# Base designs

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

April 15, 2021

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

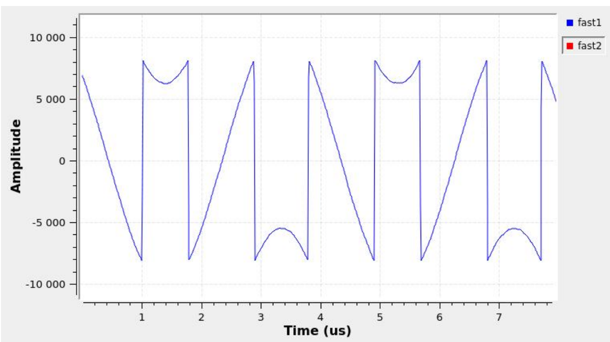
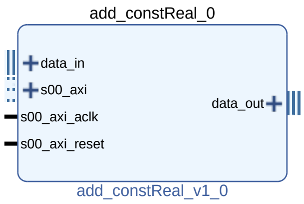
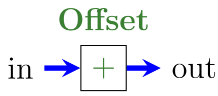
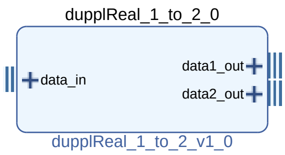
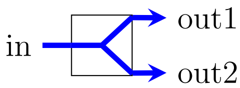
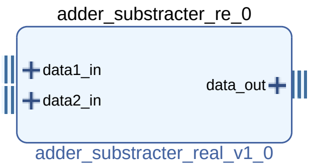
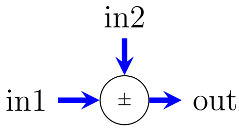
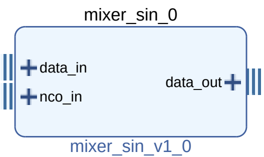
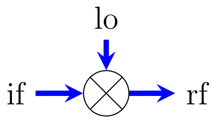
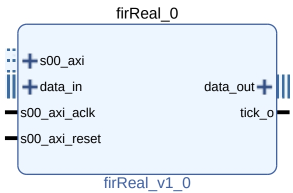
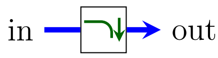
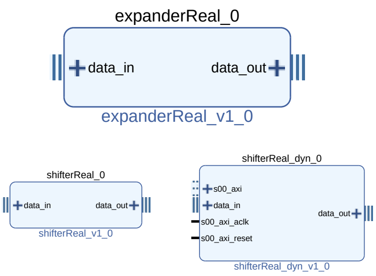
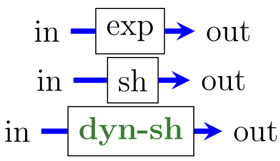
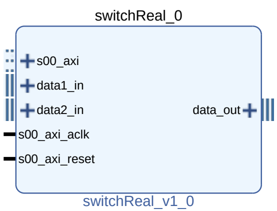
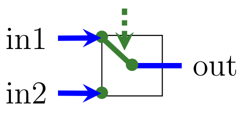
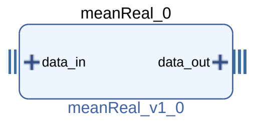
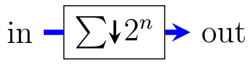
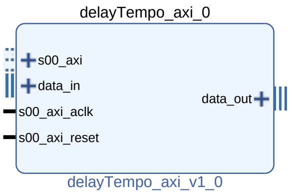
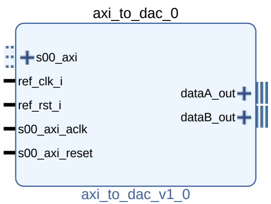
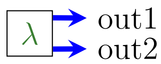
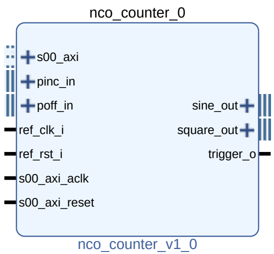
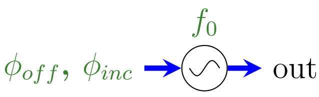
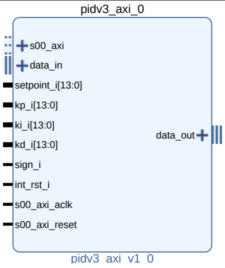
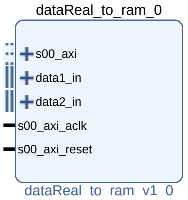
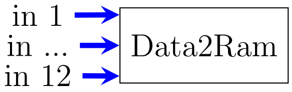
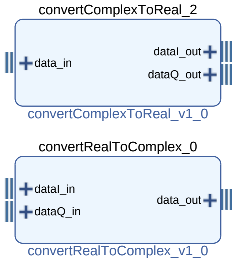
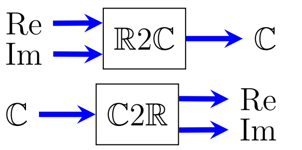



Figure 1: Overflow on the top and bottom of a sine.

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

| **IP** | **Equivalent RF function** **or numeric function** | **Equivalent scheme** **with tunable entries** |
|---|---|---|
|   | 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_recv_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[1], 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_recv_conf(filename, reg):
    file = ctypes.create_string_buffer(str.encode(filename))
    my_val = c_int()
    ret_val = lib.fpgagen_recv_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
```

[1]Download and Faq : https://www.remigui.com/

```python
class MyApp(App):
    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.onchange.do(self.sd_MY_VALUE_changed)
        self.sb_MY_VALUE = gui.SpinBox(0, -8192, 8191, 1, width="20%", margin="10px")
        self.sb_MY_VALUE.onchange.do(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.onchange.do(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:

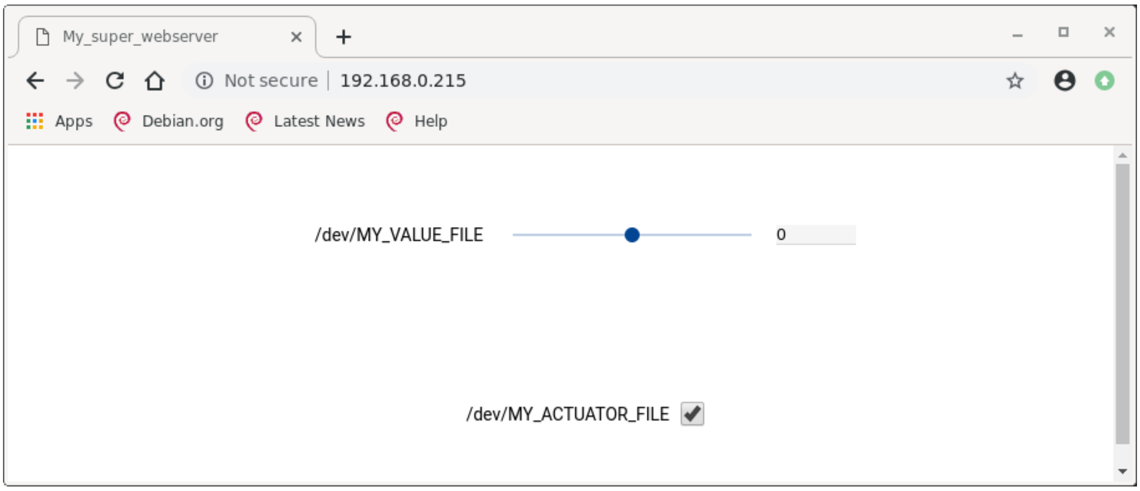



Figure 2: Example of webserver with MY_VALUE_FILE represented by a slider and a spinbox, and MY_ACTUATOR_FILE represented by a checkbox.

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. The block diagram is as follows :

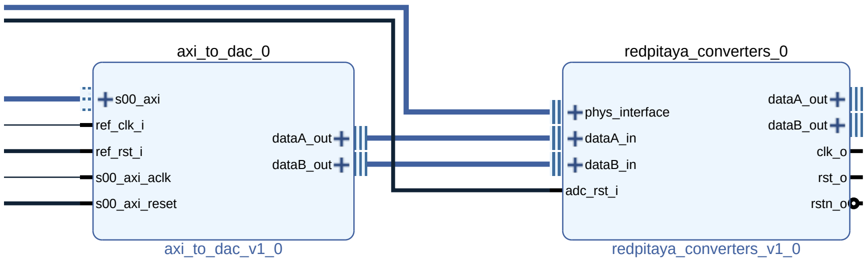
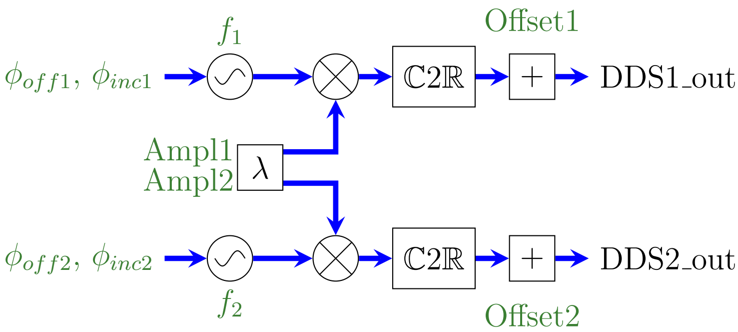



Figure 3: Part of the block diagram for the tunable voltage source.

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:

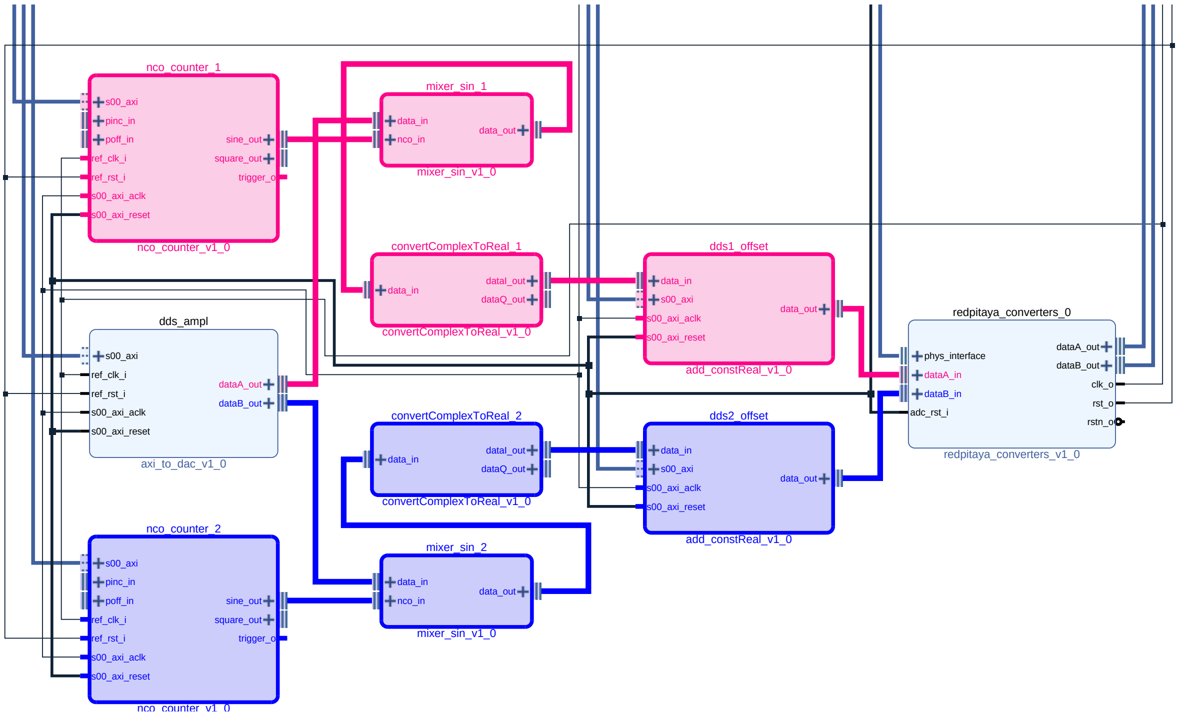



Figure 4: Part of the block diagram for the double DDS.

## 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 *bits*<br>Data size: 16 *bits*<br>Lut size: 12 *bits* |
| dds_ampl | Data size: 14 *bits* |
| mixer_sin_1/2 | Data in/out size: 14 *bits*<br>Nco_size: 16 *bits* |
| dds1/2_offset | Data in size: 14 *bits*<br>Data out size: 14 *bits*<br>Signed |
| convert$\mathbb{C}$to$\mathbb{R}$_1/2 | Data size: 14 *bits* |

## 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:

- $1^{st}$ slider+spinbox: the frequency control, up to the half clock frequency (Hz)
- $2^{nd}$ 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.

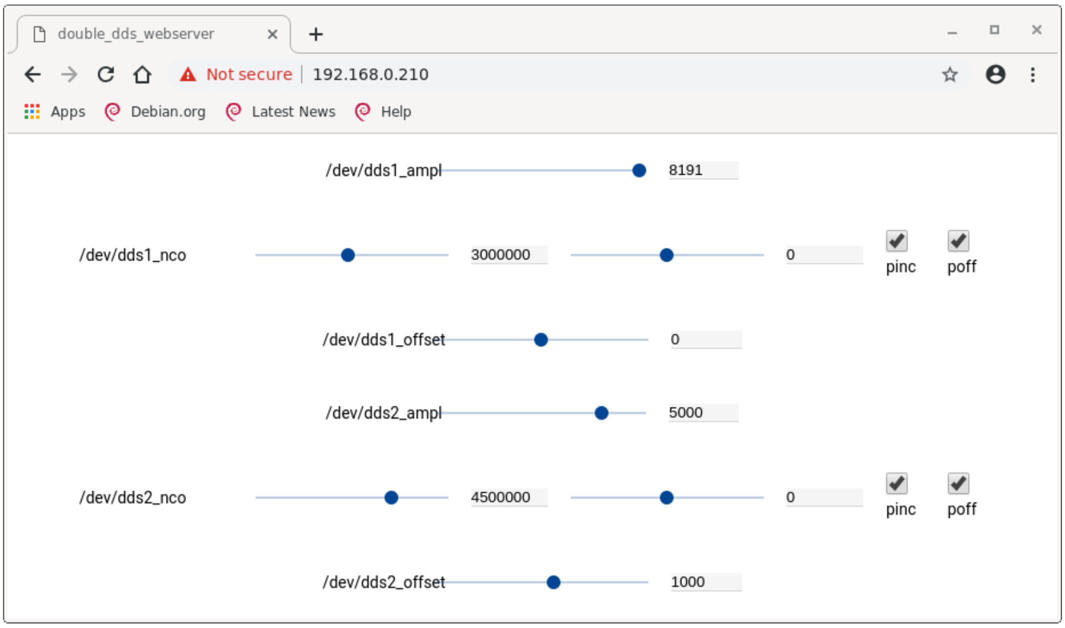



Figure 5: Screenshot of the double DDS webserver.

## 4.3 Expected output

We show in fig.6 an example of two signals generated.
Ch1: $f_0 = 30\ MHz$, $dds1\_ampl = 8191\ arb.\ unit$, $dds1\_offset = 0\ arb.\ unit$, $\phi_{off1} = 0\ arb.\ unit$.
Ch2: $f_0 = 45\ MHz$, $dds2\_ampl = 3000\ arb.\ unit$, $dds2\_offset = 5000\ arb.\ unit$, $\phi_{off2} = 0\ arb.\ unit$.

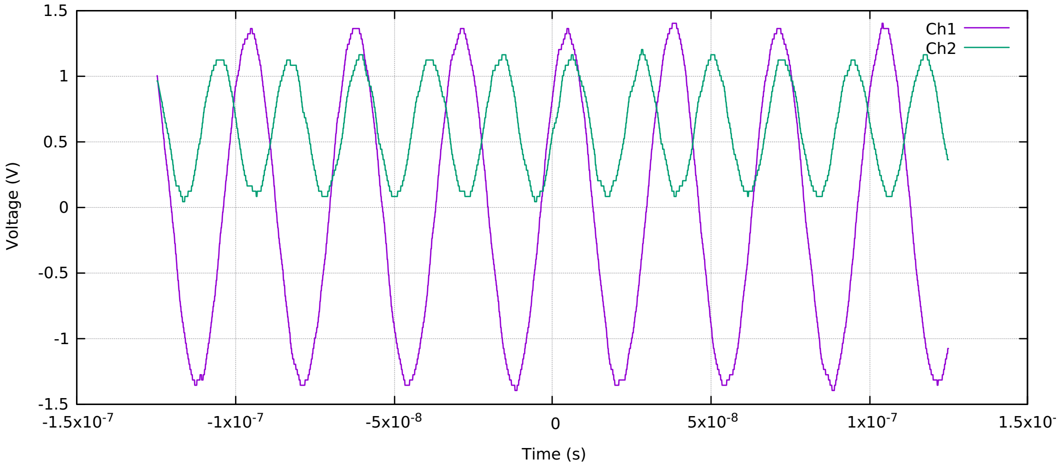



Figure 6: Expected output.

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:

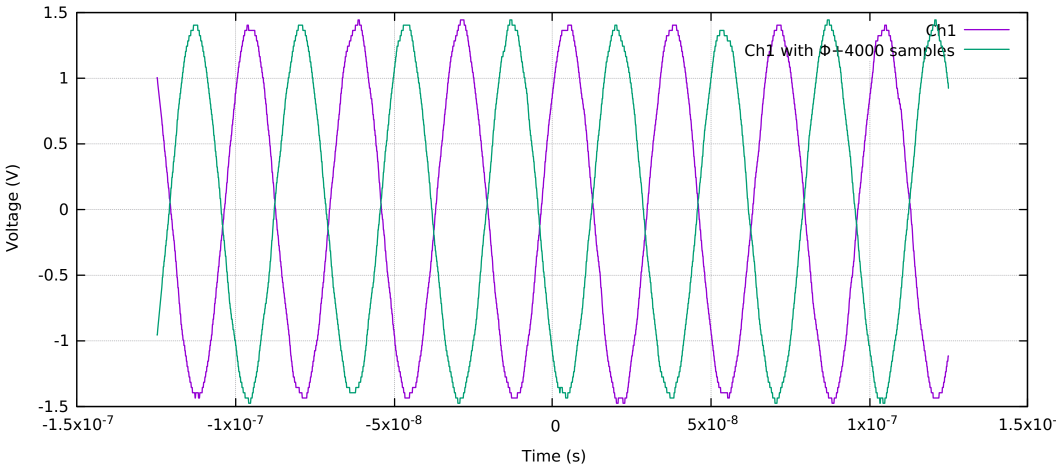



Figure 7: Phase offset.

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

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

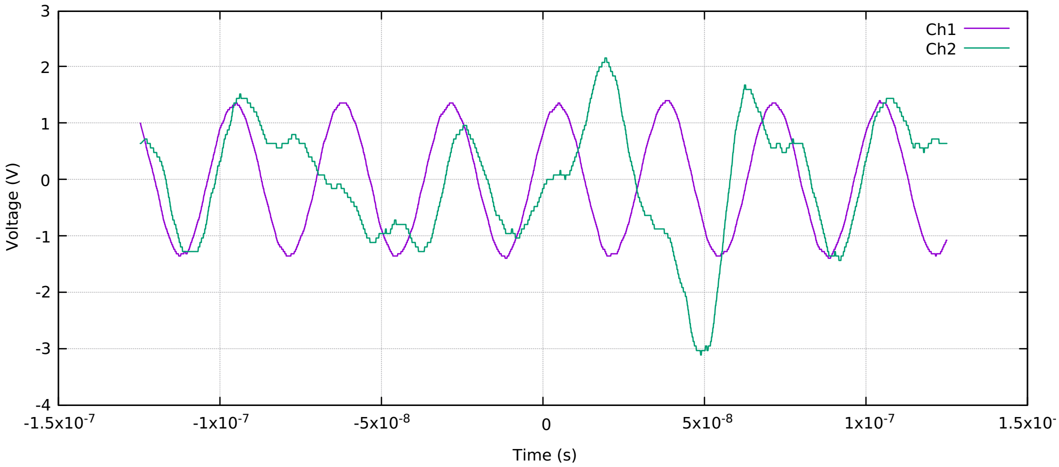
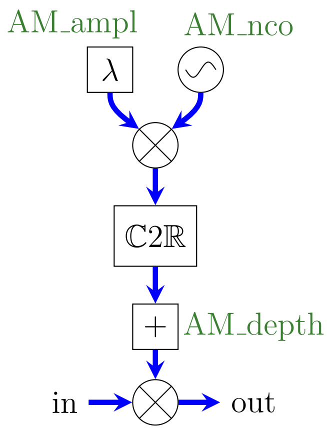



Figure 8: Unexpected output due to overflow in ch2.

# 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 $AM\_ampl = 4096\ arb.\ unit$ and $AM\_depth = 8191\ arb.\ unit$
- $h = 1$ with $AM\_ampl = 8191\ arb.\ unit$ and $AM\_depth = 8191\ arb.\ unit$
- $h = 2$ with $AM\_ampl = 8191\ arb.\ unit$ and $AM\_depth = 4096\ arb.\ unit$

The block diagram corresponding to this scheme is as follows:

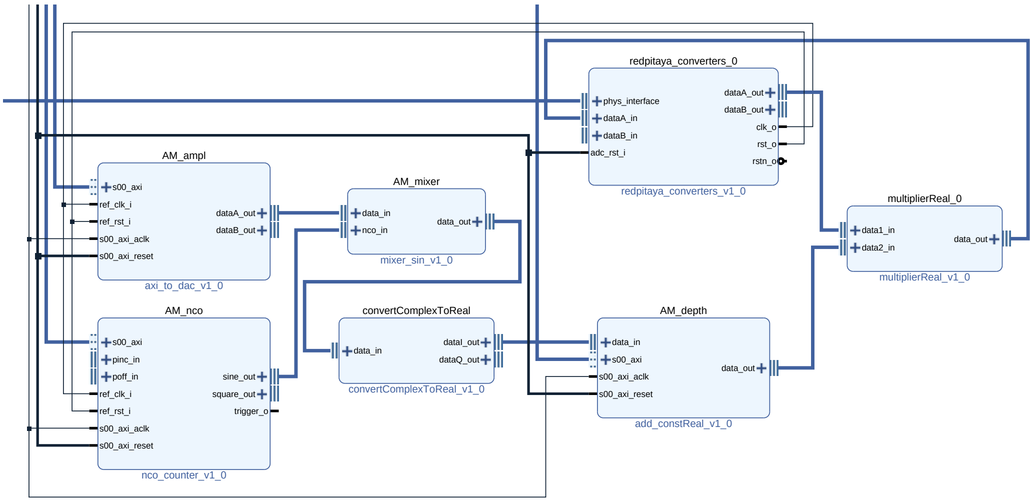



Figure 9: Part of the block diagram for this amplitude modulation.

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 *bits*<br>Data size: 16 *bits*<br>Lut size: 12 *bits* |
| AM_ampl | Data size: 16 *bits* |
| AM_mixer | Data in/out size: 16 *bits*<br>Nco_size: 16 *bits* |
| multiplierReal | Input data1 size: 14 *bits*<br>Input data2 size: 16 *bits*<br>Output data size: 14 *bits* |
| AM_depth | Data in size: 16 *bits*<br>Data out size: 16 *bits*<br>Format: Signed |
| convert$\mathbb{C}$to$\mathbb{R}$_1 | Data size: 16 *bits* |

## 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:

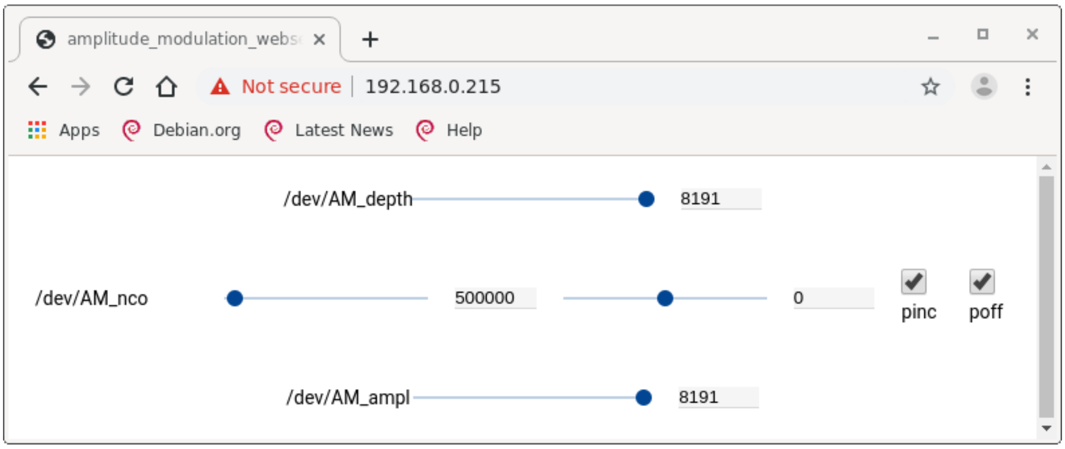



Figure 10: Screenshot of the amplitude modulation part of the webserver.

## 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 $MHz$ and 0 $dBm$. The modulating signal is set to 500 $kHz$.

Input and output with $h = 0.5$

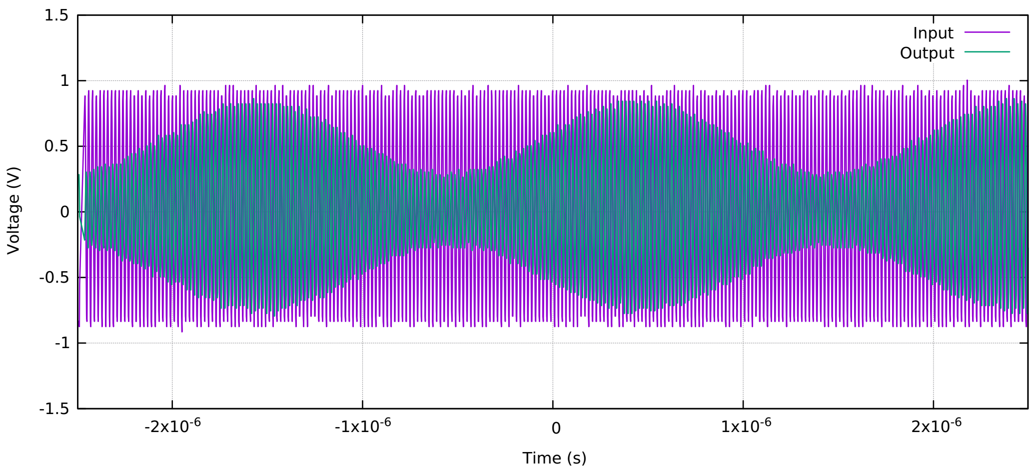



Figure 11: Expected behavior for $h = 0.5$.

Output with $h = 1$

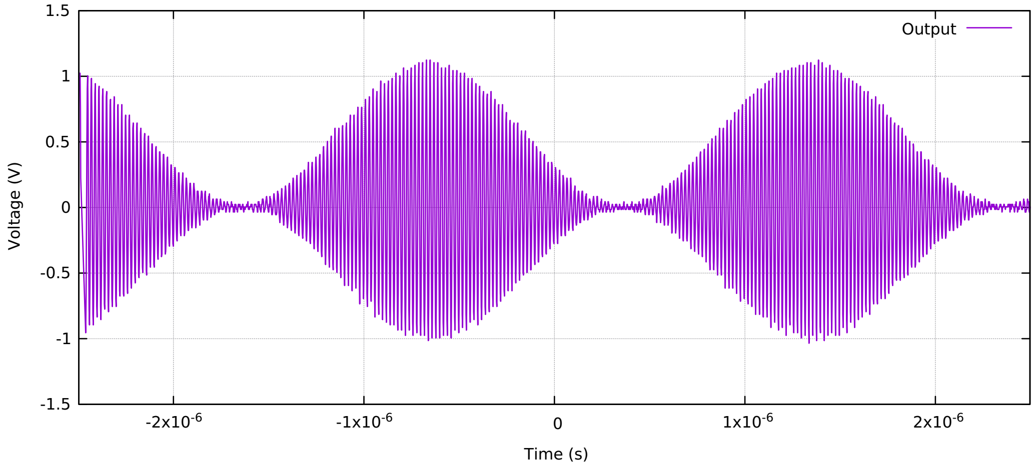



Figure 12: Expected behavior for $h = 1$.

Output with $h = 2$ (overmodulation)

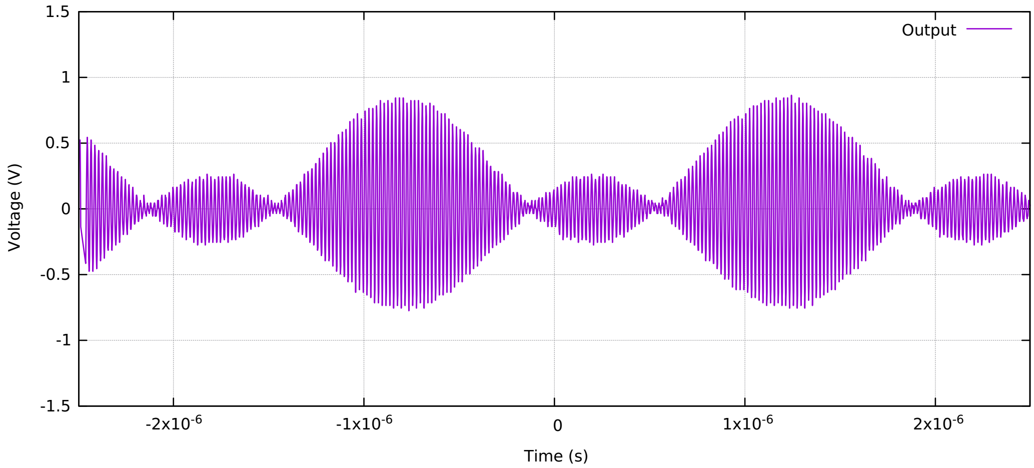



Figure 13: Expected behavior for $h = 2$: overmodulation.

## 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.

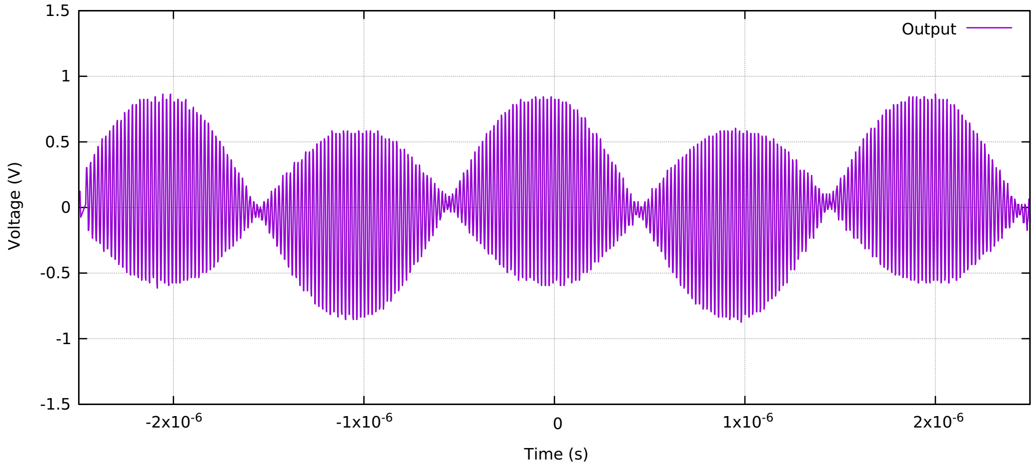
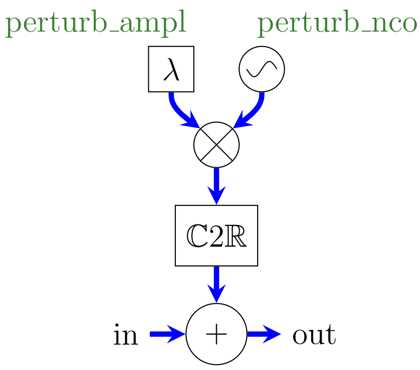



Figure 14: Case of a carrier frequency below the input signal frequency.

# 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:

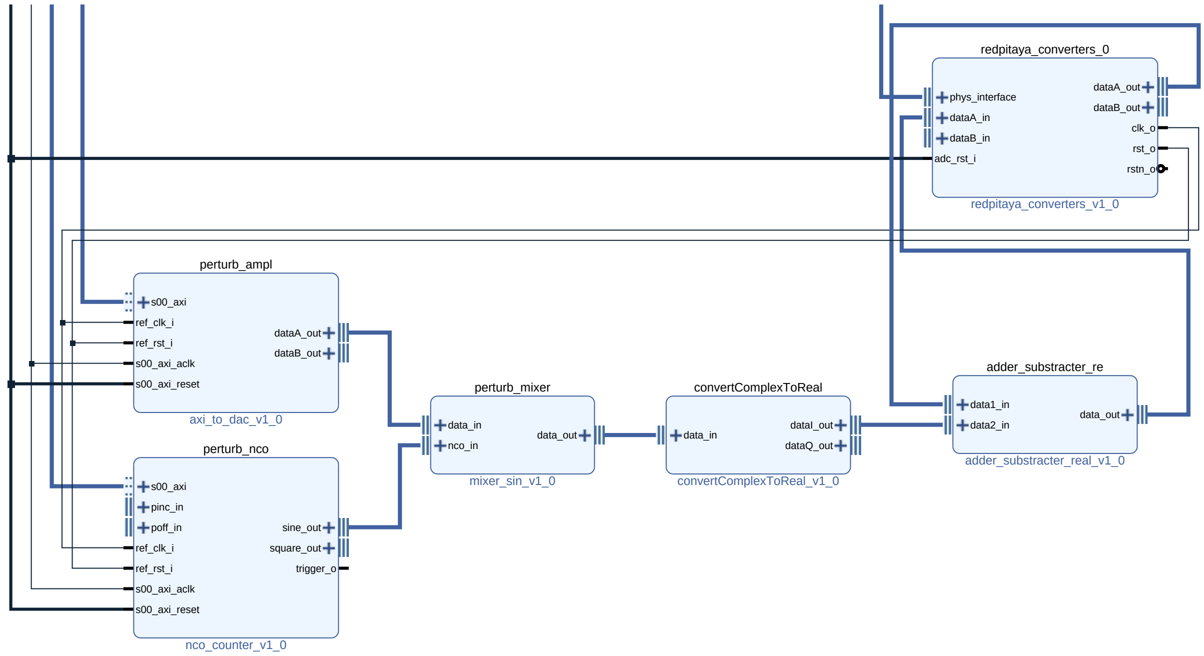



Figure 15: Part of the block diagram for the amplitude modulation.

## 6.1 IP configuration

IPs configuration in this example:

| IP | Configuration |
|---|---|
| perturb_nco | Counter size: 40 *bits*<br>Data size: 16 *bits*<br>Lut size: 12 *bits* |
| perturb_ampl | Data size: 14 *bits* |
| perturb_mixer | Data in/out size: 14 *bits*<br>Nco_size: 16 *bits* |
| adder_substracter_re | Data size: 14 *bits*<br>Operation: add<br>Format: Signed |
| convert$\mathbb{C}$to$\mathbb{R}$_1 | Data size: 14 *bits* |

## 6.2 Webserver configuration

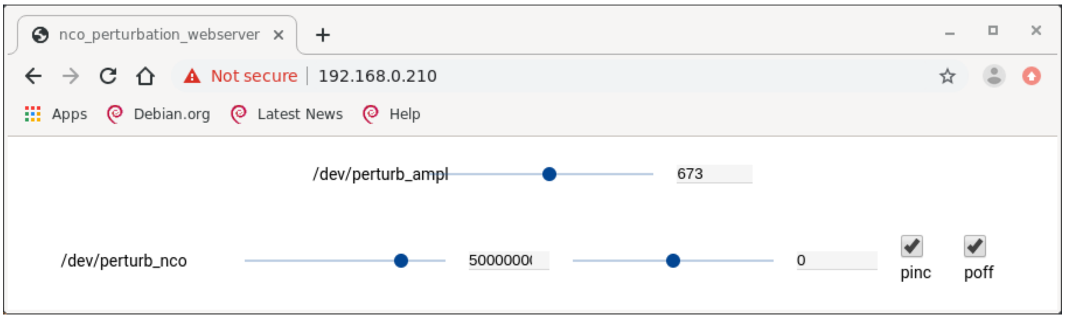



Figure 16: Screenshot of the sine perturbation part of the webserver.

## 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:

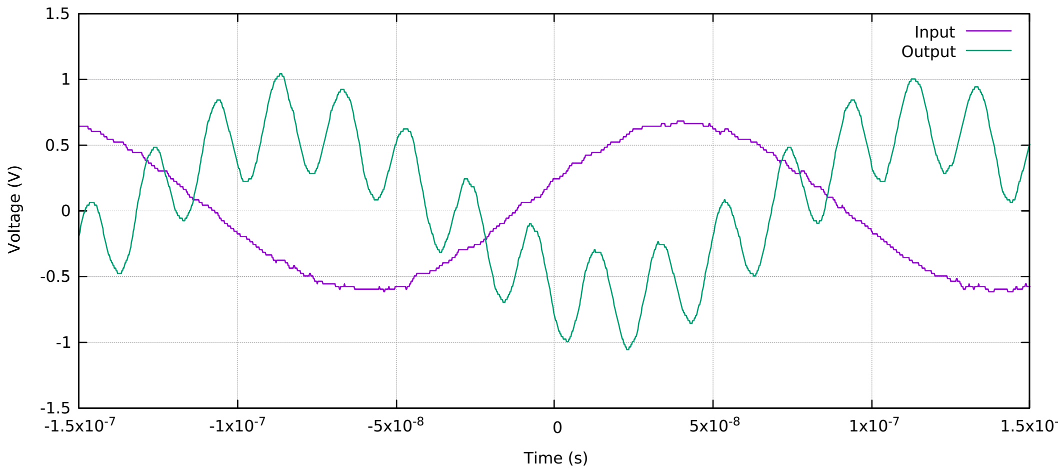



Figure 17: Expected behavior with input sine at 5 $MHz$ and sine perturbation of 50 $MHz$.

## 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.

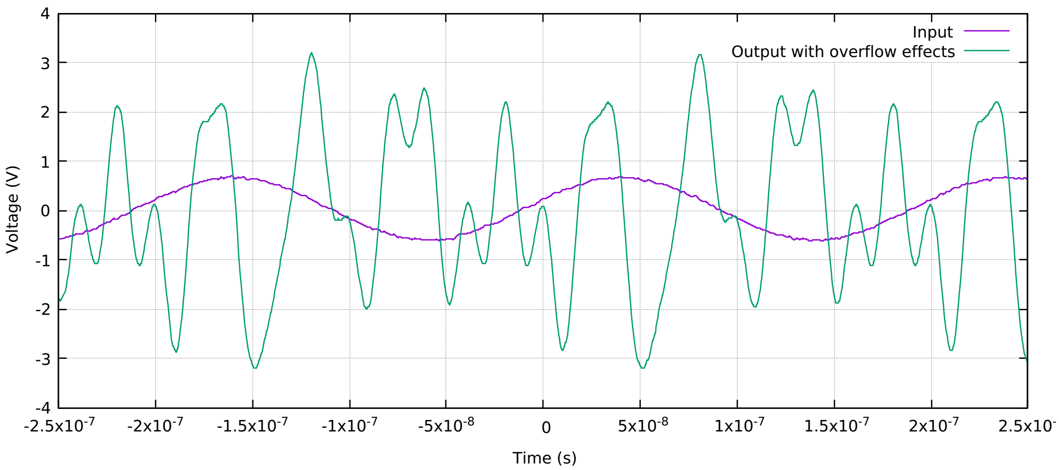
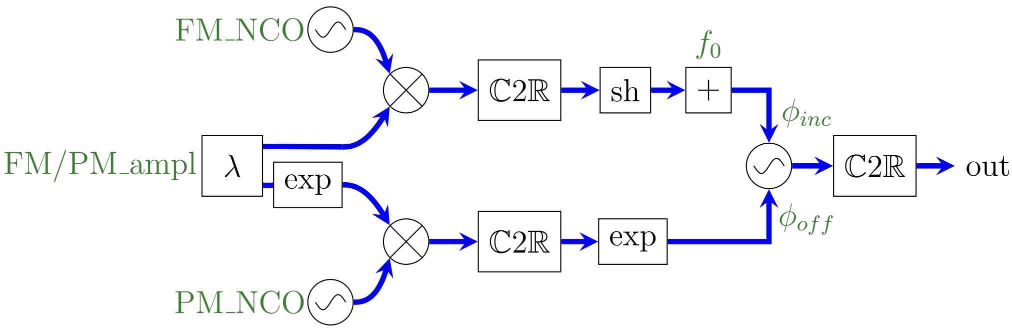



Figure 18: Unexpected behavior with input sine at 5 $MHz$ and sine perturbation of 50 $MHz$.

# 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:

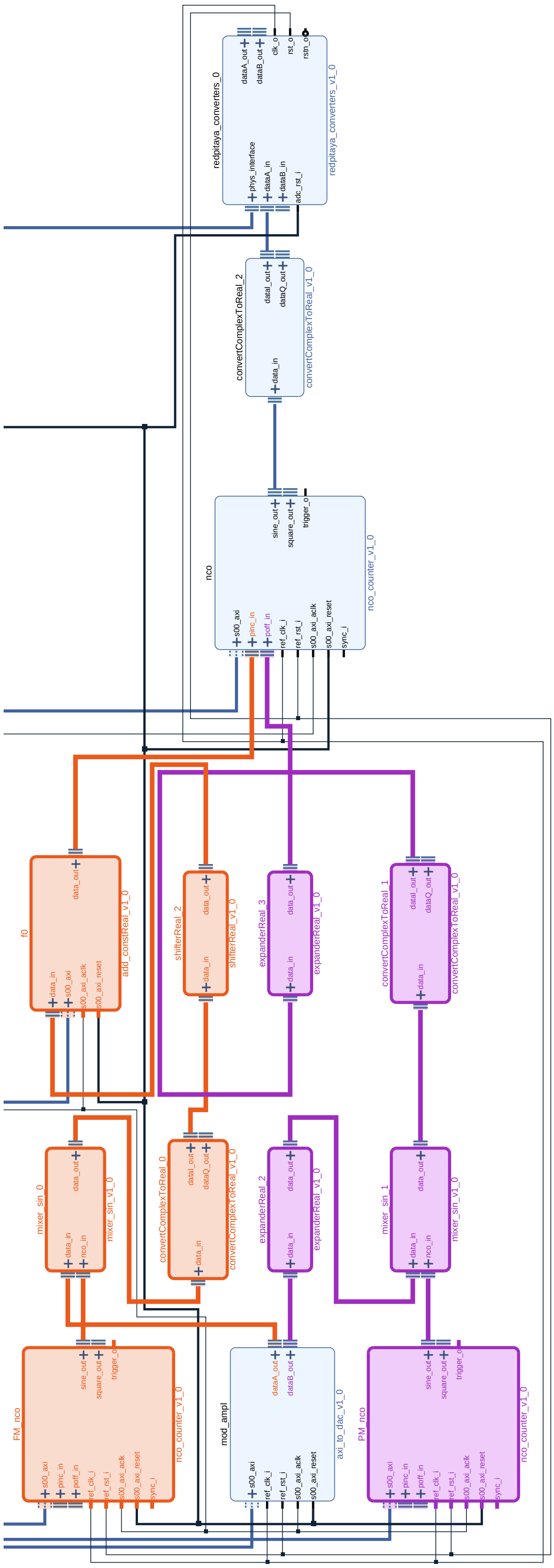



Figure 19: Part of the block diagram for frequency (orange) and phase (purple) modulation of a NCO.

| IP | Configuration |
| --- | --- |
| FP_nco/PM_nco/nco | Counter size: 40 $bits$<br>Data size: 16 $bits$<br>Lut size: 12 $bits$ |
| mod_ampl | Data size: 27 $bits$ |
| expanderReal_1 | Data in size: 27 $bits$<br>Data in size: 16 $bits$<br>Format: Signed |
| expanderReal_2 | Data in size: 16 $bits$<br>Data in size: 12 $bits$<br>Format: Signed |
| shiftReal_1 | Data in size: 27 $bits$<br>data out size: 40 $bits$ |
| mixer_sin_1 | Data in/out size: 27 $bits$<br>Nco_size: 16 $bits$ |
| mixer_sin_2 | Data in/out size: 16 $bits$<br>Nco_size: 16 $bits$ |
| $f_0$ | Data in/out size: 40 $bits$<br>Format: Signed |
| convert$\mathbb{C}$to$\mathbb{R}$_1 | Data size: 27 $bits$ |
| convert$\mathbb{C}$to$\mathbb{R}$_2 | Data size: 16 $bits$ |
| convert$\mathbb{C}$to$\mathbb{R}$_3 | Data size: 14 $bits$ |

## 7.2 Webserver configuration

Here, only the PM_deviation is reconfigured between $-8192$ and $8191$ $arb.\ unit$. Both FM_nco, PM_nco, f0, FM_deviation and nco are configured between 0 and 62000000 $Hz$.

Also f0 must be scaled to correspond to a phase increment. This takes into account the sampling rate, here 125 $MHz$, and the size (i.e. frequency resolution) of f0. Then the control of f0 in the webserver takes the following form:

```
liboscimp_fpga.add_const_set_offset("/dev/f0", int(round(float(value)/(125e6/2**40))))
```

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.

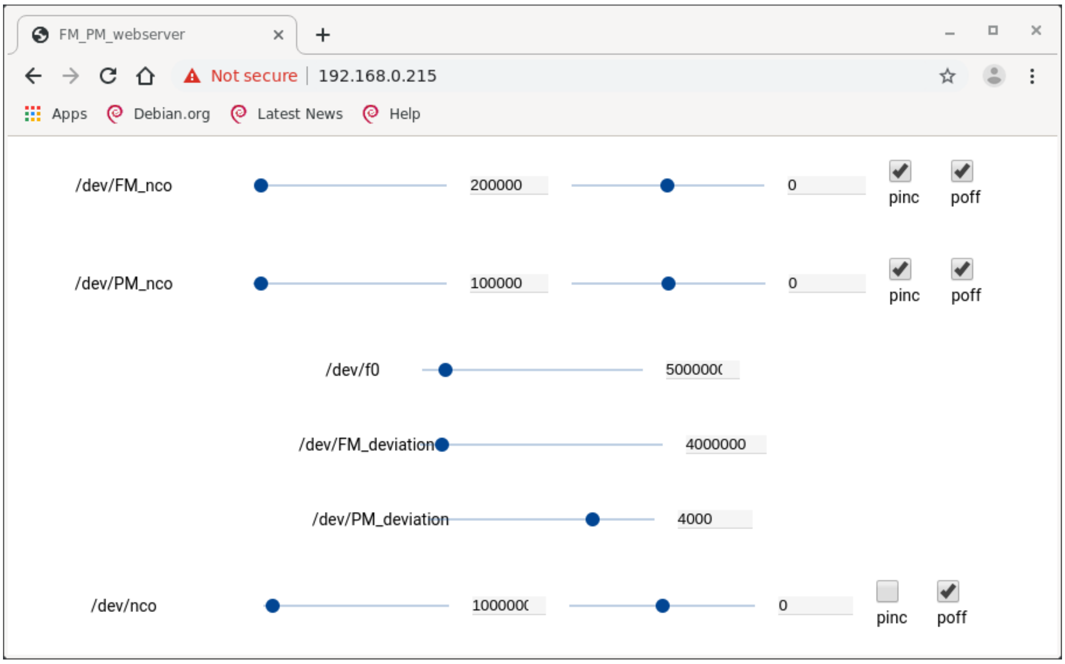



Figure 20: Screenshot of the phase and frequency modulation webserver. "pinc" checkbox is unchecked: frequency modulation.

To show an exaggerated example of the frequency modulation, we use a carrier frequency f0 of 5 $MHz$. The frequency of the FM_nco and the FM_deviation must remain lower than the carrier frequency. Here we used a frequency modulation of 200 $kHz$ and a deviation of 4 $MHz$. We expect the following output:

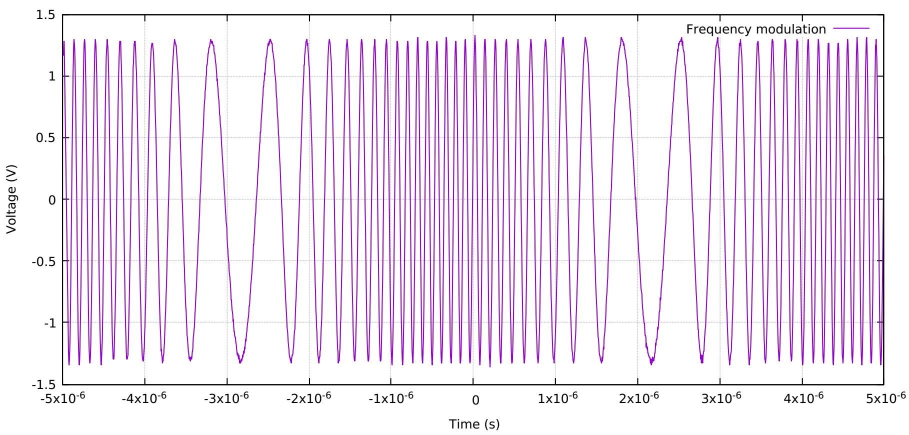



Figure 21: Expected frequency modulation with a carrier at 5 $MHz$, and a deviation of 4 $MHz$.

## 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 $\pm 8191$ *arb. unit* corresponds approximately to deviation of $\pm 4\pi$.

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

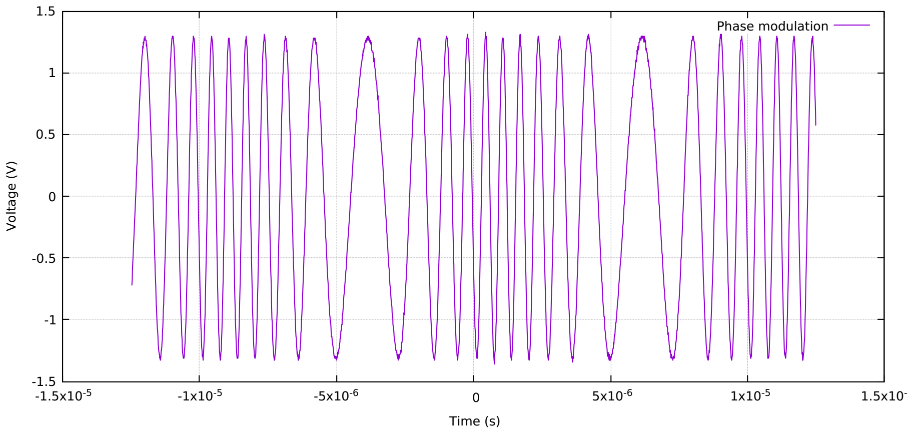



Figure 22: Expected phase modulation with a carrier at 1 $MHz$, and a deviation of $2\pi$..

## 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[2] with the function fir1[3] 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 =
128
-397
-1337
3775
```

[2] https://octave.sourceforge.io/signal/index.html

[3] https://octave.sourceforge.io/signal/function/fir1.html

```
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 $54th$ 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)
```

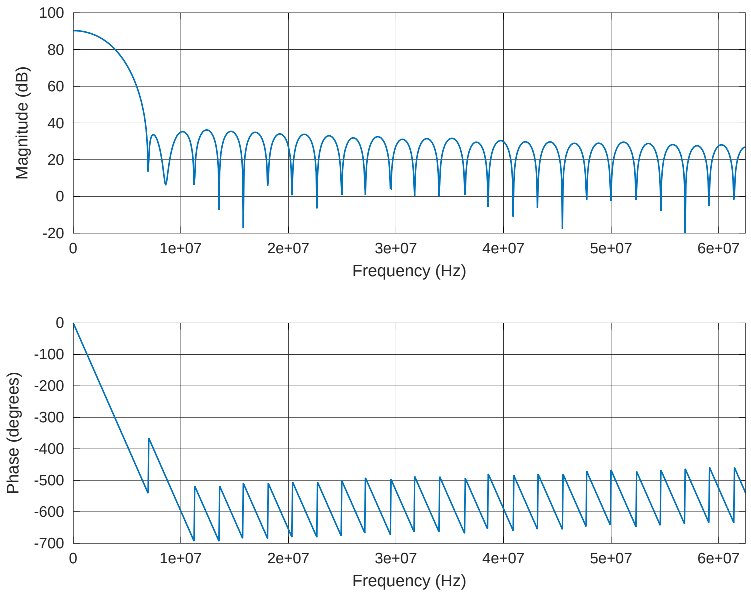



Figure 23: Lowpass filter at a 3 $MHz$ cutoff frequency with 55 coefficients.

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 $dB$ 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)
```

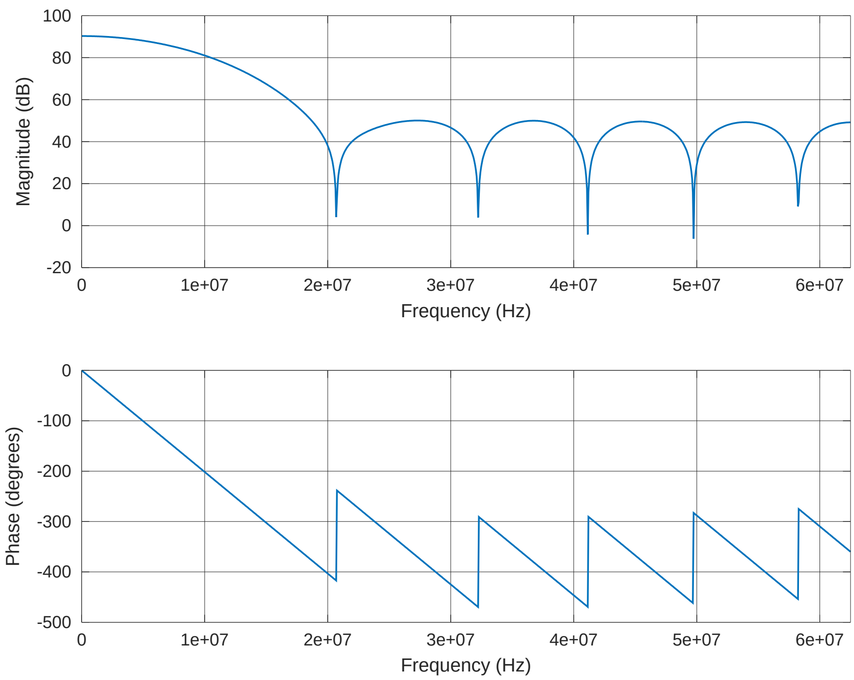



Figure 24: Lowpass filter at a 3 $MHz$ cutoff frequency with 15 coefficients.

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 $MHz - 25$ $MHz$ and 40 $MHz - 50$ $MHz$:

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

## 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 /oscimpDigital/lib/fir_conf.h file :

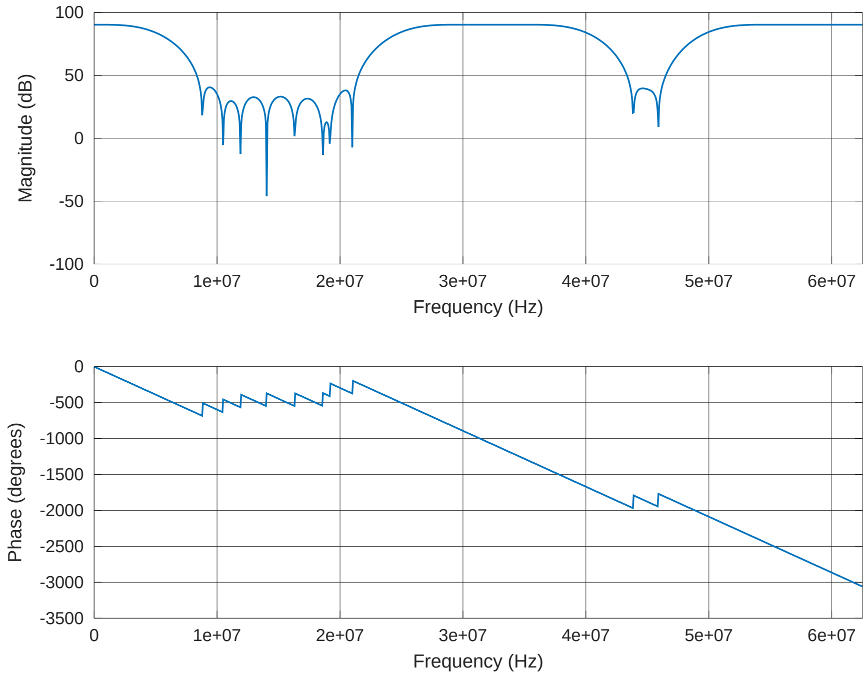



Figure 25: Bandstop filter at at 5 $MHz - 25$ $MHz$ and 40 $MHz - 50$ $MHz$, with 55 coefficients.

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

In the python wrapper liboscimp_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 :

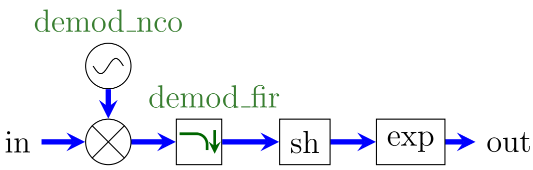
```
#!/usr/bin/env python

import liboscimp_fpga

liboscimp_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.

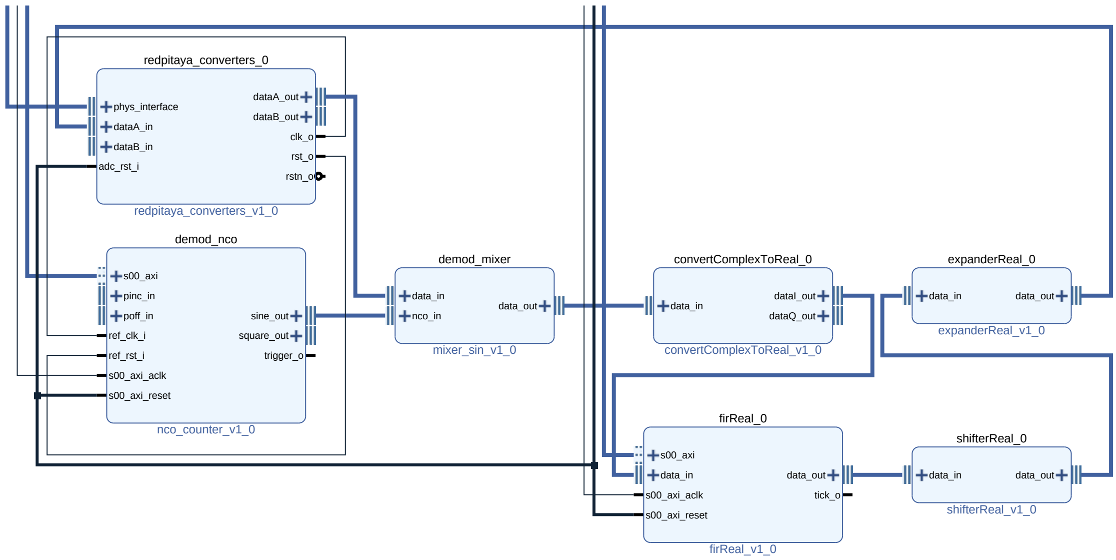



Figure 26: Part of the block diagram for the demodulation, including the filtering.

## 9.1 IP configuration

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

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

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

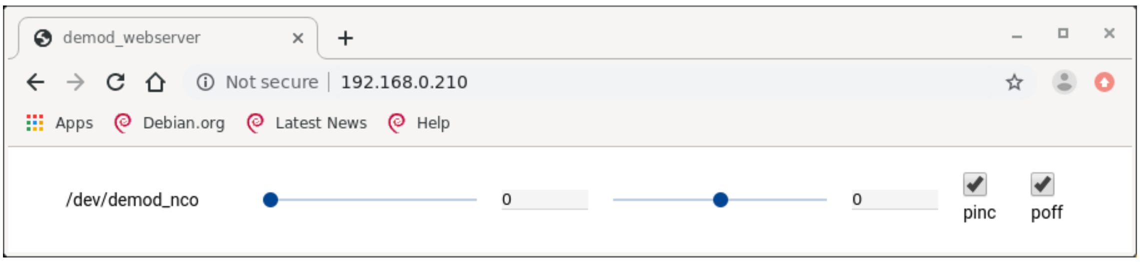



Figure 27: Screenshot of the demodulation webserver: only one nco is controlled here.

## 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 $MHz$, thus the demodulation frequency is set to 40 $MHz$. 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:

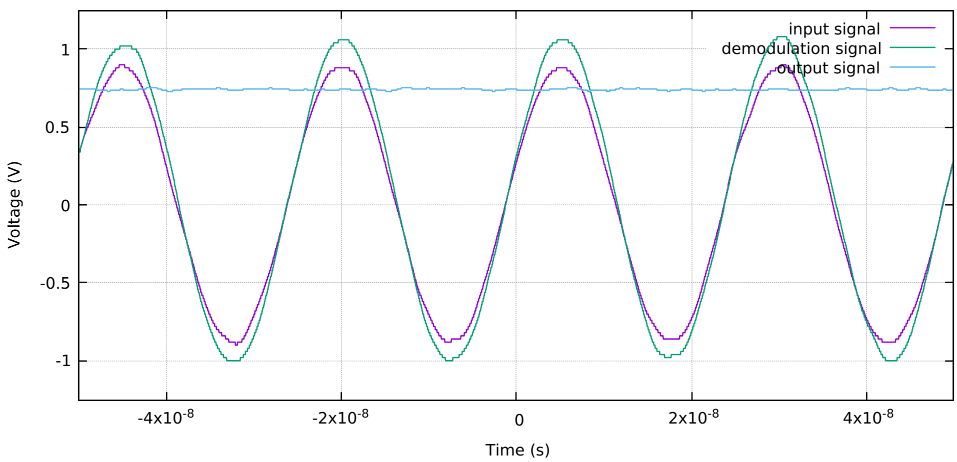



Figure 28: The carrier and the demodulation signal are in phase.

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

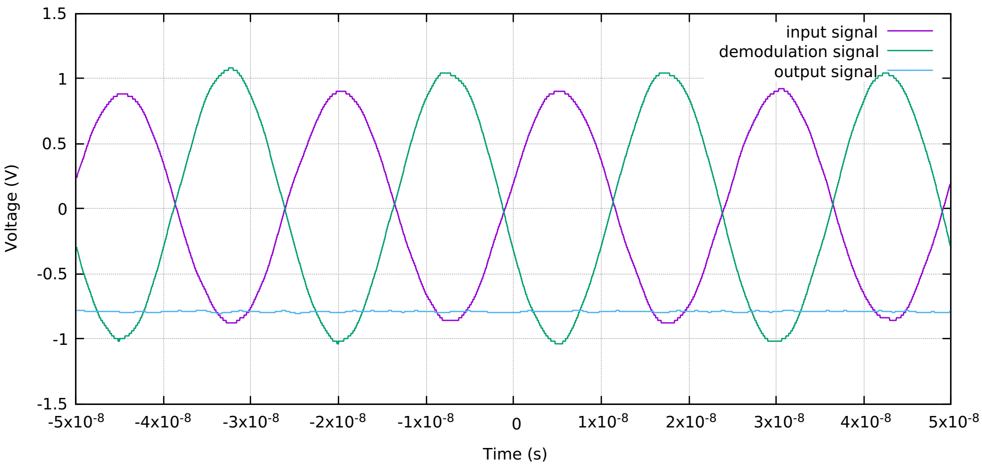



Figure 29: The carrier and the demodulation signal are in antiphase.

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

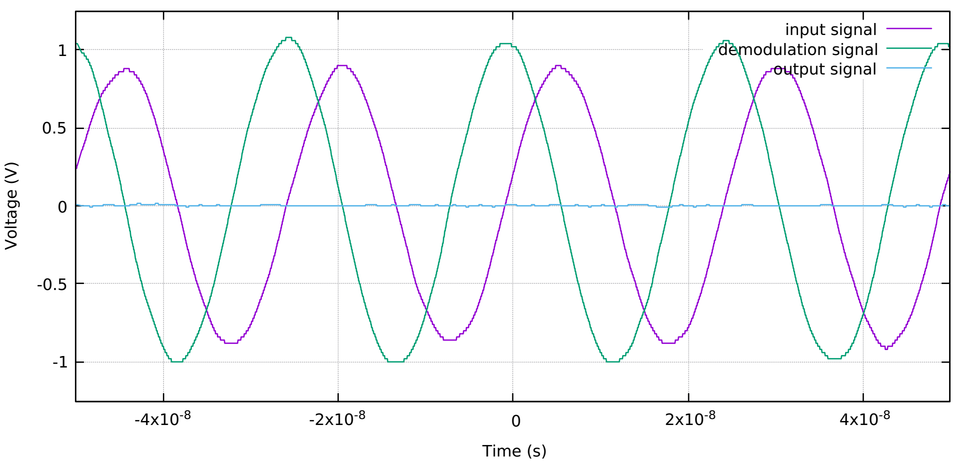



Figure 30: The carrier and the demodulation signal are in quadrature.

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 $MHz$, and the modulating signal is a sine function at 50 $kHz$, and a modulation depth of 50%:

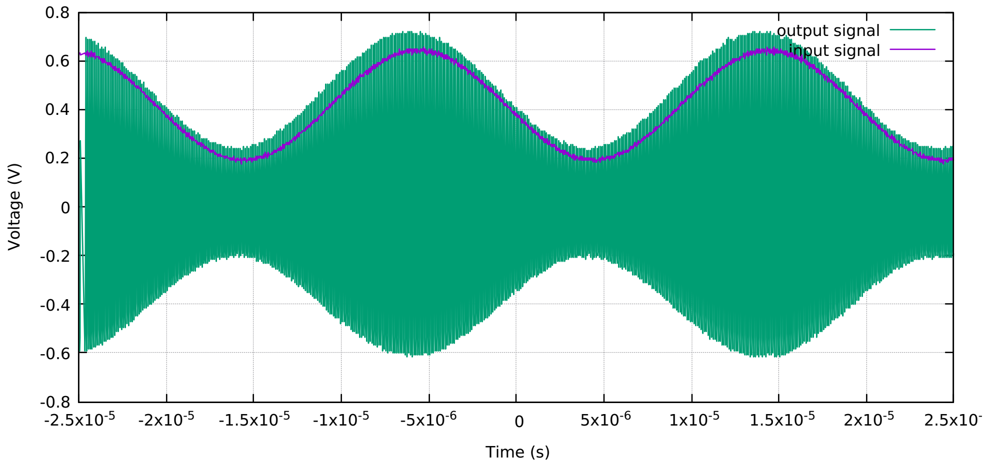



Figure 31: Amplitude demodulation.

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 $MHz$, and the modulating signal is a square signal at 50 $kHz$ with a deviation of 50°:

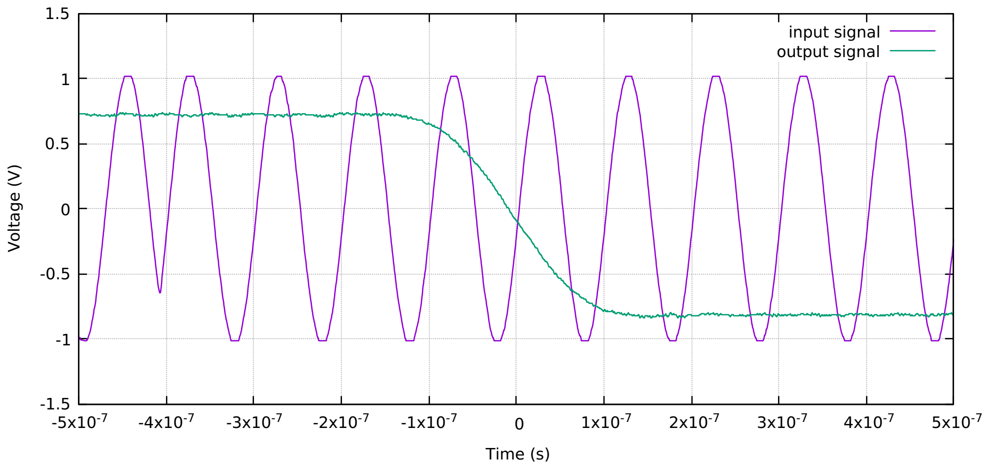



Figure 32: Phase demodulation.

In this figure, only the transition between the two levels of the square is visible, at the time $-410\ ns$ for the input signal and around $0\ s$ 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.

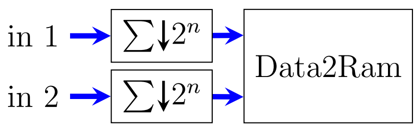
### 9.4 Unexpected output

- 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:

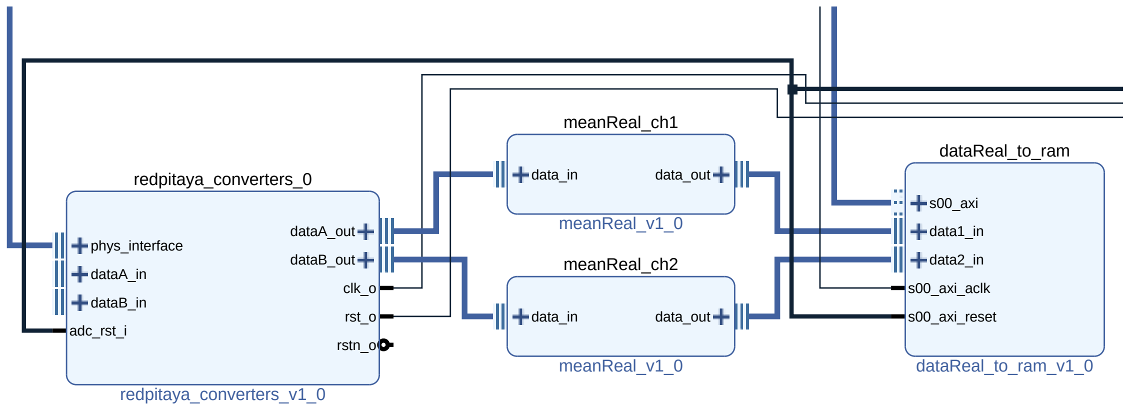



Figure 33: Part of the block diagram for monitoring with the data_to_ram block.

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 *bits*<br>Data format: signed<br>NB input: 2<br>Nb Sample: $= \ K$ |
| meanReal_ch1/2 | Input Data size: 14 *bits*<br>Output Data Size: 16 *bits*<br>Nb Accum: $= \ 2^n$ *bits*<br>Shift: $= \ n$ *bits* |

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\x0e\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[4],
2. Receiving and plotting the data using GNU Radio Companion[5] 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'))
```

[4]https://zeromq.org/
[5]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 remain 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:

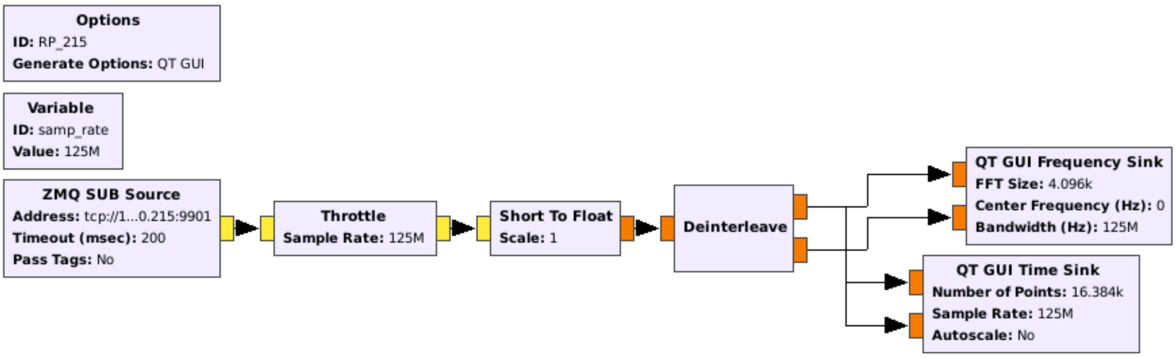



Figure 34: Flow graph for the data monitoring.

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 $MHz$), and with an averaging of $2^{13}$ (i.e. sampling rate of $\simeq 15.26\ kHz$).

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 $MHz$, and the signal in the second channel CH2 is a sine signal at 2 $kHz$.

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

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

- samp_rate: $125 MHz$
- 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 $s$
- Other parameters: up to you !

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

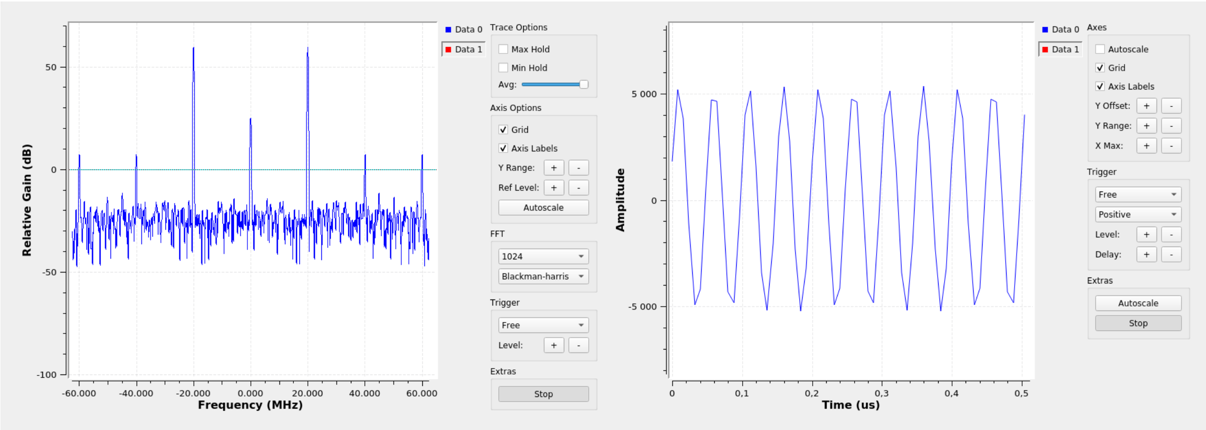



Figure 35: First channel at 20 $MHz$, no decimation.

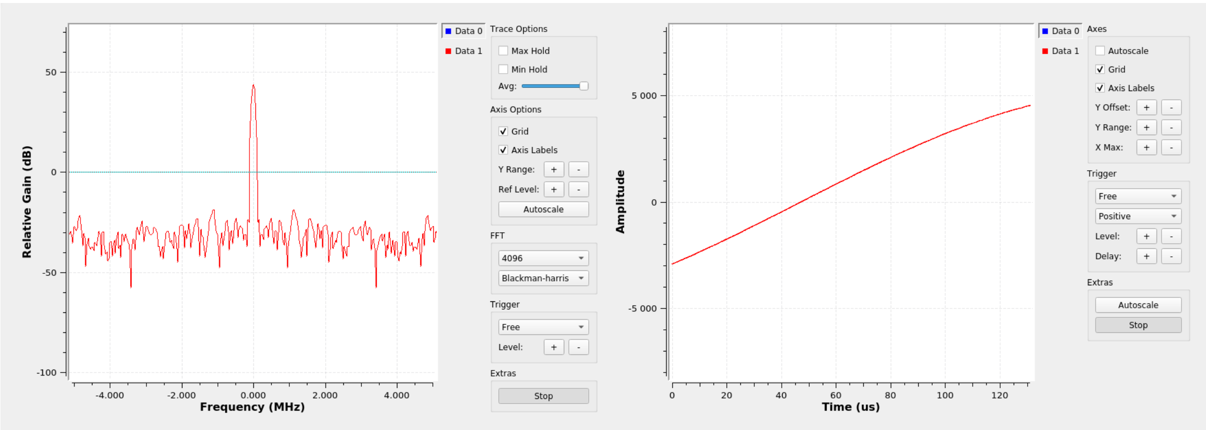



Figure 36: Second channel at 2 $kHz$, no decimation.

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, ant 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 $\Delta t_{buffer} = \dfrac{2048 \times 8192}{125 \cdot 10^6} = 1.34 \cdot 10^{-1}\ s.$

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:

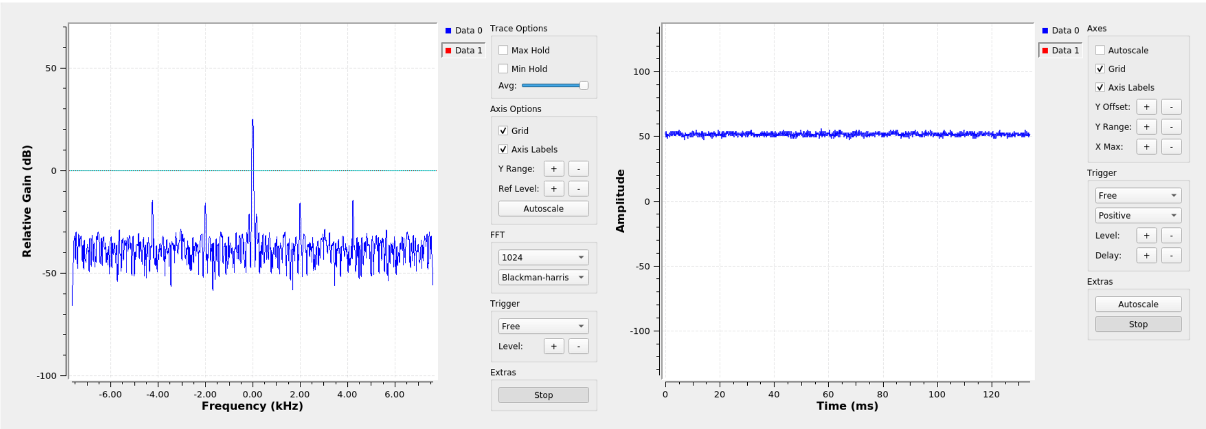



Figure 37: First channel at 20 $MHz$, with decimation.

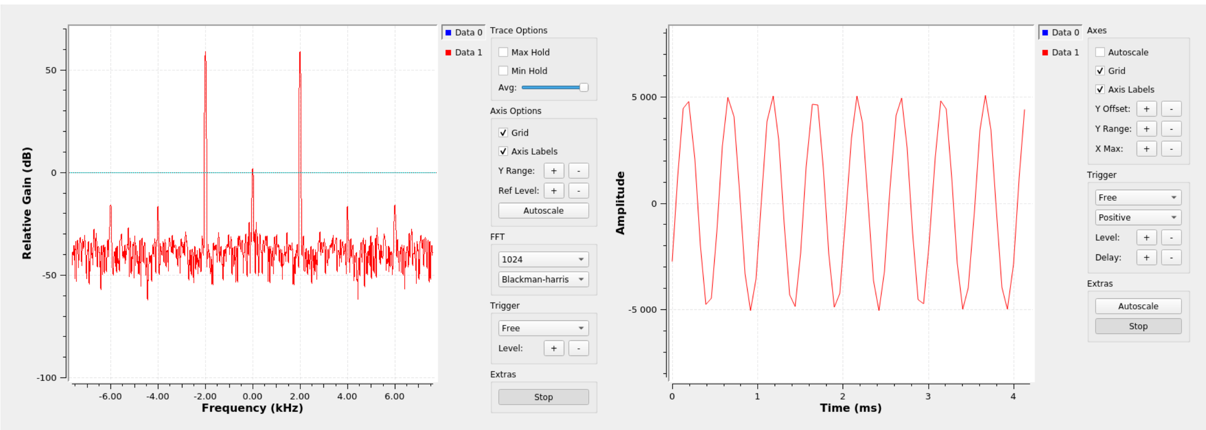



Figure 38: First channel at 2 $kHz$, with decimation.

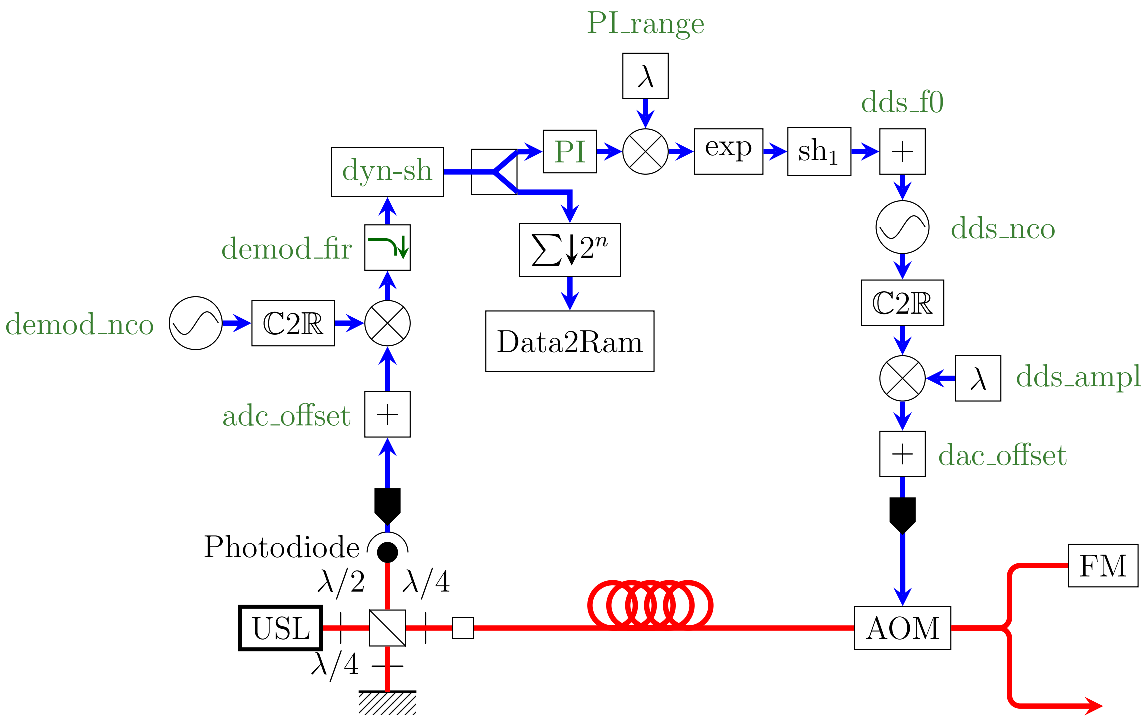
This time the signal at 20 $MHz$ is severely undersampled and can not be interpreted in the time domain. However the signal at 2 $kHz$ is this time visible with a sufficient resolution both in the time and the frequency domains.

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

| | |
|---|---|
| dataReal_to_ram | Data size: 16 *bits*<br>Data format: signed<br>NB input: 1<br>Nb Sample: = 16384 |
| IP | Configuration |
| PI | Data in/out size: 14 *bits*<br>I shift: 19 *bits*<br>I size: 18 *bits*<br>P shift: 13 *bits*<br>P size: 14 *bits* |
| PI_range<br>dds_ampl | Data size: 14 *bits* |
| exp | Data in size: 14 *bits*<br>Data out size: 20 *bits* |
| $sh_1$ | Data in size: 20 *bits*<br>Data out size: 40 *bits* |

Here, the expander (exp) and shifter ($sh_1$) 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 ($sh_1$) must be adapted.

## 11.2 Webserver configuration

Rather than providing static Python web generator scripts which might evolve with newer versions of REMI [6], we generate the web server using the following sequence:

1. assuming Vivado binaries are accessible in the $PATH and that the variables defined in `settings.sh` have been designed, then in the `design` directory execute: `make project && make xml` to generate the XML configuration file `file.xml` needed to configure `webserver_generator.py`;
2. execute `app/tools/webserver_generator/webserver_generator.py file.xml` to generate the webserver REMI Python script.

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.

[6] https://pypi.org/project/remi/

The dynamic shifter is also represented. It's increase corresponds to a division of the signal range in power of 2, and conversely it's decrease to a multiplication. It is used here to adapt the range of the dynamic of the signal to the range available, and therefore take advantage of the maximum resolution available.

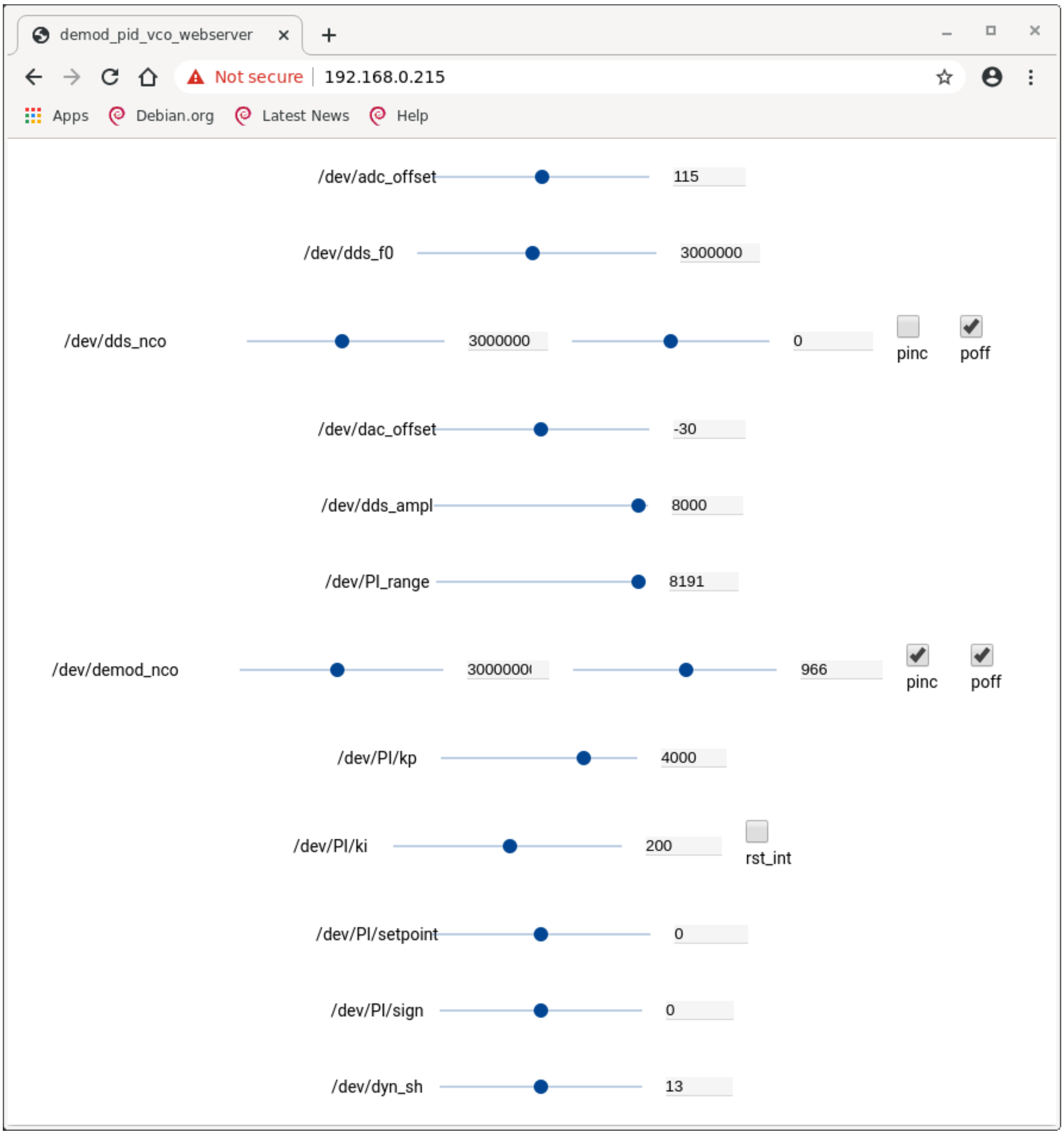
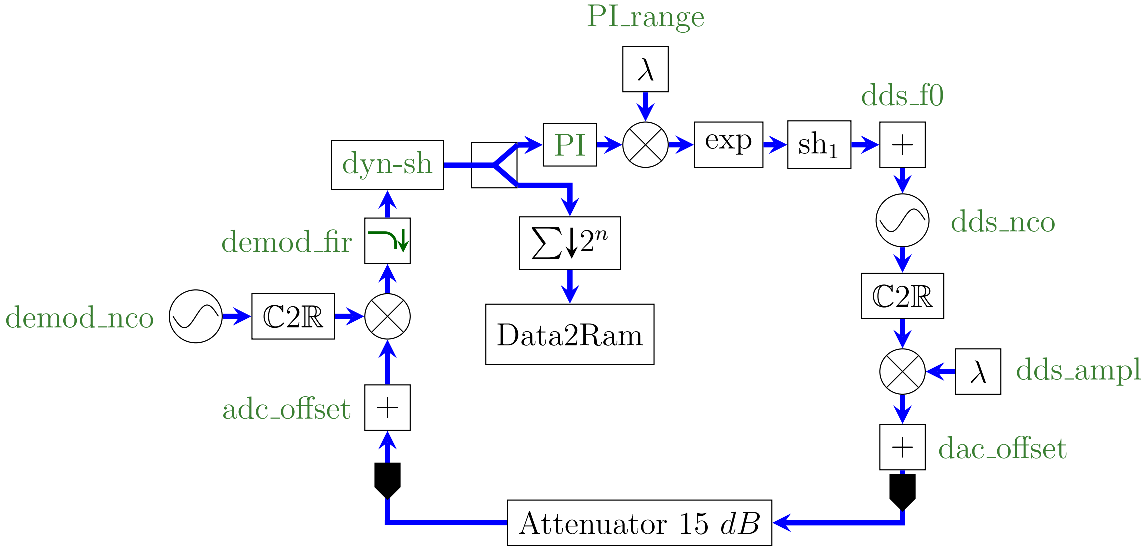



Figure 39: Screenshot of the control loop webserver.

### 11.3 Expected output

Although this design was developed for the lock of the phase noise in a fiber link, it can be used for any RF lock loop between 0 and 62.5 $MHz$. To make a demonstration of use of this design, we will therefore not use a full optical fiber link setup, but simply reconnect the output to the input. A 15 $dB$ attenuator is placed between the output and the input to decrease the input power, as shown below:




In the following demonstrations, we show step by step how to adjust the parameters of this design. We will proceed as follows:

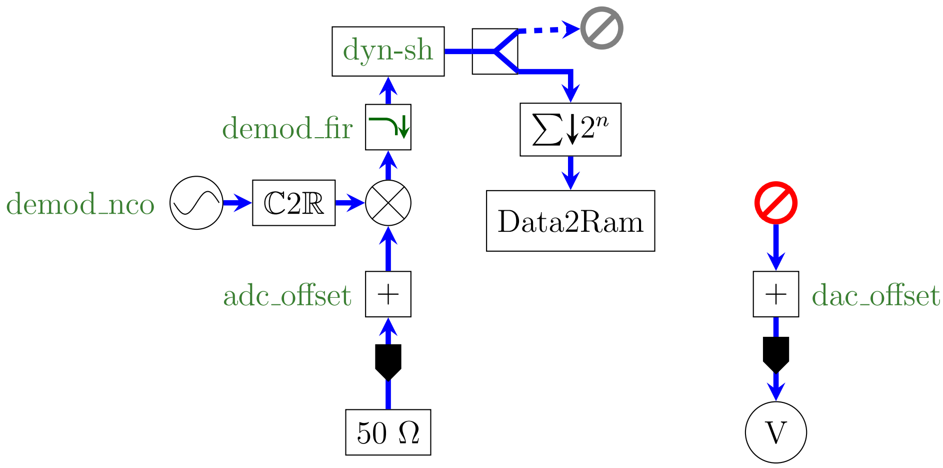
1. Load the fir coefficients
2. Adjust the ADC and DAC offsets
3. Make the modulation/demodulation chain
4. Adjust the dynamic range
5. Lock the noise of the small RF link
6. Phase lock the NCO on a second NCO

#### 11.3.1 Load the fir coefficients

The fir coefficients can be computed and loaded with the help of the section 8. In this example we used a lowpass filter with a cutoff frequency of 3 $MHz$ and 25 coefficients. In the case the modulation and demodulation frequency are fixed, the fir coefficients can be calculated so that the sum frequency resulting of the mixing is situated between two bounces of the frequency response of the fir.

### 11.3.2 Adjust the ADC and DAC offsets

Although the voltage offsets of the converters is not very high, depending on the application, it can be useful to compensate those offsets. The adc_offset and dac_offset blocks are used for this purpose. The adjustment of the adc_offset lays on the left part of the scheme below, and the adjustment of the dac_offset on its right part:




In order to deploy as few resources as possible, we show here methods using the presented design. However the voltage offset of the converters is a property of the board currently used (that may vary in the long term), and a method using a simpler design may give more precise results.

To compensate the ADC offset, we place a 50 Ω SMA resistor terminason at the input of the board. The principle of this adjustment is to set up a demodulation demod_nco that is only mixed with the signal resulting of the ADC voltage offset. Then the compensation of the offset is characterized by a minimization of the signal after the demodulation mixer, using the adc_offset block. To do this, proceed as follows:

1. Load the fir coefficients (if not done yet).
2. Put on the signal monitoring (see section 10)
3. Set a demodulation frequency that is not blocked by the fir, and included in the frequency range monitored by the Data2Ram (here 488 $kHz$). Here we choose a demodulation of 10 $kHz$.
4. Decrease the dyn-sh from 18 *bit* so that the signal range is roughly close to the maximum range (a sine above i.e. 8191 *arb. unit* in the time domain), and without overflow.
5. Adjust the adc_offset to minimize the demodulation sine.

This final step is represented in the figure below, before adjustment (adc_offset = 0 *arb. unit*) to the left, and after adjustment. Here we adjusted the adc_offset to 115 *arb. unit*, meaning an offset of around −115 *arb. unit*.

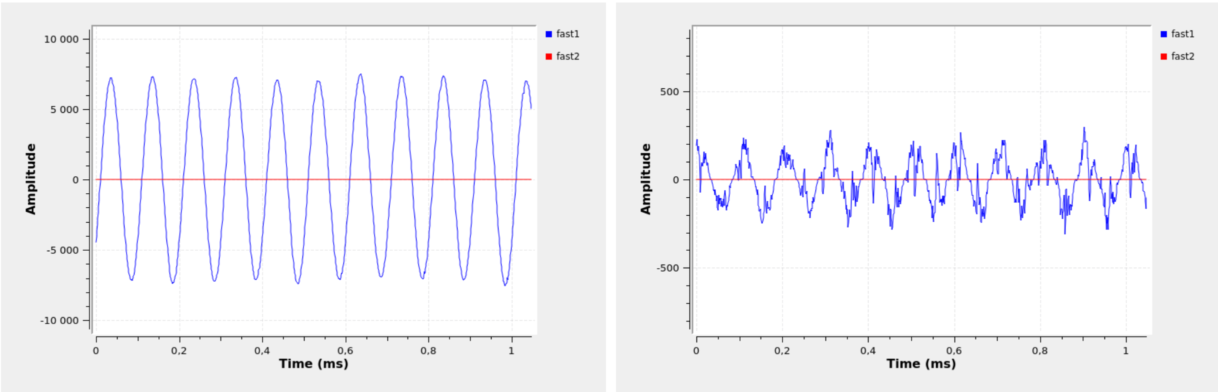



Figure 40: Demodulation signal before compensation of the offset with adc_offset = 0 *arb. unit* (left), and with the compensation with adc_offset = −115 *arb. unit* (right).

Thereafter to compensate the DAC offset voltage, the principle is just to measure it at the output with a voltmeter. The procedure is simpler:

1. Plug the voltmeter to the SMA output.
2. Disable the output DDS by setting its amplitude dds_ampl to 0 *arb. unit*.
3. Measure the voltage: it corresponds to the DAC offset.
4. Compensate the DAC offset using the dac_offset value, so that the measured voltages is brought to 0 $V$.

As an example, here we measured a DAC offset of 3.6 $mV$, compensated with a dac_offset −30 *arb. unit*.

### 11.3.3 Make the modulation/demodulation chain

To start using the design, we must generate the sine signal with the nco, and make its demodulation:

1. Reconnect the input to the output, with the attenuator in between.

2. Set the dds_nco frequency, here we use 30 $MHz$. The NCO offset is kept at 0, and the phase offset "poff" is kept internal.
3. Since the aim is to apply a correction to the frequency of the NCO, the "pinc" checkbox is unchecked so that the phase increment is external.
4. Set the parameter dds_f0 to the modulation frequency, here 30 $MHz$, so that the external phase increment without corrections corresponds to the desired modulation frequency.
5. Set the amplitude parameter dds_ampl close to the maximum, for instance 8000 *arb. unit*. Do not use the maximum value avoids to create an overflow resulting from the dac_offset value.
6. Align the demod_nco with the modulation frequency, i.e. 30 $MHz$ here.

Here it seems to be useless to set the value dds_nco since only the value of dds_f0 is taken into account, with the external increment. However, this external frequency will be modified by the correction signal of the lock loop, and can quickly drift if the lock gains are not well adjusted. In this case, a reset of the modulation frequency can be performed by switching between an external and an internal phase increment.

Then the dynamic range must be adapted before to choose either to perform an amplitude or a phase demodulation. This avoids to perform a wrong adjustment due to some overflow.

### 11.3.4 Adjust the dynamic range

The dynamic range is primarily affected by the input power of the signal, and this one may be adapted first (see section 1). In case it is not desired or for thinner adjustments, then the dynamic shifter can be used To adjust the dynamic range of the signal after the fir:

1. Slightly shift the demodulation frequency to make a beatnote between the modulation and demodulation signals, for instance a 10 $kHz$ shift. This beatnote shows the range of the signal after mixing.
2. Increase the shift of the dynamic shifter from 0 until the 10 $kHz$ beatnote is well drown, and without overflow. Here we set is to $\pm 13$ *bits*.
3. If there is an overflow, increase the shift.
4. If the range is below $\pm 4000$ *arb. unit*, decrease the shift.

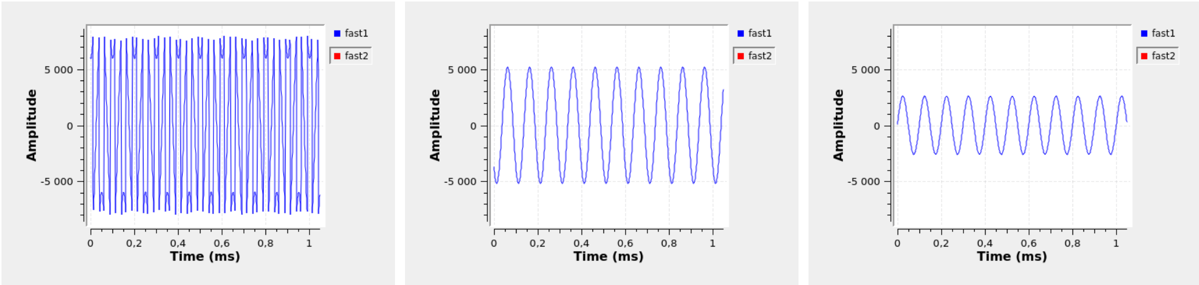



Figure 41: Dynamic shift of 12 ✗, 13 ✓, and 14 ✗.

#### 11.3.5 Lock the noise of the small RF link

This design offers two possibilities: implement a phase lock loop (PLL) or an amplitude lock loop (ALL). The PLL is set up with a phase demodulation (modulation and demodulation are in quadrature, see section 9) while the ALL is set up with an amplitude demodulation (modulation and demodulation are in phase).

To make the demonstration of this design more visual, we will set up here an ALL. The procedure to adjust the lock loop is as following:

1. Make sure the PI integrator is reset: the "rst_int" checkbox of the PI/ki gain must be checked.
2. Make sure the "pinc" checkbox of the dds_nco is unchecked, to allow the reception of the corrected phase increment.
3. Choose the amplitude or phase demodulation, using the phase offset slider of the demod_nco. Here we choose the amplitude demodulation:

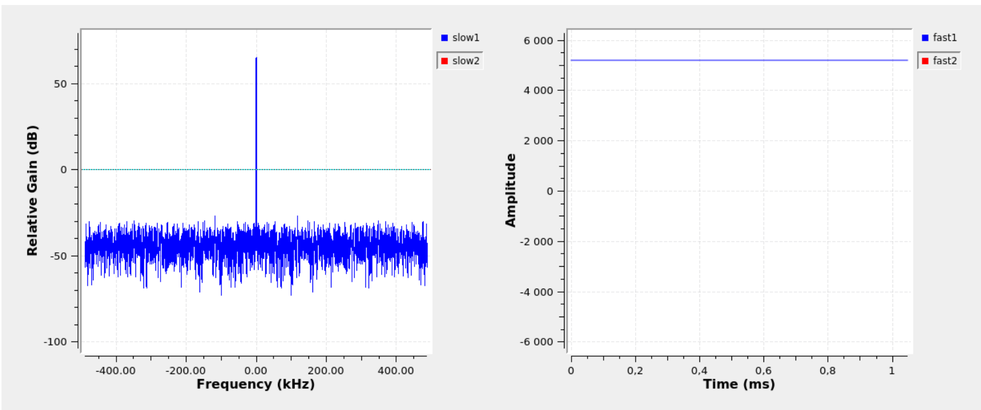



Figure 42: Signal after amplitude demodulation. No gain yet.

4. Initially set the PI range to the maximum. It may be readjusted later.
5. Gradually increase the PI/kp gain from 0 until the system starts to oscillate, and set the PI/kp gain roughly to a value before the oscillation, see fig.43.

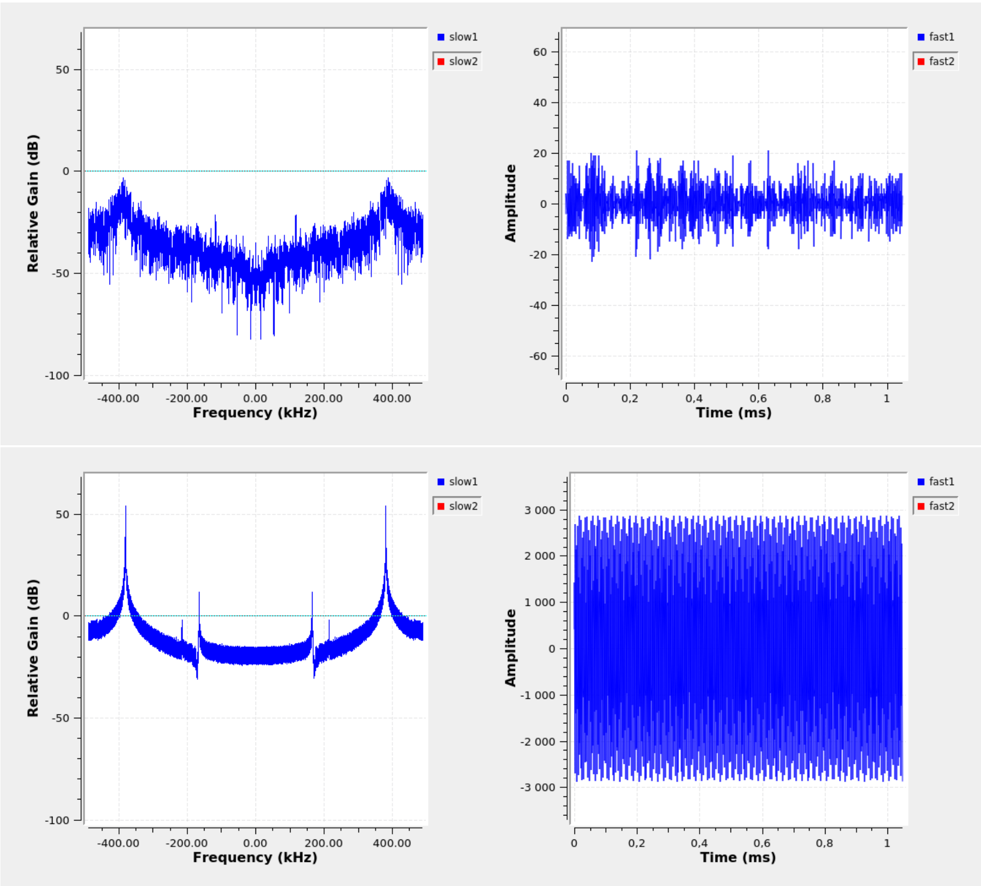



Figure 43: System close to oscillation above, and system oscillating below.

6. If no oscillation is observed and if the signal is below ±4000 *arb. unit*, decrease the shift of the dynamic shifter and make the previous adjustment. However if the system oscillates, return to the previous shift.
7. If the oscillations occur for a too small gain, either increase the shift of the dynamic shifter, or decrease the value of PI_range.
8. Set the PI/ki gain to 0 and enable the integrator (i.e. uncheck the rst_int checkbox).
9. Gradually increase the PI/ki gain from 0. Before reaching the oscillations, there should be a PI/ki gain for which the thickness of the error signal is minimum.
10. If the system oscillates, reset and disable the integrator, decrease the PI/ki gain and continue the previous adjustment.
11. Readjust eventually the PI/kp gain to minimize the thickness of the error signal.

12. Once the system is locked roughly (see fig.44) and depending on the application, thinner adjustments of the gains can be performed: noise analysis on the signal, response to a perturbation...

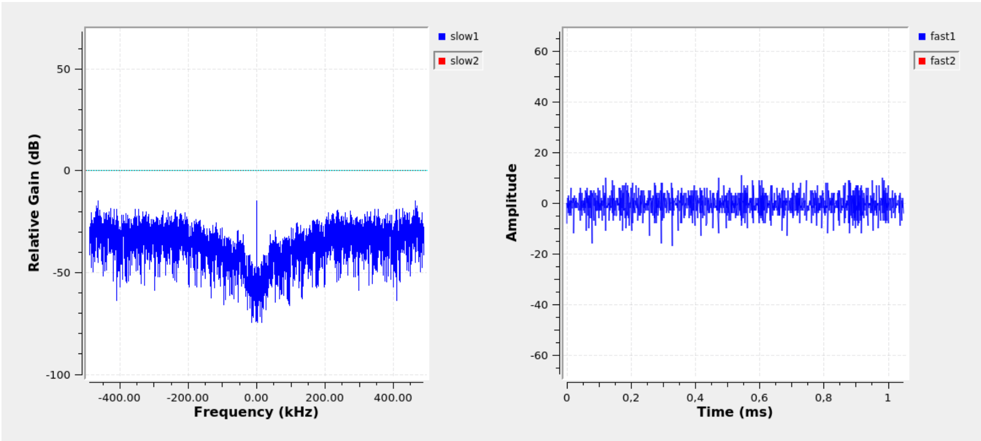



Figure 44: Signal locked. $PI/kp = 4000$ $PI/ki = 200$.

Although this protocol is presented with the example of an amplitude demodulation, it is also valid with a phase demodulation for a PLL. In this case the efficiency of the lock can be tested by locking the locking the NCO on the phase of a second NCO.

### 11.3.6 Frequency reference

Although the internal reference clock is a good base, its drift in the long term can be a problem for the transfer of an ultra stable frequency. In this case an external frequency reference with a better stability can be used. See:
https://redpitaya.readthedocs.io/en/latest/developerGuide/125-14/extADC.html.

# 12 FAQ

- **What are "pinc" and "poff"?**
  *"pinc" and "poff" for phase increments and phase offsets, are actuators represented by checkboxes, that allows to choose to have either internal or external phase increments or phase offsets of the NCO. If the phase increments and offsets are internal, the NCO is independent. If it is external, the NCO must have phase increments and offsets inputs.*
- **What is an overflow?**
  *An overflow is situation where the signal dynamic exceed the dynamic range available.*

*This leads to a misinterpretation or a saturation of the exceeding signal. With a digital signal the exceeding signal returns to the beginning of the range available.*

- **How to make a buildroot for the redpitaya board?**
  *See https://github.com/trabucayre/redpitaya*