de communication (protocole d'échange des bits). La fonction principale se charge alors de communiquer des caractères sur le port série,

— exploitation des bibliothèques standard du C (`stdio` et `stdlib`) et, en particulier dans cet exemple `printf()` qui nécessite le point d'entrée (*stub*) `_write()`.

Côté PC, la communication par port série se fait au moyen de `minicom` : `CTRL A-O` pour définir le port (`/dev/ttyUSB0`) et le contrôle de flux (ni matériel, ni logiciel). `CTRL A-P` pour définir la vitesse (ici 115200 bauds, N81).

```
#include <libopencm3/cm3/common.h> // BEGIN_DECL
#include <libopencm3/stm32/f1/rcc.h>
#include <libopencm3/stm32/f1/gpio.h>
#include <libopencm3/stm32/usart.h>

//#define avec_newlib
#ifdef avec_newlib
#include <errno.h>
#include <stdio.h>
#include <unistd.h>
int _write(int file, char *ptr, int len);
#endif

static void clock_setup(void)
{ rcc_clock_setup_in_hse_8mhz_out_24mhz();
  rcc_periph_clock_enable(RCC_GPIOC); // Enable GPIOC clock
  rcc_periph_clock_enable(RCC_GPIOA); // Enable GPIOA clock
  rcc_periph_clock_enable(RCC_USART1);
}

static void usart_setup(void)
{ // Setup GPIO pin GPIO_USART1_TX/GPIO9 on GPIO port A for transmit. */
  gpio_set_mode(GPIOA, GPIO_MODE_OUTPUT_50_MHZ,
      GPIO_CNF_OUTPUT_ALTFN_PUSHPULL, GPIO_USART1_TX);

  usart_set_baudrate(USART1, 115200);    // USART_BRR(USART1) = (uint16_t)((24000000<<4)/(38400*16));
  usart_set_databits(USART1, 8);
  usart_set_stopbits(USART1, USART_STOPBITS_1);
  usart_set_mode(USART1, USART_MODE_TX);
  usart_set_parity(USART1, USART_PARITY_NONE);
  usart_set_flow_control(USART1, USART_FLOWCONTROL_NONE);
  usart_enable(USART1);
}

static void gpio_setup(void)
{ gpio_set_mode(GPIOC, GPIO_MODE_OUTPUT_2_MHZ, GPIO_CNF_OUTPUT_PUSHPULL, GPIO9|GPIO12);
}

int main(void)
{ int i, c = 0;
  clock_setup();
  gpio_setup();
  usart_setup();
  while (1) {
    gpio_toggle(GPIOC, GPIO9);
    gpio_toggle(GPIOC, GPIO12);
    c = (c == 9) ? 0 : c + 1; // cyclic increment c
#ifndef avec_newlib
    usart_send_blocking(USART1, c + '0'); // USART1: send byte
    usart_send_blocking(USART1, '\r');
    usart_send_blocking(USART1, '\n');
#else
    printf("%d\r\n", (int)c);
#endif
    for (i = 0; i < 800000; i++) __asm__("NOP");
  }
  return 0;
```

5. distribués à `https://github.com/libopencm3/libopencm3-examples`

```
}

#ifdef avec_newlib
int _write(int file, char *ptr, int len)
{int i;
 for (i = 0; i < len; i++) usart_send_blocking(USART1, ptr[i]);
 return i;
}
#endif
```

Nous avons vu que l'utilisation de la bibliothèque d'entrée-sortie standard (`stdio`), et en particulier son implémentation pour systèmes embarqués, est gourmande en ressources. Pour s'en convaincre, compiler le programme avec (`#define avec_newlib`) et sans (commenter cette ligne). Nous constatons une augmentation significative du temps de programmation du micro-contrôleur associée à une croissance significative de la taille du code – de 1580 octets à 29032 – qui inclut maintenant toutes ces fonctions.

Exercice :

1. constater le bon fonctionnement du programme en observant les trames communiquées sous `minicom`,
2. que se passe-t-il si un mauvais débit de communication est sélectionné ?
3. activer ou désactiver les fonctions standard du C implémentées par `newlib`. Quelle conséquence ? analyser en particulier ce que fait la fonction `_write()`, un des *stubs* de newlib.
4. envoyer une valeur hexadécimale (par exemple 0x55 et 0xAA) en ASCII, *i.e.* définir deux variables avec ces valeurs et écrire la fonction qui permette d'en afficher le contenu sous `minicom`,
5. envoyer une valeur en décimal (par exemple 250) en ASCII
6. sachant qu'une chaîne de caractères est représentée en C par un tableau de caractères se terminant par le caractère de valeur 0, ajouter une fonction d'envoi de chaîne de caractères.

L'interface de communication RS232 est le mode de communication privilégié entre un système embarqué et l'utilisateur. À peu près tous les circuits, même aussi complexes qu'un disque NAS Lacie [6], un routeur ethernet [7] ou NeufBox [8], exploitent ce genre d'interface en phase de développement.

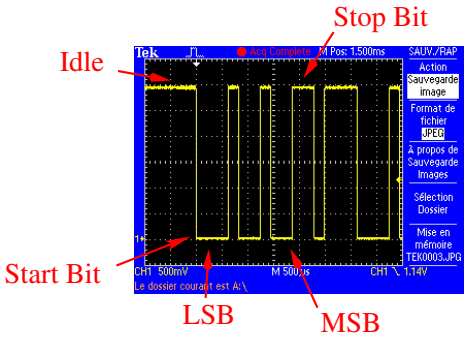
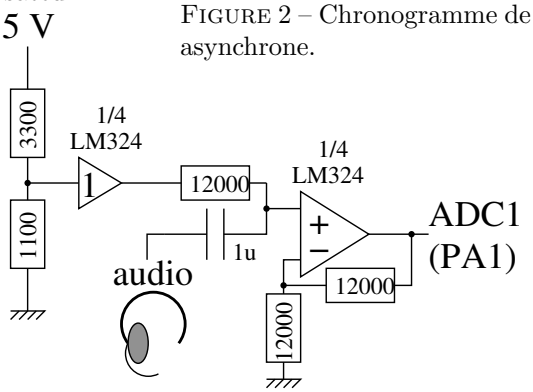


FIGURE 2 – Chronogramme de la liaison asynchrone.

3 Lire une valeur analogique

Tous les programmes réalisés jusqu'ici vont nous permettre d'atteindre le but de ce TP : mesurer une grandeur physique à intervalles de temps réguliers et retranscrire ces informations pour les rendre exploitables par un utilisateur.




```
//#include <stm32f1/stm32f10x.h>     // definition uint32_t
#include <libopencm3/cm3/common.h> // BEGIN_DECL
#include <errno.h>
#include <stdio.h>
#include <unistd.h>

#include <libopencm3/cm3/nvic.h>
#include <libopencm3/stm32/gpio.h>
#include <libopencm3/stm32/usart.h>
#include <libopencm3/stm32/f1/rcc.h>
#include <libopencm3/stm32/f1/adc.h>

int _write(int file, char *ptr, int len);

#define nbr 1024

static void clock_setup(void)
{ rcc_clock_setup_in_hse_8mhz_out_24mhz();
  rcc_peripheral_enable_clock(&RCC_APB2ENR, RCC_APB2ENR_USART1EN); // USART1
  rcc_peripheral_enable_clock(&RCC_APB2ENR, RCC_APB2ENR_IOPCEN); // port C (diodes)
  rcc_peripheral_enable_clock(&RCC_APB2ENR, RCC_APB2ENR_IOPAEN); // port A (USART1)
  rcc_peripheral_enable_clock(&RCC_APB2ENR, RCC_APB2ENR_ADC1EN);
}

static void usart_setup(void)
{ gpio_set_mode(GPIOA, GPIO_MODE_OUTPUT_50_MHZ,
    GPIO_CNF_OUTPUT_ALTFN_PUSHPULL, GPIO_USART1_TX);

  [... voir auparavant]
```

6. http://lacie.nas-central.org/wiki/Serial_port_%282Big_2%29
7. http://maschke.de/wag54g/
8. http://www.neufbox4.org/wiki/index.php?title=Port_s%C3%A9rie

```
}

int _write(int file, char *ptr, int len)
{
  [... voir auparavant]
}

static void adc_setup(void)
{ int i;
  gpio_set_mode(GPIOA, GPIO_MODE_INPUT, GPIO_CNF_INPUT_ANALOG, GPIO0|GPIO1|GPIO2);
  adc_off(ADC1);
  adc_disable_scan_mode(ADC1);           // single conversion
  adc_set_single_conversion_mode(ADC1);
  adc_disable_external_trigger_regular(ADC1);
  adc_set_right_aligned(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"); // demarrage ADC ...
  adc_reset_calibration(ADC1);
  adc_calibration(ADC1);
}

static uint16_t read_adc_naiive(uint8_t channel)
{ uint8_t channel_array[16];
  channel_array[0] = channel;                  // choix du canal de conversion
  adc_set_regular_sequence(ADC1, 1, channel_array);
  adc_start_conversion_direct(ADC1);
  while (!adc_eoc(ADC1));                      // attend la fin de conversion
  return(adc_read_regular(ADC1));
}

int main(void)
{int i,tab[nbr];
 int j = 0;
 clock_setup();
 usart_setup();
 gpio_set_mode(GPIOC, GPIO_MODE_OUTPUT_2_MHZ, GPIO_CNF_OUTPUT_PUSHPULL, GPIO8);
 gpio_set(GPIOC, GPIO8);
 adc_setup();

 while (1) {
      for (i=0;i<nbr;i++) tab[i]= read_adc_naiive(1);
      for (i=0;i<nbr;i++) printf("%d\n", tab[i]);
      gpio_toggle(GPIOC, GPIO8); /* LED on/off */
      for (i = 0; i < 100000; i++) __asm__("NOP");
 }
 return 0;
}
```

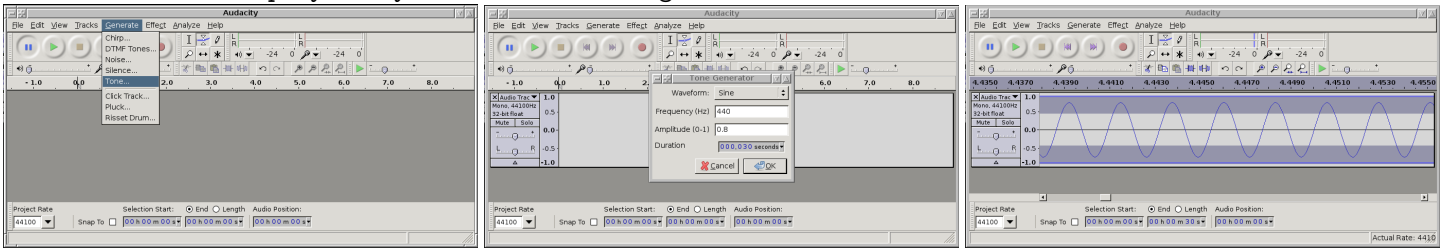
Le canal 16 est un cas particulier : il s'agit d'une sonde interne de température. Prendre soin cependant de l'activer pour l'utiliser.

Afin de générer des signaux de l'ordre de quelques dizaines de Hz à quelques kilohertz, nous allons exploiter la carte son qui équipe la majorité des ordinateurs personnels. Un logiciel, `audacity`[9] propose de synthétiser un signal et de l'émettre sur la sortie de sa carte son.

Sous GNU/Linux, un outil de configuration en ligne de commande, `play`, est fourni avec la boite à outils pour fichiers son, `sox`. Pour générer un signal à fréquence fixe sur la carte son : `play -n synth sin 440 gain -5` (éviter un gain trop élevé pour ne pas saturer l'étage de sortie de la carte son, qui se configure par ailleurs au moyen de `alsamixer`).

Pour balayer la fréquence de sortie (fonction *chirp* de `audacity`) :

```
while [ TRUE ]; do play -n synth 3 sin 440-660 gain -5;done
```



Nous nous contenterons pour le moment de générer une sinusoïde, pour n'exploiter que plus tard les formes d'ondes plus complexes (*chirp*).

Exercices :

1. enregistrer avec l'ADC du STM32 une séquence issue de la carte son du PC

9. disponible pour GNU/Linux, MacOS X et MS-Windows à `http://audacity.sourceforge.net/`

2. visualiser au moyen de GNU/Octave ou `gnuplot` [10]. le résultat de l'acquisition
   `gnuplot` affiche des courbes lues dans des fichiers :

   ```
   plot "fichier"
   ```

   on peut préciser les colonnes à afficher, l'axe des abscisses et ordonnées... :

   ```
   plot "fichier" using 2:3 with lines
   ```

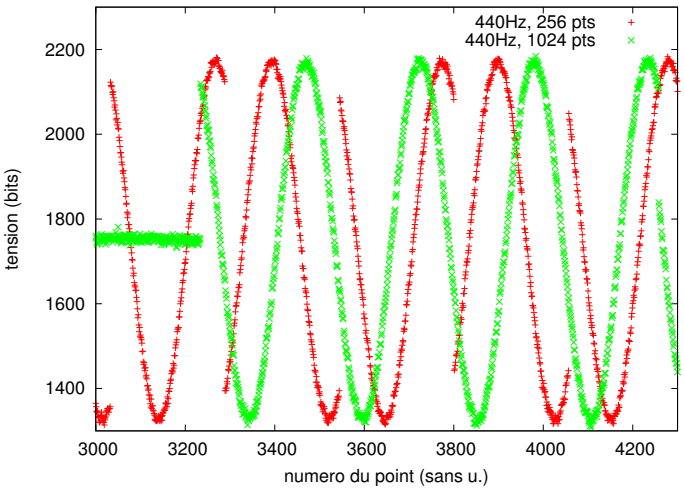
   pour utiliser la seconde colonne comme abscisse et la troisième colonne comme ordonnée, les points étant reliés par des lignes. `help plot` pour toutes les options. `gnuplot` présente l'avantage, par rapport à GNU/Octave (ou Matlab), de ne pas imposer un format strict des informations stockées dans le fichier (format matriciel avec un nombre de colonnes constant à toutes les lignes du fichier),
3. établir la fréquence d'échantillonnage.

Nous pouvons suivre deux stratégies pour établir la fréquence d'échantillonnage f_e : temporelle ou fréquentielle. Dans le premier cas nous comptons le nombre M d'échantillons dans N périodes et établissons, connaissant la fréquence f_s du signal mesuré, alors

$$f_e = \frac{M}{N} f_s$$

D'un point de vue fréquentiel, la transformée de Fourier sur M points d'un signal numérisé à f_e s'étend de $-f_e/2$ à $+f_e/2$. Connaissant la fréquence de ce signal numérisé qui se présente dans la transformée de Fourier comme une raie de puissance maximale en position N, nous déduisons que

$$N/M = f_s/f_e \Leftrightarrow f_e = \frac{M}{N} f_s$$



4 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 [4] (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 [11] (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` (Fig. 3).

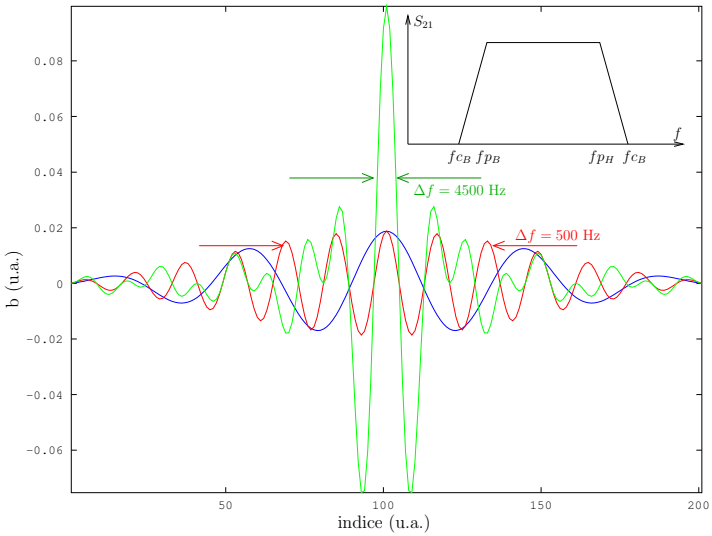
L'application d'un FIR de coefficients `b` (avec `a=1`) ou de façon plus générale d'un IIR de coefficients `a` et `b` sur un vecteur de données expérimentales `x` s'obtient par `filter(b,a,x);`

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. Le nombre de coefficients est déterminé par la bande de transition : pour une fréquence d'échantillonnage de f_e, la résolution spectrale pour une transformée de Fourier sur N points est f_e/N. Par bijection de la transformée de Fourier, on pourra estimer le nombre de coefficients $N \sim f_e/(fp_B - fc_B)$ avec fp_B et fc_B les fréquences de coupure et passante basses du filtre passe-bande, du même ordre de grandeur que $fc_H - fp_H$ avec fc_H et fp_H les fréquences de coupure et passante hautes du filtre passe-bande.

Exercices :

10. Pour une description plus exhaustive des fonctionnalités de `gnuplot` et `octave`, on pourra consulter `http://jmfriedt.free.fr/lm_octave.pdf`.

11. `octave` est une version opensource de Matlab et en reprend toutes les syntaxes (pour ce qui nous concerne ici, `load` et `plot`). Noter que contrairement à `gnuplot` qui peut extraire des colonnes de données d'à peu près m'importe que format de fichier ASCII (*i.e.* quelquesoit le nombre de colonnes par ligne), Matlab et GNU/Octave exigent de charger des fichiers contenant une matrice dont le nombre de colonnes est constant dans toutes les lignes, et sans chaînes de caractères autres que les commentaires commençant par %. Seule fonctionnalité nouvelle de GNU/Octave par rapport à Matlab qui nous sera utile ici : `octave` peut lire des données au format hexadécimal : `f=fopen("fichier","r");d=fscanf(''%x'');fclose(d);` La variable `d` contient alors les informations qui étaient stockées dans le fichier `fichier` au format hexadécimal. Si le fichier contenait `N` lignes, les données sont lues par colonne et les informations d'une colonne donnée d'obtiennent par `d(1:N:length(d))`.




```
pkg load signal
fe=100;  % tout en kHz
b=firls(200,[0 1.5 2.0 2.5 3.0 fe/2]*2/fe,[0 0 1 1 0 0]);
plot(b);hold on
b=firls(200,[0 5.5 6.0 6.5 7.0 fe/2]*2/fe,[0 0 1 1 0 0]);
hold on;plot(b,'r')
b=firls(200,[0 3.5 4.0 8.5 9.0 fe/2]*2/fe,[0 0 1 1 0 0]);
plot(b,'g')
xlabel('indice (u.a.)');ylabel('b (u.a.)')
axis tight
```

FIGURE 3 – Coefficients du filtre (b) pour une bande passante de 2000 à 2500 Hz (bleu) ou 6000 à 6500 Hz (rouge) : la largeur de la réponse indicielle du filtre reste constante et seule la fréquence de la porteuse est modifiée. Pour un filtre de 4000 à 8500 Hz (vert), la largeur de la réponse indicielle est réduite. Dans tous les cas la bande de transition $fp_B - fc_B$ et $fc_H - fc_B$ est constante et égale à 500 Hz. La fréquence d'échantillonnage est toujours supposée égale à 100 kHz.

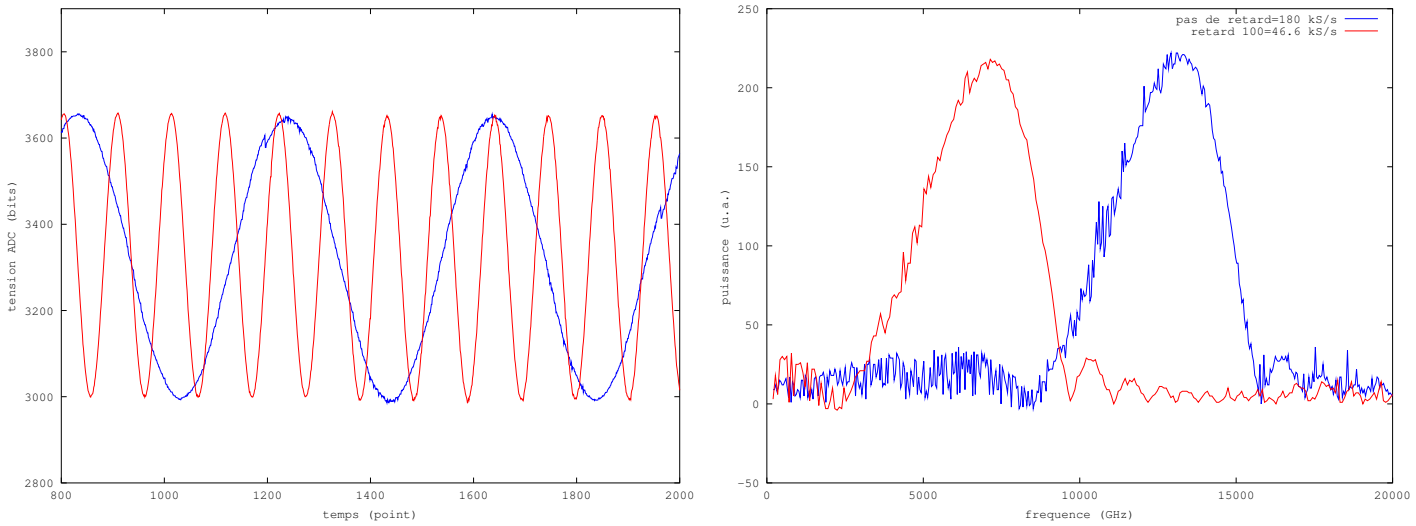


FIGURE 4 – Gauche : le même signal (sinusoïde à 440 Hz) échantilloné à deux fréquences différentes. Droite : les mêmes coefficients de filtres appliqués à un *chirp* balayant de 200 à 20000 Hz mais pour des fréquences d'échantillonage différentes.

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 ?

5 Application pratique du filtre

Exercices :

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) ?

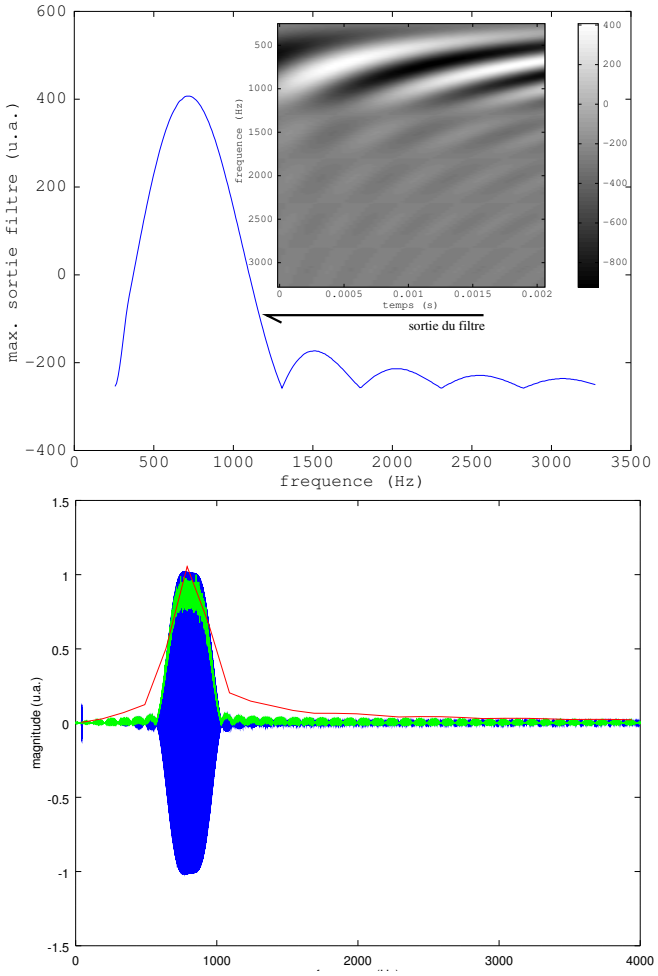
GNU/Octave fournit des coefficients en nombre à virgules inférieurs à 1. Un microcontroleur ne manipule pas efficacement des opérations sur ce type de données, en l'absence d'unité de calcul sur nombre à virgule flottante (*Floating Point Unit* – FPU) et il est préférable de représenter ces coefficients sur des entiers. Dans ce cas, le risque de dépassement de capacité de stockage des variables est significatif, et il faut calculer rigoureusement le nombre de bits sur lesquels les coefficients sont représentés pour garantir l'exactitude du résultat du calcul. En supposant que le résultat y du calcul est sur Q bits, que les mesures x sont sur P bits, et que nous opérons sur un filtre de M coefficients, démontrer que le nombre de bits N sur lesquels encoder les coefficients b du filtre est

$$Q = P + N + \log_2(M) \Leftrightarrow N = Q - P - \log_2(M)$$

Implémenter sur STM32 :

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.

Dans l'exemple de programme fourni ci-dessous, nous avons remplacé la détermination empirique de la fréquence d'échantillonnage par l'utilisation d'un périphérique matériel – le *timer* – pour imposer la cadence des échantillons. De cette façon, l'intervalle de temps entre deux mesures est imposé et indépendant du compilateur ou des options d'optimisation. Cette méthode est cependant un peu plus complexe à implémenter car elle nécessite de maîtriser un nouveau périphérique ainsi que les *interruptions*, un mécanisme qui romp l'exécution séquentielle du programme.




```
fe=16000;
ffin=4000;
fdeb=40;
% excitation par un chirp
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));hold on
% excitation par un bruit blanc et autocorr
x=randn(1,length(f)*2);
y=filter(b,1,x);
rxy=xcorr(x,y);
Ryx=abs(fft(rxy));
Ryx=Ryx(1:length(f));
plot(linspace(0,ffin,length(f)),Ryx/max(Ryx),'g')
% balayage explicite de frequence
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] = 1; // 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;
}
```

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 sinusoïde à fréquence variable) que de déverminer le programme directement sur microcontrôleur.

6 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.

Références

[1] G. Goavec-Mérou, J.-M Friedt, *Le microcontroleur STM32 : un cœur ARM Cortex-M3*, GNU/Linux Magazine France n.148 (Avril 2012), disponible à `jmfriedt.free.fr`

[2] J.- M Friedt, G. Goavec-Mérou, *Traitement du signal sur système embarqué – application au RADAR à onde continue*, GNU/Linux Magazine France n.149 (Mai 2012), disponible à `jmfriedt.free.fr`

[3] G. Brown, *Discovering the STM32 Microcontroller*, version du 8 Janvier 2013, disponible à `www.cs.indiana.edu/~geobrown/book.pdf`

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