

# Base designs

S. Denis, B. Maréchal G. Goavec-Mérou, J.-M Friedt

February 12, 2020

This document aims at making a brief description of basic RF functions that can be set-up, using Vivado and the fpga\_ip repository. It assumes acquired the a priori knowledge on the OscillatorIMP ecosystem, otherwise refer to : <https://github.com/oscimp/oscimpDigital>. The points discussed, listed below, are wrapped up in the example of a control loop design, with modulation and demodulation. A summary table of the IP blocks that can be used to build a numerical RF setup is given in the next pages.

- |                                  |  |
|----------------------------------|--|
| 1. Reminder on signal dynamics   | 7. Frequency and phase modulation of a NCO |
| 2. Webserver                     | 8. Filtering                               |
| 3. Double voltage source         | 9. Demodulation                            |
| 4. Double DDS                    | 10. Monitoring                             |
| 5. Amplitude modulation          | 11. Example to a control loop              |
| 6. Sine perturbation of a signal | 12. FAQ                                    |

## 1 Reminder on signal dynamics

Regardless of the presented functions, it's obviously better to optimize the dynamic of a signal to the range of data available. This first minimizes the part of noise of the electronics with respect to the signal. Secondly if the signal dynamic exceeds the range available, there is an overflow. For instance 14 *bits* signed data represent a range of  $\pm 13$  *bits* ie. from  $-8192$  to  $8191$  *arb. unit*. Above  $8191$  *arb. unit*, there is an overflow and the signal returns to the beginning of the range, ie.  $-8192$ . Representation in Fig.1.

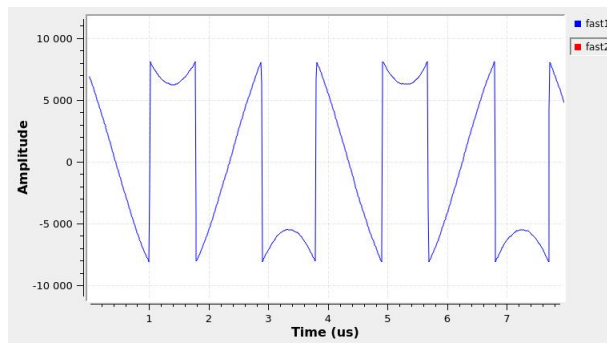
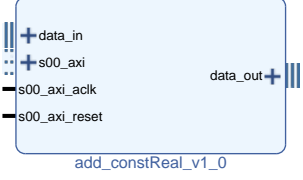
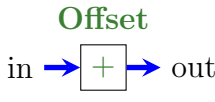
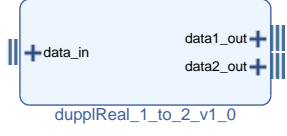
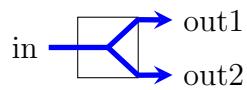
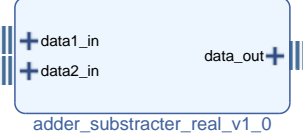
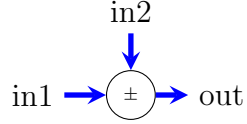
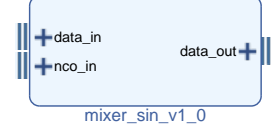
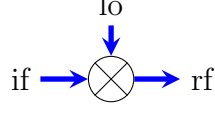
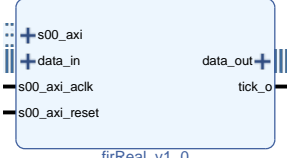

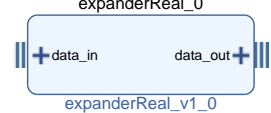

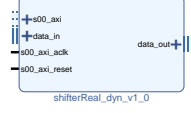
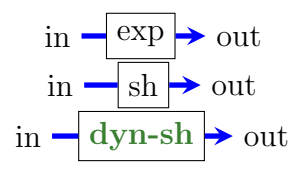
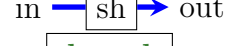

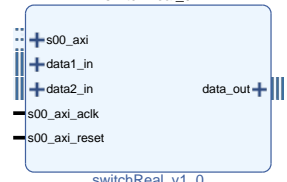
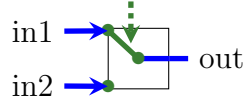
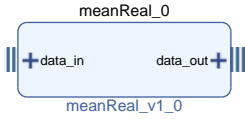
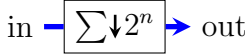


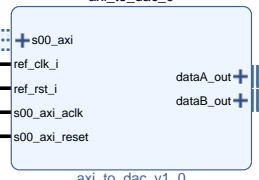
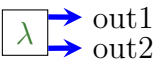
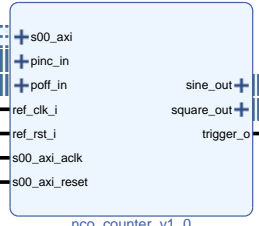
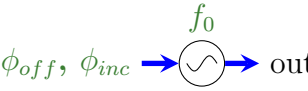
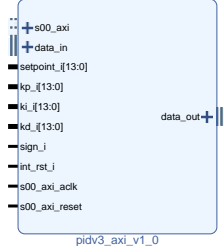

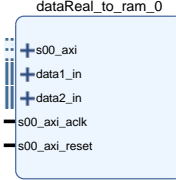

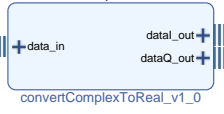
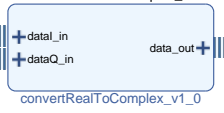
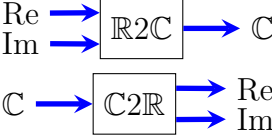


Figure 1: Overflow on the top and bottom of a sine.

IP	Equivalent RF function or numeric function	Equivalent scheme with tunable entries
	<p>Tunable amplitude offset, bias.</p> <p>The added offset value is internal to the block.</p>	<p>Offset</p> 
	<p>Splitter</p>	
	<p>Combiner.</p> <p>Add or subtract signals.</p> <p>The added/subtracted signal is external to the block unlike the add_const block.</p>	
	<p>Mixer, multiplier.</p>	
	<p>Tunable filter.</p> <p>FIR with decimation option.</p>	
  	<p>Can be assimilated to <math>2^m</math> amplifiers or attenuators.</p> <p>Are used to adapt the data size between blocks, or to select the range of the numeric signal.</p> <p>Expander: crop end of word, expand beginning of word.</p> <p>Shift: crop beginning of word, expand end of word.</p>	  
	<p>Switch</p>	

IP	Equivalent RF function or numeric function	Equivalent scheme with <b>tunable entries</b>
	Moving average. Decimation of $2^n$ with averaging: slows the data flow.	
	Tunable delay line ie. cables.	
	Tunable voltage source. Controllable states/constants.	
	DDS NCO	
	PID	
	Monitoring: oscilloscope, spectrum analyzer... Can also be used to process the signal in the CPU. Up to 12 channels.	
 	Split or combine In-phase and Quadrature components. Convert $\mathbb{R}$ to $\mathbb{C}$ or $\mathbb{C}$ to $\mathbb{R}$ .	

## 2 Webserver

The tunable parameters of the IPs are controlled through C coded functions, visible in the /oscimpDigital/lib/my\_lib.h library files. Example of functions with a generic driver "fpgagen":

```
fpgagen_send_conf(char *filename, int reg, int value);  
fpgagen_rcv_conf(char *filename, int reg, int *value);
```

Those functions can be implemented in a graphic interface to constitute a user friendly control of the IPs. Here we show an example of webserver using RemI<sup>1</sup>, a cross platform remote gui for python. The wrapper liboscimp\_fpga.py makes the intermediary between the webserver and the libraries. It takes the following form:

```
import ctypes  
from ctypes import *  
lib = ctypes.CDLL('/usr/lib/liboscimp_fpga.so')  
  
def fpgagen_send_conf(filename, reg, value):  
    file = ctypes.create_string_buffer(str.encode(filename))  
    my_val = int(value)  
    lib.fpgagen_send_conf(file, reg, my_val)  
  
def fpgagen_rcv_conf(filename, reg):  
    file = ctypes.create_string_buffer(str.encode(filename))  
    my_val = c_int()  
    ret_val = lib.fpgagen_rcv_conf(file, reg, byref(my_val))  
    return (ret_val, my_val.value)
```

Then the example of functions implemented in the webserver to configure the IPs is:

```
import liboscimp_fpga  
liboscimp_fpga.fpgagen_send_conf("/dev/my_file", my_reg, my_value)
```

In the webserver, values sent to the IPs can either take the form of slider, a spinbox, a checkbox, a button... In our case, a simple actuator will be represented by a checkbox, and any other controllable value by both a slider and a spinbox. The structure of the webserver is as follows:

```
#!/usr/bin/env python  
  
import liboscimp_fpga  
import ctypes  
import remi.gui as gui  
from remi import start, App  
  
class MyApp(App):
```

---

<sup>1</sup>Download and Faq : <https://www.remigui.com/>

```

def __init__(self, *args):
    super(MyApp, self).__init__(*args)

def main(self):
    self.w = gui.VBox()

    #Create the slider and the spinbox, whose value is restricted to -8192 to 8191 (no overflow)
    self.hbox_MY_VALUE = gui.HBox(margin="10px")
    self.lb_MY_VALUE = gui.Label("/dev/MY_VALUE_FILE", width="20%", margin="50px")
    self.sd_MY_VALUE = gui.Slider(0, -8192, 8191, 1, width="60%", margin="10px")
    self.sd_MY_VALUE.set_oninput_listener(self.sd_MY_VALUE.changed)
    self.sb_MY_VALUE = gui.SpinBox(0, -8192, 8191, 1, width="20%", margin="10px")
    self.sb_MY_VALUE.set_on_change_listener(self.sb_MY_VALUE.changed)
    self.sd_MY_VALUE.changed(self.sd_MY_VALUE, self.sd_MY_VALUE.get_value())
    self.hbox_MY_VALUE.append(self.lb_MY_VALUE)
    self.hbox_MY_VALUE.append(self.sd_MY_VALUE)
    self.hbox_MY_VALUE.append(self.sb_MY_VALUE)
    self.w.append(self.hbox_MY_VALUE)

    #Create the checkbox
    self.hbox_MY_ACTUATOR = gui.HBox(margin="10px")
    self.lb_MY_ACTUATOR = gui.Label("/dev/MY_ACTUATOR_FILE", width="20%", margin="50→
        ↪px")
    self.cb_MY_ACTUATOR = gui.CheckBox(True, width="5%", margin="10px")
    self.cb_MY_ACTUATOR.set_on_change_listener(self.cb_MY_ACTUATOR.changed)
    self.hbox_MY_ACTUATOR.append(self.lb_MY_ACTUATOR)
    self.hbox_MY_ACTUATOR.append(self.cb_MY_ACTUATOR)
    self.w.append(self.hbox_MY_ACTUATOR)

    return self.w

#Function called by the slider
def sd_MY_VALUE_changed(self, widget, value):
    print("/dev/MY_VALUE_FILE", MY_REG, int(value))
    liboscimp_fpga.fpgagen_send_conf("/dev/MY_VALUE_FILE", MY_REG, int(value))
    self.sb_MY_VALUE.set_value(int(value))

#Function called by the spinbox
def sb_MY_VALUE_changed(self, widget, value):
    print("/dev/MY_VALUE_FILE", MY_REG, int(value))
    liboscimp_fpga.fpgagen_send_conf("/dev/MY_VALUE_FILE", MY_REG, int(value))
    self.sd_MY_VALUE.set_value(int(float(value)))

#Function called by the checkbox
def sb_MY_ACTUATOR_changed(self, widget, value):
    print("/dev/MY_ACTUATOR_FILE", MY_REG, int(value))
    liboscimp_fpga.fpgagen_send_conf("/dev/MY_ACTUATOR_FILE", MY_REG2, int(value))
    self.sd_adc1_offset.set_value(int(float(value)))

#Launch oh the webserver on the local machine
start(MyApp, address="0.0.0.0", port=80, title="My_super_webserver")

```

Preview of the webserver created:

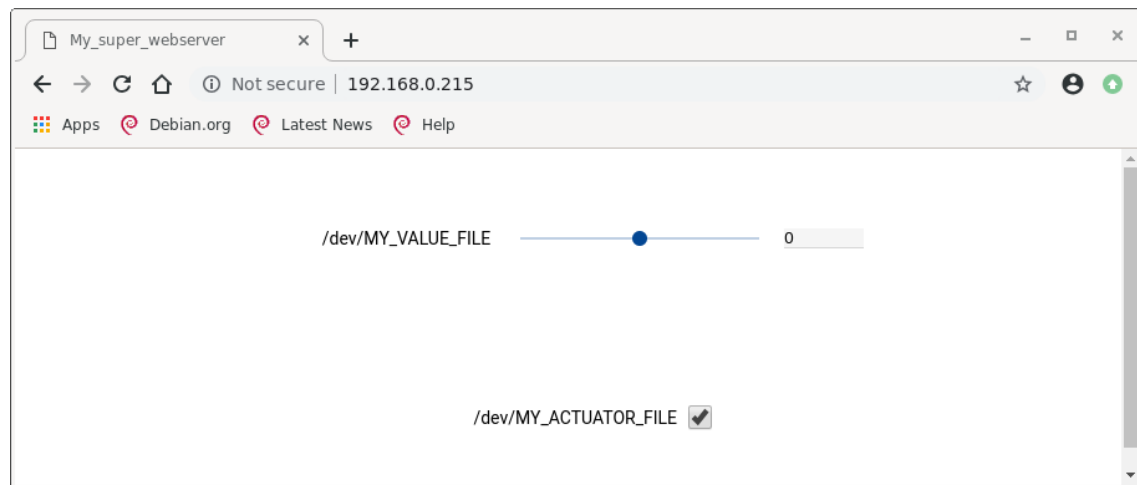


Figure 2: Example of webserver with MY\_VALUE\_FILE represented by a slider and a spinbox, and MY\_ACTUATOR\_FILE represented by a checkbox.

Here the generic driver `fpgagen` is used as an example, however the use of a different driver can lead to various requirements: arguments, data type, or several functions. Some specific cases will be treated in the next sections. In the other cases, refer to the `/oscimpDigital/lib/my.lib.h` library files, or the `oscimpDigital` documentation: <https://github.com/oscimp/oscimpDigital/tree/master/doc>

A webserver generator is available in the `/oscimpDigital/app/tools/webserver_generator` repository, and build a standard webserver using the same `My_super_design.xml` file than the one used by the module generator :

```
./webserver_generator.py My_super_design.xml
```

### 3 Double voltage source

A double tunable voltage source can be set up si using the `axi_to_dac` IP. However it's only one of the many functions that can be imagined with this IP.

See [https://github.com/oscimp/oscimpDigital/blob/master/doc/IP/axi\\_to\\_dac.md](https://github.com/oscimp/oscimpDigital/blob/master/doc/IP/axi_to_dac.md). The block diagram is as follows :

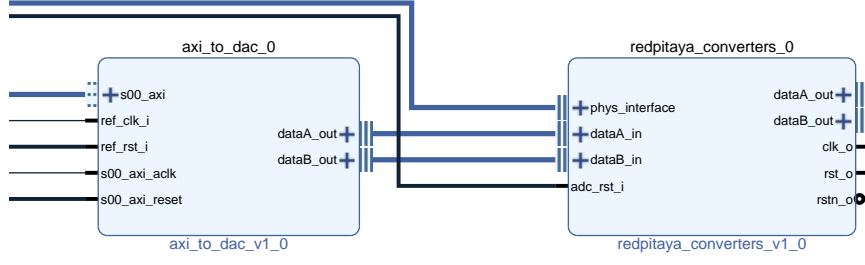
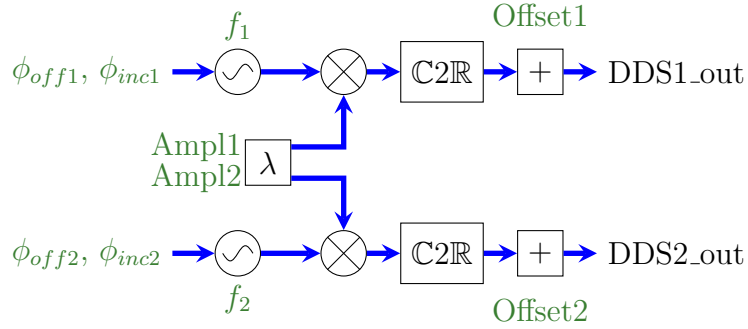


Figure 3: Part of the block diagram for the tunable voltage source.

In this configuration and with data on 14 *bits*, all the parameters of the axi\_to\_dac block keep their default value. The tuning of the output is performed with the webserver (see section 2). With the Redpitaya board, the maximum voltage is  $\pm 1$  V per output.

## 4 Double DDS

The schematic configuration of the double DDS we propose here is shown below:



It corresponds to two DDS with adjustable frequency, amplitude, output offset, and referenced on the same clock. Frequency/phase and amplitude modulation are not represented here, but can be added using sections 5 and 7. The block diagram associated with this scheme is as follows:

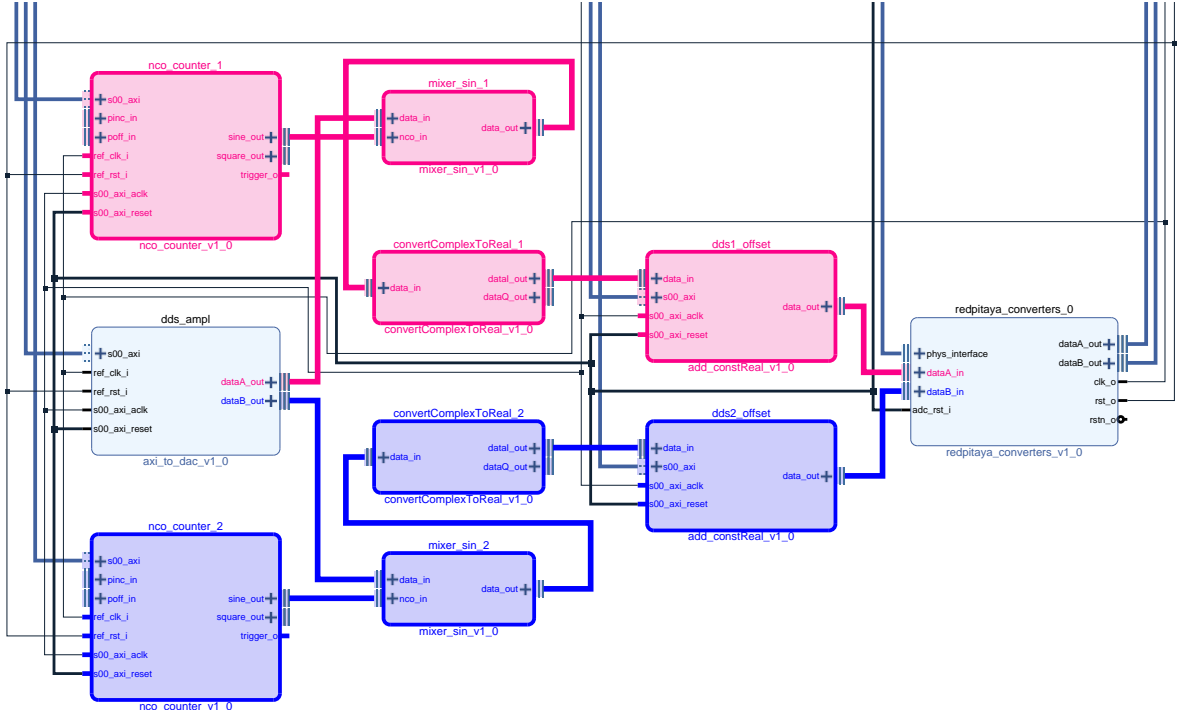


Figure 4: Part of the block diagram for the double DDS.

## 4.1 IP configuration

The IPs configuration may change depending on the board/application, however in this example we used the following configurations:

IP	Configuration
nco_counter_1/2	Counter size: 40 <i>bits</i> Data size: 16 <i>bits</i> Lut size: 12 <i>bits</i>
dds_ampl	Data size: 14 <i>bits</i>
mixer_sin_1/2	Data in/out size: 14 <i>bits</i> Nco_size: 16 <i>bits</i>
dds1/2_offset	Data in size: 14 <i>bits</i> Data out size: 14 <i>bits</i> Signed
convertCtoR_1/2	Data size: 14 <i>bits</i>



## 4.2 Webserver configuration

For the DDS offset and amplitude blocks we only use one value to be controlled, ie. one slider/spinbox. However the NCO block offers several options as the phase offset and increment can be internal or external to the block. Therefore we keep a default construction for the NCO in the webserver, including all these possibilities:

- 1<sup>st</sup> slider+spinbox: the frequency control, up to the half clock frequency (Hz)
- 2<sup>nd</sup> slider+spinbox: the phase offset control
- pinc checkbox: internal or external phase increment
- poff checkbox: internal or external phase increment

In the present case there is no external connections for the frequency and phase increments, thus the pinc and poff checkbox will remain checked. A preview of the webserver for the double DDS design is given in fig.5.



Figure 5: Screenshot of the double DDS webserver.

## 4.3 Expected output

We show in fig.6 an example of two signals generated.

Ch1:  $f_0 = 30 \text{ MHz}$ ,  $dds1\_ampl = 8191 \text{ arb. unit}$ ,  $dds1\_offset = 0 \text{ arb. unit}$ ,

$\phi_{off1} = 0 \text{ arb. unit}$ .

Ch2:  $f_0 = 45 \text{ MHz}$ ,  $dds2\_ampl = 3000 \text{ arb. unit}$ ,  $dds2\_offset = 5000 \text{ arb. unit}$ ,

$\phi_{off2} = 0 \text{ arb. unit}$ .

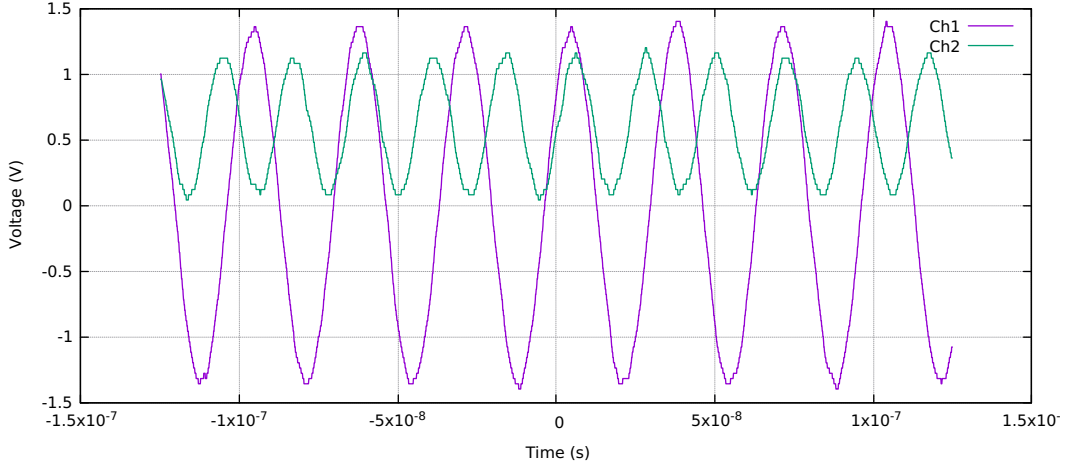


Figure 6: Expected output.

With an internal phase increment, the output signals are generated with an arbitrary phase. This phase can be adjusted according to the intended application, using the phase offset slider, as represented in fig.7:

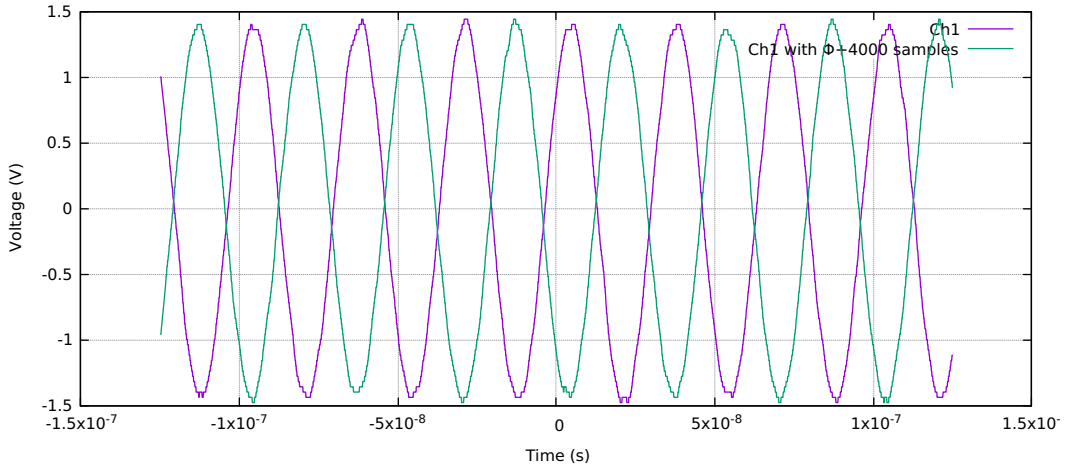


Figure 7: Phase offset.

#### 4.4 Unexpected output

In fig.8 the Ch1 signal is the same, but there is an overflow in Ch2 due to the sum of  $dds1\_ampl$  and  $dds1\_offset$ .

Ch2:  $f_0 = 45 \text{ MHz}$ ,  $dds2\_ampl = 8191 \text{ arb. unit}$ ,  $dds2\_offset = 8191 \text{ arb. unit}$ ,  $\phi_{off2} = 0 \text{ arb. unit}$ .

Solution: decrease either  $dds1\_ampl$  or  $dds1\_offset$ , such as  $dds1\_ampl + dds1\_offset < 8191$ .

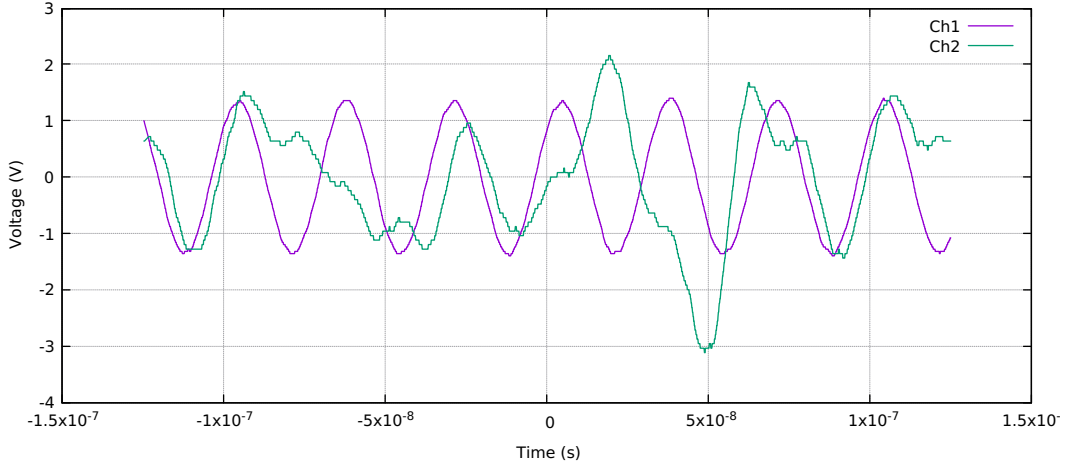
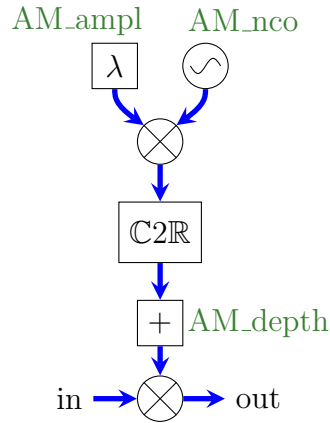


Figure 8: Unexpected output due to overflow in ch2.

## 5 Amplitude modulation

An amplitude modulation can be performed easily, in the same way as with RF components:



This scheme is equivalent to the expression of the amplitude modulation:

$$y(t) = [1 + h \cos(\omega_m t)]z(t)$$

$$\Rightarrow out = [1 + \frac{AM\_ampl}{AM\_depth} AM\_nco] AM\_depth \times in$$

Then the modulation depth is  $h = \frac{AM\_ampl}{AM\_depth}$ . Thereafter:

- $h = 0$  with  $AM\_ampl = 0$
- $h = 0.5$  with  $AM\_ampl = 4096 \text{ arb. unit}$  and  $AM\_depth = 8191 \text{ arb. unit}$
- $h = 1$  with  $AM\_ampl = 8191 \text{ arb. unit}$  and  $AM\_depth = 8191 \text{ arb. unit}$
- $h = 2$  with  $AM\_ampl = 8191 \text{ arb. unit}$  and  $AM\_depth = 4096 \text{ arb. unit}$

The block diagram corresponding to this scheme is as follows:

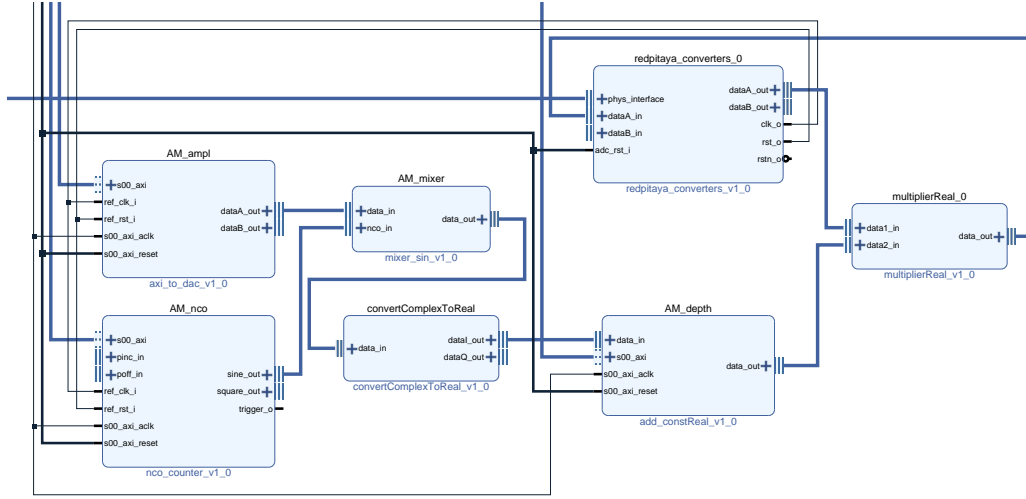


Figure 9: Part of the block diagram for this amplitude modulation.

In this block diagram the input and output are connected to the converter blocks to make an example with external signal. However this principle can be included to other applications, such as the double dds design to add an amplitude modulation option to the generated signals.

## 5.1 IP configuration

IPs configuration in this example:

IP	Configuration
AM_nco	Counter size: 40 <i>bits</i> Data size: 16 <i>bits</i> Lut size: 12 <i>bits</i>
AM_ampl	Data size: 16 <i>bits</i>
AM_mixer	Data in/out size: 16 <i>bits</i> Nco_size: 16 <i>bits</i>
multiplierReal	Input data1 size: 14 <i>bits</i> Input data2 size: 16 <i>bits</i> Output data size: 14 <i>bits</i>
AM_depth	Data in size: 16 <i>bits</i> Data out size: 16 <i>bits</i> Format: Signed
convertCtoR_1	Data size: 16 <i>bits</i>

## 5.2 Webserver configuration

Here the webserver configuration is similar to the double dds one. In case refer to subsection 4.2. preview of the amplitude modulation part of the webserver, with the mod\_nco

controlling the modulation frequency, and mod\_ampl controlling its amplitude:

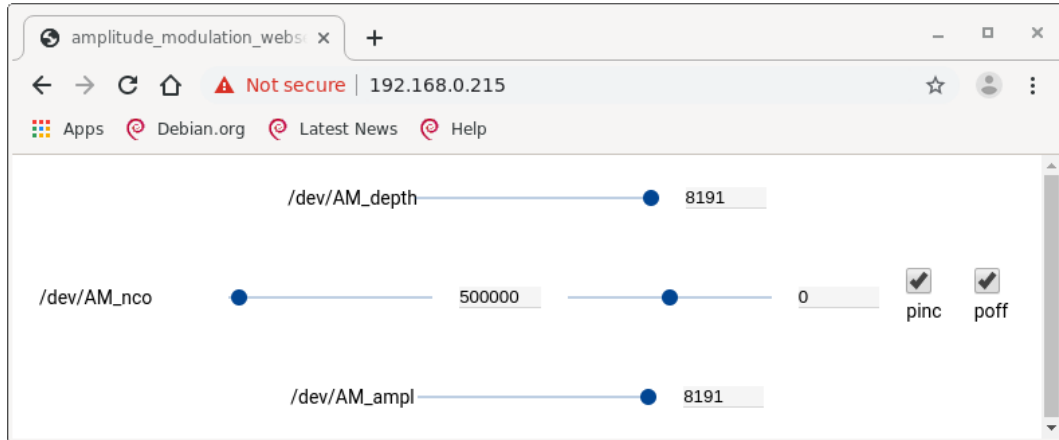


Figure 10: Screenshot of the amplitude modulation part of the webserver.

### 5.3 Expected output

To make a small preview of the expected behavior of the amplitude modulation, we will present the cases  $h = 0.5$ ,  $h = 1$ , and the surmodulation  $h = 2$ . We use at the input a sine signal of  $50\text{ MHz}$  and  $0\text{ dBm}$ . The modulating signal is set to  $500\text{ kHz}$ .

Input and output with  $h = 0.5$

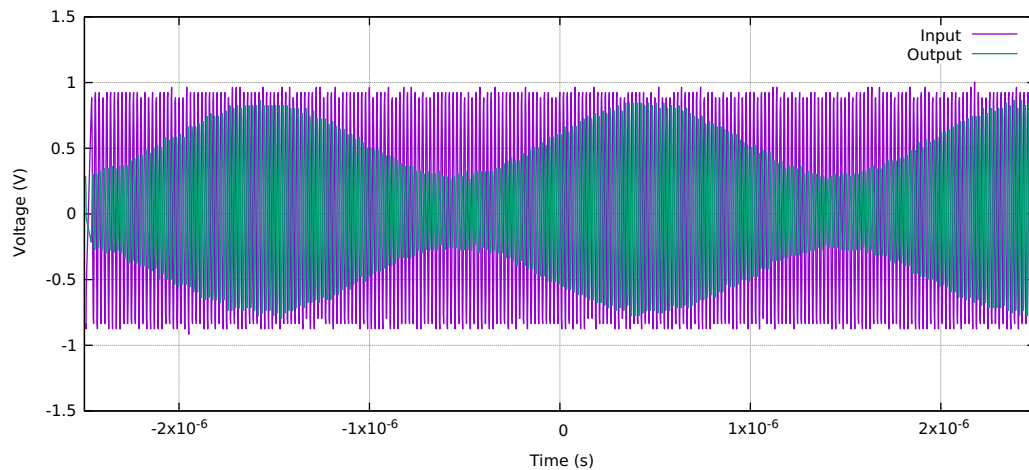


Figure 11: Expected behavior for  $h = 0.5$ .

Output with  $h = 1$

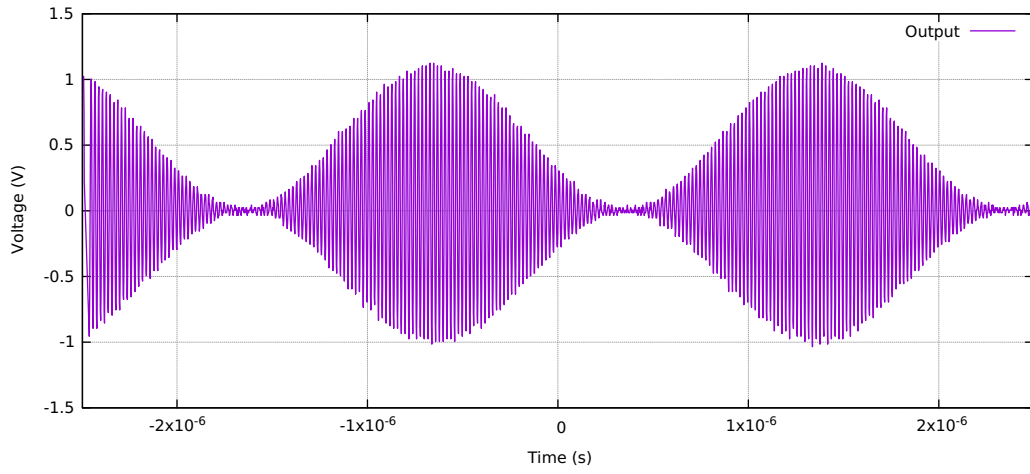


Figure 12: Expected behavior for  $h = 1$ .

Output with  $h = 2$  (overmodulation)

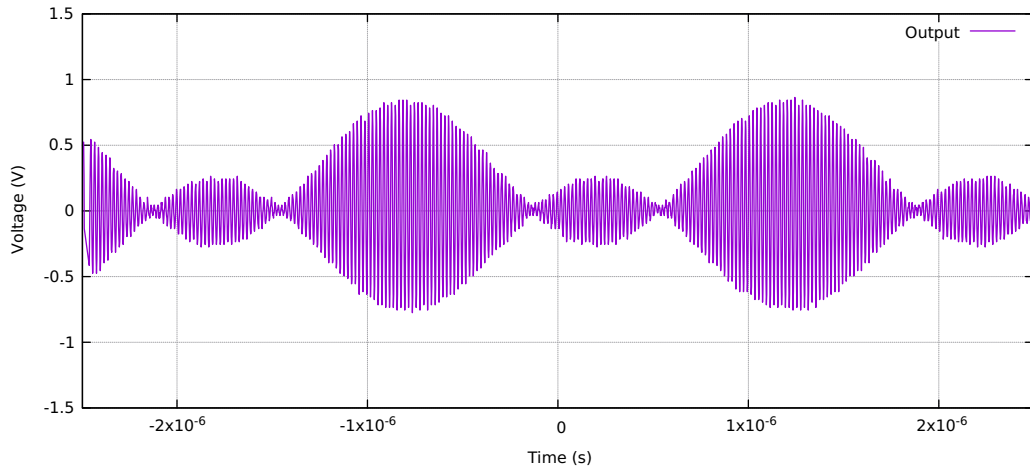


Figure 13: Expected behavior for  $h = 2$ : overmodulation.

## 5.4 Unexpected output

Although the following case is not due to the numerical aspect of the amplitude modulation presented here, it can be seen as an unexpected output:

This kind of shape is due to a carrier frequency below the input signal frequency.

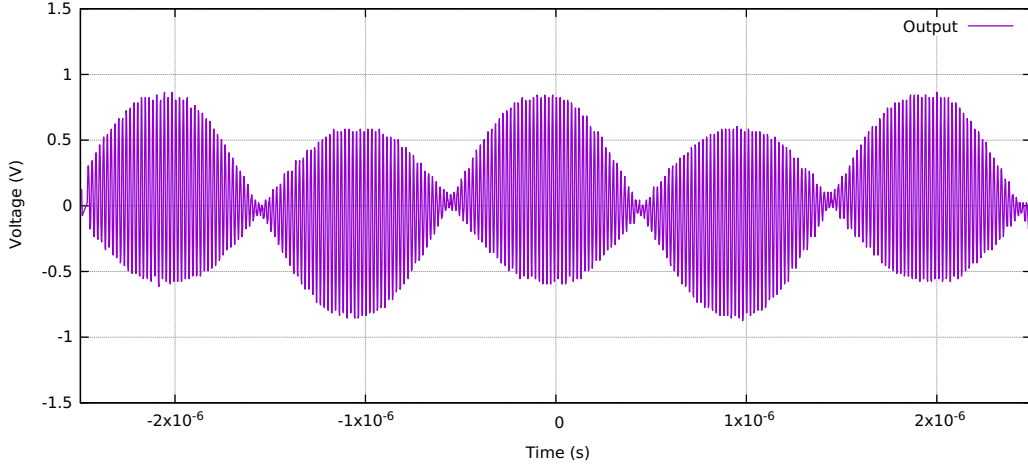
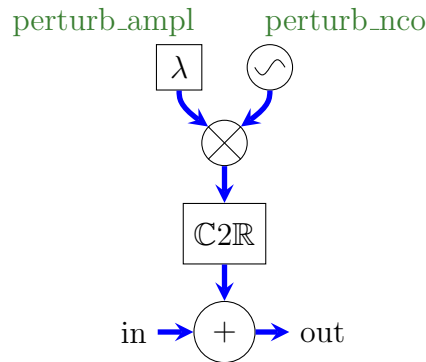


Figure 14: Case of a carrier frequency below the input signal frequency.

## 6 Sine perturbation of a signal

A way to study the response of a system is to apply a perturbation to this system. Most of the time, the perturbations take the form of steps applied to the input signal of the system by adding to it a square signal. In the case of optical oscillators, the working point of the system, i.e. the resonant frequency, can be determined by making a frequency scan of the system. This scan takes the form of a sine perturbation of the system. In this section, we show the example of a sine perturbation to a signal. The scheme used is quite similar to the amplitude modulation:



The block diagram corresponding to this scheme is as follows:

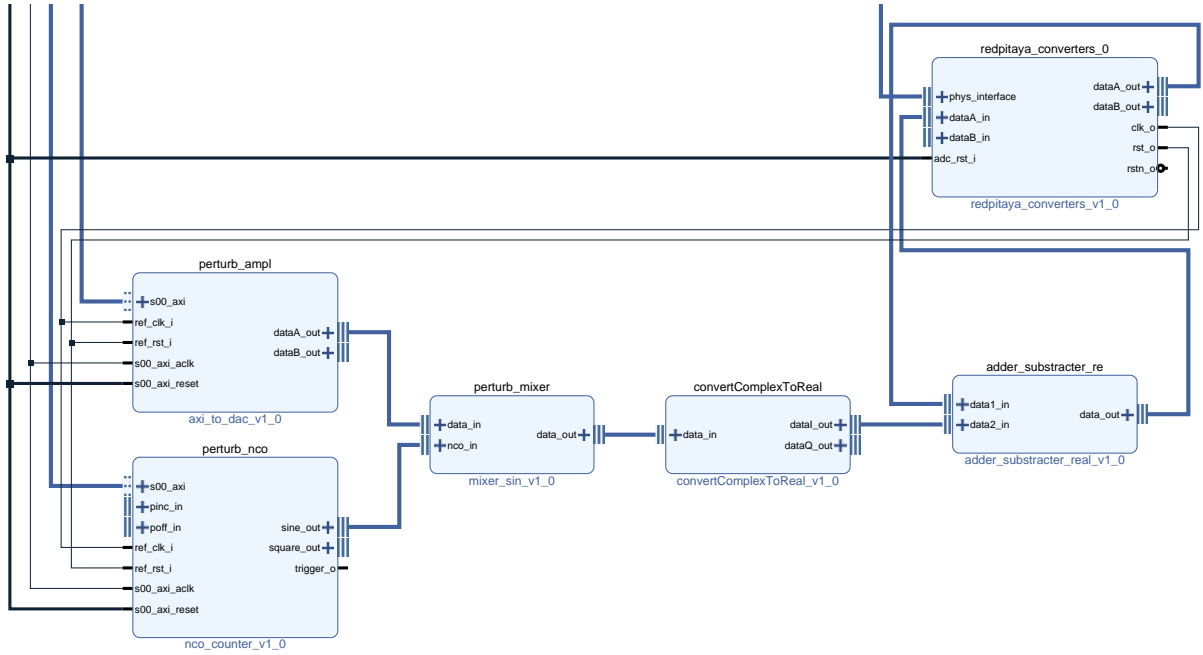


Figure 15: Part of the block diagram for the amplitude modulation.

## 6.1 IP configuration

IPs configuration in this example:

IP	Configuration
perturb_nco	Counter size: 40 <i>bits</i> Data size: 16 <i>bits</i> Lut size: 12 <i>bits</i>
perturb_ampl	Data size: 14 <i>bits</i>
perturb_mixer	Data in/out size: 14 <i>bits</i> Nco_size: 16 <i>bits</i>
adder_substracter_re	Data size: 14 <i>bits</i> Operation: add Format: Signed
convertCtoR_1	Data size: 14 <i>bits</i>



## 6.2 Webserver configuration

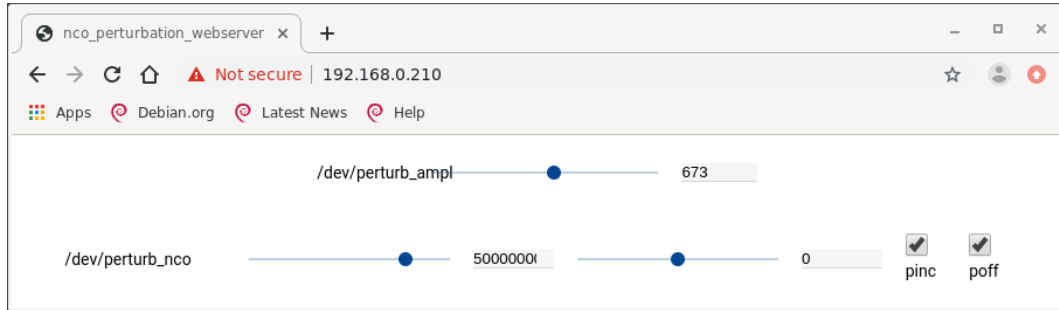


Figure 16: Screenshot of the sine perturbation part of the webserver.

## 6.3 Expected output

We use at the input a sine signal of 5  $MHz$  and 0  $dBm$ . With a sine perturbation of 50  $MHz$  and 1000 *arb. unit*, we expect:

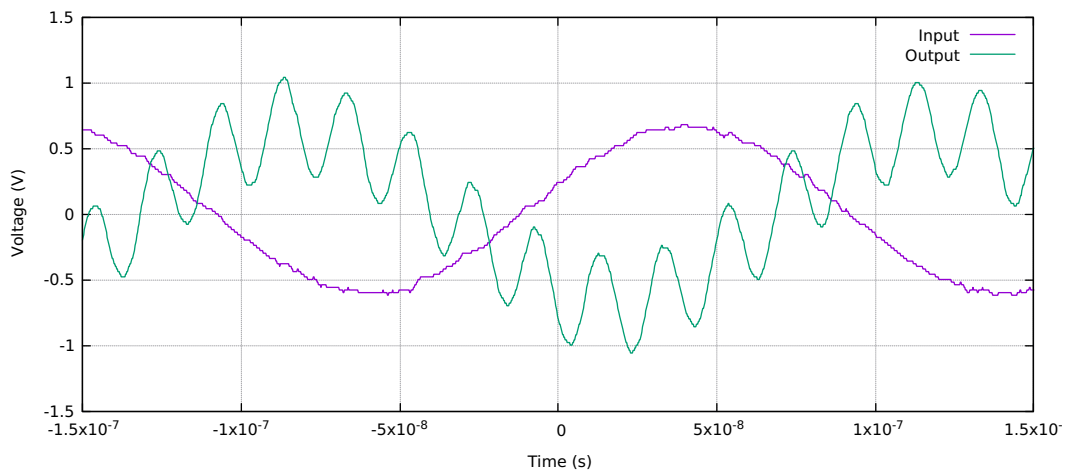


Figure 17: Expected behavior with input sine at 5  $MHz$  and sine perturbation of 50  $MHz$ .

## 6.4 Unexpected output

An unexpected situation can be the output signal visible in fig.18:

This situation is very similar to the fig.8 in subsection 4.4: this output is a result of an overflow happening during the signal processing. This situation is due to an input signal too powerful with respect to the perturbation amplitude. Solution: decrease the perturbation amplitude or the power of the input signal.

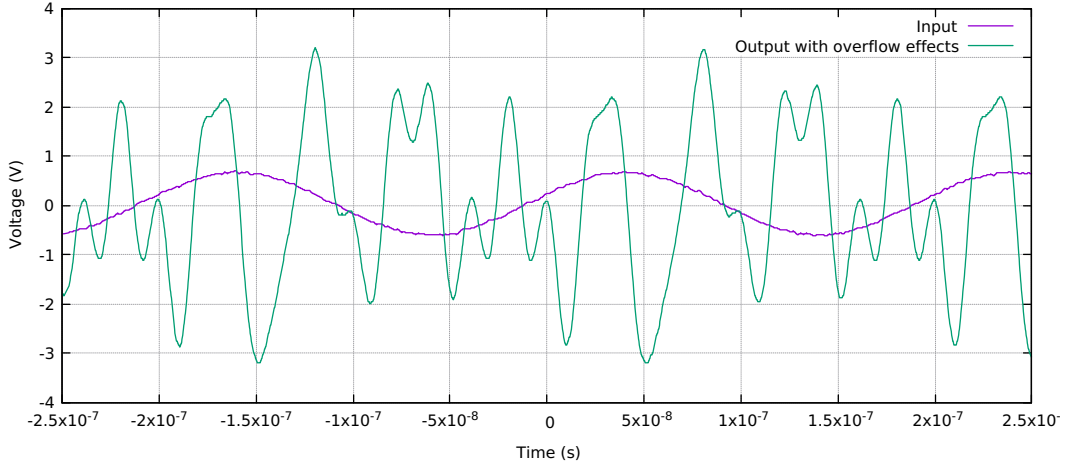
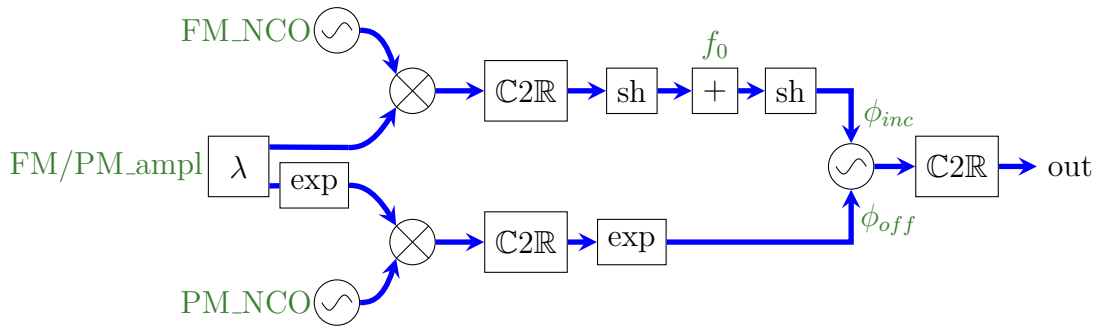


Figure 18: Unexpected behavior with input sine at 5  $MHz$  and sine perturbation of 50  $MHz$ .

## 7 Frequency and phase modulation of a NCO

A frequency and a phase modulation can also be performed using the phase increment and the phase offset input of the NCO. The scheme presented below corresponds to the block diagram presented in fig.19.



### 7.1 IP configuration

IPs configuration in this example:

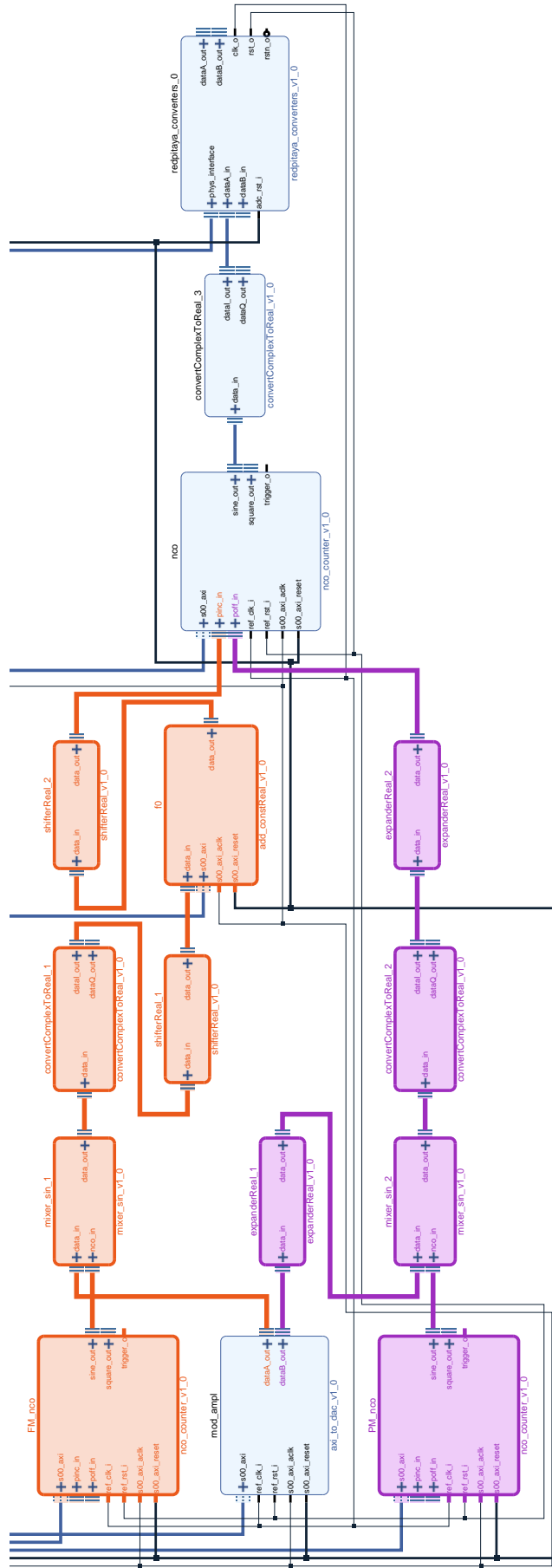


Figure 19: Part of the block diagram for frequency (orange) and phase (purple) modulation of a NCO.

IP	Configuration
FP_nco/PM_nco/nco	Counter size: 40 <i>bits</i> Data size: 16 <i>bits</i> Lut size: 12 <i>bits</i>
mod_ampl	Data size: 27 <i>bits</i>
expanderReal_1	Data in size: 27 <i>bits</i> Data in size: 16 <i>bits</i> Format: Signed
expanderReal_2	Data in size: 16 <i>bits</i> Data in size: 12 <i>bits</i> Format: Signed
shiftReal_1	Data in size: 27 <i>bits</i> data out size: 32 <i>bits</i>
shiftReal_2	Data in size: 32 <i>bits</i> data out size: 40 <i>bits</i>
mixer_sin_1	Data in/out size: 27 <i>bits</i> Nco_size: 16 <i>bits</i>
mixer_sin_2	Data in/out size: 16 <i>bits</i> Nco_size: 16 <i>bits</i>
$f_0$	Data in/out size: 32 <i>bits</i> Format: Signed
convertCtoR_1	Data size: 27 <i>bits</i>
convertCtoR_2	Data size: 16 <i>bits</i>
convertCtoR_3	Data size: 14 <i>bits</i>

## 7.2 Webserver configuration

Here, only the PM\_deviation is reconfigured between  $-8192$  and  $8191$  *arb. unit*. Both FM\_nco, PM\_nco,  $f_0$ , FM\_deviation and nco are configured between 0 and  $62000000$  *Hz*. The webserver is represented in fig.20

## 7.3 Expected frequency modulation

For the frequency modulation, we use the phase increment input of the nco. In the webserver this requires to uncheck the "pinc" checkbox. This means that the phase increment is external and whatever is the frequency mentioned in /dev/nco, it will not be taken into consideration. In this case, the frequency of the carrier must be written in /dev/f0: this adds to the FM\_nco a constant corresponding to the carrier frequency.

To show an exaggerated example of the frequency modulation, we use a carrier frequency  $f_0$  of  $5$  *MHz*. The frequency of the FM\_nco and the FM\_deviation must remain lower than

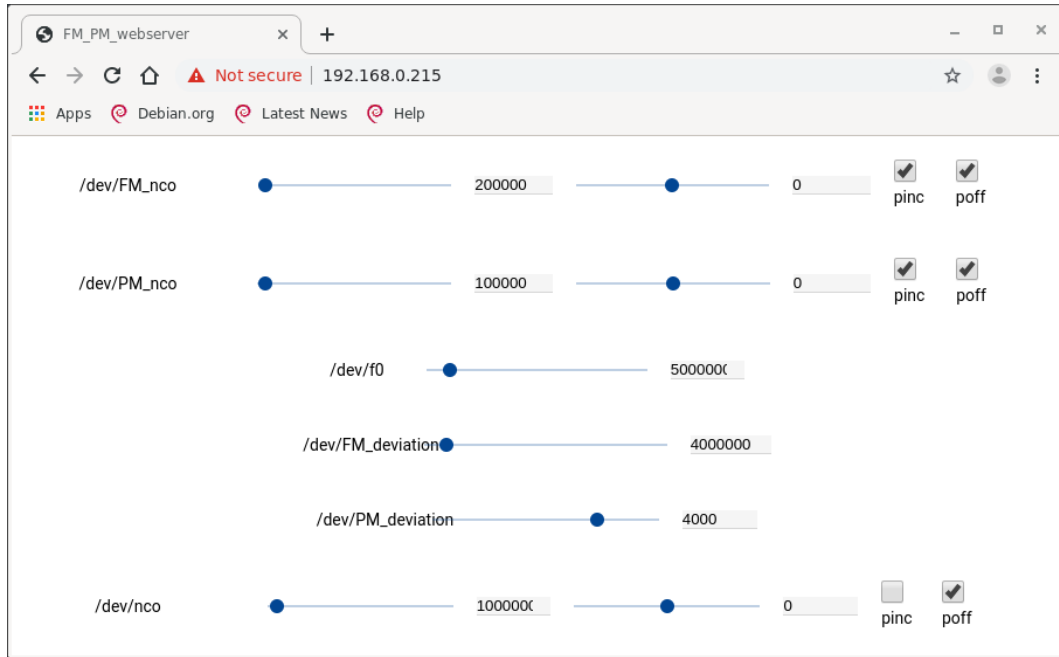


Figure 20: Screenshot of the phase and frequency modulation webserver. "pinc" checkbox is unchecked: frequency modulation.

the carrier frequency. Here we used a frequency modulation of  $200\text{ kHz}$  and a deviation of  $4\text{ MHz}$ . We expect the following output:

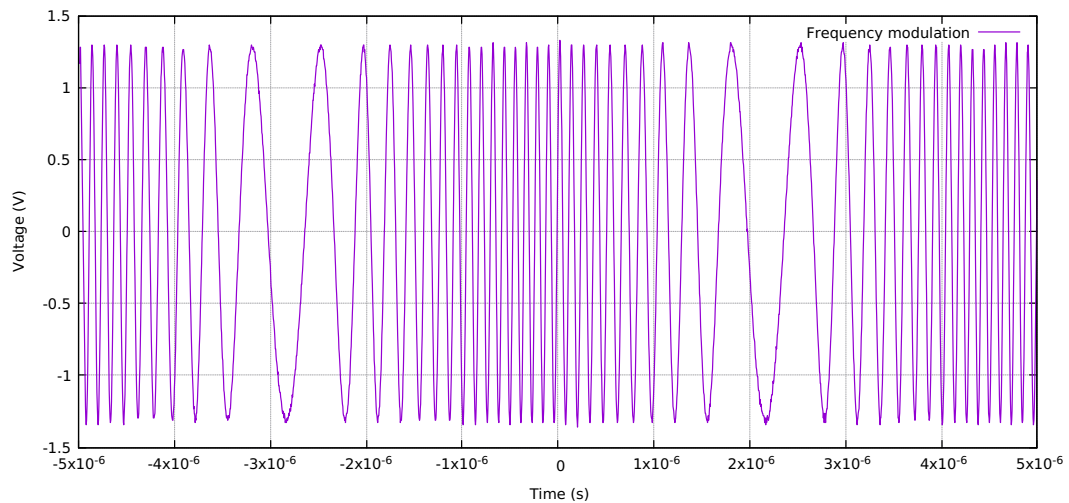


Figure 21: Expected frequency modulation with a carrier at  $5\text{ MHz}$ , and a deviation of  $4\text{ MHz}$ .

## 7.4 Expected phase modulation

For the phase modulation, we use the phase offset input of the nco and keep an internal phase increment. In the webserver this requires to uncheck the "poff" checkbox and keep checked the "pinc" checkbox. This means this time that the phase offset is external, and the nco frequency taken into consideration is the one mentionned in /dev/nco. The phase offset range goes from  $-4\pi$  to  $4\pi$ , therefore a PM.deviation of  $\pm 8191 \text{ arb. unit}$  corresponds approximately to deviation of  $\pm 4\pi$ .

The example in fig.22 represents a phase modulation for a carrier at  $1 \text{ MHz}$ , a deviation slightly below  $2\pi$  (PM.deviation =  $4000 \text{ arb. unit}$ ), and a modulation frequency of  $100 \text{ kHz}$ .

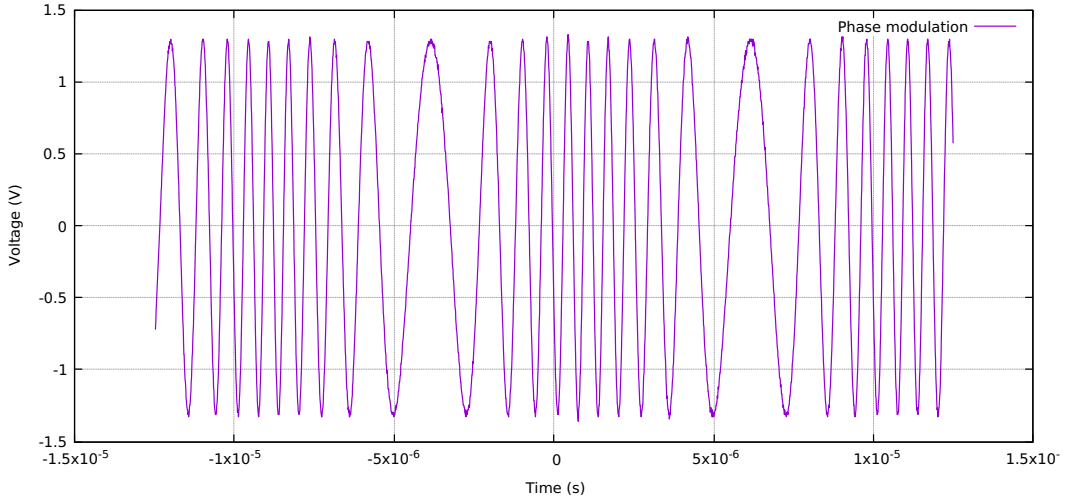


Figure 22: Expected phase modulation with a carrier at  $1 \text{ MHz}$ , and a deviation of  $2\pi$ .

## 7.5 Unexpected output

Unexpected outputs or a lack of signal can have several origins. Here is a non-exhaustive list of unexpected situations, and the potential solutions:

- The observed deviation is higher or lower than expected:

*The data sizes in the expanders/shifters in the block design are chosen so that the range of the mod\_ampl block fit with the phase increment/offset inputs of the nco. If the deviation is not the one expected, check if the data sizes between the mod\_ampl block and the nco block are adapted.*

- There is no output signal or the signal is fixed:
  1. Check the connections in the block design.

2. *Check if the data sizes are adapted.*
  3. *Check if the mod\_ampl block (axi\_to\_dac block) is configured to have a data output always enabled: its state must remain high.*
- There seems to be an amplitude modulation with the frequency/phase modulation, or the output amplitude seems to vary with the frequency.  
*The input impedance of your monitoring device may be quite low.*

## 8 Filtering

The FIR (Finite Impulse Response) filter has the role of filter with a number of coefficients that is configurable during the creation of a design. Similarly to the mean block that makes a moving average, the FIR has a decimation option. The decimation is performed before the filtering and slows the data flow. To learn more about the FIR, visit: <https://github.com/oscimp/oscimpDigital/tree/master/doc/tutorials/redpitaya/4-FIR>

### 8.1 Calculation of the coefficients

The number of coefficients and their values determine the transfer function of the FIR. Therefore the coefficient must be calculated according to the situation. Also, the coefficients must be integers with a size determined in the FIR block, by default and in our case on 16 *bits* including one bit dedicated to the sign.

To compute the FIR coefficients, octave proposes the signal package<sup>2</sup> with the function `fir1`<sup>3</sup> that allows to design either lowpass, highpass, bandpass or bandstop filters. First, load the signal package of octave:

```
octave:1> pkg load signal
```

Then the syntax is as follows:

```
COEFF = fir1(n, w, type)
```

With  $n$  the order of the filter, resulting in  $n + 1$  coefficients,  $w$  the normalized cutoff frequency(ies), and *type* the type of the filter "low", "high", "stop" or "pass".

For example, the calculation of 10 coefficients (9th order filter) on 16 *bits* ( $\times 2^{15}$ , without the bit dedicated to the sign) for a lowpass filter with a cutoff frequency at the half range (0.5), gives the following output:

```
octave:2> COEFF = transpose(int16(fir1(9,0.5)*2**15))
COEFF =
```

<sup>2</sup><https://octave.sourceforge.io/signal/index.html>

<sup>3</sup><https://octave.sourceforge.io/signal/function/fir1.html>

```

128
-397
-1337
3775
14214
14214
3775
-1337
-397
128

```

It should be noted that the FIR coefficients are recognizable by their symmetry. The function `freqz` of the signal package allows to obtain and visualize the frequency and phase responses of the designed FIR coefficients. Some examples are represented below:

#### Lowpass filter with 55 coefficients

Filter of 54<sup>th</sup> order. The cutoff frequency of 3 *MHz* is calculated on the basis of a sampling frequency of 125 *MHz*, i.e. a Nyquist frequency of 62.5 *MHz*. Then the cutoff is  $3/62.5 = 0.048$ . In the following, `freqz` has 4 arguments. The first is our FIR coefficients. The second represents the coefficients of feedback of an IIR (Infinite Impulse Response) filter, here the value 1 means we consider a simple FIR. The third (1024) is the resolution of the frequency/phase response diagram, and the last is the sampling frequency 125 *MHz*:

```
octave:2> figure(1); freqz(int16(fir1(54,0.048)*2**15),1,1024,125e6)
```

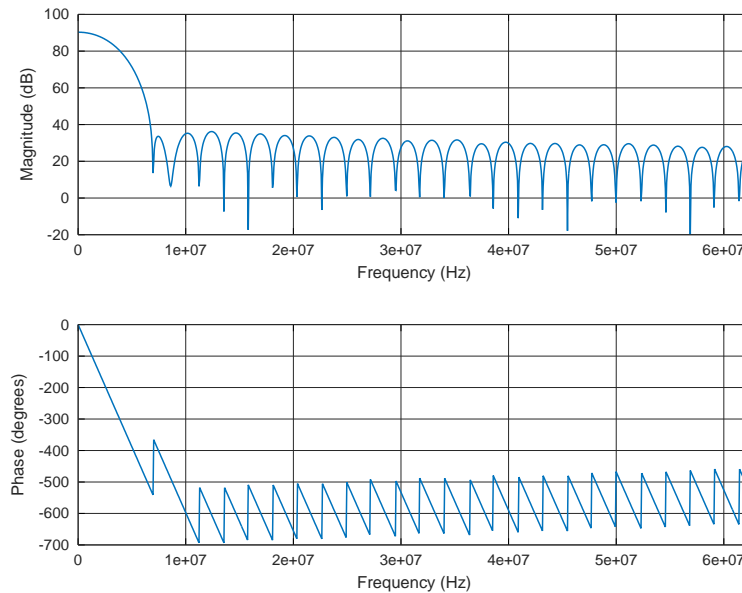


Figure 23: Lowpass filter at a 3 *MHz* cutoff frequency with 55 coefficients.



The importance of designing the filter coefficients adapted to one precise situation is clearly visible here, since the blocking range is not flat but constituted of small bounces. The rejection between the top and the bottom of the bounces can exceed  $20\text{ dB}$  depending on the FIR coefficients. Thus, it is interesting to design the FIR coefficients so that it rejects a precise spectral component (such as harmonics).

### Lowpass filter with 15 coefficients

The effect of the number of coefficient can be seen by designing the same fir with much less coefficients, here 15 coefficients:

```
octave:3> figure(2); freqz(int16(fir1(14,0.048)*2**15),1,1024,125e6)
```

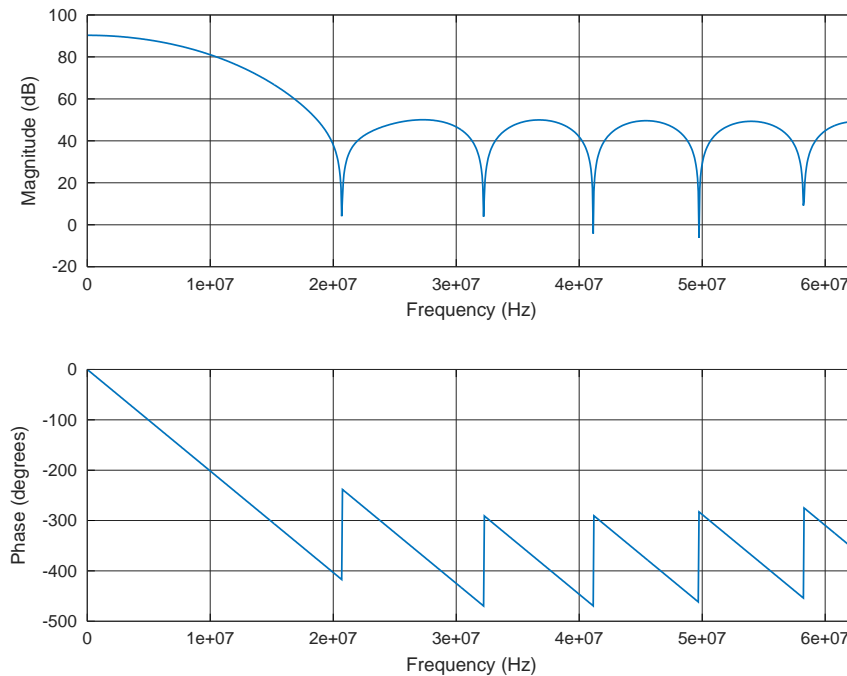


Figure 24: Lowpass filter at a  $3\text{ MHz}$  cutoff frequency with 15 coefficients.

In this case, the whole rejection of the filter is lower, and the filtering less precise.

### Dual bandstop filter

Example of syntax for the designing of a more exotic filter. Here a filter with two stop bands at  $5\text{ MHz} - 25\text{ MHz}$  and  $40\text{ MHz} - 50\text{ MHz}$ :

```
octave:4> figure(3); freqz(int16(fir1(54,[0.08 0.4 0.64 0.8], 'stop')*2**15),1,1024,125e6)
```

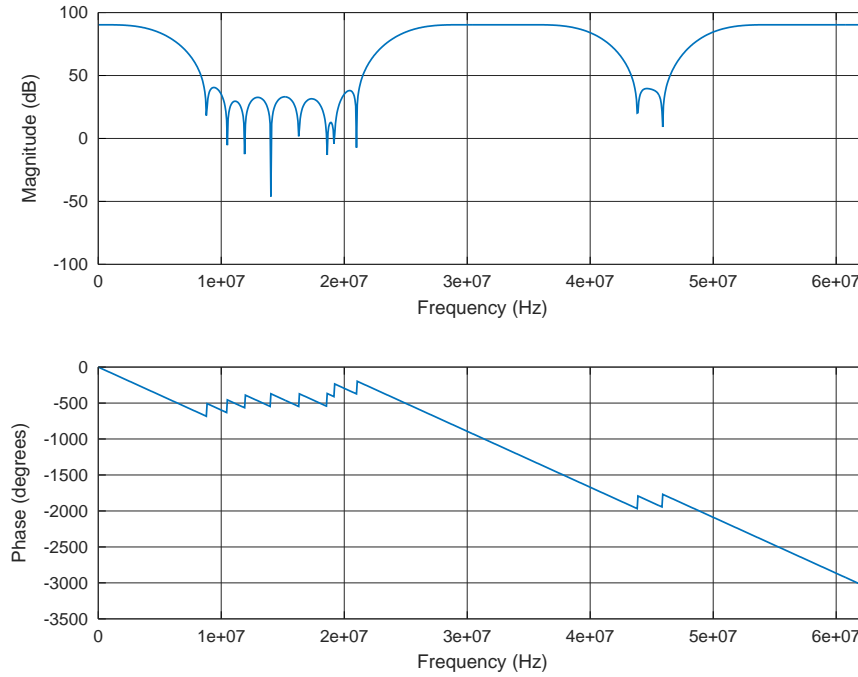


Figure 25: Bandstop filter at at  $5\text{ MHz} - 25\text{ MHz}$  and  $40\text{ MHz} - 50\text{ MHz}$ , with 55 coefficients.

## 8.2 Loading of the coefficients

The loading of the coefficients in the FIR is performed in our case using the function visible in the `/oscompDigital/lib/fir_conf.h` file :

```
fir_send_confSigned(const char *basename, const char *fileCoeff, const int coeffSize);
```

In the python wrapper `liboscomp-fpga.py` (see section 2), it takes the form:

```
def fir_send_confSigned(basename, nbFir, fileCoeff, coeffSize):
    file = ctypes.create_string_buffer(str.encode(basename))
    coeffFile = ctypes.create_string_buffer(str.encode(fileCoeff))
    lib.fir_send_confSigned(file, nbFir, coeffFile, coeffSize)
```

Where `coeffFile` is a data file where the FIR coefficients are stored in column, and `coeffSize` the number of coefficients mentioned in the FIR block and contained in `coeffFile`. Then an example of python script to load one FIR with  $N$  coefficients stored in the `My_FIR_coefficients.dat` file is :

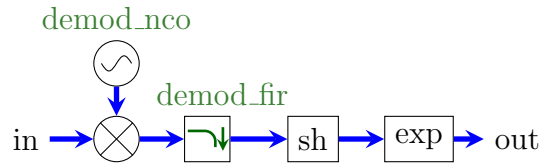
```
#!/usr/bin/env python
```

```
import libosimp_fpga
```

```
libosimp_fpga.fir_send_confSigned('/dev/MY_FIR_FILE', 'My_FIR_coefficients.dat', N)
```

## 9 Demodulation

A demodulation in amplitude or frequency/phase can be performed using the scheme below:



The FIR is configured as a lowpass filter, to reject mainly the sum frequency component after mixing. The block diagram corresponding to this scheme is presented in fig.26.

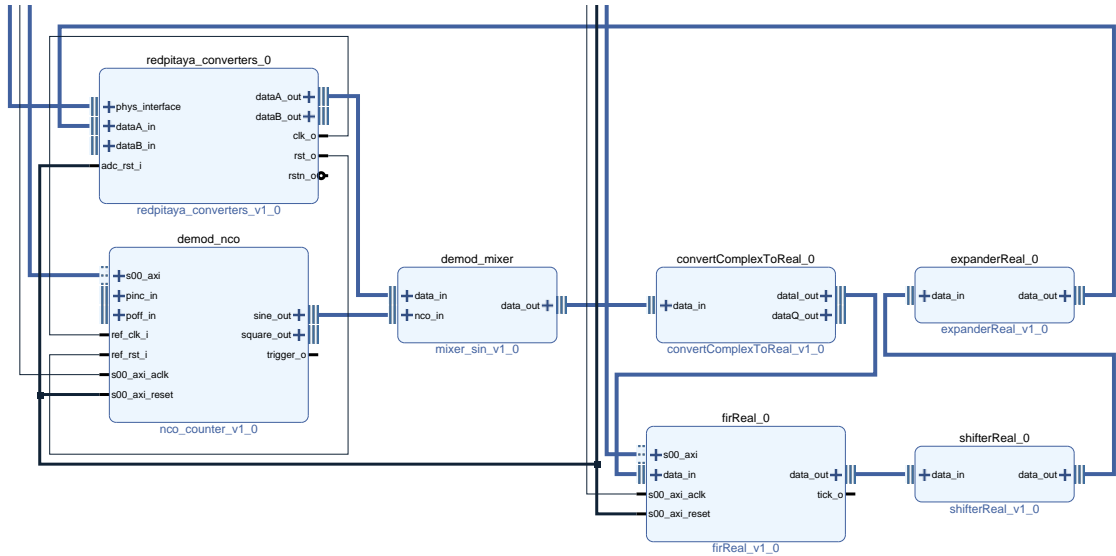


Figure 26: Part of the block diagram for the demodulation, including the filtering.

### 9.1 IP configuration

The IPs configuration in this example is given in the following table :

IP	Configuration
demod_nco	Counter size: 40 <i>bits</i> Data size: 16 <i>bits</i> Lut size: 12 <i>bits</i>
demod_mixer	Data in/out size: 14 <i>bits</i> Nco_size: 16 <i>bits</i>
convertCtoR_1	Data size: 14 <i>bits</i>
demod_fir	COEFF Size: 16 <i>bits</i> Data In size: 14 <i>bits</i> Data Out size: 32 <i>bits</i> Decim Factor: 1 Nb COEFF: 55
shiftReal_0	Data in size: 32 <i>bits</i> Data out size: 18 <i>bits</i>
expanderReal_0	Data in size: 18 <i>bits</i> Data out size: 14 <i>bits</i>

The FIR coefficients used in this section are the one calculated for the 55 coefficients lowpass filter given at the previous section. Then the data sizes assigned to the shifter and expander blocks, are chosen to maximize the range of the output signal for a maximum input amplitude. Those parameters must also be adapted depending on the FIR coefficients to avoid an overflow. In the case the block design must remain adapted to several situations, a dynamic shifter can be used instead of the succession shifter + expander. It allows choosing from the webserver the beginning of the 14 *bits* output word among the 32 output bits of the FIR, and therefore choosing the output range.

## 9.2 Webserver configuration

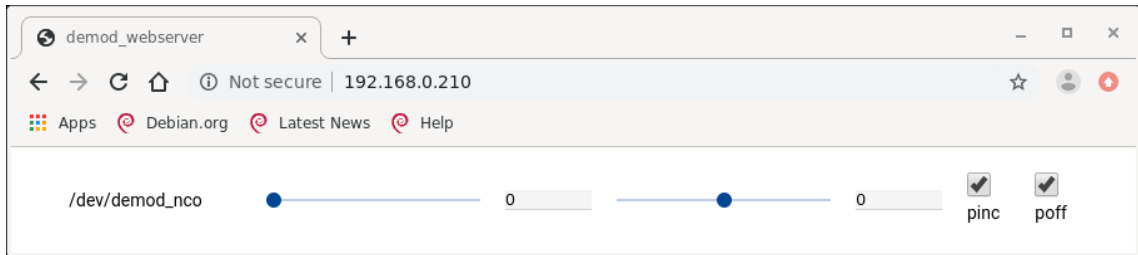


Figure 27: Screenshot of the demodulation webserver: only one nco is controlled here.

## 9.3 Expected output

To make small review of the behavior of this demodulation setup, we display here four cases:

- Mix of two sine signal (i.e. not a demodulation)
- Demodulation of an unmodulated sine signal
- Amplitude demodulation
- Frequency demodulation

### Mix of two sine signal

A first test that can be done with this setup, is the mixing of two sine signal. The beatnote of the two signals shows the range of the output signal. If the output signal is too weak or if there is an overflow, then the data sizes of the expander and shifter blocks must be changed.

### Demodulation of an unmodulated sine signal

The type of demodulation depends on the phase shift between the two signals. To show the effect of this phase shift on the demodulated signal, we show here the demodulation of an unmodulated sine signal. In this example the input is a simple sine signal at  $40\text{ MHz}$ , thus the demodulation frequency is set to  $40\text{ MHz}$ . The phase offset of the demodulation signal allows changing the phase shift between the two signals. Then the output signal is a direct signal whose amplitude depend on the phase shift between the two signals.

First, with a demodulation phase offset of 1550, the signals are **in phase**. Then the amplitude of the output signal is maximum:

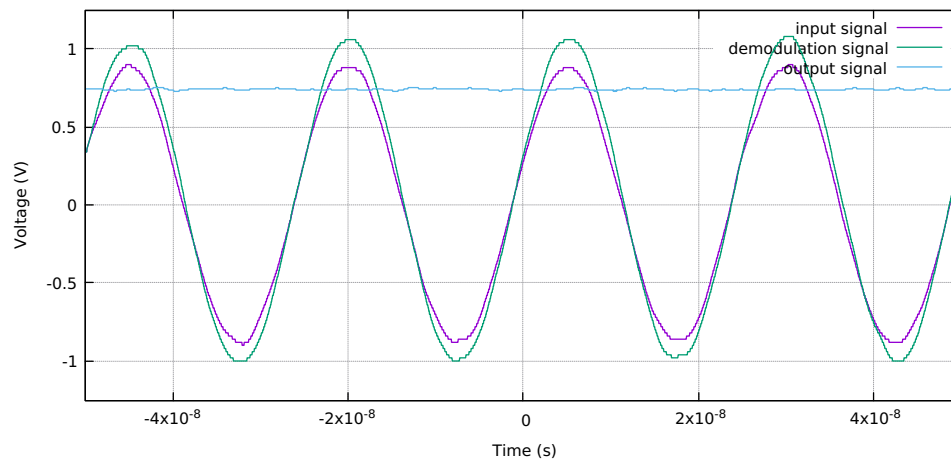


Figure 28: The carrier and the demodulation signal are in phase.

Secondly, with a demodulation phase offset of 3600, the signals are **in antiphase**. Then the amplitude of the output signal is minimum:

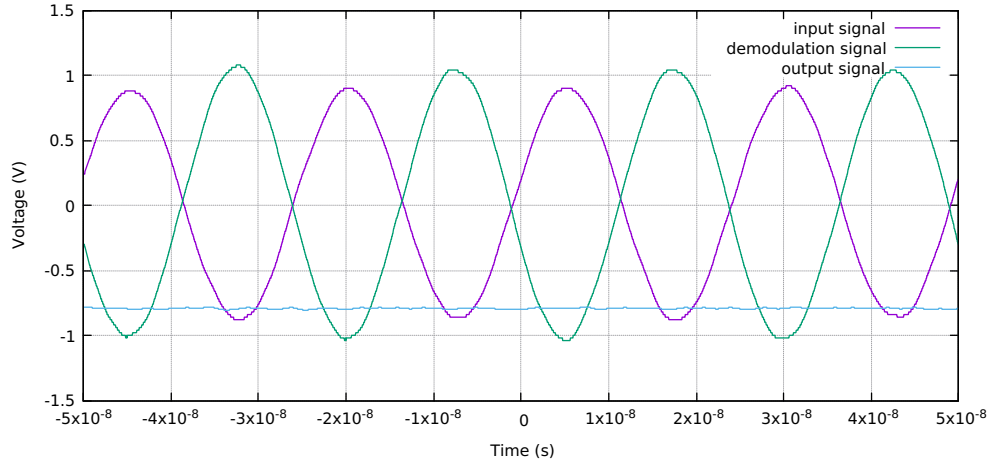


Figure 29: The carrier and the demodulation signal are in antiphase.

Finally, with a demodulation phase offset of 2500, the signals are **in quadrature**. Then the amplitude of the output signal is zero:

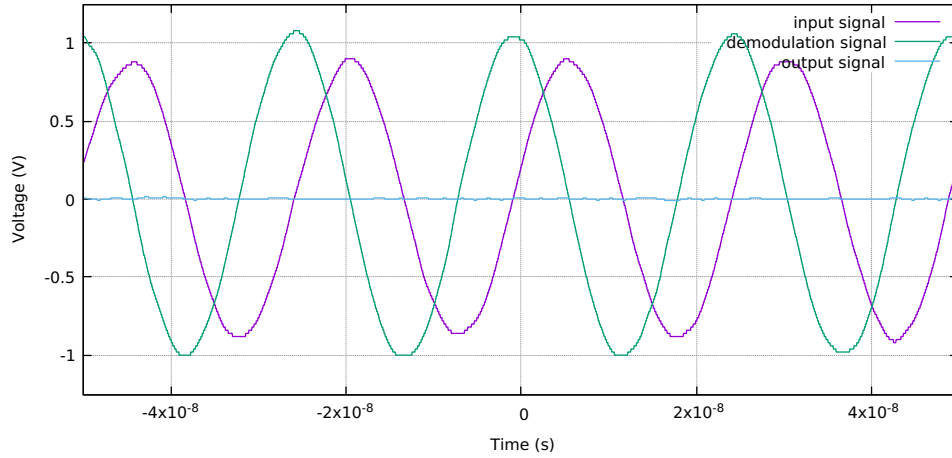


Figure 30: The carrier and the demodulation signal are in quadrature.

As a reminder, the phase offset allows a phase shift of  $\pm 4\pi$ , i.e. approximately 2048 *arb. unit* between the "in phase" and "in antiphase" situations. The phase offsets mentioned above allows verifying that. However those phase offsets does not always correspond to the situations described, since the nco has an arbitrary phase. Therefore, it may be adjusted each time the nco status is modified. This point also applies to the input oscillator.

### Amplitude demodulation

For the amplitude demodulation, the carrier and the demodulation signal must be in phase. Here we use a carrier at  $10\text{ MHz}$ , and the modulating signal is a sine function at  $50\text{ kHz}$ , and a modulation depth of 50%:

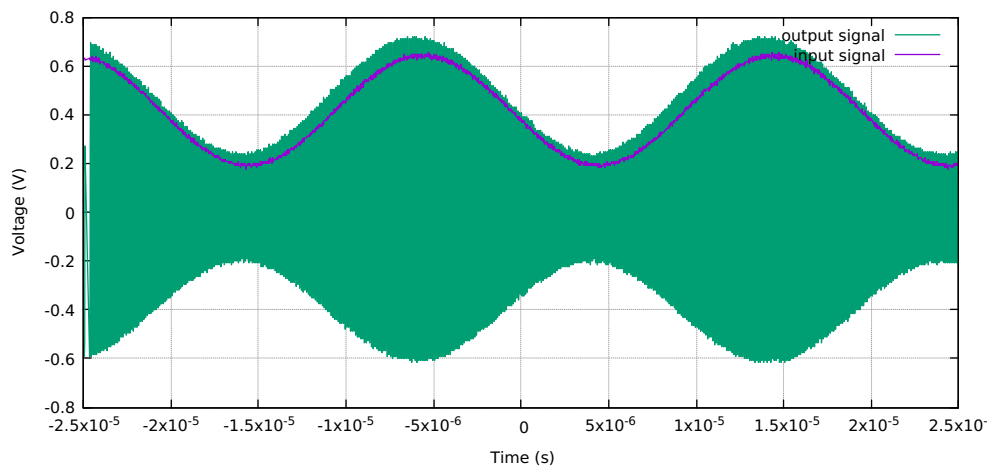


Figure 31: Amplitude demodulation.

As it is visible on the fig.31, a slight delay is visible between the input and the demodulated signal. This is due to the signal processing in the FPGA.

### Phase demodulation

For the phase demodulation, the carrier and the demodulation signal must be in quadrature. In this example the carrier frequency is  $10\text{ MHz}$ , and the modulating signal is a square wave at  $50\text{ kHz}$  with a deviation of  $50^\circ$ :

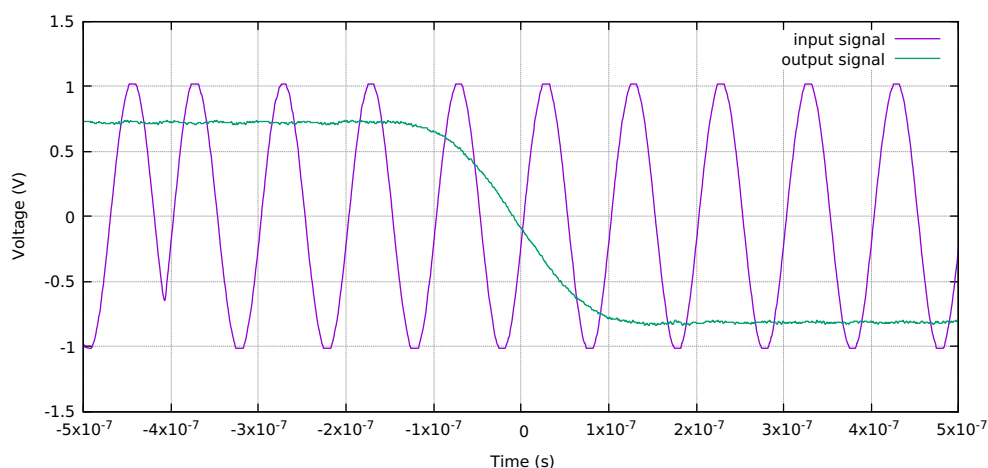


Figure 32: Phase demodulation.

In this figure, only the transition between the two levels of the square is visible, at the time  $-410\text{ ns}$  for the input signal and around  $0\text{ s}$  for the output signal. This shows the delay due to the signal processing in the FPGA. In this example the delay is mostly due to the FIR, since it requires 55 cycles to work, i.e. the number of coefficients of the FIR.

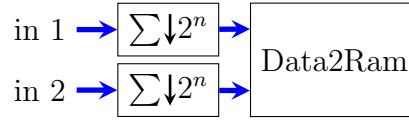
## 9.4 Unexpected output

- The demodulation is not perfect, or the modulation and demodulation frequency seems to be slightly different:

*Make sure your modulating and demodulating devices are referenced on the same oscillator.*

## 10 Monitoring

In this section we show an example of use of the dataReal\_to\_ram block, used to monitor data. Basically monitoring a data flow only requires the dataReal\_to\_ram block, however we show a setup where the meanReal block is placed upstream to decimate/slow down the flow. The scheme is presented below:



The block diagram corresponding to this scheme is as follows:

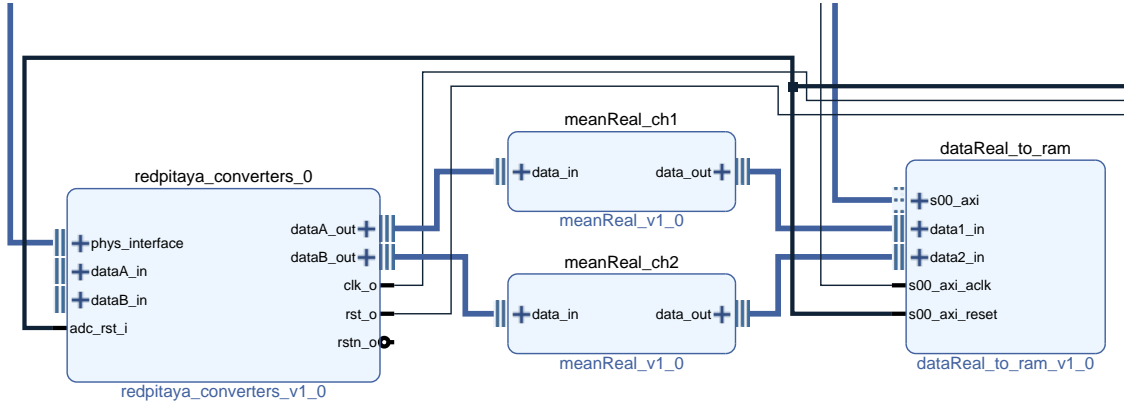


Figure 33: Part of the block diagram for monitoring with the data\_to\_ram block.

For an average/decimation  $\geq 2$ , this configuration results in spectral aliasing and is a disadvantage when used without filtering. However when the useful signal spreads on a shorter range than the spectral range available, it allows monitoring the data with more points than with a full range.



## 10.1 IP configuration

he IPs configuration in this example is given in the following table :

IP	Configuration
dataReal_to_ram	Data size: 16 <i>bits</i> Data format: signed NB input: 2 Nb Sample: = $K$
meanReal_ch1/2	Input Data size: 14 <i>bits</i> Output Data Size: 16 <i>bits</i> Nb Accum: = $2^n$ <i>bits</i> Shift: = $n$ <i>bits</i>

The number of samples of the dataReal\_to\_ram block, and the shift and number of accumulation of the meanReal blocks can vary upon the situations. Some examples will be listed in the expected output section (10.6).

## 10.2 Data reading

The data stored in the data\_to\_ram buffer are C signed shorts, over 2 *bytes*. With a data\_to\_ram configured with 2 inputs and  $K$  samples, this means the buffer contains  $4 \times K$  *bytes*. An example of reading of the data\_to\_ram block with a C script is presented here:

<https://github.com/oscimp/oscimpDigital/tree/master/doc/tutorials/redpitaya/3-PLPS>

In python the C structs must be converted to a tuple using the package struct. In the following example we read, convert and deinterleave the two first values of each channel, stored in the buffer:

```
import struct

with open('/dev/data', 'rb') as f:
    short_data = f.read(8) # read the 8 first data of the buffer
    interleaved_channels = struct.unpack('4h', short_data) #4h means 4 signed shorts
    ch1=interleaved_channels[0:2] #deinterleave: first input
    ch2=interleaved_channels[1:2] #deinterleave: second input
```

Which gives as an example of outputs:

```
short_data = b'\R\rz\xff\xb9\xe\x87\xff' #bytes
interleaved_channels = (3410, -134, 3769, -121) #tuple
ch1 = (3410, 3769) #tuple
ch2 = (-134, -121) #tuple
```

Obviously, the number of values read per channel can go up to the  $K$  samples mentioned in the `data_to_ram` block.

### 10.3 Data sending

A live monitoring of the `data_to_ram` data can be performed on a distant computer of the local network by:

1. Sending the data through the local network using a ZeroMQ (ZMQ) protocol<sup>4</sup>,
2. Receiving and plotting the data using GNU Radio Companion<sup>5</sup> or any script.

In this example we use a publish-subscribe protocol, where the publisher is the embedded linux system of the board, and the subscriber is the monitoring system. The sending of the data is performed using the ZMQ protocol, thus it requires to include the ZMQ tools to the buildroot. The following script can be launched as a background task. The data of the `/dev/data_to_ram` file is read and sent through the local network, via the port 9901 of the local system, with the publish protocol of ZMQ:

```
#!/usr/bin/env python

import zmq, time

context = zmq.Context()
sock = context.socket(zmq.PUB) #publisher socket type
sock.bind("tcp://*:9901") #port 9901 of the local machine

while True:
    time.sleep(0.05) #be kind to the CPU, adapt it to the buffer delay
    with open('/dev/data_to_ram', 'rb') as f:
        sock.send(f.read(8192)) #read and send the 8192 data of the data_to_ram buffer
```

The sleep time in the loop can be adapted to the delay between the transmission of two buffers by the `data_to_ram` block. See the examples in section 10.6.

### 10.4 Data receiving

The reception of the data using the ZMQ subscribe protocol can be performed with the following script:

```
import zmq, time, struct

context = zmq.Context()
sock = context.socket(zmq.SUB) #subscriber socket type
sock.setsockopt(zmq.SUBSCRIBE, "").encode('utf-8'))
```

---

<sup>4</sup><https://zeromq.org/>

<sup>5</sup><https://wiki.gnuradio.org/>

```
sock.connect("tcp://192.168.0.215:9901") #connect to the port 9901 of the 192.168.0.215 machine
received_data = sock.recv() #the data received remains to be converted, deinterleaved...
```

Then the data remains to be processed.

## 10.5 Data monitoring

The reception of the data using the ZMQ subscribe protocol can also directly be achieved using GNU Radio Companion. The flow graph is as following:

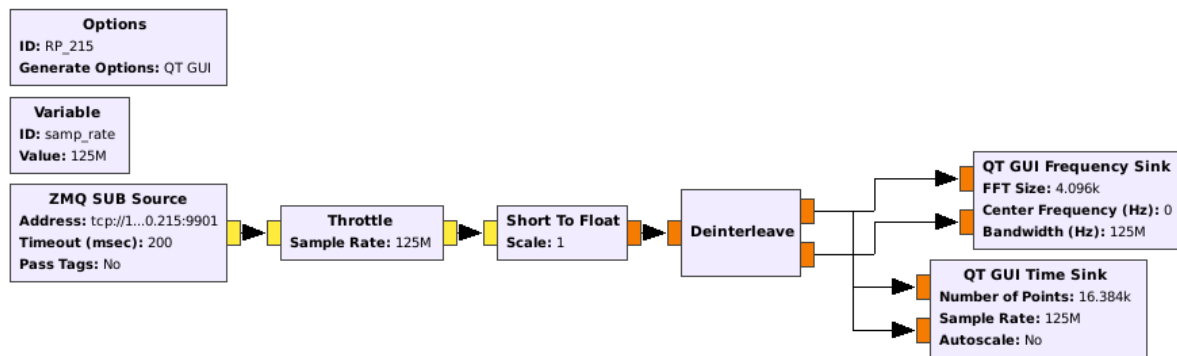


Figure 34: Flow graph for the data monitoring.

The principle of the data processing is very similar to the data reading in section 10.2:

1. ZQM SUB Source: subscribes to the sender at the address `tcp://IP:PORT` (in this example `tcp://192.168.0.215:9901`).
2. Throttle: regulates the flow. Type: short.
3. Short To Float: converts short data into float data.
4. Deinterleave: separates the two channels received. IO Type: float. Num Streams: 2.
5. QT GUI Frequency Sink: plots the power spectrum density on a data set (FFT Size) that can be adapted to the `data_to_ram` configuration. Type: float. Number of Inputs: 2.
6. QT GUI Time Sink: plots a data set (Number of Points) in the time domain. Can also be adapted to the `data_to_ram` configuration. Type: float. Number of Inputs: 2.

The variable block is used here to mention the sampling rate *samp\_rate* of the `data_to_ram`, since it is used by several blocks. Its value depend on the average/decimation *n* performed before the `data_to_ram`, that also divides the data flow. Thus for an initial sampling rate of 125 MHz, we have  $samp\_rate = 125/n$  MHz.

## 10.6 Expected output

Reminder on the effect of the sampling rate:

One would expect to monitor the data in the time domain with a satisfying resolution. However we should keep in mind that this resolution depends on the sampling rate with respect to the frequency/variations of the signal. This implies first that above the half Nyquist frequency, the signal is not reconstituted with a lot of point, but still can be interpolated. Secondly, if the input frequency is above the Nyquist frequency, there is spectral aliasing. In this situation the reconstituted signal can only be correctly interpreted with your a priori knowledge on the input signal.

Those points have an impact here since we propose to use the meanReal block that decimates the signal and divides the sampling rate, as mentioned in the section above. To highlight the pros and cons of this configuration, two situations are presented here: monitoring of a sine signal without averaging (i.e. sampling rate of  $125\text{ MHz}$ ), and with an averaging of  $2^{13}$  (i.e. sampling rate of  $\simeq 15.26\text{ kHz}$ ).

In both examples, the data is monitored with GNU Radio Companion, and no webserver is required. The signal in the first channel CH1 is a sine signal at  $20\text{ MHz}$ , and the signal in the second channel CH2 is a sine signal at  $2\text{ kHz}$ .

No averaging

Configurations in the block design:

- dataReal\_to\_ram:  $K = 16384$
- meanReal\_ch1/2:  $n = 0$  (Nb Accum: 1 *bit*, Shift: 0 *bit*)

Then the time delay between two buffers is  $\Delta t_{buffer} = \frac{16384 \times 1}{125 \cdot 10^6} = 1.31 \cdot 10^{-4}\text{ s}$ . The loop time for the data sending (see section 10.3) can be quite low. In this example we set it to  $0.05\text{ s}$ .

The data is then monitored with GNU Radio Companion. The blocks configuration is:

- samp\_rate:  $125\text{ MHz}$
- QT GUI Frequency Sink, FFT Size: up to 4096 *points*
- QT GUI Time Sink, Number of points: up to  $2^{14}$
- Update Period:  $0.05\text{ s}$
- Other parameters: up to you !

We obtain the following results for the first (fig.35) and second (fig.36) channel:

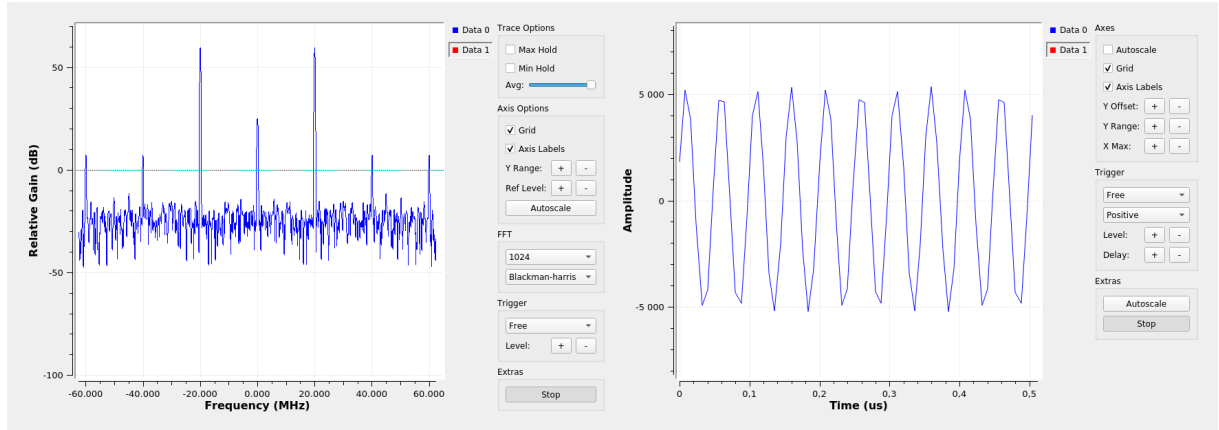


Figure 35: First channel at 20  $MHz$ , no decimation.

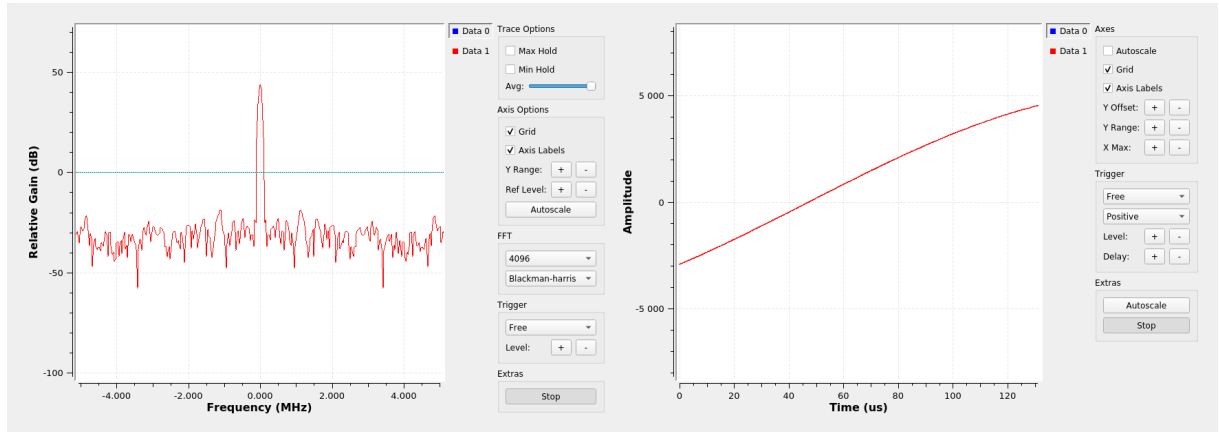


Figure 36: Second channel at 2  $kHz$ , no decimation.

The signal at 20  $MHz$  is clearly visible in the time and frequency domains, although harmonics are also represented. However the signal at 2  $kHz$  is not noticeable at all in the frequency domain, and oversampled in the time domain. In this case the second example is more adapted.

### Averaging of $2^{13}$

In this case the averaging is used to divide the sampling rate and reduce the spectral range. Then it is more adapted to lower frequencies. Configurations in the block design:

- dataReal\_to\_ram:  $K = 2048$
- meanReal\_ch1/2:  $n = 13$  (Nb Accum: 8192 *bit*, Shift: 13 *bits*)

Then the time delay between two buffers is  $\Delta t_{buffer} = \frac{2048 \times 8192}{125 \cdot 10^6} = 1.34 \cdot 10^{-1} \text{ s}$ .

Since this delay is higher than in the previous example, the loop time for the data sending is increased to 0.5 s.

The blocks configuration in GNU Radio Companion is:

- samp\_rate: 15.26  $kHz$
- QT GUI Frequency Sink, FFT Size: up to 2048 *points*
- QT GUI Time Sink, Number of points: up to  $2^{11}$
- Update Period: 0.5 s
- Other parameters: up to you !

The results obtained are presented in fig.37 and fig.38:

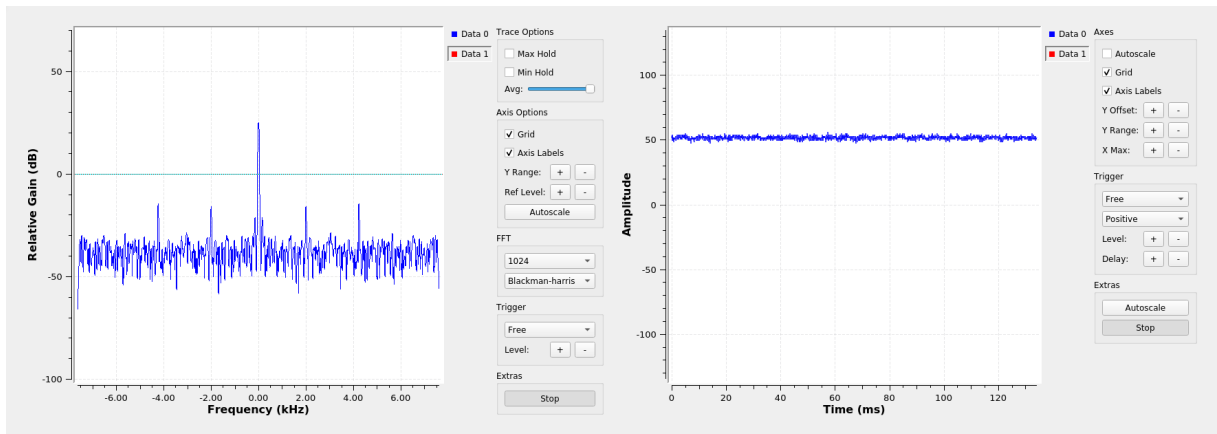


Figure 37: First channel at 20  $MHz$ , with decimation.

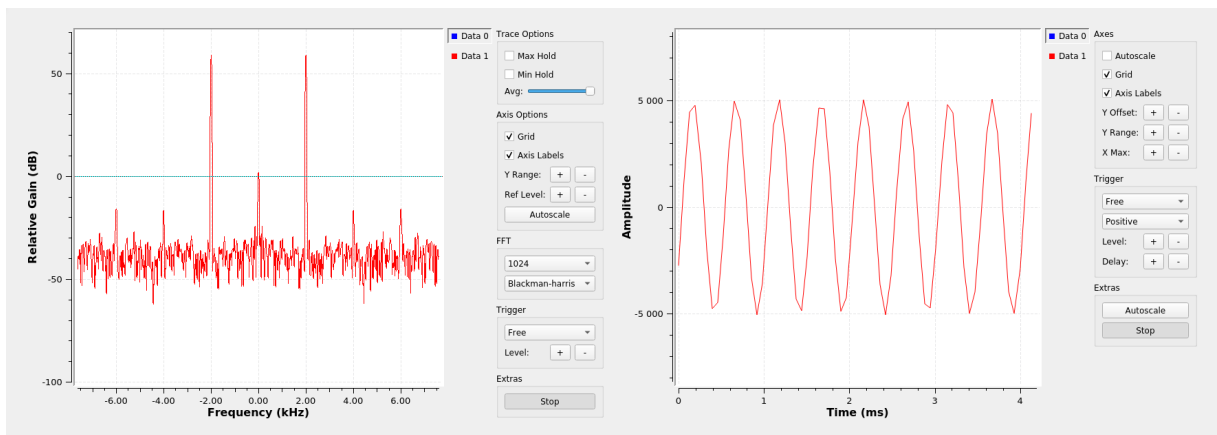


Figure 38: First channel at 2  $kHz$ , with decimation.

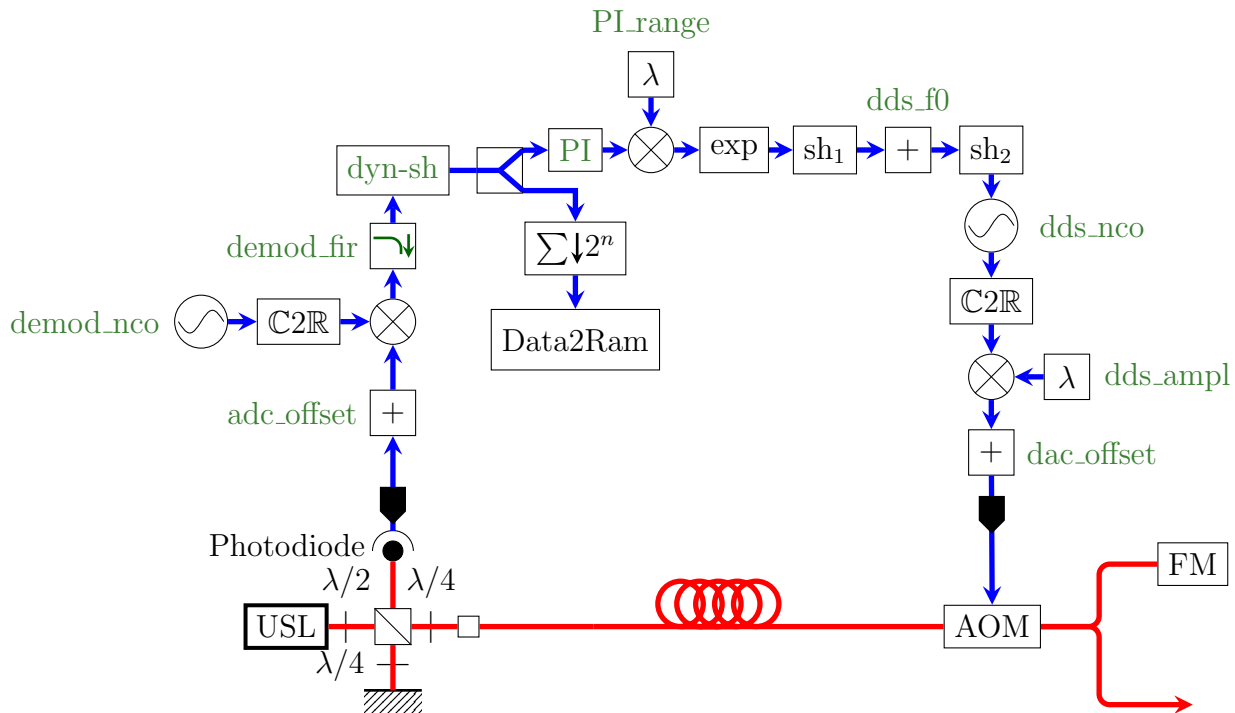
This time the signal at  $20\text{ MHz}$  is severely undersampled and can not be interpreted in the time domain. However the signal at  $2\text{ kHz}$  is this time visible with a sufficient resolution both in the time and the frequency domains.

## 10.7 Unexpected output

## 11 Example to a control loop

An example of control loop is presented here. Although it can be used both for amplitude and phase locked loops, only a phase locked loop (PLL) is shown here.

In this example the PLL aims at compensating the phase fluctuations that occur during the transfer of an ultra stable laser (USL), through an optical fiber. The scheme presenting the operating principle and all the RF processing is presented below:



The optical processing lays on the use of a Michelson interferometer that includes a first arm with a long optical fiber link, and a second arm much shorter. The second arm is kept as a reference, and is considered to have negligible phase fluctuations with respect to the ultra stable signal. The first arm is used to transfer the ultra stable signal, but is subject to external constraints and therefore undesirable phase fluctuations. A way to measure those phase fluctuations is first to modulate the ultra stable laser with a RF signal, using an acousto optic modulator (AOM). Secondly the phase fluctuation must be isolated. In this purpose, the modulated signal is reflected by a Faraday mirror (FM) placed at the end of the link, and sent back to the beginning of the link. The recombination between the

modulated signal and the reference signal mainly represents the undesirable phase fluctuations.

Subsequently the RF signal is detected after recombination and demodulated in phase (see section 9) to extract an error signal that correspond to those phase fluctuations. Then the PI block generates a correction signal that is used to modulate the frequency of the NCO that drives the AOM. Thereby, the undesirable phase fluctuations are directly compensated by the AOM modulation on the transmitted ultra stable signal.

## 11.1 IP configuration

The IPs configuration is given in the following table :

IP	Configuration
adc_offset dac_offset	Data in/out size: 14 <i>bits</i>
----- dds_f0	----- Data in/out size: 32 <i>bits</i> Unsigned
demod_nco dds_nco	Counter size: 40 <i>bits</i> Data size: 14 <i>bits</i> Lut size: 12 <i>bits</i>
$\mathbb{C}2\mathbb{R}$	Data size: 14 <i>bits</i>
multiplierReal	Input data1/2 size: 14 <i>bits</i> Output data size: 14 <i>bits</i>
firReal	COEFF size: 16 <i>bits</i> Data in size: 14 <i>bits</i> Data out size: 32 <i>bits</i> Decimate Factor: 1 Nb COEFF: 25
dyn_sh	Data in size: 32 <i>bits</i> Data out size: 14 <i>bits</i> Default Shift: 18 <i>bits</i>
dupplReal	Data size: 14 <i>bits</i>
meanReal	Input Data size: 14 <i>bits</i> Output Data Size: 16 <i>bits</i> Nb Accum: = 128 <i>bits</i> Shift: = 7 <i>bits</i>



dataReal_to_ram	Data size: 16 <i>bits</i> Data format: signed NB input: 1 Nb Sample: = 16384
IP	Configuration
PI	Data in/out size: 14 <i>bits</i> I shift: 19 <i>bits</i> I size: 18 <i>bits</i> P shift: 13 <i>bits</i> P size: 14 <i>bits</i>
PI_range dds_ampl	Data size: 14 <i>bits</i>
exp	Data in size: 14 <i>bits</i> Data out size: 20 <i>bits</i>
sh <sub>1</sub>	Data in size: 20 <i>bits</i> Data out size: 32 <i>bits</i>
sh <sub>2</sub>	Data in size: 32 <i>bits</i> Data out size: 40 <i>bits</i>

Here, the expander (exp) and shifter (sh<sub>1</sub>) are used to roughly adapt the range of the correction for the application described above. Although the correction range can also be adjusted (more precisely) with the PI\_range variable, it is possible that depending on the application the values of the expander (exp) and shifter (sh<sub>1</sub>) must be adapted.

## 11.2 Webserver configuration

In the following webserver, the PI block is represented with 5 variables: the set point PI/setpoint, the sign PI/sign, the proportional gain PI/kp, the integral gain PI/ki, and the rst\_int checkbox that corresponds both to a disable integral correction and reset of the integral accumulator.

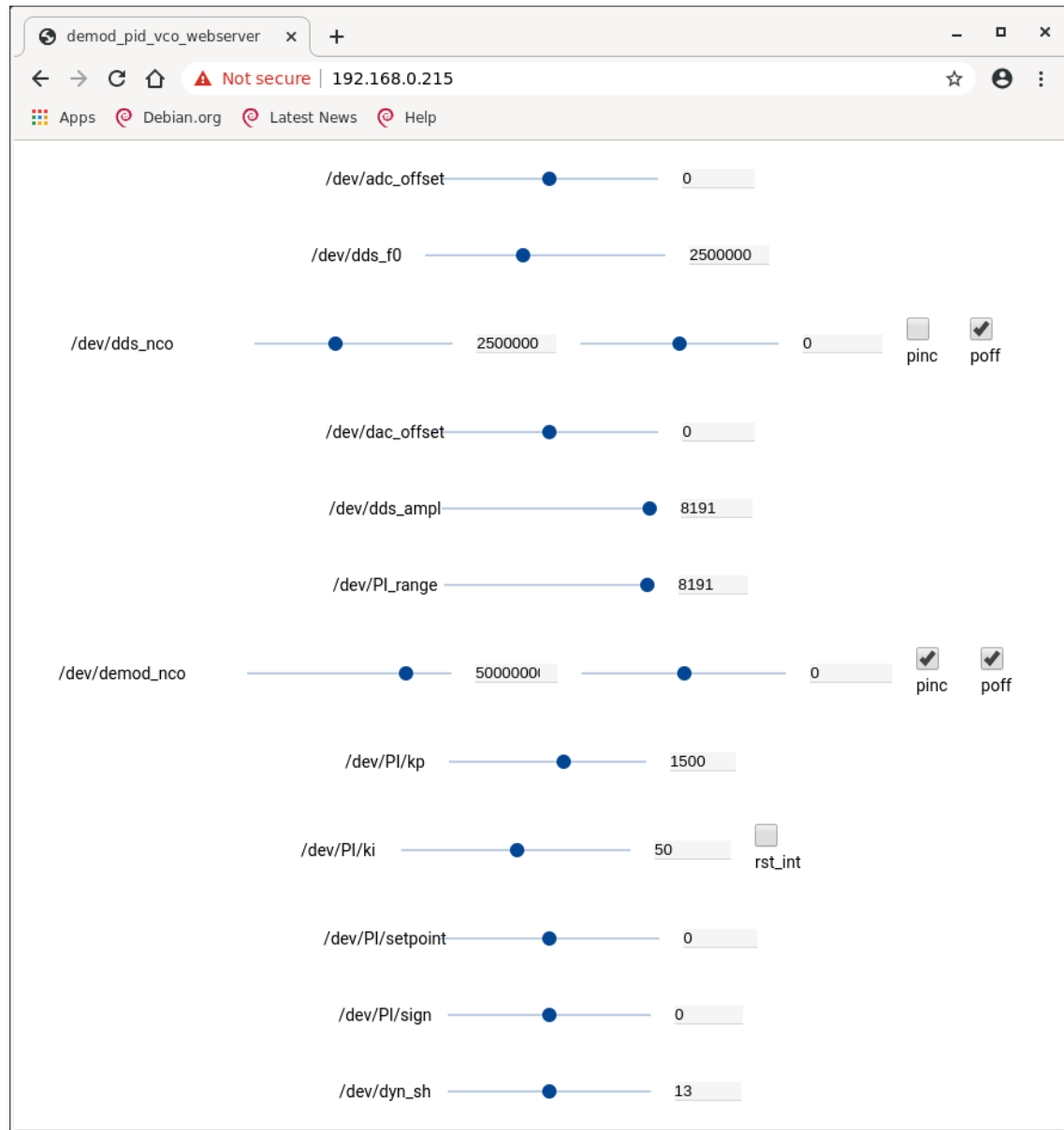


Figure 39: Screenshot of the control loop webserver.

### 11.3 Expected output

