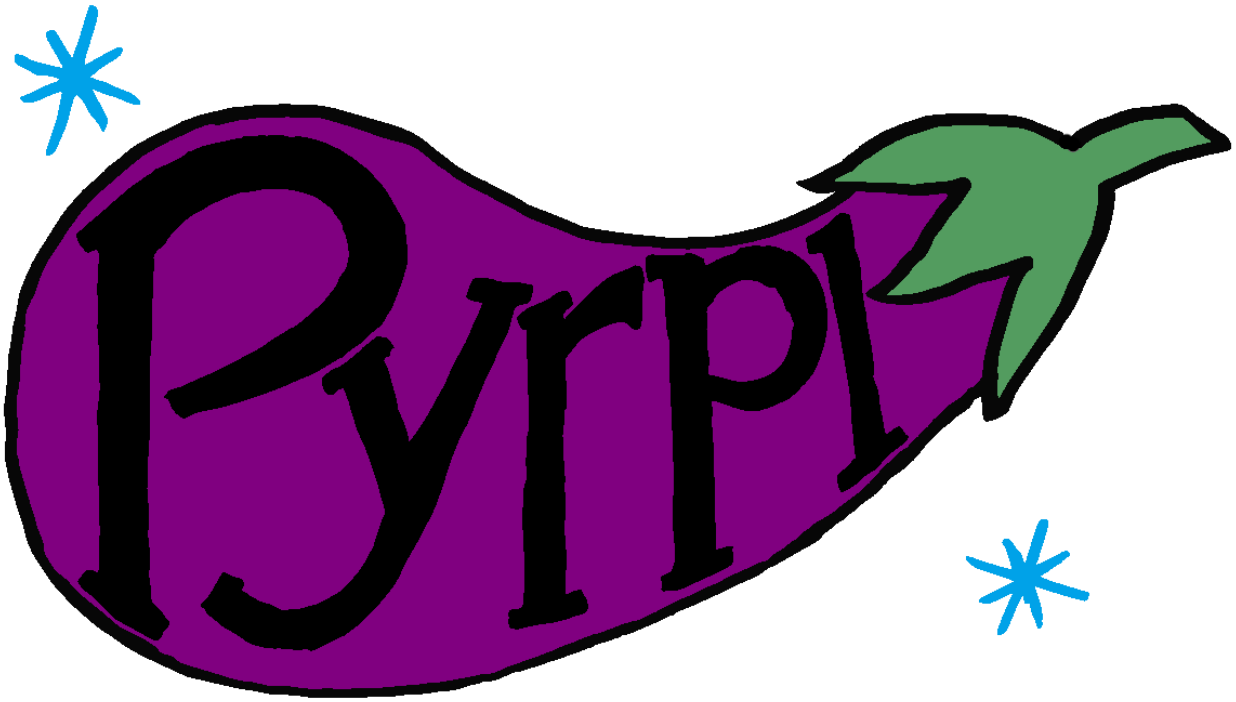


pyrpl Documentation

Release 0.9.3.6

Leonhard Neuhaus

Nov 15, 2017

Contents

The Red Pitaya (a.k.a. STEM Lab) (see full documentation) is an affordable FPGA board with fast analog inputs and outputs. This makes it interesting also for quantum optics experiments.

The software package PyRPL (Python RedPitaya Lockbox) is an implementation of many devices that are needed for optics experiments every day. Its user interface and all high-level functionality is written in python, but an essential part of the software is hidden in a custom FPGA design (based on the official RedPitaya software version 0.95). While most users probably never want to touch the FPGA design, the Verilog source code is provided together with this package and may be modified to customize the software to your needs.

CHAPTER 1

Gallery of PyRPL usage examples

We should add a gallery with use cases of Pyrpl to attract potential users.

This can be essentially a copy-paste from the PyRPL poster presented at CLEO Europe 2017 and the publication.

CHAPTER 2

User's guide to PyRPL

2.1 Installation

2.1.1 Hardware installation for PyRPL

The RedPitaya is an affordable FPGA board with fast analog inputs and outputs. Before starting, we need to prepare a bootable SD card for the RedPitaya. In principle, PyRPL is compatible with the lattest version of the RedPitaya OS, however, to prevent any compatibility issues, we provide here the version of the Redipatay OS against which PyRPL has been tested.

SD card preparation

Option 0: Download abd 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)

Communication with the Redpitaya

To make sure the SD card is bootable, insert it into the slot of the Redpitaya and plug the power supply. Connect the redpitaya to your local network with an ethernet cable and enter the IP-adress of the repitaya into an internet browser. The redpitaya welcome screen should show-up!

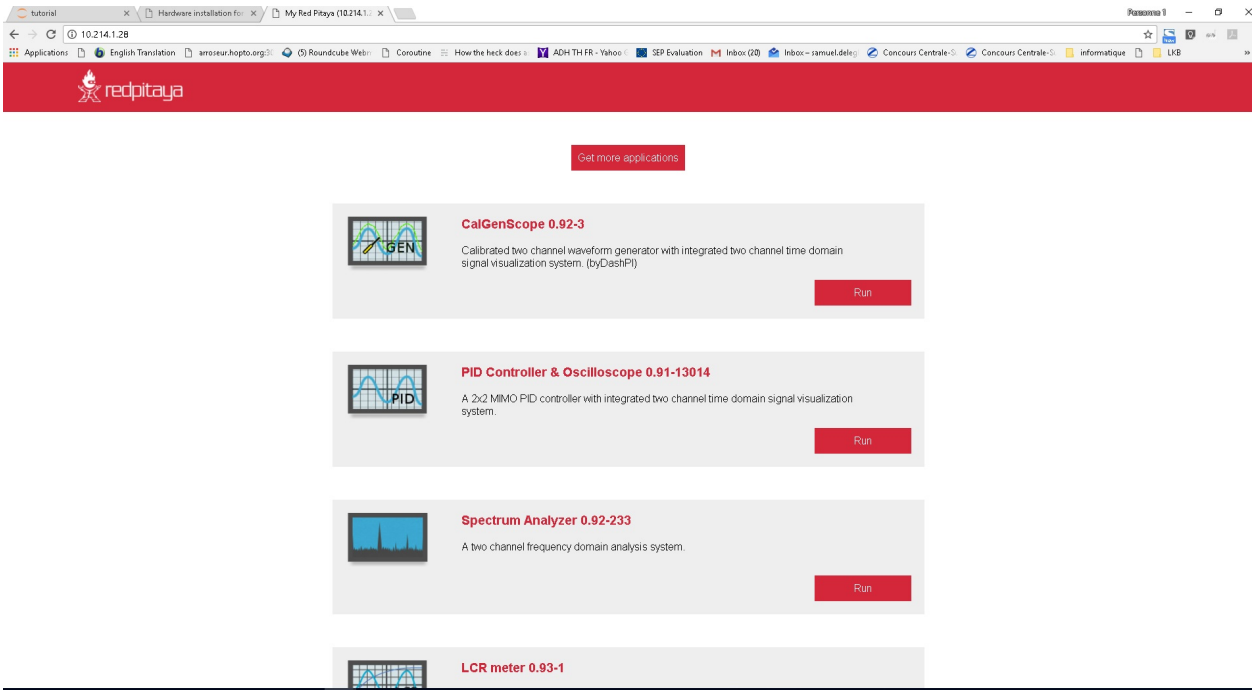


Fig. 2.1: This is the Redpitaya welcome screen.

Quick start

First, hook up your Red Pitaya / STEMlab to a LAN accessible from your computer (follow the instructions for this on redpitya.com and make sure you can access your Red Pitaya with a web browser by typing its ip-address / hostname into the address bar). In an IPython console or JuPyter notebook, type

```
from pyrpl import Pyrpl
p = Pyrpl(config='your_configuration_name', hostname='your_redpitaya_ip_address')
```

The GUI should open and you can start playing around with it. By calling pyrpl with different strings for 'your_configuration_name', your settings for a given configuration will be automatically remembered by PyRPL. You can drop the hostname argument after the first call of a given configuration. Different RedPitayas with different configuration names can be run simultaneously.

2.1.2 Installing PyRPL

The Fastest way: Running from binary files

The easiest and fastest way to get PyRPL running is to download and execute the precompiled executable for windows or linux. This option requires no extra programs to be installed on the computer. If you want the Pyrpl binaries for a Mac, please let us know by creating a new issue and we will prepare them for you.

The Hacker's way: Running the Python source code

If you would like to use and/or modify the source code, make sure you have an installation of Python (2.7, 3.4, 3.5, or 3.6).

Prerequisites: Getting the right Python installation

There are many ways to get the Python working with Pyrpl. The following list is non-exhaustive

Option 1: Installation from Anaconda

If you are new to Python or unexperienced with fighting installation issues, it is recommended to install the Anaconda Python distribution, which allows to install all PyRPL dependencies via:

```
conda install numpy scipy paramiko pandas nose pip pyqt qtpy pyqtgraph pyyaml
```

Check *Common issues with Anaconda* for hints if you cannot execute conda in a terminal. Alternatively, if you prefer creating a virtual environment for pyrpl, do so with the following two commands:

```
conda create -y -n pyrpl-env numpy scipy paramiko pandas nose pip pyqt qtpy pyqtgraph␣
↪pyyaml
activate pyrpl-env
```

Option 2: Installation on a regular (non-Anaconda) python version

If you are not using Anaconda, installing all Python packages required to run pyrpl can be difficult, frustrating and time-consuming, at least on Windows systems, so even if you have Python (should be one of the versions 2.7, 3.4, 3.5, or 3.6) already installed, you should reconsider "*Option 1: Installation from Anaconda*", especially since Anaconda

can be installed to work side-by-side with pre-existing Python installations if the (default) installation option to not modify environment variables is chosen.

The main reason for the difficulty to work without Anaconda on windows is that you need a C-compiler that works in harmony with the python package manager "pip" to compile the C-extensions of these packages. A workaround is to indivitually download and install the precompiled Python wheels by Christoph Gohlke. You should install the following packages this way before attempting installing Pyrpl: numpy, scipy, paramiko, pandas, PyQt4, pyqtgraph.

Even if you have a working C-compiler on your system, you must manually install the python package PyQt5 or PyQt4, since this package is not mangaged by the python package manager "pip".

All remaining dependencies will be automatically installed by setuptools when Pyrpl is installed (see the section *Downloading and installing PyRPL from source*).

Downloading and installing PyRPL from source

Various channels are available to obtain the PyRPL source code.

Option 1: Installation with pip (recommended, for standard users) ~~~~~~

If you have pip correctly installed, executing the following line in a command line should install pyrpl and all missing dependencies:

```
pip install pyrpl
```

Option 2: From github.com (for developers and tinkerers)

If you have a git client installed (recommended), clone the pyrpl repository to your computer with:

```
git clone https://github.com/lneuhaus/pyrpl.git YOUR_PYRPL_DESTINATION_FOLDER
```

If you do not want to install git on your computer, just download and extract the repository from github.com () the repository.

Install PyRPL by navigating with the command line terminal (the one where the pyrpl-env environment is active in case you are using anaconda) into the pyrpl root directory and typing:

```
python setup.py develop
```

2.1.3 Common installation problems

Common issues with Anaconda

Problem: The `conda` command does not work

Reason:

The default installation of Anaconda in Windows does not add the conda to the PATH environment variable. Therefore, the windows terminal does not find the program by default. To make it work:

Solution:

- Execute the following command in your terminal to activate the conda environment: `C:\Users\YOUR_USERNAME\Anaconda3\Scripts\activate`. Of course, you should insert

your own username, and possibly replace the initial part of the command with the actual conda installation directory on your computer. After this, the conda command should also work without problems.

- Another solution is to execute the Anaconda navigator, click on the left on "Environments", left-click on the arrow next to "root" in the list of environments, and click on "Open Terminal". In the terminal that opens, the conda command should now work such that you can install the PyRPL dependencies.

Problem: I cannot launch PyRPL, even though the installation finished successfullly

Reason:

The installation created the virtual environment "pyrpl-env" to install all required pyrpl dependencies in. When trying to launch PyRPL, you must do so from this virtual env.

Solution:

Make sure that you are in the virtual environment pyrpl-env. This can be accomplished by various ways:

- Launch Jupyter console or notebook from the Anaconda navigator after having selected the pyrpl-env environment and load pyrpl from there.
- Execute the following command in your terminal to activate the pyrpl-env environment: `C:\Users\YOUR_USERNAME\Anaconda3\Scripts\activate pyrpl-env`. Of course, you should insert your own username, and possibly replace the initial part of the command with the actual conda installation directory on your computer. After this, load Python, Jupyter, Ipython or the notebook as usual from the same terminal.
- Change the path in the batch file / link that you use to launch python. Instead of calling the executables in `C:\Users\YOUR_USERNAME\Anaconda3\Scripts\*` or the equivalent directory on your computer, use the ones in `C\UsersYOUR_USERNAME\Anaconda3\envs\pyrpl-env\Scripts\*`. Your python terminal will be directly in the right environment 'pyrpl-env'.

2.1.4 Directory for user data "PYRPL_USER_DIR"

By default, PyRPL will search the home folder "`HOME`" of the user and create the folder "`HOME`/pyrpl_user_dir". This folder contains three subfolders:

- `config` for the PyRPL configuration files of the user.
- `lockbox` for custom lockbox scripts, similar to the files "interferometer.py" or "custom_lockbox_example.py".
- `curve` for saved curves / measurement data.

By setting the environment variable `PYRPL_USER_DIR` to an existing path on the file system, the location of the PyRPL user directory can be modified. This is useful when the user data should be syncronized with a designated github repository, for example.

2.2 Quickstart Tutorial for PyRPL

You can download the tutorial in form of a `Jupyter notebook file` or in `HTML-form`.

2.3 Basics of PyRPL

???

CHAPTER 3

Reference guide to PyRPL

A basic hand-written page that lists the different python modules with a link to their autogenerated doc

CHAPTER 4

Developer's guide

4.1 Building the FPGA firmware

4.1.1 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.

4.1.2 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.

4.2 Unittests

4.3 Coding style guidelines

4.3.1 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

4.3.2 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.

4.3.3 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:

```
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.
        """
```

4.4 Workflow to submit code changes

4.4.1 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.

4.4.2 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.

4.5 API specifications from the moment of their development

4.5.1 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*.

4.5.2 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.

Constantly calling APP.processEvents()

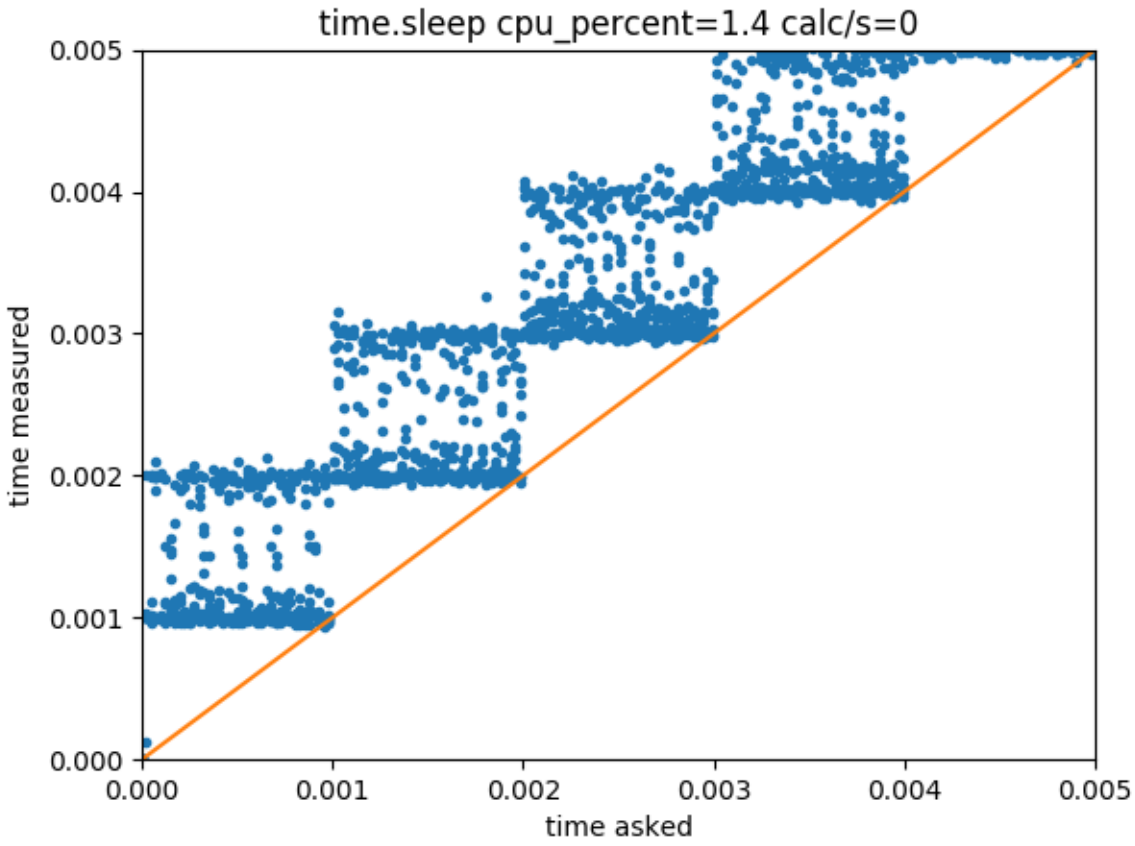
The first work around, is to manually call processEvents() regularly to make sure events are processed while our process is sleeping.

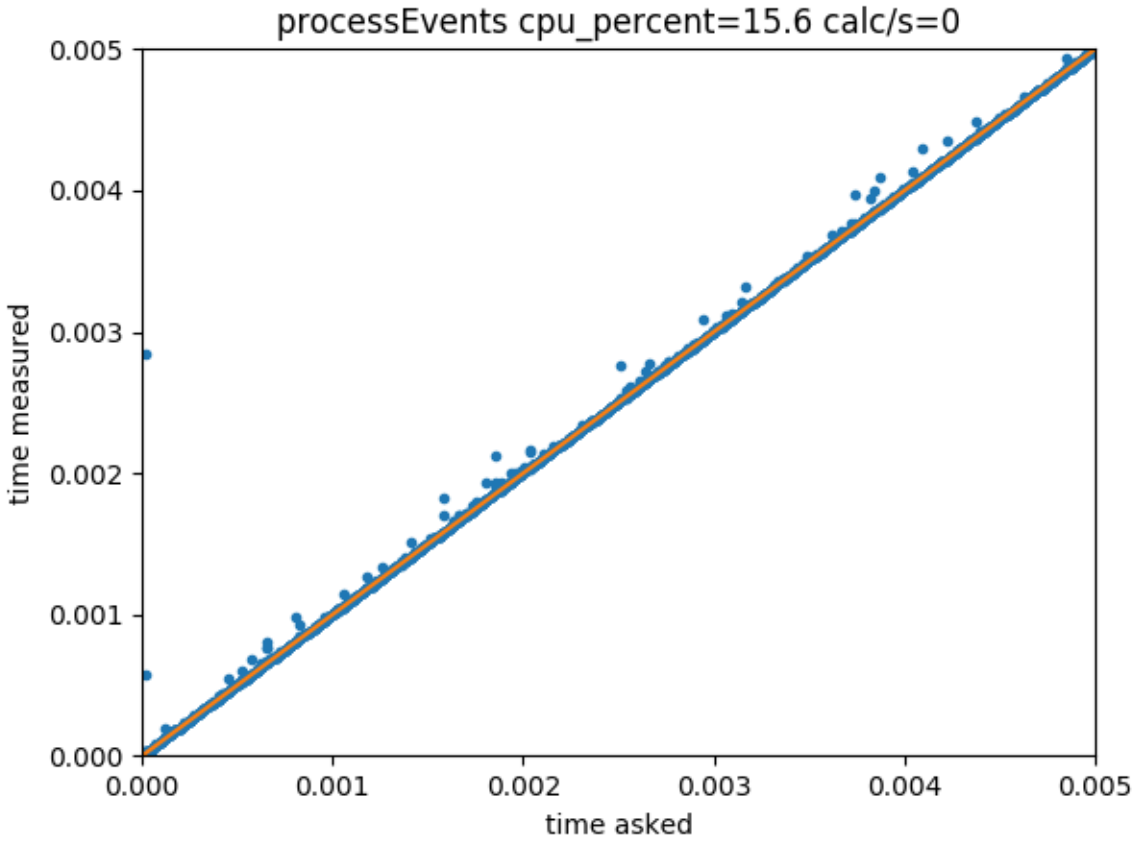
```
from timeit import default_timer

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.


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

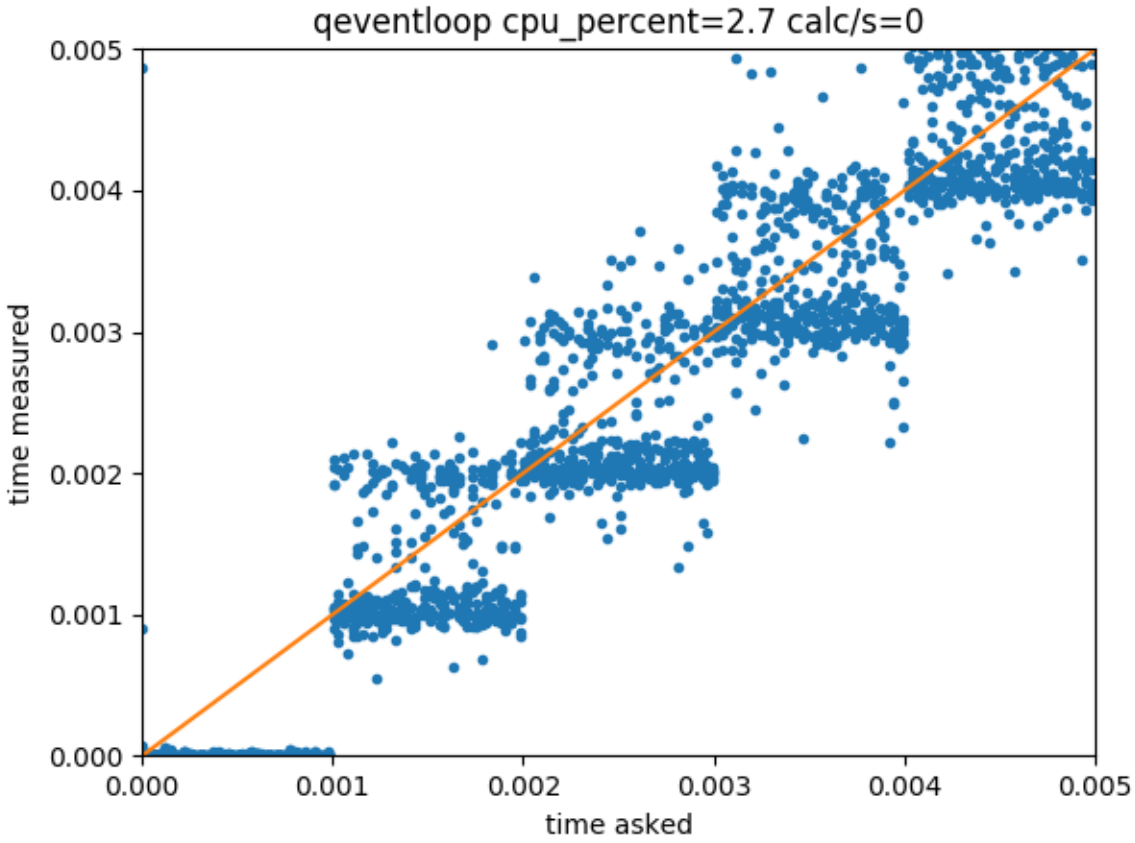

processEvents cpu_percent=15.6 calc/s=0
time measured
time asked
0.000
0.001
0.002
0.003
0.004
0.005

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.

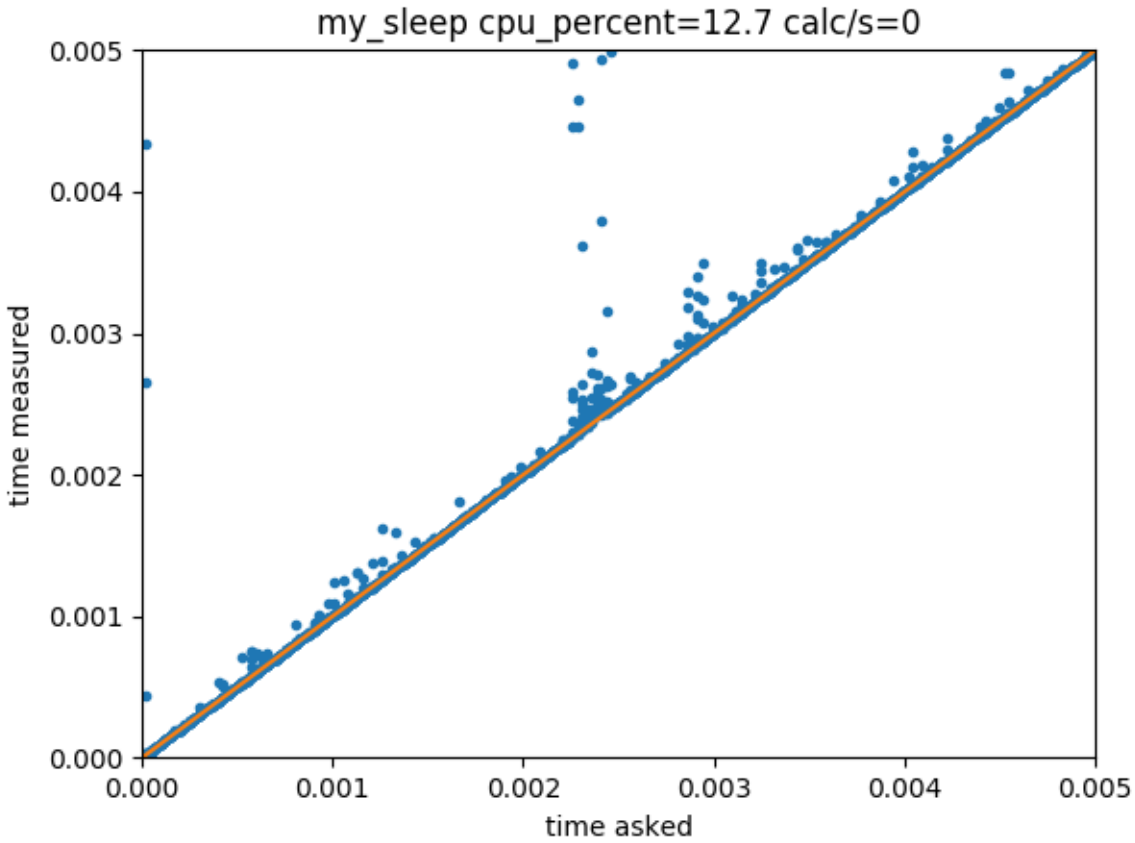


The hybrid approach

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:

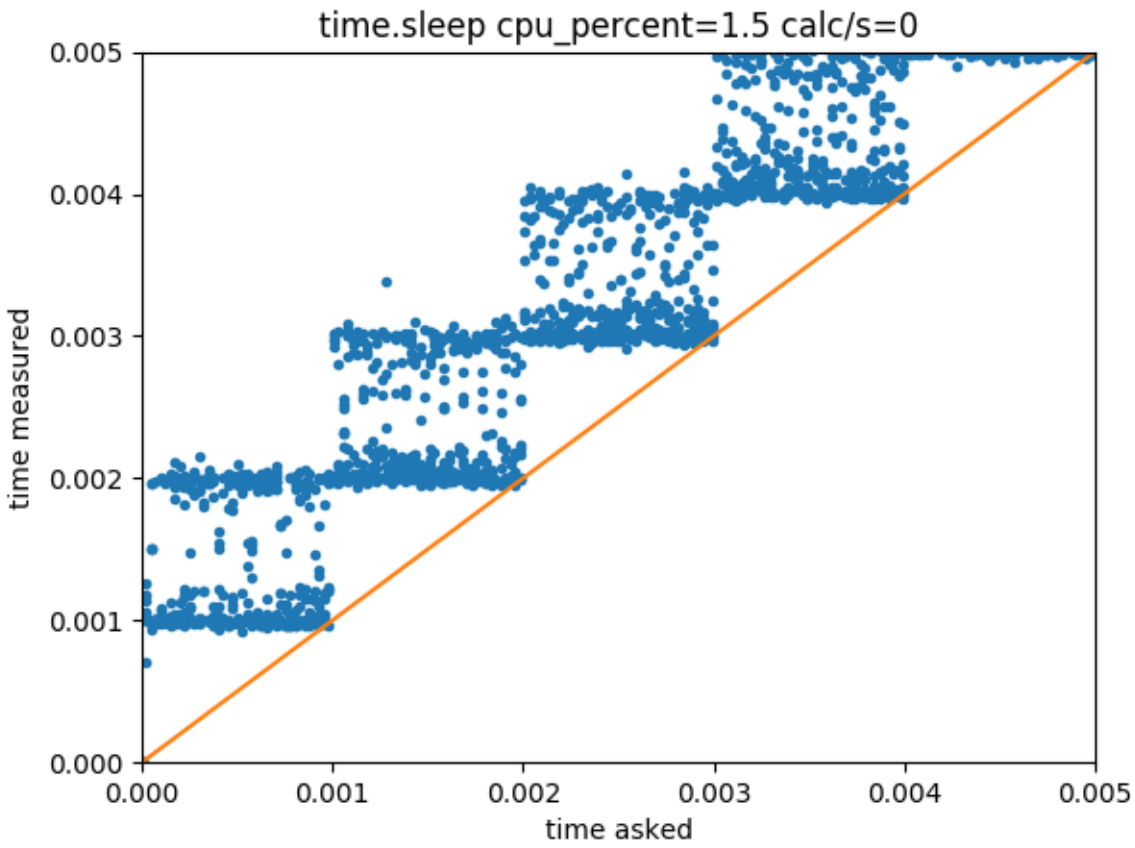
```
from PyQt4 import QtCore, QtGui
n_calc = [0]
def calculate():
    sin(rand(1000))
    n_calc[0]+=1
    timer.start()

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

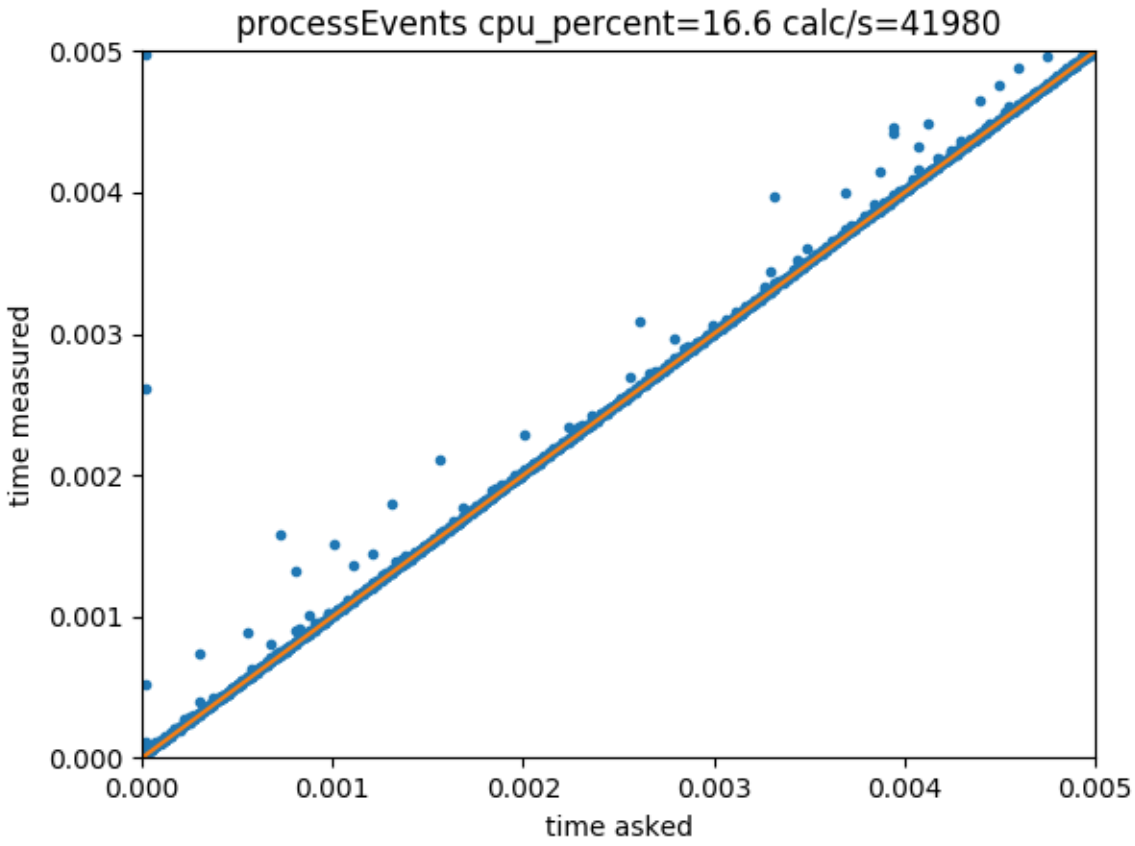


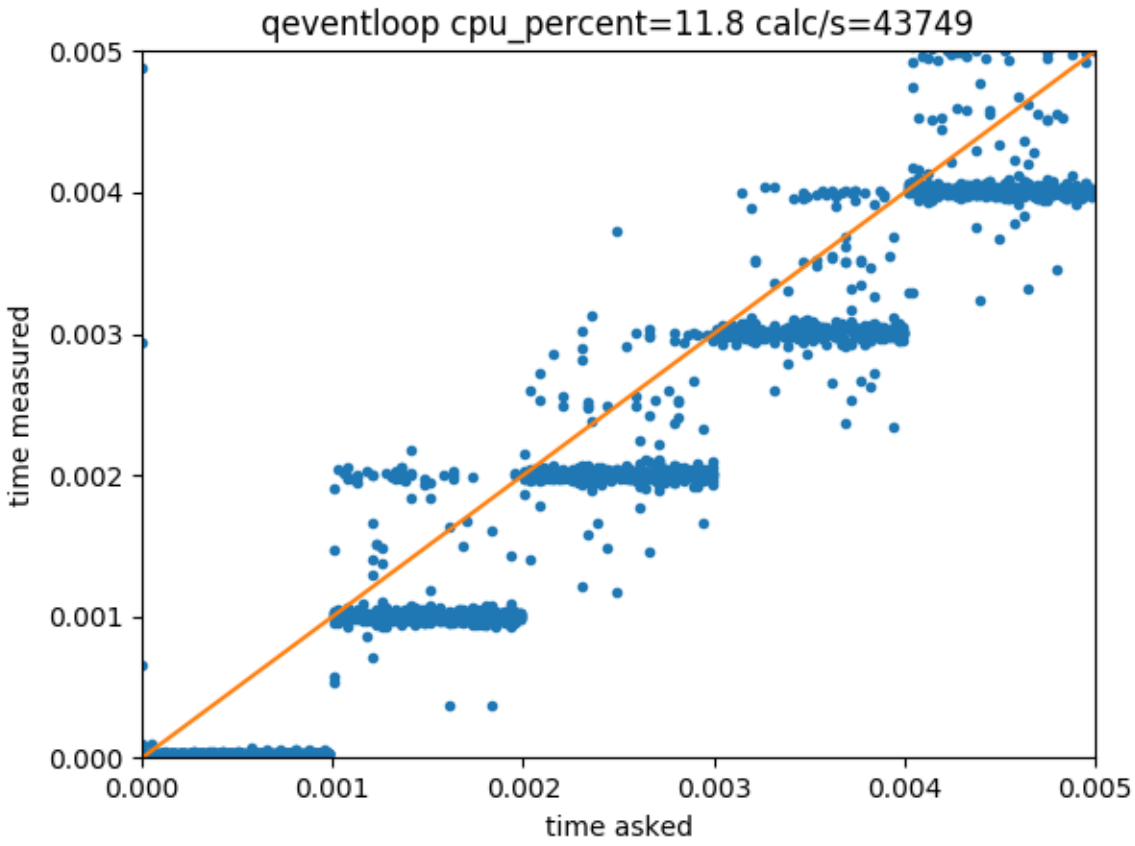
calling processEvents

40 000 events/seconds.

running the eventLoop locally

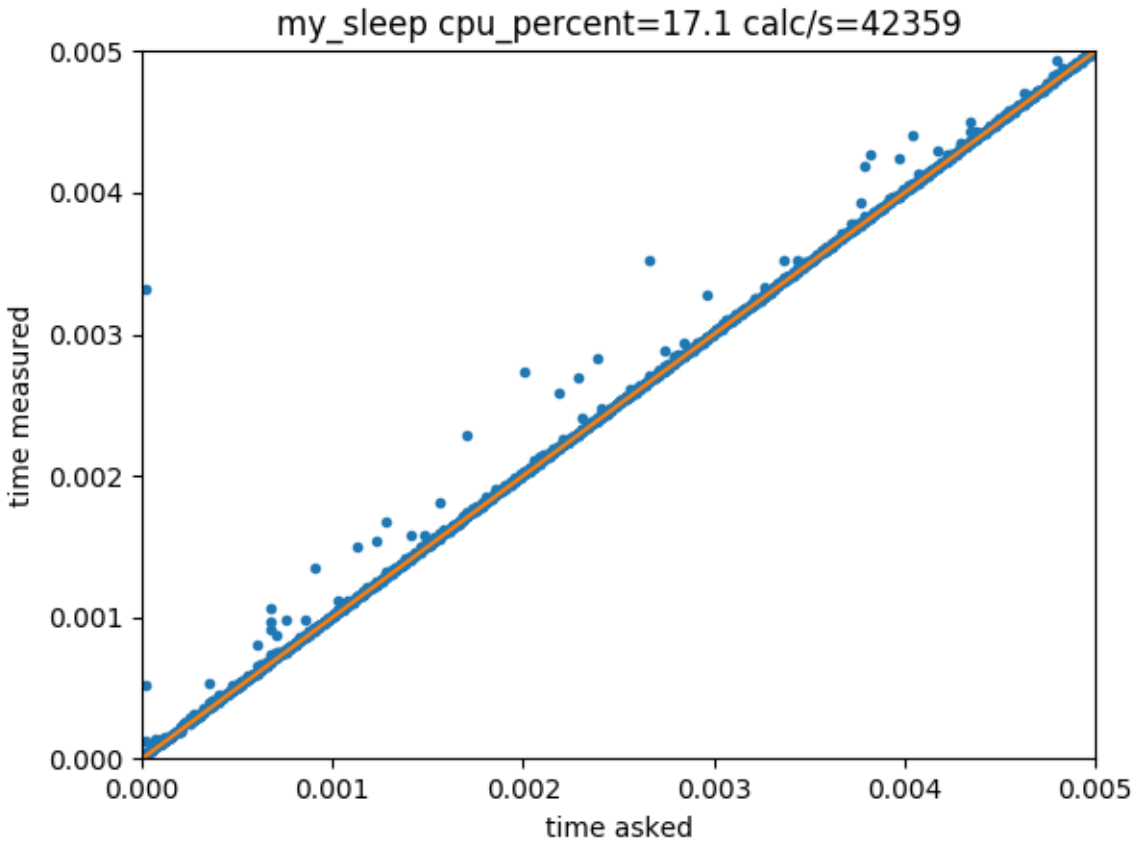
That's approximately identical


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).



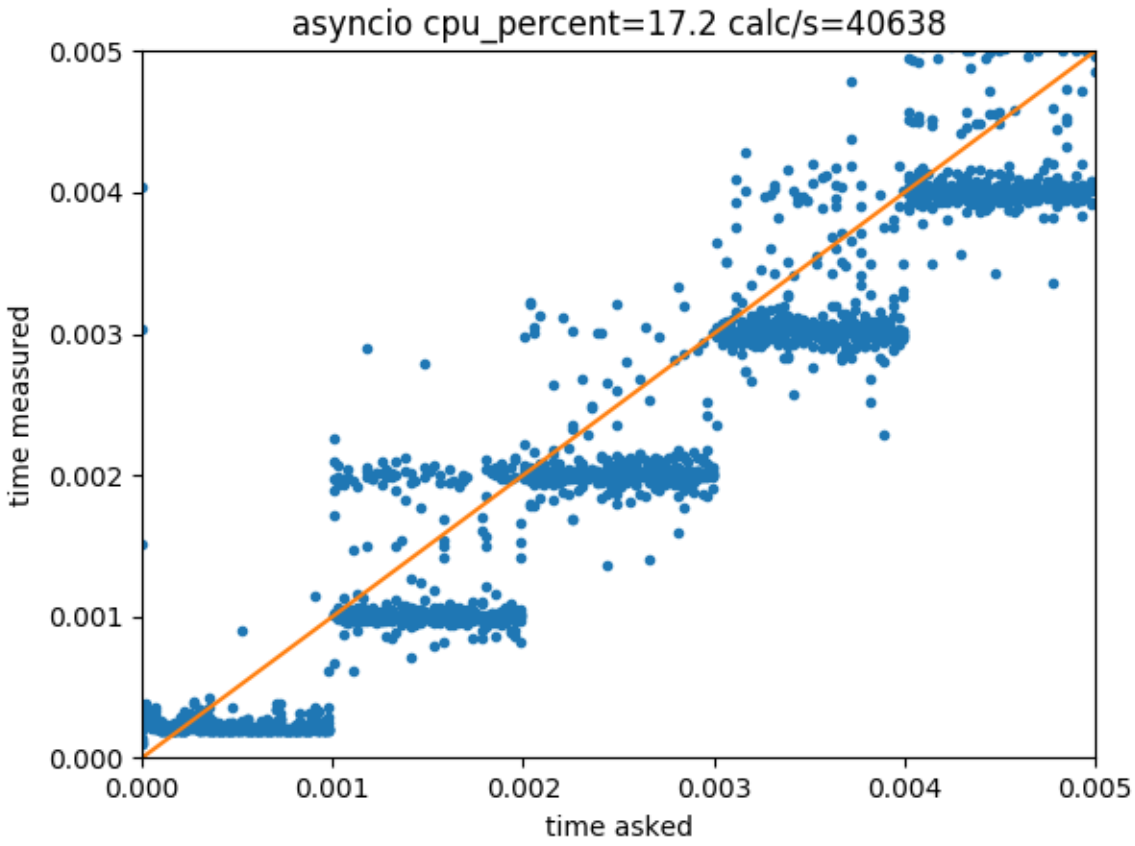
Async programming in python3(.5):

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":

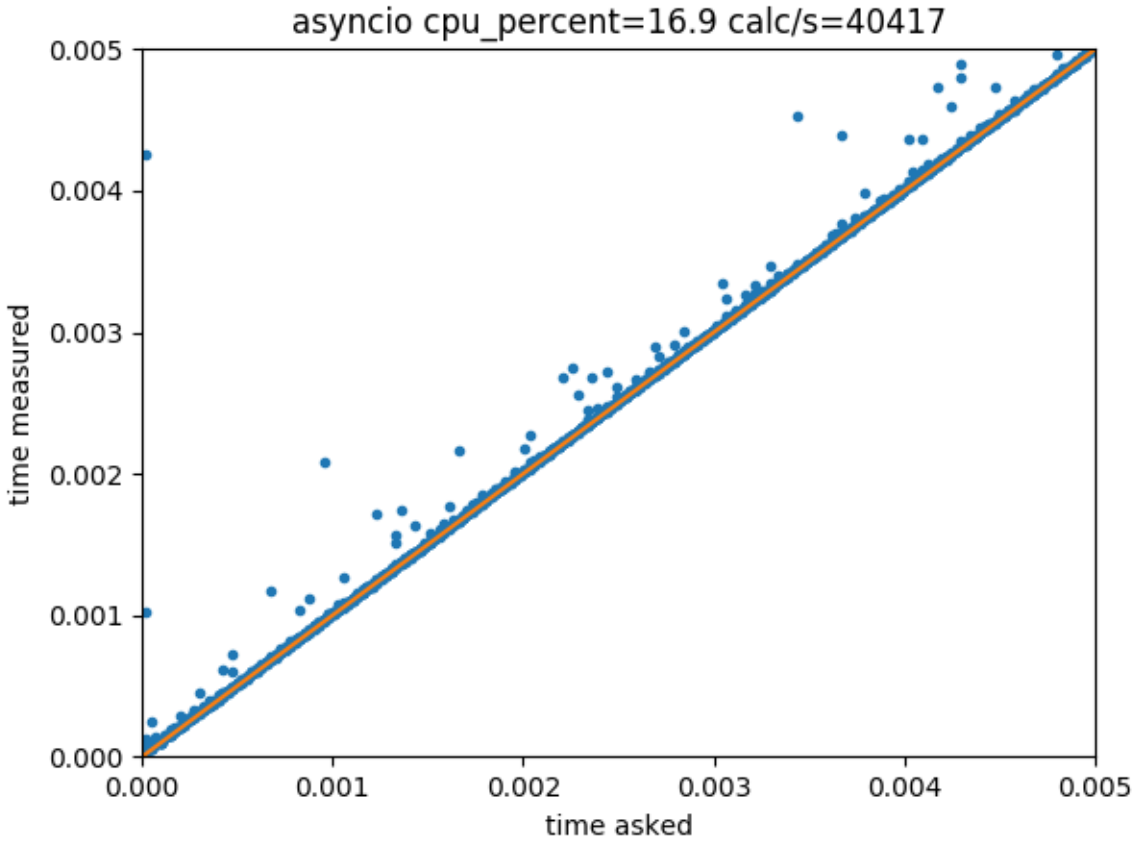
```
async def sleep_coroutine(delay):
   await asyncio.sleep(delay)
```

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()
```



4.5.3 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.
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".

Normalization of windowing functions

The Fourier transform of the series a_n is defined by

A_m = 1/N sum(a_k exp(-2 i pi m k/N)) [1]

With this convention, we need to pay attention that the DC-component is for m=0, and the "negative frequencies" are actually located in the second half of the interval [N/2, N]. Indeed, we can show that because of the discretization, A_{N - m} = A_{-m}

We can also show that the Fourier transform of the product of time series a_n and windowing function f_n, is the convolution of their Fourier Transform:

FT(a_n f_n)_m = A_m * F_m [2]

Let's first consider the case of a pure sinusoid a_n = cos(2 pi l n/N). The Fourier transform is A_m = (delta(l-m) + delta(l+m))/2. Hence, the Fourier transform is given by

FT(a_n f_n) = (F_(N-l) + F(l))/2 [3]

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.

Normalization for coherent signals

If we want the maximum value of the peaks to correspond to the amplitude of the sine (that is to say, a correct normalization in terms of Vpk), we need to make sure that the peak value of F_m is 2. Since the maximum of the windowing function is the DC-component F_0, we need to take:

F_0 = 2 [4]

or equivalently

1/N sum(f_n) = 2 [5]

To be complete, if the sinusoid has a frequency close to 0, the two terms in [3] take significant values for the same values of frequency, and they will interfere with each other (since they are complex numbers). A consequence of that is that the phase of the signal, which reflects into the complex arguments of F_(N-l) and F(l) starts to play a significant role, and thus, the spectrum is not stationary anymore, but oscillates in time with the frequency of the signal. The oscillations are most visible on the DC bin, where they oscillate between 0 and F(0), with the dependence:

|FT[a_n f_n]|^2 = 4 sin^2(2 pi l/N) [6]

but the oscillation is also present, with a reduced contrast on the neighboring frequency bins. There is not much we can do about it, except maybe to correct to make sure the average value of the oscillations is 1 (instead of 2 in formula [6], due to the fact that negative and positive frequency components both contribute to the averaged spectrum in this regime).

Normalization for noise spectral density:

On the other hand, let's consider a white Gaussian noise with variance 1. From Wiener Khintchine theorem, it should correspond to a flat spectrum of value 2 pi. From equation [2], since the variance of A_m fulfills Wiener Khintchine theorem, we deduce that the windowing function should fulfill:

sum(F_m^2) = 1 [7]

Using Parseval's theorem, this is equivalent to

sum(f_n^2) = 1 [8]

By comparing eq [8] and eq [5], we arrive at the interesting conclusion that the windowing function should be normalized with a linear summation for coherent signal measurements in Vpk^2 and with a quadratic summation for power spectral densities in Vpk^2/Hz. This is the time-domain counterpart of the fact that coherent signals are only sensitive to a single value of F_m, while noise spectra are integrated over the whole spectrum of the filtering window (eq [4] and [7]).

In pyrpl, we actually decided to make sure both conditions are fulfilled simultaneously by defining the rbw of a given filtering window to be:

rbw = sum(f_n^2)/sum(f_n) [9]

With this choice, the correct results are retrieved if we make all calculations in Vpk^2, and divide the results by the rbw to convert them in Vpk^2/Hz.

For this reason, the rbw is not exactly the width at 3 dB of the filter spectrum, but actually depends on the precise shape of the window over the whole frequency range via eq [9].

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. 4.1: 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.

4.5.4 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)

4.5.5 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.

4.5.6 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

4.5.7 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...

4.5.8 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?

4.5.9 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...

CHAPTER 5

Indices and tables

- genindex
- modindex
- search