Introduction

As you’ve seen in the previous labs and will see in future labs, many of these of the sensors we interact with in this class have plenty of tuning knobs. Many of them are digitally controllable, as we set a register value through a serial bus and let the chip handle the rest of it. Some, however, have bias currents or voltages that are required to get the sensor operating in the ideal mode for our use case. While turning dials or pressing buttons is ok for a quick testing environment, it would be pretty obnoxious to have to test a sensor from 2V to 5V in steps of 10mV. In this lab, we’ll deal with automating the bench instruments, so you can edit the hardware values as easily as the software ones.

Follow all the steps described in this document to learn how to control various instruments from Python. An additional tutorial on how to use Python to control instruments can be found here.

VISA Overview

The majority of the bench instruments we use communicate over the VISA specification: Virtual Instrument Software** Architecture interface. While this technically only describes the serial interface, many instruments make it simple to use VISA over USB, Ethernet, RS-232 or GPIB. Many companies provide the VISA drivers, and we’ll be using National Instrument’s (NI) in this class. Every instrument in the lab supports the VISA interface, and even better, we can use them all at once.

Since we can connect so many things at once, we (unfortunately?) have more terminology to learn. There’s no need to memorize this, but it will probably be helpful to know the terms when debugging:

- **VISA Resource**: Any resource (instrument) that is readable on the VISA bus
- **VISA Session**: An active communication channel to a VISA resource
- **Instrument Descriptor**: Protocol, device address, (usually encoded) device name
  - Example: USB0::0x0957::0x2707::MY52301256::INSTR
- **Query**: A message that we send to the device (read or write)

Traditionally, one might use LabVIEW or C++ to interface over VISA, but we’ll be sticking to Python thanks to the excellent PyVISA package. An example file is provided on the website, and we’ll annotate it here:

```python
import visa
rm = visa.ResourceManager()
reslist = rm.list_resources()
```
First we go through and list all the instruments (resources) that are available to us. In our case, we only have device attached. In this case, it's the waveform generator, although it's hard to tell from here.

```python
wavegen = rm.open_resource(reslist[0])
print(wavegen.query("*IDN?"))
```

Now we open a session with the first instrument in our list. Since we're not sure which one it is, we'll query it and ask for its identity (this is SCPI, described in the next section). Now we know exactly which instrument we're talking to.

**SCPI Overview**

Now here, we'll get slightly into the weeds. Technically, VISA refers to the I/O abstraction layer when it comes to test. We don't actually program the devices using VISA, we communicate over the VISA specification. When it comes to actually programming instruments, we use the **Standard Commands for Programmable Instruments** (SCPI). While it may sound complex, it's just a bunch of ASCII strings that allow us to configure the device. There are many commands - at least one for each setting on the instrument. The oscilloscope manual, for example, is 1400 pages of commands! Luckily, they're organized in a sensible way, and tend to be common among instruments.

The following section is copied from the Keysight manual, as it's probably the best overview of SCPI available. You can skip over this if you're comfortable debugging on your own, but it's a good reference:

SCPI (Standard Commands for Programmable Instruments) is an ASCII-based instrument command language designed for test and measurement instruments. SCPI commands are based on a hierarchical structure, also known as a tree system. In this system, associated commands are grouped together under a common node or root, thus forming subsystems. A portion of the OUTPut subsystem is shown below to illustrate the tree system.

**OUTPut:**

```plaintext
SYNC {OFF|0|ON|1}
SYNC:
    MODE {NORMAL|CARRIER}
    POLarity {NORMAL|INverted}
```

OUTPut is the root keyword, SYNC is a second-level keyword, and MODE and POLarity are third-level keywords. A colon (:) separates a command keyword from a lower-level keyword.
Syntax Conventions

The format used to show commands is illustrated below:

[SOURce[1|2]:]VOLTage:UNIT {VPP|VRMS|DBM}
[SOURce[1|2]:]FREQuency:CENTer {<frequency>|MINimum|MAXimum|DEFault}

The command syntax shows most commands (and some parameters) as a mixture of uppercase and lowercase letters. The upper-case letters indicate the abbreviated spelling for the command. For shorter program lines, you can send the abbreviated form. For better program readability, you can send the long form.

For example, in the above syntax statement, VOLT and VOLTAGE are both acceptable forms. You can use uppercase or lowercase letters. Therefore, VOLTAGE, volt, and Volt are all acceptable. Other forms, such as VOL and VOLTAGE, are not valid and will generate an error.

- A query that ends in a question mark (?) denotes a read command, else a write command is assumed.
- Braces ( {} ) enclose the parameter choices for a given command string. The braces are not sent with the command string.
- A vertical bar ( | ) separates multiple parameter choices for a given command string. For example, {VPP|VRMS|DBM} in the above command indicates that you can specify "VPP", "VRMS", or "DBM". The bar is not sent with the command string.
- Triangle brackets in the second example ( < > ) indicate that you must specify a value for the enclosed parameter. For example, the above syntax statement shows the parameter enclosed in triangle brackets. The brackets are not sent with the command string. You must specify a value for the parameter (for example "FREQ:CENT 1000") unless you select another option shown in the syntax (for example "FREQ:CENT MIN").
- Some syntax elements (for example nodes and parameters) are enclosed in square brackets ([ ]). This indicates that the element is optional and can be omitted. The brackets are not sent with the command string. If you do not specify a value for an optional parameter, the instrument chooses a default value. In the examples above the "SOURce[1|2]" indicates that you may refer to source channel 1 either by "SOURce", or by "SOURce1", or by "SOUR1" or by "SOUR". In addition, since the whole SOURce node is optional (in brackets) you also may refer to channel 1 by entirely leaving out the SOURce node. This is because Channel 1 is the default channel for the SOURce language node. On the other hand, to refer to Channel 2, you must use either "SOURce2" or "SOUR2" in your program lines.
Practical SCPI + Example

Now we'll get into a quick example of practical SCPI. The sensors PCB has a test diode and resistor connected in series and the test circuit is shown below. There are three terminals that allow us to measure voltages and currents from the test circuit and the connection ports are located in the upper left corner of the board. The color coding of the terminal pins matches the description on the schematic shown below.

IMPORTANT: Do not connect a voltage supply directly to the red and blue port. If you do so, you will most likely blow up the diode. You need to limit the current that flows through the diode and a resistor connected in series with the diode will do the job. You will need to connect a power supply between port red and black to get the diode to operate correctly.

Next, we will measure the current-voltage characteristic of the diode. As shown in the initial example, the most common command is *IDN?, which asks for the instrument name and details. This is common among effectively every device, so you should always start with this just to make sure we're not sending information to the wrong device.

The full example code for this is posted on the website under the resource website, and we'll annotate it here again. Make sure you download the complete and latest version of this code on the resource website.

```python
rm = visa.ResourceManager()
reslist = rm.list_resources()
dc_supply = rm.open_resource(reslist[0])
print(dc_supply.query("*IDN?"))
```

Same as before, we go through and open a session with the device. Here you should of course check whether or not you have the power supply connected before you continue, but I'm assuming it is the only thing connected for now.

To determine the correct port that your power supply is connected, open “Device Manager” in Control Panel in Windows. Under “Universal Serial Bus controller”, look up the port number that your USB controlled power supply is connected. The actual port number for
the power supply is computed by subtracting one from the port number listed in your Device Manager. For example, if Device Manager lists the serial port number as 5, in python you will initialize the instrument with port number 4.

\[
\text{volts_desired} = \text{np.arange}(0, 1.5, 0.05) \\
\text{volts_compliance} = \text{np.array}([]) \# \text{empty list to hold our values} \\
\text{currs} = \text{np.array}([]) \# \text{empty list to hold our values}
\]

We initialize our I and V variables. Our voltage array is all the values we intend to sweep over, here from 0 to 3V, in 10mV steps. Our current array is an empty array we'll populate with results.

You must enable the output of the power supply before you can start collecting the data. This is achieved by the following command:

\[
\text{print}(\text{dc_supply.write("OUTPUT ON")})
\]

Next we will automatically sweep the power supply and collect data for each voltage applied to the diode. We will read the current that is flowing through the diode by reading the current value that the power supply is providing to the circuit.

\[
\text{for v in volts_desired:} \\
\quad \# \text{write voltage and current limit to device} \\
\quad \text{dc_supply.write("APPLY P6V, %0.2f, 0.5" % v)} \\
\quad \# \text{pause 50ms to let things settle} \\
\quad \text{time.sleep(0.05)} \\
\quad \# \text{read current voltage value} \\
\quad \text{val_volts_compliance = dc_supply.query("MEAS:VOLTage:DC? P6V")} \\
\quad \text{volts_compliance = np.append(volts_compliance, val_volts_compliance)}
\]

\[
\text{# read current current value [heh]} \\
\text{val_currs = dc_supply.query("MEAS:CURRent:DC? P6V")} \\
\text{currs = np.append(currs, val_currs)}
\]

Now we iterate over every setting in the voltage array. We use the SCPI APPLY command here to set the output operating mode, specifically: - Talk to the P6V (+6 volt) source - Write v volts and limit the current to 500mA.

We pause 50ms each time just to let things settle. We then read the current value and add it to our array.

Before we proceed with plotting the results, don’t forget to turn off the power supply. If you don’t execute this instruction, you are running at a risk of damaging the diode. We also need to close the device so that we can open it again in the future without any errors. Here is the instruction to turn off the power supply and release the resources:
# disable output
print(dc_supply.write("OUTPUT OFF"))

# close device
rm.close()

Last we plot the results on the screen

# plot results (desired values)
plt.figure()
plt.plot(volts_desired, currs)
plt.title("Volts (Desired) vs. Current (Measured) for Diode")
plt.xlabel("Volts (Desired) [V]")
plt.ylabel("Current (Measured) [A]")
plt.draw()

# plot results (compliance values)
plt.figure()
plt.plot(volts_compliance, currs)
plt.title("Volts (Compliance) vs. Current (Measured) for Diode")
plt.xlabel("Volts (Compliance) [V]")
plt.ylabel("Current (Measured) [A]")
plt.draw()

# show all plots
plt.show()

Lastly we plot our results. The whole script should take approximately 15s to run, or significantly faster than any manual analysis would.

Save the I-V plot and included it in your report.

**Troubleshooting common problems**

1. If an error occurs during the communication with an instrument, you will observe typically two things: the instrument will beep and will display an error message on the instrument display panel. Once an error occurs, subsequent instructions to the instrument are typically not executed properly. You will need to clear the errors before proceeding. For example, if you want to clear the errors in the dc power supply, then execute the following instruction:

   dc_supply.write("*CLS")

2. The power supply uses RS 232 communication protocol to talk to the PC. Hence, the power supply needs to be set to use this protocol for communication. You can press the "I/O config" button on the instrument panel and choose RS232 using the rotating knob. Once you adjust the communication protocol, the instrument saves this setting and will use it next time you power up the instrument.
Checkpoint 1 (30 pts)

Write a Python code that allows you to cycle through the different instruments connected to the PC. Identify how many instruments are connected to the PC. List the instruments and the corresponding port number. Make sure that all instruments are turned on before proceeding with the lab.

Q1-a Demonstrate that your code is working to the TA.
Q1-b How many instruments are connected to the PC and what are these devices?
Q1-c What is the port number for each of the instruments connected to the PC?
Q1-d What is the maximum number of instruments that can be connected to the PC via the VISA protocol?
Q1-e Print both your Python code and the output results from running this code and include them in your lab report.

Checkpoint 2 (30 pts)

Plot the current-voltage characteristic of the diode. Change the python code such that the voltage range is swept from 0V to 6V in increment of 100mV. Save the I-V plot for this circuit.

Q2-a Demonstrate that your code is working to the TA.
Q2-b Change the increment to 0.8V and comment on the difference between the results obtained in question Q2-a.
Q2-c What would be a reasonable step function when you are plotting the I-V characteristics of a device and how would you determine it? What are some of the tradeoffs when you are determining the increment size?
Q2-d Print both your Python code and the output results from running this code and include them in your lab report.
Q2-e Plot the current-voltage characteristic of the diode and include it in the final report.

Checkpoint 3 (40 pts)

Next, we will use both the multimeter and the oscilloscope. Connect the multimeter in series with the power supply so that you can accurately measure the current consumption of the circuit. Next connect the oscilloscope probe on one of the diode terminals so that you can measure the voltage across the diode (i.e. blue test port).

You will need to sweep the voltage applied across the circuit and for each step measure the voltage across the diode and the current. The oscilloscope will capture the voltage value and multimeter will capture the current. You will need to figure out the correct instruction
that will allow you to read data from each of the instruments. Use the online manual for both instruments which you can locate on the course website reference page.

Sweep the voltage applied across the circuit between 0V and 6V in steps of 100mV. Plot the current-voltage properties of the diode.

Q3-a Demonstrate that your code is working to the TA.

Q3-b Print both your Python code and the output results from running this code and include them in your lab report. Comment on the results you have obtained from this checkpoint.