ECE437: Sensors and Instrumentation

Lab 4 – Controlling Instruments from Python

Introduction

One of the key requirements in many sensory applications is acquiring electrical or optical characterization of the sensory board. For example, we might be interested in the noise performance of the temperature sensor as a function of different ambient temperatures. To measure the noise performance of a sensor, typically more than one hundred readings at a given operating points are necessary. If you had to evaluate the noise performance of the temperature sensor over 100 different temperature points, you will have to log 10,000 readings from the sensor. This is a time consuming and error prone task if it is done manually.

Luckily, the instruments in the lab are interfaced with a PC and can send and receive data with the PC. Many different programming languages, such as Matlab, Python and C, have software APIs that enables easy interface with instruments. In this lab, we will learn how to write Python code that will enable you to collect data from different instruments available in the lab. You will use these skills in later labs, where you will be collecting large data sets from various sensors using computer-controlled instruments.

The lab includes a tutorial on how to use Python to control instruments in the lab. You should follow all the steps outlined in this tutorial. At the end of the tutorial, you will demonstrate a working Python code that can communicate with different instrument and acquire current-voltage data from a test diode.

Relevant Documents for this Lab

Required reading material for this lab:

1. Python example code on the course website: *PowerSupplyControl.py.*

Additional reference material:

1. Reference manuals for the power supply, multimeter and oscilloscope instruments in the lab.
2. Online reference for Python VISA commands.

The goals of this lab are:

1. Learn how to communicate with the different instruments using Python programming language.
2. Generate current-voltage characteristics of a test diode using computer controller instruments from Python.

Prelab Questions (20 pts)

1. What is the most common interface for communication with instruments from the PC?
2. What is the maximum number of instruments that can be connected to the PC via the VISA protocol?
3. What is the SCPI command that will turn ON the outputs of the power supply in the lab?
4. What is the SCPI command that will turn OFF the outputs of the power supply in the lab?

VISA Overview

Most benchtop instruments that are commonly used in an optics or electronics lab communicate over the VISA specification: **V**irtual **I**nstrument **S**oftware** **A**rchitecture 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. All instruments in the lab supports the VISA interface over USB, which makes the instrument-PC communication very easy and straight forward.

First, we will introduce some useful terminology for VISA. 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.

SCPI Overview

Technically, VISA refers to the I/O abstraction layer when it comes to test. We don't program the devices using VISA, we communicate over the VISA specification. When it comes to programming instruments, we use the **S**tandard **C**ommands for **P**rogrammable **I**nstruments (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 many instructions 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 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, thus forming subsystems. A portion of the OUTPut subsystem is shown below to illustrate the tree system.

```
OUTPut:
        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 VOLTAG, are not valid and will generate an error.

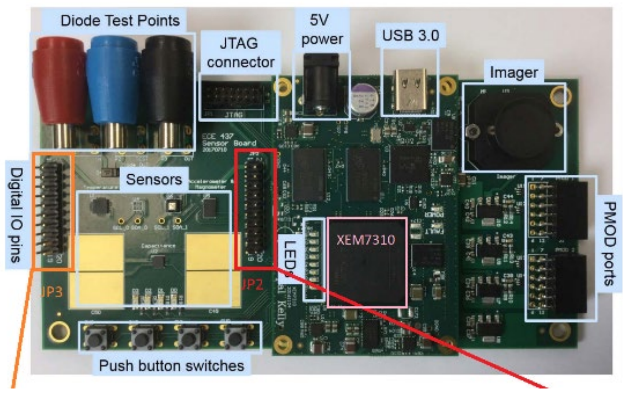
- A query that ends in a question mark (?) denotes a read command, else a write command is assumed.
- Braces ({ }) enclose the parameter choices for a given command string. The braces are not sent with the command string.
- A vertical bar (|) separates multiple parameter choices for a given command string. 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.

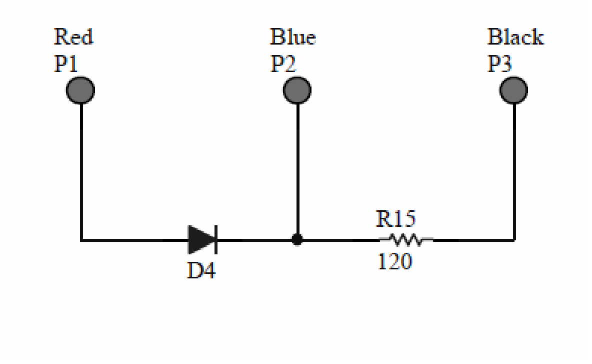
Example Python Code for Instrument Communication

We have written an example code that provides a framework for how to communicate with the lab instruments from Python. The goal of the example code is to generate a current-voltage characteristic for the test diode on the sensor's PCB. You will need to download the example code from the course website before proceeding with the tutorial. Open the code in Spyder and scroll through the code. The code will establish communication with the power supply, sweep different voltages on the 6V power supply and measure the supplied current from this power supply to the diode under test.

The sensor's 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. The connection ports are in the upper left corner on the board. The color coding of the terminal pins matches the description on the schematic shown below.

IMPORTANT: Do not connect a power 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 the 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 the red and black ports to get the diode to operate correctly. **Failure to do so will result in 50% less points on this lab exercise.**

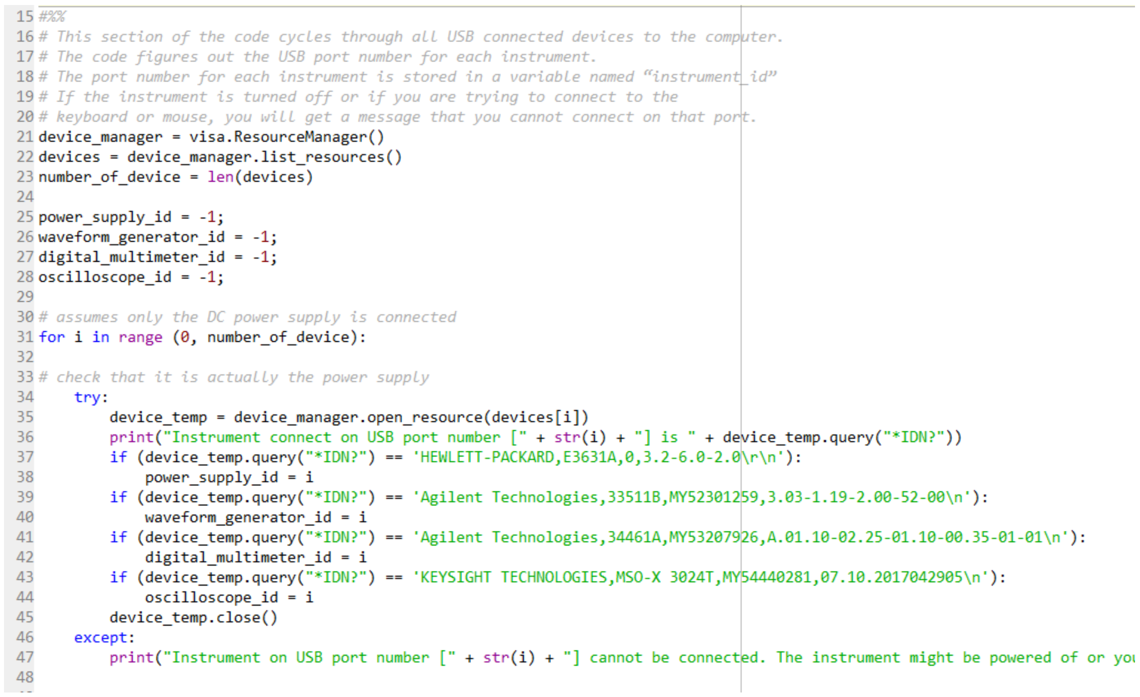




The example code (*PowerSupplyControl.py*) starts with a note on what to do if there is an instrument error when you executed your Python code. For example, if you send a command to the instrument and the instrument cannot correctly interpret your command, the instrument will beep and send an error message. At this point, you cannot send any more commands to the instrument. This will happen more often than you think especially as you debug your code and figure out the commands for the different instrument. You will need to clear the error message and you can do so by sending the *power_supply.write("*CLS")* SCPI instruction.

First, we need to figure out the USB port number for each instrument that is connected to the PC. This is established with the code presented in the figure bellow and it is the code situated between lines 15 and 48 of *PowerSupplyControl.py* file. Once an instrument is physically connected to the PC, it will communicate over a specific USB port number. This will be different for each computer as different instruments are connected to different USB ports. At the same time, there are other USB devices connected to the PC, such as the mouse, keyboard and our sensor's board. We will need to first figure out how many USB ports are there on the PC and cycle through all of them. For each USB port, we will try to connect to the device and figure out which instrument is it. Once we know that information, we create an id number for each one of the instruments. If the instrument is turned off or if we are trying to connect to the mouse, keyboard or sensor board, we will get a warning message.

NOTE: the code bellow might not always correctly identify the ports for the different instruments. You will need to double check the instrument identification string and update it accordingly in the code bellow so that the instrument ports are identified correctly.



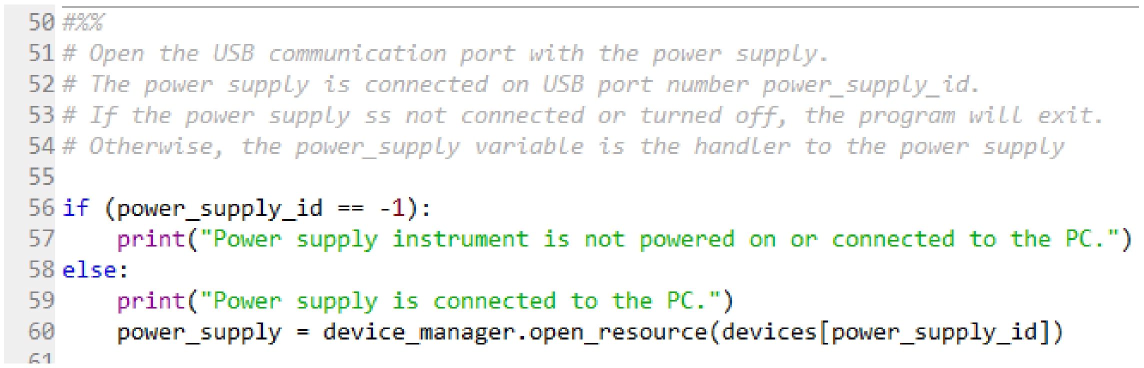
```
#%%
# This section of the code cycles through all USB connected devices to the computer.
# The code figures out the USB port number for each instrument.
# The port number for each instrument is stored in a variable named "instrument_id"
# If the instrument is turned off or if you are trying to connect to the
# keyboard or mouse, you will get a message that you cannot connect on that port.
device_manager = visa.ResourceManager()
devices = device_manager.list_resources()
number_of_device = len(devices)

power_supply_id = -1;
waveform_generator_id = -1;
digital_multimeter_id = -1;
oscilloscope_id = -1;

# assumes only the DC power supply is connected
for i in range (0, number_of_device):

# check that it is actually the power supply
    try:
        device_temp = device_manager.open_resource(devices[i])
        print("Instrument connect on USB port number [" + str(i) + "] is " + device_temp.query("*IDN?"))
        if (device_temp.query("*IDN?") == 'HEWLETT-PACKARD,E3631A,0,3.2-6.0-2.0\r\n'):
            power_supply_id = i
        if (device_temp.query("*IDN?") == 'Agilent Technologies,33511B,MY52301259,3.03-1.19-2.00-52-00\n'):
            waveform_generator_id = i
        if (device_temp.query("*IDN?") == 'Agilent Technologies,34461A,MY53207926,A.01.10-02.25-01.10-00.35-01-01\n'):
            digital_multimeter_id = i
        if (device_temp.query("*IDN?") == 'KEYSIGHT TECHNOLOGIES,MSO-X 3024T,MY54440281,07.10.2017042905\n'):
            oscilloscope_id = i
        device_temp.close()
    except:
        print("Instrument on USB port number [" + str(i) + "] cannot be connected. The instrument might be powered of or yo
```

Since we have figured out the id number for the power supply's USB port, we will use this info to connect to this instrument. It is always a good coding practice to check for errors when you try to connect to the various devices. Before we connect to the power supply, we check if our previous code found out the port number for the power supply. If the power supply is not turned on, then we will not connect to the instrument and the program will exit. If the power supply port number is identified (i.e. it is not equal to -1), we will connect to the power supply and establish USB connection. This is all achieved with the code bellow:



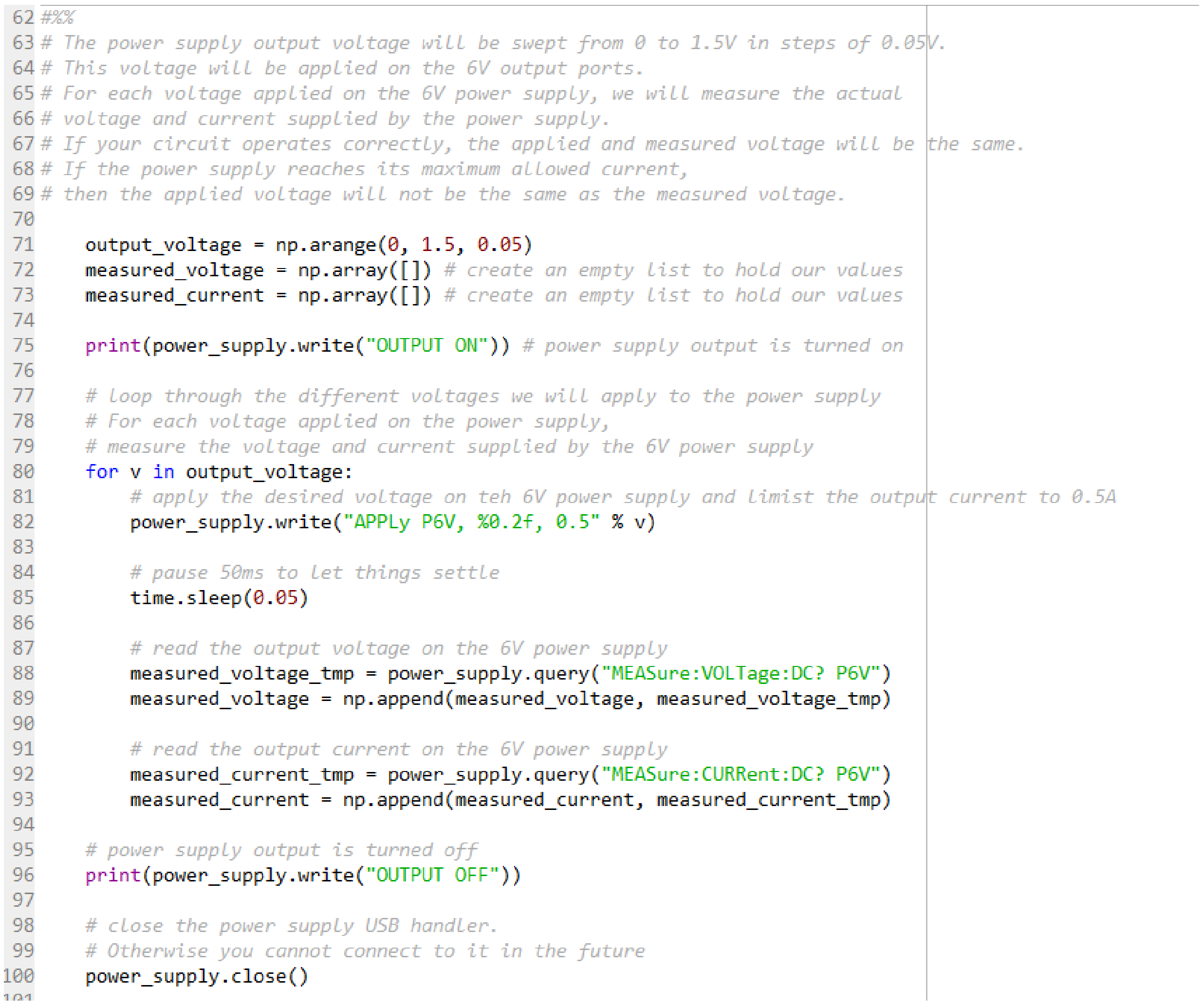
```
#%%
# Open the USB communication port with the power supply.
# The power supply is connected on USB port number power_supply_id.
# If the power supply ss not connected or turned off, the program will exit.
# Otherwise, the power_supply variable is the handler to the power supply

if (power_supply_id == -1):
    print("Power supply instrument is not powered on or connected to the PC.")
else:
    print("Power supply is connected to the PC.")
    power_supply = device_manager.open_resource(devices[power_supply_id])
```

The next step is to apply set of voltage on the diode and measure the current supplied by the power supply. We will also measure the actual voltage that is applied by the power supply. If the circuit demands more current then what the power supply can provide, then the output voltage on the power supply will not reach the voltage level that we want to program. In this case, the power supply operates in a current limited mode and the output voltage will be lowered then the one we have programmed. It is a good practice to double check what voltage is supplied to our test circuit.

The *output_voltage* variable generates a list of voltages between 0 V and 1.5 V in steps of 0.05 V. The *measured_voltage* and *measured_current* are variables that will contain the measured voltage and current provided by the power supply, respectively.

Line 75 of the example sends an instruction to the power supply to turn on its outputs. This is the same as if manually pressed the ON button on the power supply. The power supply has an instruction OUTPUT ON that achieves this task. Since we want a particular task to be executed on the power supply, we write an instruction to the power supply. If we want to read a value from the power supply, such as determine what is the supplied current and voltage by the power supply, we will execute a read instruction. This will be explained shortly. Here is this portion of the code:



```
62 #%%
63 # The power supply output voltage will be swept from 0 to 1.5V in steps of 0.05V.
64 # This voltage will be applied on the 6V output ports.
65 # For each voltage applied on the 6V power supply, we will measure the actual
66 # voltage and current supplied by the power supply.
67 # If your circuit operates correctly, the applied and measured voltage will be the same.
68 # If the power supply reaches its maximum allowed current,
69 # then the applied voltage will not be the same as the measured voltage.
70
71     output_voltage = np.arange(0, 1.5, 0.05)
72     measured_voltage = np.array([]) # create an empty list to hold our values
73     measured_current = np.array([]) # create an empty list to hold our values
74
75     print(power_supply.write("OUTPUT ON")) # power supply output is turned on
76
77     # loop through the different voltages we will apply to the power supply
78     # For each voltage applied on the power supply,
79     # measure the voltage and current supplied by the 6V power supply
80     for v in output_voltage:
81         # apply the desired voltage on teh 6V power supply and limist the output current to 0.5A
82         power_supply.write("APPLy P6V, %0.2f, 0.5" % v)
83
84         # pause 50ms to let things settle
85         time.sleep(0.05)
86
87         # read the output voltage on the 6V power supply
88         measured_voltage_tmp = power_supply.query("MEASure:VOLTage:DC? P6V")
89         measured_voltage = np.append(measured_voltage, measured_voltage_tmp)
90
91         # read the output current on the 6V power supply
92         measured_current_tmp = power_supply.query("MEASure:CURRent:DC? P6V")
93         measured_current = np.append(measured_current, measured_current_tmp)
94
95     # power supply output is turned off
96     print(power_supply.write("OUTPUT OFF"))
97
98     # close the power supply USB handler.
99     # Otherwise you cannot connect to it in the future
100    power_supply.close()
```

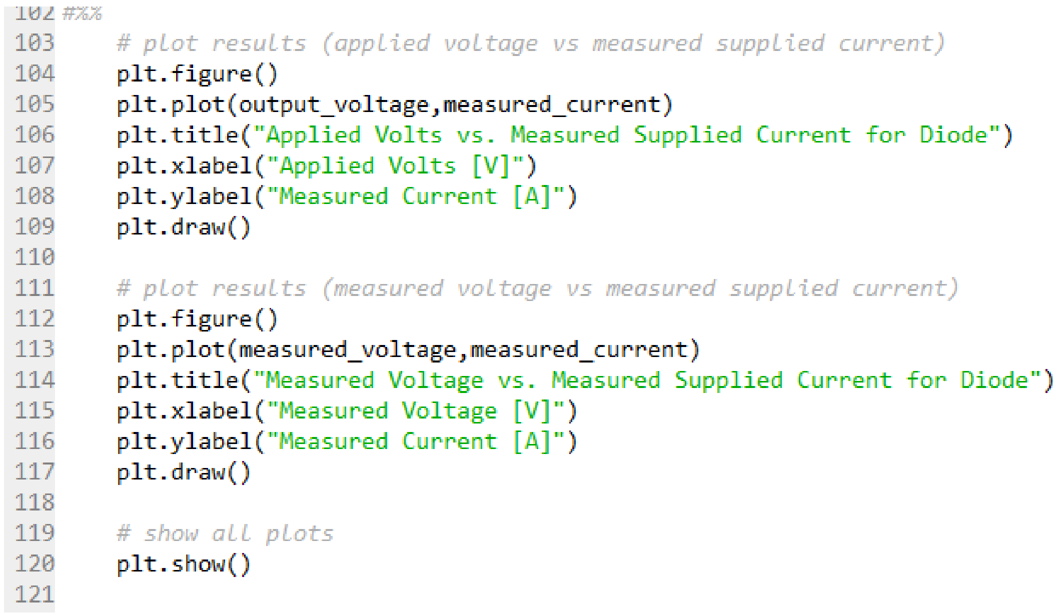
Next, we will sweep through the range of voltages via the *for* loop and apply these values to the power supply. This is achieved via the APPL instruction on line number 82. This instruction sets the output voltage on the 6V power supply to the voltage value defined by the variable *v*, which is swept between 0 V and 1.5 V. We also limit the output current to 0.05 A. If the current level exceeds 0.05A we can damage the diode. Hence, with this instruction, the power supply should never supply current larger than 50 mA.

Since the serial communication with the instrument is slower then the execution speed of the PC, we will wait 50 msec before proceeding with the code. This will give ample time for the power supply to stabilize to the output voltage we like to apply.

Next, we will measure the supplied current and voltage from the power supply. Since we are reading a value from the power supply, we will use the *query* instruction. The SCPI instruction starts with *MEASure* followed by the parameter we like to measure, i.e. voltage or current. We also specified which output port on the power supply we like to execute this measurement. In our case, we have connected the test diode to the 6V power supply and hence, we will measure the current and voltage on this power supply.

Once we sweep through all the voltage values we like to send to the power supply, we turn off the power supply and close the USB interface with the power supply. This is achieved with the code on lines 96 and 100, respectively. It is always a good idea to turn off the power supply. Otherwise, you will be supplying power the test circuit and you can potentially damage it if you do this for prolonged periods of time. We also need to close the USB interface with the power supply because we will not be able to connect to the instrument next time we run our Python code. The previous code will keep the USB port open and will prevent other programs to communicate with the instrument if the port is not closed.

The measured voltage and current are plotted with the following code snippet:



```python
#%%
    # plot results (applied voltage vs measured supplied current)
    plt.figure()
    plt.plot(output_voltage,measured_current)
    plt.title("Applied Volts vs. Measured Supplied Current for Diode")
    plt.xlabel("Applied Volts [V]")
    plt.ylabel("Measured Current [A]")
    plt.draw()

    # plot results (measured voltage vs measured supplied current)
    plt.figure()
    plt.plot(measured_voltage,measured_current)
    plt.title("Measured Voltage vs. Measured Supplied Current for Diode")
    plt.xlabel("Measured Voltage [V]")
    plt.ylabel("Measured Current [A]")
    plt.draw()

    # show all plots
    plt.show()

```

The *plt* variable refers to the *matplotlib.pyplot* library. We define a new figure by executing *plt.figure()* instruction. The instruction *plt.plot* prepares the figure to plot an x-y data. The two argument that follow with this instruction is the x-axis data followed by y-axis data. You have to execute the instruction *plt.draw* to virtually draw this figure. All figures will be displayed on the screen only when you execute *plt.show* instruction.

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 property. 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:

```
power_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)

Connect the test diode to the 25V power supply. Write a Python code that will enable you to plot the current-voltage characteristics of the test diode. The voltage applied to the diode should be swept between 0 V and 8 V in steps of 0.1 V. Limit the maximum output current that the power supply can provide to the diode to 60 mA. For each applied voltage, measure the supplied voltage and current supplied to the diode.

Q1 Plot the current -voltage characteristic of the diode and submit it with your post-lab report. Include your Python code with the report. (10 pts)

Q2 Change the increment step size to 1 V and comment on the difference between the results obtained in question Q1. Include a plot of the current-voltage characteristics of the diode with the new increment step size. (10 pts)

Q3 What would be a reasonable step function when you are plotting the I-V characteristics of a device and how will you determine it? What are some of the tradeoffs when you are determining the increment size? (10 pts)

Checkpoint 2 (60 pts)

Next, we will use both the multimeter and the oscilloscope to determine the current-voltage characteristics of the diode. 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 both the voltage across the diode using the oscilloscope and current through the diode using a multimeter. You will need to figure out the correct SCPI 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.

Sweep the voltage applied across the circuit between 0V and 8V in steps of 200mV. Plot the current-voltage properties of the diode.

Q4 Demonstrate to the TA that your test bench code is working properly. (40 pts)

Q5 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. Why is this current-voltage characteristic different compared to the one you collected in the first checkpoint? (20 pts)