# VISUALISING ORBITALS USING ELECTRON DISTRIBUTIONS

CW3 FP

Jack Bauermeister

Emma Cooper

Contents

[VISUALISING ORBITALS USING ELECTRON DISTRIBUTIONS 1](#_Toc166592273)

[PROJECT PROPOSAL 2](#_Toc166592274)

[Project Overview 2](#_Toc166592275)

[Project Aims 2](#_Toc166592276)

[PLANNING 4](#_Toc166592277)

[Initial Planning 4](#_Toc166592278)

[Detailed Process Planning 5](#_Toc166592279)

[CODE IMPLEMENTATION 13](#_Toc166592280)

[TESTING 13](#_Toc166592281)

[Goals 13](#_Toc166592282)

[Abstract Testing 13](#_Toc166592283)

[System Testing 13](#_Toc166592284)

[CONCLUSION 13](#_Toc166592285)

## PROJECT PROPOSAL

### Project Overview

This project aims to allow the user to select an orbital through a graphical user interface (GUI) and select a visual representation of the orbital (1D, 2D, or simple rendered model), which is then displayed to the user through the GUI. The program should have stored rendered models that can be pulled from the directory and displayed and should be able to calculate and graph both the 1D and 2D representations using radical wavefunctions and distribution functions.

### Project Aims

The following aims for this project cover inputs, any GUIs, calculations, and outputs, that should be achieved through the program code. These aims can be used as guidelines for the final testing of both abstract selection of code and the overall final version.

|  |  |  |
| --- | --- | --- |
| **Selection/Function** | **Overview** | **Aims** |
| **Inputs**   * Orbitals type * Visual representation | The user should give both the orbital shape (principle quantum number, and angular momentum) and a visual representation of the orbital. | * Selection buttons for the orbital shape. * Selection buttons for the visual representation. * A button to start the generation of the graph or pull the rendered orbital from the directory. |
| **GUIs** | The GUIs should be aesthetically appealing, clear, and simple for the user to understand. The output and input GUIs should appear in separate windows. | * Labels/prompts should provide clear and understandable instructions to the user. * Any elements in the GUIs should not merge due to colour or overlap/interception. * Should have separate windows to collect the inputs and display output to the user. |
| **Calculations**   * Probability distribution of electrons in orbital | Using the inputs provided through the GUI, if required the probability distribution should be calculated and then graphed in sensible units and in a format for the user. | * Should identify the representation selected. * Should calculate the values to graphed if required. * Should create a graph in a sensible format, including title, axis titles, and units. |
| **Outputs**   * Graph/or Rendered model | Should give the visual representation selected into the output GUI for the user. | * Should produce the visual representation selected, graph, or rendering from the directory. * Should provide any variables/constants used in any calculations completed to the user through the GUI. |

A computer screen with a white screen

Description automatically generatedA screenshot of a computer

Description automatically generatedBelow are examples of what the GUIs (graphical user interface) could look like according to the aims stated.

Figure 1. An example of the output window GUI.

Figure 2. An example of the input window GUI.

## PLANNING

### Initial Planning

Before coding starts, planning out the steps to produce the required output and goals is essential. To do this effectively, a flowchart has been used to plan the basic processes and any key points or functions within the process, which will then be broken down further into proposed pseudocode for each key step which can be exampled in further detail, explaining, and describing the methods and techniques used.

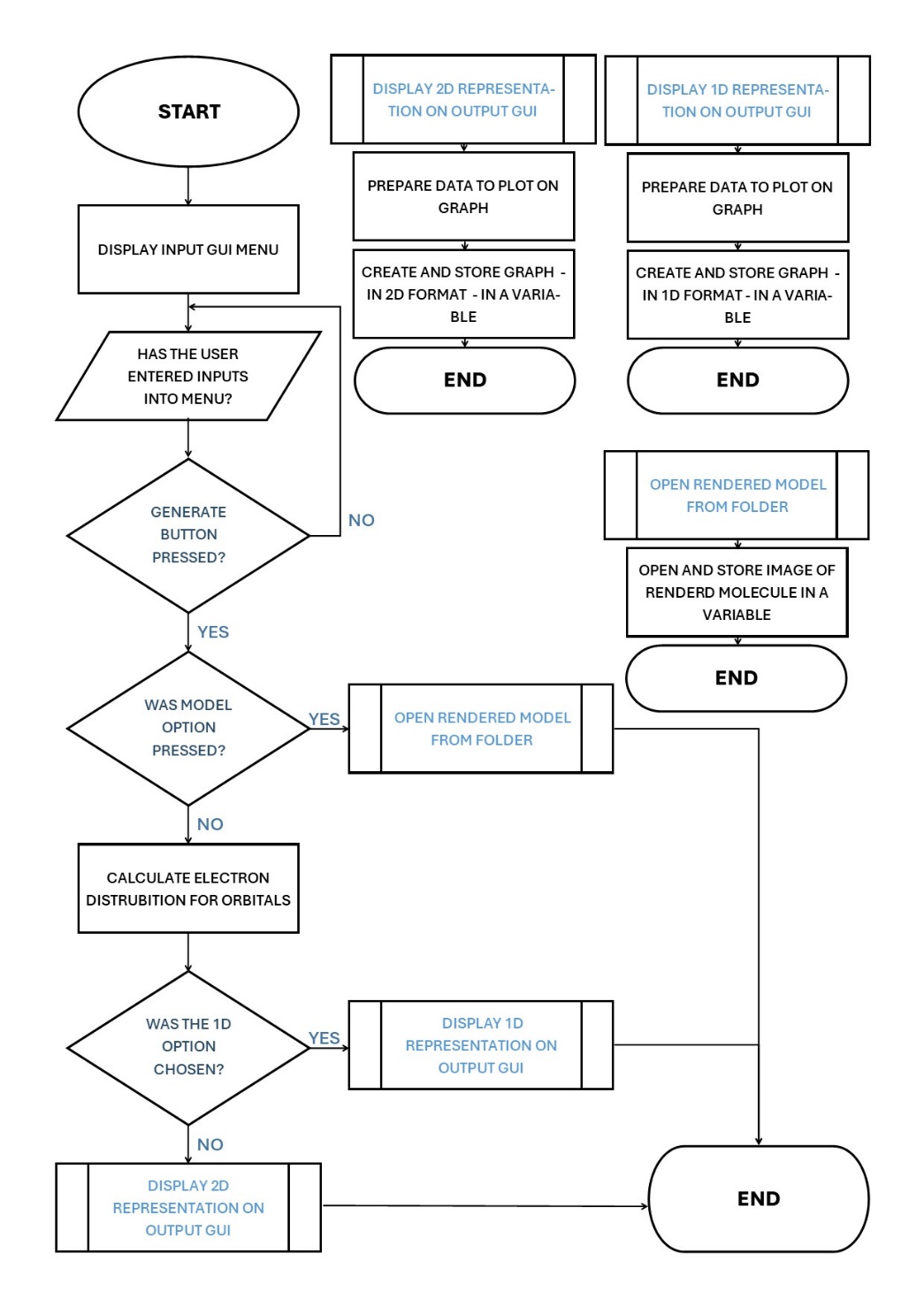
The following flowchart shows the overall steps that will be followed, as well as separate functions. This flowchart can also be used as a structure for the cod layout. It shows what processes and functions need to be completed for the following selections of code to work. For example, the GUI must be used to get the users' inputs before calculating the electron distributions, even if default values are set within the code for any constants or variables.

Figure 3. A flowchart showing the overall structure and key points for the final code to roughly follow.

Now, that an initial plan has been completed, the key points should be planned in further detail, in pseudocode. The specific areas that should be planned in greater detail are:

* Input and Output GUIs.
* Opening and storing the image file of the rendered molecule.
* Calculating the electron distribution
* Plotting the electron distribution in a 1D graphical format.
* Plotting the electron distribution in a 2D graphical format.

### Detailed Process Planning

#### Calculations and Plotting

To calculate the electron distribution and therefore plot a 1D and 2D representation of the select electron orbitals. We will need to create a function that calculates the electron distribution based on the principal quantum number, angular momentum, and radius range inputted by the user of the GUI. I will also create functions that plot the 1D and 2D visualization of the electron distribution calculated in the previously created function. I will use a maths plotting library to achieve this.

Below is the proposed pseudocode for this process:

import numpy as np

import matplotlib.pyplot as plt

from scipy.constants import physical\_constants

def radial\_wavefunction(n, l, r):

a0 = physical\_constants['Bohr radius'][0] \* 1e10 # Convert Bohr radius to angstroms

Z = 1 # Assume Z=1 for hydrogen-like orbitals in a water molecule approximation

if n == 1 and l == 0: # 1s

return 2 \* (Z/a0)\*\*1.5 \* np.exp(-Z \* r / a0)

elif n == 2 and l == 0: # 2s

return (1/4\*np.sqrt(2)) \* (Z/2/a0)\*\*1.5 \* (2 - Z\*r/a0) \* np.exp(-Z\*r/(2\*a0))

elif n == 2 and l == 1: # 2p

return (1/(2\*np.sqrt(6))) \* (Z/2/a0)\*\*1.5 \* (Z\*r/a0) \* np.exp(-Z\*r/(2\*a0))

return np.zeros\_like(r)

…

def plot\_orbitals(n, l, r\_min, r\_max, plot\_type):

r = np.linspace(r\_min, r\_max, 400)

R = radial\_wavefunction(n, l, r)

if plot\_type == '1D':

plt.figure(figsize=(8, 4))

plt.plot(r, R\*\*2, label=f'n={n}, l={l}', linewidth=2)

plt.title(f'Radial Probability Distribution: {n}{["s", "p"][l]} Orbital')

plt.xlabel('Radius (angstroms)')

plt.ylabel('Probability Density')

plt.legend()

plt.grid(True)

plt.show()

elif plot\_type == '2D':

x = np.linspace(-r\_max, r\_max, 400)

y = np.linspace(-r\_max, r\_max, 400)

X, Y = np.meshgrid(x, y)

r = np.sqrt(X\*\*2 + Y\*\*2)

if l == 0:

Z = radial\_wavefunction(n, l, r)

elif l == 1:

Z = radial\_wavefunction(n, l, r) \* X / r

plt.figure(figsize=(8, 8))

plt.pcolormesh(X, Y, Z\*\*2, shading='auto', cmap='inferno')

plt.title(f'Orbital Shape in 2D: {n}{["s", "p"][l]} Orbital')

plt.xlabel('X (angstroms)')

plt.ylabel('Y (angstroms)')

plt.axis('scaled')

plt.colorbar(label='Probability Density')

plt.xlim(-2,2)

plt.ylim(-2,2)

plt.show()

…

#### Input and Output GUIs

…

# Example usage to plot 1s, 2s, and 2p in 1D and 2D

orb\_type = "1\_s"

plot\_type = '1D'

n, l = (int(orb\_type.split('\_')[0]), 0 if 's' in orb\_type.split('\_')[1] else 1)

print(f"Plotting {orb\_type} in {plot\_type} view")

plot\_orbitals(n=n, l=l, r\_min=0, r\_max=10, plot\_type=plot\_type)

For this program, a GUI should be used to gather the user’s inputs and show the user the selected representation and output of calculations, with all variables used. Both the input and output GUI should be in separate windows to allow the user to select new inputs while seeing the previous output at the same time. The output window should either pop up or update whenever a button is pressed by the user to update the output.

In a GUI, many of the widgets use the same parameter, because of this a class could be used to simplify the creation, building, and packing of each widget in the GUI windows. The class should collect the parameter consistent across all widgets, and then a daughter class should be created for any widgets requiring different or additional parameters, but still inheriting the parent’s class attributes. The class should contain any functions to build and pack the widgets onto the GUI windows.

Additionally, a graphical interface library could be imported and therefore used to create both the input and output GUIs.

Below is the proposed pseudocode of this process:

from graphicimport \* *//’graphic’ represents a GUI library that should be used.*

function run(orb\_type, plot\_type, canvas, pqn\_label, amn\_label):

pqn\_label.config(text = f"Principle Quantum Number:{n:>13.0f}") *//Updating the values on the output window after the user input*

amn\_label.config(text = f"Angular Momentum Number:{l:>10.0f}")

return //All code that is required to initiate the main body of the program

…

class elements:

*//General class with all attributes common to many elements and functions for these elements in Tkinter.*

function \_\_init\_\_ (self, window, geo, colour, co\_ords):

self.height = geo[1]

self.width = geo[0]

self.colour = colour

self.window = window

self.co\_ords = co\_ords

self.font = "Arial Rounded MT Bold"

self.fontsize = 14

function build\_canvas(self)

*//Creates canvas from the common attributes*

\_canvas = Canvas(self.window, bg = self.colour, width = self.width, height = self.height)

\_canvas.pack()

\_canvas.place(y -= self.co\_ords[1], self.co\_ords[2])

Return \_canvas

class label(elements):

//Daughter class for labels – extra attribute (text) and functions

function \_\_init\_\_(self, window, geo, colour, co\_ords, text):

elements.\_\_init\_\_(self, window, geo, colour, co\_ords)

self.text = text

function build(self):

//Builds label using graphic interface library

\_label = Label(self.window, text = self.text, width = self.width, height = self.height, bg = self.colour, font = (self.font, str(self.fontsize)))

\_label.pack()

\_label.place(y -= self.co\_ords[1], self.co\_ords[2])

return \_label

…

…

…

class dropdwon(elements):

//Daughter class for dropdown – extra attribute (options) and functions

function \_\_init\_\_(self, window, geo, colour, co\_ords, options):

elements.\_\_init\_\_(self, window, geo, colour, co\_ords)

self.options = options

function build(self):

//Builds dropdown menu using graphic interface library

user\_input = StrongVar()

user\_input.set(self.options[0])

\_dropdown= OptionMenu(self.window, user\_input, \*self.options)

\_dropdown.config(width = self.width, height = self.height, bg = self.colour, font = (self.font , str(self.fontsize-4)))

\_dropdown.pack()

\_dropdown.place(y -= self.co\_ords[1], self.co\_ords[2])

return user\_input

class button(elements):

*//Daughter class for buttons – extra attribute (text and function) and functions*

function \_\_init\_\_(self, window, geo, colour, co\_ords, text, function):

elements.\_\_init\_\_(self, window, geo, colour, co\_ords)

self.text = text

self.funtion = funtion

function build(self):

*//Builds button* using graphic interface library

\_button = Button(self.window, text = self.text, width = self.width, height = self.height, bg = self.colour, font = (self.font, str(self.fontsize)), command = self.function)

\_button.pack()

\_button.place(y -= self.co\_ords[1], self.co\_ords[2])

…

…

function update\_graphic\_box(graphic\_box, pic\_file): *//Function that updates canvas to contain image file given*

img = PhotoImage(file = pic\_file, master = output\_window)

graphic\_box.create\_image(0, 0, image = img, anchor = “nw”)

graphic\_box.image=img

return

*//Creates window for the input GUI*

input\_window = Graphic()

input\_window.geometry("550x450")

input\_window.title("INPUT WINDOW")

input\_window.configure(bg = 'GreyGreen')

*//Creates window for the output GUI*

output\_window = Graphic()

output\_window.geometry("600x500")

output\_window.title("OUTPUT WINDOW")

output\_window.configure(bg = 'GreyGreen')

n = 2 *//Pretend variable that will be filled in by other selections of code*

l = 2 *//Pretend variable that will be filled in by other selections of code*

*//Creates all the widgets as objects for the output window*

graphics\_box = elements(output\_window, [550,300], 'white smoke', [25,25])

pqnumber\_label = label(output\_window, [30,1] , 'DarkSeaGreen3', [25,350], (f"Principle Quantum Number:{n:>13.0f}"))

amnumber\_label = label(output\_window, [30,1] , 'DarkSeaGreen3', [25,375], (f"Angular Momentum Number:{l:>10.0f}"))

…

Handling of an Image File

…

*//Builds all the widgets for the output window*

graphic\_box\_built = graphics\_box.build\_canvas() *//Storing graphic library objects ion variable so widgets can be updated later*

pqnumber\_label \_built = pqnumber\_label.build()

amnumber\_label\_built = amnumber\_label.build()

n = 2 *//Pretend updated variable that will be filled by other selections of code*

l = 2 *//Pretend updated variable that will be filled by other selections of code*

*//Creates all the widgets as objects for the input window*

orb\_label = label(input\_window, [45,2] , ' GreyGreen ', [10,10], "PICK THE ORBITAL TYPE:")

represent\_label = label(input\_window, [45,2] , ' GreyGreen’, [10,150], "PICK THE REPRESENTATION TYPE:")

orb\_choice = dropdown(input\_window, [25,3], 'light grey', [160, 75], ["1\_s", "2\_s", "2\_p"])

represent\_choice = dropdown(input\_window, [25,3], 'light grey', [160, 225], ["1D", "2D", "RENDERING"])

generate\_button = button(input\_window, [15,3], "light grey", [210, 350], "GENERATE", lambda: run(orb\_type.get(), plot\_type.get(), graphic\_box\_built, pqnumber\_label \_built, amnumber\_label\_built))

*//Builds all the widgets for the input window*

orb\_label.build()

represent\_label.build()

orb\_type = orb\_choice.build()

plot\_type = represent\_choice.build()

generate\_button.build()

input\_window.mainloop() *//Runs window, checking for nay updates or interactions with the window*

output\_window.mainloop()

One option that the user can use to show orbitals, is a pre-rendered image of the orbital posted into the output GUI. For this, the correct file name for the orbital selected by the user should be used to open and then the image stored in a variable so that it can be called and displayed to the user through the output GUI. This should be created as a function that can be called at any point within the code so that once the user has pressed the generate button the function can be called.

As the images of the orbitals are pre-generated, they should be stored in a separate directory specifically for these images, and therefore the image should be called from this directory. This may lead to the path for the image file needing to be specified when the image file is opened.

In addition, the graphs that are created presenting the orbital selected in a 1D or 2D format should be saved under a sensible name that will not be repeat on recalculation, so the user can look or use the graph outside of the program. This should be done by calling the plot as a parameter then saving and return the filename. The filename should include the orbital type, display format, and the date and time of creation.

Below is the proposed pseudocode for this function:

import image *//’image’ represents a library used for handling image files within the code script*

import datatime

function open\_orbital\_image(orb\_type): *//Function to be called when rendered orbital image is needed*

dir\_name = “Rendered\_orbitals\\”

try:

filename = str(orb\_type + “\_rendered\_orbital.png” *//Requires as all photos of rendered orbitals to be named in a specific format*

pic\_file = image.open(dir\_name+filename) *//Opens file using filename created in variable and directory name defined in variable ‘dir\_name’*

return pic\_file

except:

print(“Error: No file found”) *//If the file is not found, prints error message to the shell, the main use for this debugging code during development*

return *//Returns from function*

…

function create\_filename(orb\_type, plot\_type): *//Function to be called when new filename is needed*

return (orb\_type + “\_” + plot\_type + “\_” + str(datetime.today()) + “\_” + str(datatime.now()))

*//Return filename in format: orbital type\_plot type\_date\_time*

function save\_plot(plot, orb\_type, plot\_type) *//Function to save a plot given to function*

filename = create\_filename(orb\_type, plot\_type)

plt.savefig(filename + ”.png”)

return filename + ”.png” *//Returns the filename that plot is saved as*

function open\_plot\_image(filename): *//Function to open a plot image under the filename given*

pic\_file = image.open(filename)

return pic\_file *//Return variable that open image is saved under filename*

## CODE IMPLEMENTATION

[Code with any additional explanations or notes, not including in code comments]

## TESTING

### Goals

[Summary of the goals the code is tested against, should be similar to the aims]

### Abstract Testing

[Testing of any functions or specific processes individually – apart from the overall code]

### System Testing

[Testing the whole of code including as functions or selection of code tested in abstract testing]

## CONCLUSION