Phase Locked Loops Automation Guide

Stasiu Wolanski, Jesus College

June 2022

1 Introduction

To gather the data for this experimental project it is intended that you will make use of automated data collection. This should be an exciting prospect: in contrast to the process of tweaking some parameter, taking a measurement, noting a value down in an excel spreadsheet and repeating ad nauseam (of which you grew so fond in parts 1A and 1B), here you will program a computer to do all such tweaking and recording rapidly and autonomously whilst you watch. As well as being deeply satisfying to one's inner work-shy self, such methods have the advantage of allowing you to acquire much larger data-sets than are feasible manually. You will then analyse these data-sets to gain physical insights into the systems you are studying.

The tools and techniques that are described here are intended to provide an accessible introduction to programming hardware for the purpose of physical measurement. Work has been done to spare you the gory details of the underlying communication protocols. These methods are sufficient to perform all the experiments required to complete the project to a high standard. If you have a programming background, or are just up for the challenge, you are encouraged to tinker with, hack, and extend the provided tools - or cast them aside entirely and do things your own way - if you can improve the experiment by doing so. With that, let us describe the provided tools.

2 Overview of the Tools and Automation Scheme

You have access to a Picoscope 2000 series oscilloscope and an Arduino microcontroller as weapons of mass data collection.

The Arduino is a microcontroller (read 'simple, small computer') capable of receiving and producing digital signals. You will program the Arduino by uploading programs ('sketches') which you will write in the a variant of the C programming language. You will communicate instructions during the course of the experiment from your computer to the Arduino using a python interface. The Picoscope, a device you know and love from parts 1A and 1B, is a device that can record the voltage of a signal as a function of time. You will control and record data from the Picoscope using another python interface.

The overall scheme suggested here, therefore, is as follows. You will program the Arduino to produce a variety of input signals to the circuit being measured. This is the independent variable - for example this could be the frequency of an input signal. A python script which you write will then direct the Arduino to sweep through all of these input signals in succession, and at each point use the Picoscope to record some feature of the output signal (the dependent variable). Alternatively, you may record entire blocks of voltage data directly to the hard drive and do the data analysis afterwards.

In order to facilitate the above, a python library `PLL_Lib` is provided that provides convenient wrapper methods for connecting to and communicating with both the Arduino and the Picoscope.

3 Programming the Arduino

Refer to the Class Manual for an introduction to programming the Arduino.

The existing section on the Arduino in the Manual can be left mostly unchanged. I would suggest however that the section on serial communication be pared down to just receiving and not transmitting serial data, and providing more explicit examples of changing the functionality of the Arduino in response to received data.

4 Controlling the Arduino during the experiment

4.1 Python Code

As you have seen, we can control the behaviour of the Arduino by sending it data over serial, which up until now you have typed into the serial monitor of the Arduino software. In order to automate out experiments, we need to be able to send instructions automatically. In order to do this, you can use the `Arduino` class provided in `PLL_Lib`. For maximum reliability and simplicity, the library restricts you to sending integer numeric codes over serial. These are 32-bit and signed so can send any integer between $-2^{31}+1 = -2147483647$ and $2^{31}-1 = 2147483647$ inclusive.

```
from PLL_Lib import Arduino
import time

# Connect to the Arduino
with Arduino() as arduino:
    # You can refer to 'arduino' anywhere inside this indented block.
    for i in range(10, 60, 5):
        print(f'Sending code {i}...')
        arduino.send_code(i)
        time.sleep(1)
# The arduino automatically disconnects at the end of this indented block.
print('Done!')
```

If you have not come accross python f-strings before (`f'blah{var}blah...'`) then you should google them.

4.2 Arduino Code

To receive the signal we can use the `Serial.parseInt()` function described earlier, with the caveat that this function will stop short of the `newline` character that is transmitted after the integer to signal that the full number has been transmitted. We therefore need to read this single character, which is encoded as a single byte, from the stream, even though we don't care about its value (it is normally written '\n' - note this represents a single character). Thus we can write:

```
int half_period = 100;

void setup() {
  Serial.begin(9600);
  pinMode(9, OUTPUT);
}

void loop() {
  if (Serial.available() > 0) {
    // This is the line of code required to receive an integer from the python script.
    long incomingCode = Serial.parseInt(); Serial.read();
    half_period = incomingCode;
  }
  // Basic implementation of a square wave oscillator.
  digitalWrite(9,HIGH);
  delayMicroseconds(half_period);
  digitalWrite(9,LOW);
  delayMicroseconds(half_period);
}
```

Upload the above sketch to the Arduino. Then *close the Arduino IDE* so the python script can access the serial port, and run it, whilst observing pin 9 of the Arduino using the Picoscope software. Check it behaves as you expect.

5 Using the Picoscope with Python

Now setup the Arduino to output a simple oscillating waveform at around 1kHz (without need of external control) and check that it is working using the Picoscope software. Now close the Picoscope software - we are going to write

our own!

```
from PLL_Lib import Picoscope

# Connect to the Picoscope
with Picoscope() as scope:
    # Keep capturing traces until the a key is pressed
    scope.wait_for_key('a')
# The Picoscope will automatically be disconnected at the end of the indented block.
```

5.1 Picoscope Arguments

An interface will appear, displaying traces as fast as it can display them. There are many ways we may wish to specify the behaviour of the scope: we would like to have a trigger, so that the traces are more consistent from one to the next; we may wish to change the timebase to show more or fewer oscillations on the screen; we may wish to introduce compensation for the fact we might be using a probe with a 10× voltage reduction. We do this by supplying optional arguments to the Picoscope at the point at which we connect to it[1]. As an example:

```
from PLL_Lib import Picoscope

# Python allows a certain freedom of formatting
# when it comes to listing arguments, which can be nicer to read
with Picoscope(
        time_per_sample='10micro_s',
        probe_10x=True,
        trigger_channel='a'
        ) as scope:
    times, voltages_a, voltages_b = scope.wait_for_key('a')

print(f'The maximum voltage on channel A was {max(voltages_a)}V.')
```

Here we demonstrate another feature of `wait_for_key()` - it *returns* the times (in seconds) and the voltages (in volts) of the samples of the last trace captured when the relevant key was pressed. They are returned as `numpy` arrays[2]. Another thing to note is that as we have now set a trigger, the picoscope will not capture anything until a trigger event occurs. This may cause your program to appear stuck if you set up the trigger incorrectly or pass in the wrong (or no) signal. If this occurs it may be wise to disable the trigger to check the signal is as you expect. Bear in mind there needs to be a certain amplitude of variation in the signal around the trigger voltage before the trigger triggers. The full list of optional arguments is:

- `time_per_sample`: The time per sample, or temporal resolution, given as a string. Options are `10ns, 20ns, 40ns, 80ns, 160ns, 320ns, 640ns, 1micro_s, 3micro_s, 5micro_s (Default), 10micro_s, 20micro_s, 41micro_s 82micro_s, 164micro_s, 328micro_s, 655micro_s, 1ms, 3ms, 5ms, 10ms`.
- `probe_10x` If `True`, apply a 10x multiplier to the output voltages in the display window and output arrays. Does not affect the input voltage range or trigger voltage, which should be set as if this is not enabled. Default is False.
- `voltage_range`: The voltage range, given as a string representing the maximum positive and minimum negative voltage that can be measured. This should be set taking into account any voltage reduction due to the probe, regardless of whether the `probe_10x` option is active. Note that if this is exceeded, a 'Channel Overrange' warning will appear. Options are `20mv, 50mv, 100mv, 1v (Default), 2v, 5v, 10v, 20v`.
- `trigger_channel`: `None` (Default) for no trigger, or `'a'` or `'b'` to trigger using that channel. All other trigger options are ignored if this is set to `None`.
- `trigger_voltage`: The voltage threshold of the edge detection for the trigger. Should be a number within the specified voltage range. This should be set taking into account any voltage reduction due to the probe, regardless of whether the `probe_10x` option is active. By default this is a quarter of the (positive) voltage range.

[1] `PLL_Lib` does not provide a direct way to change the values of such parameters during the course of the experiment, for the sake of simplicity. The underlying C API does, however, so if you are keen you will be able to achieve this with lower-level calls.

[2] for a refresher on numpy arrays, google 'numpy quickstart'.

- `rising_edge`: `True` (Default) for rising edge triggering. `False` for falling edge triggering.
- `trigger_offset`: The percentage of samples which are recorded before the trigger event. This should be an integer between 0 and 100. Default is 10.
- `show_display`: Whether to display the traces in a window. Required in order to use `wait_for_key`. Default is `True`.

Play around with the `trigger_offset` parameter. What does it do?

5.2 Capturing Data

The `wait_for_key()` method will repeatedly capture and display traces (a set of voltage samples, usually around 8000 for each channel) until the given key is pressed, whereupon it will stop and return the next trace recorded. To capture and display a single trace we instead use `get_trace()` method, which captures a single trace and returns it.

We also need a way of storing the data from the trace. There are many ways of doing this, of which I will introduce one: we can save the entirety of the trace to the hard drive using `numpy.save`.

```
from PLL_Lib import Picoscope
# It is conventional to import numpy with the abbreviated alias np
import numpy as np

# The number of traces to save
N = 100

with Picoscope(time_per_sample='1micro_s', probe_10x=True, trigger_channel='a') as scope:
    # We can use the initial trace to judge the proper shape of the arrays
    times, voltages_template, _ = scope.wait_for_key('s', 'Press to start experiment')
    # Create an empty 2D Array for the traces
    voltages_array_a = np.zeros((N,voltages_template.size))
    # Create another
    voltages_array_b = np.zeros_like(voltages_array_a)
    for i in range(N):
        # All captures will have the same set of sample times, so can ignore this
        # get_trace takes as an optional argument a message that is displayed in the bottom-left
        _, voltages_a, voltages_b = scope.get_trace(f'Capturing trace {i}...')
        # Store them into the 2D array
        voltages_array_a[i] = voltages_a
        voltages_array_b[i] = voltages_b

# Save to memory - be sure to change these names if you run an experiment twice
# or it will overwrite the existing files with no warning!
np.save('Times.npy',times)
np.save('Voltages_A.npy',voltages_array_a)
np.save('Voltages_B.npy',voltages_array_b)

print('Done!')
```

It is important to consider how much memory you have on your computer when implementing an approach like this. Having saved the data we can analyse it at our leisure. For example, to plot the traces:

```
import numpy as np
from matplotlib import pyplot as plt

# Retrieve the data from the hard drive
times = np.load('Times.npy')
voltages_array = np.load('Voltages_A.npy')

# Plot just the first trace
plt.plot(times, voltages_array[0])
# Take the point-wise mean of the traces
plt.plot(times, voltages_array.mean(axis = 0))

plt.xlabel('Times/s'), plt.ylabel('A Voltage/V')
plt.show()
```

What do the plots show, and why? For some experiments it may not be optimal to save the entirety of the data. In these cases you may instead wish to write data to a python dictionary and save it to a `.json` file using `json.dump` (google it!).

6 Putting it all Together

Upload the code from section 4.2 so that the half-period of the Arduino's oscillation can be controlled over serial. Then run the following script (again, make sure to close the Arduino IDE):

```
from PLL_Lib import Arduino, Picoscope

half_period = 500
with Arduino() as arduino:
    with Picoscope(time_per_sample='1micro_s', trigger_channel='a') as scope:
        scope.wait_for_key('s', 'Press to start experiment.')
        while True:
            arduino.send_code(half_period)
            scope.wait_for_key('n', 'Next frequency?')
            half_period += 5
```

Check that you understand everything that is going on - you should be able to manually step through the frequencies on the Arduino. Now, using the above script as a starting point, write a python script that *automatically*, without stopping, steps through a range of half-periods (say 500, 510 ... 600) and records 50 traces at each. You should parameterise everything, i.e. write the program in such a way that you could change the requirement to 100 traces per setting, or just take half-periods up to 550 rather than 600, just by changing one number at the top of each script. Save all of these traces, and in a separate python file, analyse the data to produce a graph of half-period instruction to the actual frequency of the output. You will need to write or import code to find the frequencies from the traces. By analysing all 50 traces you can get an estimate of the uncertainty.

7 Controlling the Signal Generator

For some experiments you may find it helpful to use the Picoscope's signal generation functionality. It can produce a variety of pre-set waveforms[3] at frequencies up to 100kHz, with the limitation that it can produce minimum and maximum voltages of -2V and 2V. This means to create logic-level signals you will need to create some external amplification circuit (inverting op-amp?).

To use the signal generator, call the `set_signal_generator()` method on the `Picoscope` object. The parameters are listed below:

- `frequency`: (Non-optional) the frequency in Hz of the generated signal. A number between 0 and 100,000.

[3] In fact, it can produce an arbitrary waveform, as the name suggests. However, this functionality is complex and probably unnecessary. If you feel up to it, you could access this functionality by leveraging the underlying C API.

- `wavetype`: A string representing the kind of waveform to produce. Options are `'SINE'`, `'SQUARE' (Default)`, `'TRIANGLE'`, `'RAMP_UP'`, `'RAMP_DOWN'`, `'CONSTANT_VOLTAGE'`, `'GAUSSIAN'`, `'SINC'`, `'HALF_SINE'`.
- `min_voltage`: A number between -2 and 2 representing the minimum voltage in volts of the produced waveform. Must be $\leq$ `max_voltage`. Default is -2.
- `max_voltage`: A number between -2 and 2 representing the maximum voltage in volts of the produced waveform. Must be $\geq$ `max_voltage`. Default is 2.

Experiment with the different wavetypes to see what they look like.