Document version: 1.1 (2015-11-15)

Curtin University – Department of Computing

Assignment Cover Sheet / Declaration of Originality

Complete this form if/as directed by your unit coordinator, lecturer or the assignment specification.

Last name:	Kongo	Student ID:	21473719
Other name(s):	Kevin		
Unit name:	Fundamentals of Programming	Unit ID:	COMP1005
Lecturer / unit coordinator:	Valerie Maxville	Tutor:	Harry
Date of submission:	25/5/23	Which assignment?	(Leave blank if the unit has only one assignment.)

I declare that:

- The above information is complete and accurate.
- The work I am submitting is *entirely my own*, except where clearly indicated otherwise and correctly referenced.
- I have taken (and will continue to take) all reasonable steps to ensure my work is *not accessible* to any other students who may gain unfair advantage from it.
- I have *not previously submitted* this work for any other unit, whether at Curtin University or elsewhere, or for prior attempts at this unit, except where clearly indicated otherwise.

I understand that:

- Plagiarism and collusion are dishonest, and unfair to all other students.
- Detection of plagiarism and collusion may be done manually or by using tools (such as Turnitin).
- If I plagiarise or collude, I risk failing the unit with a grade of ANN ("Result Annulled due to Academic Misconduct"), which will remain permanently on my academic record. I also risk termination from my course and other penalties.
- Even with correct referencing, my submission will only be marked according to what I have done myself, specifically for this assessment. I cannot re-use the work of others, or my own previously submitted work, in order to fulfil the assessment requirements.
- It is my responsibility to ensure that my submission is complete, correct and not corrupted.

Signature: KK Date of signature: 25/5/23

(By submitting this form, you indicate that you agree with all the above text.)

Overview:

The purpose of this program is to demonstrate an understanding of the python language, matplotlib and other programming concepts, such as neighbourhoods, accessing CSV files, list comprehension, objects, and loops. I implemented these various techniques into my code, to "plot" an animation of a spinal tap band performance. I added features such as lights, smoke simulation, backdrops and band member animation, and an extra lasers feature. I have designed them in such a way that allows for easy manipulation in the code once each class instance for each feature has been created.

I have also allowed for interactive manipulation of each feature using sliders and buttons, as well as a CSV template that allows an individual to create scenarios that are then plotted and animated.

User Guide:

If trying to generate a static scenario that can be manipulated using your own code, simply import all relevant libraries (extras, plots, smoke, images, light) using

```
from <library> import *
```

to directly access the classes.

Then set up the stage by creating an instance of the generate_stage class in plots.py. Afterwards, you can set up the backdrop, stage, and band performers using renderBackDrop, renderStage and bandAnimate respectively in images.py.

Then you can create each light object on a horizontal position of your choosing, as well as if it is on the bottom or the top of the stage. Then the next step is important – add each light to a list and pass that list as an argument to the lightSeries class. lightSeries is used to manipulate each light via a class method, otherwise each light is static after it is created. It is suggested to keep each light object in the list ordered, from the light closest to the left to the light closest to the right to allow for easy indexing when calling a lightSeries method. It is the choice of the user if they would like to add any "bottom" light to a different lightSeries instance to clearly distinguish between the bottom and the top lights, though it is suggested that you do so. The classes light and lightSeries are found in lights.py.

Other stage features can be created by setting up an instance of their respective classes.

At this point you can call plt.plot() to see the frame. Then access the methods for each class instance to update the properties of the plotted objects, for example lightSeries.setIntensity() to adjust the intensity of a plotted light.

The pre-written driver code (spinal_tap.py) has three scenarios by default. One is the interactive gui for the animation, and the other two make use of code that reads through a .CSV file and animates a pre-written scenario.

To access the gui, in the terminal run spinal_tap.py and write "interactive" as the first argument.

To run a CSV file with a prewritten scenario, in the terminal run spinal_tap.py and write the path of the csv file in the first argument.

To create your own CSV file, take a moment to understand how to call each method to update each object in the animation. The name of the instances is shown in the template. Then follow the template and create your own.

Traceability Matrix

Feature	Code Reference	Test Reference	Completion Date
1 – Use matplotlib to create 5 subplots used for the stage animation. Each subplot is saved as an attribute for easy access by other classes.	Class generate_stage() In plots.py	Create instance of generate_stage() and observe that all axes are displayed correctly. Test: Passed	18/5
2 – Takes an integer (horizontal position), an instance of generate_stage, and two strings (colour and side) and use matplotlib.pyplot.fill to create a matplotlib.patches.polygon object that simulates a light beam using the given position, colour and intensity.	Class light() In lights.py	Create instance of light() and observe that the light is correctly plotted. Test: Passed	19/5
3 – A class that takes a list of light objects on instantiation and allows for the properties of the lights to be manipulated	Class lightSeries In lights.py	Create instance of lightSeries, will raise an exception if the values aren't correct. Any methods will also return an exception if the values in the list are not light objects. Test: Passed	19/5
3.1 – takes a list of integers and returns an iterable containing each light object. The string 'all' can also be passed to return all objects in the lightSeries.	Method ifAll() In lights.py	Insert a test case (a list of a few integers) and ensure the returned value is the correct light object. Then pass 'all' and ensure the full list of light objects that was passed when the class was initialized is returned. Test: Passed	19/5
3.2 – takes a float number between 0 and 1, and a list of integers, and updates the light objects to change their transparency by changing the alpha value	Method setIntensity() In lights.py	Insert test values and observe the plotted light to ensure changes were done correctly. Test: Passed	19/5
3.3 – Takes a string with the name of a colour, and a list of integers, and updates the colour of the light using matplotlib.patches.polygon.set_color()	Method setColour() In lights.py	Insert test colours and observe the plotted light to ensure changes were done correctly. Test: Passed	19/5
3.4 – takes an integer and a list of integers, and updates the direction of the light, rotating it around the source in degrees. Using matplotlib.tranforms.affine2D() and	Method setDirection() In lights.py	Insert an angle and observe the plotted light to ensure changes were correctly implemented. Test: Passed	19/5

matplotlib.patches.polygon.set_transform			
3.5 – takes an integer and a list of integers, and updates the width of the light beam using matplotlib.patches.polygon.set_xy()	Method setFocus() In lights.py	Insert a width value and observe the plotted light to ensure changes were correctly implemented. Test: Passed	19/5
4 – takes an axis (meant to be the main stage axis set up in the generate_stage instance) and plots the stage image onto the axis using imshow().	Function renderStage() In images.py	Save test images, and observe the plot to ensure the stage was plotted to the correct position. Test: Passed	15/5
5 – takes an axis(same as 4) and plots the backdrop image onto the axis using imshow().	Function renderBackDrop() In images.py	Save test iamge, and observe the plot to ensure the backdrop was plotted to the correct position. Test: Passed	15/5
6 – A class that takes the main stage axis (same as 4) and plots the band performers onto the axis using imshow(). Allows for animation of the band by calling the nextFrame method.	Class bandAnimate() In images.py	Save test pictures and observe the plot to ensure the band was plotted to the correct position. Test: Passed	15/5
6.1 – Takes an integer representing the current frame and updates the band performer frame. Rotates between three images to animate the playing and dancing of the performers	Method nextFrame() In images.py	After creating an instance of bandAnimate, Create a for loop with a certain range of integers, and pass each integer as an argument to nextFrame(), then call plt.pause(1) to see the next frame for one second. Ensure that the frame updates every four frames. Test: Passed	15/5
6.2 – Takes an integer and updates the bandAnimate.speed attribute. Changes how often the band performer frame updates. Essentially change the speed of the animation.	Method changeSpeed() In images.py	Using the test code in 6.1, call changeSpeed before the for loop. Observe the changes to ensure the frequency of the updated frame has changed. Test: Passed	15/5
7 – Takes an instance of stage_generate, two strings (colour and side), and three integers (vertical position, split, spread), and creates line2D objects that simulate lasers. The number of 2D objects that is created is defined by split.	Class laser() In extras.py	Create an instance of the laser class and observe the lines were correctly plotted. Test: Passed	19/5
7.1 – Takes an integer (angle). Rotates each line2D object by an equal amount, defined in self.spread when the class instance is created, to simulate multiple lasers coming from the same source. Each rotation is offset by angle degrees	Method rotate() In extras.py	Can be tested when the class instance is created, as it is called on init(). Can also be tested when calling setDirection in the laserSeries class.	19/5

(as defined in the parameter) to simulate all lasers pivoting around the source.		Test: Passed	
8 – A class that takes a list of laser objects created in 7, allowing for easy manipulation of the properties of each laser object.	Class laserSeries() In extras.py	Will raise a TypeError if the argument is not a list of laser objects. This can be used to test if the class was initialized correctly. Test: Passed	19/5
8.1 – Takes a string (colour) and changes the colour of all lasers.	Method setColour () In extras.py	After calling the method, observe the plot to see if the changes were implemented correctly. Test: Passed	19/5
8.2 – takes an integer (angle) and changes the direction that all lasers are facing.	Method setDirection() In extras.py	After calling the method, observe the plot to see if the changes were correctly applied. Test: Passed	22/5
8.3 – Turns all lasers off or on.	Method Toggle() In extras.py	After calling the method the first time, observe the plot to see if the line plots were removed, then call the method again to ensure the plots are replaced. Test: Passed	22/5
9 – Takes an instance of generate_stage, as well as a string (side) and sets up an array, a custom colour plot, as well as an imshow() object to simulate smoke diffusion.	Class smokeMachine() In smoke.py	After creating the class instance, observe the plot to see that one pixel either on the left or the right has been plotted. Further, an exception will be raised if the side argument is not either 'l' or 'r.' Test: Passed	22/5
9.1 – Takes a boolean (state) and updates the smoke simulation.	Method nextFrame() In smoke.py	Create a for loop with a given range and call nextFrame in the for loop, with the state parameter set to True. Then call plt.pause(0.5) in the for loop and observe that the smoke is updating every 0.5 seconds and that it is diffusing. Test: Passed	22/5
10 – Driver code for the assignment.	Spinal_tap.py	Run the python file with "interactive" as an argument and see if the gui interface for the stage is correctly displayed. Test: Passed Run the python file with a .csv file as an argument and see if the scenario is correctly generated, using choreo1.csv or choreo2.csv as a template. Test: Passed	24/5

11 – Class that inherits FuncAnimation from matplotlib and provides a gui to interactively manipulate object properties and see changes immediately.	Class animate() In interactive.py	Will raise an exception if values are not valid, as the __init__ function calls the __init__ of its super. Test: Passed	24/5
11.1 – called on __init__ to set up gui.	Method setup() In interactive.py	After calling plt.show() when the class instance is created, the gui should appear. Test: Passed	24/5
11.2 – applies any change made when changing slider values.	Method applyChange() In interactive.py	After manipulating any slider on the gui, the respective object should reflect the changes on the other window. Test: Passed	24/5
11.3 – Turns the smoke machine off/on	Method toggleSmoke() In interactive.py	After selecting 'toggle smoke machine' button, the smoke machine should turn either off or on. Test: Passed	24/5
11.3 – Turns the lasers off/on	Method toggleLasers() In interactive.py	After selecting the 'toggle Lasers' button, the lasers should turn either off or on. Test: Passed	24/5
11.4 – Randomizes the light colours	Method randomizerLaserColours() In interactive.py	After clicking the 'randomize light colours' button, each light should have its colour randomized. Test: Passed	24/5
11.5 – Randomizes the laser colours	Method randomizeLightColours() In interactive.py	After clicking the Randomize laser colours' button, all lasers should be changed to a different random colour. Test: Passed	24/5

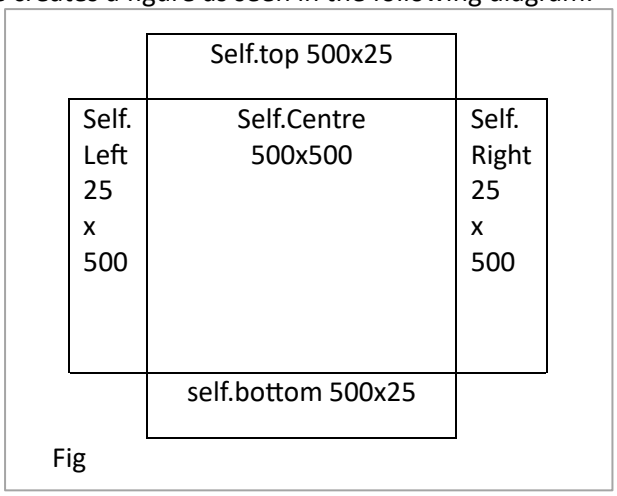
Discussion:

Feature 1: generate_stage()

This class is intended to be instantiated at the start of your code. It is required for other functions in this library to work properly. It starts by creating a figure with a figsize of 10x10. My assignment requires four subplots around the centre subplot, used to indicate a "source," whether that be a light source, smoke source, etc. To create those subplots, I use matplotlib.figure.add_gridspec, which creates a template of sorts which is then used to create subplots afterwards by indexing the gridspec object. Lines 24 through 28 set up those subplots and saves them to an attribute indicating which side each subplot is relative to the centre plot. This is important as these 5 attributes are accessed by

the rest of the library to plot onto these subplots. To make the code shorter, the sharex and sharey attributes allow changes to the axis labelling on one subplot to be reflected in any other subplot. Line 30, 31 changes the colour of each outer subplot to black. Lines 33 through 37 set up the x_lim and y_lim of each axis, and removes the ticks for all axes as they are unnecessary in this case. Line 39 creates a transData object which is important for any transformations (rotation, translation) made to objects in the centre plot.

As a result, the code creates a figure as seen in the following diagram:



Fig

Note that further references to any of these attributes in the rest of the report or in the code is directly referring to these subplots. All other functions take an instance of this class as an argument and are usually referred to using the variable: stage. Mentions of stage.centre, etc, still refer to this diagram.

Feature 2: light()

This Class plots a shape which is used to simulate a light beam in the animation, and can be placed on either the bottom or the top of the stage. Among other parameters, it takes an instance of feature 1, Generate_stage() so that the light beam can be plotted. Line 44 establishes the horizontal position of the light, from 0 to 500. Line 46 sets up a matplotlib.patches.circle object which is to be plotted later. Line 49 through 59 checks if the parameter side was set to either ‘t’ or ‘b,’ determining if the light is coming from the bottom or the top of the stage. It generates the y positions of the light plot (x positions are independent of whether the light is on the top or the bottom), and plots the circle object that was created in line 46 onto stage.bottom. Lines 62 and 63 plot the beam and saves the newly created polygon object to self.beam. The x positions (the first argument in stage.centre.fill) are based on the horizontal position that was passed as an argument to __init__ and saved in line 44. The y positions (second argument) are as defined in line 53, self.yPositions. The variable zorder in line 54/58 is set depending on whether the light is at the bottom or top, as a light that is on the bottom needs to be rendered above the stage that is created in feature 4, renderStage(), which has a zorder of 10, while a light that is on the top needs to stop when it hits the stage. The Zorder argument states the order in which each object is layered on top of each other.

Feature 3: lightSeries()

The __init__ function of this class takes a list or array containing each light object created using the light class from feature 2, and saves it to self.series.

Lines 79 through 84 are used to check if each value in the list is a light object, and will raise an exception if the check failed.

Feature 3.1: Method ifAll()

This method is only called within the class. Every other method in the class takes a list of integers as one of their arguments, representing the indexes which are used to grab the lights in self.series that you want to adjust. All methods pass that list as an argument to this method, which returns a list containing only the lights that you selected. It also takes 'all' as an alternative, which simply returns all lights in self.series.

Feature 3.2: Method setIntensity()

This method iterates through the list returned by self.ifAll() and uses the function set_alpha() to adjust the intensity of each light, by changing the alpha value of the polygon object. It also does this for the light source, which is a patches.circle object, defined as self.source in __init__.

Feature 3.3: Method setColour()

This method iterates through the list returned by self.ifAll() and uses the function set_colour() to adjust the colour of each light. It also does this for the light source.

Feature 3.4: Method setDirection()

This method iterates through the list returned by self.ifAll(). For each light, it determines if the light is on the bottom or the top, and reverses the angle that was passed as the first argument (using list comprehension), seen in line 122. This is to ensure some visual continuity, as now whether on the bottom or the top, a positive angle will always make a light point towards the right and a negative, to the left. Lines 123 and 124 use two transformation data objects, the first being a matplotlib.transforms.Affine2D() object and the second being a transData object from the light object's self.stage.transData attribute (which is really the transData for generate_stage's self.centre subplot). The first defines the pivot point for the light and how much it will rotate in degrees. The second is necessary for the transformation to work correctly.

This translation data is combined and applied using the function set_transform in line 124.

Feature 3.5: Method setFocus()

This method iterates through the list returned by self.ifAll, and replots each light. The two x values that define the end point for the light are increased by an amount defined by the argument foc. This simulates a widening of the focus of the light. Afterwards, self.setDirection is called in line 139 to rotate the angle back to its current position, as setFocus() returns a light to its default position.

Feature 4: Function renderStage()

This function takes an image that has been reformatted specifically for this use and plots it at the bottom of stage.centre. the argument origin is set to 'lower' to ensure the image is plotted at the bottom of the subplot.

Feature 5: Function renderBackDrop()

This function takes the backdrop image, plots it, then translates up it to its correct position in lines 64 and 65, as by default it is too far down.

Feature 6: Class bandAnimate()

The __init__ function of this class only takes an instance of feature 1, generate_stage(). It then takes three saved images, saves the references to those images into a list (self.frames) and uses the transformation data in line 31 to position the imshow data to the centre of the stage. It only plots the first frame. Frames are updated in Feature 6.1

Self.speed defines how many frames will pass until the band image frame is updated, while self.count is used internally so that the number of times the band image has been updated can be remembered.

Feature 6.1: Method nextFrame()

Every time this is called, it modulo divides the current frame by self.speed, and if the result is zero, it updates the frame using the set_data function. It loops infinitely over self.frames by modulo dividing self.count by 3 and then increases self.count by one, meaning that the index will loop from 0 to 2 indefinitely.

Feature 6.2: Method changeSpeed()

When this method is called, it changes self.speed, allowing the user to decide how many frames must pass until the band image is updated.

Feature 7: Class laser()

This class is is similar to light, in that it is only intended for creating objects that plot lines resembling lasers. After variables are initialized, lines 42 and 43 create the x and y positions as well as the width for 9 circles. These circles are plotted in lines 46 through 57, which first checks if the laser is to be plotted on either the right or the left side, then plots a 3x3 square of circles representing the laser source. Matplotlib.Collections.CircleCollections is used to plot multiple circles at once. Subplots have a function called add_collection which is used to apply the CircleCollection object.
As seen in lines 47/52, self.pos is used to store the two x values for each laser beam. The reason they can not be identical is that when the laser is plotted, the length of the beam must slightly exceed the border of the subplot so that when the beam is rotated, it does not appear to stop before it reaches the border of the stage area in stage.centre. The same was done for Feature 3, lightSeries.

Self.split (an integer) defines how many lasers are coming out of the one source. In line 60, an array is created with its height determined by self.split, and it contains the y values of each laser.

Line 61 then plots <self.split> lasers, with the first argument being the x values and the second being the y values.

At this point, the lasers are all plotted on identical positions, so they are overlaying on top of each other. To split the lasers, Feature 7.1, rotate() is used.

Feature 7.1: Method rotate()

This method iterates through an np.linspace array in line 30. The first two arguments are self.spread, which was defined in __init__. These determine how wide apart each laser is from each other. The third argument is self.split, which allows np.linspace to only return an array equal to the amount of lasers there are.

Enumerate is used in the for loop to index self.lasers (self.lasers is a list containing each line object), then applies each rotation in line 32. Each iteration will rotate a different laser by a different angle,

resulting in an even split of lasers coming out of one source. Due to the constraints of the assignment time, this split can only be defined on instantiation. It can not be changed in Feature 8, laserSeries().

As shown in line 31, the third argument (which defines the angle of rotation), includes the rotation from the for loop (y), but also includes the angle argument from the function, which is used to change the angle of all the lights relative to their original position.

Feature 8: Class laserSeries()

Similarly to feature 3, the class laserSeries takes a list containing instances of the laser objects from feature 7, and saves it to self.allLasers.. self.state is set to True by default and allows for the lasers to be toggled off and on by saving their current state.

Due to time constraints, the three methods in the class only support functionality that applies to all lasers in self.allLasers. There is no indexing functionality like that in feature 3, lightSeries. Lines 73 through 79 check if all items in the list are instances of the laser objects, and raises an exception if it is not the case.

Feature 8.1: Method setColour()

This Method iterates through each laser Object in self.allLasers, and sets the colour of the collections object (the one that shows the 3x3 square of lasers in the right or left subplot) to the colour argument of the method, using collection.set_color().

It then runs a for loop nested within the outer for loop in line 85 and 86 to iterate through each individual line object in the light object, by accessing the laser.lasers attribute of the laser object. It then changes the colour for each one using set_color() to the colour that was passed as an argument.

Feature 8.2: Method setDirection()

This method iterates through each laser Object in self.allLasers. line 91 then sets the angle that was passed as an argument to its negative if the laser.side attribute was ‘r,’ indicating that the laser is on the right side. This allows for lasers on both sides to move upwards with a positive angle and downwards with a negative angle.
line 92 then calls the laser.rotate() function from feature 7, passing the first value in ang from line 91 as an argument. That function performs the re-splitting of the lines and also moves them the additional angle equal to the amount passed into the function.

Feature 8.3: Method toggle()

This method changes self.state, a Boolean, to its opposite. It then runs a nested for loop through each laser object in self.allLasers and each individual line object and sets their alpha value to 0 if self.state was False, or 1 if self.state was True. This therefore can be called to turn the lasers off and on.

Feature 9: Class SmokeMachine()

This class sets up a smoke machine. Line 32 sets up an array of zeros, with dimensions as shown.

It is done this way for two reasons:

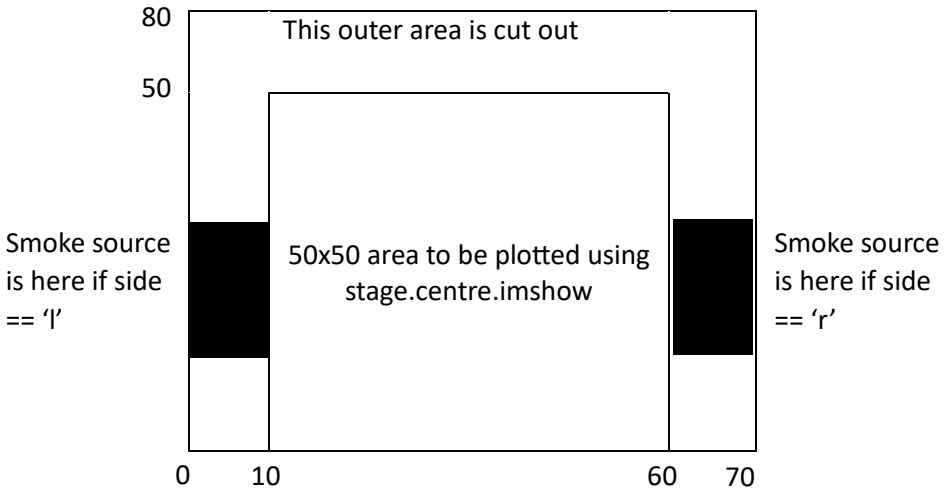
- By making self.size equivalent to 50, then upscaling the image to 500x500 for plotting (using np.kron as seen in line 57 and line 96), operations are calculated faster and frame speeds are therefore less bottlenecked by the smoke diffusion calculations, as compared to calculating values using a 500x500 array.
- All entries on the border of the array (the first and last row, and the first and last column) need to be cleared every frame, to prevent smoke from 'teleporting' from one side to the other. This is due to something that will be discussed shortly. By making the dimensions of the array slightly taller and wider, the array that results after calculation can be sliced back to a 50x50 array and will appear as if the smoke is simply flowing out of the view of the image, instead of abruptly stopping. The edge value removals are done in lines 89 through 93, and the indexing can be seen in line 57 and 96.

Due to the nature of matplotlib.axis.imshow(), which plots an image over the top of all other plots in the axis, and makes other stage features invisible, a custom colourmap must be created. This colourmap contains the colour data of the 'Greys' default colourmap from matplotlib. However, the matplotlib.colors.LinearSegmentedColormap function is used to add alpha values to the colour values. As a result, a higher value in an array results in a darker pixel and will also have an alpha value close to 1, and as the value in the array decreases, the pixel will become lighter and its alpha value will also move closer to zero. This allows the smoke to be transparent and allows the rest of the animation to still be visible. The creation of this colourmap is seen in lines 48 through 54.

Lines 37 through 45 check which side the smoke machine was placed on, then plots a circle indicating the source in either stage.left or stage.right. It also sets up one source pixel in the array, set to the maximum possible value (self.max). This is required for a point of reference to be created in the colourmap, otherwise it will assume 0 is the highest value when imshow is called in line 58, and no image can be displayed.

Feature 9.1: Method nextFrame()

This method updates the self.air array to reflect the diffusion of smoke. If self.state (defined at __init__) is True, some values in the array will be filled with self.max every frame, to simulate a constant smoke source. The location of the smoke source is shown in the following diagram:

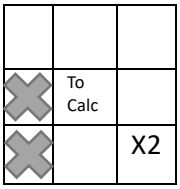


The smoke source is placed in the area that is cut out of the array, to avoid the jarring appearance of a square in the animation. The diffusion moves into the 50x50 section of the array, and so its effects can be seen.

Lines 67 through 83 perform a Van Neumann Neighbourhood calculation of a value in a cell. It does so using vectorization. Lines 67 through 70 shift self.air in every direction; up, down, left, right, up+left, up+right, down+left, down+right, using np.roll, and then saves each shifted array to the list in line 67. Performing the Van Neumann calculation for a given cell can therefore be done all at once using vectorization and simply requires finding the sum of all arrays, as seen in line 82. This bears massively greater speeds than an iterative method which usually involves using nested for loops to calculate the next value for each cell one by one. This major optimization essentially removes the concern of the smoke machine being a bottleneck to frame speeds.

To allow the smoke to tend towards a certain direction, some values in the Van Neimann neighbourhood must be weighted/removed. Consider the following diagram:

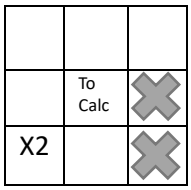
If self.side = 'l':



Direction of flow ->

If self.side = 'r':

Direction of flow <-



As shown in the diagram, depending on whether self.side is 'l' or 'r', two particular values are removed from the calculation, and one is multiplied by two. This allows the diffusion to tend towards the direction where there is a greater number of values being included in the neighbourhood. As a result, the smoke appears to move towards the left or the right and in a slightly upwards direction.

This is also why more than one smoke machine shouldn't be used, as it is beyond the requirements of this assignment to properly simulate smoke diffusion and factors such as wind, temperature, mixing etc. and as a result, two smoke machines would begin to overlap each other as they plot over each other, instead of diffusing into each other.

Feature 10: Driver code

The driver code takes a single argument and sees if it is the string "interactive." If that is the case, most of that block of code is dedicated to setting up objects for stage features. It then calls the animate function from Feature 11, and allows for an interactive gui to demonstrate the functionality of the classes.

If the argument is not 'interactive,' it should instead be a csv file following the template found in choreo1.csv or choreo2.csv. Lines 71 through 90 create the objects and line 99 is the Mainloop that executes the functions in the csv file. It allows two main pieces of functionality: If the line in the csv is an integer, a for loop runs at an amount equivalent to that integer, updating the frames without executing any new functions. If it's not an integer, it should be a function or instance method.

Feature 11: class animate()

Provides a visible GUI to allow for interactive and manual interaction with your instances for various stage features.

This class inherits from matplotlib.animation.FuncAnimation, and calls super().__init__ to allow the child class (animate) to run a Mainloop.

In line 42, __init__ calls the setup() method, used to create the buttons and sliders used for the gui.

Feature 11.1: Method setup()

Lines 52 through 54 creates a new figure (a second window) with a 2x3 grid of subplots, and then removes the axis for all subplots so that the figure only displays widgets.

Lines 57 through 61 allow for a reference to each subplot so they can be further split down into individual spaces, as only one widget can be plotted per subplot.

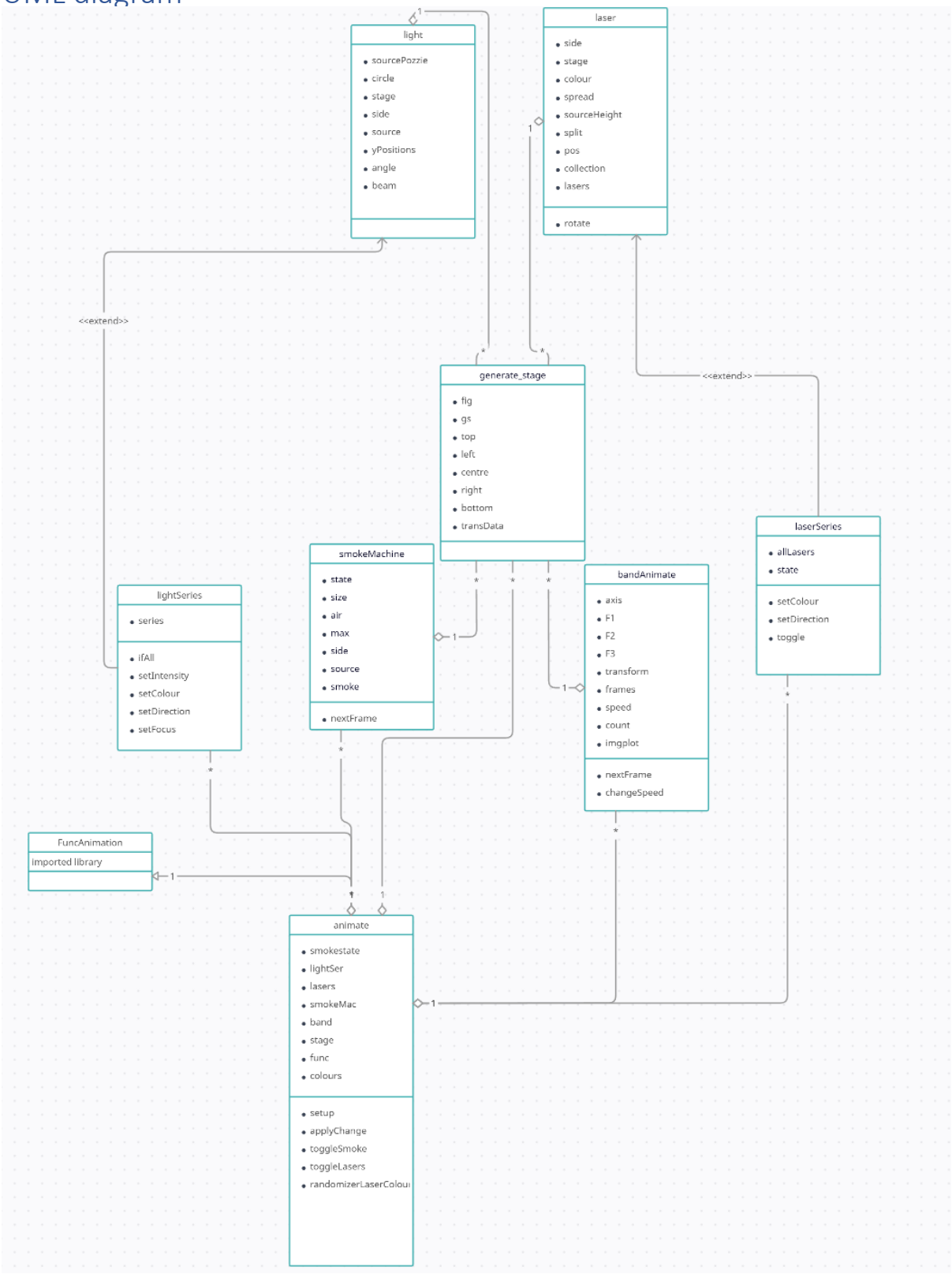
In lines 71 through 112, any.append_axes() function that can be seen is being used to create "sub-subplots" to place a widget in. After each time that function is called, a button or slider is set up using matplotlib.widgets.<Slider/Button>. Then an input event is created, which calls another method in the class that is used for applying the changed values.

In the for loops in lines 71 through 91, they also apply Transformation data to the labels; as labels are usually placed to the left of the slider, the set_transform function moves each label back into the slider so that they do not overlap adjacent sliders. The for loop in line 71 creates three sliders for Direction, Width, and Intensity, and does this for 8 lights. The for loop in line 90 creates events that are run when a slider value is changed.

Features 11.2, 11.3, 11.4, 11.5, 11.6

These Methods are called when their respective button or slider is pressed or has its value changed. They all perform their operations according to what the name of each method is.

UML diagram


light
sourcePozzie
circle
stage
side
source
yPositions
angle
beam
laser
side
stage
colour
spread
sourceHeight
split
pos
collection
lasers
rotate
<<extend>>
generate_stage
flg
gs
top
left
centre
right
bottom
transData
<<extend>>
laserSeries
allLasers
state
setColour
setDirection
toggle
smokeMachine
state
size
air
max
side
source
smoke
nextFrame
bandAnimate
axis
F1
F2
F3
transform
frames
speed
count
imgplot
nextFrame
changeSpeed
lightSeries
series
ifAll
setIntensity
setColour
setDirection
setFocus
FuncAnimation
imported library
animate
smokestate
lightSer
lasers
smokeMac
band
stage
func
colours
setup
applyChange
toggleSmoke
toggleLasers
randomizerLaserColou

Showcases

These showcases are intended to demonstrate the flexibility of my code, and how the methods I have provided allow for a high level of manipulation of the band performance animation and scenes that can be plotted. To run my showcases, you can run the driver code, spinal_tap.py, then the first argument must be either “interactive” or the path to choreo1.csv or choreo2.csv. Each scenario displays the functionality of my code in different ways.

Showcase One:

In the terminal, write:

python3 spinal_tap.py interactive

This brings up two windows: the stage plot and a window full of sliders and buttons. From there, you can essentially “mess around” with inputs and see how it affects the displayed image. You can easily create a frame and save it using the save button on the matplotlib GUI.

As a result, my output is dynamic and can be adjusted interactively by the user. I intend to demonstrate the point that my code is not based on hard-coded values and allows for a great deal of user manipulation, and user-friendliness.

Showcase Two:

In the terminal, write:

python3 spinal_tap.py choreo1.csv

This plays a scenario. It is used to demonstrate the functionality of the CSV reading component of my driver code in spinal_tap.py. It also shows that my code supports the use of a CSV which can be used to create a repeatable scenario that can be ran multiple times without additional/repetitive set up. This showcase in particular demonstrates the smoke Machine on the right side of the stage, as well as the ability to move lights into a position that is visually pleasing and similar to how it may be done in regular live performances.

Showcase Three:

In the terminal, write:

python3 spinal_tap.py choreo2.csv

This scenario is also a use of the CSV component of my driver code. It demonstrates the ability to randomize the positions of the lights and create a lightshow akin to a disco or nightclub, where lights move sporratically and unpredictably. It also shows the ability to use the smoke machine on the left side, however it is harder to see due to the greater use of ligher colours in this scenario.

Conclusion:

This assignment demonstrates my understanding of key programming concepts as well as a deep understanding of how to use matplotlib for graphical representations of information.

Future work:

I could possibly implement a more fleshed out laser system, more knobs and dials for the interactive GUI scenario, and use libraries or other methods to create a much more versatile and realistic smoke machine.

References:

None used