pyrpl Documentation

Release 0.9.4.0

Leonhard Neuhaus

Nov 19, 2017

Contents

PyRPL is an open-source software package providing many instruments on cheap FPGA hardware boards, e.g.:

- oscilloscopes,
- network analyzers,
- lock-in amplifiers,
- multiple automatic feedback controllers,
- digital filters of very high order (24),
- and much more.

PyRPL currently runs exclusively on the Red Pitaya.

The Red Pitaya (a.k.a. STEM Lab) (http://www.redpitaya.com, see full documentation) is an affordable (ca. 260 Euros) FPGA board with fast (125 MHz) analog inputs and outputs.

PyRPL comes with a graphical user interface (GUI).

See our *GUI manual* or the video tutorial on youtube.

PyRPL has a convenient Python API.

See *High-level API example* or *Low-level API example*, and the *full API documentation* .

PyRPL binary executables for Windows, Linux, or Mac OS X

can be easily *downloaded* and run without any installation work.

PyRPL's code is entirely public on github and can be customized,

including the Verilog source code for the FPGA which is based on the official Red Pitaya software version 0.95.

PyRPL is already used in many research groups all over the world.

See for yourself the user_feedback.

PyRPL is free software and comes with the GNU General Public License v3.0.

Read the license for more details!

CHAPTER 1

Manual

1.1 Installation

1.1.1 Preparing the hardware

For PyRPL to work, you must have a working Red Pitaya / StemLab (official documentation) connected to the same local area network (LAN) as the computer PyRPL is running on. PyRPL is compatible with all operating system versions of the Red Pitaya and does not require any customization of the Red Pitaya. If you have not already set up your Red Pitaya:

- download and unzip the Red Pitaya OS Version 0.92 image,
- flash this image on 4 GB (or larger) micro SD card using Win32DiskImager (see a step-by-step guide for all operating systems), and insert the card into your Red Pitaya, and
- connect the Red Pitaya to your LAN and connect its power supply.

user_guide/installation/hardware_installation gives more detailed instructions in case you are experiencing any trouble.

1.1.2 Installing PyRPL

The easiest and fastest way to get PyRPL running is to download and execute the latest precompiled executable for

- **windows**: pyrpl-windows.exe,
- **linux**: pyrpl-linux, or
- **Mac OS X**: pyrpl-mac.

If you prefer an installation from source code, go to installation_from_source.

1.1.3 Compiling the FPGA code (optional)

A ready-to-use FPGA bitfile comes with PyRPL. If you want to build your own, possibly customized bitfile, go to *Building the FPGA firmware*.

1.2 GUI instruments manual

In this section, we show how to control the main modules of Pyrpl with the Graphical User Interface (GUI).

1.2.1 Video tutorial

Get started by watching the video tutorial below on locking a Michelson interferometer with PyRPL. The video covers:

- how to set up the hardware for a typical Red Pitaya use case (interferometer locking)
- how to get started by *Starting the GUI*
- how to use the *Scope Widget* and Arbitrary Signal Generator GUI
- how to set up and configure the *Lockbox Widget*
- how to measure a transfer function with the *Network Analyzer Widget*

1.2.2 Starting the GUI

If you use the windows or linux binary files, just launch the executable and the GUI should start. Passing the command-line argument `--help` to the executable shows a list of optional command-line arguments.

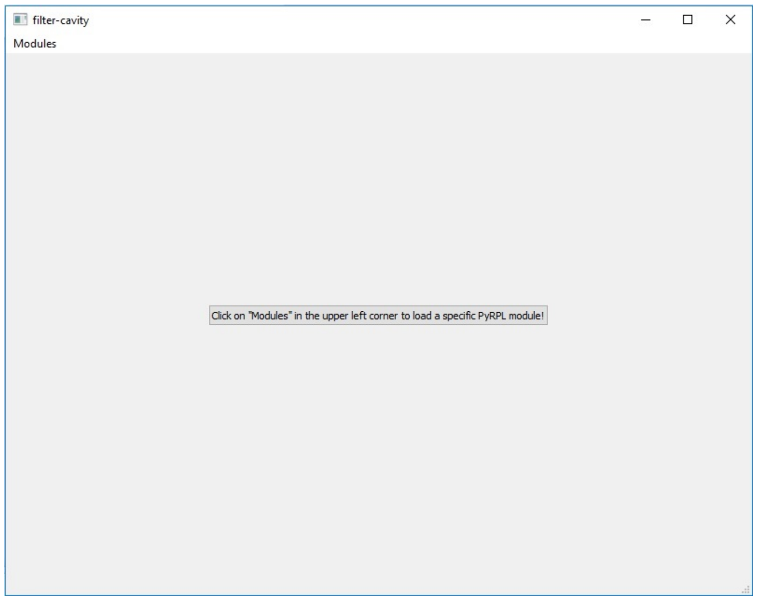
If instead you have a source code installation, then you can either launch PyRPL from a terminal with

```
python -m pyrpl example_filename
```

or execute the following code block in Python:

```python
# import pyrpl library
import pyrpl

# create a Pyrpl object and store the configuration in a file 'example_filename.yml'
# by default, the parameter 'gui' is set to True
p = pyrpl.Pyrpl(config='example_filename')
```

If you are using the file 'example_filename.yml' for the first time, a screen will pop-up asking you to choose among the different RedPitayas connected to your local network. After that, the main Pyrpl widget should appear:



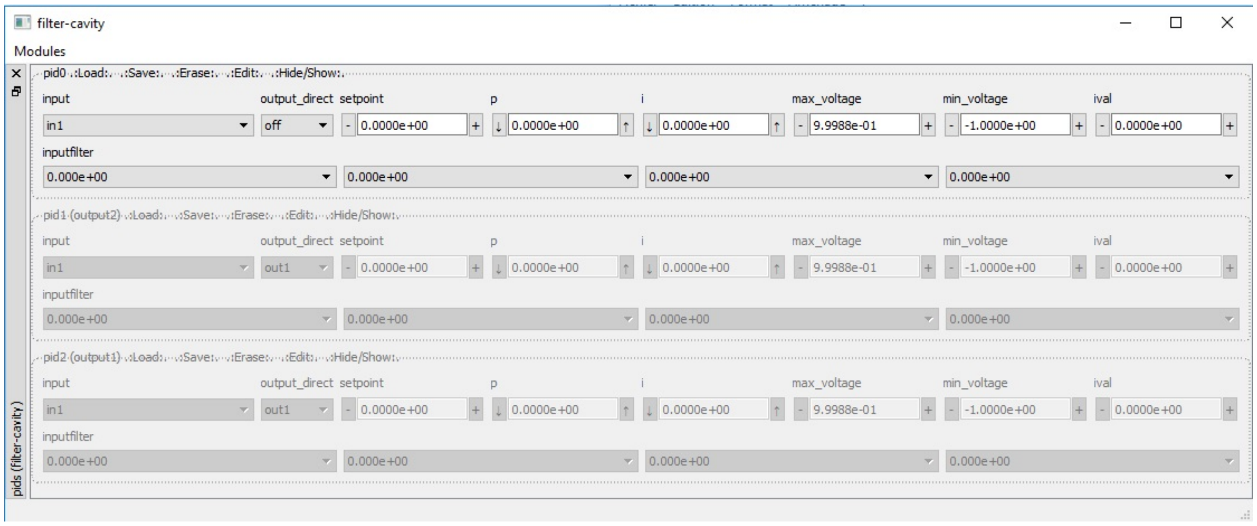
Click on "Modules" in the upper left corner to load a specific PyRPL module!

The main pyrpl widget is initially empty, however, you can use the "modules" menu to populate it with module widgets. The module widgets can be closed or reopened at any time, docked/undocked from the main module window by drag-and-drop on their sidebar, and their position on screen will be saved in the config file for the next startup.

We explain the operation of the most useful module widgets in the following sections.

1.2.3 A typical module widget: PID module

The image below shows a typical module widget, here for the PID modules.



pid0 .:Load:. .:Save:. .:Erase:. .:Edit:. .:Hide/Show:.

input	output_direct	setpoint	p	i	max_voltage	min_voltage	ival
in1	off	0.0000e+00	0.0000e+00	0.0000e+00	9.9988e-01	-1.0000e+00	0.0000e+00

inputfilter: 0.000e+00 | 0.000e+00 | 0.000e+00 | 0.000e+00

pid1 (output2) .:Load:. .:Save:. .:Erase:. .:Edit:. .:Hide/Show:.

input	output_direct	setpoint	p	i	max_voltage	min_voltage	ival
in1	out1	0.0000e+00	0.0000e+00	0.0000e+00	9.9988e-01	-1.0000e+00	0.0000e+00

inputfilter: 0.000e+00 | 0.000e+00 | 0.000e+00 | 0.000e+00

pid2 (output1) .:Load:. .:Save:. .:Erase:. .:Edit:. .:Hide/Show:.

input	output_direct	setpoint	p	i	max_voltage	min_voltage	ival
in1	out1	0.0000e+00	0.0000e+00	0.0000e+00	9.9988e-01	-1.0000e+00	0.0000e+00

inputfilter: 0.000e+00 | 0.000e+00 | 0.000e+00 | 0.000e+00

The basic functionality of all module widgets are inherited from the base class `ModuleWidget`.

A module widget is delimited by a dashed-line (a QGroupBox). The following menu is available on the top part of each ModuleWidget, directly behind the name of the module (e.g. `pid0`, `pid1`, ...). Right click on the item (e.g. `.:Load:.`, `.:Save:.`, ...) to access the associated submenu:

- `.:Load:.` Loads the state of the module from a list of previously saved states.
- `.:Save:.` Saves the current state under a given state name.

- `.:Erase:.` Erases one of the previously saved states.
- `.:Edit:.` Opens a text window to edit the yml code of a state.
- `.:Hide/Show:.` Hides or shows the content of the module widget.

Inside the module widget, different attribute values can be manipulated using the shown sub-widgets (e.g. `input`, `setpoint`, `max_voltage`, ...). The modifications will take effect immediately. Only the module state `<current state>` is affected by these changes. Saving the state under a different name only preserves the state at the moment of saving for later retrieval.

At the next startup with the same config file, the :code:<current state> of all modules is loaded.

If a module-widget is grayed out completely, it has been reserved by another, higher-level module whose name appears in parentheses after the name of the module (e.g. `pid2 (output1)` means that the module `pid2` is being used by the module `output1`, which is actually a submodule of the `lockbox` module). You can right-click anywhere on the grayed out widget and click on "Free" to override that reservation and use the module for your own purposes.

Warning: If you override a module reservation, the module in parenthesis might stop to function properly. A better practice is to identify the top-level module responsible for the reservation, remove its name from the list in your configuration file (located at /HOME/pyrpl_user_dir/config/<string_shown_in_top_bar_of_the_gui>.yml) and restart PyRPL with that configuration.

1.2.4 Acquisition Module Widgets

Acquisition modules are the modules used to acquire data from the Red Pitaya.

At the moment, they include the

- `scope`,
- `network_analyzer`,
- `spectrum_analyzer`.

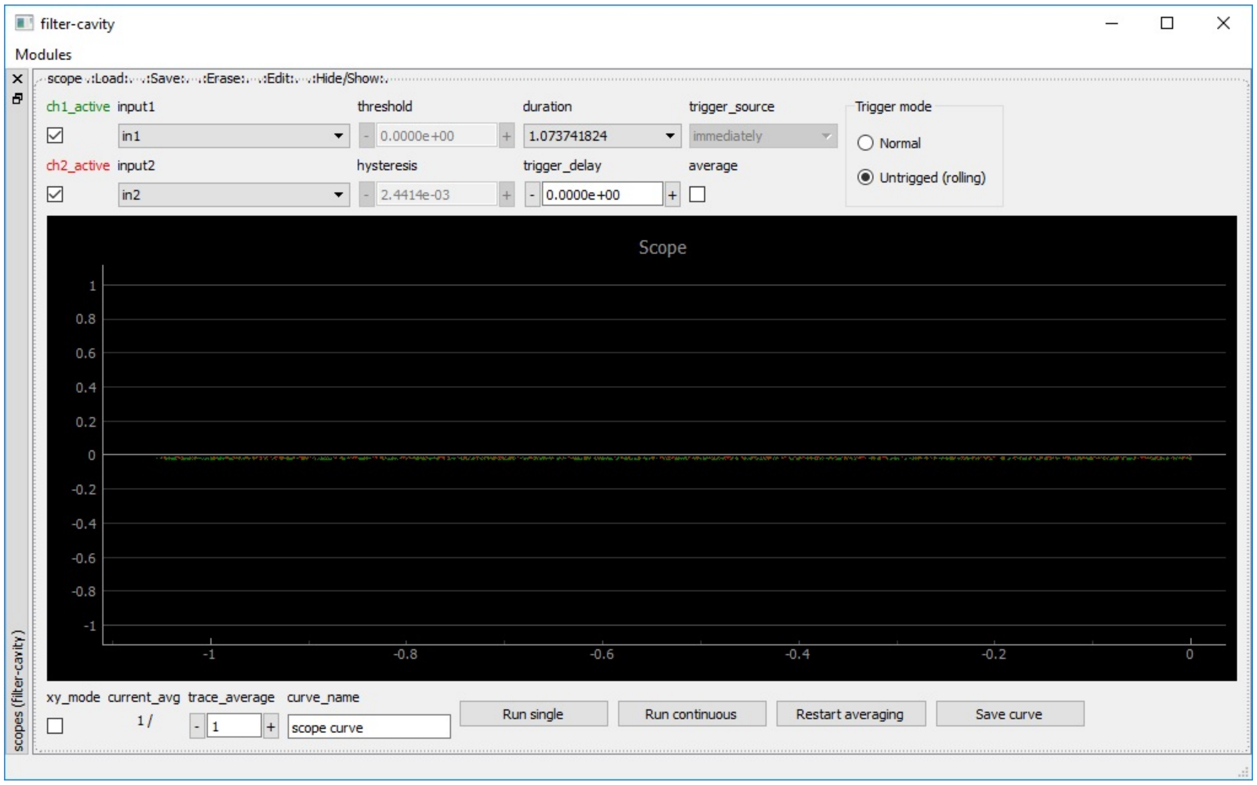
All the acquisition modules have in common a plot area where the data is displayed and a control panel BELOW the plot for changing acquisition settings. Widgets for specialized acquisition modules (e.g. `Scope`) have an additional control panel ABOVE the plot are for settings that are only available for that module.

The different buttons in the acquisition module control panel below the plot are:

- `trace_average` chooses the number of successive traces to average together.
- `curve_name` is the name for the next curve that is saved.
- `Run single` starts a single acquisition of `trace_average` traces (calls `AcquisitionModule.single()`).
- `Run continuous` starts a continuous acquisition with a running average filter, where `trace_average` is the decay constant of the running average filter (calls `AcquisitionModule.continuous()`).
- `Restart average` resets trace averages to zero to start a new measurement from scratch.
- `Save curve` saves the current measurement data to a new `pyrpl.curvedb.CurveDB` object under the name `curve_name`.

Scope Widget

The scope widget is represented in the image below.



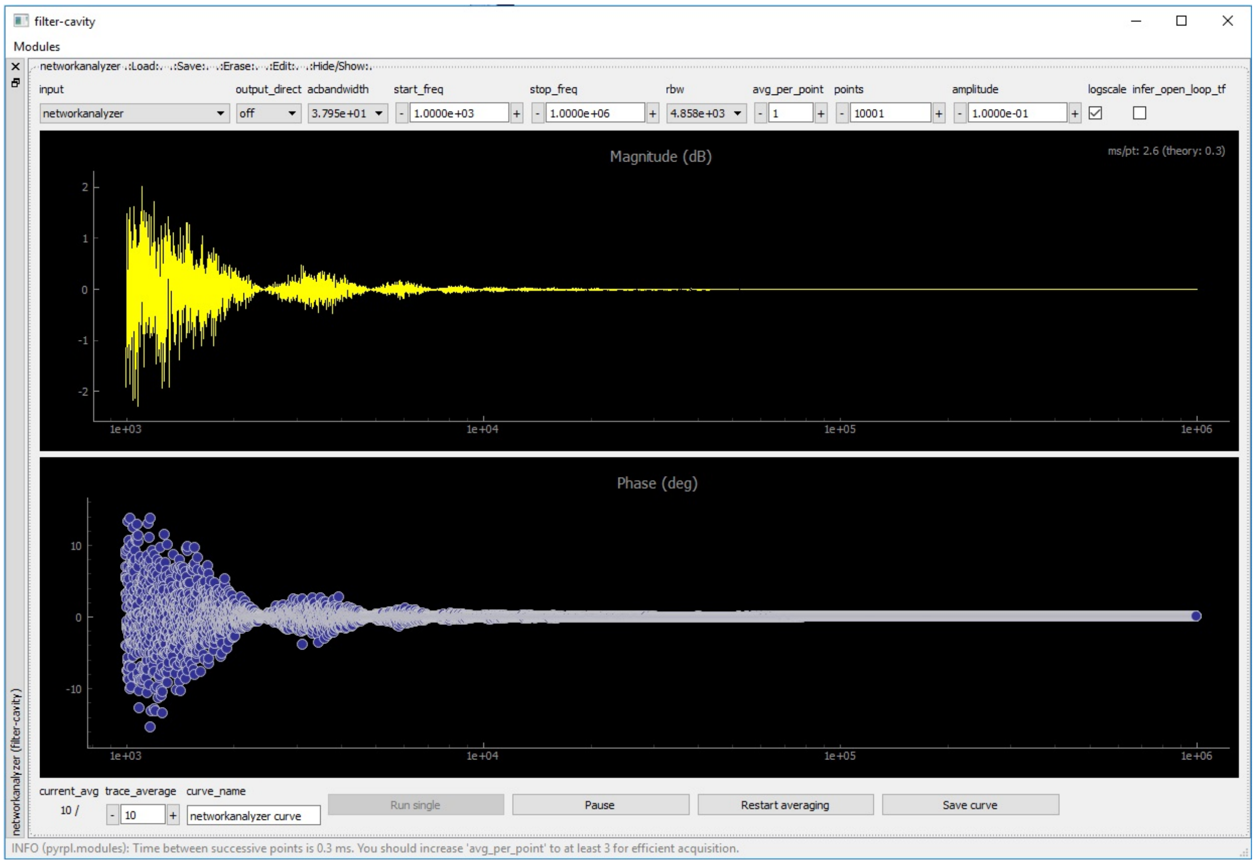
ch1_active: ☑ input1: in1 threshold: 0.0000e+00 duration: 1.073741824 trigger_source: immediately
ch2_active: ☑ input2: in2 hysteresis: 2.4414e-03 trigger_delay: 0.0000e+00 average: ☐
Trigger mode: Normal [✓ Untrigged (rolling)]
xy_mode: ☐ current_avg: 1 / trace_average: 1 curve_name: scope curve

The control panel above the plotting area allows to manipulate the following attributes specific to the `Scope`:

- `ch1_active/ch2_active`: Hide/show the trace corresponding to ch1/ch2.
- `input1/input2`: Choose the input among a list of possible signals. Internal signals can be referenced by their symbolic name e.g. `lockbox.outputs.output1`.
- `threshold`: The voltage threshold for the scope trigger.
- `hysteresis`: Hysteresis for the scope trigger, i.e. the scope input signal must exceed the `threshold` value by more than the hysteresis value to generate a trigger event.
- `duration`: The full duration of the scope trace to acquire, in units of seconds.
- `trigger_delay`: The delay beteween trigger event and the center of the trace.
- `trigger_source`: The channel to use as trigger input.
- `average`: Enables "averaging" a.k.a. "high-resolution" mode, which averages all data samples acquired at the full sampling rate between two successive points of the trace. If disabled, only a sample of the full-rate signal is shown as the trace. The averaging mode corresponds to a moving-average filter with a cutoff frequency of `sampling_time` $^{-1} = 2^{14}/\text{duration}$ in units of Hz.
- `trigger_mode`: Multiple options are available.
  - `Normal` is used for triggered acquisition.
  - `Untriggered (rolling)` is used for continuous acquisition without requiring a trigger signal, where the traces "roll" through the plotting area from right to left in real-time. The rolling mode does not allow for trace averaging nor durations below 0.1 s.

Network Analyzer Widget

The network analyzer widget is represented in the image below.



input: networkanalyzer
output_direct: off
acbandwidth: 3.795e+01
start_freq: 1.0000e+03
stop_freq: 1.0000e+06
rbw: 4.858e+03
avg_per_point: 1
points: 10001
amplitude: 1.0000e-01
logscale: ☑
infer_open_loop_tf: ☐

ms/pt: 2.6 (theory: 0.3)

current_avg: 10 /
trace_average: 10
curve_name: networkanalyzer curve

INFO (pyrpl.modules): Time between successive points is 0.3 ms. You should increase 'avg_per_point' to at least 3 for efficient acquisition.

The network analyzer records the coherent response of the signal at the port `input` to a sinusoidal excitation of variable frequency sent to the output selected in `output_direct`.

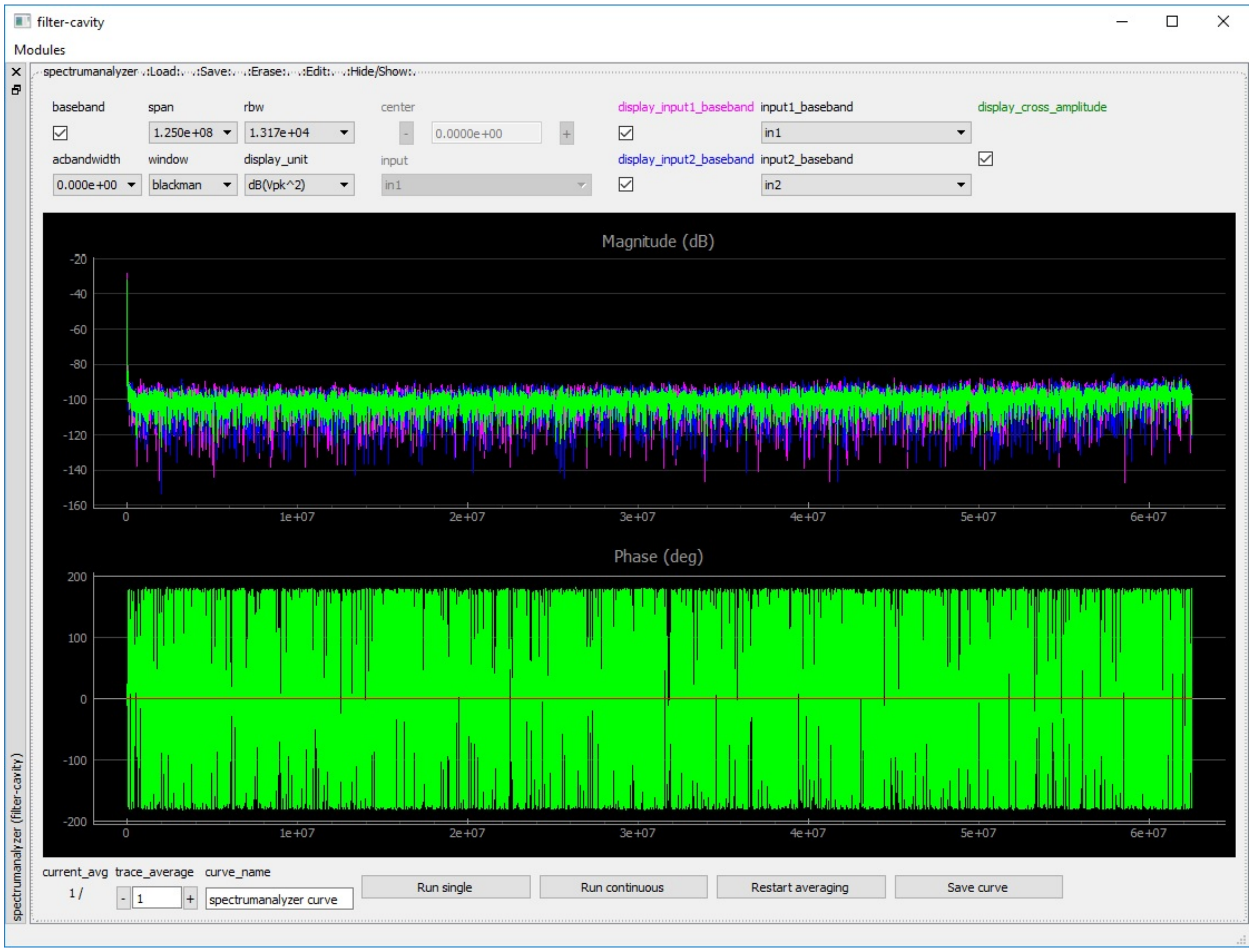
Note: If `output_direct='off'`, another module's input can be set to `networkanalyzer` to test its response to a frequency sweep.

- `amplitude` sets the amplitude of the sinusoidal excitation in Volts.
- `start_freq`/`stop_freq` define the frequency range over which a transfer function is recorded. Swapping the values of `start_freq` and `stop_freq` reverses the direction of the frequency sweep. Setting `stop_freq = start_freq` enables the "zero-span" mode, where the coherent response at a constant frequency is recorded as a function of time.
- `points` defines the number of frequency points in the recorded transfer function.

- `rbw` is the cutoff frequency of the low-pass filter after demodulation. Furthermore, the time τ spent to record each point is $\tau =$

Spectrum Analyzer Widget

The spectrum analyzer widget is represented in the image below.


filter-cavity
Modules
spectrumanalyzer .:Load:. .:Save:. .:Erase:. .:Edit:. .:Hide/Show:.
Magnitude (dB)
Phase (deg)
Run single
Run continuous
Restart averaging
Save curve
spectrumanalyzer (filter-cavity)

baseband: ✓
span: 1.250e+08
rbw: 1.317e+04
center: 0.0000e+00
display_input1_baseband: ✓
input1_baseband: in1
display_cross_amplitude: ✓
acbandwidth: 0.000e+00
window: blackman
display_unit: dB(Vpk^2)
input: in1
display_input2_baseband: ✓
input2_baseband: in2
current_avg: 1 /
trace_average: 1
curve_name: spectrumanalyzer curve

The spectrum-analyzer has 2 different working modes:

- iq-mode (not available in the current version): the data are first demodulated by an IQ-module around a center frequency and then Fourier Transformed. This mode allows to study a narrow span around the center frequency
- baseband: The Fourier transform is directly applied on the sampled data. Two inputs can be used in baseband mode, such that the complex cross-spectrum between the two inputs can be computed.

The following attributes can be manipulated by the module widget:

- acbandwidth (IQ mode only): The cut-off frequency of the high-pass filter for the iq-demodulator.
- span: frequency range of the analysis. In baseband mode, the span has to be divided by a factor 2.
- rbw: residual bandwidth of the analysis (span and bandwidth are linked and cannot be set independently)
- window: type of filtering window used (see scipy.signal.get_window for a list of windows available)
- diplay_unit: the unit in which the spectrum is represented (internally, all spectra are represented in V_pk^2)

Warning: Because the spectrum analyzer uses the data sampled by the scope to perform measurements, it is not possible to use both instruments simultaneaously. When the spectrum-analyzer is running, the scope-widget appears greyed-out to show that it is not available.

1.2.5 Iq Widget

The iq widget is represented in the image below. A schematic of the internal connection of the IQ-module can be shown or hidden with the arrow button.

The IQ-module is a very flexible Digital Signal Processing tool. Different values of the internal registers can be configured to perform various tasks:

Pound Drever Hall signal generation

The PDH locking technique is widely used to lock a laser beam to a high-finesse optical cavity. The principle is to generate a strong phase modulation of the laser beam (for instance, with an electro-optic modulator) at a frequency exceeding the cavity bandwidth and to detect the amplitude modulation in the beam reflected by the cavity. The amplitude modulation is caused by the abrupt phase response of the cavity affecting independently the sidebands from the carrier, and its sign with respect to the imposed modulation depends on cavity detuning. The high-speed digital signal processing of the redpitaya allows us to perform all the modulation/demodulation steps inside the FPGA, with modulations frequencies up to Nyquist frequecies (62.5 MHz). The correct IQ-module settings for PDH generation are (refer to the IQ signal schematic for explanations):

- gain=0. # no link from demodulation to modulation stage
- amplitude=1. # amplitude of the modulation
- frequency=50e6 # Modulation frequency
- phase=0 # adjust to compensate for cable length delays
- output_direct='out1' # output to optical phase modulator
- output_signal='quadrature'
- input='in1' # input from photodiode
- bandwidth=1e5 # trade-off between noise and error-signal bandwidth
- quadrature_factor=256 # adjust for saturation level
- acbandwidth=1e4 # to prevent internal saturation problems

Network analyzer

The network analyzer uses an IQ internally to accumulate the demodulated signal. The Network analyzer module automatically sets the following settings for the IQ module registers:

```
gain=0
quadrature_factor=0
output_direct=output_direct  # use output_signal to excite an internal signal
frequency=frequency # is value is scanned over time
bandwidth=rbw # bandwidth of the frequency analysis
input=input
acbandwidth=acbandwidth
```

Phase-frequency detector

The IQ-module can be used to perform phase/frequency comparison between the internal frequency reference and an input signal. This is done by connecting the output multiplexer to a frequency comparator (not represented in the schematic):

```
output_signal='pfd'
```

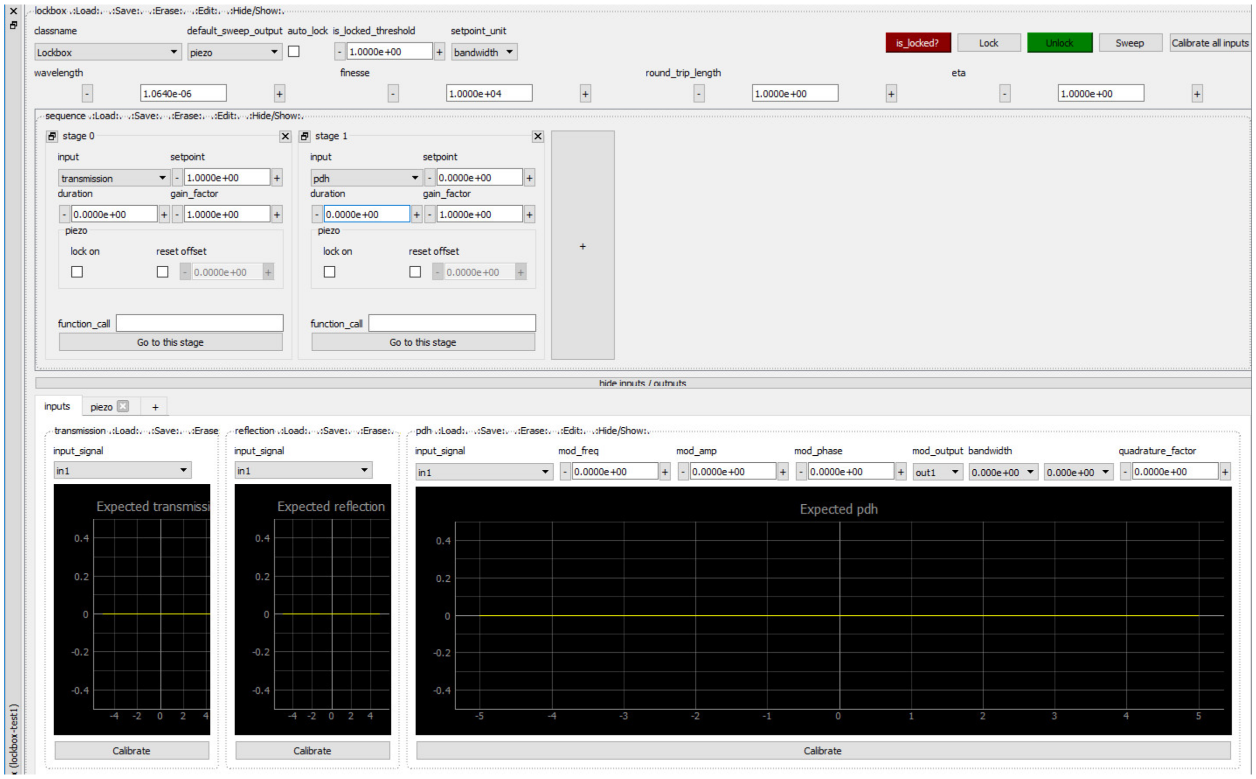
Tuanble bandpass filter

It is possible to realize very narrow bandpass filters by combining a demodulation and a remodulation stage. The correct settings are:

```
gain=1. # demod-> modulation gain
amplitude=0. # no internal excitation
frequency=1e6 # filter center frequency
bandwidth=100 # filter bandwidth (use a tuple for high-order filters)
quadrature_factor=0
output_signal='ouptut_direct' # if the signal needs to be used internally
phase=30 # eventually include some dephasing to the filter
```

1.2.6 Lockbox Widget

The lockbox widget is represented in the image below.



classname: Lockbox; default_sweep_output: piezo; auto_lock: ☐; is_locked_threshold: 1.0000e+00; setpoint_unit: bandwidth
wavelength: 1.0640e-06; finesse: 1.0000e+04; round_trip_length: 1.0000e+00; eta: 1.0000e+00

stage 0 — input: transmission; setpoint: 1.0000e+00; duration: 0.0000e+00; gain_factor: 1.0000e+00; piezo: lock on ☐, reset offset ☐ 0.0000e+00; function_call:

stage 1 — input: pdh; setpoint: 0.0000e+00; duration: 0.0000e+00; gain_factor: 1.0000e+00; piezo: lock on ☐, reset offset ☐ 0.0000e+00; function_call:

transmission — input_signal: in1
reflection — input_signal: in1
pdh — input_signal: in1; mod_freq: 0.0000e+00; mod_amp: 0.0000e+00; mod_phase: 0.0000e+00; mod_output: out1; bandwidth: 0.000e+00, 0.000e+00; quadrature_factor: 0.0000e+00

The Lockbox widget is used to produce a control signal to make a system's output follow a specified setpoint. The system has to behave linearly around the setpoint, which is the case for many systems. The key parts of the widget are:

- General controls: "classname" selects a particular Lockbox class from the ones defined in lockbox/models folder, and will determine the overall behaviour of the lockbox. "Calibrate all inputs" performs a sweep and uses acquired data to calibrate parameters relevant for the selected Lockbox class. Before attempting to lock, it's recommendable, and sometimes even mandatory, to press this button.

- Stages: In many situations, it might be desirable to start locking the system with a certain set of locking parameters, and once this has been achieved, switch to a different set with possibly a different signal. For example, when locking a Fabry–Pérot interferometer, the first stage might be locking on the side of a transmission fringe, and later transferring to locking on-resonance with Pound-Drever-Hall input signal. It is possible to have as many stages as necessary, and they will be executed sequentially.
- Stage settings: each stage has its own setpoint (whose units can be chosen in the general setting setpoint_unit) and a gain factor (a premultiplier to account for desired gain differences among different stages). In addition, based on the state of the “lock on” tri-state checkbox, a stage can enable (checkbox checked), disable (checkbox disabled) or leave unaffected (checkbox greyed out) the locking state when the stage is activated. The checkbox and field “reset offset” determine whether the lockbox should reset its output to a certain level when this stage is reached.
- Inputs and outputs: the PI parameters, together with limits, unit conversions and so on, are set in these tabs.

The lockbox module is completely customizable and allows to implement complex locking logic by inheriting the “Lockbox” class and adding the new class into lockbox/models. For example, below is an end-to-end locking scenario for a Fabry–Pérot interferometer that uses the included “FabryPerot” class:

You should start the lockbox module and first select the model class to FabryPerot. Then continue to configure first the outputs and inputs, filling in the information as good as possible. Critical fields are:

- Wavelength (in SI units)
- Outputs: configure the piezo output in the folding menu of inputs/outputs:
  - Select which output (out1 or out2) is the piezo connected to.
  - If it is the default_sweep_output, set the sweep parameters
  - Fill in the cutoff frequency if there is an analog low-pass filter behind the redpitaya, and start with a unity-gain frequency of 10 Hz.
  - Give an estimate on the displacement in meters per Volt or Hz per Volt (the latter being the obtained resonance frequency shift per volt at the Red Pitaya output), you ensure that the specified unit-gain is the one that Red Pitaya is able to set.
- Inputs:
  - Set transmission input to “in1” for example.
  - If PDH is used, set PDH input parameters to the same parameters as you have in the IQ configuration. Lockbox takes care of the setting, and is able to compute gains and slopes automatically
- Make sure to click “Sweep” and test whether a proper sweep is performed, and “Calibrate” to get the right numbers on the y-axis for the plotted input error signals
- At last, configure the locking sequence:
  - Each stage sleeps for “duration” in seconds after setting the desired gains.
  - The first stage should be used to reset all offsets to either +1 or -1 Volts, and wait for 10 ms or so (depending on analog lowpass filters)
  - Next stage is usually a “drift” stage, where you lock at a detuning of +/- 1 or +/- 2 bandwidths, possibly with a gain_factor below 1. make sure you enable the checkbox “lock enabled” for the piezo output here **by clicking twice on it** (it is actually a 3-state checkbox, see the information on the 1-click state when hovering over it). When you enable the locking sequence by clicking on lock, monitor the output voltage with a running scope, and make sure that this drift state actually makes the output voltage swing upwards. Otherwise, swap the sign of the setpoint / or the initial offset of the piezo output. Leave enough time for this stage to catch on to the side of a resonance.
  - Next stages can be adapted to switch to other error signals, modify setpoints and gains and so on.

1.3 API manual

This manual will guide you step-by-step through the programming interface of each PyRPL modules.

1.3.1 1 First steps

If the installation went well, you should now be able to load the package in python. If that works you can pass directly to the next section 'Connecting to the RedPitaya'.

```
from pyrpl import Pyrpl
```

Sometimes, python has problems finding the path to pyrpl. In that case you should add the pyrplockbox directory to your pythonpath environment variable (http://stackoverflow.com/questions/3402168/permanently-add-a-directory-to-pythonpath). If you do not know how to do that, just manually navigate the ipython console to the directory, for example:

```
cd c:\lneuhaus\github\pyrpl
```

Now retry to load the module. It should really work now.

```
from pyrpl import Pyrpl
```

Connecting to the RedPitaya

You should have a working SD card (any version of the SD card content is okay) in your RedPitaya (for instructions see http://redpitaya.com/quick-start/). The RedPitaya should be connected via ethernet to your computer. To set this up, there is plenty of instructions on the RedPitaya website (http://redpitaya.com/quick-start/). If you type the ip address of your module in a browser, you should be able to start the different apps from the manufacturer. The default address is http://192.168.1.100. If this works, we can load the python interface of pyrplockbox by specifying the RedPitaya's ip address.

```
HOSTNAME = "192.168.1.100"
```

```
from pyrpl import Pyrpl
p = Pyrpl(hostname=HOSTNAME)
```

If you see at least one '>' symbol, your computer has successfully connected to your RedPitaya via SSH. This means that your connection works. The message 'Server application started on port 2222' means that your computer has sucessfully installed and started a server application on your RedPitaya. Once you get 'Client started with success', your python session has successfully connected to that server and all things are in place to get started.

Basic communication with your RedPitaya

```
# Access the RedPitaya object in charge of communicating with the board
r = p.rp

#check the value of input1
print r.scope.voltage1
```

With the last command, you have successfully retrieved a value from an FPGA register. This operation takes about 300 ?s on my computer. So there is enough time to repeat the reading n times.

```
#see how the adc reading fluctuates over time
import time
from matplotlib import pyplot as plt
times, data = [],[]
t0 = time.time()
n = 3000
for i in range(n):
    times.append(time.time()-t0)
    data.append(r.scope.voltage_in1)
print("Rough time to read one FPGA register: ", (time.time()-t0)/n*1e6, "?s")
%matplotlib inline
f, axarr = plt.subplots(1,2, sharey=True)
axarr[0].plot(times, data, "+")
axarr[0].set_title("ADC voltage vs time")
axarr[1].hist(data, bins=10,normed=True, orientation="horizontal")
axarr[1].set_title("ADC voltage histogram")
```

You see that the input values are not exactly zero. This is normal with all RedPitayas as some offsets are hard to keep zero when the environment changes (temperature etc.). So we will have to compensate for the offsets with our software. Another thing is that you see quite a bit of scatter beetween the points - almost as much that you do not see that the datapoints are quantized. The conclusion here is that the input noise is typically not totally negligible. Therefore we will need to use every trick at hand to get optimal noise performance.

After reading from the RedPitaya, let's now try to write to the register controlling the first 8 yellow LED's on the board. The number written to the LED register is displayed on the LED array in binary representation. You should see some fast flashing of the yellow leds for a few seconds when you execute the next block.

```
#blink some leds for 5 seconds
from time import sleep
for i in range(1025):
    r.hk.led=i
    sleep(0.005)
```

```
# now feel free to play around a little to get familiar with binary representation by␣
↪looking at the leds.
from time import sleep
r.hk.led = 0b00000001
for i in range(10):
    r.hk.led = ~r.hk.led>>1
    sleep(0.2)
```

```
import random
for i in range(100):
    r.hk.led = random.randint(0,255)
    sleep(0.02)
```

1.3.2 2 RedPitaya (or Hardware) modules

Let's now look a bit closer at the class RedPitaya. Besides managing the communication with your board, it contains different modules that represent the different sections of the FPGA. You already encountered two of them in the example above: "hk" and "scope". Here is the full list of modules:

```
r.hk #"housekeeping" = LEDs and digital inputs/outputs
r.ams #"analog mixed signals" = auxiliary ADCs and DACs.
```

```
r.scope #oscilloscope interface

r.asg0 #"arbitrary signal generator" channel 0
r.asg1 #"arbitrary signal generator" channel 1

r.pid0 #first of three PID modules
r.pid1
r.pid2

r.iq0 #first of three I+Q quadrature demodulation/modulation modules
r.iq1
r.iq2

r.iir #"infinite impulse response" filter module that can realize complex transfer␣
↪functions
```

ASG and Scope module

Arbitrary Signal Generator

There are two Arbitrary Signal Generator modules: asg1 and asg2. For these modules, any waveform composed of 2^{14} programmable points is sent to the output with arbitrary frequency and start phase upon a trigger event.

Let's set up the ASG to output a sawtooth signal of amplitude 0.8 V (peak-to-peak 1.6 V) at 1 MHz on output 2:

```
asg.output_direct = 'out2'
asg.setup(waveform='halframp', frequency=20e4, amplitude=0.8, offset=0, trigger_
↪source='immediately')
```

Oscilloscope

The scope works similar to the ASG but in reverse: Two channels are available. A table of 2^{14} datapoints for each channel is filled with the time series of incoming data. Downloading a full trace takes about 10 ms over standard ethernet. The rate at which the memory is filled is the sampling rate (125 MHz) divided by the value of 'decimation'. The property 'average' decides whether each datapoint is a single sample or the average of all samples over the decimation interval.

```
s = r.scope # shortcut
print("Available decimation factors:", s.decimations)
print("Trigger sources:", s.trigger_sources)
print("Available inputs: ", s.inputs)
```

Let's have a look at a signal generated by asg1. Later we will use convenience functions to reduce the amount of code necessary to set up the scope:

```
asg = r.asg1
s = r.scope

# turn off asg so the scope has a chance to measure its "off-state" as well
asg.output_direct = "off"

# setup scope
s.input1 = 'asg1'
```

```
# pass asg signal through pid0 with a simple integrator - just for fun (detailed␣
↪explanations for pid will follow)
r.pid0.input = 'asg1'
r.pid0.ival = 0 # reset the integrator to zero
r.pid0.i = 1000 # unity gain frequency of 1000 hz
r.pid0.p = 1.0 # proportional gain of 1.0
r.pid0.inputfilter = [0,0,0,0] # leave input filter disabled for now

# show pid output on channel2
s.input2 = 'pid0'

# trig at zero volt crossing
s.threshold_ch1 = 0

# positive/negative slope is detected by waiting for input to
# sweep through hysteresis around the trigger threshold in
# the right direction
s.hysteresis_ch1 = 0.01

# trigger on the input signal positive slope
s.trigger_source = 'ch1_positive_edge'

# take data symetrically around the trigger event
s.trigger_delay = 0

# set decimation factor to 64 -> full scope trace is 8ns * 2^14 * decimation = 8.3 ms␣
↪long
s.decimation = 64

# launch a single (asynchronous) curve acquisition, the asynchronous
# acquisition means that the function returns immediately, eventhough the
# data-acquisition is still going on.
res = s.curve_async()

print("Before turning on asg:")
print("Curve ready:", s.curve_ready()) # trigger should still be armed

# turn on asg and leave enough time for the scope to record the data
asg.setup(frequency=1e3, amplitude=0.3, start_phase=90, waveform='halframp', trigger_
↪source='immediately')
sleep(0.010)

# check that the trigger has been disarmed
print("After turning on asg:")
print("Curve ready:", s.curve_ready())
print("Trigger event age [ms]:",8e-9*((
s.current_timestamp&0xFFFFFFFFFFFFFFFF) - s.trigger_timestamp)*1000)

# The function curve_async returns a *future* (or promise) of the curve. To
# access the actual curve, use result()
ch1, ch2 = res.result()

# plot the data
%matplotlib inline
plt.plot(s.times*1e3, ch1, s.times*1e3, ch2)
plt.xlabel("Time [ms]")
plt.ylabel("Voltage")
```

What do we see? The blue trace for channel 1 shows just the output signal of the asg. The time=0 corresponds to the trigger event. One can see that the trigger was not activated by the constant signal of 0 at the beginning, since it did not cross the hysteresis interval. One can also see a 'bug': After setting up the asg, it outputs the first value of its data table until its waveform output is triggered. For the halframp signal, as it is implemented in pyrpl, this is the maximally negative value. However, we passed the argument start_phase=90 to the asg.setup function, which shifts the first point by a quarter period. Can you guess what happens when we set start_phase=180? You should try it out!

In green, we see the same signal, filtered through the pid module. The nonzero proportional gain leads to instant jumps along with the asg signal. The integrator is responsible for the constant decrease rate at the beginning, and the low-pass that smoothens the asg waveform a little. One can also foresee that, if we are not paying attention, too large an integrator gain will quickly saturate the outputs.

```
# useful functions for scope diagnostics
print("Curve ready:", s.curve_ready())
print("Trigger source:",s.trigger_source)
print("Trigger threshold [V]:",s.threshold_ch1)
print("Averaging:",s.average)
print("Trigger delay [s]:",s.trigger_delay)
print("Trace duration [s]: ",s.duration)
print("Trigger hysteresis [V]", s.hysteresis_ch1)
print("Current scope time [cycles]:",hex(s.current_timestamp))
print("Trigger time [cycles]:",hex(s.trigger_timestamp))
print("Current voltage on channel 1 [V]:", r.scope.voltage_in1)
print("First point in data buffer 1 [V]:", s.ch1_firstpoint)
```

PID module

We have already seen some use of the pid module above. There are three PID modules available: pid0 to pid2.

```
print r.pid0.help()
```

Proportional and integral gain

```
#make shortcut
pid = r.pid0

#turn off by setting gains to zero
pid.p,pid.i = 0,0
print("P/I gain when turned off:", pid.i,pid.p)
```

```
# small nonzero numbers set gain to minimum value - avoids rounding off to zero gain
pid.p = 1e-100
pid.i = 1e-100
print("Minimum proportional gain: ", pid.p)
print("Minimum integral unity-gain frequency [Hz]: ", pid.i)
```

```
# saturation at maximum values
pid.p = 1e100
pid.i = 1e100
print("Maximum proportional gain: ", pid.p)
print("Maximum integral unity-gain frequency [Hz]: ", pid.i)
```

Control with the integral value register

```python
import numpy as np
#make shortcut
pid = r.pid0

# set input to asg1
pid.input = "asg1"

# set asg to constant 0.1 Volts
r.asg1.setup(waveform="dc", offset = 0.1)

# set scope ch1 to pid0
r.scope.input1 = 'pid0'

#turn off the gains for now
pid.p,pid.i = 0, 0

#set integral value to zero
pid.ival = 0

#prepare data recording
from time import time
times, ivals, outputs = [], [], []

# turn on integrator to whatever negative gain
pid.i = -10

# set integral value above the maximum positive voltage
pid.ival = 1.5

#take 1000 points - jitter of the ethernet delay will add a noise here but we dont␣
↪care
for n in range(1000):
    times.append(time())
    ivals.append(pid.ival)
    outputs.append(r.scope.voltage_in1)

#plot
import matplotlib.pyplot as plt
%matplotlib inline
times = np.array(times)-min(times)
plt.plot(times,ivals,times,outputs)
plt.xlabel("Time [s]")
plt.ylabel("Voltage")
```

Again, what do we see? We set up the pid module with a constant (positive) input from the ASG. We then turned on the integrator (with negative gain), which will inevitably lead to a slow drift of the output towards negative voltages (blue trace). We had set the integral value above the positive saturation voltage, such that it takes longer until it reaches the negative saturation voltage. The output of the pid module is bound to saturate at +- 1 Volts, which is clearly visible in the green trace. The value of the integral is internally represented by a 32 bit number, so it can practically take arbitrarily large values compared to the 14 bit output. You can set it within the range from +4 to -4V, for example if you want to exloit the delay, or even if you want to compensate it with proportional gain.

Input filters

The pid module has one more feature: A bank of 4 input filters in series. These filters can be either off (bandwidth=0), lowpass (bandwidth positive) or highpass (bandwidth negative). The way these filters were implemented demands that the filter bandwidths can only take values that scale as the powers of 2.

```
# off by default
r.pid0.inputfilter
```

```
# minimum cutoff frequency is 1.1 Hz, maximum 3.1 MHz (for now)
r.pid0.inputfilter = [1,1e10,-1,-1e10]
print(r.pid0.inputfilter)
```

```
# not setting a coefficient turns that filter off
r.pid0.inputfilter = [0,4,8]
print(r.pid0.inputfilter)
```

```
# setting without list also works
r.pid0.inputfilter = -2000
print(r.pid0.inputfilter)
```

```
# turn off again
r.pid0.inputfilter = []
print(r.pid0.inputfilter)
```

You should now go back to the Scope and ASG example above and play around with the setting of these filters to convince yourself that they do what they are supposed to.

IQ module

Demodulation of a signal means convolving it with a sine and cosine at the 'carrier frequency'. The two resulting signals are usually low-pass filtered and called 'quadrature I' and 'quadrature Q'. Based on this simple idea, the IQ module of pyrpl can implement several functionalities, depending on the particular setting of the various registers. In most cases, the configuration can be completely carried out through the setup function of the module.

Lock-in detection / PDH / synchronous detection

```
#reload to make sure settings are default ones
from pyrpl import Pyrpl
r = Pyrpl(hostname="192.168.1.100").rp

#shortcut
iq = r.iq0

# modulation/demodulation frequency 25 MHz
# two lowpass filters with 10 and 20 kHz bandwidth
# input signal is analog input 1
# input AC-coupled with cutoff frequency near 50 kHz
# modulation amplitude 0.1 V
# modulation goes to out1
# output_signal is the demodulated quadrature 1
# quadrature_1 is amplified by 10
iq.setup(frequency=25e6, bandwidth=[10e3,20e3], gain=0.0,
         phase=0, acbandwidth=50000, amplitude=0.5,
         input='in1', output_direct='out1',
         output_signal='quadrature', quadrature_factor=10)
```

After this setup, the demodulated quadrature is available as the output_signal of iq0, and can serve for example as the input of a PID module to stabilize the frequency of a laser to a reference cavity. The module was tested and is in daily use in our lab. Frequencies as low as 20 Hz and as high as 50 MHz have been used for this technique. At the present time, the functionality of a PDH-like detection as the one set up above cannot be conveniently tested internally. We plan to upgrade the IQ-module to VCO functionality in the near future, which will also enable testing the PDH functionality.

Network analyzer

When implementing complex functionality in the RedPitaya, the network analyzer module is by far the most useful tool for diagnostics. The network analyzer is able to probe the transfer function of any other module or external device by exciting the device with a sine of variable frequency and analyzing the resulting output from that device. This is done by demodulating the device output (=network analyzer input) with the same sine that was used for the excitation and a corresponding cosine, lowpass-filtering, and averaging the two quadratures for a well-defined number of cycles. From the two quadratures, one can extract the magnitude and phase shift of the device's transfer function at the probed frequencies. Let's illustrate the behaviour. For this example, you should connect output 1 to input 1 of your RedPitaya, such that we can compare the analog transfer function to a reference. Make sure you put a 50 Ohm terminator in parallel with input 1.

```
# shortcut for na
na = p.networkanalyzer
na.iq_name = 'iq1'

# setup network analyzer with the right parameters
na.setup(start=1e3,stop=62.5e6,points=1001,rbw=1000, avg=1,
amplitude=0.2,input='iq1',output_direct='off', acbandwidth=0)

#take transfer functions. first: iq1 -> iq1, second iq1->out1->(your cable)->adc1
iq1 = na.curve()
na.setup(input='in1', output_direct='out1')
in1 = na.curve()

# get x-axis for plotting
f = na.frequencies

#plot
from pyrpl.hardware_modules.iir.iir_theory import bodeplot
%matplotlib inline
bodeplot([(f, iq1, "iq1->iq1"), (f, in1, "iq1->out1->in1->iq1")], xlog=True)
```

If your cable is properly connected, you will see that both magnitudes are near 0 dB over most of the frequency range. Near the Nyquist frequency (62.5 MHz), one can see that the internal signal remains flat while the analog signal is strongly attenuated, as it should be to avoid aliasing. One can also see that the delay (phase lag) of the internal signal is much less than the one through the analog signal path.

Note: The Network Analyzer is implemented as a software module, distinct from the iq module. This is the reason why networkanalyzer is accessed directly at the Pyrpl-object level *p.networkanalyzer* and not at the redpitaya level *p.rp.networkanalyzer*. However, an iq module is reserved whenever the network analyzer is acquiring data.

If you have executed the last example (PDH detection) in this python session, iq0 should still send a modulation to out1, which is added to the signal of the network analyzer, and sampled by input1. In this case, you should see a little peak near the PDH modulation frequency, which was 25 MHz in the example above.

Lorentzian bandpass filter

The iq module can also be used as a bandpass filter with continuously tunable phase. Let's measure the transfer function of such a bandpass with the network analyzer:

```
# shortcut for na and bpf (bandpass filter)
na = p.networkanalyzer
bpf = p.rp.iq2

# setup bandpass
bpf.setup(frequency = 2.5e6, #center frequency
          bandwidth=1.e3, # the filter quality factor
          acbandwidth = 10e5, # ac filter to remove pot. input offsets
          phase=0, # nominal phase at center frequency (propagation phase lags not␣
↪accounted for)
          gain=2.0, # peak gain = +6 dB
          output_direct='off',
          output_signal='output_direct',
          input='iq1')

# setup the network analyzer
na.setup(start=1e5, stop=4e6, points=201, rbw=100, avg=3,
                         amplitude=0.2, input='iq2',output_direct='off')

# take transfer function
tf1 = na.curve()

# add a phase advance of 82.3 degrees and measure transfer function
bpf.phase = 82.3
tf2 = na.curve()

f = na.frequencies

#plot
from pyrpl.hardware_modules.iir.iir_theory import bodeplot
%matplotlib inline
bodeplot([(f, tf1, "phase = 0.0"), (f, tf2, "phase = %.1f"%bpf.phase)])
```

Note: To measure the transfer function of an internal module, we cannot

use the *output_direct* property of the network ananlyzer (only 'out1', 'out2' or 'off' are allowed). To circumvent the problem, we set the input of the module to be measured to the network analyzer's iq.

Frequency comparator module

To lock the frequency of a VCO (Voltage controlled oscillator) to a frequency reference defined by the RedPitaya, the IQ module contains the frequency comparator block. This is how you set it up. You have to feed the output of this module through a PID block to send it to the analog output. As you will see, if your feedback is not already enabled when you turn on the module, its integrator will rapidly saturate (-585 is the maximum value here, while a value of the order of 1e-3 indicates a reasonable frequency lock).

```
iq = p.rp.iq0

# turn off pfd module for settings
iq.pfd_on = False

# local oscillator frequency
iq.frequency = 33.7e6

# local oscillator phase
iq.phase = 0
iq.input = 'in1'
iq.output_direct = 'off'
iq.output_signal = 'pfd'

print("Before turning on:")
print("Frequency difference error integral", iq.pfd_integral)

print("After turning on:")
iq.pfd_on = True
for i in range(10):
    print("Frequency difference error integral", iq.pfd_integral)
```

IIR module

Sometimes it is interesting to realize even more complicated filters. This is the case, for example, when a piezo resonance limits the maximum gain of a feedback loop. For these situations, the IIR module can implement filters with 'Infinite Impulse Response' (https://en.wikipedia.org/wiki/Infinite_impulse_response). It is the your task to choose the filter to be implemented by specifying the complex values of the poles and zeros of the filter. In the current version of pyrpl, the IIR module can implement IIR filters with the following properties:

- strictly proper transfer function (number of poles > number of zeros)
- poles (zeros) either real or complex-conjugate pairs
- no three or more identical real poles (zeros)
- no two or more identical pairs of complex conjugate poles (zeros)
- pole and zero frequencies should be larger than :math:'

rac{f_ m{nyquist}}{1000}' (but you can optimize the nyquist frequency of your filter by tuning the 'loops' parameter) - the DC-gain of the filter must be 1.0. Despite the FPGA implemention being more flexible, we found this constraint rather practical. If you need different behavior, pass the IIR signal through a PID module and use its input filter and proportional gain. If you still need different behaviour, the file iir.py is a good starting point. - total filter order <= 16 (realizable with 8 parallel biquads) - a remaining bug limits the dynamic range to about 30 dB before internal saturation interferes with filter performance

Filters whose poles have a positive real part are unstable by design. Zeros with positive real part lead to non-minimum phase lag. Nevertheless, the IIR module will let you implement these filters.

In general the IIR module is still fragile in the sense that you should verify the correct implementation of each filter you design. Usually you can trust the simulated transfer function. It is nevertheless a good idea to use the internal network analyzer module to actually measure the IIR transfer function with an amplitude comparable to the signal you expect to go through the filter, as to verify that no saturation of internal filter signals limits its performance.

```
#reload to make sure settings are default ones
from pyrpl import Pyrpl
p = Pyrpl(hostname="192.168.1.100")
```

```
#shortcut
iir = p.rp.iir

#print docstring of the setup function
print(iir.setup.__doc__)
```

```
#prepare plot parameters
%matplotlib inline
import matplotlib
matplotlib.rcParams['figure.figsize'] = (10, 6)

#setup a complicated transfer function
zeros = [ 4e4j+300, +2e5j+1000, +2e6j+3000]
poles = [ 1e6, +5e4j+300, 1e5j+3000, 1e6j+30000]
iir.setup(zeros=zeros, poles=poles, loops=None, plot=True)
print("Filter sampling frequency: ", 125./iir.loops,"MHz")
```

If you try changing a few coefficients, you will see that your design filter is not always properly realized. The bottleneck here is the conversion from the analytical expression (poles and zeros) to the filter coefficients, not the FPGA performance. This conversion is (among other things) limited by floating point precision. We hope to provide a more robust algorithm in future versions. If you can obtain filter coefficients by another, preferrably analytical method, this might lead to better results than our generic algorithm.

Let's check if the filter is really working as it is supposed:

```
# first thing to check if the filter is not ok
print("IIR overflows before:", bool(iir.overflow))
na = p.networkanalyzer

# measure tf of iir filter
iir.input = na.iq
na.setup(iq_name='iq1', start=1e4, stop=3e6, points = 301, rbw=100, avg=1,
         amplitude=0.1, input='iir', output_direct='off', logscale=True)
tf = na.curve()

# first thing to check if the filter is not ok
print("IIR overflows after:", bool(iir.overflow))

# retrieve designed transfer function
designdata = iir.transfer_function(na.frequencies)


#plot with design data
%matplotlib inline
import matplotlib
matplotlib.rcParams['figure.figsize'] = (10, 6)
from pyrpl.hardware_modules.iir.iir_theory import bodeplot
bodeplot([(na.frequencies, designdata, "designed system"),
(na.frequencies, tf, "measured system")], xlog=True)
```

As you can see, the filter has trouble to realize large dynamic ranges. With the current standard design software, it takes some 'practice' to design transfer functions which are properly implemented by the code. While most zeros are properly realized by the filter, you see that the first two poles suffer from some kind of saturation. We are working on an automatic rescaling of the coefficients to allow for optimum dynamic range. From the overflow register printed above the plot, you can also see that the network analyzer scan caused an internal overflow in the filter. All these are signs that different parameters should be tried.

A straightforward way to impove filter performance is to adjust the DC-gain and compensate it later with the gain of a subsequent PID module. See for yourself what the parameter g=0.1 (instead of the default value g=1.0) does here:

```
#rescale the filter by 20 fold reduction of DC gain
iir.setup(zeros=zeros, poles=poles, g=0.1, loops=None, plot=False)

# first thing to check if the filter is not ok
print("IIR overflows before:", bool(iir.overflow))

# measure tf of iir filter
iir.input = na.iq
tf = na.curve()

# first thing to check if the filter is not ok
print("IIR overflows after:", bool(iir.overflow))

# retrieve designed transfer function
designdata = iir.transfer_function(na.frequencies)


#plot with design data
%matplotlib inline
import matplotlib
matplotlib.rcParams['figure.figsize'] = (10, 6)
from pyrpl.hardware_modules.iir.iir_theory import bodeplot
bodeplot([(na.frequencies, designdata, "designed system"),
(na.frequencies, tf, "measured system")], xlog=True)
```

You see that we have improved the second peak (and avoided internal overflows) at the cost of increased noise in other regions. Of course this noise can be reduced by increasing the NA averaging time. But maybe it will be detrimental to your application? After all, IIR filter design is far from trivial, but this tutorial should have given you enough information to get started and maybe to improve the way we have implemented the filter in pyrpl (e.g. by implementing automated filter coefficient scaling).

If you plan to play more with the filter, these are the remaining internal iir registers:

```
iir = p.rp.iir

# useful diagnostic functions
print("IIR on:", iir.on)
print("IIR bypassed:", iir.shortcut)
print("IIR copydata:", iir.copydata)
print("IIR loops:", iir.loops)
print("IIR overflows:", bin(iir.overflow))
print("Coefficients (6 per biquad):")
print(iir.coefficients)

# set the unity transfer function to the filter
iir._setup_unity()
```

1.3.3 3 Pyrpl (or Software) modules

Software modules are modules that don’t have an FPGA counterpart. They are directly accessible at the root pyrpl object (no need to go through the redpitaya object). We have already encountered a software module above. Remember how we accessed the network analyzer module:

```
HOSTNAME = "192.168.1.100"
from pyrpl import Pyrpl
p = Pyrpl(hostname=HOSTNAME)

# hardware modules are members of the redpitaya object
p.rp.iq0

# software modules are members of the root pyrpl object
p.networkanalyzer
```

Software modules usually perform higher-level tasks than hardware modules. Moreover, accessing a hardware module without care could be harmful to some acquisition already running on the redpitaya. For this reason, it is advisable to access hardware modules via module managers only.

Using Module Managers

Module managers are lightweight software modules that manage the access to hardware modules. For example, to use the scope:

```
HOSTNAME = "192.168.1.100"
from pyrpl import Pyrpl
p = Pyrpl(hostname=HOSTNAME)

# directly accessing the scope will not *reserve* it
scope = p.rp.scope
print(scope.owner)
scope.duration = 1.

# using the scope manager changes its ownership
with p.scopes.pop('username') as scope:
   print(scope.owner)
   scope.duration =0.01
   print(scope.duration)
# The scope is freed (and reset to its previous state) after the with
# construct
print(scope.owner)
print(scope.duration)
```

In case several identical modules are available on the FPGA, the first one (starting from the end of the list) is returned by the module manager:

```
HOSTNAME = "192.168.1.100"
from pyrpl import Pyrpl
p = Pyrpl(hostname=HOSTNAME)

# directly accessing the scope will not *reserve* it
pid2 = p.rp.pid2
pid2.owner = 'foo'

# Pid manager returns the first free pid module (in decreasing order)
with p.pids.pop('username') as pid:
   print("pid0's owner: ", p.rp.pid0.owner)
   print("pid1's owner: ", p.rp.pid1.owner)
   print("pid2's owner: ", p.rp.pid2.owner)
print("pid0's owner: ", p.rp.pid0.owner)
print("pid1's owner: ", p.rp.pid1.owner)
print("pid2's owner: ", p.rp.pid2.owner)
```

Spectrum Analyzer

The spectrum analyzer measures the magnitude of an input signal versus frequency. There are two working modes for the spectrum analyzer implemented in pyrpl:

- iq mode: the input signal is demodulated around the center_frequency of the analysis window (using iq2). The slowly varying quadratures are subsequently sent to the 2 channels of the scope. The complex IQ time trace is built from the sum I(t) + iQ(t). The spectrum is then evaluated by performing a Fourier transforom of the the complex iq signal.
- baseband mode: up to 2 channels are available in baseband mode. The channels are digitized by the scope and the real traces are directly Fourier transformed. Since both channels are acquired simultaneously, it is also possible to retrieve the cross spectrum between channel 1 and channel 2 (the relative phase of the fourier transform coefficients is meaningful)

At the moment, the iq mode is deactivated since we haven't yet implemented the sharp antialiasing filters required to avoid polluting the analysis windows from aliased noise originating from outside the Nyquist frequency of the scope acquisition. However, we are planning on implementing such a filter with the iir module in the near future.

In the following example, we are going to demonstrate how to measure a sinusoidal signal and a white noise originating from an asg

```
# let's use a module manager for the asg
with p.asgs.pop('user') as asg:
    # setup a sine at 100 kHz
    asg.setup(frequency=1e5, waveform='sin', trigger_source='immediately',␣
↪amplitude=1., offset=0)

    # setup the spectrumanalyzer in baseband mode
    p.spectrumanalyzer.setup(input1_baseband=asg, #note that input1_baseband!=input)
                             baseband=True, # only mod eavailable right now
                             span=1e6, # span of the analysis (/2 in iq mode)
                             window=blackman # filter window)

    # the return format is (spectrum for channel 1, spectrum for channel 2,
    # real part of cross spectrum, imaginary part of cross spectrum):
    ch1, ch2, cross_re, cross_im = p.spectrumanalyzer.curve()

# plot the spectrum
import matplotlib.pyplot as plt
%matplotlib inline
plt.plot(p.spectrumanalyzer.frequencies, ch1)
```

We notice that the spectrum is peaked around 100 kHz (The width of the peak is given by the residual bandwidth), and the height of the peak is 1.

The internal unit of the spectrum analyzer is V_pk^2, such that a 1 V sine results in a 1 Vpk^2 peak in the spectrum. To convert the spectrum in units of noise spectral density, a utility function is provided: data_to_unit()

```
# let's use a module manager for the asg
with p.asgs.pop('user') as asg:
    # setup a white noise of variance 0.1 V
    asg.setup(frequency=1e5, waveform='noise', trigger_source='immediately',␣
↪amplitude=0.1, offset=0)

    # setup the spectrumanalyzer in baseband mode and full span
    p.spectrumanalyzer.setup(input1_baseband=asg, baseband=True, span=125e6)

    # the return format is (spectrum for channel 1, spectrum for channel 2,
    # real part of cross spectrum, imaginary part of cross spectrum):
    ch1, ch2, cross_re, cross_im = p.spectrumanalyzer.curve()

# convert to Vrms^2/Hz
data = p.spectrumanalyzer.data_to_unit(ch1, 'Vrms^2/Hz', p.spectrumanalyzer.rbw)

# plot the spectrum
import matplotlib.pyplot as plt
%matplotlib inline
plt.plot(p.spectrumanalyzer.frequencies, data)

# integrate spectrum from 0 to nyquist frequency
df = p.spectrumanalyzer.frequencies[1] - p.spectrumanalyzer.frequencies[0]
print(sum(data)*df)
```

As expected, the integral of the noise spectrum over the whole frequency range gives the variance of the noise. To know more about spectrum analysis in Pyrpl, and in particular, how the filtering windows are normalized, please refer to the section How a spectrum is computed in PyRPL.

Lockbox

Coming soon

1.4 Basics of the PyRPL Architecture

This section presents the basic architecture of PyRPL. The main goal here is to quickly give a broad overview of PyRPL's internal logic without distracting the reader with too many technical details. For a more detailled description of the individual components described in this page, please, refer to the corresponding section *Notes for developers*.

1.4.1 Motivation

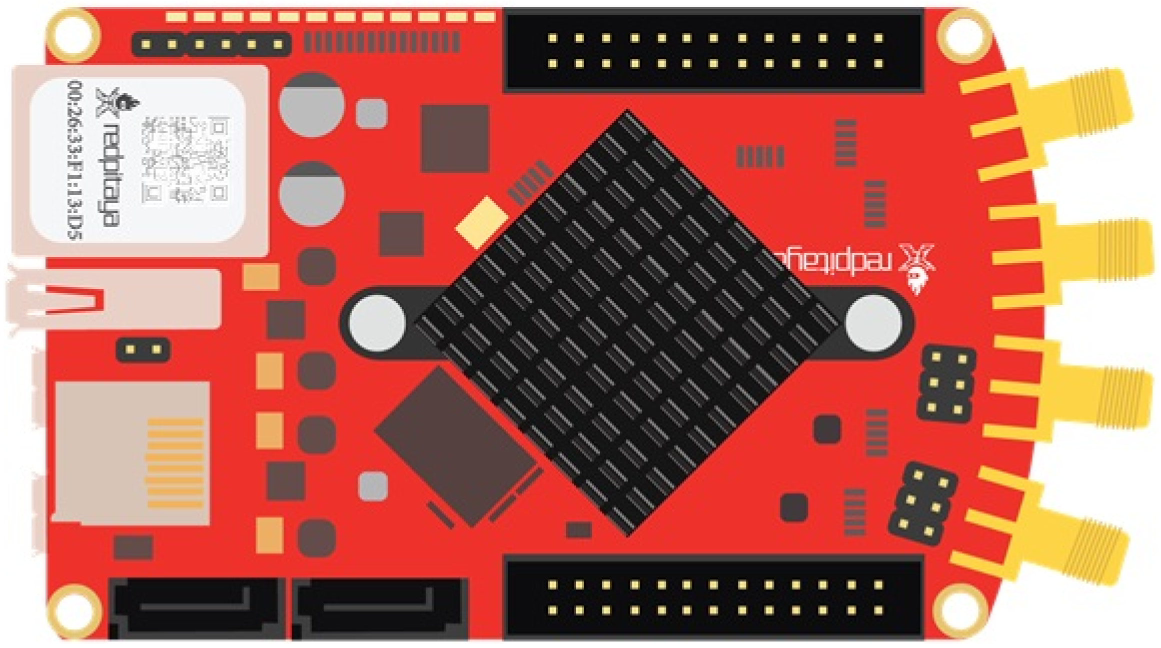
Available hardware boards featuring FPGAs, CPUs and analog in- and outputs makes it possible to use digital signal processing (DSP) to control quantum optics experiments. Running open-source software on this hardware has many advantages:

- Lab space: small size, less different devices
- Money: cheap hardware, free software
- Time: connect cables once, re-wire digitally automate experiments work from home
- Automated measurements incite to take more data-points perform experiments more reproducibly record additional, auxiliary data
- Functionality beyond analog electronics
- Modify or customize instrument functionality

However, learning all the subtelties of FPGA programming, compiling and debugging FPGA code can be extremely time consumming. Hence, PyRPL aims at providing a large panel of functionalities on a precompiled FPGA bitfile. These FPGA modules are highly customizable by changing register values without the need to recompile the FPGA code written in Hardware Description Language. High-level functionalities are implemented by a python package running remotely and controlling the FPGA registers.

1.4.2 Hardware Platform - Red Pitaya

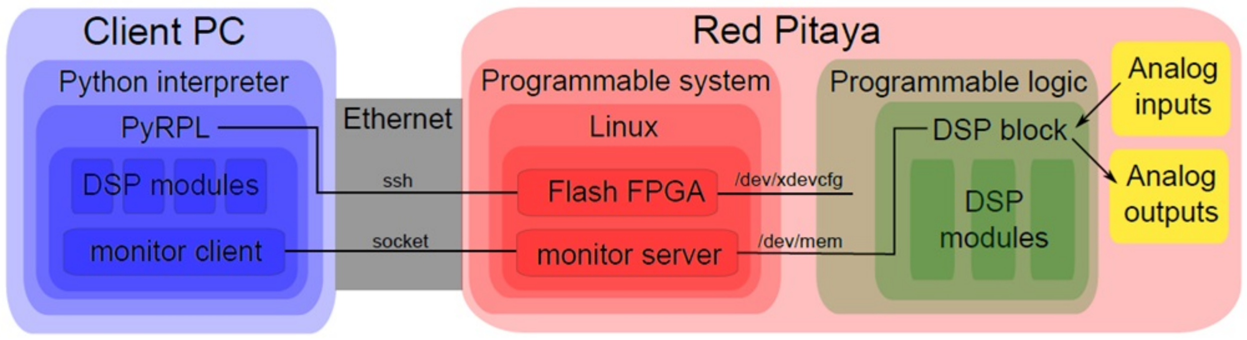
At the moment, Red Pitaya is the only hardware platform supported by PyRPL.



The RedPitaya board is an affordable FPGA + CPU board running a Linux operating system. The FPGA is running at a clock rate of 125 MSps and it is interfaced with 2 analog inputs and 2 analog outputs (14 bits, 125 MSps). The minimum input-output latency is of the order of 200 ns and the effective resolution is 12 bits for inputs and 13 bits for outputs. 4 slow analog inputs and outputs and 16 I/O ports are also available. Visit the The Red Pitaya homepage (http://www.redpitaya.com) for more details on the platform.

1.4.3 Software Infrastructure

The FPGA functionalities of PyRPL are organized in various DSP modules. These modules can be configured and arbitrarily connected together using a python package running on a client computer. This design offers a lot of flexibility in the design and control of various experimental setups without having to recompile the FPGA code each time a different fonctionality is needed. A fast ethernet interface maps all FPGA registers to Python variables. The read/write time is around 250 microseconds for a typical LAN connection. High-level functionalities are achieved by performing successive operations on the FPGA registers using the Python API. A Graphical User Interface is also provided to easily visualize and modify the different FPGA registers. We provide a description of the different software components below.



FPGA modules

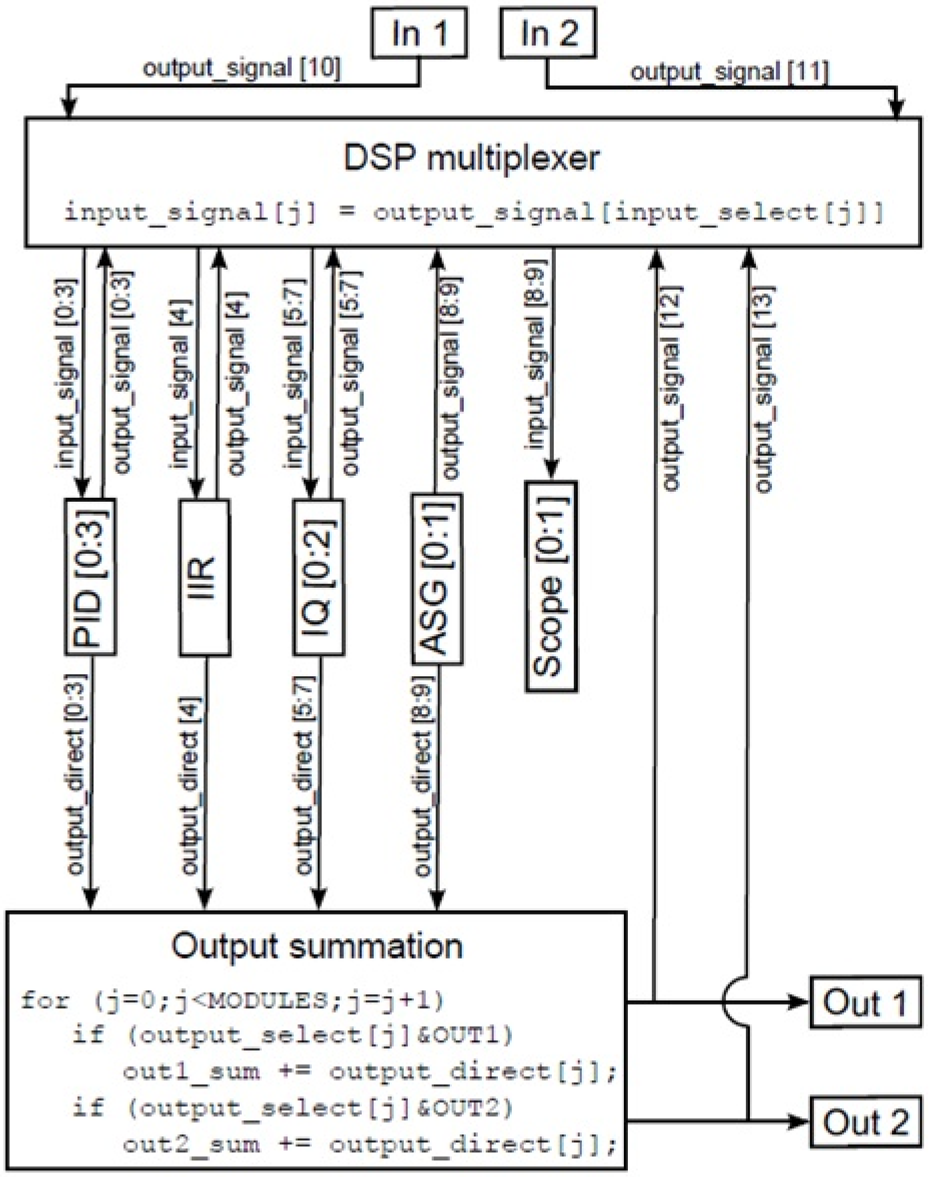
At the moment, the FPGA code provided with PyRPL implements various Digital Signal Processing modules:

Module name	# available	Short description
Scope	1	A 16384 points, 2 channels oscilloscope capable of monitoring internal or external signals
ASG	2	An arbitrary signal generator capable of generating various waveforms, and even gaussian white noise
IQ modulator/ demodulator	3	An internal frequency reference is used to digitally demodulate a given input signal. The frequency reference can be outputed to serve as a driving signal. The slowly varying quadratures can also be used to remodulate the 2 phase-shifted internal references, turning the module into a very narrow bandpass filter. See the page *Iq Widget* for more details
PID	3	Proportional/Integrator/Differential feedback modules (In the current version, the differential gain is disabled). The gain of each parameter can be set independently and each module is also equiped with a 4th order linear filter (applied before the PID correction)
IIR	1	An Infinite Impulse Response filter that can be used to realize real-time filters with comlex transfer-functions
Trigger	1	A module to detect a transition on an analog signal.
Sampler	1	A module to sample each external or external signal
Pwm	4	Modules to control the pulse width modulation pins of the redpitaya
Hk	1	House keeping module to monitor redpitaya constants and control the LED status

Modules can be connected to each other arbitrarily. For this purpose, the modules contain a generic register **input_select** (except for ASG). Connecting the **output_signal** of submodule **i** to the **input_signal** of submodule **j** is done by setting the register **input_select[j]** to **i**;

Similarly, a second, possibly different output is allowed for each module (except for scope and trigger): **output_direct**. This output is added to the analog output 1 and/or 2 depending on the value of the register **output_select**.

The routing of digital signals within the different FPGA modules is handled by a DSP multiplexer coded in VHDL in the file red_pitaya_dsp.v. An illustration of the DSP module's principle is provided below:



Monitor Server

The monitor server is a lightweight application written in C (the source code is in the file monitor_server.c) and running on the redpitaya OS to allow remote writing and monitoring of FPGA registers.

The program is launched on the redpitaya with (automatically done at startup):

```
./monitor-server PORT-NUMBER, where the default port number is 2222.
```

We allow for bidirectional data transfer. The client (python program) connects to the server, which in return accepts the connection. The client sends 8 bytes of data:

- Byte 1 is interpreted as a character: ‘r’ for read and ‘w’ for write, and ‘c’ for close. All other messages are ignored.
- Byte 2 is reserved.

- Bytes 3+4 are interpreted as unsigned int. This number n is the amount of 4-byte-units to be read or written. Maximum is 2^16.
- Bytes 5-8 are the start address to be written to.

If the command is read, the server will then send the requested 4*n bytes to the client. If the command is write, the server will wait for 4*n bytes of data from the server and write them to the designated FPGA address space. If the command is close, or if the connection is broken, the server program will terminate.

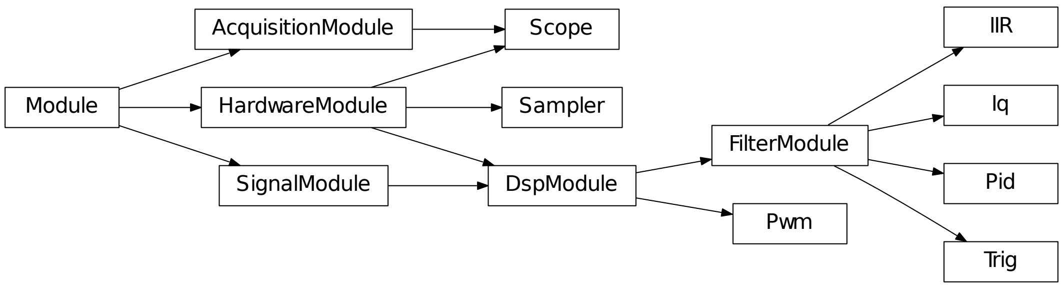
After this, the server will wait for the next command.

Python package PyRPL

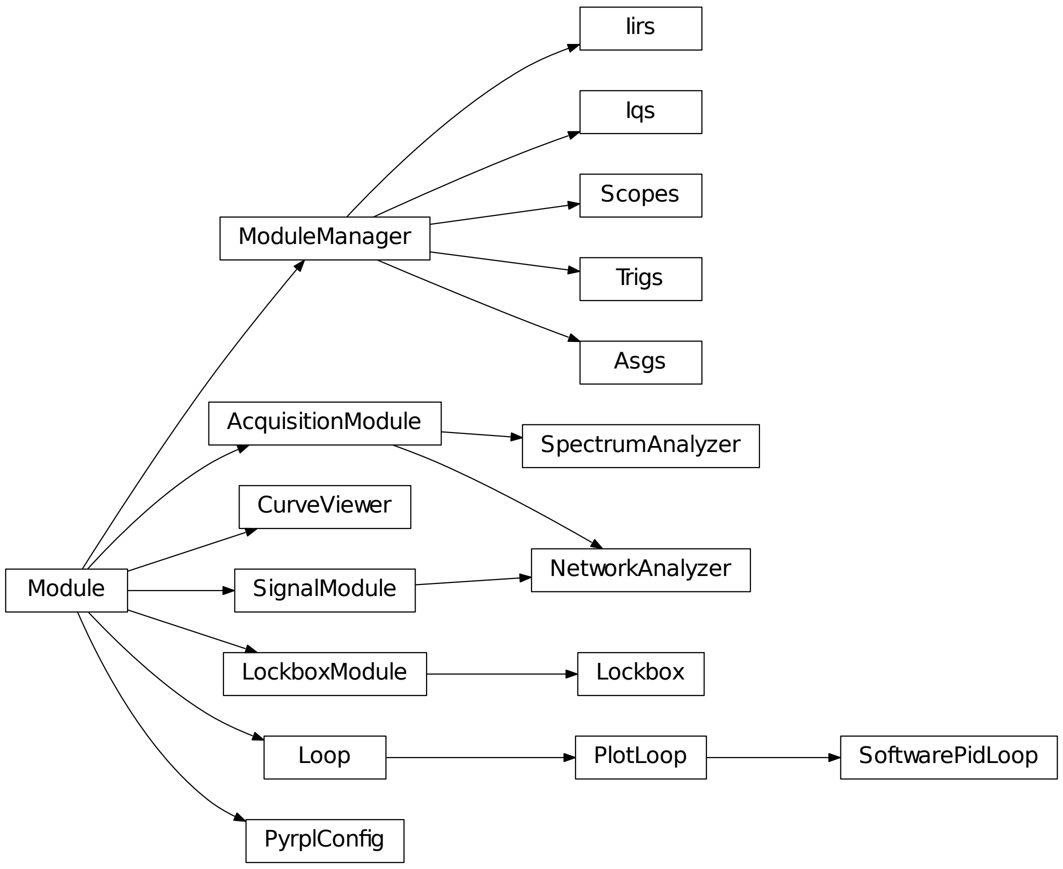
The python package PyRPL defines all the necessary tools to abstract the communication layer between the client-computer and the redpitaya. In this way, it is possible to manipulate FPGA registers transparently, as if they were simple attributes of local python objects. We give here a brief overview of the main python objects in PyRPL.

The Module class

Each FPGA module has a python counterpart: an instance of the class HardwareModule. The inheritance diagram of all HardwareModules is represented below:



For more complex functionalities, such as those involving the concurrent use of several FPGA modules, purely software modules can be created. Those modules only inherit from the base class Module and they don’t have an FPGA counterpart. Below, the inheritance diagram of all software modules:



In addition, to prevent a hardware resource from being used twice, HardwareModules should be accessed via the ModuleManagers which takes care of reserving them for a specific user or Module. For example:

```
 # import pyrpl library
 from pyrpl import Pyrpl

 # create an interface to the Red Pitaya
 pyrpl = Pyrpl()

# reserve the scope for user 'username'
 with pyrpl.scopes.pop('username') as mod:
      curve = mod.single() # acquire a curve
# The scope is freed for latter use at this point
```

The Proprety descriptors

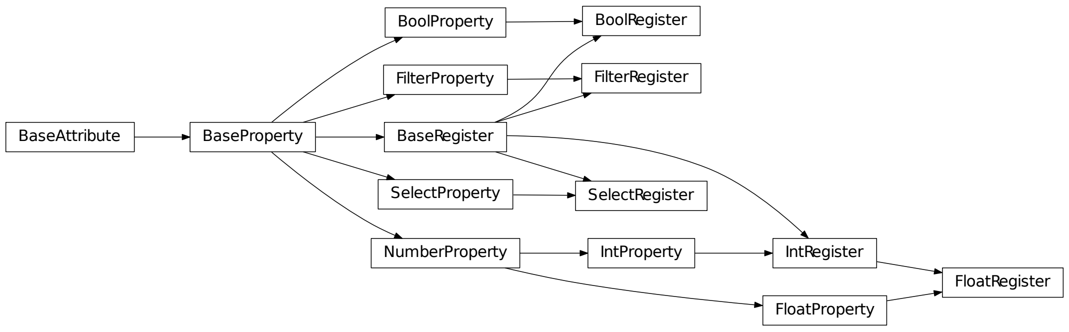
HardwareModules are essentially a list of FPGA registers that can be accessed transparently such as on the following example:

```
# import pyrpl library
import pyrpl

# create an interface to the Red Pitaya
r = pyrpl.Pyrpl().redpitaya

print(r.hk.led) # print the current led pattern

r.hk.led = 0b10101010  # change led pattern
```

Changing a register's value should trigger the following actions:

- communicating the new value to the monitor_server for the FPGA update via a TCP-IP socket.
- the new value should be saved on-disk to restore the system in the same state at the next startup.
- in case a Graphical User Interface is running, the displayed value should be updated.

To make sure all these actions are triggered by the simple python affectation, we use a descriptor pattern. The idea is to define setter and getter functions inside an auxilary "descriptor" class. The diagram below shows the inheritance diagram for the most common attribute descriptor types.



As for the distinction between software modules and hardware modules above, the properties that inherit from BaseRegister are directly mapping an FPGA register. On the other hand, software modules are using properties that are not in direct correspondance with an FPGA register. However, since they inherit from BaseAttribute, the load/save and GUI update mechanism is still implemented.

Module states

An important member of the Module class is the list **_setup_attributes**. This is a list of attribute names forming a subset of all module attributes. The value of the attributes in **_setup_attributes** constitutes the current state of the module. When the PyRPL instance has been created with a configuration file, the current state of each module is kept in-sync with the configuration file. This is particularly useful for GUI users who would like to keep the previous settings of all modules from one session to the next.

Warning: The config file is *not* kept in-sync with modules that are reserved by a user or another module. It is the responsibility of the user-script or owner module to keep track of the slave module state. Moreover, the slave-module is restored to the last current state whenever it becomes free.

The state of a module can be saved for latter use in a separate section of the config file. The following example shows the basic use of the load/save API:

```
# import pyrpl library
from pyrpl import Pyrpl

# create an interface to the Red Pitaya
scope = Pyrpl('new_config_file').redpitaya.scope

scope.duration = 1. # set curve duration to 1s
scope.save_state('slow_scan') # save state with label 'slow_scan'
scope.duration = 0.01 # set curve duration to 0.01s
scope.save_state('fast_scan') # save state with label 'fast_scan'
scope.load_state('slow_scan') # load state 'slow_scan'
scope.single() # acquire curve with a 1s duration
```

Automatic GUI creation

Designing Graphical User Interface can be a tedious work. However, since module attributes are defined in a uniform fashion across the project, most of the GUI creation can be handled automatically. Our GUI is based on the very popular and cross platform library PyQt in conjonction with the qtpy abstraction layer to make PyRPL compatible with PyQt4, PyQt5 and PySide APIs.

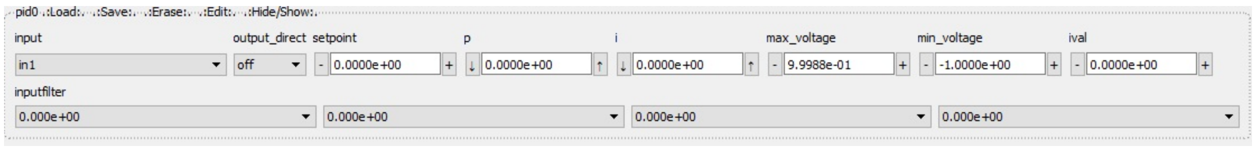
Each PyRPL module is represented by a widget in the Main PyRPL window. The list of attributes to display in the GUI is defined in the Module class by the class member **_gui_attributes**. When the module widget is created, sub-widgets are automatically created to manipulate the value of each attribute listed in **_gui_attributes**.

Example: definition of the Pid class

The following is extracted from pid.py

```
class Pid(FilterModule):
    # Type of widget to use for this Module class
    # should derive from ModuleWidget
    _widget_class = PidWidget

    # QObject used to communicate with the widget
    _signal_launcher = SignalLauncherPid

    # List of attributes forming the module state
    _setup_attributes = ["input", # defined in base class FilterModule
                         "output_direct", # defined in base class FilterModule
                         "setpoint",
                         "p",
                         "i",
                         #"d", # Not implemented in the current version of PyRPL
                         "inputfilter",
                         "max_voltage",
                         "min_voltage"]

    # list of attribtue to display in the GUI
    _gui_attributes = _setup_attributes + ["ival"]

    # Actions to perform immediately after a state has been loaded
    def _setup(self):
        """
        sets up the pid (just setting the attributes is OK).
        """
        pass

    # Below are the different attributes of a PID module (mostly registers)

    ival = IValAttribute(min=-4, max=4, increment= 8. / 2**16, doc="Current "
            "value of the integrator memory (i.e. pid output voltage offset)")

    setpoint = FloatRegister(0x104, bits=14, norm= 2 **13,
                             doc="pid setpoint [volts]")

    min_voltage = FloatRegister(0x124, bits=14, norm= 2 **13,
                                doc="minimum output signal [volts]")
    max_voltage = FloatRegister(0x128, bits=14, norm= 2 **13,
                                doc="maximum output signal [volts]")

    p = GainRegister(0x108, bits=_GAINBITS, norm= 2 **_PSR,
                      doc="pid proportional gain [1]")
    i = GainRegister(0x10C, bits=_GAINBITS, norm= 2 **_ISR * 2.0 * np.pi *
                                                  8e-9,
                      doc="pid integral unity-gain frequency [Hz]")
    (...)
```

The generated widget is represented below:



```
input          output_direct  setpoint      p             i             max_voltage   min_voltage    ival
in1            off            0.0000e+00    0.0000e+00    0.0000e+00    9.9988e-01    -1.0000e+00    0.0000e+00
inputfilter
0.000e+00      0.000e+00      0.000e+00      0.000e+00
```

1.5 Notes for developers

1.5.1 Contributing to PyRPL

Contributions to the PyRPL are welcome. To submit your changes for inclusion, please follow this procedure:

1. Fork this repository to your own github account using the fork button in the upper right corner on www.github.com/lneuhaus/pyrpl.
2. Clone (download) the fork to a local computer using git clone.
3. Modify anything you find useful, from the Python source code to the FPGA design. If you modify the FPGA, make sure to include the bitfile (see *Building the FPGA firmware*).
4. Modify the documentation in the docs-subfolder if necessary.
5. Use git add–>git commit–>git push to add changes to your fork.
6. Then submit a pull request by clicking the pull request button on your github repo.

Check the guide to git for more information.

1.5.2 Building the FPGA firmware

Compiling the FPGA code

- Install Vivado 2015.4 from the xilinx website, or directly use the windows web-installer or the linux web installer.
- Get a license as described at "*How to get the right license for Vivado 2015.4*".
- Navigate into `your_pyrpl_root_directory/pyrpl/fpga` and execute `make` (in linux) or `make.bat` (windows).
- The compilation of the FPGA code should take between 10 and 30 minutes, depending on your computer, and finish successfully.

How to get the right license for Vivado 2015.4

- After having created an account on xilinx.com, go to https://www.xilinx.com/member/forms/license-form.html.
- Fill out your name and address and click "next".
- select Certificate Based Licenses/Vivado Design Suite: HL WebPACK 2015 and Earlier License
- click Generate Node-locked license
- click Next
- get congratulated for having the new license file 'xilinx.lic' mailed to you. Download the license file and import it with Xilinx license manager.
- for later download: the license appears under the tab 'Managed licenses' with asterisks (*) in each field except for 'license type'='Node', 'created by'='your name', and the creation date.

The license problem is also discussed in issue #272 with screenshots of successful installations.

1.5.3 Unittests

1.5.4 Coding style guidelines

General guidelines

We follow the recommendations from PEP8.

Concerning **line length**, we have tried to stick to the 79 characters allowed by PEP8 so far. However, since this definitely restricts the readability of our code, we will accept 119 characters in the future (but please keep this at least consistent within the entire function or class). See the section on *Docstrings* below.

Other interesting policies that we should gladly accept are given here. - Django style guide

Naming conventions

- Capital letters for each new word in class names, such as *class TestMyClass(object):*.
- Lowercase letters with underscores for functions, such as *def test_my_class():*.
- Any methods or attributes of objects that might be visible in the user API (i.e. which are not themselves hidden) should serve an actual purpose, i.e. *pyrpl.lockbox.lock()*, *pyrpl.rp.iq.bandwidth* and so on.

- Any methods or attributes that are only used internally should be hidden from the API by preceeding the name with an underscore, such as *pyrpl.rp.scope._hidden_attribute* or *pyrpl.spectrum_analyzer._setup_something_for_the_measurement()*.
- Anything that is expected to return immediately and does not require an argument should be a property, asynchronous function calls or one that must pass arguments are implemented as methods.

Docstrings

Since we use sphinx for automatic documentation generation, we must ensure consistency of docstrings among all files in order to generate a nice documentation:

- follow PEP257 and docstrings in google-style — please read these documents **now**!!!
- keep the maximum width of docstring lines to 79 characters (i.e. 79 characters counted from the first non-whitespace character of the line)
- stay consistent with existing docstrings
- you should make use of the markup syntax allowed by sphinx
- we use docstrings in google-style, together with the sphinx-extension napoleon to format them as nice as the harder-to-read (in the source code) sphinx docstrings
- the guidelines are summarized in the napoleon/sphinx documentation example or in the example below:

```python
class SoundScope(Module):
    """
    An oscilloscope that converts measured data into sound.

    The oscilloscope works by acquiring the data from the redpitaya scope
    implemented in pyrpl/fpga/rtl/red_pitaya_scope_block.v, subsequent
    conversion through the commonly-known `Kolmogorov-Audio algorithm
    <http://www.wikipedia.org/Kolmogorov>`_ and finally outputting sound
    with the python package "PyAudio".

    Methods:
        play_sound: start sending data to speaker
        stop: stop the sound output

    Attributes:
        volume (float): Current volume
        channel (int): Current channel
        current_state (str): One of ``current_state_options``
        current_state_options (list of str): ['playing', 'stopped']
    """

    def play_sound(self, channel, lowpass=True, volume=0.5):
        """
        Start sending data of a scope channel to a speaker.

        Args:
            channel (int): Scope channel to use as data input
            lowpass (bool, optional): Turns on a 10 kHz lowpass
                filter before data sent to the output. Defaults to True.
            volume (float, optional): volume for sound output.
                Defaults to 0.5.

        Returns:
            bool: True for success, False otherwise.
        Raises:
            NotImplementedError: The given channel is not available.
            CannotHearAnythingException: Selected volume is too loud.
        """
```

1.5.5 Workflow to submit code changes

Preliminaries

While our project PyRPL is yet too small to make it necessary to define collaboration guidelines, we will just stick to the guidelines of the Collective Code Construction Contract (C4). In addition, if you would like to make a contribution to PyRPL, please do so by issuing a pull-request that we will merge. Your pull-request should pass unit-tests and be in PEP-8 style.

Use git to collaborate on code

As soon as you are able to, please use the git command line instead of programs such as gitHub, since their functionality is less accurate than the command line's.

1. Never work on the master branch. That way, you cannot compromise by mistake the functionality of the code that other people are using.
2. If you are developing a new function or the like, it is best to create your own branch. As soon as your development is fully functional and passes all unit tests, you can issue a pull request to master (or another branch if you prefer). Add a comment about the future of that branch, i.e. whether it can be deleted or whether you plan to work on the same branch to implement more changes. Even after the pull request has been merged into master, you may keep working on the branch.
3. It often occurs that two or more people end up working on the same branch. When you fetch the updates of other developers into your local (already altered) version of the branch with `git pull` you will frequenctly encounter conflicts. These are mostly easy to resolve. However, they will lead to an ugly history. This situation, along with the standard issue, is well described on stackoverflow. There are two ways to deal with this:

   (a) If you have only minor changes that can be summarized in one commit, you will be aware of this when you type:

   ```
   git fetch
   git status
   ```

   and you are shown that you are one or more commits behind the remote branch while only one or two local files are change. You should deal with this situation as follows:

   ```
   git stash
   git pull
   git stash pop
   ```

   This way, your local changes are saved onto the 'stash', then you update your local repository with the remote version that includes other developers' changes, and then you pop the stash onto that altered repository. The result is that only your own changes and the way you resolved the conflict will appear in the git history.

   (b) If you have a considerable amount of changes, we can accept the ugly merge commits. Just stay with `git pull` and put the keyword 'merge' into the commit message. To understand what is going on, read the copy-paste from the above link (copy-paste follows):

For example, two people are working on the same branch. The branch starts as:

```
...->C1
```

The first person finishes their work and pushes to the branch:

```
...->C1->C2
```

The second person finishes their work and wants to push, but can't because they need to update. The local repository for the second person looks like:

```
...->C1->C3
```

If the pull is set to merge, the second persons repository will look like:

```
...->C1->C3->M1
 \
   ->C2->
```

It will appear in the merge commit that the second person has committed all the changes from C2. Nevertheless, C2 remains in the git history and is not completely lost. This way, the merge commit accuratly represents the history of the branch. It just somehow spams you with information, so you should always use the former option 3.i when you can.

1.5.6 API specifications from the moment of their development

Requirements for an asynchronous interface compatible with python 3 asyncio

asynchronous programming in python 3

The idea behind async programming in python 3 is to avoid callbacks because they make the code difficult to read. Instead, they are replaced by "coroutines": a coroutine is a function that is declared with the `async def` keyword. Its execution can be stopped and restarted upon request at each `await` statement. This allows not to break loops into several chained timer/signal/slot mechanisms and makes the code much more readable (actually, very close to the corresponding synchronous function). Let's see that on an example:

```
%pylab qt # in a notebook, we need the qt event loop to run in the background
import asyncio
import scipy.fftpack
import quamash # quamash allows to use the asyncio syntax of python 3 with the Qt␣
↪event loop. Not sure how mainstream the library is...
from PyQt4 import QtCore, QtGui
import asyncio
loop = quamash.QEventLoop()
asyncio.set_event_loop(loop) # set the qt event loop as the loop to be used by asyncio


class Scope(object):
    async def run_single(self, avg):
        y_avg = zeros(100)
        for i in range(avg):
            await asyncio.sleep(1)
            y_avg+=rand(100)
        return y_avg
```

```
class SpecAn(object):
    scope = Scope()

    async def run_single(self, avg):
        y = zeros(100, dtype=complex)
        for i in range(avg):
            trace = await self.scope.run_single(1)
            y+= scipy.fftpack.fft(trace)
        return y

sa = SpecAn()

v = asyncio.ensure_future(sa.run_single(10)) # to send a coroutine to the asyncio␣
↪event loop, use ensure_future, and get a future...

v.result() # raise InvalidStateError until result is ready, then returns the averaged␣
↪curve
```

Wonderful !! As a comparison, the same code written with QTimers (in practice, the code execution is probably extremely similar)

```
%pylab qt
import asyncio
import scipy.fftpack
import quamash
from PyQt4 import QtCore, QtGui
APP = QtGui.QApplication.instance()
import asyncio
from promise import Promise
loop = quamash.QEventLoop()
asyncio.set_event_loop(loop)


class MyPromise(Promise):
    def get(self):
        while self.is_pending:
            APP.processEvents()
        return super(MyPromise, self).get()


class Scope(object):
    def __init__(self):
        self.timer = QtCore.QTimer()
        self.timer.setSingleShot(True)
        self.timer.setInterval(1000)
        self.timer.timeout.connect(self.check_for_curve)

    def run_single(self, avg):
        self.current_avg = 0
        self.avg = avg
        self.y_avg = zeros(100)
        self.p = MyPromise()
        self.timer.start()
        return self.p

    def check_for_curve(self):
        if self.current_avg<self.avg:
            self.y_avg += rand(100)
            self.current_avg += 1
            self.timer.start()
        else:
            self.p.fulfill(self.y_avg)


class SpecAn(object):
    scope = Scope()

    def __init__(self):
        timer = QtCore.QTimer()
        timer.setSingleShot(True)
        timer.setInterval(1000)

    def run_single(self, avg):
        self.avg = avg
        self.current_avg = 0
        self.y_avg = zeros(100, dtype=complex)
        p = self.scope.run_single(1)
        p.then(self.average_one_curve)
        self.p = MyPromise()
        return self.p

    def average_one_curve(self, trace):
        print('av')
        self.current_avg+=1
        self.y_avg+=scipy.fftpack.fft(trace)
        if self.current_avg>=self.avg:
            self.p.fulfill(self.y_avg)
        else:
            p = self.scope.run_single(1)
            p.then(self.average_one_curve)

sa = SpecAn()
```

... I dont blame you if you do not want to read the example above because its so lengthy! The loop variables have to be passed across functions via instance attributes, there's no way of clearly visualizing the execution flow. This is terrible to read and this pretty much what we have to live with in the asynchronous part of pyrpl if we want pyrpl to be compatible with python 2 (this is now more or less confined in AcquisitionManager now).

Can we make that compatible with python 2

The feature presented here is only compatible with python 3.5+ (by changing slightly the syntax, we could make it work on python 3.4). On the other hand, for python 2: the only backport is the library trollius, but it is not under development anymore, also, I am not sure if the syntax is exactly identical).

In other words, if we want to stay python 2 compatible, we cannot use the syntactic sugar of coroutines in the pyrpl code, we have to stick to the spaghetti-like callback mess. However, I would like to make the asynchronous parts of pyrpl fully compatible (from the user point of view) with the asyncio mechanism. This way, users of python 3 will be able to use functions such as run_single as coroutines and write beautiful code with it (eventhough the inside of the function looks like spaghetti code due to the constraint of being python 2 compatible).

To make specifications a bit clearer, let's see an example of what a python 3 user should be allowed to do:

```
async def my_coroutine(n):
    c1 = zeros(100)
    c2 = zeros(100)

    for i in range(n):
        print("launching f")
        f = asyncio.ensure_future(scope.run_single(1))
        print("launching g")
        g = asyncio.ensure_future(na.run_single(1))
        print("=======")
        c1+= await f
        c2+= await g
        print("f returned")
        print("g returned")

    return c1 + c2

p = asyncio.ensure_future(my_coroutine(3))
```

In this example, the user wants to ask *simultaneously* the na and the scope for a single curve, and when both curves are ready, do something with them and move to the next iteration. The following python 3 classes would easily do the trick:

```
%pylab qt
import asyncio
import scipy.fftpack
import quamash
from PyQt4 import QtCore, QtGui
import asyncio
loop = quamash.QEventLoop()
asyncio.set_event_loop(loop)


class Scope(object):
    async def run_single(self, avg):
        y_avg = zeros(100)
        for i in range(avg):
            await asyncio.sleep(1)
            y_avg+=rand(100)
        return y_avg


class Na(object):
    async def run_single(self, avg):
        y_avg = zeros(100)
        for i in range(avg):
            await asyncio.sleep(1)
            y_avg+=rand(100)
        return y_avg

scope = Scope()
na = Na()
```

What I would like is to find a way to make the same happen without writing any line of code in pyrpl that is not valid python 2.7... Actually, it seems the following code does the trick:

```
try:
    from asyncio import Future, ensure_future
except ImportError:
    from concurrent.futures import Future
```

```
class MyFuture(Future):
    def __init__(self):
        super(MyFuture, self).__init__()
        self.timer = QtCore.QTimer()
        self.timer.timeout.connect(lambda : self.set_result(rand(100)))
        self.timer.setSingleShot(True)
        self.timer.setInterval(1000)
        self.timer.start()

    def _exit_loop(self, x):
        self.loop.quit()

    def result(self):
        if not self.done():
            self.loop = QtCore.QEventLoop()
            self.add_done_callback(self._exit_loop)
            self.loop.exec_()
        return super(MyFuture, self).result()

class AsyncScope(object):
    def run_single(self, avg):
        self.f = MyFuture()
        return self.f

a = AsyncScope()
```

Asynchronous sleep function benchmarks

This is contained in *Asynchronous sleep function and benchmarks*.

Asynchronous sleep function and benchmarks

An asynchronous sleep function is highly desirable to let the GUI loop (at the moment, the Qt event loop) run while pyrpl is waiting for curves from the instruments.

The benchmark can be found in `timers.ipynb`. It was executed on python 3.5 on a windows 10 anaconda system.

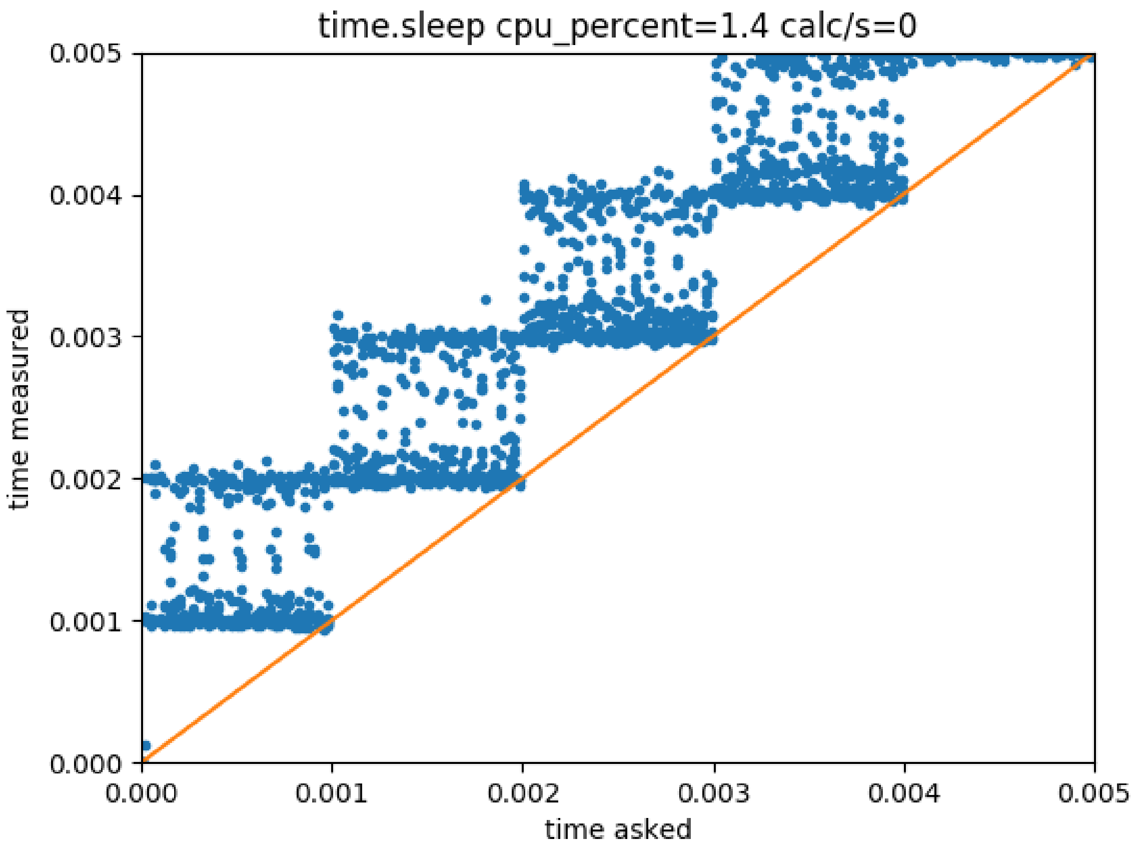
Methods compatible with python 2:

We first compare 4 different implementations of the sleep function that are all fully compatible between python 2 and python 3.

The normal time.sleep function (which is not asynchronous)

Calling time.sleep(delays) with delays ranging continuously from 0 to 5 ms gives the following distribution of measured delay vs requested delay:

As stated in the doc, sleep never returns before the requested delay, however, it will try its best not to return more than 1 ms too late. Moreover, we clearly have a problem because no qt events will be processed since the main thread is blocked by the current execution of time.sleep: for instance a timer's timeout will only be triggered once the sleep function has returned, this is what's causing freezing of the GUI when executing code in the jupyter console.

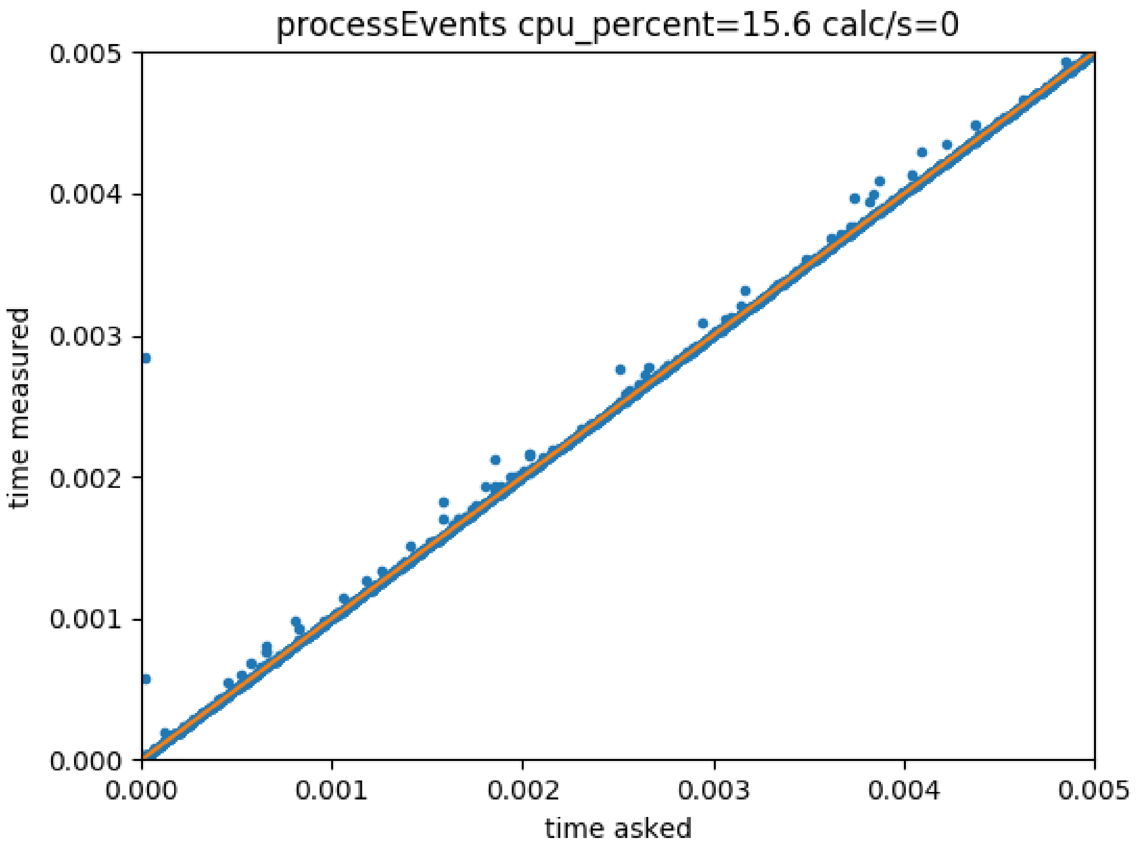

time.sleep cpu_percent=1.4 calc/s=0
time measured
time asked
0.000
0.001
0.002
0.003
0.004
0.005

Constantly calling APP.processEvents()

The first work around, is to manually call processEvents() regularly to make sure events are processed while our process is sleeping.

```python
from timeit import default_timer

def sleep_pe(delay):
    time0 = default_timer()
    while(default_timer()<time0+delay):
        APP.processEvents()
```

first comment: we need to use timit.default_timer because time.time has also a precision limited to the closest millisecond.



We get, as expected, an almost perfect correlation between requested delays and obtained delays. Some outliers probably result from the OS interrupting the current process execution, or even other events from the GUI loop being executed just before the requested time.

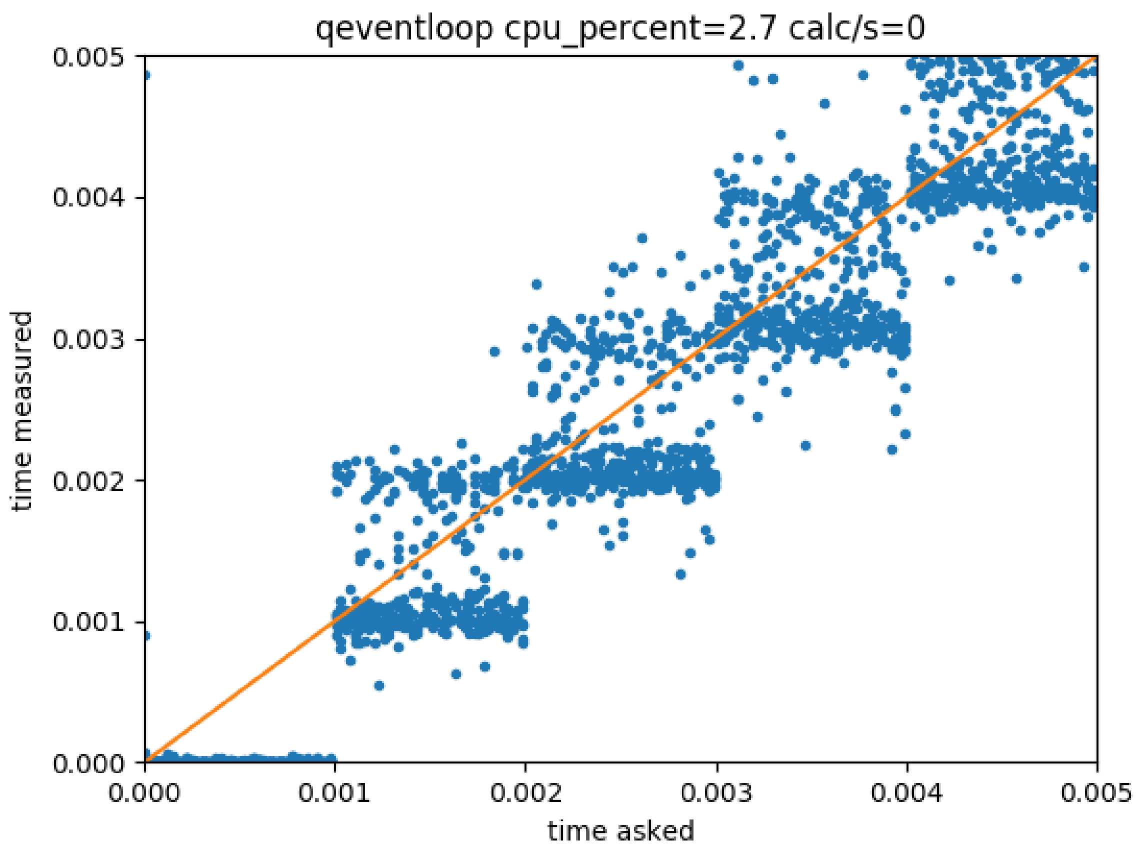
We also see that the CPU load is quite high, even though we don't do anything but waiting for events. This is due to the loop constantly checking for the current time and comparing it to the requested delay.

Running the QEventLoop locally

A better solution, as advertised here, is to run a new version of the QEventLoop locally:

```
def sleep_loop(delay):
    loop = QtCore.QEventLoop()
    timer = QtCore.QTimer()
    timer.setInterval(delay*1000)
    timer.setSingleShot(True)
    timer.timeout.connect(loop.quit)
    timer.start()
    loop.exec() # la loop prend en charge elle-même l'évenement du timer qui va la␣
↪faire mourir après delay.
```

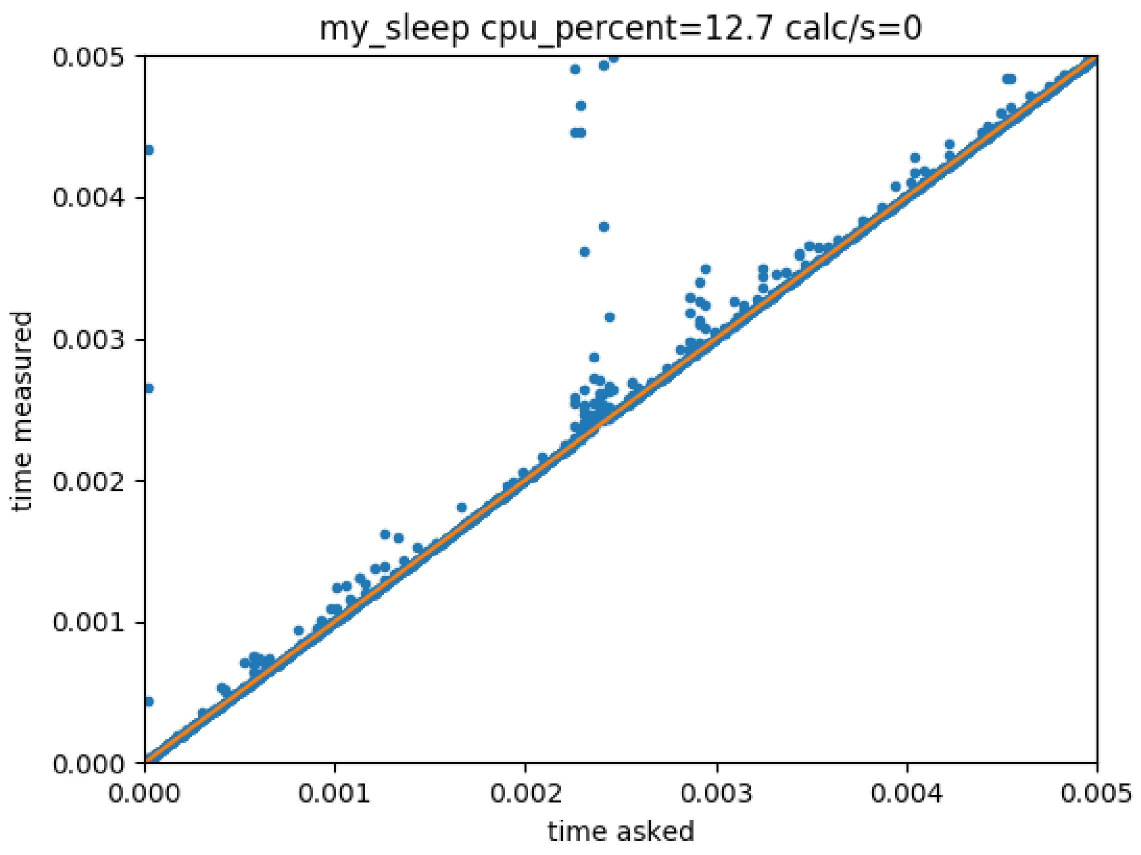
The subtlety here is that the loop.exec() function is blocking, and usully would never return. To force it to return after some time delay, we simply instanciate a QTimer and connect its timeout signal to the quit function of the loop. The timer's event is actually handled by the loop itself. We then get a much smaller CPU load, however, we go back to the situation where the intervals are only precise at the nearest millisecond.



The hybrid approach

A compromise is to use a QTimer that will stop 1 ms earlier, and then manually call processEvents for the remaining time. We get at the same time a low CPU load (as long as delay >> 1 ms, which is not completely verified here), and a precise timing.

```
def my_sleep(delay):
    tic = default_timer()
    if delay>1e-3:
        sleep_loop(delay - 1e-3)
    while(default_timer()<tic+delay):
        APP.processEvents()
```



Benchmark in the presence of other events

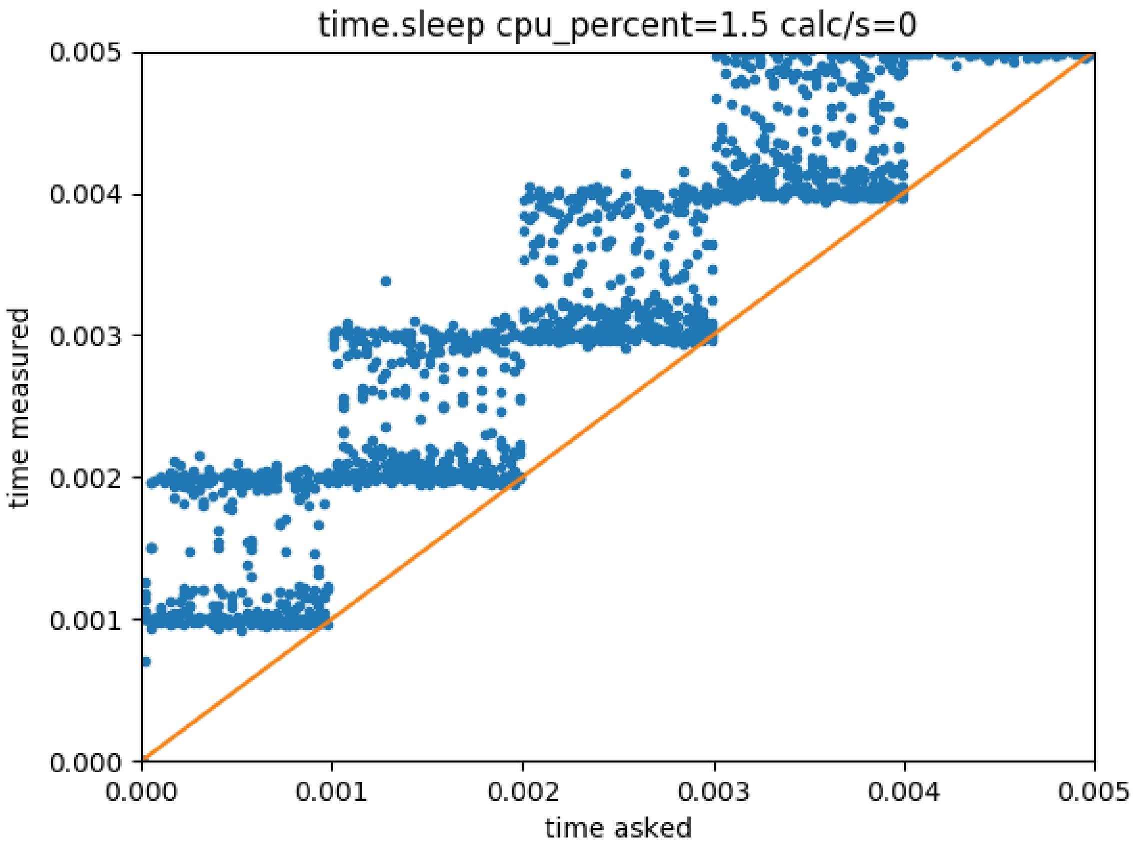
To simulate the fact that in real life, other events have to be treated while the loop is running (for instance, user interactions with the GUI, or another instrument running an asynchronous measurement loop), we run in parallel the following timer:

```
from PyQt4 import QtCore, QtGui
n_calc = [0]
def calculate():
    sin(rand(1000))
    n_calc[0]+=1
    timer.start()

timer = QtCore.QTimer()
timer.setInterval(0)
timer.setSingleShot(True)
timer.timeout.connect(calculate)
```

By looking at how fast `n_calc[0]` gets incremented, we can measure how blocking our sleep-function is for other events. We get the following outcomes (last number "calc/s" in the figure title):

time.sleep

As expected, time.sleep prevents any event from being processed

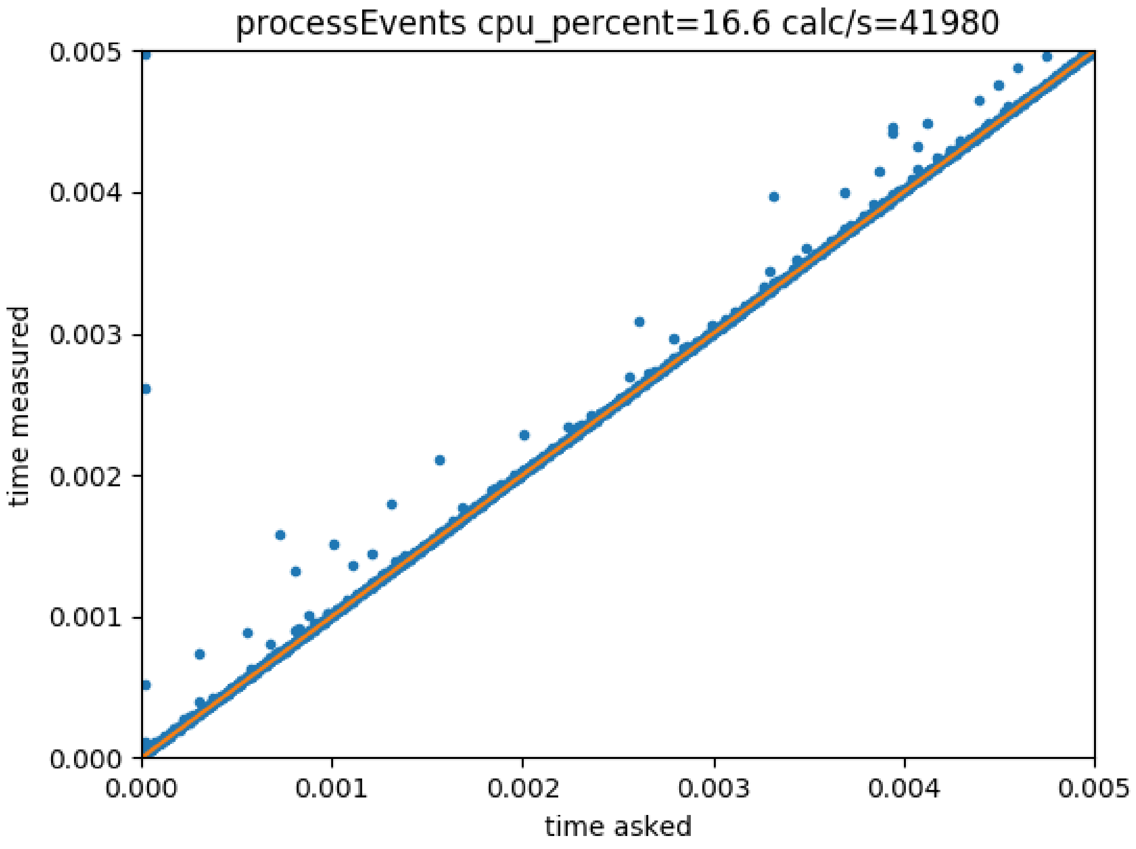


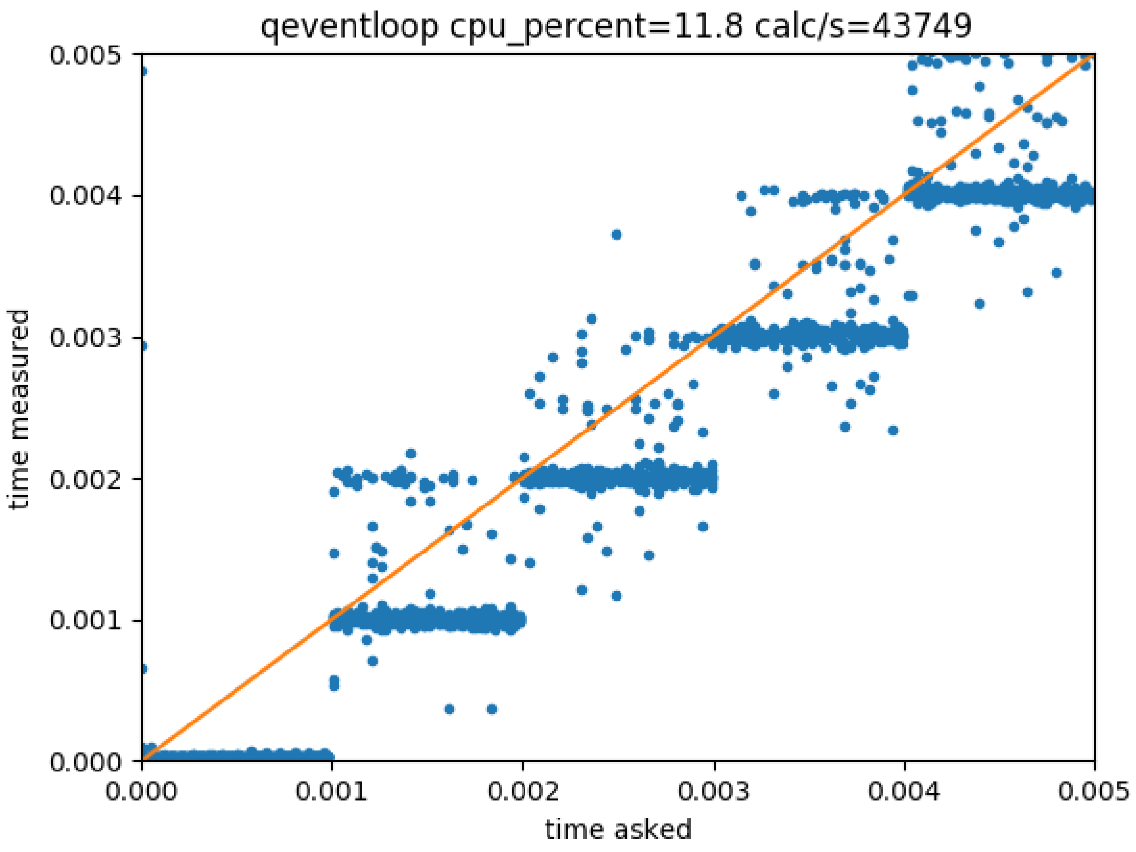
calling processEvents

40 000 events/seconds.

running the eventLoop locally

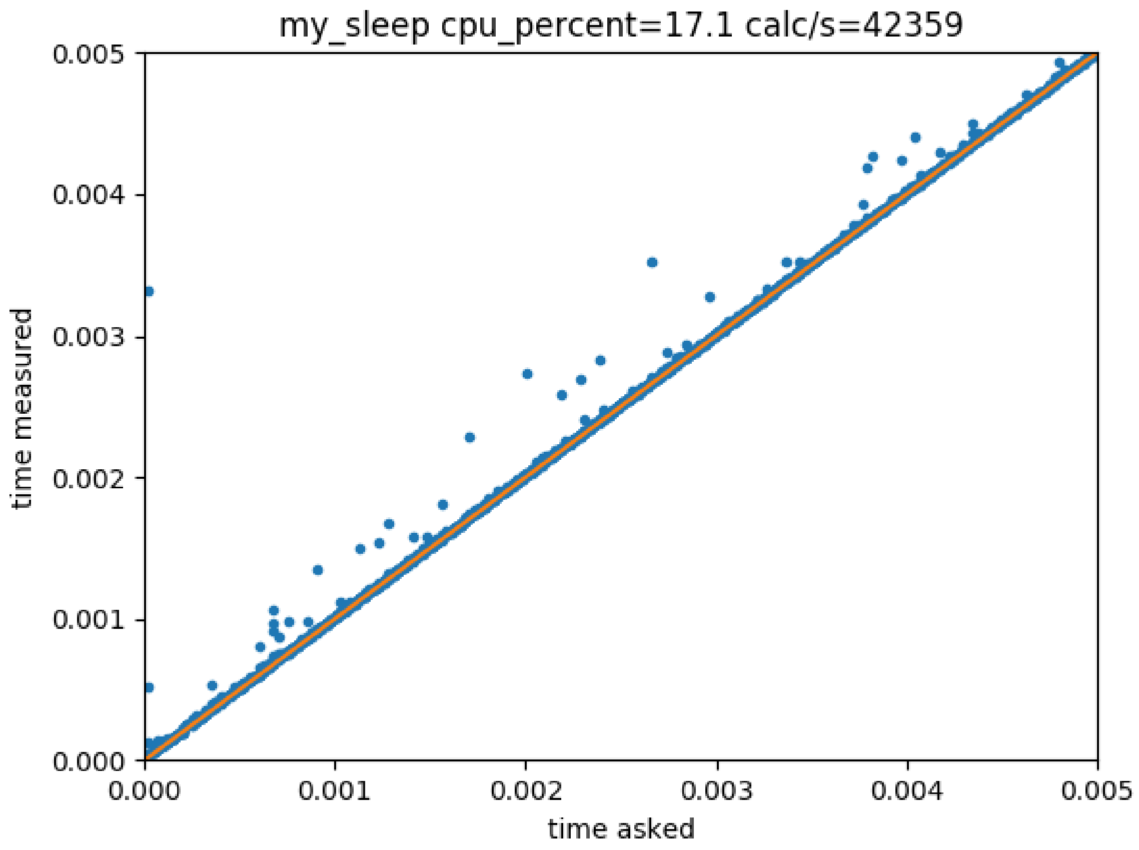
That's approximately identical


processEvents cpu_percent=16.6 calc/s=41980
time measured
time asked
0.000
0.001
0.002
0.003
0.004
0.005


qeventloop cpu_percent=11.8 calc/s=43749
time measured
time asked
0.000
0.001
0.002
0.003
0.004
0.005

our custom function

Still more or less identical (but remember that the big advantage compared to the previous version is that in the absence of external events, the CPU load is close to 0).



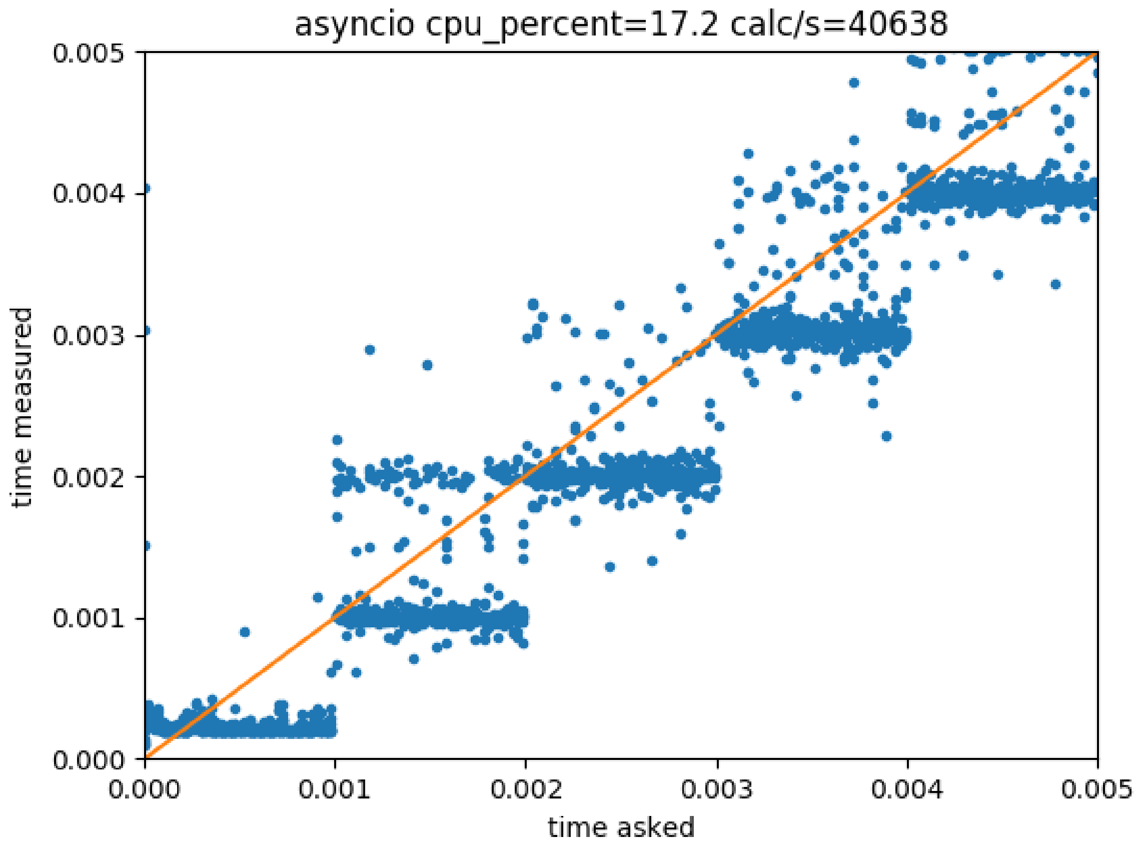
Async programming in python3(.5):

A description of async programming in python 3.5 is given in “*Requirements for an asynchronous interface compatible with python 3 asyncio*”. To summarize, it is possible to use the Qt event loop as a backend for the beautiful syntax of coroutines in python 3 using quamash. Of course, because the quamash library is just a wrapper translating the new python asynchronous syntax into QTimers, there is no magic on the precision/efficiency side: for instance, the basic coroutine `asyncio.sleep` gives a result similar to “Running a local QEventLoop”:

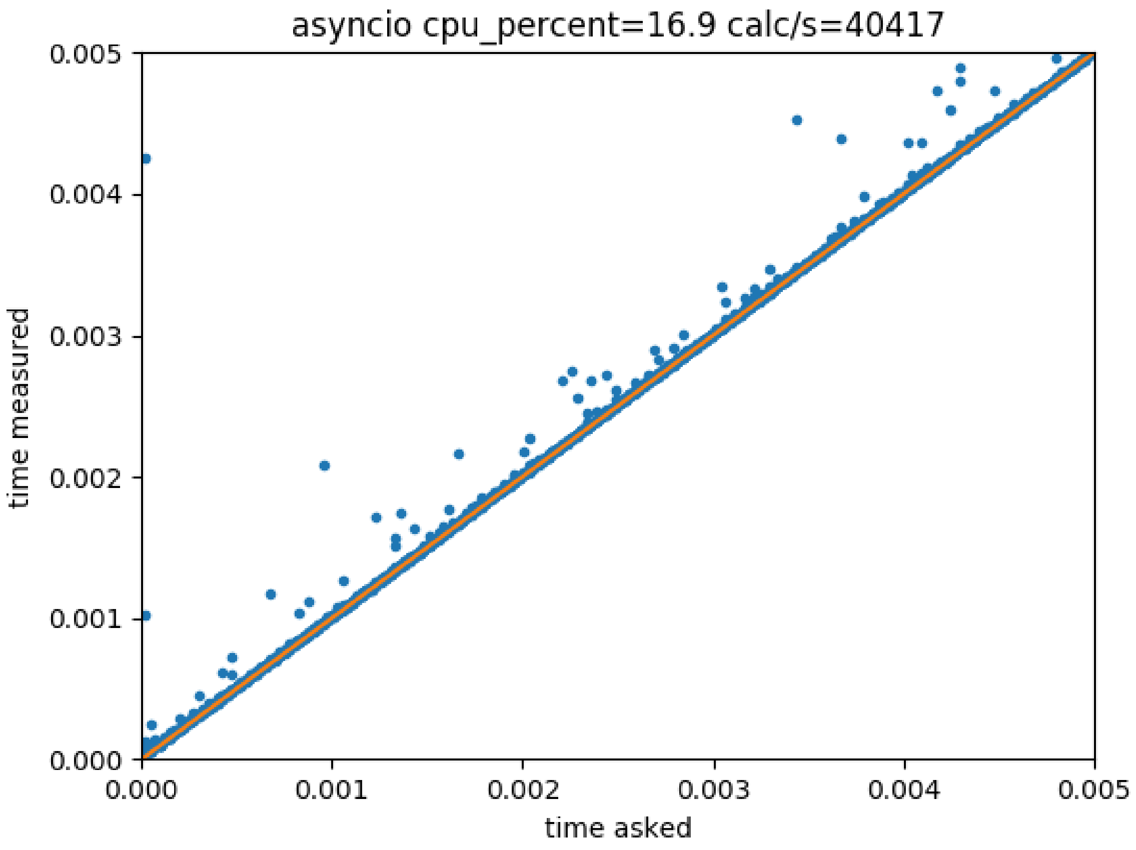
```
async def sleep_coroutine(delay):
    await asyncio.sleep(delay)
```

But, obviously, we can play the same trick as before to make a precise enough coroutine:

```
async def sleep_coroutine(delay):
    tic = default_timer()
    if delay>0.001:
        await asyncio.sleep(delay - 0.001)
```


asyncio cpu_percent=17.2 calc/s=40638
time measured
time asked
0.000
0.001
0.002
0.003
0.004
0.005

```
    while default_timer() < tic + delay:
        APP.processEvents()
```



How a spectrum is computed in PyRPL

Inspiration comes from Oppenheim & Schaefer 1975 and from Agilent

The spectrum analyzer in Pyrpl estimates the spectrum of internal or external signals by performing Fast-Fourier Transforms of traces recorded by the scope. Since in the current version of Pyrpl, the stream of data from the scope is made of discontiguous segments of 2^14 samples, we are currently using the Bartlett method, which consists in the following steps:

1. Each segment is multiplied by a symmetric window function of the same size.
2. The DFT of individual segments is performed. The segment is padded before the FFT by a number of 0s to provide more points in the estimated spectrum than in the original time segment.
3. The square modulus of the resulting periodograms are averaged to give the estimate of the spectrum, with the same size as the initial time-segments.

A variant of this method is the Welch method, in which the segments are allowed to be overlapping with each other. The advantage is that when a narrow windowing function (ie a large number of "points-per-bandwidth" in the frequency domain) is used, the points far from the center of the time-segments have basically no weight in the result. With overlapping segments, it is basically possible to move the meaningful part of the window over all the available data.

This is the basic principle of real-time spectrum analyzers. This cannot be implemented “as is” since the longest adjacent time-traces at our disposal is 2^14 sample long.

However, a possible improvement, which would not require any changes of the underlying FPGA code would be to apply the welch method with subsegments smaller than the initial scope traces: for instance we would extract 2^13 points subsegments, and we could shift the subsegment by up to 2^13 points. With such a method, even with an infinitely narrow windowing function, we would only “loose” half of the acquired data. This could be immediately implemented with the Welch method implemented in scipy.

In the following, we discuss the normalization of windowing functions, and then, the basic principle of operation of the two modes “iq” and “baseband”.

Definitions

name	definition
Original time series	x[k], 0<=k<N
Fourier Transform	X[r] = sum_k x[k] exp(-2 i pi r k/N)
Inverse Fourier Transform (equivalently)	x[k] = 1/N sum_r X[r] exp(2 i pi k r/N)
Time window	w[k]
Fourier transformed time window	W[r]
Singly averaged spectrum (in V_pk)	Y[r]=sum_k x[k] w[k] exp(-2 i pi r k/N)
Singly averaged spectrum (in Vrms^2/Hz)	Z(r) = |Y(r)|^2/ (2 rbw)

We can show that the Fourier transform of the product is the convolution of the Fourier Transforms, such that:

Y[r] = 1/N sum_r’ X[r’] W[r-r’] (1)

To make sure the windowing function is well normalized, and to define the noise equivalent bandwidth of a given windowing function, we will study the 2 limiting cases where the initial time series is either a sinusoid or a gaussian distributed white noise.

Sinusoidal input

To simplify the calculations, we assume the period of the sinusoid is a multiple of the sampling rate:

x[k] = cos[2 pi m k/N]

= 1/2 (exp[i 2 pi m k/N] + exp[-2 pi i (N - m) k/N])

We obtain the Fourier transform:

X[r] = N/2 (delta[r-m] + delta[r-(N-m)]).

We deduce using (1), that the estimated spectrum is:

Y[r] = 1/2 (W[r - m] + W[r - (N-m)])

With the discrete fourier transform convention used here, we need to pay attention that the DC-component is for r=0, and the ?negative frequencies? are actually located in the second half of the interval [N/2, N]. If we take the single sided convention where the negative frequency side is simply ignored, the correct normalization in terms of V_pk (for which the maximum of the spectrum corresponds to the amplitude of the sinusoid) is the one for where max(W[r]) = 2.

Moreover, a reasonable windowing function will only have non-zero Fourier components on the few bins around DC, such that if we measure a pure sinusoid with a frequency far from 0, there wont be any significant overlap between the two terms, and we will measure 2 distinct peaks in the positive and negative frequency regions, each of them with the shape of the Fourier transform of the windowing function. Since the maximum of W[r] is located in r=0, we finally have:

sum_k w[k] = 2

White noise input

Once the normalization of the filter window has been imposed by the previous condition, we need to define the bandwidth of the window such that noise measurements integrated over frequency give the right variances.

Let's take a white noise of variance 1.

<x[k] x[k']> = delta(k-k').

We would like the total spectrum in units of Vrms^2/Hz, integrated from 0 to Nyquist frequency to yield the same variance of 1. This is ensured by the Equivalent noise bandwidth of the filter window. To convert from V_pk^2 to V_rms^2/Hz, the spectrum is divided by the residual bandwidth of the filter window.

Let's calculate:

sum_r <|Y[r]|^2> = (...) = N sum_k w[k]^2 <|x[k]|^2>

If we remind that x[k] is a white noise following <|x[k]|^2> = 1, we get:

sum_r <|Y[r]|^2> = N sum_k w[k]^2

So, since we want:

sum_r <|Z[r]|^2> df = 2, (indeed, we want to work with single-sided spectra, such that integrating over positive frequencies is enough)

with df the frequency step in the FFT, we need to choose:

rbw = N sum_k w[k]^2 df /4

In order to use dimensionless parameters for the filter windows, we can introduce the equivalent noise bandwidth:

ENBW = sum_k w[k]^2/(sum_k w[k])^2 = 1/4 sum_k w[k]^2

Finally, we get the expression of the rbw:

rbw = sample_rate ENBW

IQ mode

In iq mode, the signal to measure is fed inside an iq module, and thus, multiplied by two sinusoids in quadrature with each other, and at the frequency `center_freq`. The resulting I and Q signals are then filtered by 4 first order filters in series with each other, with cutoff frequencies given by `span`. Finally, these signals are measured simultaneously with the 2 channels of the scope, and we form the complex time serie c_n = I_n + i Q_n. The procedure described above is applied to extract the periodogram from the complex time-serie.

Since the data are complex, there are as many independent values in the FFT than in the initial data (in other words, negative frequencies are not redundant with positive frequency). In fact, the result is an estimation of the spectrum in the interval [center_freq - span/2, center_freq + span/2].

Baseband

In baseband mode, the signal to measure is directly fed to the scope and the procedure described above is applied directly. There are 2 consequences of the fact that the data are real:

1. The negative frequency components are complex conjugated (and thus redundant) wrt the positive ones. We thus throw away the negative frequencies, and only get a measurement on the interval [0, span/2]

2. The second scope channel can be used to measure another signal.

It is very interesting to measure simultaneously 2 signals, because we can look for correlations between them. In the frequency domains, these correlations are most easily represented by the cross-spectrum. We estimate the cross-spectrum by performing the product `conjugate(fft1)*fft2`, where `fft1` and `fft2` are the DFTs of the individual scope channels before taking their modulus square.

Hence, in baseband mode, the method `curve()` returns a 4x2^13 array with the following content: - spectrum1 - spectrum2 - real part of cross spectrum - imaginary part of cross spectrum

Proposal for a cleaner interface for spectrum analyzer:

To avoid baseband/2-channels acquisition from becoming a big mess, I suggest the following:

- The return type of the method `curve` should depend as little as possible from the particular settings of the instrument (`channel2_baseband_active`, `display_units`). That was the idea with scope, and I think that makes things much cleaner. Unfortunately, for `baseband`, making 2 parallel piplines such as `curve_iq`, `curve_baseband` is not so trivial, because `curve()` is already part of the `AcquisitionModule`. So I think we will have to live with the fact that `curve()` returns 2 different kinds of data in `baseband` and `iq-mode`.
- Moreover, in baseband, we clearly want both individual spectra + cross-spectrum to be calculated from the beginning, since once the `abs()` of the `ffts` is taken, it is already too late to compute `conjugate(fft1)*fft2`
- Finally, I suggest to return all spectra with only one "internal unit" which would be `V_pk^2`: indeed, contrary to rms-values unittesting doesn't require any conversion with peak values, moreover, averaging is straightforward with a quadratic unit, finally, `.../Hz` requires a conversion-factor involving the bandwidth for unittesting with coherent signals

I suggest the following return values for `curve()`:

- In normal (iq-mode): `curve()` returns a real valued 1D-array with the normal spectrum in `V_pk^2`
- In baseband: `curve()` returns a 4xN/2-real valued array with `(spectrum1, spectrum2, cross_spectrum_real, cross_spectrum_imag)`. Otherwise, manipulating a complex array for the 2 real spectra is painful and inefficient.

Leo: Seems okay to me. One can always add functions like spectrum1() or cross_spectrum_complex() which will take at most two lines. Same for the units, I won't insist on rms, its just a matter of multiplying sqrt(1/2). However, I suggest that we then have 3-4 buttons in the gui to select which spectra and cross-spectra are displayed.

Yes, I am actually working on the gui right now: There will be a baseband-area, where one can choose `display_input1_baseband`, `input1_baseband`, `display_input2_baseband`, `input2_baseband`, `display_cross_spectrum`, 'display_cross_spectrum_phase'. And a "iq-area" where one can choose `center_frequency` and `input`. I guess this is no problem if we have the 3 distinct attributes `input`, `input1_baseband` and `input2_baseband`, it makes thing more symmetric...

IQ mode with proper anti-aliasing filter

When the IQ mode is used, a part of the broadband spectrum of the two quadratures is to be sampled at a significantly reduced sampling rate in order to increase the number of points in the spectrum, and thereby resolution bandwidth. Aliasing occurs if significant signals above the scope sampling rate are thereby under-sampled by the scope, and results in ghost peaks in the spectrum. The ordinary way to get rid of this effect is to use excessive digital low-pass filtering with cutoff frequencies slightly below the scope sampling rate, such that any peaks outside the band of interest will be rounded off to zero. The following code implements the design of such a low-pass filter (we choose an elliptical filter for maximum steepness):

```
import numpy as np
from scipy import signal
import matplotlib.pyplot as plt

# the overall decimation value
decimation = 8

# elliptical filter runs at ell_factor times the decimated scope sampling rate
ell_factor = 4

wp = 0.8/ell_factor # passband ends at xx% of nyquist frequency
ws = 1.0/ell_factor # stopband starts at yy% of nyquist frequency
gpass = 5. # jitter in passband (dB)
gstop = 20.*np.log10(2**14)  # attenuation in stopband (dB)
#gstop = 60  #60 dB attenuation would only require a 6th order filter
N, Wn = signal.ellipord(wp=wp, ws=ws, gpass=gpass, gstop=gstop, analog=False)  # get␣
↪filter order
z, p, k = signal.ellip(N, gpass, gstop, Wn, 'low', False, output='zpk')  # get␣
↪coefficients for implementation
b, a = signal.ellip(N, gpass, gstop, Wn, 'low', False, output='ba')  # get␣
↪coefficients for plotting
w, h = signal.freqz(b, a, worN=2**16)
ww = np.pi / 62.5  # scale factor for frequency axis (original frequency axis goes up␣
↪to 2 pi)

# extent w to see what happens at higher frequencies
w = np.linspace(0, np.pi, decimation/ell_factor*2**16, endpoint=False)
# fold the response of the elliptical filter
hext = []
for i in range(decimation/ell_factor):
    if i%2 ==0:
        hext += list(h)
    else:
        hext += reversed(list(h))
h = np.array(hext)
# elliptical filter
h_abs = 20 * np.log10(abs(h))

# 4th order lowpass filter after IQ block with cutoff of decimated scope sampling rate
cutoff = np.pi/decimation
butter = 1.0/(1.+1j*w/cutoff)**4
butter_abs = 20 * np.log10(abs(butter))

# moving average decimation filter
M = float(decimation) # moving average filter length
mavg = np.sin(w*float(M)/2.0)/(sin(w/2.0)*float(M))
mavg_abs = 20 * np.log10(abs(mavg))

# plot everything together and individual parts
h_tot = h_abs + mavg_abs + butter_abs
plt.plot(w/ww, h_tot, label="all")
plt.plot(w/ww, h_abs, label="elliptic filter")
plt.plot(w/ww, butter_abs, label="butterworth filter")
plt.plot(w/ww, mavg_abs, label="moving average filter")


plt.title('Elliptical lowpass filter of order %d, decimation %d, ell_factor %d'%(N,␣
↪decimation, ell_factor))
```

```
plt.xlabel('Frequency (MHz)')
plt.ylabel('Amplitude (dB)')
plt.grid(which='both', axis='both')
plt.fill([ws/ww*np.pi/decimation*ell_factor, max(w/ww), max(w/ww), ws*np.pi/ww/
↪decimation*ell_factor], [max(h_abs), max(h_abs), -gstop, -gstop], '0.9', lw=0) #␣
↪stop
plt.fill([wp/ww*np.pi/decimation*ell_factor, min(w/ww), min(w/ww), wp*np.pi/ww/
↪decimation*ell_factor], [min(h_abs), min(h_abs), -gpass, -gpass], '0.9', lw=0) #␣
↪stop
plt.axis([min(w/ww), max(w/ww), min(h_abs)-5, max(h_abs)+5])
plt.legend()
plt.show()
plt.savefig('c://lneuhaus//github//pyrpl//doc//specan_filter.png',DPI=300)

print "Final biquad coefficients [b0, b1, b2, a0, a1, a2]:"
for biquad in signal.zpk2sos(z, p, k):
    print biquad
```

Fig. 1.1: Resulting filter

We see that a filter of 8th order, consisting of 4 sequential biquads is required. Since we do not require the span / sampling rate of the spectrum analyzer to be above roughly 5 MHz, we may implement the four biquads sequentially. Furthermore, for even lower values of the span, the filter can be fed with a reduced clock rate equal to the scope decimation factor divided by the variable 'decimation' in the filter design code above (4 in the example). For the aliasing of the lowpass filter passband not to cause problems in this case, we must in addition use the 4th order butterworth lowpass already available from the IQ module and the moving average filter of the scope. Then, as the plot shows, we can be sure that no aliasing occurs, given that no aliasing from the ADCs is present (should be guaranteed by analog Red Pitaya design).

The problem with our scheme is the complexity of introducing 2 (for the two quadratures) 4-fold biquads. This will not fit into the current design and must therefore be postponed to after the FPGA cleanup.

We could however opt for another temporary option, applicable only to stationary signals: Measure the spectrum twice or thrice with slightly shifted IQ demodulation frequency (at +- 10% of span and the actual center, as required above), and only plot the pointwise-minimum (with respect to the final frequency axis) of the obtained traces. This is simple and should be very effective (also to reduce the central peak at the demodulation freuqency), so i suggest we give it a try. Furthermore, it prepares the user that IQ spectra will only have 80% of the points in baseband mode, which will remain so after the implementation of the lowpass filter. The plot above shows that we do not have to worry about aliasing from multiple spans away if the bandwidth if the IQ module is se to the scope sampling rate (or slightly below). I am not aware that this method is used anywhere else, but do not see any serious problem with it.

MemoryTree

In general, Memory Tree is satisfactory.

Problems

1. The MemoryBranch doesn't implement the full API of a dict. This is not nice because things like set_setup_attributes(**self.c) are not possible. I guess the reason is that a dict needs to implement some public methods such as keys(), values() iter()... and that's another argument to remove the support for point-notation. -> Of course, this full API is impossible to implement when one assumes all properties without leading underscore to be dictionary entries. If you want to use **, you should read the API documentation of memoryTree and do set_setup_attributes(**self.c._dict)

Base classes Attributes and Module

Two concepts are central to almost any object in the API: Attributes and Modules.

Attributes are essentially variables which are automatically synchronized between a number of devices, i.e. the value of an FPGA register, a config file to store the last setting on the harddisk, and possibly a graphical user interface.

Modules are essentially collections of attributes that provide additional functions to usefully govern the interaction of the available attributes.

It is recommended to read the definition of these two classes, but we summarize the way they are used in practice by listing the important methods:

Module (see BaseModule in module.py)

A module is a component of pyrpl doing a specific task, such as e.g. Scope/Lockbox/NetworkAnalyzer. The module can have a widget to interact with it graphically.

It is composed of attributes (see attributes.py) whose values represent the current state of the module (more precisely, the state is defined by the value of all attributes in _setup_attributes)

The module can be slaved or freed by a user or another module. When the module is freed, it goes back to the state immediately before being slaved. To make sure the module is freed, use the syntax:

with pyrpl.mod_mag.pop('owner') as mod: mod.do_something()

public methods

- get_setup_attributes(): returns a dict with the current values of the setup attributes
- set_setup_attributes(**kwds): sets the provided setup_attributes (setup is not called)
- save_state(name): saves the current "state" (using get_setup_attribute) into the config file
- load_state(name): loads the state 'name' from the config file (setup is not called by default)
- create_widget(): returns a widget according to widget_class
- setup(**kwds): first, performs set_setup_attributes**(kwds), then calls _setup() to set the module ready for acquisition. This method is automatically created by ModuleMetaClass and it combines the docstring of individual setup_attributes with the docstring of _setup()
- free: sets the module owner to None, and brings the module back the state before it was slaved equivalent to module.owner = None)

Public attributes:

- name: attributed based on _section_name at instance creation (also used as a section key in the config file)
- states: the list of states available in the config file
- owner: (string) a module can be owned (reserved) by a user or another module. The module is free if and only if owner is None
- pyrpl: recursively looks through parent modules until it reaches the pyrpl instance

class attributes to be implemented in derived class:

- individual attributes (instances of BaseAttribute)
- _setup_attributes: attribute names that are touched by setup(**kwds)/ saved/restored upon module creation
- _gui_attributes: attribute names to be displayed by the widget
- _callback_attributes: attribute_names that triggers a callback when their value is changed in the base class, _callback just calls setup()
- _widget_class: class of the widget to use to represent the module in the gui(a child of ModuleWidget)
- _section_name: the name under which all instances of the class should be stored in the config file

methods to implement in derived class:

- _init_module(): initializes the module at startup. During this initialization, attributes can be initialized without overwriting config file values. Practical to use instead of **init** to avoid calling super().**init**()
- _setup(): sets the module ready for acquisition/output with the current attribute's values. The metaclass of the module autogenerates a function like this: def setup(self, **kwds): *** docstring of function _setup * ***** for attribute in self.setup_attributes: print-attribute-docstring-here ****

  ```
  self.set_setup_attributes(kwds)
  return self._setup()
  ```

- _ownership_changed(old, new): this function is called when the module owner changes it can be used to stop the acquisition for instance.

Attributes

The parameters of the modules are controlled by descriptors deriving from BaseAttribute.

An attribute is a field that can be set or get by several means:

- programmatically: module.attribute = value
- graphically: attribute.create_widget(module) returns a widget to manipulate the value
- via loading the value in a config file for permanent value preservation

Attributes have a type (BoolAttribute, FloatAttribute...), and they are actually separated in two categories:

- Registers: the attributes that are stored in the FPGA itself
- Properties: the attributes that are only stored in the computer and that are not representing an FPGA register

A common mistake is to use the Attribute class instead of the corresponding Register or Property class (FloatAttribute instead of FloatRegister or FloatProperty for instance): this class is abstract since it doesn't define a set_value/get_value method to specify how the data is stored in practice.

Starting to rewrite SelectAttribute/Property

Guidelines: - Options must not be a bijection any more, but can be only an injection (multiple keys may correspond to the same value). - Options can be given as a dict, an OrderedDict, a list (only for properties - automatically converted into identity ordereddict), or a callable object that takes 1 argument (instance=None) and returns a list or a dict. - Options can be changed at any time, and a change of options should trigger a change of the options in the gui. -

Options should be provided in the right order (no sorting is performed in order to not mess up the predefined order. Use pyrpl_utils.sorted_dict() to sort you options if you have no other preferrence.

- The SelectProperty should simply save the key, and not care at all about the value.
- Every time a set/get operation is performed, the following things should be confirmed:
- the stored key is a valid option
- in case of registers: the stored value corresponds to the stored key. if not: priority is given to the key, which is set to make sure that value/key correspond. Still, an error message should be logged.
- if eventually, the key / value does not correspond to anything in the options, an error message should be logged. the question is what we should do in this case:

1. keep the wrong key -> means a SelectRegister does not really fulfill its purpose of selecting a valid options
2. issue an error and select the default value instead -> better

Default value: - self.default can be set to a custom default value (at module initialization), without having to comply with the options. - the self.default getter will be obliged to return a valid element of the options list. that is, it will first try to locate the overwritten default value in the options. if that fails, it will try to return the first option. if that fails, too, it will return None

AcquisitionModule

A proposal for a uniformized API for acquisition modules (Scope, SpecAn, NA)

Acquisition modules have 2 modes of operation: the synchronous (or blocking) mode, and the asynchronous (or non-blocking mode). The curves displayed in the graphical user interface are based on the asynchronous operation.

Synchronous mode:

The working principle in synchronous mode is the following:

1. setup(**kwds): kwds can be used to specify new attribute values (otherwise, the current values are used)
2. (optional) curve_ready(): returns True if the acquisition is finished, False otherwise.
3. curve(timeout=None): returns a curve (numpy arrays). The function only returns when the acquisition is done, or a timeout occurs. The parameter timeout (only available for scope and specan) has the following meaning:

   timeout>0: timeout value in seconds

   timeout<=0: returns immediately the current buffer without checking for trigger status.

   timeout is None: timeout is auto-set to twice the normal curve duration

No support for averaging, or saving of curves is provided in synchronous mode

Asynchronous mode

The asynchronous mode is supported by a sub-object “run” of the module. When an asynchronous acquisition is running and the widget is visible, the current averaged data are automatically displayed. Also, the run object provides a function save_curve to store the current averaged curve on the hard-drive.

The full API of the “run” object is the following.

public methods (All methods return immediately)

- single(): performs an asynchronous acquisition of avg curves. The function returns a promise of the result: an object with a ready() function, and a get() function that blocks until data is ready.
- continuous(): continuously acquires curves, and performs a moving average over the avg last ones.
- pause(): stops the current acquisition without restarting the averaging
- stop(): stops the current acquisition and restarts the averaging.
- save_curve(): saves the currently averaged curve (or curves for scope)
- curve(): the currently averaged curve

Public attributes:

- curve_name: name of the curve to create upon saving
- avg: number of averages (not to confuse with averaging per point)
- data_last: array containing the last curve acquired
- data_averaged: array containing the current averaged curve
- current_average: current number of averages

—> I also wonder if we really want to keep the running_state/running_continuous property (will be uniformized) inside the _setup_attribute. Big advantage: no risk of loading a state with a continuous acquisition running without noticing/big disadvantage: slaving/restoring a running module would also stop it...

Lockbox

Lockbox is the base class for all customizations of lockbox behavior. Any customized lockbox is implemented by defining a class that inherits from Lockbox. This allows to add custom functionality to preexisting lockbox types and furthermore to easily overwrite the default functions of the lockbox API with custom behaviour.

The general way to implement a custom lockbox class is to copy the file "pyrpl/software_modules/lockbox/models/custom_lockbox_example.py" into the folder "*:math:'PYRPL_USER_DIR* <https://github.com/lneuhaus/pyrpl/wiki/Installation:-Directory-for-user-data-%22PYRPL_USER_DIR%22>'__/lockbox" and to start modifying it. PyRPL will automatically search this directory for classes that have Lockbox as one base class and allow to select these by setting the corresponding class name in the property 'classname' of a Lockbox instance.

Each time the Lockbox type is changed in this way, (can happen through the API, the GUI or the configfile, i.e. `pyrpl.lockbox.classname = 'FabryPerot'`), a new Lockbox object is created from the corresponding derived class of Lockbox. This ensures that the Lockbox and all its signals are properly initialized.

To keep the API user-friendly, two things should be done - since Lockbox inherits from SoftwareModule, we must keep the namespace in this object minimum. That means, we should make a maximum of properties and methods hidden with the underscore-trick.

- the derived Lockbox object should define a shortcut class object 'sc' that contains the most often used functions.

The default properties of Lockbox are

- inputs: list or dict of inputs —> a list is preferable if we want the input name to be changeable, otherwise the "name" property becomes redundant with the dict key. But maybe we actually want the signal names to be defined in the Lockbox class itself?

- outputs: list or dict of outputs —> same choice to make
- function lock(setpoint, factor=1) —> Needs to be well documented: for instance, I guess setpoint only applies to last stage and factor to all stages ? —> Also, regarding the discussion about the return value of the function, I think you are right that a promise is exactly what we need. It can be a 5 line class with a blocking get() method and a non-blocking ready() method. We should use the same class for the method run.single() of acquisition instruments.
- function unlock()
- function sweep()
- function calibrate() –> I guess this is a blocking function ?
- property islocked
- property state

—> Sequence (Stages) are missing in this list. I would advocate for keeping a "sequence" container for stages since it can be desirable to only manipulate the state of this submodule (especially with the new YmlEditor, editing the sequence alone becomes a very natural thing to do). I agree that the current implementation where all the sequence management functions are actually delegated to Lockbox is garbage.

—> Now that we are at the point where one only needs to derive Lockbox (which I believe makes sense), we could also simplify our lives by making both the list of inputs and outputs fixe sized: they would both be specified by a list-of-class in the LockboxClass definition. If the names are also static, then it would probably be a list of tuples (name, SignalClass) or an OrderedDict(name, SignalClass). I guess adding a physical output is rare enough that it justifies writing a new class?

PS: regarding the specification of the pairs (name, signal) in the Lockbox class. I just realized that if we want the lists to be fixe-sized, the cleanest solution is to use a descriptor per input (same for outputs). This is exactly what they are made for...

@Samuel: What is the advantage of your solution to saving inputs and outputs as (Ordered)Dicts?

DataWidget

There are many places in pyrpl where we need to plot datasets. A unified API to deal with the following needs would make the code more maintainable: - Plotting several channels on the same widget (for instance with a multidimensional array as input) - Automatic switching between real and complex datasets (with appearance/disappearance of a phase plot) - Dealing with different transformations for the magnitude (linear, dB, dB/Hz...). Since we would like the internal data to stay as much as possible independent of the unit chosen for their graphical representation, I would advocate for the possibility to register different unit/conversion_functions options (mainly for the magnitude I guess) at the widget level. - For performance optimization, we need to have some degree of control over how much of the dataset needs to be updated. For instance, in the network analyzer, there is currently a custom signal: update_point(int). When the signal is emitted with the index of the point to update, the widget waits some time (typically 50 ms) before actually updating all the required points at once. Moreover, the curve is updated by small fragments (chunks) to avoid the bottleneck of redrawing millions of points every 50 ms for very long curves.

If we only care for the 3 first requirements, it is possible to make a pretty simple API based on the attribute/widget logic (eventhough we need to define precisely how to store the current unit). For the last requirement, I guess we really need to manually create a widget (not inheriting from AttributeWidget, and deal manually with the custom signal handling).

That's why, I propose a DataWidget (that doesn't inherit from AttributeWidget) which would expose an API to update the dataset point by point and a DataAttributeWidget, that would ideally be based on DataWidget (either inheritance or possession) to simply allow module.some_dataset = some_data_array.

Another option is to keep the current na_widget unchanged (since it is already developed and working nicely even for very large curves), and develop a simple DataAttributeWidget for all the rest of the program.

The last option is probably much easier to implement quickly, however, we need to think whether the point-by-point update capability of the na_widget was a one-time need or whether it will be needed somewhere else in the future...

1.5.7 Distribution of pyrpl

How to generate the automatic documentation with Sphinx

For the automatic documentation to work, please follow the code style guidelines for docstrings. To compile the autodoc with sphinx, simply install sphinx > 1.3 (`pip install sphinx`) and type (starting from the pyrpl root directory)

```
cd doc/sphinx
make html
```

An extensive discussion of the (automatic) documentation can be found in issue #291.

A few useful links and older information (from issue #85):

- We should implement this in order to view the autodoc online, preferentially by having travis perform a build of the autodoc at each commit: https://daler.github.io/sphinxdoc-test/includeme.html
- The good commands for building autodoc are described here: http://gisellezeno.com/tutorials/sphinx-for-python-documentation.html
- These commands are: `cd doc/sphinx sphinx-apidoc -f -o source/ ../../pyrpl/ make html`

Current version of autodoc: https://github.com/lneuhaus/pyrpl/blob/master/doc/sphinx/build/html/pyrpl.html

How to make a single-file pyrpl executable not depending on a Python installation

In the pyrpl root dir:

```
conda create -y -n py34 python=3.4 numpy scipy paramiko pandas nose pip pyqt qtpy
activate py34
python setup.py develop
pip install pyinstaller
pyinstaller --clean --onefile --distpath dist -n pyrpl ./scripts/run_pyrpl.py
```

We now use spec files in order to include the fpga bitfile in the bundle. This requires only

```
pyi-makespec --onefile -n pyrpl ./scripts/run_pyrpl.py
# add datas section to the file...
# datas=[('pyrpl/fpga/red_pitaya.bin', 'pyrpl/fpga'),
         ('pyrpl/monitor_server/monitor_server*',
          'pyrpl/monitor_server')],
pyinstaller pyrpl.spec
```

Prepare a new release

The process of deploying new releases is automated in the file `.travis.yml` and is triggered when a new tag is created on github.com. This page contains what to do if you want to manually deploy a new release.

First, we install a bunch of programs:

```
conda create -y -n py34 python=3.4 numpy scipy paramiko pandas nose pip pyqt qtpy
activate py34
python setup.py develop
pip install pyinstaller
```

Then, for the actual build:

```
# do everything in python 3.4 for compatibility reasons
activate py34

# Readme file must be converted from Markdown to ReStructuredText to be displayed␣
↪correctly on Pip
pandoc --from=markdown --to=rst --output=README.rst README.md

# Next, we must build the distributions (we provide source and binary):
python setup.py sdist
python setup.py bdist_wheel --universal

# Last, make a windows executable file
pyinstaller pyrpl.spec

# Eventually we upload the distribution using twine:
twine upload dist/*
```

1.5.8 SD card preparation

Option 0: Download and unzip the Red Pitaya OS Version 0.92 image. Flash this image on a >= 4 GB SD card using a tool like Win32DiskImager, and insert the card into your Red Pitaya.

Option 1: flash the full image at once

For the SD card to be bootable by the redpitaya, several things need to be ensured (Fat32 formatting, boot flag on the right partition...), such that simply copying all required files onto the SD card is not enough to make it bootable. The simplest method is to copy bit by bit the content of an image file onto the sd card (including partition table and flags). On windows, this can be done with the software Win32DiskImager. The next section provides a detailed procedure to make the SD card bootable starting from the list of files to be copied.

Option 2: Format and copy a list of files on the SD card

The previous method can be problematic, for instance, if the capacity of the SD card is too small for the provided image file (Indeed, even empty space in the original 4 GB card has been included in the image file). Hence, it can be advantageous to copy the files individually on the SD card, however, we need to pay attention to make the SD-card bootable. For this we need a Linux system. The following procedure assumes an Ubuntu system installed on a virtualbox:

1. Open the ubuntu virtualbox on a computer equipped with a SD card reader.
2. To make sure the SD card will be visible in the virtualbox, we need to go to configuration/usb and enable the sd card reader.
3. Open the ubuntu virtual machine and install gparted and dosfstools with the commands:

   ```
   sudo apt-get install gparted
   sudo apt-get install dosfstools
   ```

4. Insert the sd card in the reader and launch gparted on the corresponding device (/dev/sdb in this case but the correct value can be found with “dmesg | tail”):

```
sudo gparted /dev/sdb
```

5. In the gparted interface, delete all existing partitions, create a partition map if there is not already one, then create 1 fat32 partition with the maximum space available. To execute these operations, it is necessary to unmount the corresponding partitions (can be done within gparted).
6. Once formatted, right click to set the flag “boot” to that partition.
7. Close gparted, remount the sd card (by simply unplugging/replugging it), and copy all files at the root of the sd card (normally mounted somewhere in /media/xxxx)

- *Full documentation structure*

1.6 Full documentation structure

- *Installation*
- *GUI instruments manual*
- *API manual*
- *Basics of the PyRPL Architecture*
- *Notes for developers*
- *Full documentation structure*

CHAPTER 2

Low-level API example

```
# import pyrpl library
import pyrpl

# create an interface to the Red Pitaya
r = pyrpl.Pyrpl().redpitaya

r.hk.led = 0b10101010  # change led pattern

# measure a few signal values
print("Voltage at analog input1: %.3f" % r.sampler.in1)
print("Voltage at analog output2: %.3f" % r.sampler.out2)
print("Voltage at the digital filter's output: %.3f" % r.sampler.iir)

# output a function U(t) = 0.5 V * sin(2 pi * 10 MHz * t) to output2
r.asg0.setup(waveform='sin',
             amplitude=0.5,
             frequency=10e6,
             output_direct='out2')

# demodulate the output signal from the arbitrary signal generator
r.iq0.setup(input='asg0',   # demodulate the signal from asg0
            frequency=10e6,  # demodulaltion at 10 MHz
            bandwidth=1e5)  # demodulation bandwidth of 100 kHz

# set up a PID controller on the demodulated signal and add result to out2
r.pid0.setup(input='iq0',
             output_direct='out2',  # add pid signal to output 2
             setpoint=0.05, # pid setpoint of 50 mV
             p=0.1,  # proportional gain factor of 0.1
             i=100,  # integrator unity-gain-frequency of 100 Hz
             input_filter = [3e3, 10e3])  # add 2 low-passes (3 and 10 kHz)

# modify some parameters in real-time
r.iq0.frequency += 2.3  # add 2.3 Hz to demodulation frequency
r.pid0.i *= 2  # double the integrator unity-gain-frequency
```

```
# take oscilloscope traces of the demodulated and pid signal
data = r.scope.curve(input1='iq0', input2='pid0',
                     duration=1.0, trigger_source='immediately')
```

CHAPTER 3

High-level API example

```
# import pyrpl library
import pyrpl

# create a Pyrpl object and store the configuration in a file 'filter-cavity.yml'
p = pyrpl.Pyrpl(config='filter-cavity')

# ... connect hardware (a Fabry-Perot cavity in this example) and
#     configure its paramters with the PyRPL GUI that shows up

# sweep the cavity length
p.lockbox.sweep()

# calibrate the cavity parameters
p.lockbox.calibrate()

# lock to the resonance with a predefined sequence
p.lockbox.lock()

# launch two different measurements simultaneously
transfer_function = p.network_analyzer.single_async(
        input='lockbox.reflection', output='out2',
        start=1e3, stop=1e6, points=10000, rbw=1000)
spectrum = p.spectrum_analyzer.single_async(
        input='in2', span=1e5, trace_averages=10)

# wait for measurements to finish
while not transfer_function.done() and not spectrum.done():
    # check whether lock was lost
    if not p.lockbox.is_locked():
        # re-lock the cavity
        p.lockbox.relock()
        # re-start measurements
        transfer_function = p.network_analyzer.single_async()
        spectrum = p.spectrum_analyzer.single_async()
```

```
# display a measurement result in the curve browser
p.curve_viewer.curve = transfer_function.result()
```

CHAPTER 4

Feedback by PyRPL users

PyRPL was developed for and is one of the core components of three ongoing experiments in Quantum Cavity Optomechanics

in the Optomechanics and Quantum Measurement Group at Laboratoire Kastler Brossel in Paris.

“Brilliant toolbox with an impressive number of functionalities. After quickly tuning a few parameters, we were able to lock our Fabry-Perot cavity with almost no effort.”

Dr. Jérôme Degallaix from the Laboratoire des Matériaux Avancés in Lyon.

“I am trying PyRPL to see if we can replace the really expensive ZI-lockings to generate parametric feedback at low frequencies.”

Dr. Pau Mestres from Rainer Blatt’s Quantum Optics and Spectroscopy Group at University of Innsbruck.

“IT WORKS! Thanks for your wonderful project :)”

Dr. Kun Huang from the State Key Laboratory of Precision Spectroscopy at East China Normal University.

“Thank you very much for your amazing PyRPL module! I have been using it continuously since last week, and it has saved us a lot of trouble already!”

Ivan Galinsky from the Quantum Membranes Lab, QUANTOP, Niels Bohr Institute, University of Copenhagen.

PyRPL is furthermore used by

- Team of Dr. Jonas Schou Neergaard-Nielsen at the Quantum Physics and Information Technology division of the Technical University of Denmark,
- Development team at Sacher Lasertechnik,
- Team of Dr. Pierre Verlot in the Luminescence Group at ILM, University Claude Bernard Lyon 1,
- Dr. Gordon A. Shaw, Mass and Force Group at the Quantum Metrology division of the National Institute of Standards and Technology (NIST).

If you are using PyRPL and would like to help to promote the project, please send your feedback to pyrpl.readthedocs.io@gmail.com and we will include your voice on this page.

CHAPTER 5

Publications about PyRPL

1. L. Neuhaus, R. Metzdorff, S. Chua, T. Jacqmin, T. Briant, A. Heidmann, P.-F. Cohadon, S. Deléglise, "PyRPL (Python Red Pitaya Lockbox) — An open-source software package for FPGA-controlled quantum optics experiments", 2017 Conference on Lasers and Electro-Optics Europe & European Quantum Electronics Conference (CLEO/Europe-EQEC), Munich, Germany, 2017.
2. L. Neuhaus, "Red Pitaya DAC performance', blog post, 2016. URL https://ln1985blog.wordpress.com/2016/02/07/red-pitaya-dac-performance/.
3. L. Neuhaus, "Adding voltage regulators for the RedPitaya output stage", blog post, 2016. URL https://ln1985blog.wordpress.com/2016/02/07/adding-voltage-regulators-for-the-redpitaya-output-stage/.

CHAPTER 6

Contributors

Below is partial list of PyRPL contributors. We do our best to keep this list updated. If you've been left off, please change this file by yourself or send an email to the maintainer (currently neuhaus@lkb.upmc.fr).

- Leonhard Neuhaus
- Samuel Deléglise
- Jonas Neergard-Nielsen
- Xueshi Guo
- Jerome Degallaix
- Pierre Clade
- Matthew Winchester
- Remi Metzdorff
- Kevin Makles
- Clement Chardin

CHAPTER 7

Funding and support

Work on PyRPL was partially funded and/or supported by the following organizations:

- Optomechanics and Quantum Measurement Group at the Laboratoire Kastler Brossel in Paris
- Agence nationale de recherche (ANR)
- Marie Curie Initial Training Network (ITN) “Cavity Quantum OptoMechanics” (CQOM)
- Centre National de la Recherche Scientifique (CNRS)
- Universite Pierre et Marie Curie (UPMC)

CHAPTER 8

About

PyRPL is open source software that allows to use FPGA boards with analog interfaces for measurement and control of real-world devices in physics and engineering, notably experiments in quantum optics. It was started in 2014 by Leonhard Neuhaus for controlling experiments in the field of quantum physics at the Laboratoire Kastler Brossel in Paris, France. Its was initially based on the open-source code for the Red Pitaya and gradually diverged away from it. In 2016, large parts of the graphical user interface were added to the project by Samuel Deleglise. PyRPL was finally published as an open-source project under the GNU General Public License, Version 3 and has been online since July 2017.

CHAPTER 9

Old documentation sections

The old documentation is obsolete and will soon be deleted. Please refer to the more recent documentation in the *Manual* section.

- gallery/index
- user_guide/index
- reference_guide/index
- *Notes for developers*
- indices_and_tables/index
- *Full documentation structure*

CHAPTER 10

Current build status

CHAPTER 11

Releases

11.1 Version 0.9.4.0

- Smoother transitions of output voltages during stage transitions in lockbox.
- Automatic Red Pitaya device search extended to multiple network adapters and most recent STEMLab OS v0.98.
- Improved documentation hosted on www.pyrpl.org and video tutorial on youtube.
- Binaries for Windows, Linux and Mac OSX automatically generated for new releases and available on source-forge.

11.2 Version 0.9.3.X

There are no release notes for PyRPL versions prior to version 0.9.4.

Index