1 Connection with the instrument

The purpose of interfacing an instrument is to send instructions from the computer so that it can perform operations automatically.

1.1 Connection

There are several types of connections between a computer and an instrument. These include GPIB, RS232, USB and ethernet.

GPIB and RS232 are communication interfaces that are obsolete, but are still frequently found on older devices. These are simple interfaces, since byte words are exchanged between the two devices. The RS232 is the serial port (COM under windows).

USB is much more complex because it does not precisely define what can be exchanged. For USB, it is necessary to have a driver for each communication mode. As a communication mode, let's mention the emulation of a serial port (this is for example what we find on arduinos). For measurement instruments, a standard called USBTMC is used by several manufacturers (USB Test and Measurement Class). However, some manufacturers may have developed their own USB interface.

Finally, more and more instruments are connected via ethernet. Things are generally much simpler than USB because there is no need for a specific driver. Communication standards exist for the instruments, in particular VXI-11.

The VXI-11 and USBTMC are intended to replace the GPIB. These protocols allow instructions to be exchanged according to a specific standard, the SCPI.

1.2 SCPI

SCPI (Standard Commands for Programmable Instruments) is the grammar that will be used. For example, to change the vertical scale of an oscilloscope you can send the string: "CH1:SCALE 10". To know the vertical scale we will send "CH1:SCALE?". The instrument will then return a string corresponding to the scale. In this example, the precise words depend on the brand of the instrument. What the SCPI defines is the syntax and some basic instructions.

The instructions are grouped in the form of a tree. Each "branch" is separated by a":". The arguments are separated from the instruction by a space and are separated from each other by commas. It is also possible to transfer binary data. When the instruction is waiting for an answer, then it is followed by a"?".

To know all the commands, refer to the part of the instrument documentation often called Programmers' User Guide.

The SCPI defines the grammar, but each manufacturer will define its own functions.

The basic and standards instructions of an instrument are among others:

- `*IDN?` which returns a unique identification string from the device
- `*RST` which makes a reset of the device.

One last important point: the SCPI is not case sensitive. In addition, each instruction has a short version and a long version. These are generally distinguished by case. For example "CHannel1:SCAle" means that CH or CHANNEL and SCA or SCALE can be used independently.

1.3 VISA

Under Windows, it is possible to use a VISA driver that provides a common interface, regardless of the connection mode or the low level protocol of communication with the instrument. The VISA library is a proprietary library provided by National Instruments. It can be used under Python with the pyvisa module (which must be installed in addition to VISA).

The NI-MAX software is installed at the same time as VISA. It allows, among other things, to list all the devices connected to the computer and to send instructions directly to the instrument.

1.4 VXI-11

Under linux, one can use VXI-11 instrument with the python-vxi11 library. To install it run the following command

```
pip install --user python-vxi11
```

2 Writing a python driver

With SCPI, only strings are exchanged, this is not convenient to use directly as we have then to convert arguments (like number) to string and convert back data from the scope to number. The goal of this part is to simplify this procedure using functions.

2.1 First step

1. Connect to the instrument using the vxi11 driver

```
import vxi11
instr =  vxi11.Instrument("134.157.xx.xx")
```

The string is the IP address of the device.

The vxi11.Instrument class provides methods to interact with the scope. See https://www.systutorials.com/docs/linux/man/1-pythonvxi11/ . We will mainly use `write` and `ask`.

Get the IDN string using

```
instr.ask('*IDN?')
```

Get the vertical scale of channel 1 using the following and then convert it to a number

```
instr.ask('CH1:SCA?')
```

Set the vertical scale of channel 1 to the value you wish using:

```
instr.write('CH1:SCA xx')
```

2. The class we will write (TektronixScope) should work as follow

```
instr = vxi11.Instrument("134.157.xx.xx")
scope = TektronixScope(instr)

scope.get_idn()
# Returns a tuple ("TEKTRONIX","DPO3014","C012048","CF:91.1CT FV:v2.16")
scope.get_channel_scale(1) # Returns a number
scope.set_channel_scale(1, xx)
```

Write this class. Actually you can simplify the code by writing generic methods like

```
    def scpi_ask(self, cmd):
        cmd = cmd if cmd.endswith('?') else cmd + '?'
        return self._instr.ask(cmd)

    def scpi_write(self, cmd, *args):
        """ Execute a SCPI command

for example :
    inst.scpi_write('FREQ', 1234)
        """
        pass

    def scpi_ask_for_float(self, cmd):
        """Ask an convert to float"""
        pass
```

Those methods will be part of a generic SCPI class, from which TektronixScope will heritate. Complete the SCPI class, an rewrite the TektronixScope class.

3. Let us write a fake SCPI instrument to test our classes. This instrument will mimic the scope by storing parameters in a dictionary:

```
class FakeSCPI(object):
    _record = {'*IDN':"TEKTRONIX,DPO3014,C012048,CF:91.1CT FV:v2.16 "}
    def write(self, val):
        if ' ' in val:
            cmd, vals = val.split(' ')
            self._record[cmd] = vals

    def ask(self, val):
        assert val[-1]=='?'
        out = self._record.get(val[:-1], '')
        return out
```

4. Optional : using properties. We want to use property to easily interact with the scope. For example we want to be able to use the following

```
scope.idn
scope.channel1.scale = 0.2
```

For the idn this is easy. For the channel it is more challenging. Basically, channel1 is a property that will return a new object with a property called scale that will call the `set_channel_scale` on scope. The intermediate object should therefore know who is the scope (the parent) and which channel will be written. Write such a class (with the following `def __init__(self, parent, index)`). Then write the channel1 property and then the scale property with a setter.

5. Now you can implement all the functions you want. Just look to the manual !

6. For the scope, there is one last, and not easy, step: retrieving the waveform. There are several way to exchange array with the scope with SCPI, either using ASCII string (for example : 10,14,16,24) or binary. Binary is faster, and we will use it. Use `vxi11.Instrument.ask_raw` to get binary data. The bytes string will look like this `#nmmmmxxxxxxxxxx`, where n is the number of m in ascii, `mmmm` is the size of the array (in ascii), and `xxxxxx` is the binary data. The previous example will be `'#40004\n\x0e\x10\x18'` in int8 (one byte per number)

To convert bytes string to numpy array use `np.frombuffer`. The data type will be

```
np.dtype('int16').newbyteorder('>')
```

Run the command `instr.ask_raw('CURVE?')`, try to decode the output. Then write a method.

The output is integer. We need to convert is to float. Look at the WFMO commands (p. 2-51) to get the vertical offset and scale to make the data correct. You can also find the horizontal offset and scale to get the x axis.

Make all of these a method to `TektronixScope`.

3 Graphical User Interface

We are using PyQT. A convenient way to install it consists in installing pyqtgraph:

```
pip install --user pyqtgraph
```

3.1 Threads

Threads allow you to execute several functions at the same time. They are fundamental with GUI (for example you want to interact with buttons even if a function is running in the background).

Bellow is a simple example of a thread

```
import threading
from time import time, sleep

class MyThread(threading.Thread):
    def __init__(self, parameter):
        threading.Thread.__init__(self)
        self.parameter = parameter

    def run(self):
        for i in range(10):
            print(self.parameter, i)
            sleep(.3)


thread = MyThread('Hello')
thread.start()
sleep(2)
print('Bonjour !!!')
thread.join()
```

1/ Modify this thread, so that parameter is a callback function that will be called in the loop. Try with

```
def callback(i):
    print('Hello', i)
```

2/ We want to run it in an infinite loop, but we have to find a way to terminate the loop. One way consists in using an attribute (`want_to_terminate`), set to False and then use a `while not self.want_to_terminate:` loop. To terminate the loop we simply have to set the attribute to True. Implement this method. Replace the index `i` by the time elapsed since the start of the thread.

3.2 Stopwatch

The stopwatch will be implemented during the lecture

3.3 Display Scope Output

1/ Starting from the stopwatch application. Replace the label with a MatplotlibWidget. In the callback method, get data from the scope and plot them. Here are some code to plot a graph inside a MatplotlibWidget

```
from pyqtgraph.widgets.MatplotlibWidget import MatplotlibWidget
plot_widget = MatplotlibWidget()

fig = plot_widget.getFigure()
fig.clf()
x = np.linspace(0, 1, 1001)
y = np.sin(2*np.pi*x)
ax = fig.subplots()
fig.canvas.draw()
```

2/ Add a save button to the scope, that will open a `QtGui.QFileDialog.getSaveFileName` dialog box. Ideally, the default directory should be something like `data/yyyy/mm/dd` where yyyy, mm and dd are the year number, month and day ! Save the waveform as txt file (np.savetxt).