Instrument-USB Interface Manual

Last updated: December 22, 2020
By: Lewis Chan, Evan Dalacu, George Lu

Outline:

1. Device information

1.1 Data Acquisition (DAQ) System

The DAQ is an instrument for high frequency voltage measurements. With the module purchased by the APL, it is capable of acquiring up to 4 channels of data at up to 800 kHz. In the software's "live-plot" mode, short, discrete samples are taken, while in "high frequency measurement" mode, a much longer sample is acquired at the lowest sampling frequency of 1.562 kHz and then further downsampled as necessary.

More information here: *Keysight DAQ970A Data Acquisition System + Keysight DAQM909A 4-Channel 24-Bit Digitizer Module* (https://www.keysight.com/en/pd-2950220-pn-DAQ970A/data-acquisition-system)

1.2 DC Power Supply

The DC Power supply is an instrument for supplying DC voltage. The software can set the voltage and current levels on the device, and turn channels on and off.

More information here: *Keysight E36311A Triple Output Power Supply* (https://www.keysight.com/ca/en/products/dc-power-supplies/bench-power-supplies/e36300-series-triple-output-power-supply-80-160w.html)

1.3 Single Channel Multimeter
The multimeter is an instrument for making current, voltage, and resistance measurements. The software is currently designed to handle current measurements only. Single samples are returned.

More information here: *Keysight 34461A 6.5 Digital Multimeter* (https://www.keysight.com/en/pdx-2891615-pn-34461A/digital-multimeter-6-digit-truevolt-dmm)

2. Setting up devices

Ensure that the computer has enough USB ports to accommodate the devices. If there are not enough ports, a USB hub can be used for additional ports. If the DAQ is used, ensure that the voltage measurement is performed on Channel 1. If a second voltage measurement is needed, then use Channel 2. Ensure all the instruments are connected to the computer by USB cables, and that they are turned on **before** starting the software. No other special preparation is needed on the devices themselves.

3. Setting up software

3.1 Prerequisite software

The interface requires the following software:
Python (≥3.8): https://www.python.org/downloads/ or https://www.anaconda.com/products/individual

- pyvisa: https://pyvisa.readthedocs.io/en/latest/introduction/getting.html
- numpy: https://numpy.org/install/
- matplotlib: https://matplotlib.org/3.3.3/users/installing.html
- *Built-in: random, time, numpy, matplotlib, tkinter*

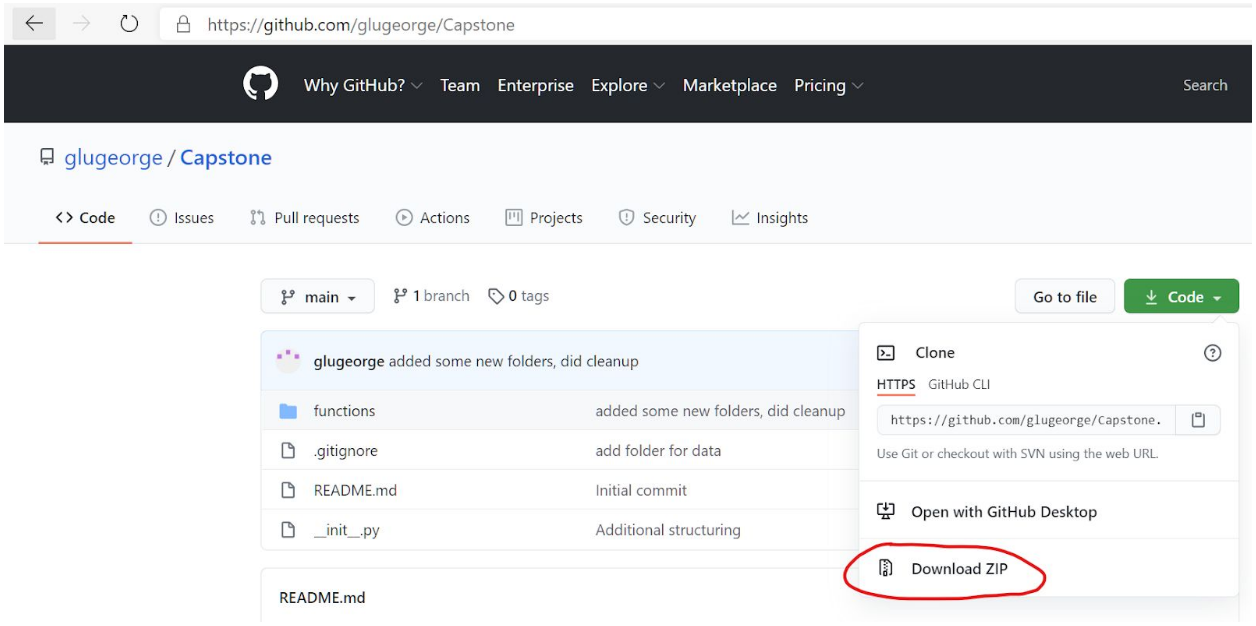
The external packages needed by Python can all be installed through conda or pip.

If Anaconda/Miniconda is installed, open an Anaconda Prompt window and type: `conda install <package>`. To verify if a package is installed, use `conda list <package>`. (Note: Anaconda already includes numpy and matplotlib as part of installation.)

If only Python is installed, open a Command Prompt window and type: `python -m pip install <package>`. To verify if a package is installed, use `python -m pip show <package>`.

3.2 Acquiring software/additional setup

The interface can be found at https://github.com/glugeorge/Capstone. The code can be saved in many ways. From the link, a ZIP file can be downloaded with all the code (process shown in the image below). Additionally, if a GIT client exists on your computer, you can clone this repository using Git or SVN and cloning from this HTTPS link: https://github.com/glugeorge/Capstone.git.



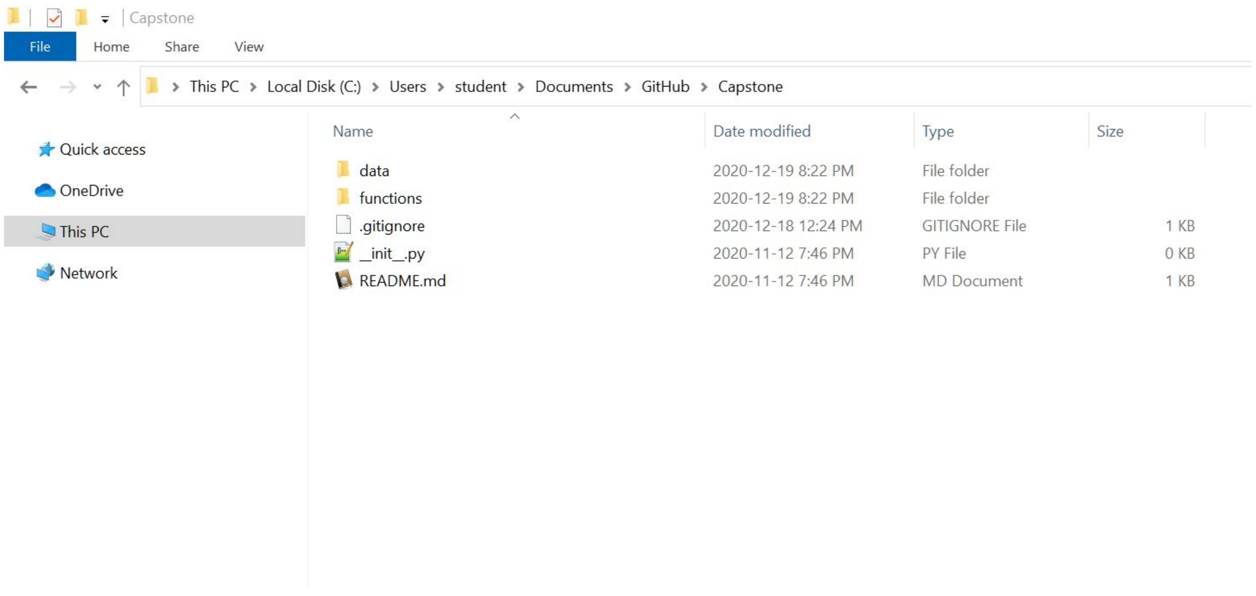
glugeorge added some new folders, did cleanup

functions	added some new folders, did cleanup
.gitignore	add folder for data
README.md	Initial commit
__init__.py	Additional structuring

```
https://github.com/glugeorge/Capstone.
```

Use Git or checkout with SVN using the web URL.

Unzip the downloaded software to `C:\Users\student\Documents\GitHub`. This is the default location, though it can be saved anywhere as long as the data directory path is updated in the code. Within the folder `Capstone`, create a folder titled `data`. Any data will be saved to this folder. The directory should look like the image below. The folder from which the program will be run is the `functions` folder. If you would like to save data to a different location, update the `data_dir_name` variable in `functions/global_variables.py` to the full path for the folder in which you would like to store the data (use either Python raw strings or double backslashes to avoid errors due to interpretation as escape characters). Ensure this folder path exists before running the software.


Capstone
File
Home
Share
View
This PC > Local Disk (C:) > Users > student > Documents > GitHub > Capstone
Quick access
OneDrive
This PC
Network

Name	Date modified	Type	Size
data	2020-12-19 8:22 PM	File folder	
functions	2020-12-19 8:22 PM	File folder	
.gitignore	2020-12-18 12:24 PM	GITIGNORE File	1 KB
__init__.py	2020-11-12 7:46 PM	PY File	0 KB
README.md	2020-11-12 7:46 PM	MD Document	1 KB

3.3 Setting up device interface

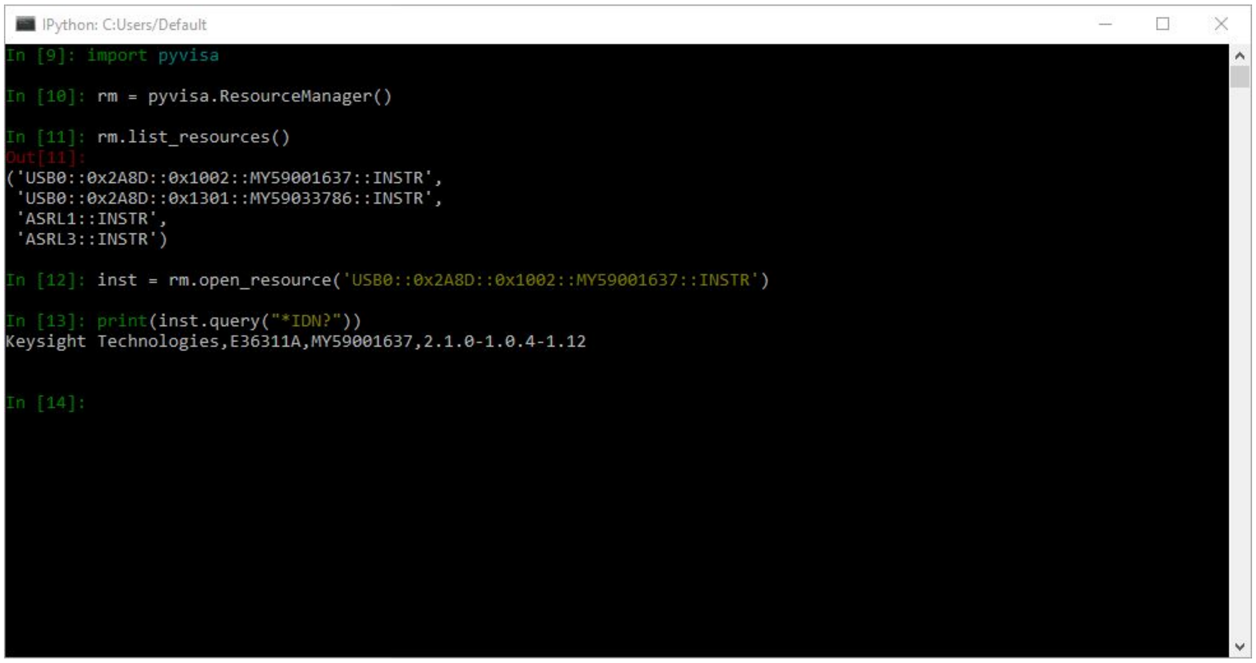
After ensuring that the prerequisite software is installed, downloading the repository with the necessary files, and having the devices connected via USB and turned on, the device IDs must be determined before the software can be run. This can be done by running a Python instance (either through an IDE like Spyder or an Anaconda Prompt / Command Prompt window) and entering the following set of commands:

```
import pyvisa
rm = pyvisa.ResourceManager()
rm.list_resources()
```

This will return a tuple of device IDs. To see which device ID corresponds with which device, enter:

```
inst = rm.open_resource('<device ID>')
print(inst.query("*IDN?"))
```

This should return some basic identifying information about the device, such as its brand and type, as shown in the figure below. Next, open `global_variables.py` in the `functions` folder and update the device name variables to the correct ID as provided by the `rm.list_resources()` command. From here, the interface should be set up and ready to run.



```
In [9]: import pyvisa

In [10]: rm = pyvisa.ResourceManager()

In [11]: rm.list_resources()
Out[11]:
('USB0::0x2A8D::0x1002::MY59001637::INSTR',
 'USB0::0x2A8D::0x1301::MY59033786::INSTR',
 'ASRL1::INSTR',
 'ASRL3::INSTR')

In [12]: inst = rm.open_resource('USB0::0x2A8D::0x1002::MY59001637::INSTR')

In [13]: print(inst.query("*IDN?"))
Keysight Technologies,E36311A,MY59001637,2.1.0-1.0.4-1.12


In [14]:
```

4. Running the interface
To run, open an Anaconda / Command Prompt window, go into the functions directory in the main folder (type `cd C:\Users\student\Documents\GitHub\Capstone\functions`, with the path determined by where you have unzipped the software in section 3.2), and then type `python gui.py`. This will open the main window for the interactive interface. The user interface window has three tabs: Chart recorder, High frequency voltage measurement and Hall effect. These tabs can be accessed at the top of the window.

4.1 First tab: Chart Recorder
The first tab controls the chart recorder software. Data from the data acquisition system can be plotted against time or against other measurements from the DAQ. To run the chart recorder software a) specify the filename to which to save the data, b) choose the measurements to plot, c) specify plot options and, d) run the live plotter. This first tab is shown in the figure below.

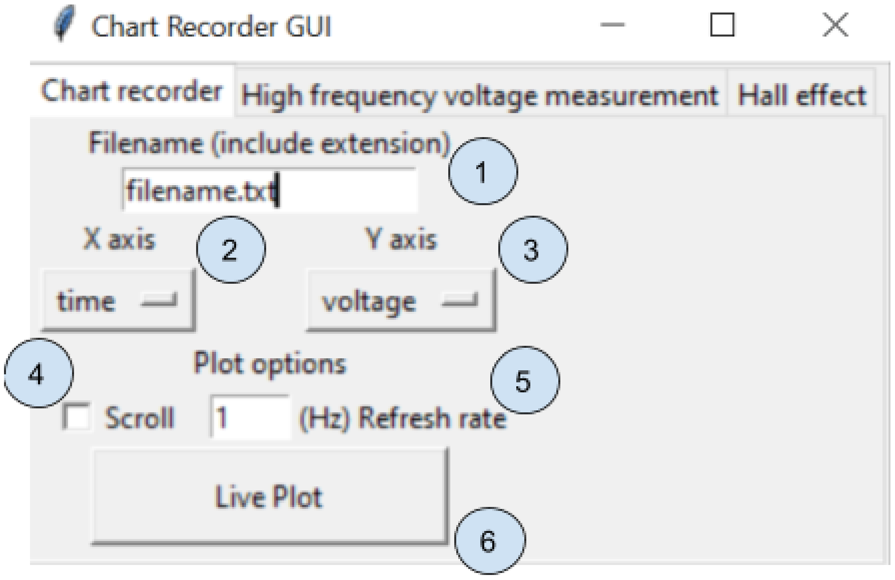


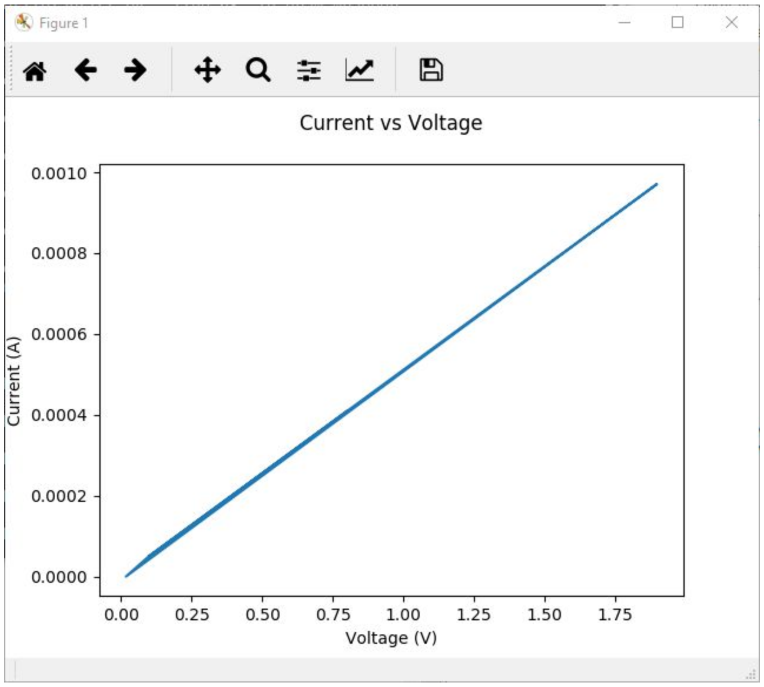
Figure labels:
1. Textbox for the file name (including a .txt or .csv extension) to which to save the data used for plotting. The data will be saved to the following folder by default: `C:\Users\student\Documents\GitHub\Capstone\data`
2. Drop down menu to specify measurements on the x-axis.
3. Drop down menu to specify measurements on the y-axis.
4. Checkbox to toggle scrolling the chart recorder window as new values are displayed.
5. Textbox to specify the chart recorder refresh rate in Hz. The maximum refresh rate which works reliably is 10 Hz. This refresh rate is also an approximate guide - for a precise time interval in between measurements for data collection, use the high frequency recorder instead.
6. Button to launch the chart recorder.

Axis options:

x-axis	y-axis	description
random	random	Returns random values between 0 and 1. Mostly used for debugging.

time	time (s)	Returns the time at which measurements are made starting from 0. Mostly useful for the x-axis.
current	current (A)	Returns current as measured by the digital multimeter.
voltage	voltage (V)	Returns voltage from channel 1 of the data acquisition system.
	two voltages (V)	Returns two voltages from channel 1 and 2 of the data acquisition system. Only available on the y-axis.

When the chart recorder launches, a new window will pop up with a display like the following figure. This chart recorder example is measuring the current and voltage across a resistor as the voltage changes.

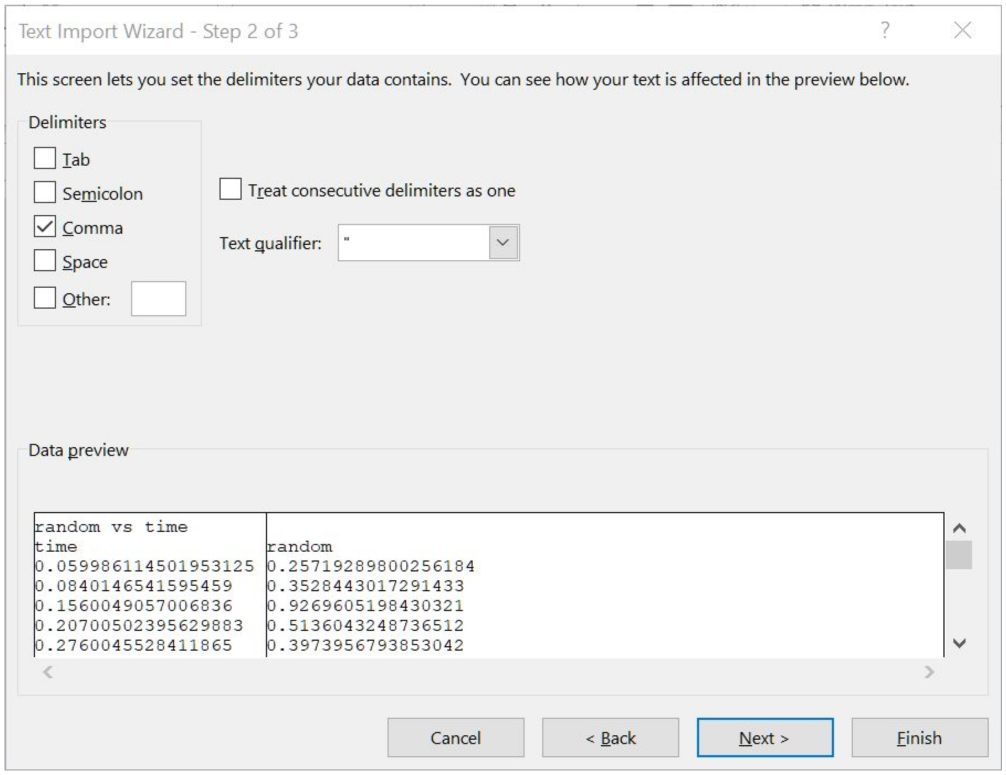


From left to right, the buttons in the toolbar are:

1. Reset original view
2. Back to previous view
3. Forward to next view
4. Pan
5. Zoom
6. Configure subplots
7. Edit axis, curve, and image parameters
8. Save figure

These buttons are generally not applicable to the live plot (they are a built-in functionality that are available with all Matplotlib plots), with the exception of the “Save figure” button, which allows the user to save the graph as a PNG image file.

The data shown in the live plot is saved as a .txt or .csv file. This file can also be viewed in Excel, which may be more useful in many cases. In the case of a text file, this is done by first launching Excel and opening the text file directly through it. A text import wizard window should appear. Click ‘Next’ to go to step 2 of 3, select ‘Comma’ as the delimiter, and then click ‘Finish’ (see screenshot below). The data values should now be properly split into cells.



This screen lets you set the delimiters your data contains. You can see how your text is affected in the preview below.

Delimiters

- [ ] Tab
- [ ] Semicolon
- [x] Comma
- [ ] Space
- [ ] Other:

- [ ] Treat consecutive delimiters as one

Text qualifier: "

Data preview

```
random vs time
time                  random
0.059986114501953125  0.25719289800256184
0.0840146541595459    0.3528443017291433
0.1560049057006836    0.9269605198430321
0.20700502395629883   0.5136043248736512
0.2760045528411865    0.3973956793853042
```

If the data was saved as a .csv file, it may be opened by Excel directly.

4.2 Second tab: High frequency voltage measurement

The second tab runs a high frequency voltage measurement on channel 1 of the DAQ. The chart recorder software is unable to update at the speed required for certain experiments. So, this tab can be used to take measurements which require higher speed and don’t require visualization. To run the high frequency measurements a) specify the filename to which to save the data, b) specify the sampling rate and the period over which to sample and, c) take the measurement. The layout of this tab is shown in the figure below.

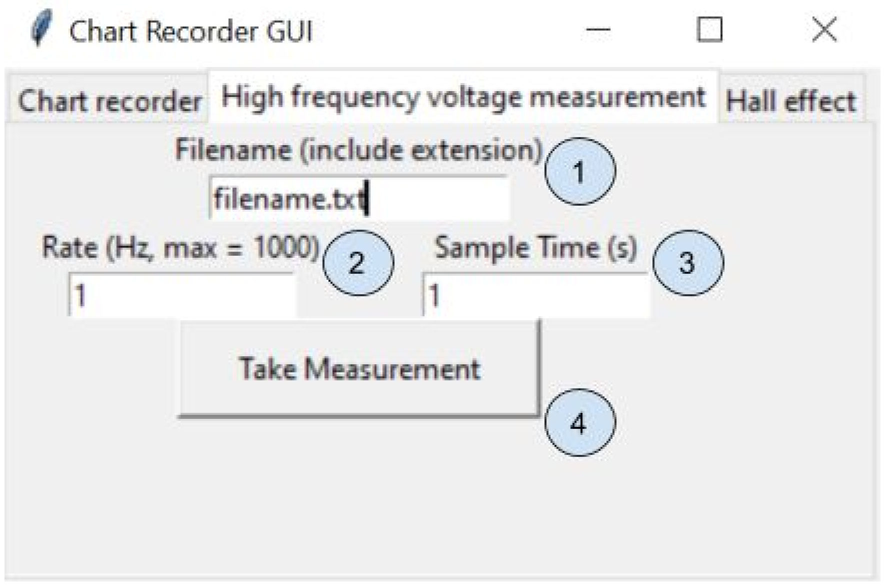


Figure labels:

1. Textbox for the file name (including a .txt or .csv extension) to which to save the data from the high frequency measurement. The data will be saved to the following folder by default: `C:\Users\student\Documents\GitHub\Capstone\data`
2. Textbox to specify the sampling rate in Hz. The maximum sampling rate is 1000Hz.
3. Textbox to specify the sampling period in seconds.
4. Button to take the high frequency measurements.

4.3 Third tab: Hall effect

The third tab is used to run Hall experiment measurements. It can be used to take the necessary measurement for both sheet resistance and Hall coefficient calculations. The method outlined in the NIST resistivity and Hall measurements page (which can be found here: https://www.nist.gov/pml/nanoscale-device-characterization-division/popular-links/hall-effect/resistivity-and-hall#resistivity) is used to organize this window. The probe ordering convention from the semiconductor resistance, band gap, and Hall effect lab (which can be found here: https://www8.physics.utoronto.ca/~phy326/hall/hall.pdf) is used. Each measurement returns a pair of supplied current and measured voltage which can be used to calculate the desired values. For Hall coefficient measurements, the magnetic field orientation must be specified as well. To obtain a current-voltage pair used to compute resistivity, a) specify the filename to which to save the data, b) toggle the magnetic field option **off**, c) specify which probes measure current and which measure voltage and, d) take the measurement. For the Hall coefficient, the same steps are followed but the magnetic field is toggled **on** and the magnetic field direction specified. The layout of the third tab can be seen below.

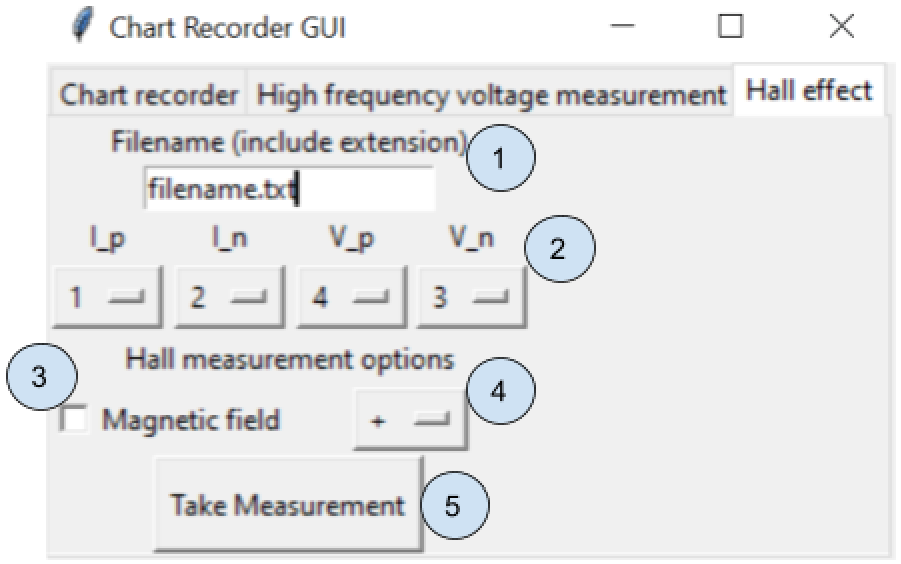


Figure labels:

1. Textbox for the file name (including a .txt or .csv extension) to which to save the data from the Hall effect. The data will be saved to the following folder by default: `C:\Users\student\Documents\GitHub\Capstone\data`
2. Dropdown menu specifying the probes associated with each current/voltage positive/negative contact. The positive contacts for current and voltage are I_p and V_p respectively and the negative contacts are I_n and V_n.
3. Checkbox to toggle the magnetic field. When the magnetic field is on, Hall coefficient measurements are taken; when it is toggled off, resistivity measurements are taken.
4. Dropdown menu specifying the orientation of the magnetic field.
5. Button to measure current-voltage pair.

Only certain combinations of current and voltage are used in experiment, hence only these values are made permissible in the user interface. For example, current and voltage cannot be probed at the same contact. The list of allowable combinations are below, where $I_{ij} = I_p - I_n$, and $V_{ij} = V_p - V_n$. For Hall coefficient measurements, V_{ij+} and V_{ij-} indicate voltage measured with either a positive or negative magnetic field applied.

Resistivity measurement current-voltage pairs		Hall coefficient current-voltage pairs	
I_{12}	V_{43}	I_{13}	V_{24+}
I_{21}	V_{34}	I_{24}	V_{13+}
I_{34}	V_{21}	I_{31}	V_{42+}
I_{43}	V_{12}	I_{42}	V_{31+}
I_{14}	V_{23}	I_{13}	V_{24-}
I_{23}	V_{14}	I_{24}	V_{13-}
I_{32}	V_{41}	I_{31}	V_{42-}
I_{41}	V_{32}	I_{42}	V_{31-}

5. Adding to interface

5.1 Adding new device

1. For a new device, ensure that it has an API like that of Keysight's that allows for a PyVISA interface. Connect the instrument to the computer via USB.
2. Follow the same steps detailed in section 3.3, adding a new device and device path to the global variables.
3. Next, create a new file `<device_name>.py`. This is similar to the files `daq.py`, `multimeter.py`, `dc_ps.py`, etc. These files contain the functions associated with each device.
4. In whichever module requires the new device, import that device - its functions can then be called with `<device_name>.<function_name>`.
5. Furthermore, this device needs to be initialized whenever it is run. This can be done with the function "init_devices" from common_functions.py. In any module which requires this device, make sure to append it to the array of devices being initialized. For example, if you need the power supply and the data acquisition device for one specific function, the command would be:
   ```
   dc_ps_dev, daq_dev = init_devices([dc_ps_name, daq_name])
   ```
6. A template for a new device named `new_device_template.py` can be found in the "functions" folder.

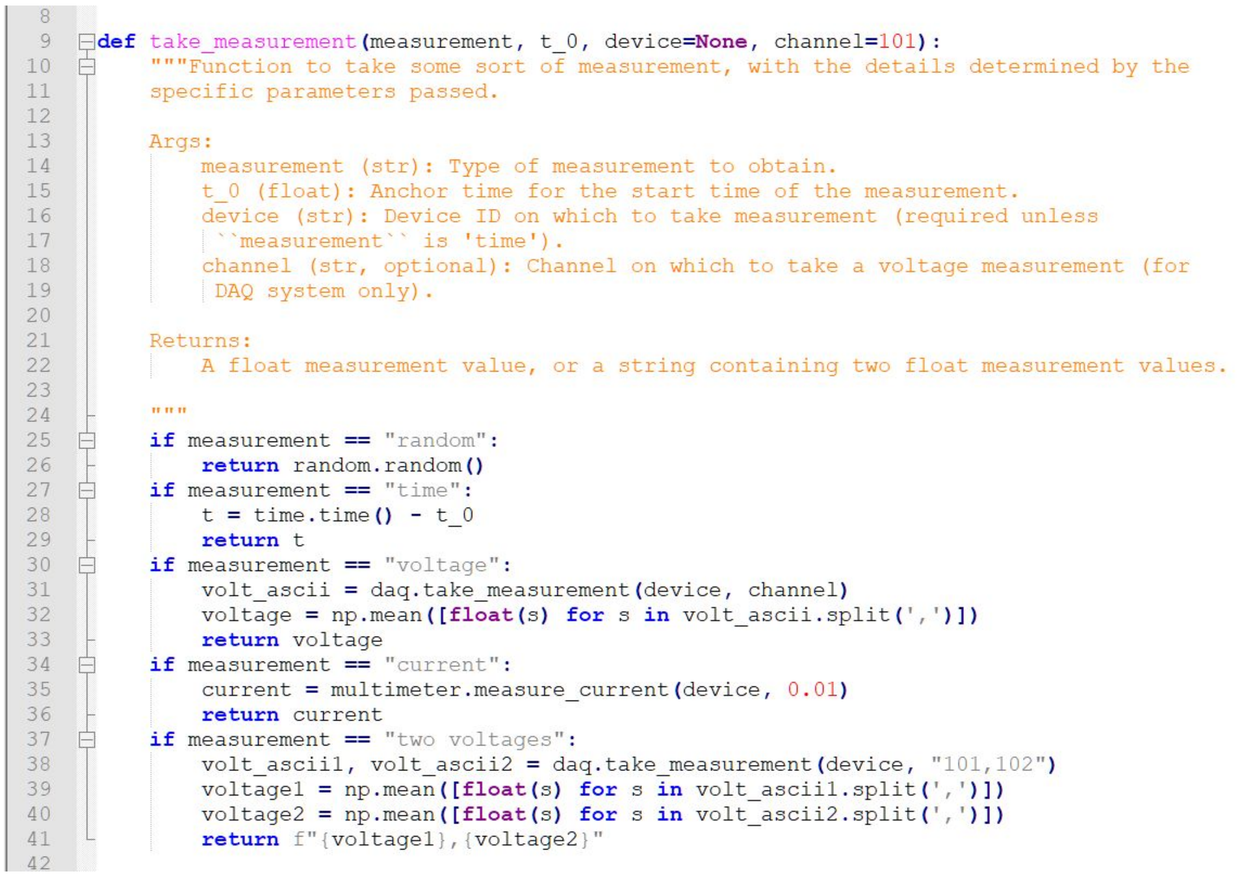
5.2 Adding new functionality

1. To add a new lab/tool, we create a new file called `<lab/tool_name>.py`. Our example is `hall.py`.
2. At the top of the file, import whatever devices will be needed for this lab/tool.
3. Within this file, write the code which yields the desired functionality. Organize your code in functions which can be called in separate modules.
4. A template for a new functionality/experiment named `new_functionality_template.py` can be found in the "functions" folder.
5. Now, in the gui.py file, import this new module, and create a function within the "functions" folder which runs the new module.
6. Next, create a new tab to run this function in the "tab" section of the `gui.py` file.
7. Finally, at the bottom of the file, specify the layout of the new tab for the new function.

6. Example of potential modifications: Adding a dynamic voltage change functionality

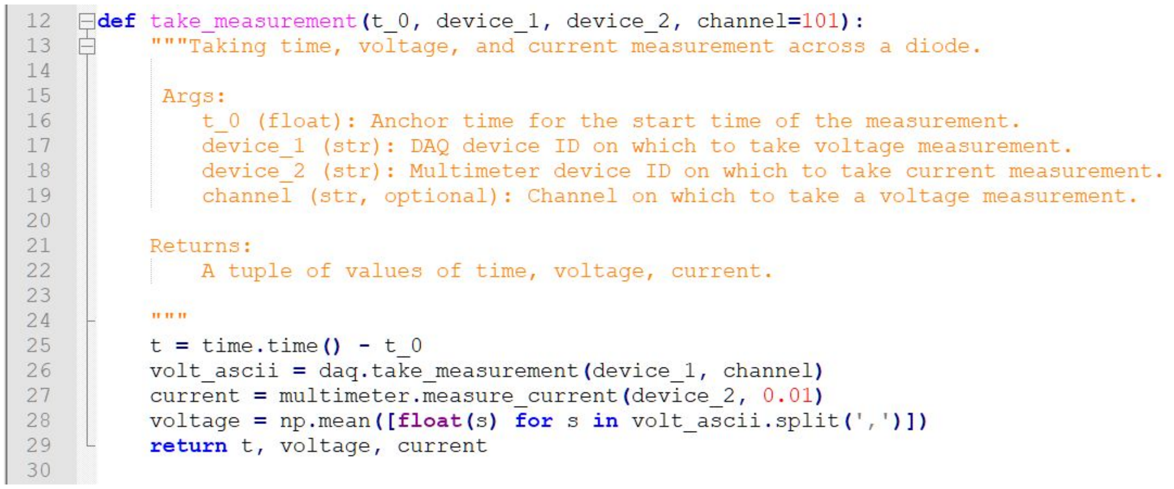
This section demonstrates how to modify our existing code to add new functionalities such as a plotting function with an automated voltage sweep. This could be applied to a diode lab for example, where we want to compare the voltage and current across the diode while the voltage being applied to it varies automatically. Here, we create a new module called `diode.py`, which is included in our software package. With this script, we are able to repurpose a lot of the existing code from our `plotter.py` module. As a result, we start off by copying the whole code into our new `diode.py` file. Next, we remove redundancies - the `plotter.py` function relies on many user inputs whereas the diode one is a one click example, so we remove some of the variables such that our diode measurement is solely focused on taking voltages. This can be seen in changes made to the `take_measurement()` function. A comparison between the functions in the two scripts is shown here:

`plotter.py`:



```python

def take_measurement(measurement, t_0, device=None, channel=101):
    """Function to take some sort of measurement, with the details determined by the
    specific parameters passed.

    Args:
        measurement (str): Type of measurement to obtain.
        t_0 (float): Anchor time for the start time of the measurement.
        device (str): Device ID on which to take measurement (required unless
          ``measurement`` is 'time').
        channel (str, optional): Channel on which to take a voltage measurement (for
          DAQ system only).

    Returns:
        A float measurement value, or a string containing two float measurement values.

    """
    if measurement == "random":
        return random.random()
    if measurement == "time":
        t = time.time() - t_0
        return t
    if measurement == "voltage":
        volt_ascii = daq.take_measurement(device, channel)
        voltage = np.mean([float(s) for s in volt_ascii.split(',')])
        return voltage
    if measurement == "current":
        current = multimeter.measure_current(device, 0.01)
        return current
    if measurement == "two voltages":
        volt_ascii1, volt_ascii2 = daq.take_measurement(device, "101,102")
        voltage1 = np.mean([float(s) for s in volt_ascii1.split(',')])
        voltage2 = np.mean([float(s) for s in volt_ascii2.split(',')])
        return f"{voltage1},{voltage2}"

```

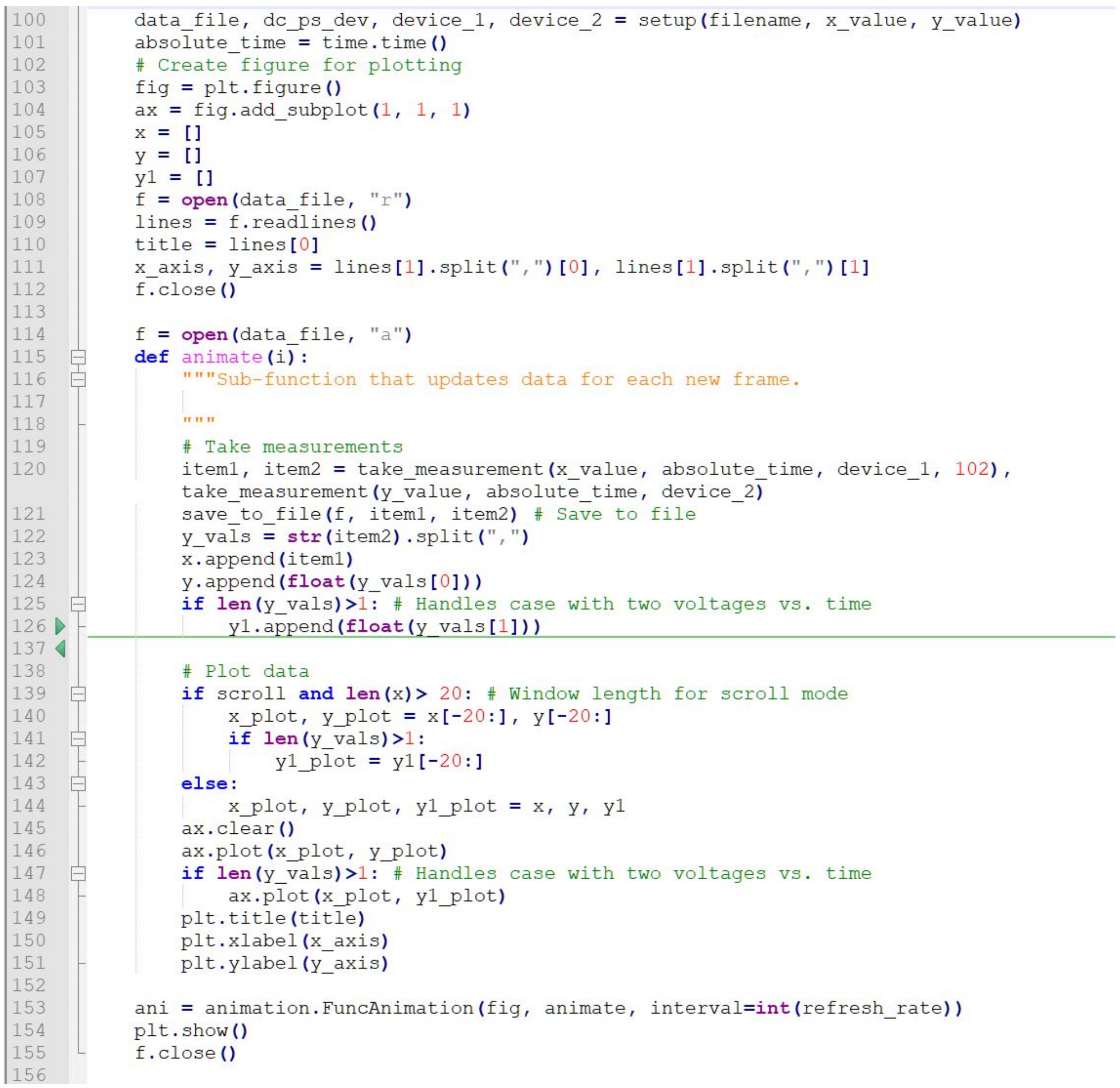
`diode.py`:



```python
def take_measurement(t_0, device_1, device_2, channel=101):
    """Taking time, voltage, and current measurement across a diode.

     Args:
        t_0 (float): Anchor time for the start time of the measurement.
        device_1 (str): DAQ device ID on which to take voltage measurement.
        device_2 (str): Multimeter device ID on which to take current measurement.
        channel (str, optional): Channel on which to take a voltage measurement.

    Returns:
        A tuple of values of time, voltage, current.

    """
    t = time.time() - t_0
    volt_ascii = daq.take_measurement(device_1, channel)
    current = multimeter.measure_current(device_2, 0.01)
    voltage = np.mean([float(s) for s in volt_ascii.split(',')])
    return t, voltage, current

```

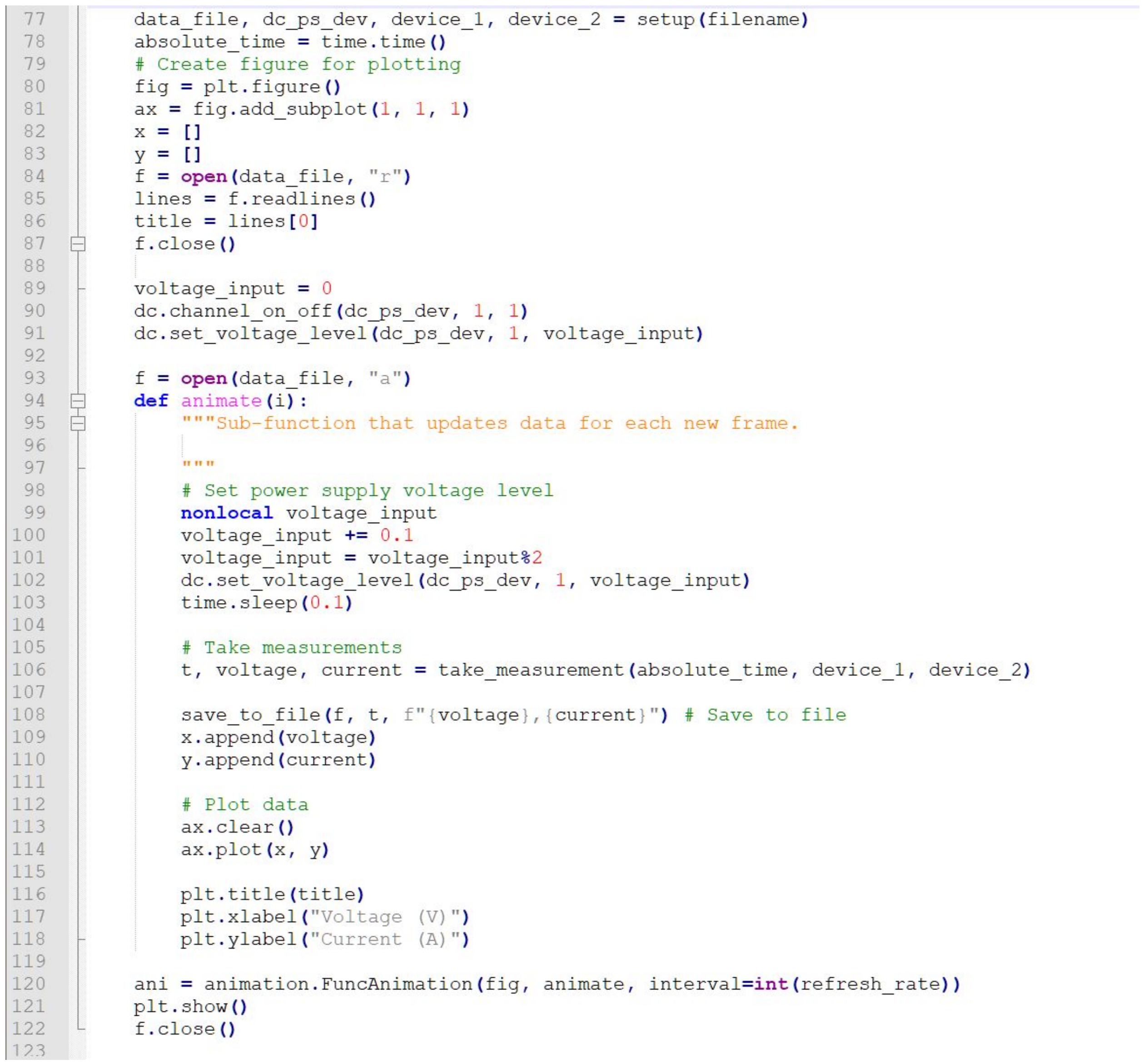
The `live_plot()` function from `plotter.py` has also been renamed to the more meaningful `diode_measurement()` and has likewise been edited to remove redundancies. The next and most crucial step is to add a dynamic time varying voltage. This is implemented using functions available to us in our `dc.py` module. Here we start at 0 V and increment the voltage up by 0.1 V with every animation frame update (default 1 Hz) until 2 V is reached, upon which the supplied voltage level jumps back to 0 V. The specific numbers can easily be changed to suit the user's needs. A comparison of the differences between the functions in the two scripts is shown below:

`plotter.py:`



```
    data_file, dc_ps_dev, device_1, device_2 = setup(filename, x_value, y_value)
    absolute_time = time.time()
    # Create figure for plotting
    fig = plt.figure()
    ax = fig.add_subplot(1, 1, 1)
    x = []
    y = []
    y1 = []
    f = open(data_file, "r")
    lines = f.readlines()
    title = lines[0]
    x_axis, y_axis = lines[1].split(",")[0], lines[1].split(",")[1]
    f.close()

    f = open(data_file, "a")
    def animate(i):
        """Sub-function that updates data for each new frame.

        """
        # Take measurements
        item1, item2 = take_measurement(x_value, absolute_time, device_1, 102),
        take_measurement(y_value, absolute_time, device_2)
        save_to_file(f, item1, item2) # Save to file
        y_vals = str(item2).split(",")
        x.append(item1)
        y.append(float(y_vals[0]))
        if len(y_vals)>1: # Handles case with two voltages vs. time
            y1.append(float(y_vals[1]))

        # Plot data
        if scroll and len(x)> 20: # Window length for scroll mode
            x_plot, y_plot = x[-20:], y[-20:]
            if len(y_vals)>1:
                y1_plot = y1[-20:]
        else:
            x_plot, y_plot, y1_plot = x, y, y1
        ax.clear()
        ax.plot(x_plot, y_plot)
        if len(y_vals)>1: # Handles case with two voltages vs. time
            ax.plot(x_plot, y1_plot)
        plt.title(title)
        plt.xlabel(x_axis)
        plt.ylabel(y_axis)

    ani = animation.FuncAnimation(fig, animate, interval=int(refresh_rate))
    plt.show()
    f.close()
```

diode.py:



```
 77     data_file, dc_ps_dev, device_1, device_2 = setup(filename)
 78     absolute_time = time.time()
 79     # Create figure for plotting
 80     fig = plt.figure()
 81     ax = fig.add_subplot(1, 1, 1)
 82     x = []
 83     y = []
 84     f = open(data_file, "r")
 85     lines = f.readlines()
 86     title = lines[0]
 87     f.close()
 88
 89     voltage_input = 0
 90     dc.channel_on_off(dc_ps_dev, 1, 1)
 91     dc.set_voltage_level(dc_ps_dev, 1, voltage_input)
 92
 93     f = open(data_file, "a")
 94     def animate(i):
 95         """Sub-function that updates data for each new frame.
 96
 97         """
 98         # Set power supply voltage level
 99         nonlocal voltage_input
100         voltage_input += 0.1
101         voltage_input = voltage_input%2
102         dc.set_voltage_level(dc_ps_dev, 1, voltage_input)
103         time.sleep(0.1)
104
105         # Take measurements
106         t, voltage, current = take_measurement(absolute_time, device_1, device_2)
107
108         save_to_file(f, t, f"{voltage},{current}") # Save to file
109         x.append(voltage)
110         y.append(current)
111
112         # Plot data
113         ax.clear()
114         ax.plot(x, y)
115
116         plt.title(title)
117         plt.xlabel("Voltage (V)")
118         plt.ylabel("Current (A)")
119
120     ani = animation.FuncAnimation(fig, animate, interval=int(refresh_rate))
121     plt.show()
122     f.close()
123
```

The critical additions allowing for automatic variation of the voltage are on lines 89-91 and 99-103 in `diode.py`. Lines 89-91 in the `diode_measurement()` function turn on channel 1 on the DC power supply and set the supplied voltage to an initial value of 0 V. Lines 99-103 within the `animate` subfunction ramp up the supplied voltage by 0.1 V increments, resetting to 0 V when 2 V is reached. A slight time delay is added to avoid measurement lag between the multimeter and the DAQ system. These lines are reproduced below.

```
voltage_input = 0
dc.channel_on_off(dc_ps_dev, 1, 1)
dc.set_voltage_level(dc_ps_dev, 1, voltage_input)
…
     nonlocal voltage_input
     voltage_input += 0.1
     voltage_input = voltage_input%2
     dc.set_voltage_level(dc_ps_dev, 1, voltage_input)
     time.sleep(0.1)
```

Our example `diode.py` module is now able to run - within the functions directory in the main folder, call `python diode.py` in an Anaconda Prompt / Command Prompt window (refer to Section 4). A future step would be to incorporate this into the GUI, as outlined in steps 5 through 7 of section 5.2, and described in detail below.

After the new module has been written - in this case `diode.py` - changes can be made to the `gui.py` file to interface with the new function via the GUI. Following from 5.2, we first import the new module (step 5).

```
from diode import *
```

Then, we write a shell function containing our module (continuing with step 5), which will be called by the user interface. The values specified in the GUI are passed through the function.

```
def plot_diode_IV():
     """Function for real-time plotting of diode I-V
     characteristic. See ``diode.py``.
          """
diode_measurement(data_dir_name+diode_file_name.get(), 500)
```

We add the diode tab into the GUI (step 6). This will keep the tool organized by separating different modules into different tabs.

```
tab_diode = ttk.Frame(tab_parent)
tab_parent.add(tab_diode, text="Diode IV Plot")
```

Next, we add the various components of our new tab which will allow the user to specify the values to pass into the function (continuing with step 6). Here we first have a textbox for the filename input and a button which calls the function we defined earlier, running the `diode_measurement.py` module.

```
diode_file_label = tk.Label(tab_diode,
                           text="Filename (include extension)")
diode_file_name = tk.Entry(tab_diode)
live_plot_diode = tk.Button(tab_diode, text='Live Plot',
                           height = 2, width = 20,
                           command=plot_diode_IV)
```

Finally, we include some lines to organize the buttons, labels and textboxes as desired (step 7).

```
diode_file_label.grid(column=1)
diode_file_name.grid(column=1, row = 1)
live_plot_diode.grid(column=1, columnspan=3, rowspan=2)
tab_diode.grid_columnconfigure(1, weight=1)
```