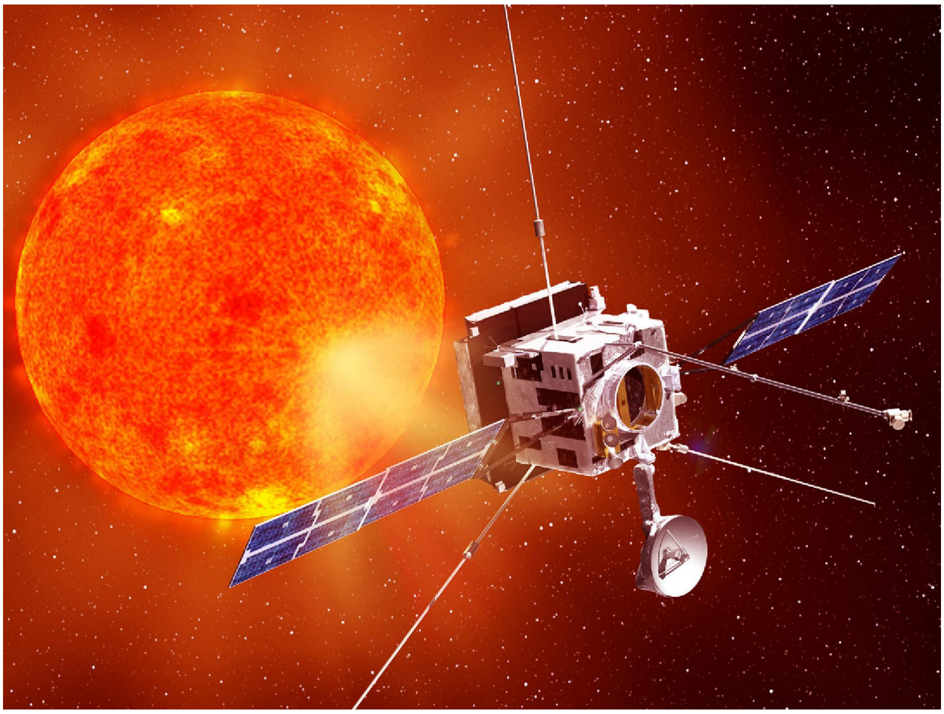
PHY571A - Projet de Recherche en Laboratoire

Embedded signal processing on space probe



September 2012 – December 2012

Pedro Luiz Coelho Rodrigues
X2010

Sous l'orientation de Alexis JEANDET et Paul LEROY

1 Table of Contents

2 Introduction

The space plasma physics of the LPP (Laboratoire de Physique des Plasmas), at the École Polytechnique, is responsible for the development and for the conception of an embedded data analyzer for the Solar Orbiter space probe, which launch is in 2017.

The LFR (Low Frequency Receiver) analyzer has the task of digitizing and processing the Electric and Magnetic Fields signals at frequencies below 10kHz (with 8 acquisition channels). The onboard processing of the analog waveforms allows the probe to send to earth only a few parameters, limiting dramatically the telemetry allocation of the instrument.

The research project that has been proposed by the LPP is the study of the functioning of the LFR analyzer and the development of an emulator for the system. This emulator, which has been developed in Python, will serve to validate the processing chain implemented in the analyzer and also to build tests cases to develop and validate the prototype of the instrument.

Furthermore, this emulator will allow the study of the influence of the numerical approximations done during the signal processing in the LFR analyzer.

This project was planned to be done during the two periods of the "Projet de Recherche en Laboratoire" course, meaning that the end of the activities is expected for the end of March/2013. The following report describes the work done in the period from September/2012 until December/2012.

3 Methodology

In order to achieve the proposed goals for this project, it was necessary to do some initial bibliographic research. This activity started with the study of basic notions of discrete signal processing, in order to understand the basic ideas behind the structure of the LFR analyzer. An extensive comprehension of the LFR specifications was also needed. For that, time was given for the reading of the specification document [1], as well as for questioning.

Then, a programming language was chosen for the development of the project, and it was Python. Time was given to learn the use of this new tool. The idea behind this development in Python is to embed the written scripts in an interpreter developed by the researchers of the LPP, making it part of a much bigger environment used for the development of the LFR analyzer.

Finally, after some weeks of experimentation on Python, tests started to be done, in order to have, in the Python environment, the same kind of simulation results obtained with MATLAB.

The final goal was to be able to code Python functions that would do exactly the same kind of operations executed in the LFR analyzer, respecting even the limited number of bits of resolution (16 bits) for the processed signals. This would allow the researchers to effectively compare the results from the prototypes with the ones in the Python program.

3.1 Discrete signal processing

3.1.1 The Fourier transform

A signal is a physical quantity, which varies with respect to time and space, and contains information from source to destination. Some examples of signals are audio, radar and videos. In the case of the LFR Analyzer, the signals that we manipulate are Electric and Magnetic fields, which are obtained by the sensors of the Solar Orbiter probe (one triaxial searchcoil magnetometer and three electric potential sensors).

A signal that varies with time may be associated with a function $f(t)$. It is often very useful to define the Fourier transform $\hat{f}(t)$ of this function, which is

$$\hat{f}(\omega) = \int_{-\infty}^{+\infty} f(t) \,.\, e^{-i\omega t}\, dt$$

The Fourier transform of a signal gives us information about its behavior in the frequency domain, which is something quite helpful.

In the figures below, we see some examples of signals on the left and the absolute values of their corresponding Fourier transform on the right (because the signals are all real, it suffices to show only the right side of their Fourier transforms, since they will always be symmetric).

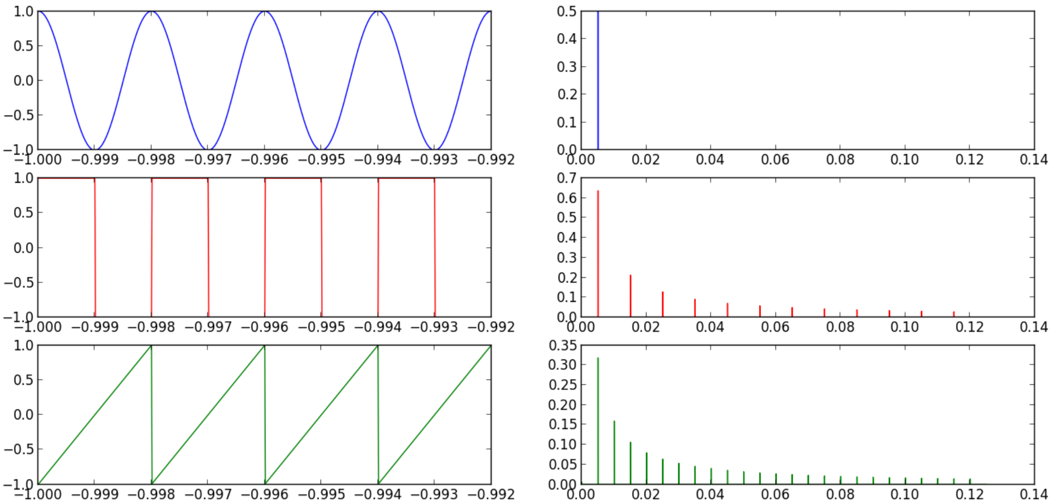


Figure 1 - Some examples of Fourier transforms

3.1.2 Filtering

A very important property of the Fourier transform is its relation to the convolution of two signals, as we shall see below.

When a signal $x(t)$ enters a linear time-invariant system, the output obtained is $y(t)$, and it is given by

$$y(t) = \int_{-\infty}^{+\infty} h(t-\tau)\,.\,x(\tau)\,d\tau$$

where the $h(t)$ function is called the impulsive response, and may be obtained by entering an impulse function $x(t) = \delta(t)$ in the system.

The operation above, which gives the output $y(t)$ in function of the input $x(t)$ and the impulse response of the system $h(t)$, is called convolution.

A very important theorem of the Fourier analysis states that the Fourier transform of $y(t)$ can be easily calculated by the product of the Fourier transforms of $x(t)$ and $h(t)$. So we have

$$\hat{y}(\omega) = \hat{x}(\omega)\,.\,\hat{h}(\omega)$$

Let's see an important application of this result.

A signal may have multiple components in frequency, as we saw it in the cases of the square and saw tooth waves on Figure **1**. These high-frequencies components may be undesirable, and some filtering may be needed.

A very common approach is to design a filter using his frequency domain characteristics and his inverse Fourier transform. The inverse Fourier transform will give the filter's impulse response in the time domain.

As an example, let's see the following figure, where we have a signal with two frequency components, one of high frequency and another one of low frequency (on the left side of the figure, we see the signal in the time domain, on the right we have the absolute value of its Fourier transform).

The input signal gets through a linear system with impulse response given by the red curve. The output result is a signal with "almost" just one frequency component, the one of low frequency. We can say that the input signal has gone through a low-pass filter, which eliminated the high-frequency components.

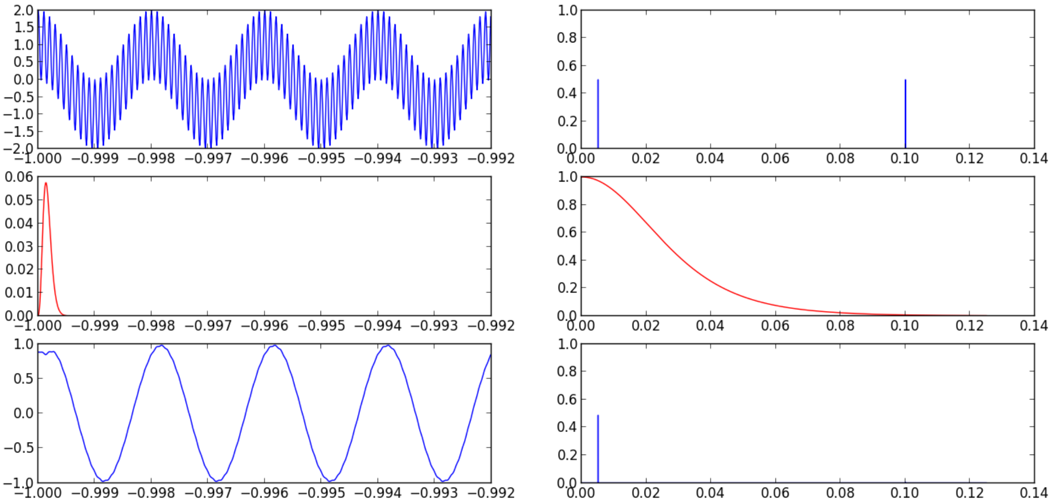


Figure 2 - Example of filtering

3.1.3 Discrete-time signals

Until now, we have only seen the behavior of signals in the continuous time domain, where we can define integrals and derivatives. However, in order to do digital signal processing, it is impossible to use continuous functions, we need to discretize them. And this is done by means of an operation called sampling.

Sampling a function $f(t)$ means to fix a value of T, which we shall call "period of sampling", and create a new function that relates to $f(t)$ by the following equation

$$f_s[n] = f(nT)$$

An example of a sampled signal $f(t)$ is shown in the graph below:

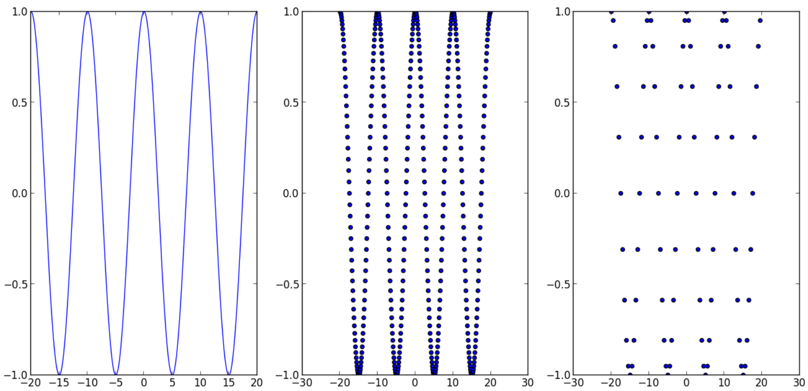


Figure 3 – The result of the sampling of a signal with two different sampling periods T

As we may see in the graph on Figure **3**, different sampling periods give different results for the function $f_s[n]$. It should be quite intuitive to see that with bigger sampling periods, we "lose more information" about the original signal. This loss of information sometimes implicates the impossibility of reconstruction of the original continuous signal.

This kind of situation, where the sampling frequency (which is the inverse of the period of sampling) is not big enough, is called aliasing. Every circuit designer must take aliasing effects into account when he wants to sample signals.

The figure Figure **4** shows an example of aliasing, where the real signal (in red) was sampled in such a way that the resulting sampled signal (in blue) is a signal of much smaller frequency.

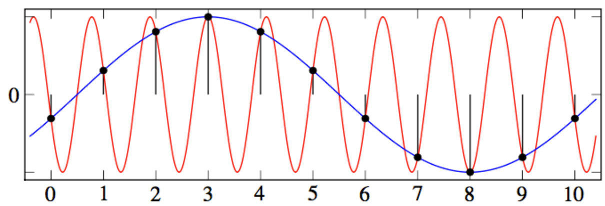


Figure 4 - Example of aliasing

One way to avoid aliasing is to use a frequency of sampling at least two times bigger than the highest frequency of the signal. This result follows from the theorem of Nyquist, which we won't demonstrate in this text.

A very common approach in the conception of electronic circuits for digital signal processing is to use a low-pass filter before sampling a signal. The reason for this is that a low pass filter will attenuate the high-frequency components of the signal to be sampled. This attenuation will allow a smaller sampling frequency and avoid problems of aliasing. That's why this low-pass filter is sometimes called "anti-aliasing" filter. One may use analog filters like the one presented on Figure **5** for this anti-aliasing operation.

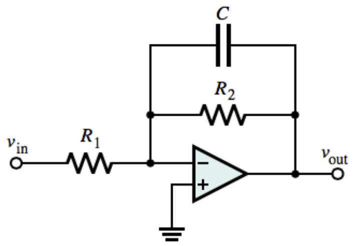


Figure 5 - Example of an analog low pass filter

3.1.4 Discrete Fourier transform

Working in the discrete time domain, it becomes quite natural to define a discrete Fourier transform, for discrete values of frequency. The DFT (Discrete Fourier Transform) of a discrete signal $f[n]$ in N points is defined by the following equation:

$$\hat{f}[k] = \sum_{n=0}^{N-1} f[n] e^{\frac{-2i\pi kn}{N}}$$

Note that it is important to precise the number N of points in which the DFT is going to be calculated, since it changes the frequency resolution of the transform (the difference in frequency corresponding to two consecutive values of k).

The values of the DFT $\hat{f}[k]$ are equivalent to 1/T times the sampled values of the Fourier transform $\hat{f}(\omega)$, where each k corresponds to a frequency value of $\frac{k}{TN}$ in Hertz.

$$\hat{f}[k] = \frac{1}{T}\hat{f}\left(\frac{2k\pi}{NT}\right)$$

However, one of the biggest limitations for the calculation of the DFT in digital systems is the number of operations needed to perform it, which is proportional to N^2 (N is the number of points used to do the DFT). Because of this polynomial complexity, for many years the idea of doing Digital Signal Processing was something seen as impractical.

Everything changed when Cooley and Tukey, in an article from 1965, proposed a new algorithm for calculating the DFT. The idea was to use the “redundant” characteristics of the complex exponentials in such a way that the complexity of the calculation could be reduced to something proportional to N log(N).

The Cooley-Tukey algorithm was the first of several algorithms that were developed for this fast calculation of the Discrete Fourier Transform, which the scientific community has given the name of Fast Fourier Transform (FFT). For instance, all the DFT’s calculated by MATLAB and Numpy are actually FFT’s.

3.2 The LFR analyzer

3.2.1 Presentation of the system

Solar Orbiter is a mission from the ESA (European Spatial Agency) that will allow scientists to know a lot more about the internal Heliosphere, by doing in-situ measurements that have never been done before.

Different antennas will detect different aspects of the Magnetic and Electric fields from the Sun, generating a total of 11 different input analog signals for the LFR analyzer.

These signals can't be directly sent to earth, since Solar Orbiter has a very limited bandwidth for sending data. This means that a lot of signal processing has to be previously done. This signal processing, which is done by the LFR analyzer, extracts the basic fundamental parameters of the input signals and allows the system to send only essential information back to earth.

The figure below shows the LFR block diagram, where we see the 11 analog input signals: BIAS_1, BIAS_2, BIAS_3, BIAS_4, BIAS_5, VHF_1, VHF_2, VHF_3, SCM_1, SCM_2, SCM_3. The signals with the prefix SCM_ are coming from the searchcoil magnetometer. All other signals are raw electric potentials (prefix VHF_) and mixed electric potential/electric field signals (prefix BIAS_). Only 8 signals are actually used for processing, as we can see by the red switches in the diagram.

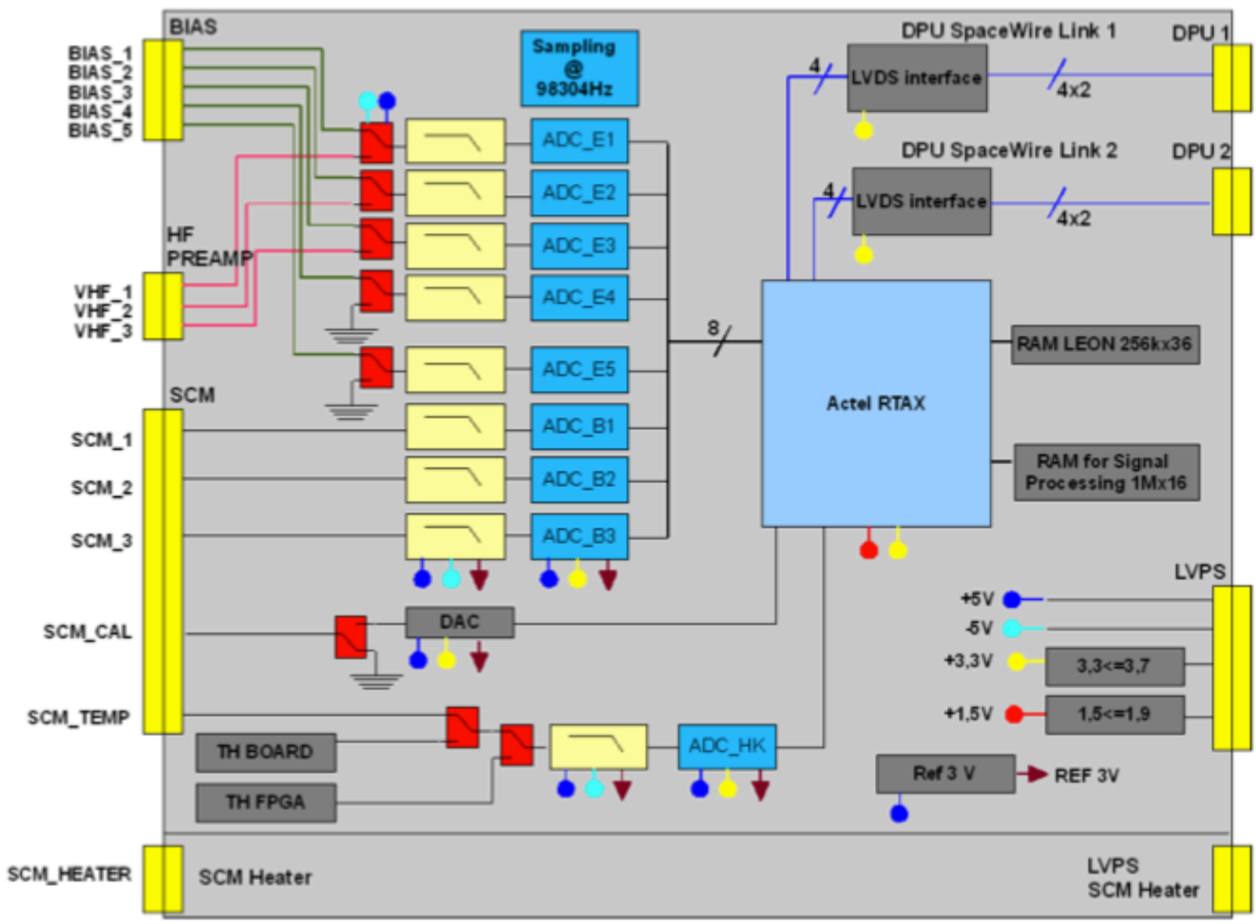


Figure 6 - LFR Block diagram

Knowing that we're only interested on signals of frequency between 0 and 10kHz, our input signals pass a first stage of anti-aliasing filtering, which guarantees a gain of -96dB for frequencies over 98304Hz, as well as a flat frequency response between 0 and 10kHz.

3.2.2 Digital data streams

After that, the filtered signal is sampled at a frequency of 98304Hz, and then converted to integer values through an Analog-to-Digital converter of 16 bits. Once in a digital form, a digital anti-aliasing filter, whose frequency response is shown in the figure below, filters the digital signal.

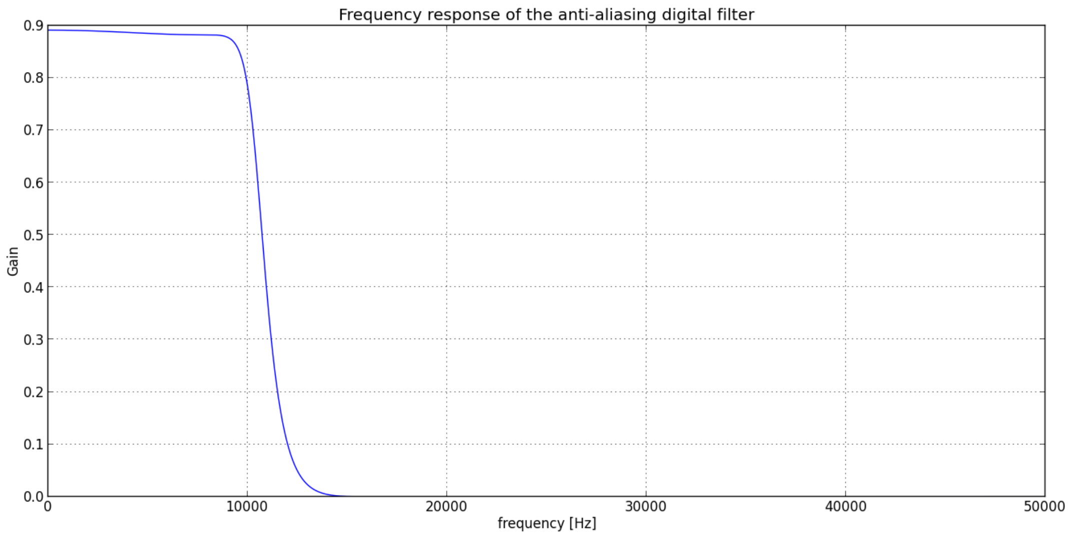


Figure 7 - Frequency response of the digital filter

This filtered digital signal is then down-sampled to a frequency of 24576Hz, which is one fourth of the original sampling frequency (which means that the procedure is simply that of taking one out of four samples, considering that the anti-aliasing digital filter is working well).

The signals that are really treated by the FGPA are these signals of frequency $f_o = 24576$ Hz, as shown in the figure below.

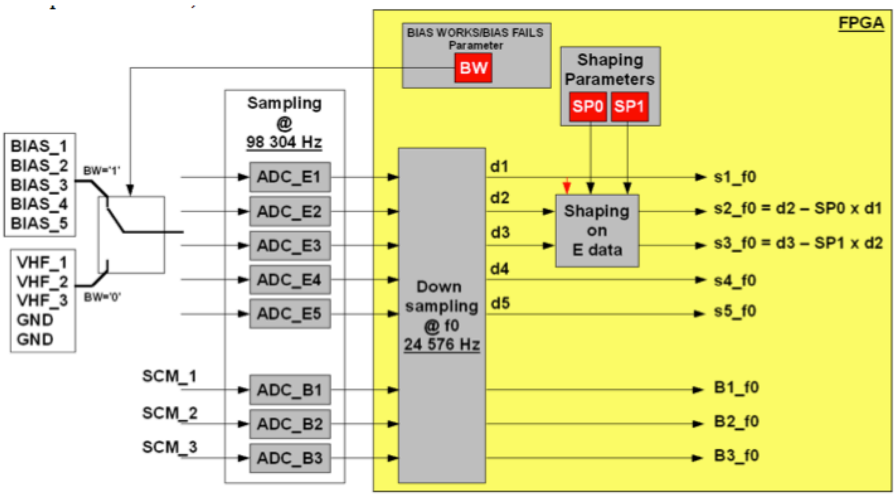


Figure 8 - Signals entering the FPGA

To obtain an optimized frequency analysis over the entire frequency band (from quasi DC to 10kHz), these signals initially at a frequency of f_o are downsampled by digital filtering. Three sub-sampling frequencies are used: f1 = 4096 Hz, f2 = 256 Hz and f3 = 16Hz.

3.2.3 Data processing

Finally, the streams of data, each at one of the possible frequencies f0, f1, f2 or f3, are processed by the FPGA. It's important to notice that we only deal with 6 components: two signals of Electric Field, three signals of Magnetic Field and one signal with the Electric Potential. The FFT of these signals is calculated and from that we obtain the Spectral Matrices (SM). Notice that no processing is done over the signals with sampling frequency of 16Hz.

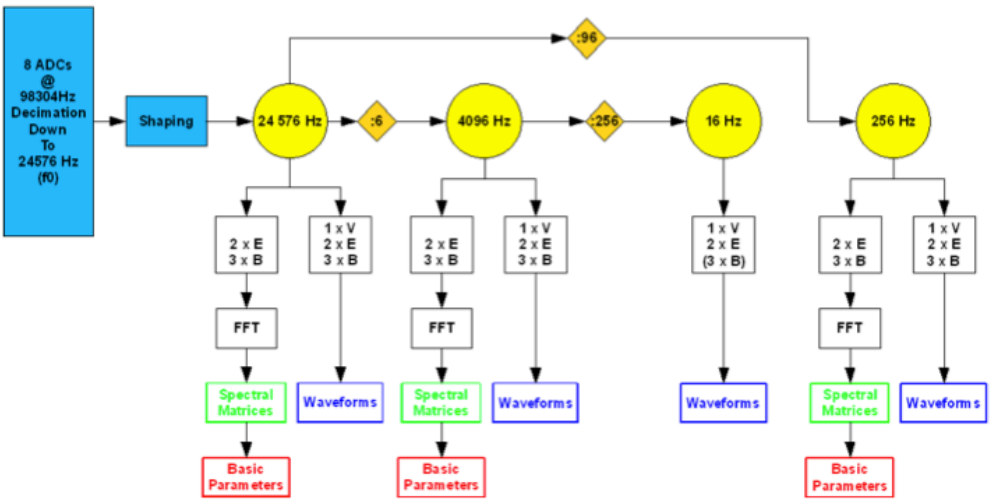


Figure 9 - Block diagram of the data processing

The Spectral Matrices are defined as follow

$$\mathbf{SM}\left(\omega_j^{(m)}\right) = \begin{bmatrix} B_1B_1^* & B_1B_2^* & B_1B_3^* & B_1E_1^* & B_1E_2^* \\ cc & B_2B_2^* & B_2B_3^* & B_2E_1^* & B_2E_2^* \\ cc & cc & B_3B_3^* & B_3E_1^* & B_1B_1^* \\ cc & cc & cc & E_1E_1^* & E_1E_2^* \\ cc & cc & cc & cc & E_2E_2^* \end{bmatrix}$$

Where the B_i and E_j are the Fourier transforms of the signals of Electric field and Magnetic field and the $\omega_j^{(m)}$ are the pulsation samples for the FFT done with N_{FFT} samples on the signal sampled at frequency f_m .

$$\omega_j^{(m)} = j\,\frac{2\pi f_m}{N_{FFT}}$$

From the Spectral Matrices we can calculate the Average Spectral Matrices, given by

$$\mathbf{ASM}\left(\omega_j^{(m)}\right) = \frac{1}{N_{SM}^{(m)}} \sum_{k=1}^{N_{SM}^{(m)}} \mathbf{SM}\left(\omega_j^{(m)}\right)$$

Other basic parameters may also be calculated from the **SM**. They represent the values of the normal wave vector, the wave's ellipticity, the wave's degree of polarization, the z-component of the normalized Poynting flux, the phase velocity estimator, the auto-correlations and normalized cross correlations.

Finally, depending on the chosen operation mode, these basic parameters and averaged spectral matrices are sent back to earth, as well as the signal's waveform.

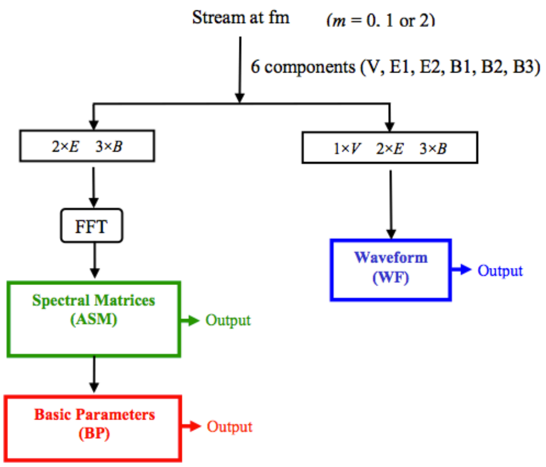


Figure 10 - Block diagram of the data processing for signals sampled at frequency $\boldsymbol{f_m}$ (m = 0, 1,2)

3.3 Python

3.3.1 Presentation of the language

Python is an interpreted programming language, whose design philosophy emphasizes code readability. Supporting object-oriented programming, as well as other programming paradigms, Python has become a very popular programming language for scientific computing.

Although MATLAB continues to be the most common choice for scientific computing, researchers have slowly begun to choose Python as an alternative to Mathwork's product. The main reasons for that are:

- Python is free;
- Python was developed in such a way that programs are easily read and comprehensible;
- Because of its layout, its easier to transform ideas into code;
- Portability: because Python is free, codes may run anywhere.

However, in order to do the same kind of job that MATLAB is capable of doing, the user must install three extra packages into his Python system: Numpy (BSD License), Scipy (library based on Numpy, BSD License) and Matplotlib (Python Software Foundation License).

Numpy and Scipy are two libraries that allow the user to use complex routines like FFT, integrals, derivatives, matrix multiplication, etc. Another advantage of Python in comparison to MATLAB is that it's possible to see exactly how each function was written. This allows the developer to be completely sure how each of the functions that he's using are being calculated.

Matplotlib is a very helpful package responsible for all kind of plotting in Python, using commands that resemble a lot to those used in MATLAB. For instance, all the graphs shown in this report were plotted using Matplotlib.

All that said, Python is also a multi-use language, capable of doing much more than scientific calculation, something that MATLAB is not capable of.

There are several options of IDE (integrated development environment) for Python - some of them are free. The one that was used during the period of activities reported in this text was the free IDE called Spyder, which has an interface like the one that we can see in the figure below.

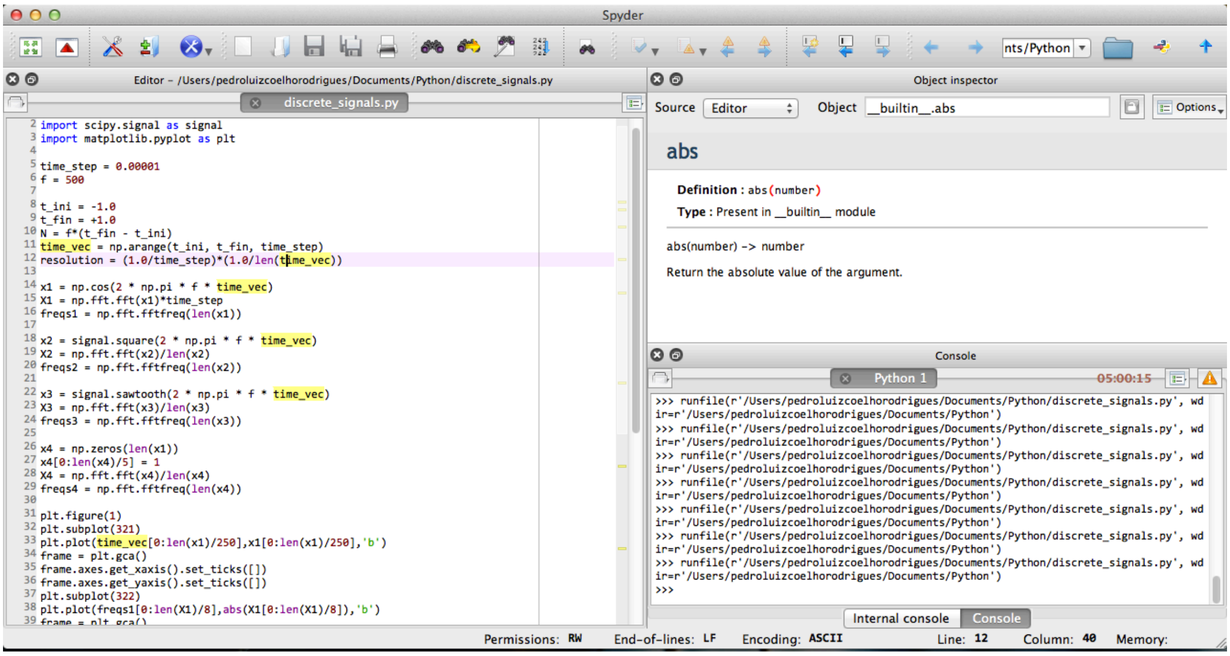


Figure 11 - IDE Spyder for Python

3.3.2 First works with Python

The first works done with Python were some simple scripts to test ways of using the Scipy and Numpy packages for digital signal processing. Tests with Matplotlib were also done.

An example of results obtained with Python is shown on Figure **12**, where we see the effects on the FFT of a signal when multiplying it by a by a window function, a very common operation used in practice.

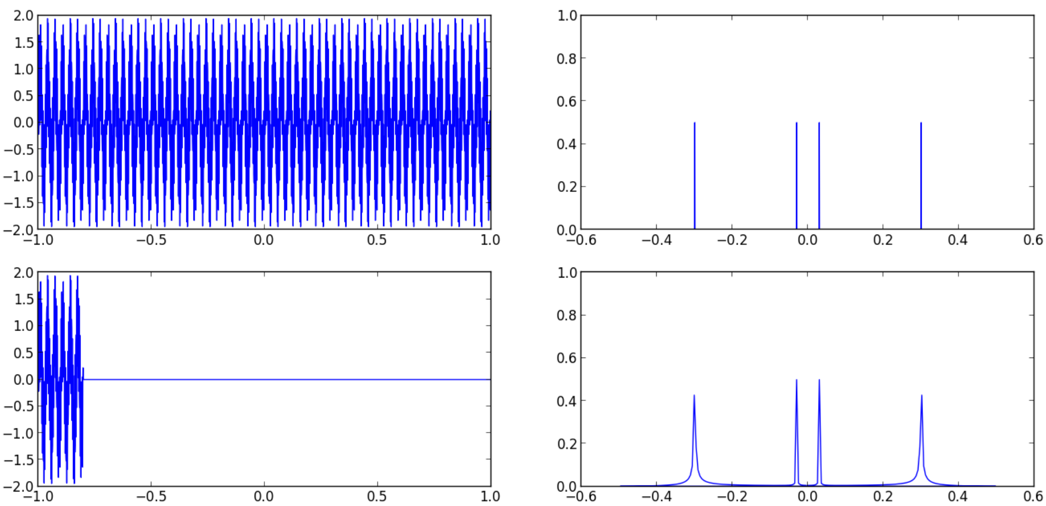


Figure 12 - Effects of windowing a signal

The figure below shows how the Python script for this example looks like.

```
import numpy as np
import matplotlib.pyplot as plt
#Time parameters
time_step = 0.001
t_ini = -1
t_fin = +1
time_vec = np.arange(t_ini,t_fin,time_step)
#Input signal
f1 = 30
f2 = 300
input_sig1 = np.sin(2 * np.pi * f1 * time_vec)
input_sig2 = np.sin(2 * np.pi * f2 * time_vec)
input_sig = input_sig1 + input_sig2
#FFT of the input signal
fft_input_sig = np.fft.fft(input_sig)/len(input_sig)
freqs1 = np.fft.fftfreq(len(input_sig))
#Window parameters
window = np.zeros(len(input_sig))
window[0:199] = 1
#Output signal
output_sig = input_sig * window
#FFT of the output signal
fft_output_sig = np.fft.fft(output_sig[0:199])/len(output_sig[0:199])
freqs2 = np.fft.fftfreq(len(output_sig[0:199]))
#Plotting
plt.figure(1)
plt.subplot(221)
plt.plot(time_vec,input_sig)
plt.subplot(222)
plt.plot(freqs1, abs(fft_input_sig))
plt.ylim(0,1)
plt.subplot(223)
plt.plot(time_vec,output_sig)
plt.subplot(224)
plt.plot(freqs2, abs(fft_output_sig))
plt.ylim(0,1)
```

Figure 13 - Python scrip

3.3.3 Developed functions and classes

The next step was to use Python's possibility of doing Object Oriented Programs. The idea was to develop classes that would to the exact kind of calculations that the filters of the LFR analyzer would do.

Although the digital filters from the LFR Analyzer are usually of order much bigger than two, their basic components are filters of order two, put in cascade. Knowing this, the classes representing these filters were done for second order filters, with the possibility of putting them in a series configuration (see Figure **14** for a schematic view of such a second order digital filter).

Attention was paid to the fact that convolution should be coded from scratch and not simply taken from the Numpy package. The reason for this is that, by coding functions exactly as they are being executed in the LFR analyzer, we will obtain exactly the same results as seen in practice. Even the rounding errors will be the same, which is something we're interested in. The Fast Fourier Transform routine should also be coded, for the same reasons discussed above, but that shall be done in the next period of activities (from January to March of 2013).

A class called *Filter_Coefficients* was created. The idea behind that was to create something like a list, where the user may insert the six coefficients of a digital second order filter. The way of creating an instance of this class is with the constructor *Filter_Coefficients*(a_0,a_1,a_2,b_0,b_1,b_2).

$$a_0 y[i] + a_1 y[i-1] + a_2 y[i-2] = b_0 x[i] + b_1 x[i-1] + b_2 x[i-2]$$

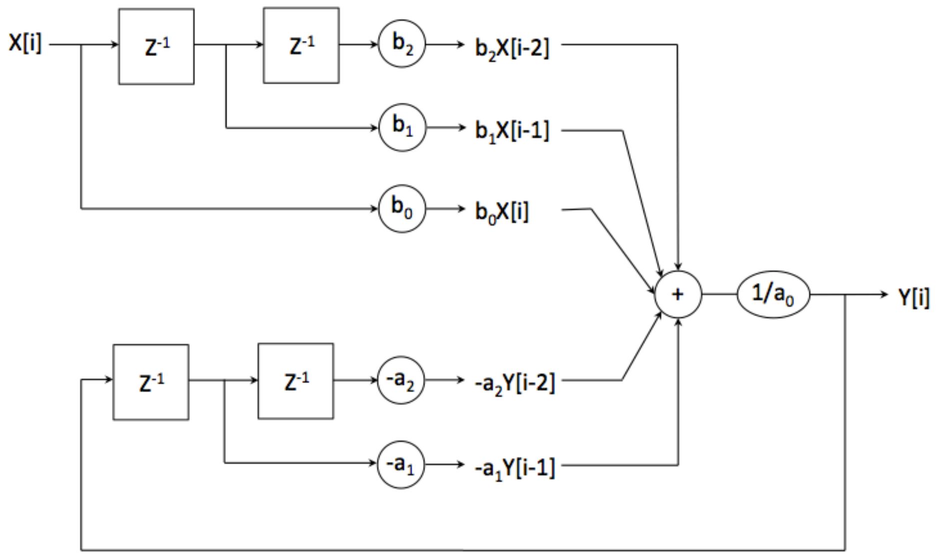


Figure 14 - Digital filter of order two

The next step was to effectively create the class for a second-order filter, where all the basic functions involving him and an input could be used. The created class is called *Filter* and it may be instantiated by calling it with a constructor using a *Filter_Coefficients* variable. The instantiation of a second order filter is presented on Figure **15**.

```
#Coefficients of the filter
Clp = Filter_Coefficients(1,-0.7,0,0.3,0,0)
#Instantiation of the filter
Flp = Filter(Clp)
```

Figure 15 - Creating a filter

Once the filter has been created, the user may convolve an input signal by using the *convolution* function of the Filter class. Other functions that were developed for this class are the *print_coefficients*, which prints all the filter coefficients, and *impulsive_response* functions, which gives the response of a filter when we have an impulse function ($\delta(x)$) in the input.

One other developed class is the one called *FilterGlobal*. The intention with this class was to allow the user to cascade several different second-order filters so as to get a filter equivalent to the one used in the LFR analyzer. The same kind of functions that we can use on a simple filter may be used for this kind of Global Filter. The instantiation of a higher order filter is presented on Figure **16**.

```
#Creation of the Higher Order Filter
Filters = [Flp,Flp,Flp,Flp,Flp]
F = FilterGlobal(Filters)
```

Figure 16 - Creating a cascade filter

Finally, one thing that was taken into account when developing the functions of the *Filter* and *FilterGlobal* classes was the fact that input signals might be very long, making it necessary to do signal processing in “slices” of the input vector. This was made possible by using internal variables in the classes, in such manner that the “joining ends” of each slice would be correctly calculated.

The next step for the development on Python is the adaptation of the functions so that they will use a limited number of resolution bits (16 bits) in each calculation, something that actually happens in the LFR Analyzer. This will allow a better understanding on the effects of approximation on the final values of the parameters sent back to earth.

4 Partial results

4.1 Results with the LFR analyzer filter

Tests were done in order to validate the classes and functions developed in Python. These tests were made on filters with the same coefficients as those used in the digital anti-aliasing filter from the LFR Analyzer (the one used before down-sampling to 24576Hz). The resulting impulse response, and its Fourier transform, was compared with the one obtained in MATLAB's DSP toolbox. As we can see in the figures below, the developed functions have exactly the same results as the ones obtained in MATLAB, except for a constant scale factor.

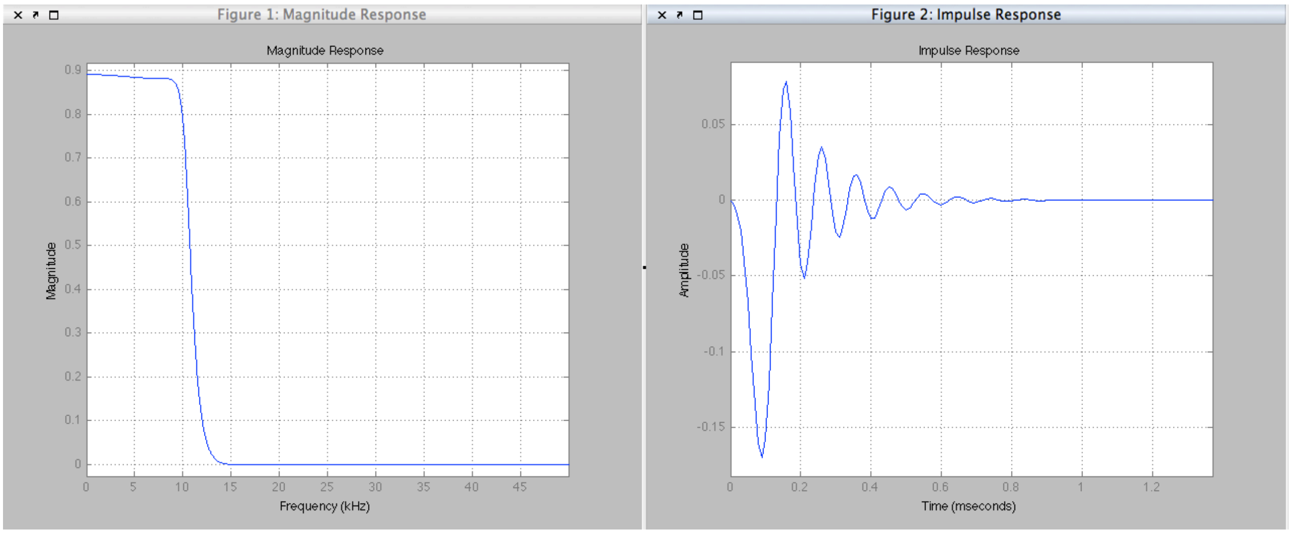


Figure 17 - Results obtained in MATLAB

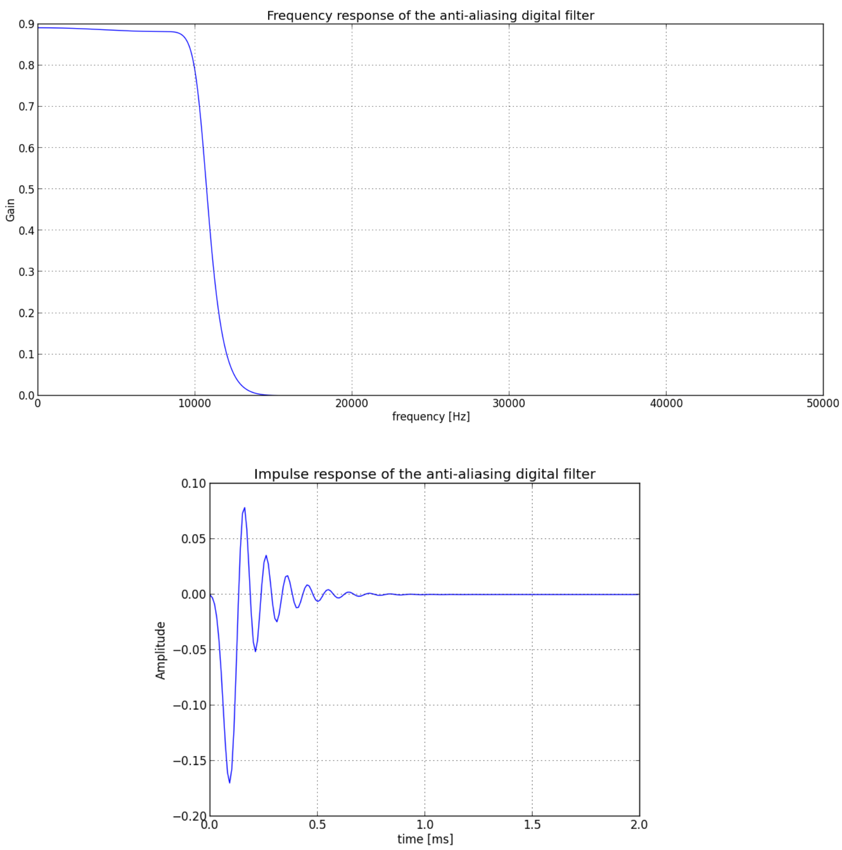


Figure 18 - Results with Python

5 Conclusion

This period of activities was really important. It allowed me to learn a new, and exciting, programming language. Although not many things were developed, the fact of spending time to really master the possibilities that Python (and its extra packages) may offer, will accelerate the process of development to be done in the second period of activities, from January to March.

The next steps to be done are:

- Finish the coding of the LFR emulator:
  - Write the function for the calculation of the FFT exactly how it is done in the FPGA of the LFR Analyzer
  - Take into account the limited number of resolution bits for the LFR analyzer
  - Code de CIC (Cascaded integrator-comb) filters used for the multi frequency resolution stage (figure 9)

- Test with typical physical signals:
  - Study of the physical meaning of the basic parameters to be sent back to earth
  - Contact the researcher responsible for the analysis of the received data
  - Construction of typical signals expected to be received by the LFR analyzer
  - Tests with these typical signals and analysis of the effects of integer approximation and integer manipulation in the FPGA
  - Build typical test cases that will be used to validate the prototype of the analyzer by comparing the results produced by the analyzer with the results produced by the simulator in Python.

6 References

[1] LFR Analyzer Documentation. Reference RPW-MEB-LFR-SPC-00003 (8 Mars 2012)

Discrete Signal Processing, Alan V. Oppenheim

Documentation for Python: http://www.python.org
Documentation for Scipy: http://www.scipy.org
Documentation for Numpy: http://numpy.scipy.org
Documentation for Matplotlib: http://matplotlib.org