A Glorious Guide on Stage Mapping and Experiment Automation using Python for Raman Spectroscopy

Sharma Raman Lab

Tarek Atyia

Introduction

Subsequent spectroscopic measurements on a single sample are common in the scientific community. Manually entering settings and moving the microscope stage with the controller to orient the sample can become exhausting, especially for experiments with longer exposure times and larger samples. This guide will describe a method that will automate stage movement and experiment acquisition. By integrating this software, this guide aims to decrease human involvement with taking measurements and increase throughput of all experiments in the lab that utilize the microscope stage and spectrometer.

LightField

What is LightField?

LightField is a program developed by Teledyne Princeton Instruments. This software is used to control and acquire data from a wide range of Princeton Instrument cameras, including spectroscopy, microscopy and various light field camera systems. This is an essential tool when taking Raman spectra as it coordinates measurements taken using the spectrometer, camera, and lasers. This program is able to compile the data from these instruments and display it as a spectra that can be exported as a .csv (excel) file.

Stage Controllers & Mapping

General Information

There are many companies that sell many different stage controllers. While there is a python package that supports many controllers (**python-microscope)**, the documentation is sparse and difficult to implement without prior knowledge of python code with the specific controller that you are using.

In this guide, the Prior Instruments Pro Scan III stage controller (*Figure 1)* will be used to demonstrate how to automate an inverted microscope system as shown in *Figure 2.*

Prior Instruments includes example code on their website that is incredibly helpful when using a Prior controller (https://www.prior.com/download-category/software). In order to begin controlling the stage with python code, you need to download the 3MB zip archive named “Prior SDK 1.7.0”. After extracting this archive with 7-zip or Winrar, the file will contain a folder called “Prior SDK 1.7.0” which will contain a folder called “examples”. That folder will contain a folder called “python” which will contain a file called “prior_interface.py”. This python file contains the necessary imports for the prior controller and provides an example of using different commands to control the microscope stage.

Figure 1. Prior ProScan III

Figure 2. Inverted Microscope

Crucial Code

The documentation for the python code is outlined in the “Prior SDK 1.7.0” folder; however, here are the most useful commands utilized in order to take control of your stage:

- *controller.stage.position.set* <*x*> <*y*> → This command sets the stage to a certain x,y position. Setting this to 0,0 at the beginning of your experiment is useful as a baseline position. This command is especially important before mapping specific patterns for the stage such as moving in a square or a spiral so the controller can be set to a position that will allow the full movement of the desired shape.
- *controller.stage.position.move-relative* <*x*> <*y*> → this command will move the stage relative to the position it was most recently at. This is nice as you will know exactly where the controller is moving since sometimes mapping the coordinate plane that the stage moves on can be confusing as every stage controller has different methods regarding how the coordinate plane of the stage is laid out.

It is important to note that every python package relating to this topic will have its own set of commands, the ones listed above are solely for Prior controllers. Other controllers will require either a different API (application programmable interface) / SDK (software development kit) or a python package (*i.e.*, python-microscope) that includes control for multiple controllers.

COM Ports

Every controller will be plugged into the computer in some fashion. Most commonly this is done via USB-A. Using USB-A to connect the controller will tell the computer to automatically specify a COM (communication) port for this device. In order to identify which COM port you are using, you will need to open the Device Manager (you can search for this using the Windows 7/8/10/11 search bar). I figured this out by going to “Ports (COM & LPT)”, unplugging the stage and noting which disappeared from the list. The stage should have a number next to it labeled (COM<X>; X being whatever port it's using). This will be the COM port you connect to with the following command at the beginning of your program so the python code can connect to the controller:

cmd(“controller.connect <X>”)

Mapping Algorithms

Depending on your sample, you will want the stage to move in different patterns in order to capture all relevant spectra at specific points along the sample. This can be done with a few lines of code depending on what shape you want the stage to move in.

This is shown in the code example for a circle algorithm below where the user is able to input settings to move the stage in the desired fashion:

```
if input == "circle":
        print("----INITIATING CIRCLE----")
        print("\n")
      #define radius (everything is in micrometers)
        radius = float(input("Enter the radius of the circle in microns: "))
        cen_x = float(input("Enter the x coordinate for the center of the
circle: "))
        cen_y = float(input("Enter the y coordinate for the center of the
circle: "))
        num_measurements = 8 #defining how many measurements (points) are on
the circle
        points = [] #this will hold the coordinates of each measurement

        for i in range(num_measurements):
            theta = 2*math.pi*i / num_measurements #equation of a circle
            x = cen_x + float(radius)*math.cos(theta) #x coord equation for
circle
            y = cen_y + float(radius)*math.sin(theta) #y coord equation for
circle
            points.append((x, y)) #add to dictionary
        for point in points: #tell the stage to move to each point
            cmd("controller.stage.move-relative {} {}".format(str(point[0]),
str(point[1])))#telling the stage to move to all the points in the newly
created list.
```

These algorithms are limited only by imagination and the Cartesian coordinate plane. Any shape that can be drawn using X, Y coordinates can be mapped with the stage mainly using the two controller commands listed above. Other

commands are listed in the Prior SDK documentation (or if you're using another controller you will find the documentation on either the python package's website or the company responsible for manufacturing your specific controller's website).

The Prior SDK will conveniently print out all of the points that it's mapping the stage to, so if you want to confirm that the stage is actually moving in the desired shape based on your code, you can plot these points using *matplotlib.pyplot* to confirm that your code is executing properly.

Configuring LightField with Python

LightField can be configured using the provided SDK from Teleydne instruments. The SDK contains many python files that are examples of automating LightField. The code from these files can be implemented into the code that is used to control the stage to simultaneously move the stage using the Prior controller and take measurements of the sample using LightField.

To use the LightField SDK, you will need to install the *pythonnet* package which includes the *clr* library. Generally, programs that are written for windows will be interfacing with the Windows API, which is written in C. The *pythonnet* package allows you to use python to simultaneously use the LightField SDK and connect your program to the Windows API. This package is absolutely necessary and can be installed using this command in your command prompt / terminal, assuming you have already installed python on your system:

pip install pythonnet

People in the past (myself included) have installed only the *clr* package using:

pip install clr

THESE TWO PACKAGES ARE NOT THE SAME AND DO NOT HAVE THE SAME FUNCTIONALITY; the program will not function with *clr.* Ensure that you install *pythonnet* before you begin.

Helpful Information

https://github.com/sliakat/SpeReadPy

This GitHub has the code from the engineer (Sabbir Liakat) at Princeton Teledyne Instruments that can be used to control LightField. There are many examples of how the python code can work with LightField which are fully explained in Sabbir's YouTube series on this topic:

(https://www.youtube.com/watch?v=ZJsrB1p1VJ4&t=1150s&ab_channel=TeledynePrincetonInstruments)

I highly encourage watching these videos and analyzing the code before you get started.

Included with the LightField SDK is a compiled HTML file (.chm) called "LightField Experiment Settings" which has all of the classes and available commands included in the LightField API, should you want to include more functionality and automate more settings.

A note about "AddInProcess.exe"

The Task Manager is your friend when automating this process. When automating LightField with python, the window that opens is actually registered as an "add-in process". This distinguishes to the computer that this is an automated instance of LightField rather than an actual LightField window. Before starting any automation or any normal instance of LightField, you must ensure that you close any instances of "LightField" or "AddInProcess.exe" in the Task Manager so that no instances run on top of each other.

THIS IS ABSOLUTELY NECESSARY FOR THE CODE TO FUNCTION PROPERLY.

Crucial Code

LightField can be opened with python code using the following lines of code:

auto = Automation(True, List[String]())

experiment = auto.LightFieldApplication.Experiment

These two lines will load the most recent experiment that was opened in LightField, which can cause problems if multiple people are using the same

computer for different experiments, therefore it is useful to have an empty experiment .spe file that you can load using the following command:

experiment.Load(“name_of_your_file”)

You will also need to add the devices you want to use in the experiment before you can utilize this empty experiment file. This can be done by opening a new experiment in a normal, non-automated instance of LightField and selecting the “DEVICES” tab. You can then drag the necessary devices from “available devices” into “experiment devices” and then save the file.

Important Information Before Programming

Using commands related to LightField within different scopes (i.e. within if statements, while loops, or for loops) will break the connection between the python code and the LightField API. Here’s what I mean:

```
if le_input == 'q':
    exit
auto = Automation(True, List[String]())
experiment = auto.LightFieldApplication.Experiment
if le_input == "square":
    print("Attempting to communicate with LightField....")


    find_spec()#functions to check for available devices
    find_cam()

    set_value(CameraSettings.ShutterTimingExposureTime, 50.0)


    print("----INITIATING SQUARE----")
    print('\n')
    print("FOR REFERENCE: THE DIAMETER OF A NICKEL IS 21000 MICRONS")
    x = int(input("X-Direction (in microns): "))
    y = int(input("Y-Direction (in microns): "))

    cmd("controller.stage.move-relative " + str(x) + " 0")
```

```
    experiment.Acquire()
```

In the code above, I instantiate the *auto* and *experiment* variables **<u>OUTSIDE</u>** of the if statement. This causes the last line, *experiment.Acquire()* to run in a different scope than the actual LightField application. In essence, when the code is run like this, *experiment.Acquire()* has no idea what it's acquiring because it is not aware of any open LightField instance because that instance is outside of the if statement. The image will still save and you will be able to open it; however, the .spe file that you do save will be corrupted and therefore useless.

It is also possible for LightField to crash during the automation process because it is overloaded with commands. Here's what I mean:

```
cmd("controller.stage.move-relative " + str(x) + " 0")
experiment.Acquire()
cmd("controller.stage.move-relative 0 " + str(y))
experiment.Acquire()
cmd("controller.stage.move-relative " + "-" + str(x) + " 0")
experiment.Acquire()
cmd("controller.stage.move-relative 0 " + "-" + str(y))
experiment.Acquire()
cmd("controller.stage.position.get")
```

This code will most likely cause LightField to crash because it requests LightField to acquire an exposure before the last exposure has time to finish. I solved this by simply making the program sleep for 2 seconds after every exposure as shown below:

```
cmd("controller.stage.move-relative " + str(x) + " 0")
experiment.Acquire()
time.sleep(2)
cmd("controller.stage.move-relative 0 " + str(y))
experiment.Acquire()
time.sleep(2)
cmd("controller.stage.move-relative " + "-" + str(x) + " 0")
experiment.Acquire()
```

```
    time.sleep(2)
    cmd("controller.stage.move-relative 0 " + "-" + str(y))
    experiment.Acquire()
    time.sleep(2)
    cmd("controller.stage.position.get")
```

You want to ensure that the instance of LightField does not close before it is done acquiring all of the images. Ending the program prematurely can corrupt your .spe file. To solve this issue at the end of every script that is automating LightField to take measurements, I added this line of code:

input(“Press ENTER to exit”)

This ensures that the program can only be ended via user input, which will allow LightField to finish taking its measurements before the program is ended.