arm Education

L.EEC025 - Fundamentals of Signal Processing (FunSP)

2023/2024 – 1st semester

Week13, 11 Dec 2023

Objectives:

-experimenting adaptive filtering in a system identification configuration highlighting:

- **the steepest descent concept**
- **the impact of the adaptation factor (β)**
- **the importance of the bandwidth of the excitation signal**

DSP Education Kit

LAB 11

Adaptive Filters

Issue 1.0

Contents

1 Introduction

1.1 Lab overview

The examples in these exercises concern variations of an adaptive FIR filter using the Least Mean Squares (LMS) algorithm, or the Normalized LMS algorithm.

2 Requirements

To carry out this lab, you will need:

- An STM32F746G Discovery board
- A PC running Keil MDK-Arm
- MATLAB
- An oscilloscope
- Suitable connecting cables

3 Adaptive Filter Using C Code [just for familiarization, not LAB assessment]

This example applies the Least Mean Square (LMS) algorithm, coded in C, to pre-determined input and desired output signals (sequences). It illustrates the following steps in the adaptation process using the adaptive structure shown in Figure 1.

1. Obtain new input values $x[n]$ and desired output sample $d[n]$.
2. Compute the output of the adaptive FIR filter $y[n]$ using equation (1).
3. Compute the instantaneous error signal $e[n]$ using equation (2).
4. Update each of the adaptive FIR filter's coefficients (weights) using equation (3). This is the (stochastic) LMS approximation of the iterative steepest descent algorithm.
5. Update the contents of the delay line containing N previous input samples.

These steps are repeated at every sampling instant.

$$y[n] = \sum_{k=0}^{N-1} h_n[k]\, x[n-k] \tag{1}$$

$$e[n] = d[n] - y[n] \tag{2}$$

$$h_{n+1}[k] = h_n[k] + 2\beta e[n] x[n-k] \tag{3}$$

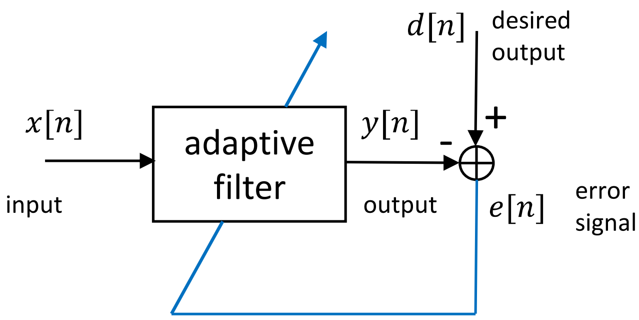

Figure 1: Block diagram of adaptive filter implemented by program `stm32f7_adaptive.c`

The following code snippet shows the program `stm32f7_adaptive.c` that implements the LMS algorithm for the adaptive filter structure shown in Figure 1.

The desired output signal used in program `stm32f7_adaptive.c` is

$$d(n) = 2\cos(2n\pi/8) \tag{4}$$

and the input signal is

$$x(n) = \sin(2n\pi/8) \tag{5}$$

The learning rate, number of filter coefficients, and number of sample instants simulated by the program are 0.01, 21, and 64, respectively.

```
// stm32f7_adaptive.c

#include "stm32f7_wm8994_init.h"
#include "stm32f7_display.h"

#define SOURCE_FILE_NAME "stm32f7_adaptive.c"

#define BETA 0.01f              // learning rate
#define N 21                    // number of filter coeffs
#define NUM_ITERS 64            // number of iterations

float32_t desired[NUM_ITERS]; // storage for results
float32_t y_out[NUM_ITERS];
float32_t error[NUM_ITERS];
float32_t w[N+1] = {0.0};        // adaptive filter weights
float32_t x[N+1] = {0.0};        // adaptive filter delay line
int i, t;
float32_t d, y, e;

int main()
{
  for (t = 0; t < NUM_ITERS; t++)
  {
    x[0] = sin(2*PI*t/8);         // get new input sample
    d = cos(2*PI*t/8);            // get new desired output
    y = 0;                        // compute filter output
    for (i = 0; i <= N; i++)
      y += (w[i]*x[i]);
    e = d - y;                    // compute error
    for (i = N; i >= 0; i--)
    {
      w[i] += (BETA*e*x[i]);      // update filter weights
```

```
      if (i != 0)
      x[i] = x[i-1];              // shift data in delay line
    }
    desired[t] = d;               // store results
    y_out[t] = y;
    error[t] = e;
  }
       stm32f7_LCD_init(0, SOURCE_FILE_NAME, GRAPH);
  while(1)
       {
              plotWave(desired, NUM_ITERS, 0, 0);
              proceed_statement();
              plotWave(y_out, NUM_ITERS, 0, 0);
              proceed_statement();
              plotWave(error, NUM_ITERS, 0, 0);
              proceed_statement();
       }
}
```

Now, run the program `stm32f7_adaptive` and observe its outputs by following these steps:

1. Build and run program `stm32f7_adaptive.c`. The program stores the desired output, output and error signals for $0 \leq n < 64$ in arrays `desired`, `y_out`, and `error` respectively. The arrays are of type `float32_t`.
2. By pressing the blue user pushbutton on the Discovery board, you can cycle through graphs on the LCD of the first 64 sample values of `desired`, `y_out`, and `error`.
3. Halt the program and save the contents of these arrays to data files by entering

   ```
   SAVE desired.dat <start address>, <start address + 0x100>
   SAVE y_out.dat <start address>, <start address + 0x100>
   SAVE error.dat <start address>, <start address + 0x100>
   ```

   in the *Command* window in the *MDK-Arm* debugger. Use the *Memory* window to find the start addresses of the arrays `desired`, `y_out`, and `error`.

4. Plot the contents of each of the data files using MATLAB function `STM32F7_BAR_real()`. The filter output should have converged to the desired output and the error should have decreased over the 64 sample instants simulated as shown in the following figures.

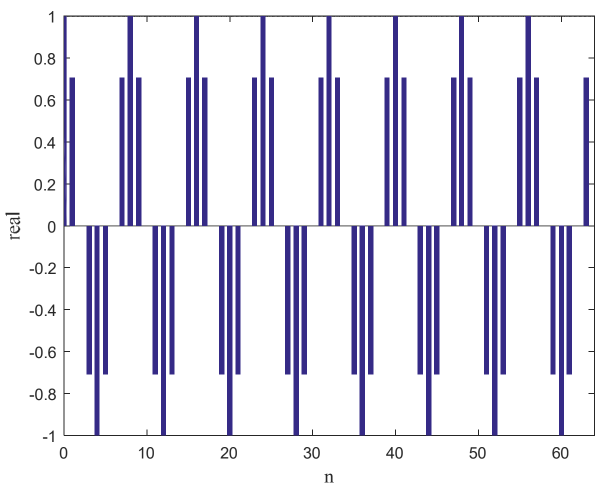

Figure 2: Desired output `desired`, simulated using program `stm32f7_adaptive.c`

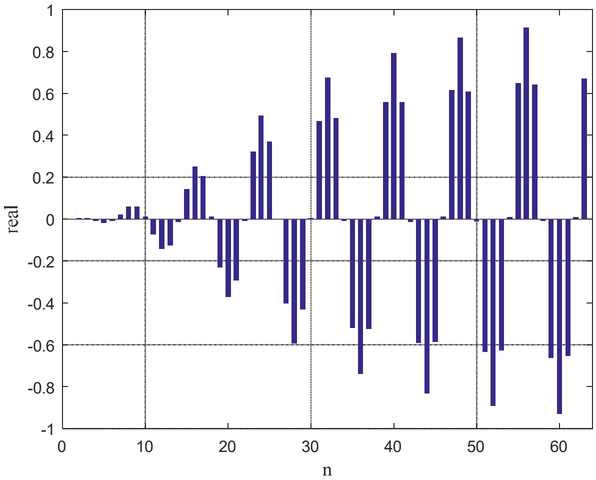

Figure 3: Adaptive filter output `y_out`, simulated using program `stm32f7_adaptive.c`

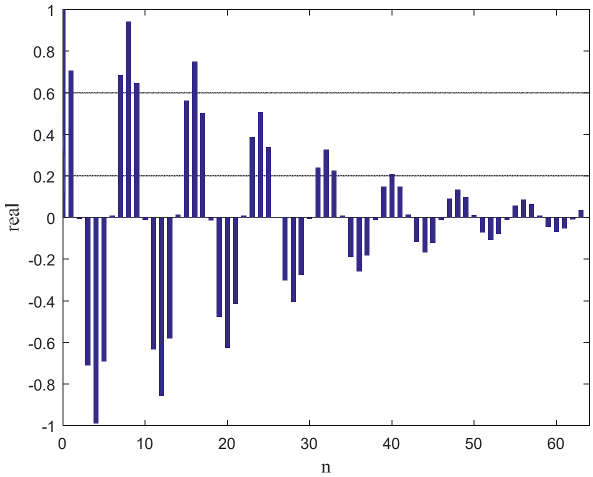

Figure 4: Error signal `error`, simulated using program `stm32f7_adaptive.c`

5. **Repeat the experiment using a learning rate (beta) of 0.02 and verify that convergence is faster.**

Program `stm32f7_adaptive.c` is an extremely simplistic demonstration of an adaptive filter. It is intended to introduce the relationships between input, output, desired output and error signals, and the role of the learning rate, and to illustrate how simple it can be to implement the LMS algorithm.

4 Adaptive FIR Filter for Noise Cancellation Using External Inputs [just for familiarization, not LAB assessment]

Program `stm32f7_noise_cancellation_intr.c` requires two external inputs, a desired signal and a reference noise signal to be input to left and right channels, respectively. Test input signals are provided in file `speechnoise.wav`. This may be played through a PC soundcard and input to the LINE IN socket on the audio card via a stereo 3.5 mm jack plug to 3.5 mm jack plug cable. The WAV file `speechnoise.wav` comprises pseudorandom noise on the left channel and speech on the right channel.

Figure 5 shows the program in a block diagram form. Within the program, a primary noise signal, correlated to the reference noise signal input on the left channel, is formed by passing the reference noise through an IIR filter. The primary noise signal is added to the desired signal (speech) input on the right channel.

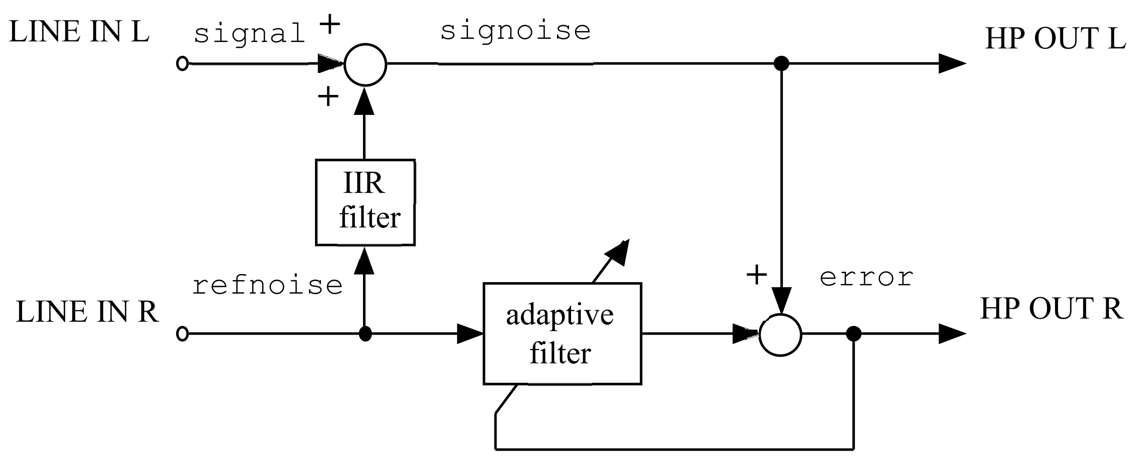

Figure 5: Block diagram representation of program `stm32f7_noise_cancellation_intr.c`

Build and run the program and test it using file `speechnoise.wav`. As adaptation takes place, the output on the left channel of HEADPHONE OUT should gradually change from speech plus noise to speech only. You may need to adjust the volume at which you play the file `speechnoise.wav`. If the input signals are too quiet, then the adaptation may be very slow.

While the program is running, use the blue user pushbutton to toggle between graphs on the LCD showing the adaptive filter coefficients (the impulse response of the adaptive filter) and the magnitude of their Fast Fourier Transform (FFT).

After adaptation has taken place, and the program has been halted, the 256 coefficients of the adaptive FIR filter, `firCoeffs32`, may be saved to a data file by typing:

```
SAVE <filename> <start address>, <end address>,
```

where `start address` is the address of array `firCoeffs32` and `end address` is equal to `start address + 0x400`, and plotted using the MATLAB function `stm32f7_logfft()`. The filter coefficients should reveal the impulse and magnitude frequency responses of the IIR filter implemented by the program and shown at the left-hand side of Figure 5. The characteristics of the IIR filter are determined by the coefficients in header file `bilinear.h`. You can substitute different coefficients by including, for example, header file `elliptic_bp.h`.

5 Normalized Least Mean Squares Algorithm [just informative, not LAB assessment]

In the previous example, you may have noticed that the rate of adaptation of the system could be influenced by the amplitudes of the signals involved. This effect can be reduced by using the *Normalized* LMS (NLMS) algorithm –the steps involved are summarized below.

1. Obtain new input and desired output sample values $x[n]$ and $d[n]$.
2. Compute the output of the adaptive FIR filter $y[n]$ using equation (1).
3. Compute the instantaneous error signal $e[n]$ using equation (2).
4. Compute the instantaneous energy, $energy[n]$ of the values stored in the filter delay line (input buffer) x, using equation (6)
5. Update each of the adaptive FIR filter's coefficients (weights) using equation (7).
6. Update the contents of the delay line containing N previous input samples.

These steps are repeated at every sampling instant.

$$energy(n) = \sum_{k=0}^{N-1} x^2(k) \tag{6}$$

$$h_{n+1}[k] = h_n[k] + \frac{2\beta}{energy} e[n]x[n-k] \tag{7}$$

Program `stm32f7_noise_cancellation_norm_CMSIS_intr.c` is a very slightly modified version of program `stm32f7_noise_cancellation_CMSIS_intr.c` that implements the normalized LMS algorithm.

Program `stm32f7_noise_cancellation_norm_CMSIS_intr.c` makes use of CMSIS library function `arm_lms_norm_f32()` in place of function `arm_lms_f32()` and uses a far larger learning rate, `beta`.

You should be able to verify that program `stm32f7_noise_cancellation_norm_CMSIS_intr.c` is relatively insensitive to the volume at which the test file `speechnoise.wav` is played.

6 Adaptive FIR Filter for System Identification of an FIR Filter [this is for LAB assessment]

Program `stm32f7_FIRadapt_intr_FPS.c` uses an adaptive FIR filter configured for system identification of another FIR filter (unknown to the adaptive filter), as shown in Figure 6.

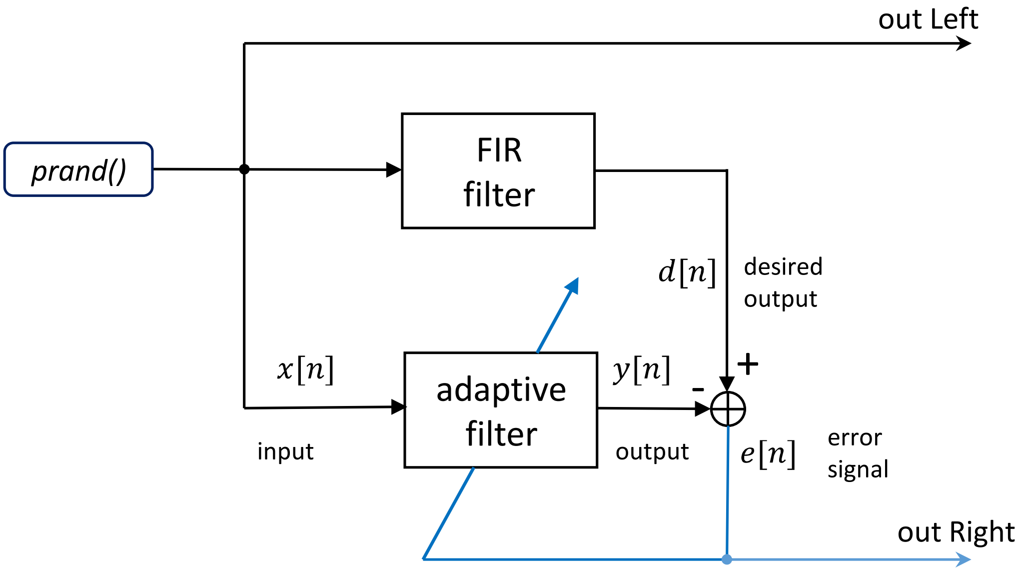

Figure 6: Block diagram representation of program `stm32f7_FIRadapt_intr_FPS.c`

Adaptation takes place in real-time while the same Pseudorandom Sequence (generated by function *prand()*) is input to both filters. You can watch on an oscilloscope the input of both filters and the difference between the outputs of the two filters, `error`. As the adaptive filter learns the characteristics of the unknown FIR filter, the variance of the error signal decreases.

For the purposes of appreciating the behavior of the adaptive filter, its rate of adaptation beta has deliberately been set very low (the range in `stm32f7_FIRadapt_intr_FPS.c` is between 1E-4 and 1E-0).

While the program is running, use the blue user pushbutton to toggle between graphs on the LCD showing the adaptive filter coefficients (the impulse response of the adaptive filter).

In one of the graphs (the most important!), two impulse responses are shown at the same time. The reference (ideal) impulse response is shown in **blue** samples. This impulse response is programmed in the `main()` of `stm32f7_FIRadapt_intr_FPS.c` . The time-varying impulse response of the adaptive filter is shown in **red** samples. This way, it is possible to visualize how the adaptive filter coefficients learns, in real-time, the impulse response of the reference filter. It should be emphasized that this filter is unknown to the adaptive filter. The adaptive filter typically starts from a vector of zeros and learns the impulse response of the unknown filter through the data (this is the fundamental idea at the origin of Machine Learning).

The length of the impulse responses of both FIR and adaptive filters is 64.

This example shares many similarities with the noise cancellation example. Both use an adaptive filter configured for system identification. However, in the case of noise cancellation, the output signal of interest is the error between the desired output and the output of the adaptive filter. On the other hand, in this example, the interest might be said to lie in the output of the adaptive filter or in its coefficients. In both examples, an FIR filter is adapted so as to take on the characteristics of the unknown FIR (which could also be an IIR filter!).

6.1 Lab introduction

In this Lab, we use the `main()` project file that is named **`stm32f7_FIRadapt_intr_FPS.c`** and that is available on the Moodle platform. Its C code is listed next.

```
// stm32f7_FIRadapt_intr_FPS.c
// uses normalized LMS

#include "stm32f7_wm8994_init.h"
#include "stm32f7_display.h"

#define BLOCK_SIZE 1
#define NUM_TAPS 64 // was 256

#define SOURCE_FILE_NAME "stm32f7_FIRadapt_intr_FPS.c"

// this is adapted from stm32f7_dft.c
typedef struct
{
  float32_t real; // this represents the ideal impulse response
  float32_t imag; // this represents the adaptive filter impulse response
} COMPLEX;

// reference impulse response versus estimated impulse response
COMPLEX refVSest[NUM_TAPS];


float32_t beta = 1E-3; // between 1E-4 and 1E-0 // using normalized LMS !

float32_t hREF[NUM_TAPS] = {0.0f};
float32_t x[NUM_TAPS] = {0.0f};
float32_t h[NUM_TAPS] = {0.0f};

extern int16_t rx_sample_L;
extern int16_t rx_sample_R;
extern int16_t tx_sample_L;
extern int16_t tx_sample_R;

// float32_t cmplx_buf[2*PING_PONG_BUFFER_SIZE];
// float32_t *cmplx_buf_ptr;
// float32_t outbuffer[PING_PONG_BUFFER_SIZE];
volatile int intr_flag = 0;


void BSP_AUDIO_SAI_Interrupt_CallBack()
{
  float32_t input;
  int16_t i, k;
       static int16_t index = -1;

  float32_t yn, adapt_out, error, dummy, energy;
```

```
  BSP_LED_On(LED1);

        index++;  index = index%32768;
  // input = (float32_t)(prbs(8000));
  // input = (float32_t)(rx_sample_L);
  input = 0.5f * prand();
        // input = 4000.0f*sin(2*PI*3000.0f/8000.0f*(float32_t)(index));
  x[0] = input; yn=0.0;
  for (k=0 ; k<NUM_TAPS ; k++)
  {
    yn += x[k] * hREF[NUM_TAPS-1-k];
  }

  adapt_out = 0.0; energy = 0.0;
  for (i=0; i<NUM_TAPS; i++)
  {
    adapt_out += (h[i]*x[i]);
    energy += x[i]*x[i];
  }
  error = yn - adapt_out;
  for (i = NUM_TAPS-1; i >= 0; i--) // update weights
  {
    dummy = beta*error;
    dummy = dummy*x[i];
    h[i] = h[i] + dummy/energy;
  }
        for (i = NUM_TAPS-1; i > 0; i--) x[i] = x[i-1]; // update delay line

  for(k=0; k < NUM_TAPS; k++)
  {
     refVSest[k].imag = h[NUM_TAPS-1-k]; // update most recent estimate
  }

  BSP_LED_Off(LED1);
  tx_sample_R = (int16_t)(error);
  tx_sample_L = (int16_t)(input);
  return;
}


int main(void)
{
  int start, k;
  int button = 0;

  // initialize our reference FIR impulse response
  start = 4;
  for(k=0; k <= 5; k++)
  {
    *(hREF+start+k)     = -0.1f * (float32_t)(k+1);
          *(hREF+start+10-k) = *(hREF+start+k);
  }
  start += 11;
  for(k=0; k <= 16; k++)
  {
    *(hREF+start+k)     = 0.15f * (float32_t)(k+1);
          *(hREF+start+32-k) = *(hREF+start+k);
  }
  start += 33;
  for(k=0; k < 11; k++)
```

```
  {
     *(hREF+start+k)    = *(hREF+4+k);
  }

  // this data is to be plotted (ideal versus estimated impulse response)
  for(k=0; k < NUM_TAPS; k++)
  {
     refVSest[k].real = *(hREF+k);
     refVSest[k].imag = 0.0f; // start with zeros
                h[k]=0.0f; x[k]=0.0f;
  }


  stm32f7_wm8994_init(AUDIO_FREQUENCY_8K,
                      IO_METHOD_INTR,
                      INPUT_DEVICE_INPUT_LINE_1,
                      OUTPUT_DEVICE_HEADPHONE,
                      WM8994_HP_OUT_ANALOG_GAIN_6DB,
                      WM8994_LINE_IN_GAIN_0DB,
                      WM8994_DMIC_GAIN_0DB,
                      SOURCE_FILE_NAME,
                     GRAPH);
  while(1)
  {
    button = checkButtonFlag();
    if (button == 1)
    {
                      plotLMS(h, NUM_TAPS, LIVE);
    }
    else if (button == 0)
    {
      plotWave(&refVSest->real, NUM_TAPS, 1, 1);
      // for(i=0; i<NUM_TAPS; i++)
      // {
      //   cmplx_buf[2*i] = h[i];
      //   cmplx_buf[2*i + 1] = 0.0;
      // }
      // arm_cfft_f32(&arm_cfft_sR_f32_len256, (float32_t *)(cmplx_buf), 0, 1);
      // arm_cmplx_mag_f32((float32_t *)(cmplx_buf),(float32_t *)(outbuffer), NUM_TAPS);
      // plotLogFFT(outbuffer, NUM_TAPS, LIVE);
    }
  }
}
```

Take a moment to analyze this code, to understand how the impulse response of the reference FIR filter is set, and the parts of the code implementing Equations (1), (2), (6) and (7).

Now we proceed, as indicated next, to compile the code, upload it to the STM32F7 board, and to run it. In this experiment, external analog signals generated by the function generator are not required.

After unzipping it, take the `stm32f7_FIRadapt_intr_FPS.c` file to the "src" directory that is located under the folder:

C:\uvision\Keil\STM32F7xx_DFP\2.9.0\Projects\STM32746G-Discovery\Examples\DSP Education Kit\

Remember that the directory where you can find the **`DSP_Education_Kit.uvprojx`** project file is:

C:\uvision\Keil\STM32F7xx_DFP\2.9.0\Projects\STM32746G-Discovery\Examples\DSP Education Kit\MDK-ARM

You can copy-paste this link directly to File Explorer in Windows for a quick and easy access. For your convenience, this link is also available on a TXT file on Moodle.

As in previous labs, we use the **`DSP_Education_Kit.uvprojx`** project file as our baseline project. This project file is represented by the icon DSP_Education_Kit.uvprojx, or just DSP_Education_Kit. Double-click on this file/icon to start the Keil MDK-Arm development environment (µVision). Replace the existing `main()` file in that project by the new `main()` that is `stm32f7_FIRadapt_intr_FPS.c` .

Now, proceed as usual to compile the code, downloading it to the STM32F746G board (by starting the debugger), and then to run the code.

6.2 Adaptive filter experiments

In `stm32f7_FIRadapt_intr_FPS.c` the reference impulse response is programmed according to the shape illustrated in Figure 7. This shape is intended to facilitate visualization and modification.

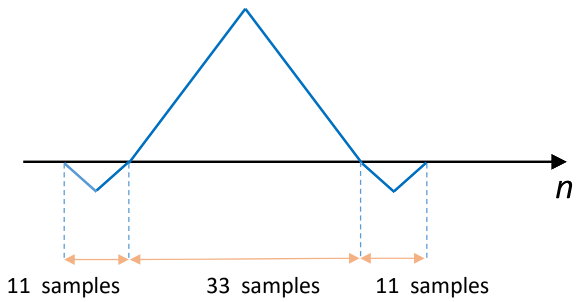

Figure 7: Shape of the reference impulse response programmed in `stm32f7_FIRadapt_intr_FPS.c`

Make sure that the code is running in real-time and take the STM32F746G LINE OUT LEFT and RIGHT output channels to the CHAN1 and CHAN2 inputs of the oscilloscope. As indicated before, in the lab we do not use the STM32F746G LINE IN inputs given that all signals are generated inside the STM32F746G kit.

Recall that Figure 6 identifies which signals are represented on the oscilloscope.

Right after you press the blue button[1] on the STM32F746G kit, you should see an evolution of the represented signals, on both the STM32F746G kit, and the oscilloscope, as figure 8 documents (please note that as pointed out at the beginning of Section 6, on the LCD display of the STM32F746G kit you observe two plots, one in **blue**, and another one in **red**).

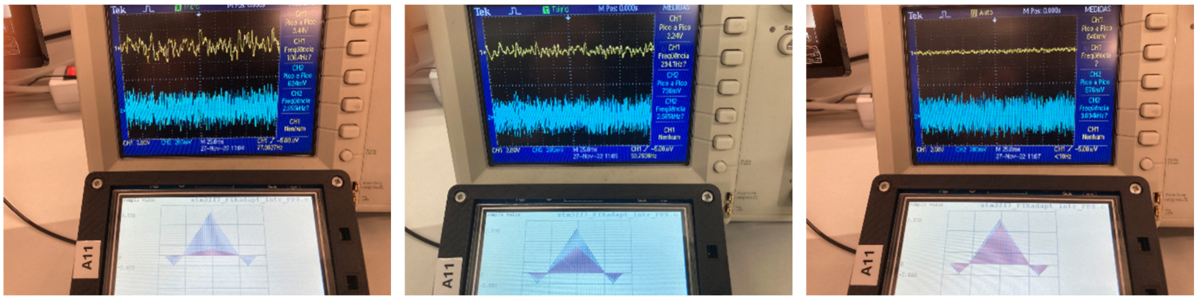

Figure 8: Screenshots of both the STM32F746G LCD and oscilloscope signals when program `stm32f7_FIRadapt_intr_FPS.c` *is running.*

Question 1 [2 pt / 10]: Identify the signals being represented on both the STM32F746G LCD and oscilloscope and explain why the amplitude of one of the signals in the oscilloscope decreases while one of the signals represented on the STM32F746G LCD converges to a target shape. Use the concepts of system identification, learning, and error in your explanation.

Now, stop the execution and modify in the `stm32f7_FIRadapt_intr_FPS.c` code the way the reference impulse response is programmed such that the shape becomes different from the original; for example, figure 9 illustrates two (easy) possibilities.

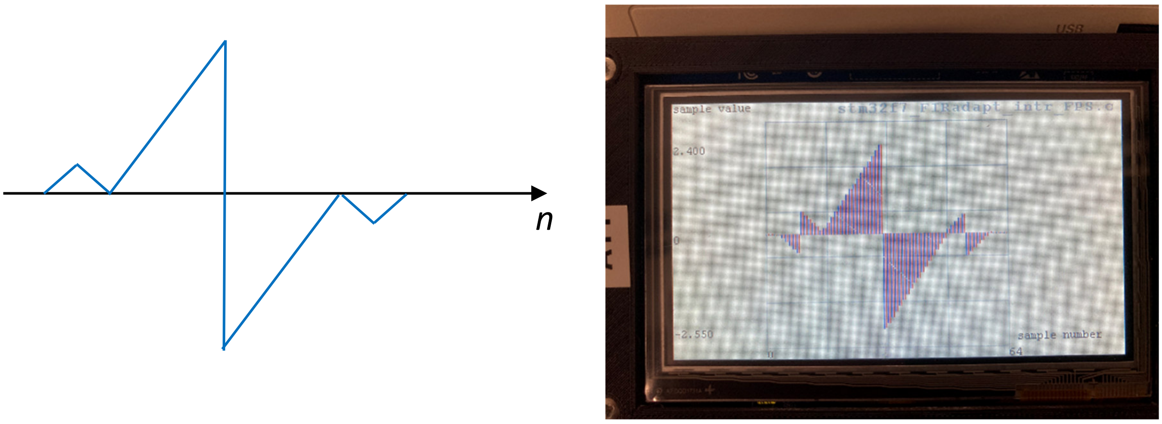

Figure 9: Two possible (and easy) modifications to the reference impulse response programmed in `stm32f7_FIRadapt_intr_FPS.c`.

[1] Please note that the STM32F7 kit has also a black button. If pressed, it restarts the code execution, which may be useful to restart the operation of the adaptive filter.

Now, proceed as usual to compile the code, downloading it to the STM32F746G board (by starting the debugger), and then to run the code.

Question 2 [2 pt / 10]: Show that convergence is still achieved independently of the modification on the pre-programmed reference impulse response.

Question 3 [2 pt / 10]: Stop the code execution and modify the value of the "beta" factor in the `stm32f7_FIRadapt_intr_FPS.c` to a new value between 1E-4 and 1E-0. Then proceed as usual to compile the code, downloading it to the STM32F746G board (by starting the debugger), and then to run the code. Try at least two alternatives. What is the impact of that change ? Do you confirm that the behavior of the execution is as expected ? In what sense ?

Question 4 [2 pt / 10]: Stop the code execution and modify the value of the "beta" factor in the `stm32f7_FIRadapt_intr_FPS.c` to a new value slightly above 1E-0, for example, 2. After you compile, download and run the code, you should then see a representation of the LCD screen as illustrated in Figure 10. How do you explain this outcome ?

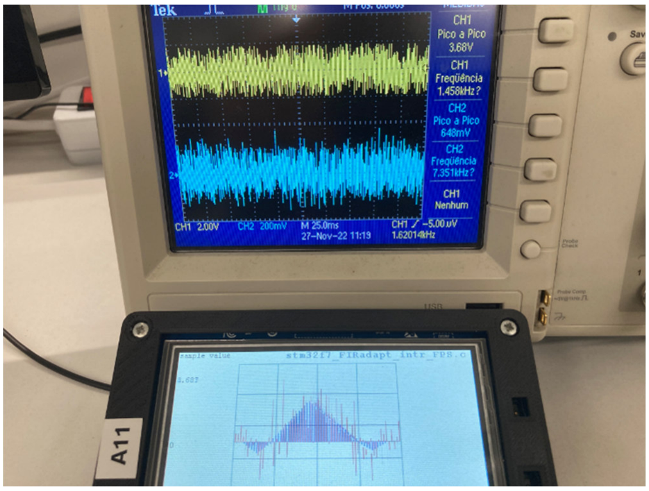

Figure 10: Illustrative result when the "beta" factor in `stm32f7_FIRadapt_intr_FPS.c` is 2.

In `stm32f7_FIRadapt_intr_FPS.c`, set the "beta" factor again to its initial value: beta = 1E-3.

Admit now that we introduce deliberately a bug in the code by changing the following code line:

```
    h[i] = h[i] + dummy/energy;
```

to

```
    h[i] = h[i] - dummy/energy;
```

Proceed to compile the code, downloading it to the STM32F746G board (by starting the debugger), and then to run the code. Watch the signals being represented on the STM32F746G LCD and oscilloscope.

Question 5 [2 pt / 10]: How do you interpret and explain the observations ?

Now, stop the code execution, reverse the previous code modification, uncomment the following code line:

```
        // input = 4000.0f*sin(2*PI*3000.0f/8000.0f*(float32_t)(index));
```

and keep beta = 1E-3. Proceed to compile the code, downloading it to the STM32F746G board (by starting the debugger), and then to run the code. After running, you should obtain a representation of the signals as suggested in Figure 11.

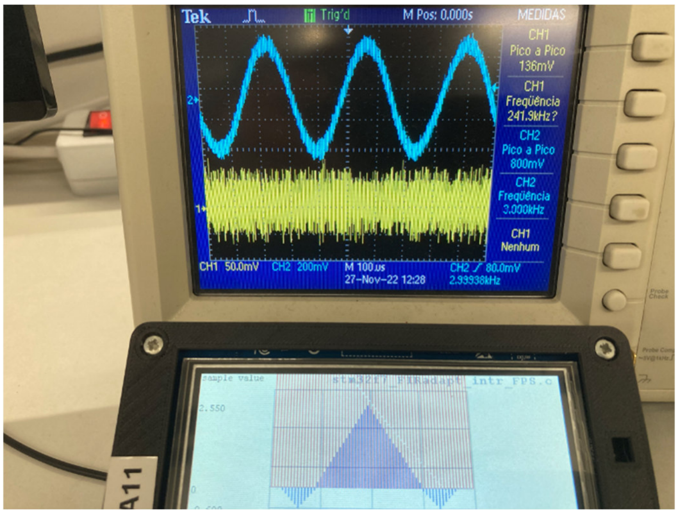

Figure 11: Illustrative results when the excitation signal is sinusoidal.

Question 6: The results suggest that the adaptive filter is not capable to operate as intended, even if the amplitude or frequency of the sinusoid is changed. How do you explain that ?

Note: the answer to this question may imply additional search beyond the information that is available on the lecture slides.

7 Conclusions

This laboratory exercise has introduced the LMS and normalized LMS algorithms for adaptive FIR filters. Real-time implementations of system identification have been demonstrated.