Chapter 4

Lab's example algorithm

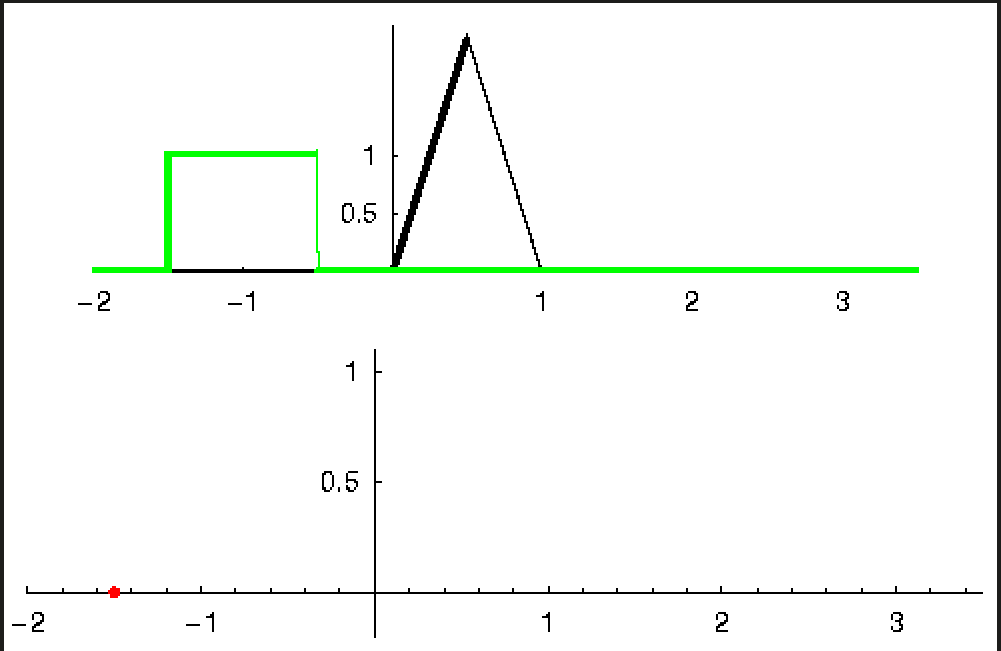
ENSI CAEN

ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

2021-2022

Lab sessions will use a well known algorithm: the **discrete convolution**.

This algorithm has a very simple structure, but it is very difficult to accelerate without mathematical refactoring.



Discrete convolution

Let's have a look at the mathematical definition of the discrete convolution

$$y(k) = \sum_{k=0}^{Y} \sum_{j=0}^{N} a(j) \cdot x(k-j)$$

Where:

- x() is the input samples vector
- y() is the output samples vector
- a() is the coefficients vector
- Y is the output vector size
- N is the number of coefficients
- k is the index of the current sample

LAB’S EXAMPLE ALGORITHM

Typical workflow for algorithm coding

Before being coded in C onto the wanted processor, the algorithm is usually validated with prototyping and simulation tools, such as Matlab/Simulink.

Validating the algorithm consists in coding its canonical implementation and check the input and output vectors values.

Typical workflow for algorithm coding

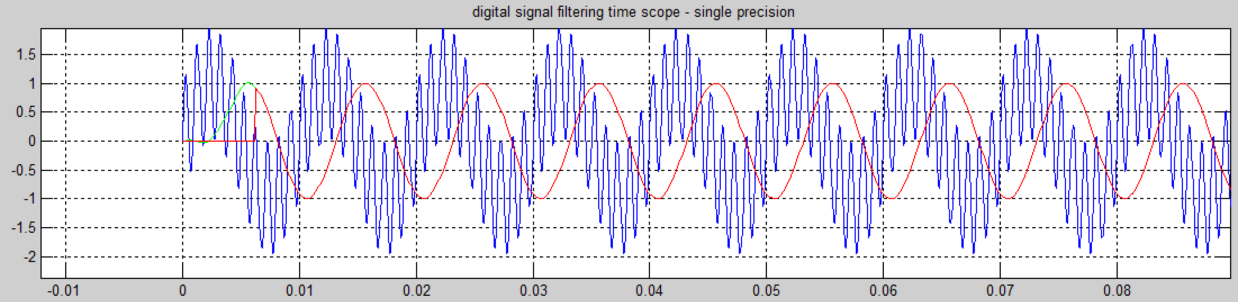
Here is the Matlab implementation of the discrete convolution algorithm.

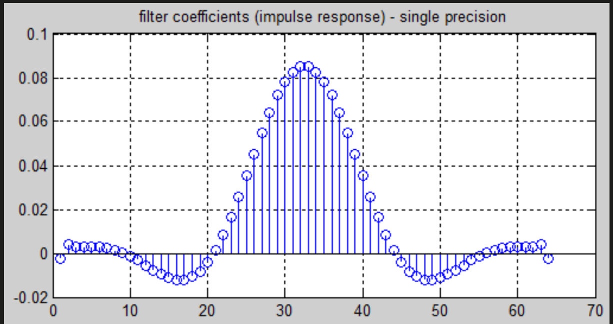
```
function yk = fir_sp(xk, coeff, coeffLength, ykLength)
    yk = single(zeros(1,ykLength));    % output array preallocation

    % output array loop
    for i=2:ykLength
        yk(i) = single(0);

        % FIR filter algorithm - dot product
        for j=1:coeffLength
            yk(i) = single(yk(i)) + single(coeff(j)) * single(xk(i+j-1));
        end
    end
end
```

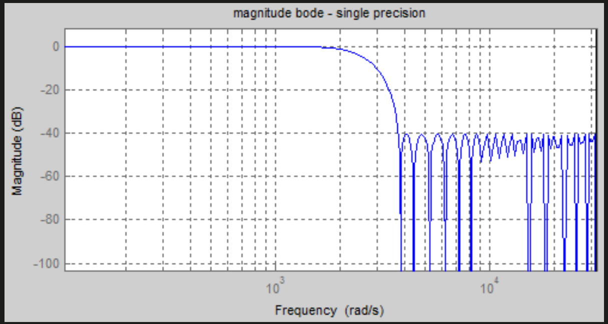
This code is given with lab materials

Typical workflow for algorithm coding

Observe some of the outputs suggested by Matlab sources, for a 64th-order FIR filter.







Matlab sources given with lab materials

Canonical C implementation

Once the algorithm has been validated, it can be implemented in the processor.

First make a canonical C implementation, using **IEEE-754 single-precision floats**.

```
void fir_sp (   const float * restrict xk,  \
                const float * restrict a,   \
                float * restrict yk,        \
                int na,                     \
                int nyk){
    int i, j;

    for (i=0; i<nyk; i++) {
        yk[i] = 0;

        /* FIR filter algorithm - dot product */
        for (j=0; j<na; j++){
            yk[i] += a[j]*xk[i+j];
        }
    }
}
```

Another canonical C implementation.

This one is given by Texas Instruments in its library **dsplib**.

```c
#pragma CODE_SECTION(DSPF_sp_fir_gen_cn, ".text:ansi");

#include "DSPF_sp_fir_gen_cn.h"

void DSPF_sp_fir_gen_cn(const float *x,
    const float *h,
    float *y,
    int nh,
    int ny)
{
    int i, j;
    float sum;

    for(j = 0; j < ny; j++)
    {
        sum = 0;

        // note: h coeffs given in reverse order: { h[nh-1], h[nh-2], ..., h[0] }
        for(i = 0; i < nh; i++)
            sum += x[i + j] * h[i];

        y[j] = sum;
    }
}
```

Another canonical C implementation, from the Texas Instruments **dsplib**.

But this time, it uses **16-bit signed integers** with the **Q1.15 format**.

```c
#pragma CODE_SECTION(DSP_fir_gen_cn, ".text:ansi");

#include "DSP_fir_gen_cn.h"

void DSP_fir_gen_cn (
    const short *restrict x,    /* Input array [nr+nh-1 elements] */
    const short *restrict h,    /* Coeff array [nh elements]      */
    short       *restrict r,    /* Output array [nr elements]     */
    int nh,                     /* Number of coefficients         */
    int nr                      /* Number of output samples       */
)
{
    int i, j, sum;

    for (j = 0; j < nr; j++) {
        sum = 0;
        for (i = 0; i < nh; i++)
            sum += x[i + j] * h[i];
        r[j] = sum >> 15;
    }
}
```

Goal

The main goal of the lab sessions is to present a **generic methodology for optimizing algorithms for a specific architecture**.

In our case, we'll optimize a **discrete convolution algorithm for a TI C6678 DSP**.