Introducing Automation for Data Acquisition and Analysis in a Sophomore-level Electronics Course

Dr. Carl K. Frederickson, University of Central Arkansas

Dr. Frederickson has taught physics at UCA for over 25 years. He is the current department chair and is leading the development of an Engineering Physics degree program.

Introducing Automation for Data Acquisition and Analysis in a Sophomore level Electronics Course

Introduction

The University of Central Arkansas started a new engineering program focused on robotics and automation in the fall of 2018. This program, the first engineering program on campus, is mix of electrical, mechanical, and computer engineering appropriate for developing robotic systems. As part of the engineering curriculum, students take an electronics course in the spring semester of their second year. The course introduces students to the behavior of circuits and elements when varying voltage sources are used. The frequency response of circuits is of particular interest. This provides an opportunity to introduce the concept of automating data acquisition and analysis. We are taking advantage of this opportunity.

Program Description

The Engineering Physics program (Program) is focused on robotics and automation. The curriculum consists of classes in electronics, mechanics, computer programming, and robotics. The progression of classes provides students with the skills to develop autonomous robotic systems as part of the senior design capstone. Senior Design students in the program have participated in the Autonomous Vehicle Challenge (AVC) as part of the National Robotics Challenge [1] each of the last two years. The Program has sent two teams to participate in the AVC each of the last two years (2022 and 2023). In the first year that AVC was available after covid (2022) the team placed 1st and 3rd in the competition.

The Program applied for accreditation as an Engineering Physics program during the 2022/2023 review cycle. The Engineering Physics designation best matches the interdisciplinary nature of the Program. Along with the 45 hours of engineering course work, students must complete 9 hours of upper division physics courses includes 6 hours of advanced electricity and magnetism.

To prepare students for the capstone sequence their senior year they complete the engineering and sciences courses in Table 1. The Program from the outset has been developed with no increase in operating budget. While funds were provided for faculty lines associated with the Program, no new operating funds have been identified. Laboratory equipment has been provided in conjunction with the construction of new space dedicated to the Program. The Program has been designed to use only open-source software for instruction. This provides a cost savings for the University as well as for students. Most of the computer systems used in the EP program have the Ubuntu 22.04 LTS system installed. This includes the computers in the electronics laboratory.

The robotics courses introduce students to the Robot Operating System (ROS) as the backbone of their robotics code. The Program's use of ROS for robotics classes aligns with this choice to focus on open-source software. NASA is adopting ROS for its robotics programs [2] [3]. For many of our students, this is the first time that they will be introduced to the Python coding language. The Program was faced with the need to introduce the Python coding language prior to the third-year robotics courses. The current curriculum combined with the requirement for general education does not allow for any general elective hours. There are ten hours of engineering electives which are needed to provide depth to the degree. An option that provides both an introduction to the use of Python and relates to the automation theme of the degree is being developed for use in the Electronics course (Electronics).

Table 1: Engineering Physics partial course list.

Engineering and Science Courses in the first three years			
First Year	Physics 1	Computer Science 1	Introduction to Engineering
Second Year	Physics 2	Computer Science 2	Electronics
	Statics	Dynamics	
Third Year	Microelectronics	Robotics 1	Engineering Elective
	Electromagnetism 1	Electromagnetism 2	
	Microcontrollers	Robotics 2	

Electronics Class Description

Electronics uses the text authored by Dr. Curtis Meyer [4]. The Course meets twice a week for 260 minutes each day. Students are in a laboratory setting for the entire course. During class, students receive instruction, present material from the homework assignments and completed lab activities, as well as working through the different laboratory activities. The course is only offered in the spring semester.

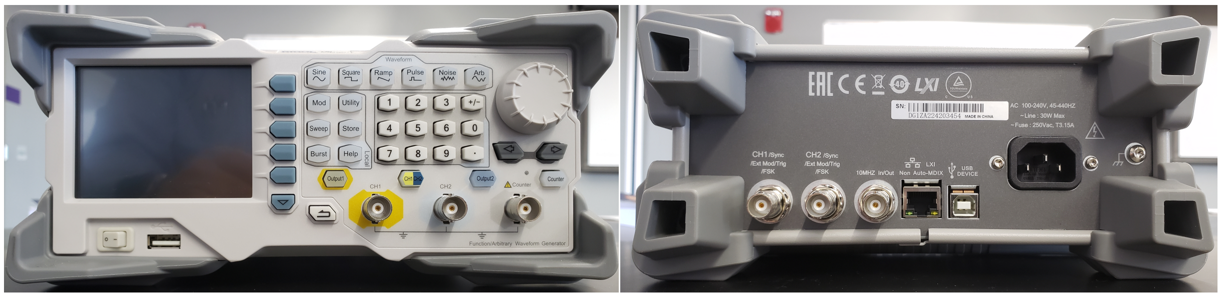


Figure 1: Rigol DG1022Z digital signal generator.

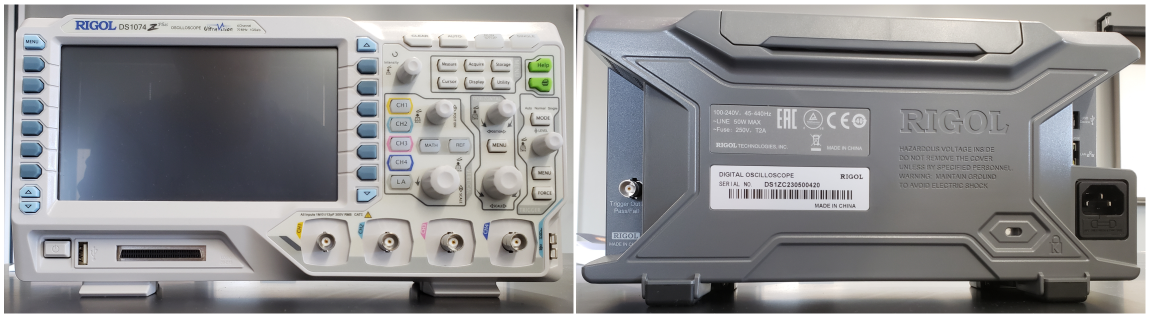


Figure 2: Rigol DS1074Z digital oscilloscope.

Each student is assigned their own laboratory bench equipped with a DC power supply, a Rigol DG1022Z digital function generator (RFG, Figure 1), and a Rigol DS1074Z digital oscilloscope (DSP, Figure 2). The RFG is a 25 MHz signal generator. It can produce a sine wave with frequencies from 1 μHz to 25 MHz as well as square, ramp, pulse, and arbitrary output signals. The amplitude of the output signals ranges from 1.0 mVpp to 10.0 Vpp for signals below 10 MHz and from 1.0 mVpp to 5.0 Vpp for signals for signals above 10 MHz. The RFG has a 50 Ω output impedance. The DSP has a 70 MHz bandwidth over 4 analog input channels. The horizontal scale runs from 5 ns/div up to 50 s/div in a 1,2,5 sequence (5 ns/div, 10 ns/div, 20 ns/div, 50 ns/div, …) for a total of 31 horizontal scale values. The vertical scale has 8-bit resolution over 13 different scales from 1 mV/div up to 10 V/div. The input impedance of the DSP is 1 MΩ.

During the spring semester, this is the only course offered in the assigned space and students can leave their workstations setup between classes. Each lab bench also has access to a Dell laptop with the Ubuntu Desktop 22.04 operating system installed. This is the same system used in the robotics courses. This system allows students to communicate with both the RFG and DSP through the USB ports.

This course builds on the circuits introduced in the second semester physics course introducing operational amplifiers at the end of the semester. More emphasis is placed on varying voltage sources than in the earlier physics course. Measuring the frequency response of circuit elements with the two Rigol instruments becomes an important component of lab activities. This type of measurement is a perfect candidate to be automated.

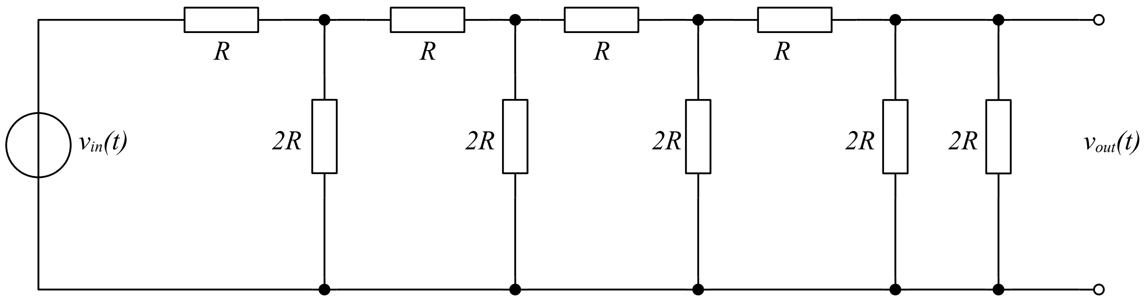


Figure 3: The R2R circuit. In the lab activity students use 10 kΩ and 20 kΩ resistors.

Class Activities

Initial Activity

The second lab introduces students to the use of the RFG and the DSP for characterizing circuits. Students learn how to optimize display on the DSP to use the measurements functions available. At the end of the lab, the frequency response (FR) of a simple circuit is measured over roughly seven decades with 3 points per decade. The circuit used is a simple *R2R* resistor ladder shown in Figure 3. Students use the RFG to provide a sinusoidal oscillating input voltage. Data recorded from the DSP are the amplitudes (either v_{pp} or v_{rms}) of both the input and output signals as well as the phase difference between the input and output signals. This data is used to display the FR of the output [using $v_{out}(f = 1\text{ kHz})$ as a reference] from 1 Hz to 3 MHz with 3 points per decade. This measurement is not automated. Data is collected by hand. Students change the signal frequency and record the data from DSP for each frequency. Figure 4 shows the FR of the circuit. The output signal begins to significantly roll off above 100 kHz. Dropping below the 3 dB level between the 220 kHz and 460 kHz level. Students will repeat this measurement using the x10 feature of the scope and probes to see the affect that has on the output.

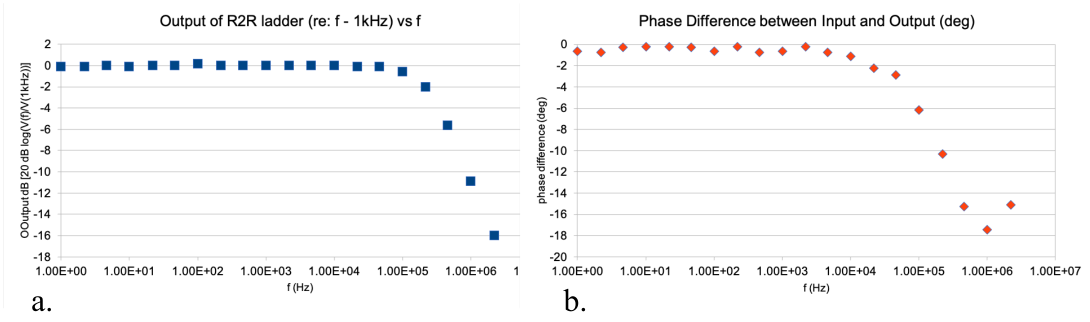


Figure 4: The frequency response of the R2R circuit found in the second lab activity.

Automation Activity

The third lab reuses the FR measurement of the circuit in Figure 3 to introduce students to the tools and software available for automating data acquisition. This activity has been used in the last two spring semesters. Each of these courses had small enrollments (5 students in 2022 and 4 in 2023). The students in the spring 2022 semester had experience with the Python programing language while those in the 2023 semester had little to none. The course uses a Jupyter Notebook as the platform for coding and reporting results.

Each lab activity is preceded by a set of pre-lab questions. To build more familiarity with Python, some of these questions are presented as coding activities starting with the spring 2023 semester. In these early activities, students are presented with nearly complete code blocks needing very little editing to complete the task. This is meant to help build their understanding of the code.

In the first activity, students are given code that generates I-V curves for a resistor and a constant current source. This is meant to provide example code that produces a plot. In the second they are asked to a plot a voltage given as $v(t) = V_0 \cos(\omega t)$, where $\omega = 10\ rad/s$. They are not given the code to answer this question. The idea being that they will use code from the first lab modified for the second. They are also provided with the code to generate the frequencies that will provide evenly spaced data in a plot that uses a logarithmic scale on the horizontal axis. This is used to complete the FR measurement of the circuit in Figure 3.

Automation is introduced in the third lab. Revisiting the FR measurement from the second lab, students are asked in the pre-lab questions to develop code calculating the frequencies needed to repeat the measurement with 10 points per decade with the frequencies evenly spaced on a semi-log plot. Students are not expected to develop the code to make this measurement. Students are provided with the code given in the Appendix A below. Appendix B provides a package of two functions that are used to optimize the horizontal and vertical scales on the DSP. Portions of this code will be discussed here in detail.

The first cell of the notebook (Figure A 1) imports packages needed for communications through the USB as well as the storage and display of the date. The *pyvisa* package provides the USB communication with the DSP and RFG. The *scale_scope* package (Appendix B) provides code to adjust the vertical and horizontal scales of the DSP to provide the most accurate measurements. The second cell (Figure A 1) identifies any instruments connected via USB. The third cell (Figure A 2) sets up the communication with each connected devise. The communication with the DSP is associated with an object labeled *scope* while the RFG is associated with an object labeled *funcgen*. Each of the instruments is reset to the factory settings when it is defined. This is done to ensure there are no lingering settings on either devise from previous use. It is stressed in class that this is necessary to assure the integrity of each measurement.

The next two cells (Figure A 4 and Figure A 3) in the notebook define initial settings for the DSP and then the RFG. In the first cell, initial vertical and horizontal scales are set for channels 1 and 2 of the DSP. Each of the channels are turned on with the coupling set to DC and probes set to X1 scaling. The scope is set to trigger off channel 2 as this is measuring the large input signal. Finally, the initial horizontal scale is set to 1 ms/div. The second cell initiates the RFG. Channel 1 of the RFG is turned on with an output of 10 Vpp, no offset voltage, and 1.5 kHz signal.

The next cell (Figure A 5 and Figure A 6) contains the code that will complete the measurement. The first three commands display measurements on the screen of the DSP. This is not strictly necessary but does provide validation of the measurements. The three quantities measured are the peak-to-peak voltage measurements on channels 1 and 2 along with the phase difference between the two channels. The next section of code initiates arrays to hold the three measured quantities and the frequencies used. The red text that follows is a placeholder for the calculation necessary to populate the frequency array. Students are expected to use the code developed in the pre-lab assignment in this section. The *for* loop that follows, adjusts the frequency of the input signal from the RFG, calls functions that adjust the DSP display to provide the best data, and records the three measurements of interest. The results of each measurement are displayed before moving on to the next.

The last two cells (Figure A 7) produce two a semi-log plots of the data and shut down the communication through the USB. The first plot is the decibel level of the output signal at each frequency relative to the 1 kHz output signal. The second is the phase difference between the input and output signals at each frequency.

Figure 5 shows the FR curves for the R2R circuit from Figure 3 measured using the Jupyter Notebook code given in Appendix A. Note that the notebook takes about 15 minutes to complete the entire measurement and display the results. The much sparser data set displayed in Figure 4 took twice as long to acquire. This Jupyter Notebook will allow students to explore in more detail the FR of RLC circuits as well as active filters developed in later labs.

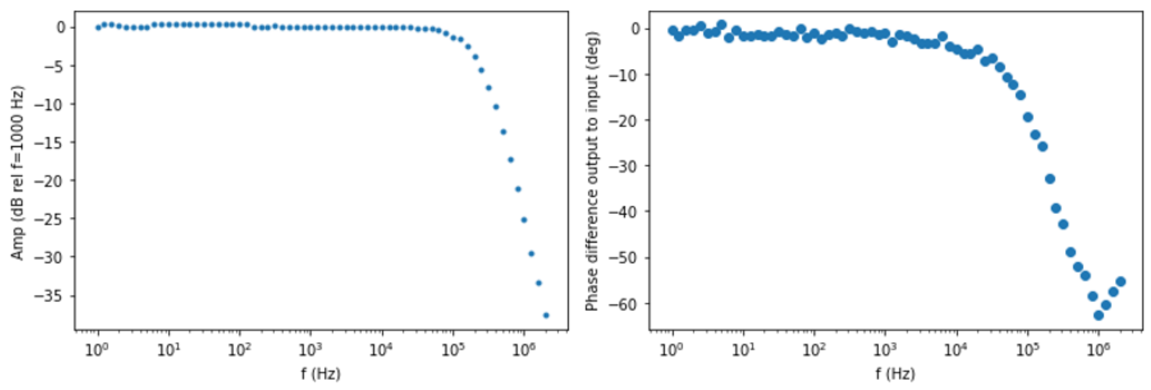


Figure 5: The frequency response of the R2R circuit measured using the Jupyter Notebook.

Discussion

This activity was introduced in the Electronics course during the spring semester of 2022. That course and the current course (spring 2023) were very small: 5 and 4 students respectively. The students in the 2022 course were somewhat proficient coding in Python before taking Electronics. They were able to use and modify the Jupyter Notebook without much prompting during later lab activities.

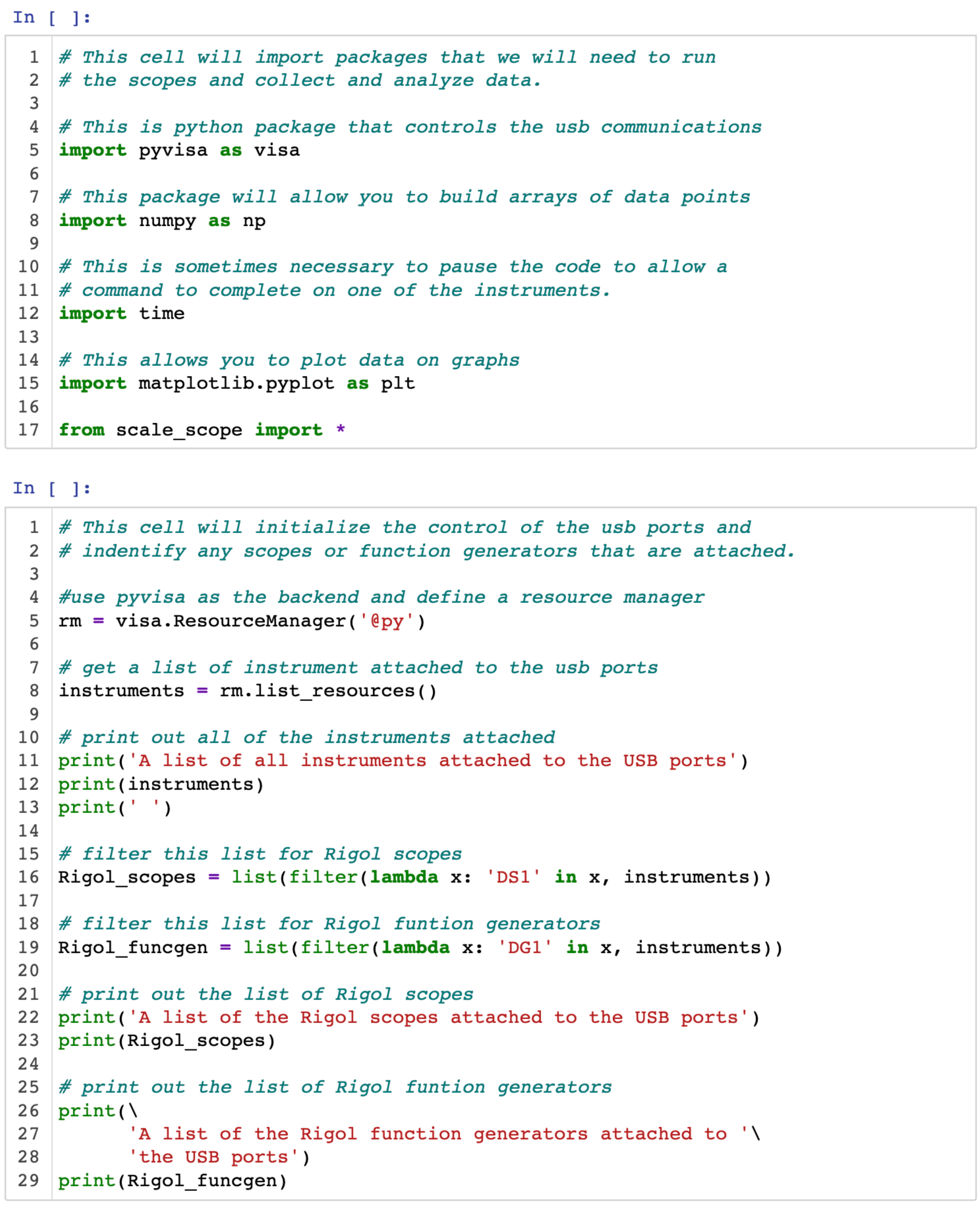
The students in the 2023 course have very little experience coding. These students are likely to be more representative of future students. This has suggested some modifications to the course. One has been mentioned already, pre-lab questions have been introduced that ask the students to use Python code to calculate and display information. This has been mildly successful. Students are still struggling with the flow of the code and what is happening.

This course is not intended to be a coding course. The focus of the course is designing, building, and testing electronic circuits. Use of the Jupyter Notebooks is intended as an introduction to some of the tools they will use in Robotics 1 and 2. Most students in the Electronics course will have completed at least one semester of computer science using C++ as the coding language. The robotics courses will use mainly Python for coding so this is an important first look at the language.

Going forward, labs for this course will be redesigned to take advantage of the computer control that is available. The use of python will be scaffolded and incorporated throughout the course to better introduce different aspects. New labs will be developed to introduce the computer control more gradually than is currently done. Students currently jump in with the code in Appendices A and B below. New labs will introduce the control sequences for each of the different instruments individually. Computer control will be paired with the manual operation of front panel controls to highlight the connections operations. This also should give students a better understanding of the code used to make measurements.

Appendix A: Jupyter Notebook for Data Acquisition

The code that follows is provided to the students at the start of the third lab. Students need to modify the code to provide the frequency values used in the measurement.



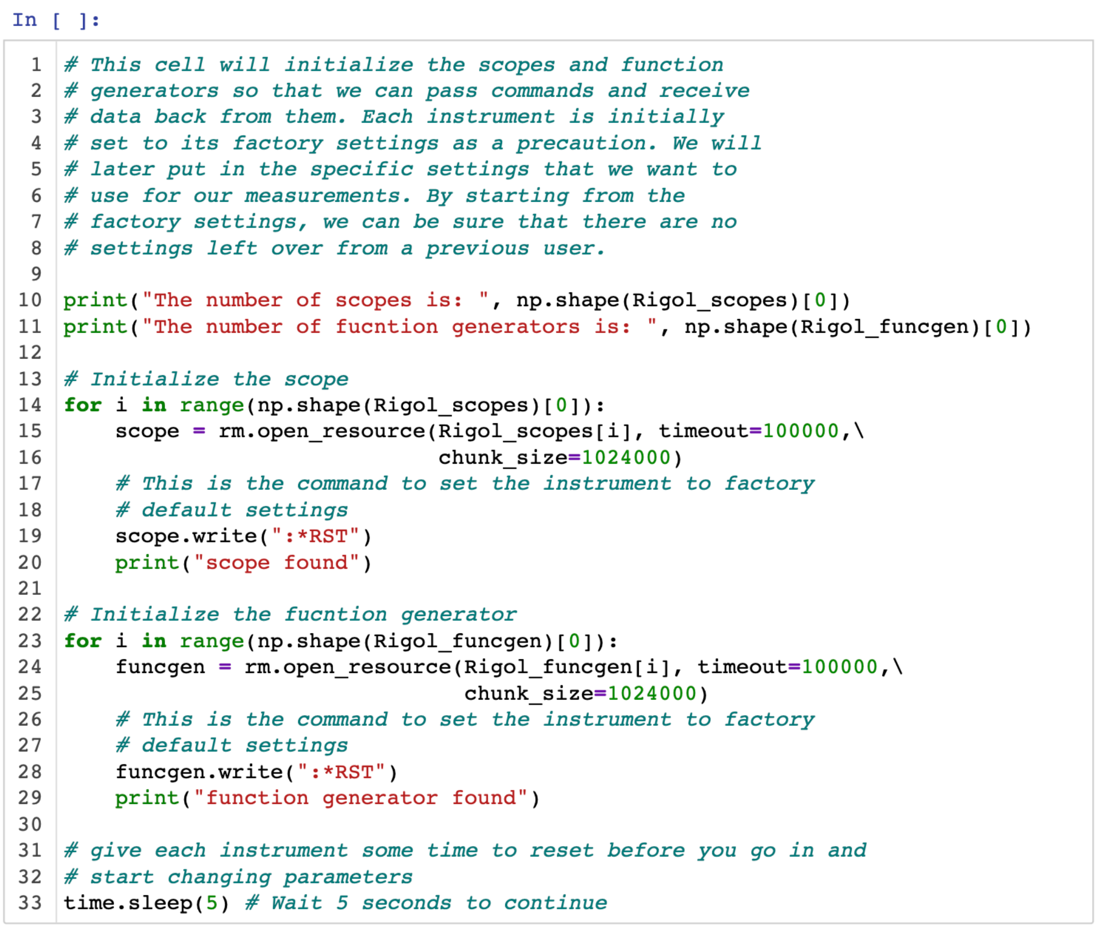
In [ ]:

```python
# This cell will import packages that we will need to run
# the scopes and collect and analyze data.

# This is python package that controls the usb communications
import pyvisa as visa

# This package will allow you to build arrays of data points
import numpy as np

# This is sometimes necessary to pause the code to allow a
# command to complete on one of the instruments.
import time

# This allows you to plot data on graphs
import matplotlib.pyplot as plt

from scale_scope import *
```

In [ ]:

```python
# This cell will initialize the control of the usb ports and
# indentify any scopes or function generators that are attached.

#use pyvisa as the backend and define a resource manager
rm = visa.ResourceManager('@py')

# get a list of instrument attached to the usb ports
instruments = rm.list_resources()

# print out all of the instruments attached
print('A list of all instruments attached to the USB ports')
print(instruments)
print(' ')

# filter this list for Rigol scopes
Rigol_scopes = list(filter(lambda x: 'DS1' in x, instruments))

# filter this list for Rigol funtion generators
Rigol_funcgen = list(filter(lambda x: 'DG1' in x, instruments))

# print out the list of Rigol scopes
print('A list of the Rigol scopes attached to the USB ports')
print(Rigol_scopes)

# print out the list of Rigol funtion generators
print(\
      'A list of the Rigol function generators attached to '\
      'the USB ports')
print(Rigol_funcgen)
```

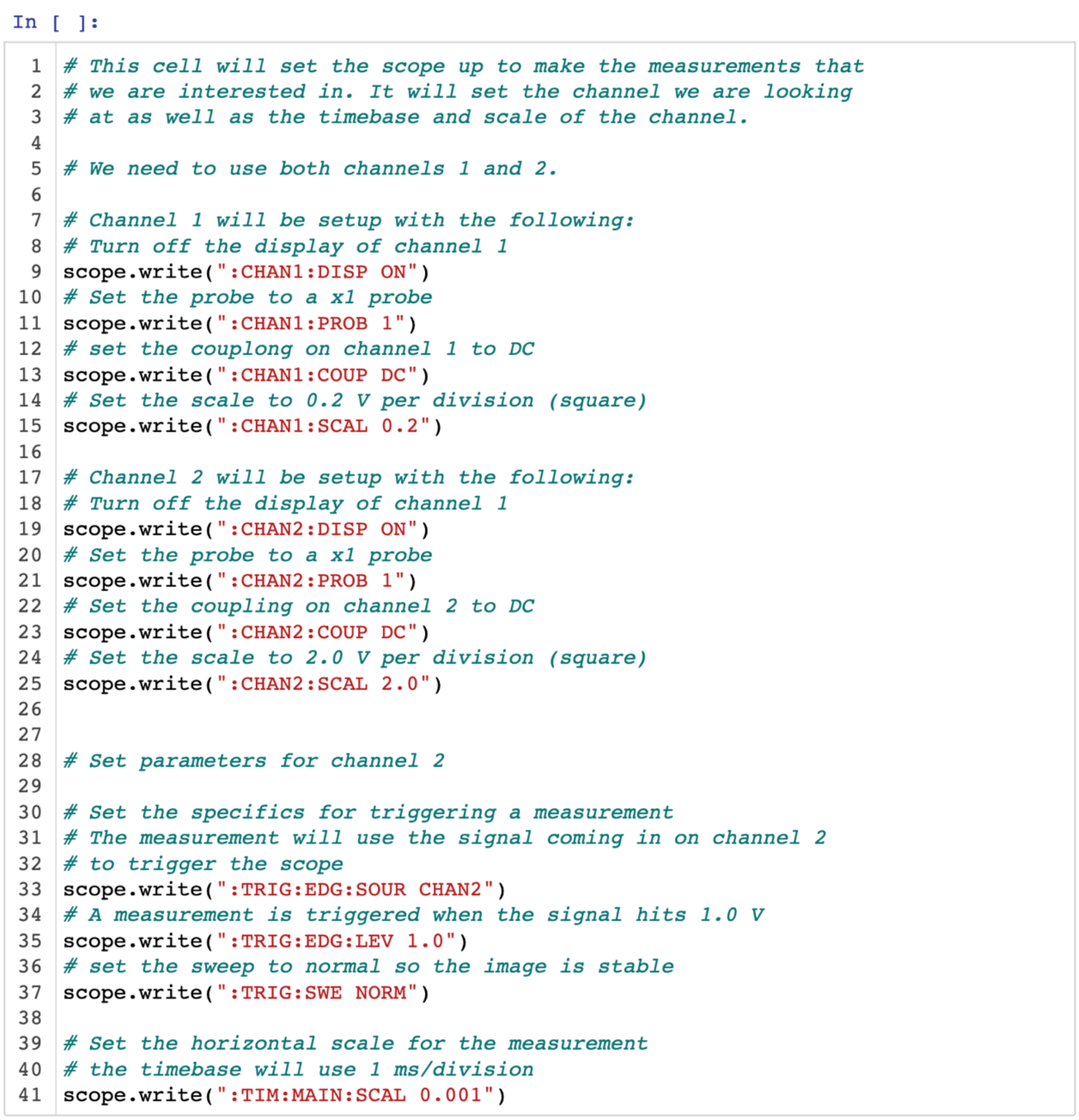
Figure A 1: The first two cells of the Jupyter Notebook given to students. The first cell imports packages while the second sets up the communication through the USB ports.



```
In [ ]:
```

```python
# This cell will initialize the scopes and function
# generators so that we can pass commands and receive
# data back from them. Each instrument is initially
# set to its factory settings as a precaution. We will
# later put in the specific settings that we want to
# use for our measurements. By starting from the
# factory settings, we can be sure that there are no
# settings left over from a previous user.

print("The number of scopes is: ", np.shape(Rigol_scopes)[0])
print("The number of fucntion generators is: ", np.shape(Rigol_funcgen)[0])

# Initialize the scope
for i in range(np.shape(Rigol_scopes)[0]):
    scope = rm.open_resource(Rigol_scopes[i], timeout=100000,\
                             chunk_size=1024000)
    # This is the command to set the instrument to factory
    # default settings
    scope.write(":*RST")
    print("scope found")

# Initialize the fucntion generator
for i in range(np.shape(Rigol_funcgen)[0]):
    funcgen = rm.open_resource(Rigol_funcgen[i], timeout=100000,\
                               chunk_size=1024000)
    # This is the command to set the instrument to factory
    # default settings
    funcgen.write(":*RST")
    print("function generator found")

# give each instrument some time to reset before you go in and
# start changing parameters
time.sleep(5) # Wait 5 seconds to continue
```

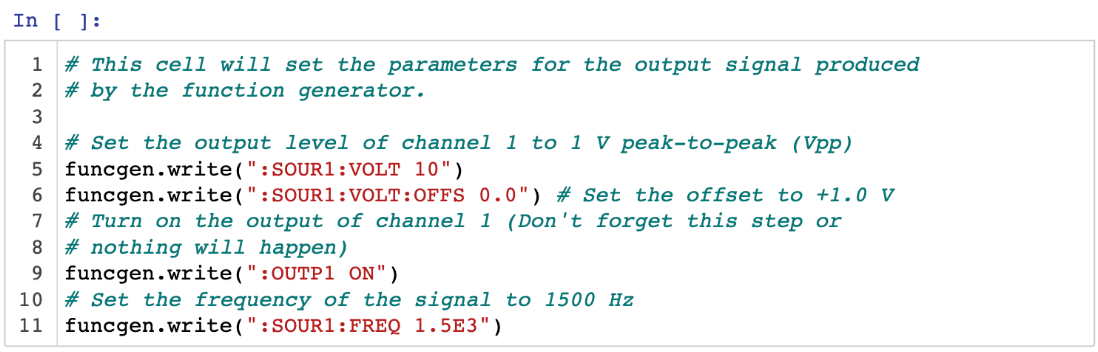
Figure A 2: This is the third cell of the notebook. It defines objects for both the DSP (scope) and RFG (funcgen) that will be used to send commands to and read data from each instrument. Each instrument is reset to factory settings in this cell to provide a common starting point for each measurement.



In [ ]:

```python
# This cell will set the scope up to make the measurements that
# we are interested in. It will set the channel we are looking
# at as well as the timebase and scale of the channel.

# We need to use both channels 1 and 2.

# Channel 1 will be setup with the following:
# Turn off the display of channel 1
scope.write(":CHAN1:DISP ON")
# Set the probe to a x1 probe
scope.write(":CHAN1:PROB 1")
# set the couplong on channel 1 to DC
scope.write(":CHAN1:COUP DC")
# Set the scale to 0.2 V per division (square)
scope.write(":CHAN1:SCAL 0.2")

# Channel 2 will be setup with the following:
# Turn off the display of channel 1
scope.write(":CHAN2:DISP ON")
# Set the probe to a x1 probe
scope.write(":CHAN2:PROB 1")
# Set the coupling on channel 2 to DC
scope.write(":CHAN2:COUP DC")
# Set the scale to 2.0 V per division (square)
scope.write(":CHAN2:SCAL 2.0")


# Set parameters for channel 2

# Set the specifics for triggering a measurement
# The measurement will use the signal coming in on channel 2
# to trigger the scope
scope.write(":TRIG:EDG:SOUR CHAN2")
# A measurement is triggered when the signal hits 1.0 V
scope.write(":TRIG:EDG:LEV 1.0")
# set the sweep to normal so the image is stable
scope.write(":TRIG:SWE NORM")

# Set the horizontal scale for the measurement
# the timebase will use 1 ms/division
scope.write(":TIM:MAIN:SCAL 0.001")
```

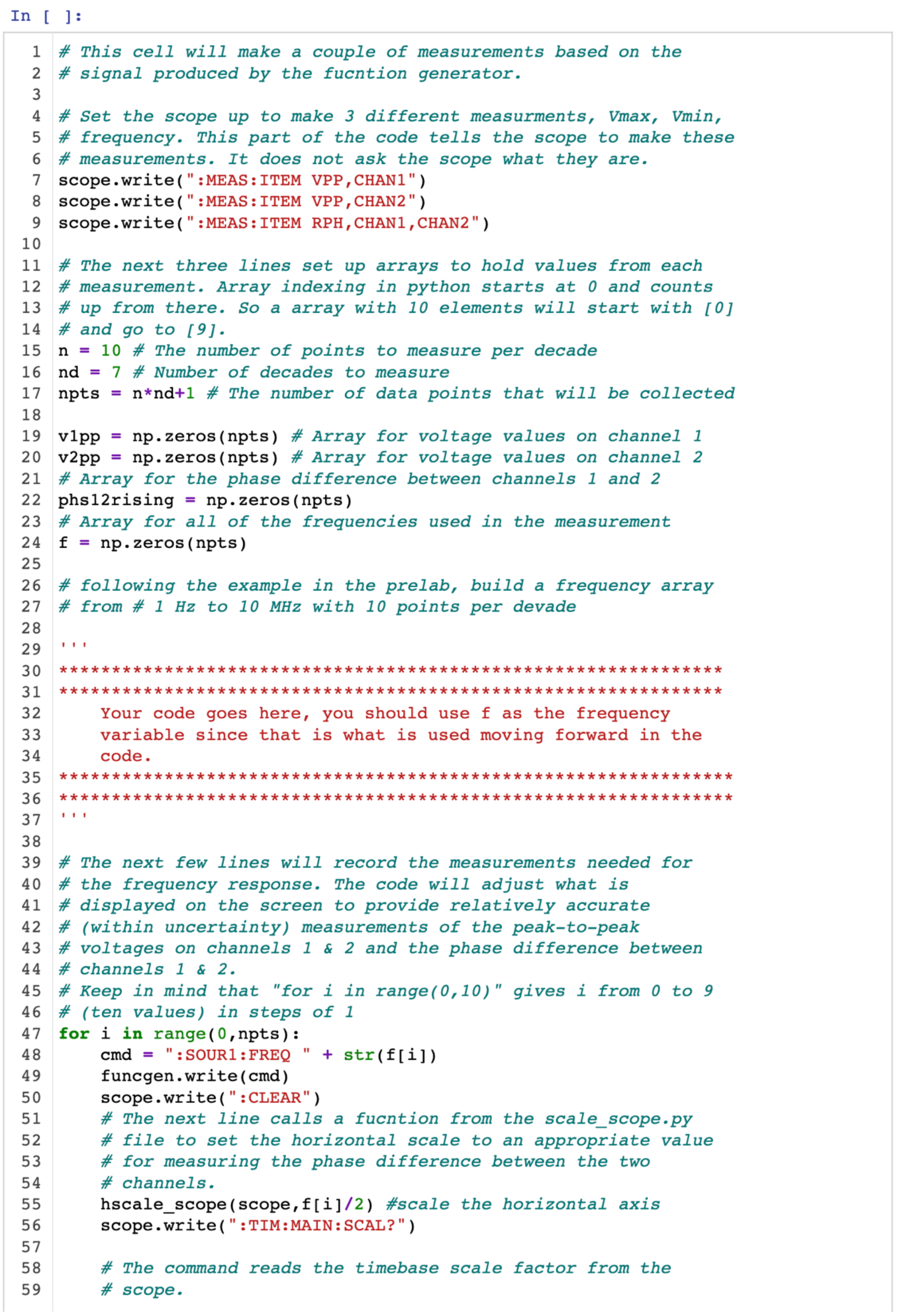
Figure A 4: This cell provides the initial set up for the DSP.



In [ ]:

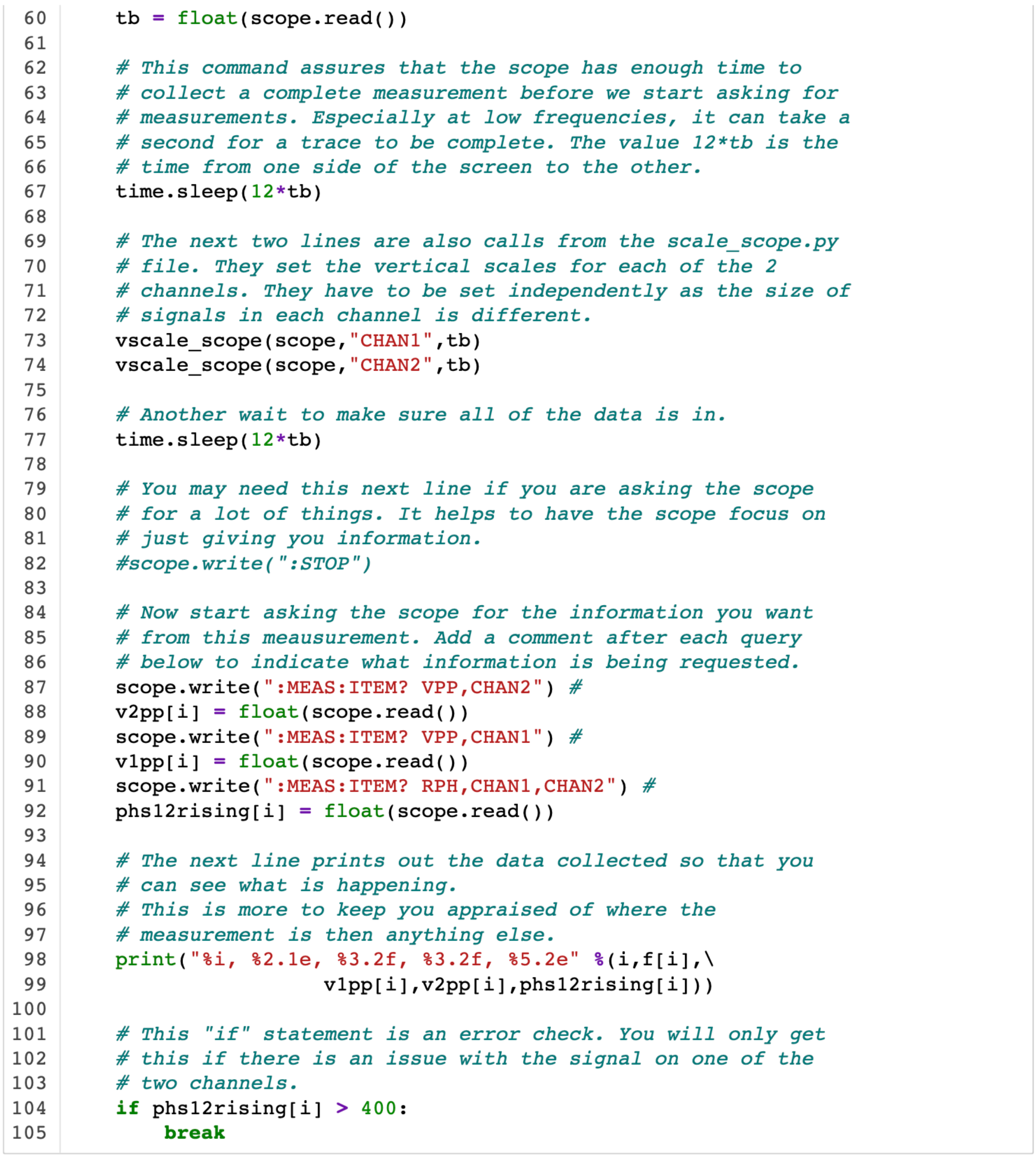
```python
# This cell will set the parameters for the output signal produced
# by the function generator.

# Set the output level of channel 1 to 1 V peak-to-peak (Vpp)
funcgen.write(":SOUR1:VOLT 10")
funcgen.write(":SOUR1:VOLT:OFFS 0.0") # Set the offset to +1.0 V
# Turn on the output of channel 1 (Don't forget this step or
# nothing will happen)
funcgen.write(":OUTP1 ON")
# Set the frequency of the signal to 1500 Hz
funcgen.write(":SOUR1:FREQ 1.5E3")
```

Figure A 3: This cell provides the initial set up for the RFG.


```
In [ ]:
```

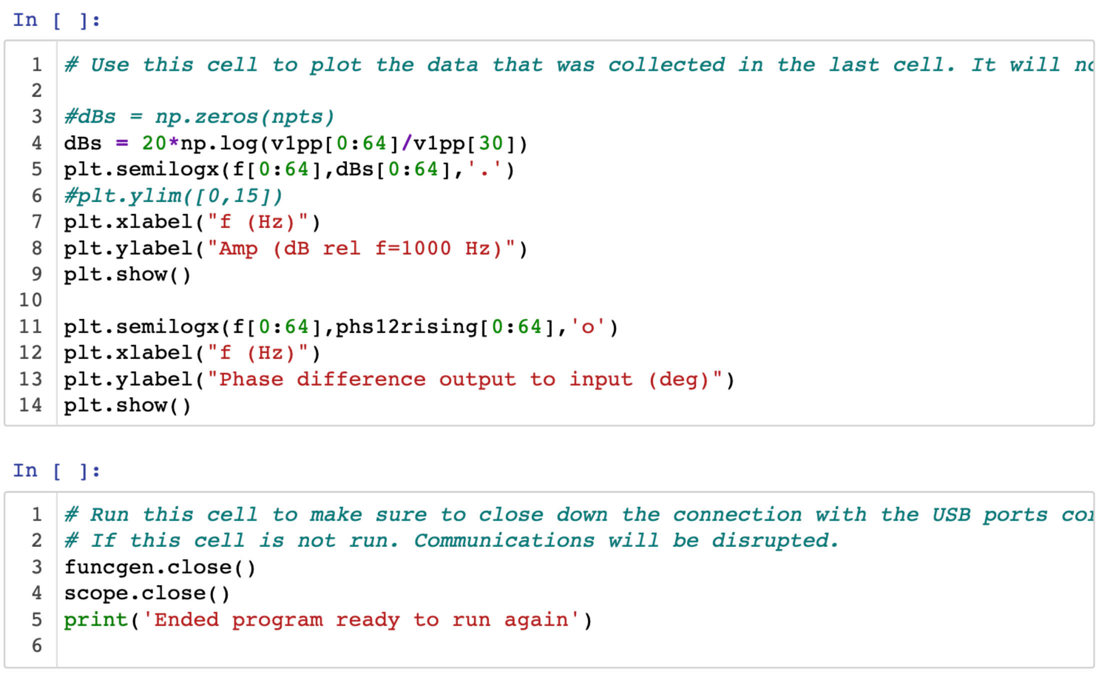
```python
# This cell will make a couple of measurements based on the
# signal produced by the fucntion generator.

# Set the scope up to make 3 different measurments, Vmax, Vmin,
# frequency. This part of the code tells the scope to make these
# measurements. It does not ask the scope what they are.
scope.write(":MEAS:ITEM VPP,CHAN1")
scope.write(":MEAS:ITEM VPP,CHAN2")
scope.write(":MEAS:ITEM RPH,CHAN1,CHAN2")

# The next three lines set up arrays to hold values from each
# measurement. Array indexing in python starts at 0 and counts
# up from there. So a array with 10 elements will start with [0]
# and go to [9].
n = 10 # The number of points to measure per decade
nd = 7 # Number of decades to measure
npts = n*nd+1 # The number of data points that will be collected

v1pp = np.zeros(npts) # Array for voltage values on channel 1
v2pp = np.zeros(npts) # Array for voltage values on channel 2
# Array for the phase difference between channels 1 and 2
phs12rising = np.zeros(npts)
# Array for all of the frequencies used in the measurement
f = np.zeros(npts)

# following the example in the prelab, build a frequency array
# from # 1 Hz to 10 MHz with 10 points per devade

'''
****************************************************************
****************************************************************
    Your code goes here, you should use f as the frequency
    variable since that is what is used moving forward in the
    code.
*****************************************************************
*****************************************************************
'''

# The next few lines will record the measurements needed for
# the frequency response. The code will adjust what is
# displayed on the screen to provide relatively accurate
# (within uncertainty) measurements of the peak-to-peak
# voltages on channels 1 & 2 and the phase difference between
# channels 1 & 2.
# Keep in mind that "for i in range(0,10)" gives i from 0 to 9
# (ten values) in steps of 1
for i in range(0,npts):
    cmd = ":SOUR1:FREQ " + str(f[i])
    funcgen.write(cmd)
    scope.write(":CLEAR")
    # The next line calls a fucntion from the scale_scope.py
    # file to set the horizontal scale to an appropriate value
    # for measuring the phase difference between the two
    # channels.
    hscale_scope(scope,f[i]/2) #scale the horizontal axis
    scope.write(":TIM:MAIN:SCAL?")

    # The command reads the timebase scale factor from the
    # scope.
```

Figure A 5: The first half of the cell that runs the measurement producing the FR.



```python
    tb = float(scope.read())

    # This command assures that the scope has enough time to
    # collect a complete measurement before we start asking for
    # measurements. Especially at low frequencies, it can take a
    # second for a trace to be complete. The value 12*tb is the
    # time from one side of the screen to the other.
    time.sleep(12*tb)

    # The next two lines are also calls from the scale_scope.py
    # file. They set the vertical scales for each of the 2
    # channels. They have to be set independently as the size of
    # signals in each channel is different.
    vscale_scope(scope,"CHAN1",tb)
    vscale_scope(scope,"CHAN2",tb)

    # Another wait to make sure all of the data is in.
    time.sleep(12*tb)

    # You may need this next line if you are asking the scope
    # for a lot of things. It helps to have the scope focus on
    # just giving you information.
    #scope.write(":STOP")

    # Now start asking the scope for the information you want
    # from this meausurement. Add a comment after each query
    # below to indicate what information is being requested.
    scope.write(":MEAS:ITEM? VPP,CHAN2") #
    v2pp[i] = float(scope.read())
    scope.write(":MEAS:ITEM? VPP,CHAN1") #
    v1pp[i] = float(scope.read())
    scope.write(":MEAS:ITEM? RPH,CHAN1,CHAN2") #
    phs12rising[i] = float(scope.read())

    # The next line prints out the data collected so that you
    # can see what is happening.
    # This is more to keep you appraised of where the
    # measurement is then anything else.
    print("%i, %2.1e, %3.2f, %3.2f, %5.2e" %(i,f[i],\
                     v1pp[i],v2pp[i],phs12rising[i]))

    # This "if" statement is an error check. You will only get
    # this if there is an issue with the signal on one of the
    # two channels.
    if phs12rising[i] > 400:
        break
```

Figure A 6: The second half of the cell that runs the measurement producing the FR.



```
In [ ]:
# Use this cell to plot the data that was collected in the last cell. It will nc

#dBs = np.zeros(npts)
dBs = 20*np.log(v1pp[0:64]/v1pp[30])
plt.semilogx(f[0:64],dBs[0:64],'.')
#plt.ylim([0,15])
plt.xlabel("f (Hz)")
plt.ylabel("Amp (dB rel f=1000 Hz)")
plt.show()

plt.semilogx(f[0:64],phs12rising[0:64],'o')
plt.xlabel("f (Hz)")
plt.ylabel("Phase difference output to input (deg)")
plt.show()
```

```
In [ ]:
# Run this cell to make sure to close down the connection with the USB ports coı
# If this cell is not run. Communications will be disrupted.
funcgen.close()
scope.close()
print('Ended program ready to run again')
```

Figure A 7: The last two cells of the notebook. The first cell plots the decibel amplitude (relative to the amplitude of the 1 kHz output) of the output voltage. The second performs a clean shutdown of the communications with the DSP and RFG.

Appendix B: scale_scope.py

This code is used to adjust the display of the DSP to provide the most accurate measurements of the measured signals.

```python
#!/usr/bin/env python3
# -*- coding: utf-8 -*-0

# This package will allow you to build arrays of data points
import numpy as np

# This is sometimes necessary to pause the code to allow a command to
# complete on one of the instruments.
import time


def vscale_scope(scope, chn, tb):
    # This function will set the vertical scale on the scope.
    # It will work off the peak-to-peak voltage measurement for the
    # current signal. Starting from the current precision setting, it
    # will adjust the voltage scale until the signal just fits on the
    # screen.

    # The DS1074 has 13 different vertical scale settings available
    # The lowest (most precise) scale is dV = 1 mV/div.
    # The highest (least precises) scale is dV = 10 V/div.
    # Each of these is set for a x1 probe and should be adjusted
    # otherwise.

    # Set up an array with the 13 possible horizontal scales available.
    vscale = np.zeros(13)
    for i in range(0,4):
        vscale[i*3+1]=5*np.power(10.0,-i)
        vscale[i*3+2]=2*np.power(10.0,-i)
        vscale[i*3+3]=1*np.power(10.0,-i)
    vscale[0]=10.

    # Add the channel designation into each of the root commands.
    # These are the commands used to query settings and measure
    # peak-to-peak voltage on the scope.

    # Query the scale on the channel indicated by "chn".
    cmd_qscl = ":"+chn+":SCAL? "

    # Base for the command setting the scale. Below the particular
    # scale to be used is appended to this string.
    cmd_setscl = ":"+chn+":SCAL "

    # Turn on the measurement of vpp for "chn". This shows the result
    # on the front panel of the scope.
    cmd_mvpp = ":MEAS:ITEM VPP," + chn

    # Query vpp on "chn".
    cmd_qvpp = ":MEAS:ITEM? VPP," + chn

    time.sleep(14*tb+1) # make sure to collect one full scane of data

    # read in the current vpp level
    scope.write(cmd_mvpp)
    scope.write(cmd_qvpp)
    vpp = float(scope.read())

    # ask for the current scale
    scope.write(cmd_qscl)
```

Figure B 1: Part of the code used to optimize the scale of the DSP for recording measurements.

```
    vscale_cur = float(scope.read())
    jj=0

    # Find the current scale in the list of possible scales
    while not((0.9*vscale[jj]<vscale_cur) and (vscale_cur<1.1*vscale[jj])):
        jj+=1

    # If the scale is apporpriate return with no change.
    if (vpp<8*vscale_cur) and (4*vscale_cur < vpp):
        return

    # This will run if the scale is too large
    while vpp<4*vscale[jj]:
        jj+=1
        scope.write(cmd_setscl+str(vscale[jj]))
        time.sleep(14*tb+1)
        scope.write(cmd_qvpp)
        vpp = float(scope.read())

    # This will run is the scale is too small
    while vpp>8*vscale[jj]:
        jj-=1
        #print(jj)
        scope.write(cmd_setscl+str(vscale[jj]))
        time.sleep(14*tb+1)
        scope.write(cmd_qvpp)
        vpp = float(scope.read())

def hscale_scope(scope, f):
    # Rescale the horizontal axis to better display the time base

    # The DS1074 has 31 different horizontal scale settings available
    # The lowest (most precise) scale is dt = 5.0 ns/div.
    # The highest (least precises) scale is dV = 50.0 s/div.

    # Set an array with the 31 possible vertical scales available.
    tb = np.zeros(31)
    for i in range(0,10):
        tb[i*3]=5*np.power(10.0,1-i)
        tb[i*3+1]=2*np.power(10.0,1-i)
        tb[i*3+2]=1*np.power(10.0,1-i)
    tb[30]=5.*np.power(10.,-9)

    # Use the frequency of the current signal to determine which of
    # the scales is appropriate.
    j = 1
    while 12*tb[j]*f > 3:
        j += 1

    # Set the timebase on the scope.
    cmd = ":TIM:MAIN:SCAL " + str(tb[j])
    scope.write(cmd)

```

Figure B 2: The rest of the code used to set the vertical and horizontal scales on the DSP.

This code is imported into the Jupyter Notebooks used to automate measurements. It is designed to set the vertical and horizontal scales on the DSP for optimal data acquisition. Both scales follow a 1-2-5 progression with the vertical scale going from 1 mV/div up to 10 V/div and the horizontal going from 5 ns/div up to 50 s/div. The function *hscale_scope* sets the horizontal scale to display, using the frequency of the input signal, to display at least 3 cycles of the signal. The function *vscale_scope* sets the horizontal scale using the peak-to-peak voltage of the current signal to set the scale. The vertical scale is set such that the signal takes up more than 50% but less than 80% of the screen. This provides the best results when using the DSP to make the measurements necessary to characterize the FR of a circuit.

Bibliography

[1] EduEverything, Inc., "Home page," 2023. [Online]. Available: https://www.thenrc.org.

[2] B. Wessling, "The Robot report, Open Robotics developing Space ROS with Blue Origin, NASA," 12 02 2022. [Online]. Available: https://www.therobotreport.com/open-robotics-developing-space-ros/.

[3] M. J. a. G. Biggs, "Space ROS: The Future of Space Robotics," 2022. [Online]. Available: https://istcolloq.gsfc.nasa.gov/wp-content/uploads/2022/11/Space-ROS-NASA.pdf.

[4] C. A. Meyer, Basic Electronics: An Introduction for Science Students Second Edition, Curtis A. Meyer, 2016.