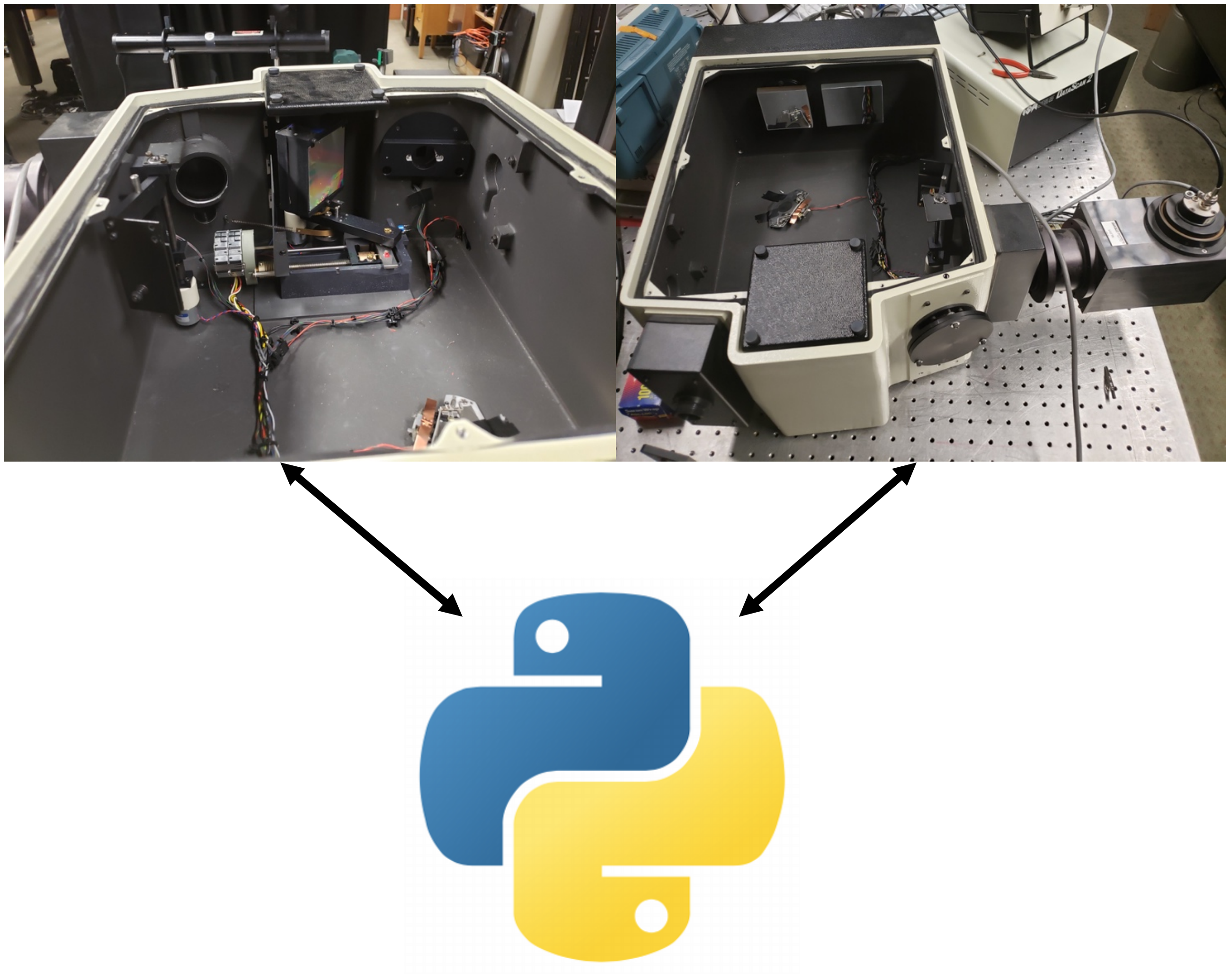
Control of Jobin-Yvon HR460 Spectrometer Using Python3

Justin deMattos, Dr. Bryan Boggs, Dr. Nima Dinyari
University of Oregon
Department of Physics



Software and Device User Manual

Contents

Device Introduction

Parts Included:

- Jobin-Yvon HR460 Base Unit
  - Holds all optical devices and contains detector attached in black box on the side
- DataScan Driver Unit
  - Runs the spectrometer and attaches to the base unit via a 25-pin serial port.
  - The 25-pin serial port only containing 9 pins is attached to the base unit while the other side of the cable gets connected to…
  - A 9-pin serial cable is connected to a monochromator control port and the other side, a 25-pin serial port is connected to the computer.
- Front Entrance slit
  - Controlled via a stepper motor
- Shutter
  - Currently broken. Did not respond to commands and is taped to the bottom of the base unit.
- Primary Mirror
  - Curved mirror used to collimate the incoming light
- 800 lines/mm grating
  - Used as primary grating. Device initialized to this grating.
- 600 lines/mm grating
  - On the flipside of the primary grating. Can be switched to using setGrating()
- Secondary Mirror
  - Curved mirror used to focus light onto the exit mirror
- Exit mirror
  - Can be set to the side “s” position for data collection using the attached detector at the side exit or set to front “f” position for data collection using another detector at the front exit slit.
- Side Exit Slit
  - Exit slit on the side used for detector data collection
- Detector Control Box
  - Used to provide power to the detector
- Nitrogen-cooled CCD Detector
  - Not currently supported by code

Device Introduction

Spectrometer Physics

The Jobin-Yvon HR460 spectrometer is used for taking high resolution spectral data. Using the 800 lines/mm grating, the device can resolve down to approximately 0.1nm in spectral width. This spectrometer is designed in a Czerny-Turner configuration as shown in Figure 1 [1].

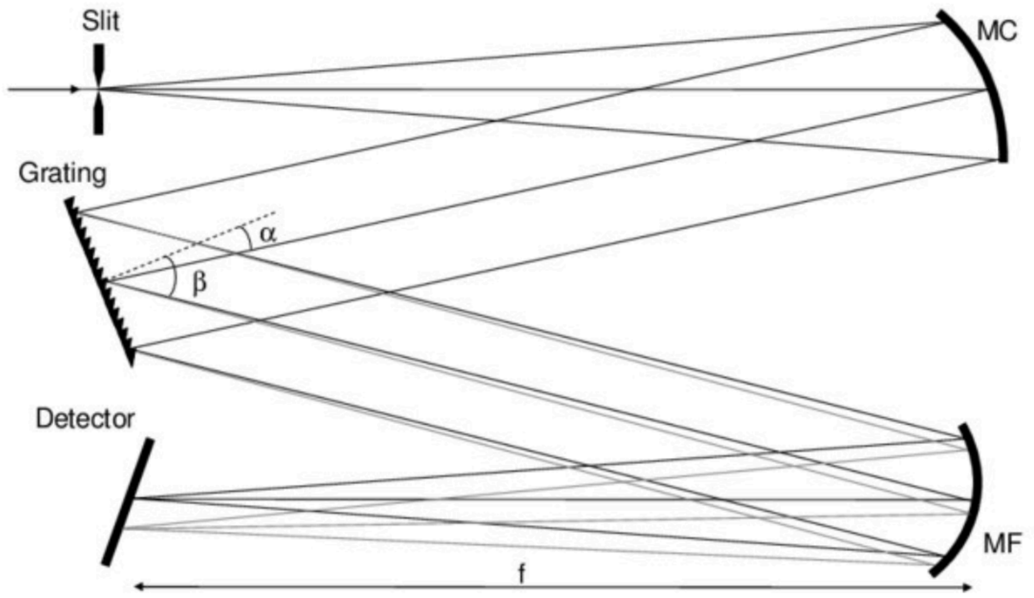


Figure 1. Czerny-Turner spectrometer configuration. MC and MF are the primary and secondary mirrors, respectively[1]. f is the focal length of the device, α is the angle of incidence and β is the diffraction angle[1].

Incoming light is to be focused down to the entrance slit opening with a lens that has an appropriate f/# corresponding to the spectrometer and grating specifications [2,3]. The user manual for the spectrometer that is used in this manual has an f/# of f/7 with a 58x58 mm^2 grating and f/5.3 with a 76x76 mm^2 grating [4]. The f/# for a lens can be calculated using equation 1 where f is the focal length of the lens and D is the diameter of the lens [5].

$$f/\# = \frac{f}{D} \tag{1}$$

Using the correct lens, the spot size can then be calculated for the correct slit opening. If using a gaussian beam, equation 2 can be used to calculate the radius of the spot size W_f after traveling through the focusing lens [6]. W_o is the original waist of the gaussian beam, λ is the peak wavelength of the beam and f is the focal length of the lens.

$$W_f = \frac{\lambda f}{\pi W_o} \tag{2}$$

By reducing the slit size to the spot size of the beam and using the correct f/# lens, the primary mirror (MC in figure 1) is filled with the incoming light after spreading from the focal point at the center of the slits. By filling the mirror, the light is collimated upon reflection to the grating. If the mirror is filled, the grating is filled, and all wavelengths detectable by the device can be scanned over with great accuracy.

After reflecting off of the primary mirror, the light strikes the diffraction grating that is also a reflective surface. If polychromatic light is shown on the grating, the different wavelengths will reflect at different angles off of the grating. Figure 2 shows the scenario of polychromatic light on a grating and the various orders of reflection from it [7].

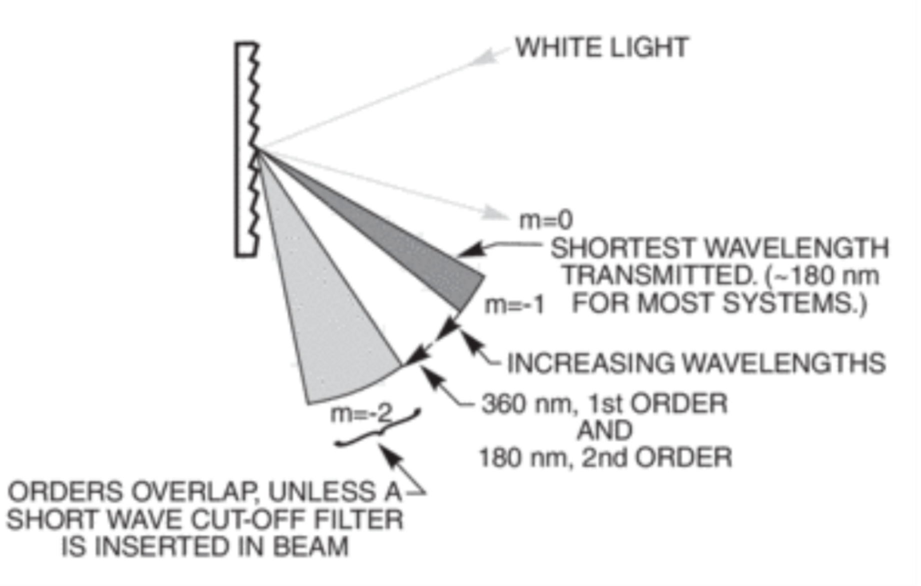

WHITE LIGHT
m=0
SHORTEST WAVELENGTH TRANSMITTED. (~180 nm FOR MOST SYSTEMS.)
m=-1
INCREASING WAVELENGTHS
360 nm, 1st ORDER AND 180 nm, 2nd ORDER
m=-2
ORDERS OVERLAP, UNLESS A SHORT WAVE CUT-OFF FILTER IS INSERTED IN BEAM

Figure 2. White light is reflected off of a grating and separated into different wavelengths that are reflected at different angles from the grating [7]. Note that the wavelengths given in this image do not correspond to the expected wavelengths in the spectrometer in this manual.

The resolution of the device for these different reflection angles of different wavelengths depends on the frequency of line spacing (lines/mm) and the blaze angle given by the diffraction gratings. A higher frequency of line spacing will improve resolution. The resolution of the spectrometer discussed in this manual is given as ≤0.00625 nm with a 1200 lines/mm grating [4]. The resolution for other gratings can be calculated using equation 3.

$$New\ Resolution = \frac{(1200\ lines/mm)(0.00625\ nm)}{New\ Grating\ (lines/mm)} \qquad (3)$$

With the polychromatic light being split at different angles from the grating, the spectrometer can move the grating, rotating it about the axis straight through the center, to essentially scan over the different wavelengths reflected off of the grating. The reflections are sent to the secondary mirror (MF) in figure 1, where it is reflected and focused down to a point on the exit mirror. Figure 1 does not show an exit mirror because of the simplicity of their design, however, if using the detector in the spectrometer discussed in this manual, the exit mirror will be used to focus the light to the detector.

Communication Setup

Connecting to a Computer

The DataScan driver unit is used to connect a computer to the spectrometer. The DataScan unit uses a serial port to communicate with the computer. If no serial port is available, a USB serial port adapter can be used for communication with the device. This is the setup assumed in this manual.

To check that the device is connected via USB, the following commands can be entered into the linux terminal to check for a connection to a USB:

```
cd /dev
ls
```

The first command moves the current working directory to the developer directory that can show all of the connections to the computer. The second command shows the user a list of all the connections to the computer. Since a USB device is being used, one of the folders in the list should appear as “ttyUSB_” where the underscore is some number depending on the USB port that the device is plugged into. For example, the “ttyUSB0” port was used in writing this manual. If the USB port folder does not appear, try connecting to a new port or try a new cable. Also be sure to check that the device is turned on.

Python Installation and Packages

The Spectrometer.py class used for communications in this manual requires the installation of Python 3 on the computer being used to communicate with the spectrometer. Python3 can be downloaded at this website: https://www.python.org/downloads/. The following command can also be entered from a linux terminal to install Python 3.6:

```
sudo apt-get install python3.6
```

To check that a version of Python 3 is installed on the computer, the following Linux command can be entered:

```
python3 --version
```

If a version number is returned, a version of Python 3 has successfully been installed on the computer. If an error is shown, it needs to be downloaded.

There are also a number of Python packages that need to be installed for Spectrometer.py to work on the computer. These packages include: struct, pyserial, and numpy. To install packages in Python, pip can be used from the terminal to install them. The following command will install new packages, where “[package]” should be replaced with the actual package name:

```
pip install [package]
```

If an error occurs relating to pip not being installed, you may have the wrong version of Python installed because any version of Python 3 greater than 3.4 comes with pip installed.

For the installation of the pyserial package, be sure that “pyserial” and not just “serial” is installed. An error will occur if “serial” is installed because of the classification of functions. If serial is installed pyserial cannot be installed over it and it must be removed before continuing. Everything should now be ready to start using the Spectrometer.py file designed for use of the Spectrometer.

Initialization and Startup

Getting Started

The Spectrometer class in the Spectrometer.py file has been written to streamline commands sent to the spectrometer and allow for the user to communicate with the device using Python. In order for this class to work, the user needs to navigate to the directory of the Spectrometer.py file using the Linux command line. Once in the directory of the file, the user can type the following to start a new Python program from the command line:

```
python3
```

The console should respond with a paragraph of text indicating that a new Python terminal has been opened and 3 “>>>” symbols should appear in place of the normal command line symbol. Once in the Python terminal, the Spectrometer class needs to be imported to run the device. The following Python code can be entered to import the Spectrometer class:

```
import Spectrometer
from Spectrometer import Spectrometer
```

This imports the Spectrometer class and all of its functions to the current Python program and allows for a Spectrometer object to be created. By creating a Spectrometer object, the link is made to the device using the specified USB port that the device is connected to. For example, the following code can be used to create a Spectrometer object, connecting to the device:

```
a=Spectrometer(‘ttyUSB0’)
```

In this case, “a” is just any variable that the user assigns, and “ttyUSB0” is the USB port directory to write to. The USB port name must be passed to the Spectrometer class constructor in order to open the link to the device. For more information on the Spectrometer constructor and it’s parameter, see the Spectrometer constructor description at the end of the manual and take a look at the example code for Spectrometer.py.

Once the link has been established, the device can then be turned on into the mode to allow programming commands to be received. The on() function is used to complete this step. This function flips the spectrometer into intelligent mode to allow programming commands, rather than the handheld keyboard commands, to be sent. The following is the example code used to call this function:

```
a.on()
```

Note that the "a" in this statement is the variable defined by the user as the Spectrometer object.

This function will display messages in the terminal informing the user of the status of the device as it goes through the process of turning it on. A "Ready!" message will display at the end if everything was successfully completed. If, a bad response was received, the terminal will display an error message with the spectrometer response and the Python program will close. The device will need to be manually reset if this occurs.

IMPORTANT: There is currently a bug in the code where it sometimes displays the "Ready!" message and then an error after it and exits the Python code. If this occurs, the spectrometer does NOT need to be restarted and a new Python terminal and a new link can be established to continue to the initialization as normal.

Initialization

There are 3 things that must be done in the initialization process before beginning any data collection:

1. Use the initialization command to initialize the device
2. Set the motor speed
3. Set the slit speed

1: All of the components of the device need to be initialized upon startup. This assures that all of the functions in the Spectrometer class run as they should, and that the device is in its starting position, as it may not be in its starting state after startup. To initialize the device, the following Python code can be entered using the Spectrometer class function initialize() after following the steps to get started:

```
a.initialize()
```

Once issued, the command for the spectrometer to initialize will be sent and the command line will respond with a message to communicate that initialization has begun. Initializing can take some time and nothing can be sent to the device while it is taking place, so many "Initializing…" statements will be printed by the console until the initialization process is complete. In the process, the exit mirror is set to the front position, the grating is moved back to position 208701, the 800 mm grating is selected, and the slits are closed. If an error occurs anywhere in the process, an error will be shown in the terminal with the response of the device, and the Python program will close. The device will then need to be restarted manually.

Note, the “a” in the line is the variable used to create the Spectrometer object and can be whatever the user defined.

2: The speed of the motor needs to be set before running it. From the appendix given in the programming manual for the device, the minimum frequency is given as 2560 steps/second, the maximum frequency is given as 5500 steps/second, and the ramp time is defined as 2000 milliseconds. These are the default parameters when using the setMotorSpeed() function in the Spectrometer class, but other parameters can be used, with caution. See the description at the end of the manual and the example code for more information on the function. Enter the following to set the motor speed:

```
a.setMotorSpeed()
```

Again, note that the “a” used in this line is the variable that the user defined as the Spectrometer object. Also, this is assuming that the spectrometer has been plugged into the “Monochromator 1” input on the DataScan driver. If the device is plugged into the “Monochromator 2” input, a “1” needs to be sent to the function.

If something is wrong with the command and the device does not accept the input or is not in a state to accept commands, the command line will issue an error statement giving the response of the device and the Python program will close. The device will then need to be turned off and back on again to restart.

3: Just like the motor speed, the slit speed also needs to be set. The speeds for the HR460 stepper motor are also given in the appendix of the programming manual as 320 steps/second. This is a default parameter of the setSlitSpeed() function but can be updated by passing in a new speed for it. The speed of each slit needs to be set separately using the setSlitSpeed() function. To do this, the parameter required for the setSlitSpeed() function is the number for the slit. Table 1 gives the slit numbers and their corresponding slits.

Slit Correspondence Numbers

Slit Number	Corresponding Slit
0	Front entrance
1	Side entrance
2	Front exit
3	Side exit

Table 1. The table for the slit number used to give commands for a certain slit on the spectrometer.

The slit number MUST be passed as a string to the function or an error will occur. This means that the number can be passed as ‘0’ or “0” in the function. The following commands are examples used to set the slit speeds of the front entrance and side exit (the slit in front of the detector):

```
a.setSlitSpeed('0')
a.setSlitSpeed("3")
```

Again, the "a" in the example code should be the user-defined variable used for the Spectrometer object. This function also assumes the use of the "Monochromator 1" input. "mono='1'" should be sent to the function if using the other monochromator input.

If the slit speed was set correctly, a message will be displayed in the terminal informing the user that the slit speed was set to the specified value. If the device does not respond as expected, an error message will be displayed, and the device will exit the Python program. The spectrometer will need to be reset manually if this occurs.

Starting a Scan

Scan Setup

For a scan to be complete, initial parameters must be sent to the spectrometer to prepare the scan. Currently, the Spectrometer class only supports data collection through the detector. However, future work will be done to allow for the front exit slit to be used. The spectrometer device requires a command with 20 different parameters to be sent in order to prepare the device for scanning. Sending these parameters as variables to the function can be tedious, making sure that all parameters are the right one. The Spectrometer class, instead, prompts the user for each parameter when using the setScan() function, which can be used with the following statement:

```
a.setScan()
```

Similar to the other example functions, the "a" represents the user-defined variable for the Spectrometer object.

The user will then be prompted with all of the available parameters to be set. Note that numbers being selected, DO NOT have to be entered as a string in these responses. They will be automatically encoded upon entry. If the parameters are not entered correctly, or something went wrong in the process, the terminal will return an error message with the response from the device. This response can be classified in the Spectrometer Control manual written for programming the device.

If the parameters were accepted by the spectrometer, the exit mirror will be set to the position to collect data at the side detector, the slits will be opened to their specified values, and the grating will be flipped to the 800 lines/mm side. The terminal will display a message saying that setup has been complete and that the spectrometer is ready to begin a scan.

IMPORTANT: Note that slit sizes and motor movements are given in steps. 1 step corresponds to 12.5 μm. From the manual, 1 motor step corresponds to 0.009375 nm. Also, the DataScan device can only hold 5001 data points. This means that 1 step cannot be used to complete a scan of the entire length of the motor range unless multiple cycles are conducted. Be sure to choose a large enough step size so an error is not thrown when entering parameters.

Starting a Scan

Make sure that the scan has been prepared using the setScan() method before beginning a scan!

Once the scanning parameters have been set, scanning is simple using the startScan() method in the Spectrometer class. The following example statement is used to start the scan:

```
a.startScan()
```

If bad confirmation is received by the spectrometer, the function will return nothing and the terminal will print an error message, along with the response from the spectrometer.

If the command went through properly, the grating will be moved across the user specified range from setScan() and the terminal will display a message that scanning is complete when the scan is fully completed.

Collecting Data

Data collection is also made simple using the Spectrometer class. A simple call to getDataScan() can be used to place the data collected by the detector into a "Spectrum.txt" file that can easily be converted to a .csv file. IMPORTANT: Before collecting data, be sure to delete any old "Spectrum.txt" file or rename it, otherwise data may be added to it that is not wanted. The following command is used to collect the data:

```
a.getDataScan(‘1’)
```

The "'1'" that is passed to the function specifies the cycle number that the function is collecting data from. For more information on how many cycles of data the spectrometer can hold, see the Spectrometer Control manual.

If good confirmation is received, the function will create a "Spectrum.txt" file in the current directory and the terminal will display the data point that is being collected all the way up to the last data point. This gives the user a sense of how fast the data collection is going and when it will be done. Otherwise, if bad confirmation is received, the terminal will display an error message with the response of the spectrometer and the function will return nothing.

Once the scan is complete, the extension of the "Spectrum.txt" file can immediately be changed to .csv and opened in an Excel document to create a plot. The data is organized in columns of step number and intensity number for easy data access. Figures 3, 4, and 5 show the data collected after changing the extension to .csv and plotting the data in Excel for a white light source, a 632.8 nm HeNe laser, and a 543 nm HeNe laser, respectively.

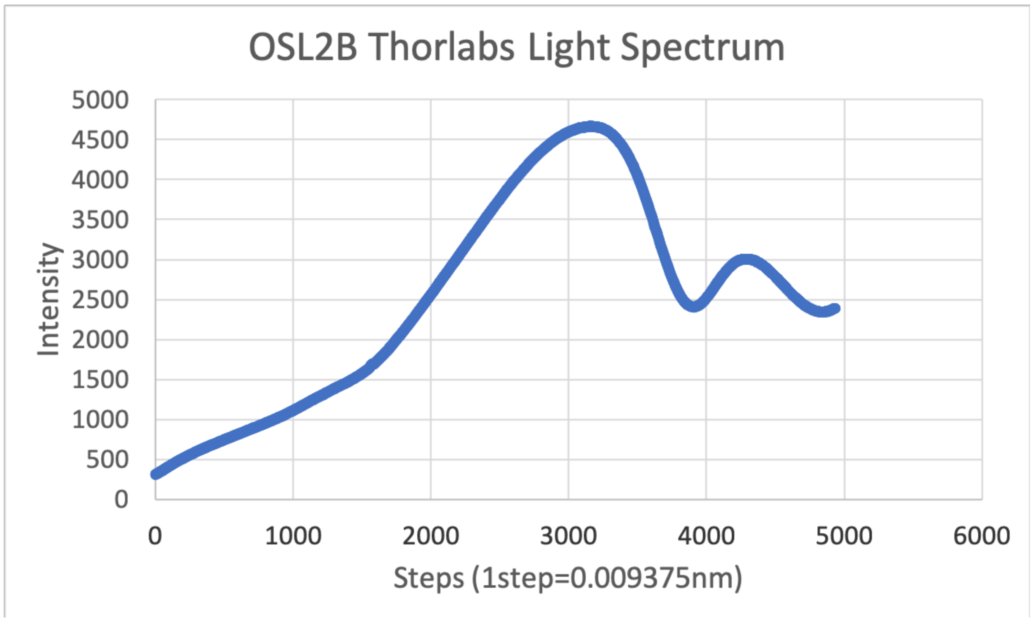


Figure 3. The spectrum taken of a Thorlabs OSL2B white light source. Note that this source was not correctly coupled to the device, so data may be slightly off but the data aligns very well with the Thorlabs specifications.

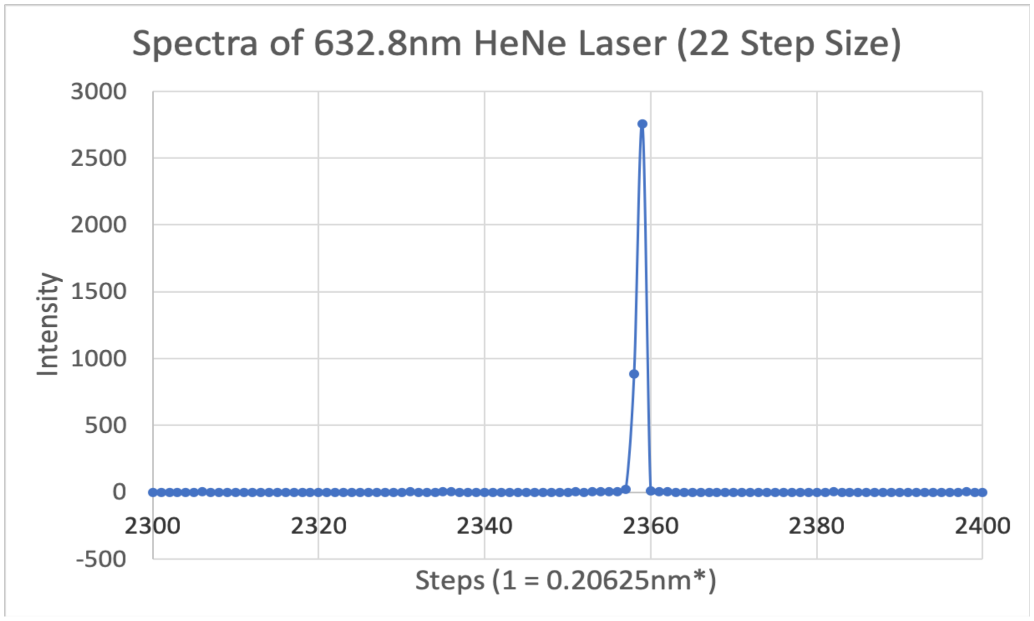


Figure 4. Data collected on a 632.8 nm HeNe laser at a step size of 22 steps/increment. The spike around 2358 steps shows a clear laser-like spectrum. This width can actually be resolved more by using smaller increments, as is shown in Figure 5.

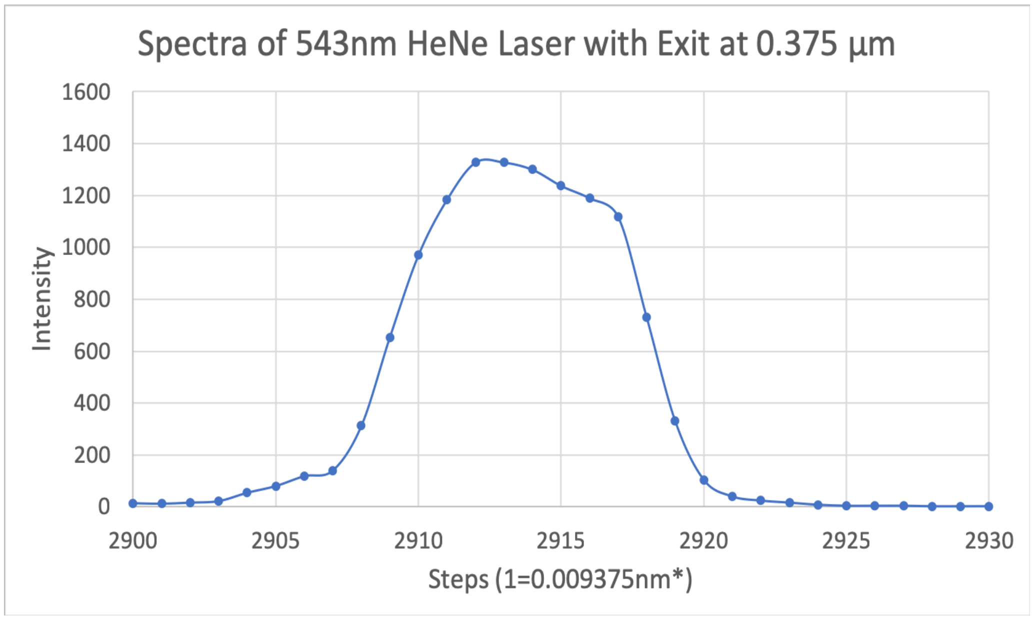


Figure 5. The spectral width of a 543 nm HeNe laser is observed by correctly coupling the beam into the device with a 75.6 mm lens (note that this is not the correct lens needed to be used with the device f/#) and adjusting the entrance and exit slits to the beam's spot sizes at those points. 1 step increments were used to better see the width of the spectrum.

Spectrometer Class Function List

All Functions Defined in Spectrometer.py

Function	**Description**	**Parameters**	**Return**
Spectrometer()	Constructor function to create a Spectrometer object	• usb = the name of the USB port to write to as a string. Ex: 'ttyUSB0'	NA. Can be assigned to a variable to create link to device.
whereAmI()	Function to check that the device is set to intelligent mode and in the "F" menu for writing to the device.	NA	NA. Will print out the response of the spectrometer. If everything is setup correctly, should print out "F"
on()	This function switches the spectrometer into intelligent mode and is used for startup of the device.	NA	NA. Will print if device is ready but will exit the Python program if an error occurs.

setMotorSpeed()	This function sets the motor speed of the stepper motor. Needs to be set before scanning use! All parameters are set to default values for the setup discussed in this manual.	• mono = '0' or '1' for monochromator 1 input or monochromator 2 input. • min = the minimum frequency in steps/second for the device. Number sent as a string. • max = the maximum frequency in steps/second. Number sent as a string. • ramp = the ramp time for the motor in milliseconds. Number sent as a string.	NA. Will print that the speed has been set if set correctly. Will exit the program if an error occurs.
getMotorSpeed()	Prints the values of the set motor speed parameters. Defaults to monochromator 1 input.	• mono = '0' or '1' for monochromator 1 input or monochromator 2 input.	NA. Prints the speeds to the terminal. Will exit the program if an error occurs.
checkMotor()	Returns the status of the motor. True if it is running and False if it is not.	NA	• True = Motor running • False = Motor not in use
moveMotor()	Moves the motor a specified number of steps. Will not allow the motor to move past the boundary steps of 208701 and 0.	• steps = the number of steps to move the grating. Number sent as a string. • mono = '0' or '1' for monochromator	• 0 if input out of stepper motor range • 1 if move completed Will also print that it has finished and will

		1 input or monochromator 2 input.	exit if bad response from the system.
getMotorPos()	Returns the step position of the motor as a string. The monochromator number is defaulted to input 1	• mono = ‘0’ or ‘1’ for monochromator 1 input or monochromator 2 input.	• output = the step number for where the motor is at in the form of a string.
stopMotor()	Stops the motor	NA	NA. Prints that the motor has stopped if good confirmation. Otherwise, exits the program with bad response printed to console.
initialize()	Initializes the spectrometer to all of its initial starting postions. Exit mirror set to front slit position, motor set at 208701 steps, slits closed and grating flipped to 800 lines/mm side.	NA	NA. Prints that it is initializing and when it has completed initialization. Exits program and prints error if bad response.
setSlitSpeed()	Sets the slit speed for the given slit. See Table 1 for the slit numbers. The speed and monochromator number is defaulted for the setup used in this manual.	• slit = the slit number in the form of a string. • speed = the slit speed as a string in steps/second. • mono = ‘0’ or ‘1’ for monochromator input 1 or monochromator input 2.	NA. Prints a message that the speed has been set successfully or exits the program if bad confirmation.
moveSlit()	Set the slit opening to a certain value. Does not allow slit to be set below 0 and moves slit opening to given	• slit = the slit number in the form of a string. • width = the width as a	NA. Prints when it is complete or exits if bad confirmation.

	width regardless of where it is currently located. mono variable defaulted to monochromator input 1.	number in steps sent as a string for the slit to be opened or closed. • mono = '0' or '1' for monochromator input 1 or 2.	
getSlitWidth()	Returns the current width of the selected slit in the form of a string. mono variable defaulted to monochromator input 1.	• slit = the slit number in the form of a string. • mono = '0' or '1' for monochromator input 1 or 2.	• output = the width of the slit opening in steps as a string.
setExitMirror()	Set the exit mirror to the desired position for data collection. The side position is used for data collection with the attached detector. mono variable defaulted to monochromator input 1.	• position = 's' for side position or 'f' for front position. • mono = '0' or '1' for monochromator input 1 or 2.	NA. Prints mirror moving status and when completed if successful. Exits program if bad confirmation.
setGrating()	Set the grating used to either the 800 lines/mm side or the 600 lines/mm side. mono variable defaulted to monochromator input 1.	• grating = 'vis' for 800 lines/mm or 'ir' for 600 lines/mm. • mono = '0' or '1' for monochromator input 1 or 2.	NA. Prints grating status as it moves and prints when completed. Exits program if bad confirmation.
measureOffsets()	Used to measure the gain offsets before completing a single point data scan. Provides user with gain offsets. mono variable defaulted to	• mono = '0' or '1' for monochromator input 1 or 2.	NA. Prints the gain offsets if successful. Exits the program if unsuccessful.

	monochromator input 1.		
setGain()	Used to set the gain before completing a single point data scan. Required for use before a single point data scan is taken. mono variable defaulted to monochromator input 1.	• gain = ‘0’, ‘1’, ‘2’, ‘3’, or ‘4’ for 1x, 10x, 100x, 1000x, or Auto gain settings. • mono = ‘0’ or ‘1’ for monochromator input 1 or 2.	NA. Prints that gain is set if successful. Exits the program if unsuccessful.
getGain()	See the current gain setting. mono variable defaulted to monochromator 1 input.	• mono = ‘0’ or ‘1’ for monochromator input 1 or 2.	NA. Prints the current gain setting to the terminal if good confirmation. Otherwise, exits the program.
setIntegrate()	Set the integration time before a single point scan. Must be set before taking single point data. mono variable defaulted to monochromator input 1.	• time = the integration time in milliseconds sent as a string. • mono = ‘0’ or ‘1’ for monochromator input 1 or 2.	NA. Prints that integration time set if successful. Exits the program if unsuccessful.
getIntegrate()	See the current integration time setting. mono variable is defaulted to monochromator input 1.	• mono = ‘0’ or ‘1’ for monochromator input 1 or 2.	NA. Prints the set integration time. Exits the program if bad response from device.
startAcq()	Start data acquisition for single point data. mono variable defaulted to monochromator input 1.	• mono = ‘0’ or ‘1’ for monochromator input 1 or 2.	NA. Prints that data acquisition has started if good confirmation. Otherwise, exits the program if bad confirmation.
stopAcq()	Stop the data acquisition process for single point data.	NA	NA. Prints that data acquisition has stopped if successful.

			Otherwise, exits the program if bad confirmation.
busyAcq()	Check to see if data is currently being taken by the spectrometer.	NA	• True = spectrometer collecting data. • False = spectrometer not collecting data.
getData()	Get single point data intensity reading. mono variable defaulted to monochromator input 1.	• mono = '0' or '1' for monochromator input 1 or 2.	• output = intensity reading for single point in the form of a string.
setScan()	Prepare the spectrometer for a scan. Will prompt the user for necessary parameters. Note that the device can only hold up to 5001 data points, so correct intervals must be used. Also sets device to be used with the detector and 800 lines/mm grating.	NA	NA. Prints that the parameters have been set if successful and gives status updates. Exits the program if unsuccessful or just returns if bad parameters given.
startScan()	setScan() must be completed before starting a scan! This function starts the scan and prints when the scan is complete.	NA	NA. Prints when the scan is complete. Returns with error message in console if bad confirmation.
getDataScan()	Collects the data for the given cycle and writes it to a "Spectrum.txt" file. Make sure that there is not already a file	• cycle = the cycle number used to collect data in the form of a string.	• steps = a numpy array of the number of steps completed

	named this in the current directory!		during the scan. • intensities = the intensity measurement collected at each step in the steps numpy array. Prints the step number that it is collecting data from to show how long the download will take and prints when completed.

Spectrometer.py Class Code

```
import struct
import serial
from serial import Serial
import sys
import math
import numpy as np
import csv


class Spectrometer:
    #Set up serial port connection.
    #Parameters: USB port name in a string format.
    #Returns: NA
    def __init__(self,usb):
        self.usb = usb
        usbdir = '/dev/' + self.usb
        self.s = serial.Serial(usbdir,19200,timeout=1)

    #Determine what menu the device is in. Prints where the device is.
    #Parameters: NA
    #Returns: NA. Prints where user is.
    def whereAmI(self):
        self.s.write(b' ')
        output=self.s.readline()
        output=output.decode('utf-8')
        print("You are here: " + output)

    #Reset the spectrometer. Print restart message
    #Parameters: NA
    #Returns: NA. Prints restarting though.
    def reset(self):
        self.s.write(struct.pack('!B',222))
        print("Restarting device")

    #Turn on the device. Includes switching to intelligent mode and F
main menu.
    #Parameters: NA
    #Returns: NA. Prints that it is done though.
    def on(self):
        self.s.write(b' ')
        output=self.s.readline()
        output=output.decode('utf-8')
        timer = 0

        #Make sure device turns on and is ready for commands
        try:
            while output[0]!='*' or output==' ' and timer<10000:
                self.s.write(b' ')
                output=self.s.readline()
                output=output.decode('utf-8')
                print('Wating for startup...')
                timer+=1
            if timer>=10000:
                sys.exit("Exited due to timeout starting 
spectrometer!")

        except IndexError:
            print('Not ready, trying again...')
            self.on()

        #Switch device to intelligent mode
        self.s.write(struct.pack('!B',247))
        output = self.s.readline()
        output = output.decode('utf-8')

        #If device does not give response back that it is in 
intelligent mode, exit
        if output != '=':
            sys.exit('Exited being unable to switch to intelligent 
mode! Response: ' + output)

        print("Switched to intelligent mode.")

        #Switch device into main mode (F mode)
        self.s.write(b'O2000'+struct.pack('!B',0))
        output = self.s.readline()
        output = output.decode('utf-8')
        print("Swtiched device to main mode (F).")

        #If the device doesn't return that the command was accepted, 
exit
        if output != '*':
            sys.exit('Exited being unable to switch to F mode! ' + 
output)

        #Check that the device is in 'F' mode
        #Parameters: NA
        #Returns: NA
        self.s.write(b' ')
        output = self.s.readline()
        output = output.decode('utf-8')
        print("Checking for main mode. Response; " + output)

        #If the device did not switch to F mode, exit
        if output != 'F':
            sys.exit('Exited because not in F mode!')

        print("Ready!")

    #Set the motor speed for the device. This needs to be done just
before or just after initialization
    #Parameters: mono = monochramtor input number  min = minimum
frequency (steps/s)  max = maximum frequency (steps/s)  ramp = ramp
time (ms)
    #Returns: NA. Prints when completed though.
    def
setMotorSpeed(self,mono='0',min='2560',max='5500',ramp='2000'):
        mono = str.encode(mono)
        min = str.encode(min)
        max = str.encode(max)
        ramp = str.encode(ramp)

        #Set motor speed
        self.s.write(b'B' + mono + b',' + min + b',' + max + b',' +
ramp + b'\r')
        output = self.s.readline()
        output = output.decode('utf-8')

        #If motor speed was not set correctly, exit
        if output != 'o':
            sys.exit('Exited being unable to set motor speed!' +
output)

        print("Set motor speed.")

    #Print values of the set motor speed
    #Parameters: mono = monochramtor input number
    #Returns: NA. Prints minimum and maximum frequency and ramp time.
    def getMotorSpeed(self,mono='0'):
        mono = str.encode(mono)

        self.s.write(b'C' + mono + b'\r')
        output = self.s.readline()
        output = output.decode('utf-8')
        try:
            if output[0]!='o':
                sys.exit("Bad confirmation for getting motor speed.
Response: " + output)
        except IndexError:
            sys.exit("Bad confirmation for getting motor speed.
Response: " + output)

        output = output[1:]
        output = output.split(",")

        print("Minimum frequency (steps/s): " + output[0] + "\n" +
"Maximum frequency (steps/s): " + output[1] + "\n" + "Ramp time (ms):
" + output[2])

    #Check the status of the motor to determine if it is still running
    #Parameters: NA
    #Returns: True if motor is still running  False if motor is not
running
    def checkMotor(self):
        self.s.write(b'E')
        output = self.s.readline()
        output = output.decode('utf-8')

        if output!='oz' and output!='oq':
            sys.exit("Bad confirmation for checking motor. Response: "
+ output)

        if output=='oz':
            return False
        else:
            return True

    #Move the motor a certain number of steps
    #Parameters: steps=a string representation of the number of steps
to move  mono=monochromator input number
    #Returns: NA. Prints when complete
    def moveMotor(self,steps,mono='0'):

        if int(steps)+int(self.getMotorPos())>208701 or
int(steps)+int(self.getMotorPos())<0:
            print("Cannot move motor beyond range of 0-208701!")
            return 0

        mono = str.encode(mono)
        steps = str.encode(steps)

        self.s.write(b'F' + mono + b',' + steps + b'\r')
        output = self.s.readline()
        output = output.decode('utf-8')

        if output!='o':
            sys.exit("Bad confirmation for moving motor. Response: " +
output)

        print("Moving motor " + steps.decode('utf-8') + " steps...")

        moving = self.checkMotor()

        while moving==True:
            moving = self.checkMotor()

        print("Move complete")

        return 1

    #Move the motor a certain number of steps
    #Parameters: steps=a string representation of the number of steps
to move  mono=monochromator input number
    #Returns: NA. Prints when complete
    def moveMotorScan(self,steps,mono='0'):

        if int(steps)+int(self.getMotorPos())>208701 or
int(steps)+int(self.getMotorPos())<0:
            print("Cannot move motor beyond range of 0-208701!")
            return 0

        mono = str.encode(mono)
        steps = str.encode(steps)

        self.s.write(b'F' + mono + b',' + steps + b'\r')
        output = self.s.readline()
        output = output.decode('utf-8')

        if output!='o':
            sys.exit("Bad confirmation for moving motor. Response: " +
output)

        print("Moving motor " + steps.decode('utf-8') + " steps...")

        return 1
    #Set current step position of motor. Used for calibration
    #Parameters: pos=step number to move grating to
mono=monochromator input number
    #Return: 0 if cannot execute  1 if executed. Prints what position
the motor was set to.
    def setMotorPos(self,pos,mono='0'):

        if int(pos)>208701 or int(pos)<0:
            print("Invalid position! Must be between 0-208701.")
            return 0

        pos = str.encode(pos)
        mono = str.encode(mono)

        self.s.write(b'G' + mono + b',' + pos + b'\r')
        output = self.s.readline()
        output = output.decode('utf-8')

        if output!='o':
            sys.exit("Bad confirmation for setting motor position.
Response: " + output)

        print("Set motor to position: " + pos.decode('utf-8'))

        return 1

    #Get the position of the motor
    #Parameters: mono=monochromator input number
    #Return: output, the current step position of the motor
    def getMotorPos(self,mono='0'):
        mono = str.encode(mono)

        self.s.write(b'H' + mono + b'\r')
        output = self.s.readline()
        output = output.decode('utf-8')

        try:
            if output[0]!='o':
                sys.exit("Bad confirmation for getting motor position.
Response: " + output)
        except IndexError:
            sys.exit("Bad confirmation for getting motor position.
Response: " + output)

        output = output[1:]
        output = output[:-1]
        return output

    #Stop the motor
    #Parameters: NA
    #Return: NA. Print that the motor was stopped.
    def stopMotor(self):
        self.s.write(b'L')
        output = self.s.readline()
        output = output.decode('utf-8')

        if output!='o':
            sys.exit("Bad confirmation for stopping motor!")

        print("Motor stopped.")

    #Initialize the device
    #Parameters: NA
    #Return: NA. Print through initialization process and print when
done.
    def initialize(self):
        #Tell device to initialize
        self.s.write(b'A')
        output = self.s.readline()
        output = output.decode('utf-8')
        print("Began initialization")

        #Check to make sure device is done initializing before
continuing.
        timer=0
        while output != 'o' and timer<10000:
            output = self.s.readline()
            output = output.decode('utf-8')
            print("Initializing...")
            timer += 1

        #If timeout occurs, exit
        if timer>=10000:
            sys.exit("Exited due to timeout initializing!")

        print("Initialization complete!")

    #Set speed of slit motor. Needs to be done either just before or
just after initialization. 320 recommended for HR460
```

```
    #Parameters: slit=string representation of slit number (0-3)
speed:string representation of speed (steps/s) mono=monochromator
input number
    def setSlitSpeed(self,slit,speed='320',mono='0'):
        slit=str.encode(slit)
        speed=str.encode(speed)
        mono=str.encode(mono)

        #Set slit speed
        self.s.write(b'g' + mono + b',' + slit + b',' + speed + b'\r')
        output = self.s.readline()
        output = output.decode('utf-8')
        speed = speed.decode('utf-8')

        #Check that command for setting speed was received correctly
        if output != 'o':
            sys.exit('Exited being unable to set slit speed!')

        print("Set slit speed to " + speed + ". Response: " + output)

    #Change slit width (0=front entrance, 1=side entrance, 2=front
exit, 3=side exit). Note: Will not allow move past 0 step position
    #Parameters: slit=slit number (0-3)  width=steps to open slit
mono=monochromator input number
    #Return: NA. Print when done
    def moveSlit(self,slit,width,mono='0'):
        self.s.write(b'j' + mono + b',' + slit + b'\r')
        output = self.s.write.readline()
        output = output.decode('utf-8')

        try:
            if output[0]!='o':
                sys.exit('Bad confirmation for checking slit position!
Response: ' + output)
        except IndexError:
            sys.exit("Bad confirmation for checking slit position.
Response: " + output)

        if float(output[1:])+float(width)<0:
            print("Setting beyond 0. Cannot close slits more! Slit
opening at: " + output[1:])

        if float(width)<0:
            negative = True
        else:
            negative = False
        mono = str.encode(mono)
        slit = str.encode(slit)
        width = str.encode(width)
        #Open the entrance slit
        self.s.write(b'k' + mono + b',' + slit + b',' + width + b'\r')
        output = self.s.readline()
        output = output.decode('utf-8')
        width = width.decode('utf-8')

        #Check that command for opening entrance slit was received
correctly
        if output != 'o':
            sys.exit('Exited being unable to open entrance slit!')

        #Check that motor has finished opening the slit
        self.s.write(b'E')
        output = self.s.readline()
        output = output.decode('utf-8')

        timer=0
        while output != 'oz' and timer<10000:
            self.s.write(b'E')
            output = self.s.readline()
            output = output.decode('utf-8')
            print("Checking motor for status after entrance slit
opened... Response: " + output)
            timer += 1
        if negative == True:
            print("Slit closed " + str(width) + " steps")
        else:
            print("Slit opened " + str(width) + " steps")

        self.s.write(b'j' + mono + b',' + slit + b'\r')
        output = self.s.write.readline()
        output = output.decode('utf-8')

        try:
            if output[0]!='o':
                sys.exit('Bad confirmation for checking slit position!
Response: ' + output)
        except IndexError:
            sys.exit("Bad confirmation for checking slit position.
Response: " + output)

        output = output[1:]
        print("Slit now at " + output + " steps.")
```

```
    #Get the current width of a slit
    #Parameters: slit=0 for front entrance, 1 for side entrance, 2 for
front exit, 3 for side exit  mono=monochromator input number
    #Return: The slit width of the selected slit in the form of a
string
    def getSlitWidth(self,slit,mono='0'):
        slit = str.encode(slit)
        mono = str.encode(mono)

        self.s.write(b'j' + mono + b',' + slit + b'\r')
        output = self.s.readline()
        output = output.decode('utf-8')

        try:
            if output[0]!='o':
                sys.exit("Bad confirmation for checking slit width!
Response: " + output)
        except IndexError:
            sys.exit("Bad confirmation for checking slit width!
Response: " + output)

        output = output[1:]

        return output


    #Set the exit mirror position (data collection through the side or
through the front)
    #Parameters: position=either 's' or 'f' for side and front data
collection  mono=monochromator input number
    #Return: NA. Prints when done.
    def setExitMirror(self,position,mono='0'):
        mono = str.encode(mono)
        #Check that right input selected
        if position=='s':
            mode = 'side detector'
            position='e'
        elif position=='f':
            mode = 'front detector'
            position='f'
        else:
            sys.exit("Exited due to improper mirror position input!")

        position = str.encode(position)
        #Set mirror to specific mode
        self.s.write(position + mono + b'\r')
        output = self.s.readline()
        output = output.decode('utf-8')

        #Check that command for moving mirror was received correctly
        if output != 'o':
            sys.exit('Exited being unable to move mirror position!')

        # Check that the mirror is not still moving before continuing
        self.s.write(b'l')
        output = self.s.readline()
        output = output.decode('utf-8')
        print("Mirror motor status: " + output)

        timer = 0
        while output!='oz' and timer<10000:
            self.s.write(b'l')
            output = self.s.readline()
            output = output.decode('utf-8')
            print("Motor moving mirror...")
            timer+=1

        #If timeout occurs, exit
        if timer>=10000:
            sys.exit("Exited due to timeout moving mirror!")

        print("Set mirror to " + mode + " mode. Response: " + output)

    #Flip the grating side
    #Parameters: grating='ir' for 1.5 micron infrared  'vis' for
visible spectrum grating  mono=monochrometer input number
    #Return: NA. Return if bad parameter input. Print when finished.
    def setGrating(self,grating,mono='0'):
        mono = str.encode(mono)
        if grating!='ir' and grating!='vis':
            print("Invalid grating selection! Must be either \'ir\' or
\'vis\'!")
            return

        if grating=='ir':
            grating = b'a'
        else:
            grating = b'b'
        self.s.write(grating + mono + b'\r')
        output = self.s.readline()
        output = output.decode('utf-8')

        if output!='o':
            sys.exit("Bad confirmation on changing grating. Response: 
" + output)

        # Check that the grating is not still moving before continuing
        self.s.write(b'l')
        output = self.s.readline()
        output = output.decode('utf-8')
        print("Grating change motor status: " + output)

        timer = 0
        while output!='oz' and timer<10000:
            self.s.write(b'l')
            output = self.s.readline()
            output = output.decode('utf-8')
            print("Motor moving grating...")
            timer+=1

        #If timeout occurs, exit
        if timer>=10000:
            sys.exit("Exited due to timeout moving grating!")

        print("Grating changed.")

    #Measure the gain offsets
    #Parameters: mono=monochromator input number
    #Return: NA. Prints the gain offsets.
    def measureOffsets(self,mono='0'):
        mono=str.encode(mono)
        self.s.write(b'w' + mono + b'\r')
        output = self.s.readline()
        output = output.decode('utf-8')

        try:
            if output[0]!='o':
                sys.exit("Bad confirmation for measuring offsets. 
Response: " + output)
        except IndexError:
            sys.exit("Bad confirmation for measuring offsets. 
Response: " + output)

        output = output[1:]
        output = output.split(',')
        print("Offset Gain 1: " + output[0] + "\n" + "Offset Gain 10: 
" + output[1] + "\n" + "Offset Gain 100: " + output[2] + "\n" + 
"Offset Gain 1000: " + output[3] + "\n")

    #Set the gain for the measurement
    #Parameters: gain=string representation of gain setting (0-4)
(1x,10x,100x,1000x,auto)  mono=monochromator input number
    #Return: NA. Print when gain set. Return nothing if invalid value 
entered
    def setGain(self,gain,mono='0'):
        if gain!= '0' and gain!='1' and gain!='2' and gain!='3' and 
gain!='4':
            print("Invalide gain selection! (0=1x, 1=10x, 2=100x, 
3=1000x, 4=AUTO")
            return

        gain=str.encode(gain)
        mono=str.encode(mono)

        self.s.write(b'R' + mono + b',' + gain + b'\r')
        output=self.s.readline()
        output=output.decode('utf-8')

        if output!='o':
            sys.exit("Bad confirmation for setting gain. Response: " + 
output)

        print("Gain set to setting " + gain.decode('utf-8'))

    #Show the set gain value
    #Parameters: mono=monochromator input number
    #Return: NA. Print the gain setting value.
    def getGain(self,mono='0'):
        mono=str.encode(mono)
        self.s.write(b'S' + mono + b'\r')
        output=self.s.readline()
        output=output.decode('utf-8')

        try:
            if output[0]!='o':
                sys.exit("Bad confirmation for getting gain. Response: 
" + output)
        except IndexError:
            sys.exit("Bad confirmation for getting gain. Response: " + 
output)
```

```
        output = output[1:]
        print("Gain setting: " + output[0])

    #Set the integration time
    #Parameters: time=string representation of time in ms
mono=monochromator input number
    #Return: NA. Print when set.
    def setIntegrate(self,time,mono='0'):
        time = str.encode(time)
        mono = str.encode(mono)

        self.s.write(b'O' + mono + b',' + time + b'\r')
        output = self.s.readline()
        output = output.decode('utf-8')

        if output!='o':
            sys.exit("Bad confirmation for setting integration time.
Response: " + output)

        print("Integration time set to " + time.decode('utf-8') +
"ms")

    #Show the value of the set integration time
    #Parameters: mono=monochromator input number
    #Return: NA. Print the set integration time
    def getIntegrate(self,mono='0'):
        mono = str.encode(mono)

        self.s.write(b'P' + mono + b'\r')
        output = self.s.readline()
        output = output.decode('utf-8')

        try:
            if output[0]!='o':
                sys.exit("Bad confirmation for getting integration
time. Response: " + output)
        except IndexError:
            sys.exit("Bad confirmation for getting integration time.
Response: " + output)

        output = output[1:]
        print("Integration time set to: " + output + "ms")

    #Start data acquisition
    #Paramters: mono=monochromator number
    #Return: NA. Print that it is starting
    def startAcq(self,mono='0'):
        mono = str.encode(mono)

        self.s.write(b'M' + mono + b'\r')
        output = self.s.readline()
        output = output.decode('utf-8')

        if output!='o':
            sys.exit("Bad confirmation for starting data acquisition.
Response: " + output)

        print("Starting data acquisition...")

    #Stop data acquisition
    #Parameters: NA
    #Return: NA. Print that it is stopped.
    def stopAcq(self):
        self.s.write(b'N')
        output = self.s.readline()
        output = output.decode('utf-8')

        if output!='o':
            sys.exit("Bad confirmation for stopping data acquisition.
Response: " + output)

        print("Data acquisition stopped!")

    #Check if taking data
    #Parameters: NA
    #Return: False if not taking data  True if taking data
    def busyAcq(self):
        self.s.write(b'Q')
        output = self.s.readline()
        output = output.decode('utf-8')

        if output!='oz' and output!='oq':
            sys.exit("Bad confirmation for checking busy data
acquisition. Response: " + output)

        if output=='oz':
            return False
        else:
            return True

    #Get the intensity measurement from the data point
    #Parameters: mono=monochromator input number
    #Return: intensity measurement as a string (can convert to int
value)
    def getData(self,mono='0'):
        mono = str.encode(mono)

        self.s.write(b'T' + mono + b'\r')
        output = self.s.readline()
        output = output.decode('utf-8')

        try:
            if output[0]!='o':
                sys.exit("Bad confirmation for collecting data.
Response: " + output)
        except IndexError:
            sys.exit("Bad confirmation for collecting data. Response:
" + output)

        output = output[1:]
        output = output.split(",")
        return output[0]

    #Prepare the spectrometer for a scan. CAUTION: input values have
not been secured. So a bad input can cause an error! Assumes use of
front entrance slit and side exit slit
    #Parameters: NA. All parameters set by user input at command line
when prompted
    #Returns: NA. Prints that parameters were set properly.
    def setScan(self):
        type1 = str.encode(input("Enter the type of scan to be
completed:\n0 - Monochromator 1 scan.\n1 - Monochromator 2 scan\n3 -
Time Base Scan\n"))
        startPos = str.encode(input("Enter the starting position for
scan in steps (0-208701): "))
        endPos = str.encode(input("Enter the ending position for scan
in steps (0-208701): "))
        steps = str.encode(input("Enter the interval of steps (This
parameter ignored if running Time Base Scan): "))
        intTime = str.encode(input("Enter the integration time in
milliseconds (2-300,000) as an even number: "))
        cycles = str.encode(input("Enter the number of scan cycles to
be completed: "))
        dwell = str.encode(input("Enter the time (in milliseconds)
that the system waits after moving the monochromator (0-2147483648):
"))
```

```
        delay = str.encode(input("Enter the time (in milliseconds)
that the system waits after completing each cycle (0-2147483648): "))
        entSlit = input("Enter the entrance slit width in steps: ")
        extSlit = input("Enter the exit slit width in steps: ")
        startPos2 = str.encode(input("Enter the starting position (in
steps) for monochromator 2 (Parameter ignored if no monochromator 2):
"))
        parkPos1 = str.encode(input("Enter the position (in steps) to
park the monochromator when finished (0-208701): "))
        timeStep = str.encode(input("Enter the time (in milliseconds)
between integration starts (Ignored if no time scan): "))
        time = str.encode(input("Enter the total time (in
milliseconds) of the scan (0-2147483648): "))
        channel = str.encode(input("Enter the channel used for data
acquisition:\n0 - channel 1\n1 - channel 2\n2 - use both channels
simultaneously\n"))
        gain1 = str.encode(input("Enter the gain setting for channel
1:\n0 - 1x\n1 - 10x\n 2 - 100x\n3 - 1000x\n4 - AUTO\n"))
        gain2 = str.encode(input("Enter gain for channel 2 using
paramters above: "))
        shutter = str.encode(input("Enter shutter mode parameter:\n0 -
AUTO\n1 - MANUAL\n"))
        trigger = str.encode(input("Enter trigger mode parameter:\n0 -
no trigger\n1 - wait for start of experiment\n2 - wait for start of
cycle\n3 - wait for start of each data point\n"))
        data = str.encode(input("Enter data mode parameter:\n0 - stack
acquired scans separately\n1 - sum acquired scans in memory\n"))

        self.s.write(b'p' + type1 + b',' + startPos + b',' + endPos +
b',' + steps + b',' + intTime + b',' + cycles + b',' + dwell + b',' +
delay + b',' + struct.pack('!B',0) + b',' + parkPos1 + b',' +
struct.pack('!B',0) + b',' + timeStep + b',' + time + b',' + channel +
b',' + gain1 + b',' + gain2 + b',' + shutter + b',' + trigger + b',' +
data + b'\r')
        output = self.s.readline()
        output = output.decode('utf-8')

        if output!= 'o0\r':
            print("Error! Returned: " + output)
            return

        currentWidth = float(self.getSlitWidth('0'))
        moveAmount = str(float(entSlit) - currentWidth)
        self.moveSlit('0',moveAmount)

        currentWidth = float(self.getSlitWidth('3'))
        moveAmount = str(float(entSlit) - currentWidth)
        self.moveSlit('3',moveAmount)

        self.moveSlit('3',extSlit)
        self.setExitMirror('s')
        self.setGrating('vis')
        self.s.readline()

        print("Set scan parameters and prepared spectrometer for 
scanning!")

    #Start the scan after being set using the setScan() function
    #Parameters:NA
    #Returns: NA. Returns if unable to start scan. Prints when 
finished.
    def startScan(self):
        self.s.write(b'q')
        output = self.s.readline()
        output = output.decode('utf-8')

        if output!='o':
            print("Bad confirmation for starting scan! Response: " + 
output)
            return

        self.s.write(b'r')
        r = self.s.readline()
        r = r.decode('utf-8')

        while r!='o0\r' and r!='o5\r':
            self.s.write(b'r')
            r = self.s.readline()
            r = r.decode('utf-8')

        print("Scan complete!")

    #Function to collect data after scan complete
    #Parameters: cycle=the cycle number that you wish to collect data 
from, in the form of a string
    #Returns: steps=the numpy array of steps  intensities=the numpy 
array of measured intensities
    def getDataScan(self,cycle):
        cycle = str.encode(cycle)

        self.s.write(b's' + cycle + b'\r')
        output=self.s.readline()
        output=output.decode('utf-8')

        if output!='o':
            print("Bad confirmation for setting cycle to read from!
Response: " + output)
            return

        self.s.write(b't')
        output = self.s.readline()
        output = output.decode('utf-8')

        try:
            if output[0]!='o':
                print("Bad confirmation for getting data length!
Response: " + output)
                return
        except IndexError:
            print("Bad confirmation for getting data length! Response:
" + output)
            return

        output = output[1:]

        output = output.split(',')

        length = int(output[0])

        print(length)

        file1 = open("Spectrum.txt","w")

        intensities = np.empty(5001)
        steps = np.empty(5001)

        for i in range(1,length):
            print(i)
            i = str(i)
            i = str.encode(i)
            self.s.write(b'u' + i + b'\r')
            output = self.s.read_until(b'\r',15)
            output = output.decode('utf-8')

            try:
                if output[0]!='o':
                    print("Bad confirmation for getting data point " +
i.decode('utf-8') + "! Response: " + output)
```

```
                    file1.close()
                    return
            except IndexError:
                print("Bad confirmation for getting data point " +
i.decode('utf-8') + "! Response: " + output)
                file1.close()
                return

            output = output[1:]

            output = output.split(',')

            intensity = output[0]
            intensities=np.append(intensities,intensity)

            file1.write(i.decode('utf-8') + ',' + str(intensity) +
'\n')

            steps=np.append(steps,int(i.decode('utf-8')))

        file1.close()
        print("Data acquisition complete!")
        return steps,intensities
```

References:

1. Oksenhendler T., "Self-Referenced Spectral Interferometry Theory", FASTLITE, Centre Scientifique d'Orsay, 2012. https://www.researchgate.net/publication/224812644_Self-referenced_spectral_interferometry_theory
2. Boggs B., Physics Department, University of Oregon, 1585 E 13 Avenue Eugene, OR 97403
3. Dinyari N., Physics Department, University of Oregon, 1585 E 13 Avenue Eugene, OR 97403
4. "Spectrometer Control: Interfacing and Programming Manual", Jobin-Yvon – Spex Instruments, 1997. http://hank.uoregon.edu/wiki/images/c/c2/InterfaceManual.pdf.
5. Jain C. R., Kasturi R., and Schunck B., *Introduction to Machine Vision,* McGraw-Hill Education, 1995.
6. Theriault, G., "The Beginner's Guide on Spot Size of Laser Beam", Gentec Co., 2018. https://www.gentec-eo.com/blog/spot-size-of-laser-beam
7. "Diffraction Grating Physics", Newport Co., https://www.newport.com/t/grating-physics