Coding a Stark Map

Julia McDowell

with Professor Thomas Carroll, Ursinus College

Abstract

During a one month internship with Dr. Thomas Carroll, a physics professor at Ursinus College, I was asked to create a computer program to make an interactive Stark map. The program needed to read a data file filled with coordinates and show the requested range of coordinates in a graph. This graph needed to be able to measure the distance between two coordinate points and to use the arrow keys to move the graph and load more coordinate points. The program I created would be used by one of Dr. Carroll's students to create a similar but more advanced version of an interactive Stark map.

Background

The Stark effect, discovered in 1913 by Johannes Stark "is a shift in the energy levels of an atom in an electric field" (Highly Excited Atoms). Different external electric fields are applied to energy levels of an atom, causing the energy levels to shift. The effect is usually applied to Rydberg atoms, or highly excited atoms made by moving their outermost electrons to a high energy level. These huge atoms can be easily manipulated. They have a larger Stark shift than regular atoms because they have a greater number of energy levels. The discovery of this effect has contributed greatly to the development and study of quantum theory. A Stark map is a visual representation of the energy shift resulting from different electric fields. The x axis represents the electric fields, measured in V/cm, and the y axis represents the resulting energy shifts, measured in cm^{-1}.

Process

Given the objective, a data file, and a recommendation of a Python library from Dr. Carroll, I began to design the program. Using Python, a computer language that neither Dr. Carroll nor his students knew how to use, I knew that I would be largely on my own concerning the code itself. Using Mac Terminal, my computer was updated to use the latest Python 2 and pip, a management system used to install python packages. I had to decide between two programs to use: Matplotlib, a python plotting library, and Plotly, an online data plotting tool. Because it was more easily customizable with the ability to add and edit

functions, I chose Matplotlib.

First, the data file, composed of packed binary data, was opened and assigned the variable name "tfile". The file included 30,001 different electric field points, each with 1,150 energies. It was believed that tfile was organized as a list with each of the 30,001 electric field numbers followed by their 1150 energies (ex: [x_1,y_1,y_2,y_3...y_{1150}, x_2...]). Each number was 8 bytes large. The code was written to put each electric field point into an "x list" 1150 times and each energy into a "y list", therefore, the x and y lists could create coordinates for the program. Both lists were unable to be printed in their entirety because of their huge size (printing a whole list resulted in the computer freezing). When small portions of the lists were printed, the data showed inconsistencies. In tfile, the electric field points were supposed to range from 0 to 300.00 V/cm, whereas the energies were supposed to range from -138.5 to -39.08 cm^{-1}. After examining the lists more closely, I determined that the data file was set up as a list of coordinate components: [x_1, y_1, x_2, y_2, x_3, y_3...$x_{17250575}$, $y_{17250575}$]. A loop was set up so that each iteration, an x-y pair, would be unpacked using the struct module and added to a list named "points" to create a coordinate. This list was then added to another list named "point_list" to create a list of coordinate lists. The number of times to loop through this process with the loop variable "electric" was 17,250,575 because, in this data file, there were 30,001 points multiplied by 1,150 energies, divided by 2 components per coordinate.

After all the data had been put into a coordinate list and small portions of the list could be printed, the next step was to choose the points to be shown. One function of the program was that the user could input the range of data points they wished to see. Using a while loop to check the validity of the user input, raw_input() was used to prompt the user for starting and ending field and energy numbers. The inputs were assigned to the variables that matched their purpose.

Next, the find_points() function was defined in order to separate the coordinates into two lists: x components and y components. The start value was subtracted from the end value of field and energy numbers to find the difference and assigned to xshift, for the field, and yshift, for the energy. The arguments of find_points() were the inputted field and energy start numbers minus the shifts and inputted field and energy end numbers plus the shifts in order to speed up the later scrolling process. After these components were added to their respective lists, the coordinate would be removed from the point_list to prevent points from being loaded twice. The figure of the graph was then defined, limits were set to what the user had inputted, the graph was titled, and axes were labeled. Using the built-in scatter() and show() functions in the Matplotlib module, a scatter graph, with points from the xlist and ylist was shown.

The second phase of creating the program was adding functions to the graph to make the program more useful. The first function needed was measure_dist() that would measure the distance between two clicked points. Its one argument was "event," so that each time a point was clicked, it would trigger the function. The variable "clicks" was set to 0, and each time a point was clicked and measure_dist() called, the number would increase by one. If the number of clicks was not divisible by 2, meaning it was the first

point in the measurement, then the program would output the coordinate. On the second click, divisible by two, the program would output the second coordinate and then the distance between the two coordinates, calculated using the Pythagorean theorem.

The second function used the arrow keys on the user's keyboard to move the graph in the desired direction. After the press() function was created and the key_press_event connected to press(), four "if" statements were set up for each arrow key: right, left, up and down. In each "if" statement, first, the current x and y minimums and maximums of the window were found and the values were assigned to their respective variables. Next, the xshift and yshift, or the difference between the min and max, was found and added or subtracted from the maximum and minimum values, depending on the requested direction. For example, the right key press would add the xshift to both the x limit minimum and x limit maximum, thereby shifting the graph completely to the right. Lastly, the find_points() function was called to find the points around the new window. For the right key press, points to the right of the current window are found from the point_list. These points are not seen in the current frame, however, it makes the program faster to load the surrounding points ahead of time. These points are plotted using Matplotlib's built-in scatter() function to finish running the press() function.

Results and Conclusion

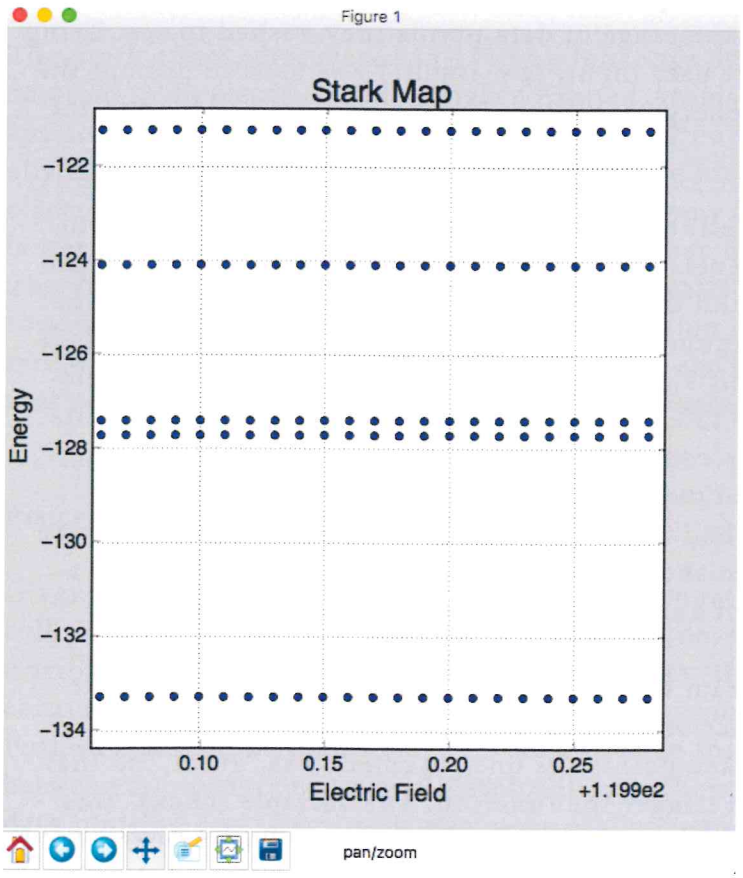

Figure 1.
Example of Code Output

In this screenshot of an output graph, you can see many different coordinate points that represents an atom's energy level at each electric field strength. The slopes of the lines are extremely small, showing the energy level changes only slightly at different electric fields. To get this graph, I inputted 120 to 120.1 V/cm for the electric field range and -128 to -127.5 cm^{-1} for the energy range. The arrow keys were used to move the graph and load more points. Lastly, the graph was zoomed out to show all the loaded points. This graph is a zoomed in version of an actual Stark map, like the one pictured below.

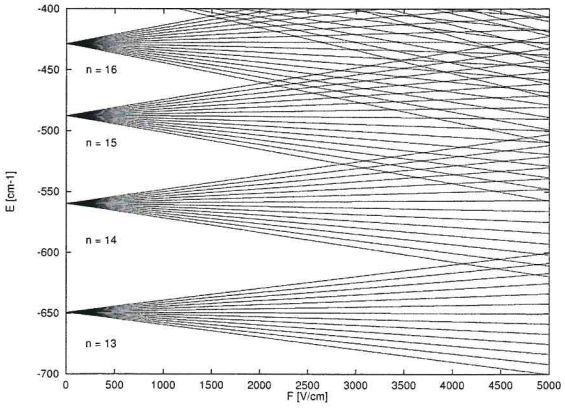

Figure 2.
Stark map of a non-chaotic Rydberg atom ("Stark Effect")

One operation that slows the current program down is unpacking the whole data file and putting coordinates into lists before the requested points are taken from the list. To make the program run faster, I would change the code to have it unpack only the requested data points. Also, this would allow more points to be loaded and shown at one time.

Creating this program has taught me many things. First, how to approach creating a program. Second, how to solve problems that seem impossible to solve. Lastly, when coding, the internet is always your best friend.

References

Carroll, Thomas. "Stark Map Software Project." Bryn Mawr College, www.brynmawr.edu/physics/MNoel/TCarroll/stark.html.

Kleppner, Daniel, et al. Highly Excited Atoms. May 1981. Scientific American, www.scientificamerican.com/article/highly-excited-atoms/.

"Stark Effect." Wikipedia, en.wikipedia.org/wiki/Stark_effect.

"Stark Effect in Atomic Spectra." Hyperphysics, hyperphysics.phy-astr.gsu.edu/hbase/Atomic/stark.html.

Acknowledgements

I would like to thank Dr. Thomas Carroll from Ursinus College and the undergraduates working in the lab for welcoming me into their space and introducing me to their work.

Appendix: Code

```
#open_data4.py
#this program opens data file, unpacks data, makes lists of point coordinates
import struct
global point_list

#open a file using the path and 'rb'(read binary) and assign to file variable
tfile = open('/Users/juliamcd/Internship/37d_0.5.dat','rb')

electric = 17250575
#electric = 17250575, which is 30001(fields) x 1150(energies)/2(coordinates/loop)
point_list = []

#this will unpack all data from file into list
while electric > 0:
   #unpack one number (8 bytes) at a time
   #make list of x,y coordinates for one point
   points = []
   x = struct.unpack('d',tfile.read(8))
   points.append(x[0])
   y=struct.unpack('d',tfile.read(8))
   points.append(y[0])
   #add list of one point's coordinates to list of all point's coordinates
   point_list.append(points)
   electric -= 1
```

```
#graph_data4.py
#this program takes the info from open_data4 and graphs the data wanted with
   #ability to move and load new data w key presses, measure distance between two points

import matplotlib.pyplot as plt
import numpy as npy
import sys
from open_data4 import point_list

#this function loads requested coordinate points into lists
def find_points(fstart, fend, estart, eend):
   global xlist, ylist
   xlist = []
   ylist = []
   #step through ALL sets of point coordinates in list (this takes awhile)
   for points in point_list[0::]:
      #append specified point to x/y lists to be graphed
      if fstart <= points[0] <= fend and estart <= points[1] <= eend:
         xlist.append(points[0])
         ylist.append(points[1])
         #remove points from list so they aren't plotted twice
         point_list.remove(points)
```

```
#this function measures the distance between two points
def measure_dist(event):
    global clicks, x1, y1
    clicks += 1
    ind = event.ind
    if clicks%2 != 0:
        #store and print coordinates of first click
        x1 = npy.take(xlist, ind)
        y1 = npy.take(ylist, ind)
        print 'x:', x1, 'y:', y1
    else:
        #store and print coordinates of second click, calculate and output distance
        x2 = npy.take(xlist, ind)
        y2 = npy.take(ylist, ind)
        print 'x:', x2, 'y:', y2
        distance = (((x2-x1)**2)+((y2-y1)**2))**0.5
        print 'distance=', distance

#this function moves the graph by responding to arrow keys
def press(event):
    global xlimmin, xlimmax
    print 'press', event.key
    sys.stdout.flush()
    if event.key == 'right':
        #get x/y limits (what is currently shown in window)
        xlimmin, xlimmax = plt.xlim()
        ylimmin, ylimmax = plt.ylim()
        #shift graph completely to the right
        xshift = (xlimmax - xlimmin)
        yshift = (ylimmax - ylimmin)
        xlimmin += xshift
        xlimmax += xshift
        plt.xlim(xlimmin, xlimmax)
        #since the points in the new limits are already plotted,
        #find the points to the right of the new limits
        find_points(xlimmin + xshift, xlimmax + xshift,
                    ylimmin - yshift, ylimmax + yshift)
        #load the points
        plt.scatter(xlist, ylist, picker = True)
        fig.canvas.draw()
    if event.key == 'left':
        xlimmin, xlimmax = plt.xlim()
        ylimmin, ylimmax = plt.ylim()
        #shift graph completely to the left
        xshift = (xlimmax - xlimmin)
        yshift = (ylimmax - ylimmin)
        xlimmin  -= xshift
        xlimmax -= xshift
        #get points to left of new limits
        plt.xlim(xlimmin, xlimmax)
```

```
            find_points(xlimmin - xshift, xlimmax - xshift,
                        ylimmin - yshift, ylimmax + yshift)
            plt.scatter(xlist, ylist, picker = True)
            fig.canvas.draw()
        if event.key == 'up':
            xlimmin, xlimmax = plt.xlim()
            ylimmin, ylimmax = plt.ylim()
            #shift graph up completely
            yshift = (ylimmax - ylimmin)
            xshift = (xlimmax - xlimmin)
            ylimmin += yshift
            ylimmax += yshift
            plt.ylim(ylimmin, ylimmax)
            #get points above new limits
            find_points(xlimmin - xshift, xlimmax + xshift,
                        ylimmin + yshift, ylimmax + yshift)
            plt.scatter(xlist, ylist, picker = True)
            fig.canvas.draw()
        if event.key == 'down':
            xlimmin, xlimmax = plt.xlim()
            ylimmin, ylimmax = plt.ylim()
            #shift graph down completely
            yshift = (ylimmax - ylimmin)
            xshift = (xlimmax - xlimmin)
            ylimmin -= yshift
            ylimmax -= yshift
            plt.ylim(ylimmin, ylimmax)
            #get points below new limits
            find_points(xlimmin - xshift, xlimmax + xshift,
                        ylimmin - yshift, ylimmax - yshift)
            plt.scatter(xlist, ylist, picker = True)
            fig.canvas.draw()

    #main

    #get start and end of field points and energy points, check validity
    valid = False
    while not valid:
        field_start = float(raw_input('Field point start?(0.00 - 300.00): '))
        if (0.00 <= field_start <= 300.00): valid = True
        else: print 'Not valid. Input a number with two decimals between 0 and 300.'

    valid = False
    while not valid:
        field_end = float(raw_input('Field points end?(0.00 - 300.00): '))
        if (0.00 <= field_end <= 300.00): valid = True
        else: print 'Not valid. Input a number with two decimals between 0 and 300.'

    valid = False
    while not valid:
```

```
    energy_start = float(raw_input('Energy start?(-138.5 to -39.08): '))
    if (-138.5 <= energy_start <= -39.08): valid = True
    else: print 'Not valid. Input a number between -138.5 and -39.08.'

valid = False
while not valid:
    energy_end=float(raw_input('Energy end?(-138.5 to -39.08): '))
    if (-138.5 <= energy_end <= -39.08): valid = True
    else: print 'Not valid. Input a number between -138.5 and -39.08.'

#field_start is xlimmin, field_end is xlimmax,
#energy_start is ylimmin, energy_end is ylimmax
xshift = field_end - field_start
yshift = energy_end - energy_start
find_points(field_start - xshift, field_end + xshift,
            energy_start - yshift, energy_end + yshift)

#create figure, define size in inches (width, height) & resolution(DPI)
fig = plt.figure(figsize = (6,6.5),  dpi = 100)
clicks = 0

#define limits(x/ymin,x/ymax), it will automatically set ticks
plt.xlim(field_start, field_end)
plt.ylim(energy_start, energy_end)

#labels axes and graph
plt.xlabel('Electric Field', size=14)
plt.ylabel('Energy', size=14)
plt.title('Stark Map', size=18)
plt.grid(True)

#plot points from x/y lists, picker allows pick_event to be used
plt.scatter(xlist, ylist, picker = True)

#override keyboard shortcuts for forward and back (usually arrow keys)
#so it won't interfere with key_press_event
plt.rcParams['keymap.back']=''
plt.rcParams['keymap.forward']=''

#connect key presses and clicks to functions and graph
fig.canvas.mpl_connect('key_press_event', press)
fig.canvas.mpl_connect('pick_event', measure_dist)

plt.show()
```