5-7-2016

Complexity and Art

Jeffrey Jenkins
Utah State University

Complexity and Art

Jeffrey Jenkins[1]
Utah State University Undergraduate

(Dated: 7 May 2016)

A python application was written with the purpose of facilitating the representation of data in both an audio and visual manner. The representations are in the artistic forms of color field paintings and music. This allows for the quick recognition of similarities and differences in the complexities of the data and art. Data was created from musical pieces to give a reference of how closely the program could recreate art. They were chosen from nursery rhymes and Mozart. Also data was generated from random white noise, 1/f, and Brownian data sets. Comparing these data sets showed the greatest similarity between Nursery Rhymes and Brownian motion, as well as between Mozart and 1/f data sets.

It is intended that the application become available to students who are beginning a study of complexity. This project is meant to establish that the application is indeed useful and the broad relationship between the power spectrum of a data set and it's resemblance to music and art. Future students should be able to compare any data set to those generated from the three most common power spectrums and approximate the spectrum of their data and see aspects that may resemble art.

I. APPLICATION DEVELOPMENT

Python was chosen as the program language in order that students with MacOS and Windows would both have access to the application. It was also desirable to use a scripting language so that if a student wishes to modify the program they just need to install Python, modify the code and run the script using the local installation of Python. While we wanted this functionality for those familiar with Python we strongly desired a self contained executable that requires no programming knowledge to use.

In consideration of these requirements the modules TKinter for the GUI and PyAudio for the audio, were chosen because of their cross-platform compatibility.

The input data format needed to be flexible enough for a wide variety of source data while being as simple as possible. It was determined to use integer and floating point pairs separated by commas as the general format. This gave enough flexibility to represent almost any music or color field. The integer is used to map to a note or a color respectively and the decimal number is the length of the note or the width of the color box. Instead of entering the data into the application GUI the data is expected as a text file and only the file path using the location of the app as the root folder is inputted to the GUI. This allows for the application as well as all data to be located in a single folder and can be moved or transferred with no complications.

Finally in order to distribute the application as an executable instead of only a script the code needed to be "frozen". I chose to use cx_Freeze for the Mac version and py2exe for the windows version. The script in each case is almost identical due to trying to use cross-platform modules as much as possible. The only difference is in how the operating systems address files. So making the app location the root location when inputting file addresses to the GUI required OS specific code.

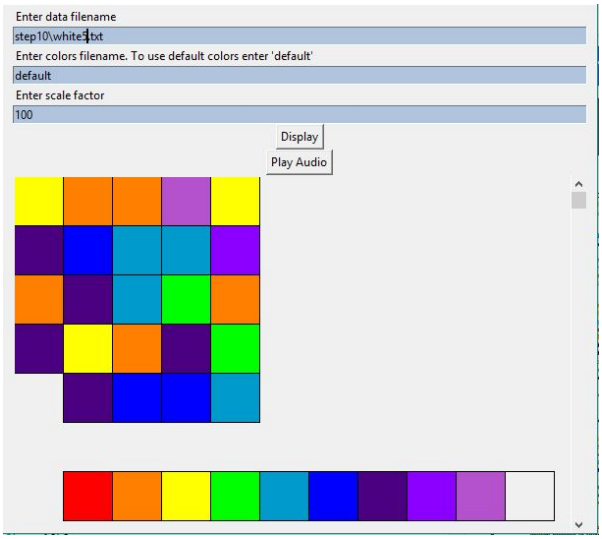

FIG. 1. Image of the running (Windows)application.

A. Future Development

Improvements to the application would include having the ability to comment lines in the data files. Often the title of the file is sufficient but more info on how the data was obtained can only improve readability and record keeping. I would also add an option for pausing at the audio at the end of a line or not. The pause helps to follow along between the audio and visual representation but most music doesn't have a pause for each line and if the data was organized in lines arbitrarily not because of something intrinsic in the data the pause puts a lot of emphasis on something not directly in the data. Cur-

rently the user can provide a color file as an alternative to the default color scheme but not an alternative notes file. Adding this feature would allow the user to try different scales, add sharps or flats to match the music and increase flexibility in general. Also I would include the ability to change the speed the music is played at. This would correspond to the existing ability to scale the color field image so that more or less data is visible at once. Currently too much adjusting has to happen manually in the source file. It would be preferable to be able to adjust these features in the GUI instead of the more permanent way of adjusting values in the file for example making the length of all of the notes twice as long. I also want to consider an adjustment field that would allow the user to adjust all of the notes by a fixed amount. For example increasing the pitch of the entire piece by one note. These features would increase the flexibility of the application and speed up the process of adjusting the representation without modifying the underlying data.

II. RANDOM DATA SET

The power spectrum is an important feature in many data sets and applications. It comes from taking the Fourier transform of the data, squaring it and doing a curve of best fit to the amplitude coefficients graphed against the frequency. Instead of performing this process on many data sets until enough with similar curves of best fit have been analyzed to understand the characteristics of data with that type of power spectrum curves, I artificially created data directly that would match a specific power spectrum exactly. Three forms were analyzed. White noise where the power spectrum is flat (f^0), $1/f$ which is typically characteristic of music, and Brownian motion ($1/f^2$). Music is typically near 1/f because white noise feels too random and unpleasant and Brownian changes too slowly so that it feels like scales. 1/f is a balance of predictability. The technique for generating the data set is to sum 40 sine functions with equally spaced frequencies. Assign their amplitudes according to the power function of interest and generate a different random phase for each of the 40 sine functions. Based on the random phases many different data sets can be created all with the same power spectrum.

Finally because there are finite defined colors and notes instead of a continuum sampling frequency and a bucket width have to be chosen. The bucket widths are chosen by dividing the range of the output by the desired number of notes. Also for some peices I used the same technique to determine note lengths independently. While the software accepts a continuum of values for the length in imitation of music I mapped it to only 1/8, 1/4, 1/2 and whole notes.

The sampling frequency choice is less well understood in this case the frequencies varied from 1 to 40 and every tenth unit of 'time' was the sample rate. The difficulty comes in because of the self similarity between the different power spectrums. If the white noise function is rescaled to stretch the time axis then it appears more like the 1/f function, and if the 1/f function is rescaled it appears more like the Brownian function. So choosing a sample frequency very large makes the data appear more Brownian and a sample frequency very small makes the data appear more white. I'm unsure how to quantify this affect so a sample rate was chosen and it was verified that data from the three distributions was distinguishable.

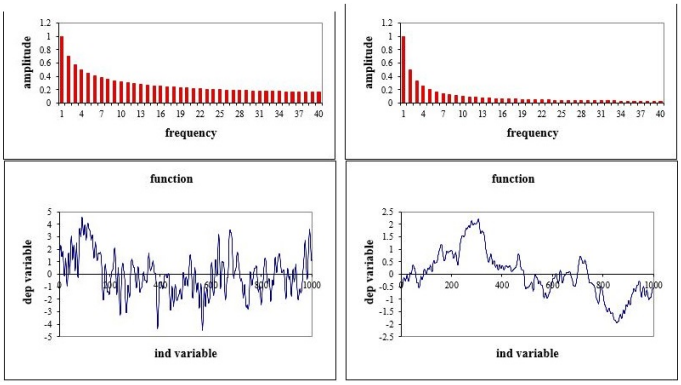

FIG. 2. Example 1/f and $1/f^2$ data sets.

FIG. 3. Scale for generating the color field images.

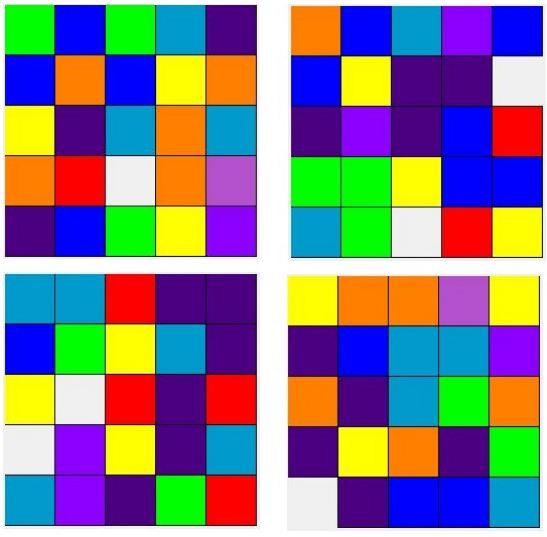

FIG. 4. White noise generated image.

Figures 4, 5, and 6 show the differences between the power distributions that generated them. The white noise examples have significantly less repetition and appear more random in the colloquial sense. The Brownian sets have the most repetition and rarely jumps more than

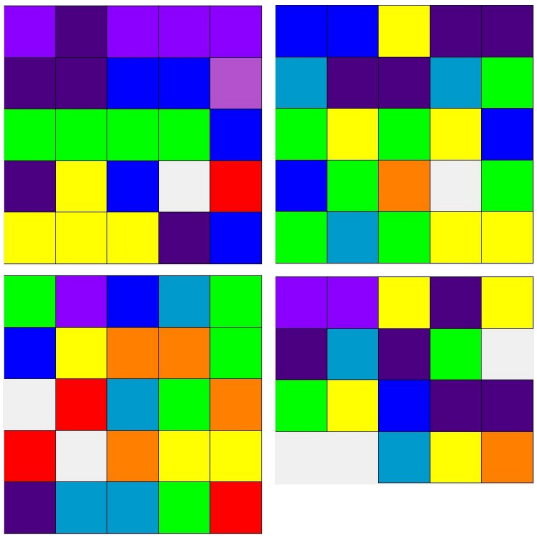

FIG. 5. 1/f generated images.

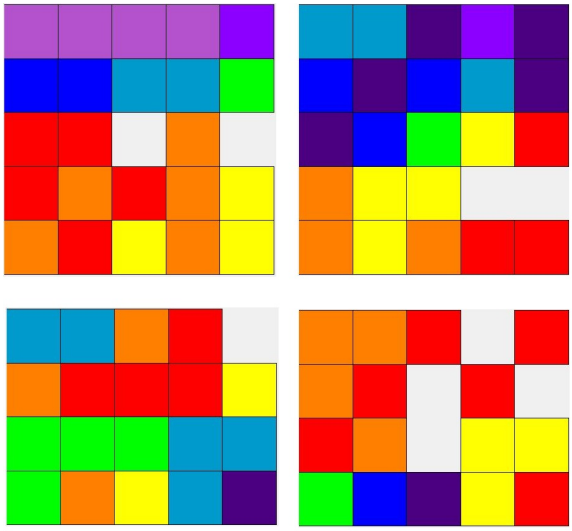

FIG. 6. Brownian motion generated images.

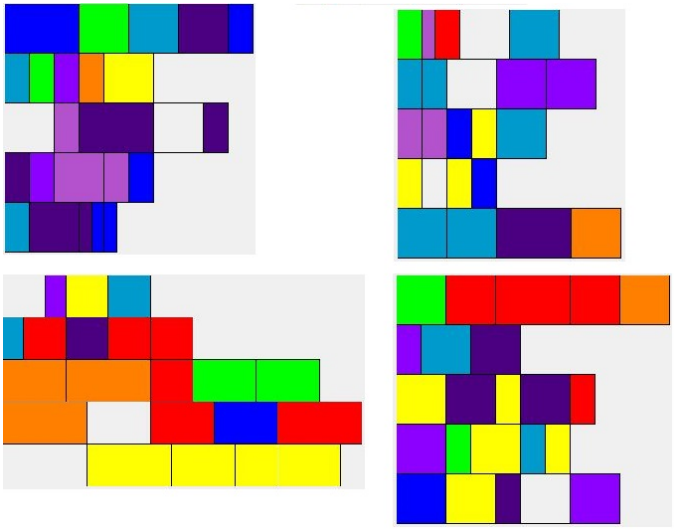

FIG. 7. Independent 1/f data sets used for the note pitch and length.

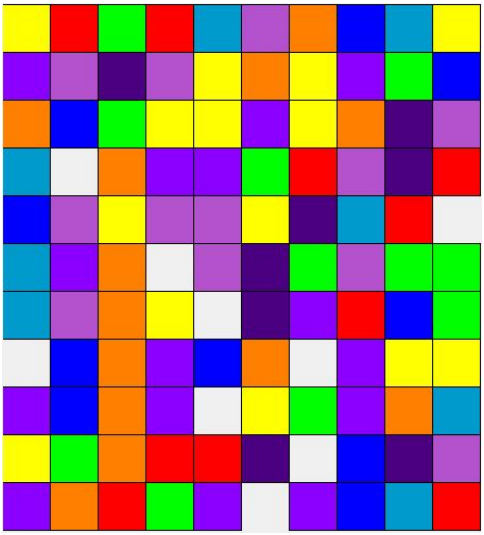

FIG. 8. 100 digits of PI.

one or two colors on the scale, while the 1/f examples are in between the two.

III. REPRESENTATIONS OF REAL DATA SETS.

A real world example of data that is expected to be similar to white noise is the digits of PI. The digits are truly random, there is no reason to expect digits to remain close to the previous digit. This expectation bears out in the visual appearance of figure 8.

An important goal is to confirm the expectation that music is 1/f. Now that characteristic examples of 1/f have been analyzed, characteristic musical examples need to be analyzed for comparison. I chose two pieces of Mozart's as well as two Nursery rhymes. I expected the nursery rhymes to have more repetition but still have 1/f appearance in each repetition but it actually resembles the Brownian motion sets the most. In retrospect this is not surprising that nursery rhymes or lullabies would be more regular and predictable than other music. Mozart's pieces appear very similar to the when a 1/f distribution is used for both the note pitch and length. When Mozart had a series of shorter notes the pitch didn't change very much. This is not true for the randomly generated pieces because the pitch and length are independently generated. Also music has the regularity of measures. This can be seen in figures 11 and 12 as where a note ends in the same place on every row. It appears as a vertical break across multiple lines. Clearly that feature is not present in the randomly generated sets. While the measures are very obvious in the visual representation it is

unnoticeable in audio form. The ability to see features that are difficult to see in the other form is the principle motivation for using both types of representation. Also the music gives a very linear presentation from beginning to end where the visual representation allows for observing the entire data set at once which can show large scale features clearly. Random walks where something has a %50 chance of going up or down is an alternative characterization of Brownian motion. So I created data sets from flipping coins and increasing or decreasing the pitch for heads and tails respectively. Figure **??**, generated using this method appears Brownian except for some overly regular characteristics. The note never remains the same and never jumps by more than a single note. Perhaps it would appear more similar to the first characterization of Brownian motion if the 'sampling rate' were cut in half. Then there would be a %50 chance of the note staying the same and %25 chance of increasing or decreasing.

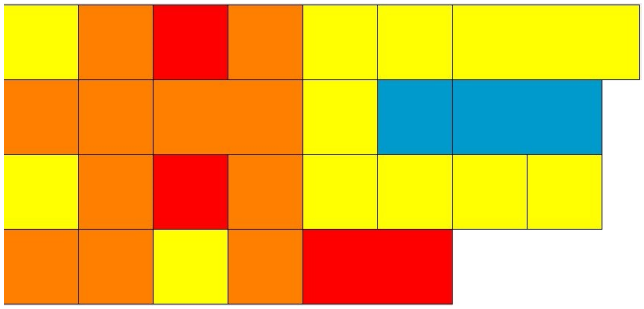

FIG. 9. Mary had a Little Lamb.

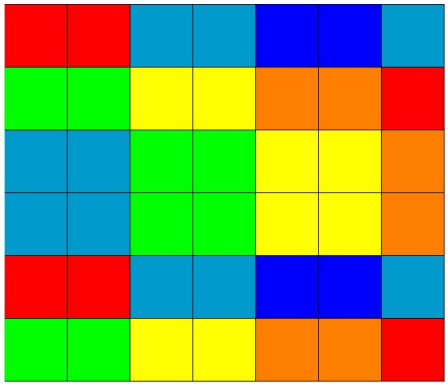

FIG. 10. Twinkle Twinkle Little Star.

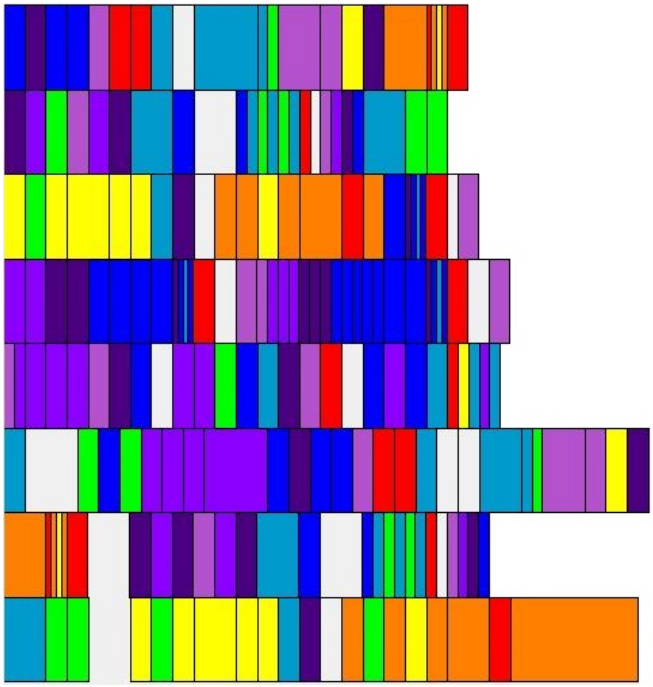

FIG. 11. Mozart's 23 Piano Concerto.

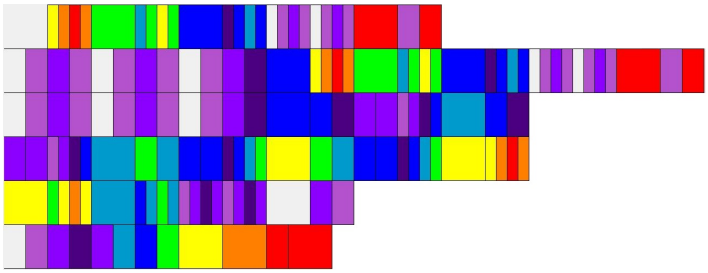

FIG. 12. Mozart's 23 Piano Concerto.

IV. CONCLUSION

This supports the idea that most music has a 1/f power spectrum. This is shown in the color field representation and in the audio representation that can't be shown in this format. The differences due to the regularity of measures is less apparent in the audio format. Also as might be expected children's music is more regular and greatly resembles Brownian motion. The application I wrote was instrumental in making this analysis faster and being able to do it in two very different formats. While some deficiencies were noticed as I used the application it is already a useful tool. After some additional features are added it will be a truly robust tool that students beginning in studying complexity will be able to easily compare any data they collect with features found in nature, art and music.

V. APPLICATION CODE

```
from tkinter import *
from tkinter import messagebox
import pyaudio
import wave
import sys
import math
import threading

class App(Frame):
    CHUNK = 1024
    play = True
    def __init__(self, master=None):
        Frame.__init__(self, master)
        self.pack(ipadx=20)
        #creates widgets and the variables that are
            attached to the Entry widgets
        self.createWidgets(master)
        #I call getColors in initialization and in
            'display'. This is because I want
        #to load the default first and then if they
            provide a valid file I'll overwrite
            default.
        self.colorArray = self.getColors()
        self.noteScale = self.getNotes()
    def createWidgets(self, master):
        self.filenameLabel = Label(master,
            text="Enter data filename", anchor="w")
        self.filenameLabel.pack(fill="x", padx = 10)

        self.dataFileEntry = Entry(width=250,
            selectborderwidth = 10)
        self.dataFileEntry["relief"] = "groove"
        self.dataFileEntry["bg"] = "#B0C4DE"
        self.dataFileEntry.pack(padx = 10)
        # here is the application variable
        self.filename = StringVar()
        # set it to some value
        self.filename.set("mary.txt")
        # tell the entry widget to watch this
            variable
        self.dataFileEntry["textvariable"] =
            self.filename

        self.colorLabel = Label(master, text="Enter
            colors filename. To use default colors
            enter \'default\'", anchor="w")
        self.colorLabel.pack(fill="x", padx = 10)

        self.colorFileEntry = Entry(width=250,
            selectborderwidth = 10)
        self.colorFileEntry["relief"] = "groove"
        self.colorFileEntry["bg"] = "#B0C4DE"
        self.colorFileEntry.pack(padx = 10)
        # here is the application variable
        self.colorfilename = StringVar()
        # set it to some value
        self.colorfilename.set("default")
        # tell the entry widget to watch this
            variable
        self.colorFileEntry["textvariable"] =
            self.colorfilename

        #scale variable
        self.scaleLabel = Label(master, text="Enter
            scale factor", anchor="w")
        self.scaleLabel.pack(fill="x", padx = 10)

        self.scaleEntry = Entry(width=250,
            selectborderwidth = 10)
        self.scaleEntry["relief"] = "groove"
        self.scaleEntry["bg"] = "#B0C4DE"
        self.scaleEntry.pack(padx = 10)
        # here is the application variable
        self.scale = StringVar()
        # set it to some value
        self.scale.set(100)
        # tell the entry widget to watch this
            variable
        self.scaleEntry["textvariable"] = self.scale

        # I want to run display on 'Key-Return' or
            clicking 'done'
        self.dataFileEntry.bind('<Key-Return>',
            self.display)
        self.colorFileEntry.bind('<Key-Return>',
            self.display)
        self.scaleEntry.bind('<Key-Return>',
            self.display)
        self.displaybutton = Button(master,
            text="Display", command=self.display)
        self.displaybutton.pack()
        self.audiobutton = Button(master,
            text="Play Audio",
            command=self.playaudio)
        self.audiobutton.pack()

        self.paintingcanvas = Canvas(master,
            width=100, height=1000)
        myscrollbar=Scrollbar(self.paintingcanvas,
            orient="vertical",
            command=self.paintingcanvas.yview)
        myscrollbar.pack(side="right", fill=Y)
        self.paintingcanvas.pack(fill="both",padx=10,
            side="left", expand=True)

    def playaudio(self, event = None):
        if(self.audiobutton["text"] == "Play
            Audio"):
            self.audiobutton["text"] = "Stop"
        else:
            self.play = False
            self.audiobutton["text"] = "Play Audio"
            return
        self.play = True
        playThread = soundThread(self, 1,
            "playThread1")
        playThread.start()

    def display(self, event = None):
        self.colorArray = self.getColors()
        self.paintingcanvas.delete(ALL)
        #Read datafile into double array
        try:
```

```
            file = open(self.filename.get(),
                'r')#'r' means read-only mode
        except:
            messagebox.showwarning("Open
                file","Cannot open this file:\n(%s)"
                % self.filename.get())
            return

        matrix = []
        for line in file:
            matrix.append([[float(x) for x in
                pair.split()] for pair in
                line.split(",")])
            #matrix.append([int(x) for x in
                line.split()])
        #Draw representation of Data


        for list in matrix:
            for pair in list:
                print(pair)
            print("\n")


        x = y = 1   #because i don't have an
             iteration counter, I can't do
        #scaleNum=100
        try:
            scaleNum=abs(int(self.scale.get()))
        except:
            messagebox.showerror("Error","Enter only
                integer values for the scale.")
        size = 50*scaleNum/100  #something like:
             i*size, j*size. That's why x,y are in a
             large scope.
        maxCat = 0
        for list in matrix:
            for pair in list:
                if not pair:
                    continue
                self.paintingcanvas.create_rectangle(x,
                    y, x+size*pair[1], y+size,
                    fill=self.colorArray[int(pair[0])
                    %len(self.colorArray)],
                    outline="" if pair[0] ==0 else
                    "#000000")
                x=x+size*pair[1]
                maxCat=max(maxCat, pair[0])
            x=1
            y=y+size
        if maxCat-1 > len(self.colorArray):
            messagebox.showinfo("Color Info",
                "Insufficient colors for all
                categories. \n Some categories will
                share colors.")

    #This function returns an array with note
         frequencies to map integers to notes
    def getNotes(self):
        notes = []
        notes.append(1046.50)#C6
        notes.append(1174.66)#D6
        notes.append(1318.51)#E6
        notes.append(1396.91)#F6
        notes.append(1567.98)#G6
        notes.append(1760)#A6
        notes.append(1975.53)#B6
        notes.append(2093)#C7
        notes.append(2349.32)#D7
        notes.append(2637.02)#E7
        notes.append(2793.83)#F7
        notes.append(3135.96)#G7
        notes.append(3520)#A7
        notes.append(3951.07)#B7

        return notes

    #this function assigns a color value to each
         integer value we will
    #be using as categories in our complexity
         representation.
    def getColors(self):
        colors = []
        try:
            file = open(self.colorfilename.get(),
                'r')#'r' means read-only mode
            for line in file:
                colors.append(line)
        except:
            if self.colorfilename.get() != "default":
                messagebox.showwarning("Open
                    file","Cannot open this
                    file:\n(%s)\nUsing default
                    colors" % self.filename.get())
            colors.append("")#0 transparent
            colors.append("#FF0000")#1
            colors.append("#FF7F00")#2
            colors.append("#FFFF00")#3
            colors.append("#00FF00")#4
            colors.append("#009ACD")#5
            colors.append("#0000FF")#6
            colors.append("#4B0082")#7
            colors.append("#8B00FF")#8
            colors.append("#B452CD")#9
        return colors
class soundThread (threading.Thread):
    def __init__(self, master, threadID, name):
        threading.Thread.__init__(self)
        self.master = master
        self.threadID = threadID
        self.name = name
    def run(self):
        #Read datafile into double array
        try:
            file = open(self.master.filename.get(),
                'r')#'r' means read-only mode
        except:
            messagebox.showwarning("Open
                file","Cannot open this file:\n(%s)"
                % self.master.filename.get())
            return

        matrix = []
        for line in file:
            matrix.append([[float(x) for x in
                pair.split()] for pair in
                line.split(",")])
```

```
RATE = 48000
p = pyaudio.PyAudio() #Maybe this shouldn't
    be in every call, it could be global
    instead.
#open stream
stream = p.open( format =
    p.get_format_from_width(1),
       channels = 1,
       rate = RATE,
       output = True)
#write data
for list in matrix:
   for pair in list:#pair is the note first
       and then the duration, 1 = .5
       seconds.
       if not pair:
           continue
       if (self.master.play):
           #int(RATE/2) makes each note half
               a second.
           freq = (self.master.
               noteScale[int(pair[0])])
           data = ''.join([chr(int(
               math.sin(x/((RATE/freq)/math.pi))
               *127+128)) for x in
               range(int(RATE/2*pair[1]))])
           stream.write(data)
       else :
           self.master.play = True
           stream.stop_stream()
           stream.close()
           p.terminate()
                   self.master.audiobutton["text"] =
                       "Play Audio"
                   return
               data = ''.join([chr(0) for x in
                   range(int(RATE/2))])
               stream.write(data)#this puts a
                   half-second of silence at the end of
                   each row in the data.

       stream.stop_stream()
       stream.close()
       p.terminate()
       self.master.audiobutton["text"] = "Play
           Audio"

# create the application
root = Tk()
myapp = App(master = root)

#
# here are method calls to the window manager class
#
myapp.master.title("Complexity App")
myapp.master.minsize(400, 250)
myapp.master.maxsize(root.winfo_screenwidth(),
    root.winfo_screenheight())

# start the TK event loop
myapp.mainloop()
```