



Curtin University

Fundamentals of Programming

COMP1005

Assignment Option 1 of 1

Project Report

Submission: 1st November 2020, Sunday

PaSciRo

By: Ibrahim Malik

20207593

Table of Contents

1.0 Abstract	3
2.0 Background	4
3.0 Methodology	5
4.0 Results	17
5.0 Conclusion and Future Work.....	21
6.0 References	22

1.0 Abstract

An extensive program was developed to monitor the life on a planet, *“PaSciRo”*, containing discrete functions present within the program. These functions will help to audit the contrasting scenarios, enable outputs to be presented and discussions to be made.

This report is written to expansively illustrate the capability of the program in assessing and demonstrating the planet in all required circumstances using assorted parameters and boundaries. It will be going in depth of the requirements and purposes of the simulations being conducted as well as the methodology by which they are carried out. This includes the various functions and commands that make up the program that is being run. Results will be displayed, compared and discussed about, giving precise synopses. The possible need for supplementary analysis will also be discussed which pose as achievable improvements to the current version of the program.

It is for the reason of thoroughness that the amount of lifeforms inhabiting the planet, the succession of each step, the potential to save the outputs and in what form they are saved, and finally the given scenario which arises on the planet, are all part of the assessed parameters that will be investigated.

Altering the parameters will most definitely give altering outputs making the need to save each simulation critical. Hence, harvested outputs have the option to be saved within newly formed directories. This will be taking place within the program and is also discussed about moving further into the report.

It is essential to recognize the capabilities of the current program with respect to how and where it can be run to ensure full functionality. Priority is given to the *“Spyder”* application which provided the basis for the formation and implementation of the project. The current version can very well be run with the *“Jupyter Notebook”* which gives, to a certain degree, clearer outputs and a more user-friendly interface than the former. Acknowledge that *“Anaconda Prompt”*, *“Anaconda PowerShell Prompt”* and other applications such as *“PyCharm”* and *“IDLE”* etc. are not supported platforms to yield results on.

2.0 Background

The planet PaSciRo consists of 3 different types of lifeforms, namely:

- a. The Red Lifeforms
- b. The Yellow Lifeforms
- c. The Blue Lifeforms

To investigate the various scenarios that may occur on the planet, the lifeforms were subjected to specific scenarios. As mentioned before, these scenarios were brought about from functions within the program and played a key role in accessing the overall inhabitation of the planet.

A proper investigation of the planet involved all the lifeforms present on it to be subjected to conditions which are likely to arise. This may be of the types which include:

1. The planet in its original form without any deletions or additions to it.
2. The planet undergoing a scenario involved with the lifeforms attacking each other.
3. The planet bearing a change resulting in the lifeforms reproducing.
4. The planet being invaded by foreign aliens, a possible threat to the lifeforms.

The main program was constructed keeping the above-mentioned plots in mind.

It was given utmost importance that the user controlling the program be in charge every step of the way. This was ensured by implementing a parameter sweep throughout the simulation process. The sweep concerned with entering a value for the population of lifeforms on the planet for every simulation as well as the time step which is initially the succession of each step showing the lifeforms move. This specific simulation can then be saved as various formats directly into specific folders. This will be comprehensively discussed in the following sections.

The parameter sweep would allow the user to swing through different simulations as well as making sure they stay within the program which does not shut down or exit until or unless the respective command by the user has been given. The simulations can then be collected and processed at will, enabling efficient comparisons to be made and adequate conclusions to be drawn out.

It is to be noted that there were extra added features, not required according to do the assignment specification, introduced into the program assignment. They have been listed as follows:

1. 3D Space Plotting
2. Displaying Statistical Outputs
3. Extensive Parameter Sweep
4. External Window plots for each time step (High Quality Visualization)
5. Interactive Graph Plots
6. New Directories Formation
7. Recurring Exception Handling
8. Saving in PDF File Format

3.0 Methodology

To initiate the program, 2 python (.py) files are required:

1. PaSciRo.py
2. PaSciRo_Classes.py

Of the two preceding files mentioned, “*PaSciRo.py*” is the main file which will be run by the user as this contains the necessary input requirements for the parameter sweep to inaugurate, and is linked with the latter file which contains the functions and classes being used by the main program. Please refer to the [User Documentation](#) for the elaborate discussion on each of these files.

As indicated previously, the application used to formulate the program was “*Spyder (Python 3.8)*” hence all subsequent explanations of the code and simulation comparisons will be made using this very application.

When the program is run from the main file, a series of introductory messages follow after which a function called “*main()*” is run. As made obvious from its name, the function contains the main sequence of codes which allows the parameter sweep to begin and continue throughout the program.

The program imports all classes and functions from the “*PaSciRo_Classes.py*” file to be used within it, as well as the “*time*” module which aids in bringing an aesthetic appeal to the running simulations.

```
import time
from PaSciRo_Classes import *
```

Figure 1

In the beginning of this program, two values namely “*XMAX*” and “*YMAX*” have been defined using integer values. These values indicate the boundaries that define the space the lifeforms on the planet will be occupying, not being allowed to be plotted outside of this. These take on the values of 200 and 100, respectively.

```
# Defining Limits for x-axis and y-axis
XMAX = 200
YMAX = 100
```

Figure 2

Moving on into the function the first sweep is initiated by asking the user to input the population size or “*POP*” value of the lifeforms. The program is built in a way to only accept a certain input based on a defined criteria: an integer value between 5 and 1000 (inclusive). In the case the conditions are not met, exception handling has been set up to throw the respective errors: either “The value is not valid” for a non-integer value and “The value is too low” or “The value is too high” if the limits are crossed. These are expected to continue displaying until an appropriate value is added (see Figure 3).

```
# Input for POP value and ensuring it is an integer between 4 and 1000
while True:
    POP = input("\nEnter the total population of the lifeforms:\n")
    try:
        val = int(POP)
        POP = int(POP)
        while POP < 5 or POP > 1000:
            if POP < 5:
                print("Population level too low. Please re-enter.")
            if POP > 1000:
                print("Population level too high. Please re-enter.")
            POP2 = int(input())
            POP = POP2
        break;
    except ValueError:
        print("'", POP, "'", "is not a valid population size, please try again!")
```

Figure 3

Depending on the size of the entered population, the boundaries of the planet are expected to change. If the value is larger than 200, the values increase by 100 and if the POP value is greater than 500, the boundaries increased by 200 to accommodate the elevated number of lifeforms present on the planet. This was made successful using the “*if*” function.

```
# Expanding ranges for x-axis and y-axis depending on POP size
if POP > 500:
    XMAX = XMAX + 200
    YMAX = YMAX + 200
elif POP > 200:
    XMAX = XMAX + 100
    YMAX = YMAX + 100
```

Figure 4

After the population input is taken from the user, the next input is required which will be for the time step parameter in a similar way to the former (i.e. by using the same method of exception handling), except only an integer between the range of 3 and 100 (inclusive) to ensure a simulation worth investigating. After this, the time interval is defined between each step which shows the commencement and preceding of each time step.

```
# Time interval between each Time STEP
STEPS_TIME = 0.75
```

Figure 5

After the appropriate response has been recorded by the program, the next input will be required which will deal with the user either being able to or not able to save the upcoming simulation. They will have the choice to either input “*y*” as yes and “*n*” as no, otherwise giving an error asking them to re-enter an appropriate entry. This was made to happen using the “*while*” loops. The program proceeds in without saving the simulation in the latter case whereas in the former, it proceeds on to display a menu to choose how to save the simulation. The options include saving as a PDF file, JPG image, CSV file or all three, each specified with a single or set of different alphabets. The user has the option to not save the program as well in which case the default alphabet selection will be “*D*”, made possible using the “*if*” and “*elif*” function. An input of an alien alphabet not mentioned will proceed to show an error: “*Invalid Entry. Try again.*” formulated with the help of a “*while*” loop. This is shown in Figure 6.

```
# Input for saving each Time STEP and ensuring input value is "Y" or "N"
figs = input("Do you want to save each Time-Step [y/n]: ")
figs = figs.upper()
while figs not in ("Y", "N"):
    figs = input("Invalid Entry. Try Again: ")
    figs = figs.upper()

# Input for method of saving and ensuring input is "A" or "B" or "C" or "D"
if figs == "Y":
    time.sleep(1)
    print("\nHow would you like to save your file?")
    print("A - PDF File")
    print("B - Images")
    print("C - CSV File")
    print("ABC - PDF File, Images and CSV File")
    print("D - None")

    save = input("Choice: ")
    save = save.upper()
    while save not in ("A", "B", "C", "D", "ABC"):
        save = input("Invalid Entry. Try Again: ")
        save = save.upper()
else:
    save = "D"
```

Figure 6

Proceeding on to the next display, which will be the main menu. This menu comprises of the different scenarios which have the potential to be simulated and arise on the planet. There are four main mechanisms of doing this, which are explained beforehand in [Section 2.0](#). The options include “S”, “W”, “R”, “I” and “E”. The options allow the planet to reveal itself in its peaceful form, battle mode, reproductive season, and invaded form respectively while the last option gives the user a chance to opt-out of the display and exit the program. This is shown using simple “*print*” statements already used before. Note on how the same error appears if an input not previously mentioned is added, again using the “*while*” function.

```
# Menu to select planet operation
print("\nPaSciRo")
time.sleep(1)
print("Choose from the following: ")
print("S - Simulate planet")
print("W - Simulate war between lifeforms")
print("R - Simulate reproduction between lifeforms")
print("I - Simulate presence of intruders")
print("E - Exit")

# Input for operation and ensuring input is "S" or "W" or "R" or "I" or "E"
option=input("Choice: ")
option=option.upper()
while option not in ("S", "W", "R", "I", "E"):
    option = input("Invalid Option. Please select an option from the menu: ")
    option = option.upper()
```

Figure 7

Once the input is recorded it will now be the time to begin generating the output planet in the form of a graph filled with the population of lifeforms moving randomly. Based on the selected scenario preference, a specific function will run which is inevitably imported from the python classes file. The functions are run, and the code is programmed to select which of these is stimulated based on the choice given, that is by using an extensive list of “*if*’s”. The code for this is given below, in Figure 8.

```

# Selecting imported function based on planet operation
if option == 'S':
    simulate(POP, XMAX, YMAX, STEPS, STEPS_TIME, figs, save)
elif option == 'W':
    if __name__ == "__main__":
        war(POP, XMAX, YMAX, STEPS, STEPS_TIME, figs, save)
elif option == 'R':
    if __name__ == "__main__":
        reproduce(POP, XMAX, YMAX, STEPS, STEPS_TIME, figs, save)
elif option == 'I':
    if __name__ == "__main__":
        intruderz(POP, XMAX, YMAX, STEPS, STEPS_TIME, figs, save)
elif option == "E":
    print("Exiting...")

```

Figure 8

After each of the time steps have concluded and the simulation has come to a halt, also provided the option selected was not to exit the program (i.e. not “E”) the program is coded in a way to ask the user if they would like to perform another simulation allowing only the following inputs: “y” or “n”. If the user does want to perform another, the entire program will repeat itself. This is made possible by using the “if choice == ‘Y’: ” code. On the other hand, if the user has obtained all the required results, they may exit the program by giving the erstwhile option. If in case the option given was “E”, the program will exit and turn off.

It is worth noting that the above described code is only for the “main()” function. The program starts after defining this function, where it is called for.

This was the in-depth analysis of the main python file which leaves “PaSciRo_Classes.py” up for an examination, being conducted below.

This python file contained a far greater amount of package imports than the previously discussed file due to the variety of requirements in the specified functions.

The contents of this file include 2 classes and 5 functions in total. As mentioned several times before, these are imported into the main file where they are then used. Each class and function are described moving on into the report.

Each lifeform created is made as an “object” from the “lifeform()” class. This contains one “instructor” and four “methods”. The instructor is at the surface of the class and is responsible for setting up attributes to each lifeform. This would imply that the object (i.e. lifeform) when created will possess specific characteristics that need to be determined as soon as the class is called. The attributes in this case are: steps, XMAX, YMAX, one (x co-ordinate), two (y co-ordinate), three (z co-ordinate), four (color code) and color:

```

# Class to form each lifeform
class lifeform():
    # Defining attributes of each lifeform
    def __init__(self, steps, XMAX, YMAX, one, two, three, four, color):
        self.steps = steps
        self.XMAX = XMAX
        self.YMAX = YMAX
        self.one = one
        self.two = two
        self.three = three
        self.four = four

```

Figure 9

After this is defined, the first 2 methods are presented which are “col()” and “col2()”. These have almost the same function where the former assigns, by several “if” statements, the specific color of the object based on the color code. This would mean a color code 0 would mean the color red whereas 1 represents yellow and 2 represents blue. At the same time, based on the color, the amount of steps is defined. This in turn makes each of the different lifeforms move at different paces resulting in the red lifeforms moving a distance of 4 steps, the yellow lifeforms moving 3 steps and the blue lifeforms moving with 2 steps. The latter method also performs the same function but uses a different index value (“l.four” instead of “l.three”) as this will be used in 3D plotting where “l.three” is already occupied with the values of the z co-ordinates of the object hence making the need for a different method imminent.

```
def col(self):
    if self.three == 0:
        self.steps = 4
        self.color = "tomato"
    elif self.three == 1:
        self.steps = 3
        self.color = "yellow"
    elif self.three == 2:
        self.steps = 2
        self.color = "royalblue"
# Method to color code and move code lifeform for 3D Simulation
def col2(self):
    if self.four == 0:
        self.steps = 5
        self.color = "tomato"
    elif self.four == 1:
        self.steps = 4
        self.color = "yellow"
    elif self.four == 2:
        self.steps = 3
        self.color = "royalblue"
```

Figure 10

The same case can be applied to the next two methods. “move(lifeforms)” is a method that requires an input “lifeforms” which is the array formed from inside an external function, discussed later. This method will be recording the x, y, z (if any), and color code of the object lifeform into an array, “e”. This will then proceed on to change the x and y values, in the case of the former method based on the step value of each lifeform, so each object moves a particular step. For example, the red life forms will move 4 steps etc. and by the end of this the lifeforms array will be updated to match the new values. In this way the life forms are not confined to start from the same place every time but from the previous updated value. The next method “move2()” will perform the same function but will be applied inside the 3D simulation function (see Figure 11).

```

def move(self, lifeforms):
    a = self.one
    b = self.two
    c = self.three
    d = self.four

    e = [a, b, c, d]

    self.steps = int(self.steps)

    self.one = self.one + random.randint(-self.steps, self.steps)
    while self.one > self.XMAX-1 or self.one < 1:
        self.one = self.one + random.randint(-self.steps, self.steps)

    self.two = self.two + random.randint(-self.steps, self.steps)
    while self.two > self.YMAX-1 or self.two < 1:
        self.two = self.two + random.randint(-self.steps, self.steps)

    f = np.where((lifeforms == e).all(1))[0]
    lifeforms[f,0] = self.one
    lifeforms[f,1] = self.two
    lifeforms[f,2] = self.three
# Method to move lifeform for 3D Simulation
def move2(self):
    self.three = self.three + random.randint(-self.steps, self.steps)
    while self.three not in range(self.XMAX):
        self.three = self.three + random.randint(-self.steps, self.steps)

```

Figure 11

The next and final class is the “intruder()” class which creates not lifeforms but intruders invading the planet as objects. This involves a similar approach as did the previous class with attributes set up for each individual object. This included values such as: XMAX, YMAX, radius, color and px (i.e. the graph figure).

```

# Class to form each intruder
class intruder():
    # Defining attributes of each object
    def __init__(self, XMAX, YMAX, radius, color, px):
        self.XMAX = XMAX
        self.YMAX = YMAX
        self.radius = radius
        self.color = color
        self.px = px

```

Figure 12

There is a total of 4 methods in this class. The first being the “positions(xs,ys)” which takes the respective x co-ordinate and y co-ordinate and then changing it with a step value of 10 which is the amount of steps each intruder moves with and is constant irrespective of the color and radius of the intruder. At the end of the method, each objects’ x co-ordinate and y co-ordinate will be updated allowing for movement of each object (i.e. not starting from the first assigned position but the latest).

```

def position(self, xs, ys):
    xs = xs + random.randint(-10, 10)
    ys = ys + random.randint(-10, 10)
    while xs > self.XMAX-self.radius or xs < self.radius:
        xs = xs + random.randint(-10, 10)
    while ys > self.YMAX-self.radius or ys < self.radius:
        ys = ys + random.randint(-10, 10)

    self.x = xs
    self.y = ys

```

Figure 13

The next two methods can be explained in conjunction with each other. The “*character()*” method plays a role in plotting the intruder object as a circle using the module, “*matplotlib.pyplot's*” circle function. Each intruder is plotted at their x and y co-ordinate with their attributed radius. This is then added into the figure “*px*” using the last code.

The last method in this class is “*distance(x2,y2)*” which will be taking in the x co-ordinate and y co-ordinate of the lifeforms in the planet and then calculating the distance between the specific lifeform and the respective intruder under consideration. This will prove to be especially useful in the “*intruderz()*” function.

```
# Creating and plotting the intruder
def character(self):
    circle1 = plt.Circle((self.x, self.y), self.radius, color = self.color)

    self.px.set_aspect(1)
    self.px.add_artist(circle1)
# Formula to calculate distance between intruder and given x and y value
def distance(self, x2, y2):
    x1 = int(self.x)
    y1 = int(self.y)

    dx = x1 - x2
    dy = y1 - y2

    return math.sqrt(math.pow(dx, 2) + math.pow(dy, 2))
```

Figure 14

This brings the discussion of the classes to an end, leaving the 5 functions to be conferred about which is done going throughout the rest of the report.

The functions involved in this particular file are:

- 1) simulate()
- 2) war()
- 3) reproduce()
- 4) intruderz()
- 5) three()

The first three of which resemble closely with each other while the latter two differ slightly with respect to the codes present in each of them.

Each of the functions starts by forming a “*lifeforms*” array which has the same number of rows as the population entered by the user and 4 columns. The first column is filled with random values within the range of XMAX, while the second column is filled with the random values within the range of YMAX. These ranges ensure that the created lifeforms stay within the limited space of the planet and do not cross the boundaries. The third column is filled with random values between 0 and 2, which will be representing the color codes of the lifeforms whereas the 4th will remain empty throughout the simulation. This will be done for all the rows in the life form array. How this is done can be seen in the Figure below.

```
# Forming the main arrays
lifeforms = np.zeros((POP, 4), dtype=int)
xvals = np.zeros((STEPS, POP), dtype=int)
yvals = np.zeros((STEPS, POP), dtype=int)
# Filling up the lifeforms array
for i in range(POP):
    randX = random.randint(0, XMAX)
    randY = random.randint(0, YMAX)
    randTYPE = random.randint(0, 2)
    lifeforms[i, 0] = randX
    lifeforms[i, 1] = randY
    lifeforms[i, 2] = randTYPE
```

Figure 15

If the user had opted for the simulation to be saved (i.e. if the input was “y”), the next set of code will check if there is already a directory made called “*Saved Simulations*”. If not, it will create this and change into it. Another directory specific to the progressing simulation is created and named after the population size and time step value. For example, if a simulation with 100 lifeforms is run for 20-time steps, the directory name would be: “*S_POP-100_STEPS-20*” where “*S*” refers to the peaceful type of scenario being simulated. Similarly, a war function would be saved as “*W_POP...*”, reproduce will be saved as “*R_POP...*” while intruders would be saved as “*I_POP...*”. If a PDF file is to be saved then a pdf file is opened, if images are to be saved, a new directory called “*Images*” is created but not switched into just yet and in case of saving a CSV file, a respective file is opened with the “*csv.writer(file, delimiter = ',')*” ensuring no unnecessary blank lines are present in the saved file. In case if all the formats are needed to be saved, each one will occur, respectively.

```
# Code carried out if user wants it to
if figs == "Y" and save != "D":
    dir = "Saved Simulations"
    if dir not in os.listdir():
        os.mkdir(dir)
    os.chdir(dir)
    name = "S_" + "POP-" + p + "_" + "STEPS-" + s
    dir = str(name)
    if dir not in os.listdir():
        os.mkdir(dir)
    os.chdir(dir)
    if save == "A" or save == "ABC":
        pdf = matplotlib.backends.backend_pdf.PdfPages(name + ".pdf")
    if save == "B" or save == "ABC":
        dir = "Images"
        if dir not in os.listdir():
            os.mkdir(dir)
    if save == "C" or save == "ABC":
        file = open(name + ".csv", "w", newline = '')
        writer = csv.writer(file, delimiter = ",")
```

Figure 16

After each relative saving method has been chosen, a figure is opened as an external window with the help of the IPython module. This window will be maximized for the user to visualize all aspects of the planet (see Figure 17).

```
# Creating Space for the planet
fig = plt.figure("Simulation")
get_ipython().run_line_magic('matplotlib', 'qt')
time.sleep(1)
print("\nGenerating external window...")
time.sleep(3)
figManager = plt.get_current_fig_manager()
figManager.window.showMaximized()
```

Figure 17

Before the plotting process begins each function is programmed to first start replacing the co-ordinates of the lifeforms with updated ones. This is done for each time step using the “for i in range(STEPS):” code by applying the “move(lifeforms)” method from the “lifeforms()” class to each object lifeform. Before this occurs, the time step value is recorded in the CSV File.

```
for i in range(STEPS):
    if save == "C" or save == "ABC":
        writer.writerow(["Time Step " + str(i + 1)])
    xvalues = []
    yvalues = []
    colours = []
    for l in lifeforms:
        l = lifeform(0, XMAX, YMAX, l[0], l[1], l[2], l[3], "color")
        l.col()
        l.move(lifeforms)
```

Figure 18

Note that there is a nested “if” code here which directly uses the “for” code to check whether the updated x and y co-ordinates of the lifeform under consideration are not overlapping with the co-ordinates of any other lifeform, in which case the “move(lifeforms)” method is re-run not allowing the lifeform to overlap with another. After the coordinates of the lifeform have been finalized, the x co-ordinates are added into the “xvalues” list while the “yvalues” contain the y co-ordinates and the “colours” adds the specific color of the lifeform.

The above described code is for the “simulate()” function and hence is not as complex as the nested if for the “war()”, “reproduce()” and “intruderz()” functions.

For the former, the collision detection will be enhanced to detect the specific lifeforms being collided. A value from the list “xvalues” is taken and compared to the x co-ordinate, which if equal continues to the next condition, comparing the value in the list “yvalues” with the same index as the value obtained previously. This is compared to the y co-ordinate of the lifeform. If they happen to be equal or within ± 3 steps, it is considered as a collision and now the colors of both the lifeforms are compared. The red lifeforms beat the yellow lifeforms which in turn beat the blue lifeforms whereas they are responsible for defeating the red lifeform. The code “d = np.where((lifeforms == [l.one, l.two, l.three, l.four]).all(1))[0]” finds the row in the lifeform array where the lifeform is located and “lifeforms = np.delete(lifeforms, d, axis=0)” will be deleting that lifeform permanently from the array. Please see Figure 19 for the detailed code.

```

for x in xvalues:
    ind = xvalues.index(x)
    if l.one == x or l.one == x + 3 or l.one == x - 3:
        y = yvalues[ind]
        if l.two == y or l.two == y + 3 or l.two == y - 3:
            c = colours[ind]
            if l.three == 0 and c == "royalblue" or l.three == 1 and c == "tomato" or l.three == 2 and c == "yellow":
                d = np.where((lifeforms == [l.one, l.two, l.three, l.four]).all(1))[0]
                lifeforms = np.delete(lifeforms, d, axis=0)
            if l.three == 0:
                DR = DR + 1
                TDR = TDR + 1
            elif l.three == 1:
                DY = DY + 1
                TDY = TDY + 1
            elif l.three == 2:
                DB = DB + 1
                TDB = TDB + 1
            elif l.three == 2 and c == "tomato" or l.three == 0 and c == "yellow" or l.three == 1 and c == "royalblue":
                if c == "tomato":
                    g = 0
                    DR = DR + 1
                    TDR = TDR + 1
                if c == "yellow":
                    g = 1
                    DY = DY + 1
                    TDY = TDY + 1
                if c == "royalblue":
                    g = 2
                    DB = DB + 1
                    TDB = TDB + 1
                d = np.where((lifeforms == [x, y, g, 0]).all(1))[0]
                lifeforms = np.delete(lifeforms, d, axis=0)
            elif l.color == c:
                l.move(lifeforms)

```

Figure 19

It can be seen that in case the color values do not match the war criteria, that is if both the lifeforms are same, a simple re-bounce takes place.

For the “*reproduce()*” function the nested “*if*” starts in the same way by first taking a value from the list “*xvalues*”, gets its index and if the x co-ordinate matches with the value, moves on to compare the y co-ordinate of the lifeform with the value of same index in the “*yvalues*” list and if they prove to be similar, the colors are compared. In case the lifeforms are of the same type and lie within ± 1 step of each other, reproduction takes place by adding a lifeform into the main lifeforms array making sure this new lifeform is the same type of that which have collided. If the same lifeforms do not collide, a re-bounce occurs.

```

for x in xvalues:
    ind = xvalues.index(x)
    if l.one == x or l.one == x + 1 or l.one == x - 1:
        y = yvalues[ind]
        if l.two == y or l.two == y + 1 or l.two == y - 1:
            c = colours[ind]
            if l.color == c:
                n1 = random.randint(-5, 5)
                n2 = random.randint(-5, 5)
                lifeforms = np.insert(lifeforms, POP, [l.one+n1, l.two+n2, l.three, 0], axis=0)
                if l.color == "tomato":
                    TNR = TNR + 1
                    NR = NR + 1
                elif l.color == "yellow":
                    TNY = TNY + 1
                    NY = NY + 1
                elif l.color == "royalblue":
                    TNB = TNB + 1
                    NB = NB + 1
            else:
                l.move(lifeforms)

```

Figure 20

In the *“intruderz()”* function, the distance between each lifeform is calculated with each of the 8 intruders and this is assigned a specific variable. This is done after they are plotted into the figure of the external window which is essentially the planet itself. The distance is calculated using the *“distance()”* method in the *“intruder()”* class. If this distance happens to be less than or equal to the radius of the specific intruder the lifeform is located in the lifeforms array and deleted in a similar manner to in the *“war()”* function as explained before.

```
if (d1 <= k1.radius or d2 <= k2.radius or d3 <= k3.radius or d4 <= k4.radius
    or d5 <= k5.radius or d6 <= k6.radius or d7 <= k7.radius or d8 <= k8.radius):
    d = np.where((lifeforms == [l[0], l[1], l[2], l[3]]).all(1))[0]
    lifeforms = np.delete(lifeforms, d, axis = 0)
    if l[2] == 0:
        DR = DR + 1
        TDR = TDR + 1
    elif l[2] == 1:
        DY = DY + 1
        TDY = TDY + 1
    elif l[2] == 2:
        DB = DB + 1
        TDB = TDB + 1
else:
    l = lifeform(0, XMAX, YMAX, l[0], l[1], l[2], l[3], "color")
    l.col()
    l.move(lifeforms)
    xvalues.append(l.one)
    yvalues.append(l.two)
    colours.append(l.color)
    if save == "C" or save == "ABC":
        writer.writerow([l.one, l.two, l.three])
```

Figure 21

Since the *“three()”* function is entirely dependent on the *“simulate()”* function, a method to extract the same x and y co-ordinates used in the latter function, into the former needed to be devised (see Figure 22a). This was done by creating two extra arrays along with the main lifeforms array with dimensions of the number of steps as the number of rows and the population value as the number of columns. When the coordinates of all the lifeforms were confirmed, and the *“xvalues”* and *“yvalues”* arrays were completely filled, they were fitted into the respective array rows. This was done for all the time steps filling the *“xvals”* and *“yvals”* arrays. These were then transported to the *“three()”* function where each respective row of these arrays was used as the *“xvalues”* and *“yvalues”* (see Figure 22b).

```
xvals[i:] = xvalues
yvals[i:] = yvalues
```

Figure 22a

```
xvalues = xvals[i]
yvalues = yvals[i]
```

Figure 22b

After all the arrays for each of the functions were prepared they proved to be a hard ground for the basis of plotting each lifeform which is done at the end of each function. The co-ordinates in the *“xvalues”* arrays were used as the x co-ordinates while the *“yvalues”* arrays were used as the y co-ordinates and in the case of the 3D function, the *“zvalues”* array was used as the basis for the z-axis.

```
plt.scatter(xvalues, yvalues, c = colours)
```

Figure 23

By the end of each function if the user wanted to save the outputs in any format the PDF file closes, the CSV file is filled and it is closed as well, while the JPG images are stored inside the *"Images"* directory which is then come out of and by the end of it, the user would be back in the original directory with the outputs saved in the *"Saved Simulations"* folder (see Figure 24).

```
if figs == "Y" and save != "D":
    if save == "A" or save == "ABC":
        pdf.close()
    if save == "C" or save == "ABC":
        file.close()
    os.chdir('..')
    os.chdir('..')
    time.sleep(1)
    print("\n\nSaved!")
```

Figure 24

4.0 Results

To visually see the program being run with the respective inputs given and outputs obtained as those explained in Section 3.0, a sample run of the program is performed using the parameters below.

```
Simulation 1
Population Size = 200
Number of Time Steps = 10
Save Time Steps = Yes
Format to save = PDF File, JPG Images, CSV File
Simulation Type = War
Another Simulation = No
```

The simulation details inform us that this simulation will be a mimicking a war on the planet, with the population of the lifeforms equal to 200 with 20 advancements of the scenario shown. The outputs will be saved as a PDF File, JPG Images and a CSV File.

The inputs entered into the program are as follows and are highlighted:

```
Welcome to Planet PaSciRo!
Preparing Planet for simulation...
Initiation --- Successful

Enter the total population of the lifeforms:
200

Enter the amount of time steps the population would advance:
10

Do you want to save each Time-Step [y/n]: y

How would you like to save your file?
A - PDF File
B - Images
C - CSV File
ABC - PDF File, Images and CSV File
D - None

Choice: abc

PaSciRo
Choose from the following:
S - Simulate planet
W - Simulate war between lifeforms
R - Simulate reproduction between lifeforms
I - Simulate presence of intruders
E - Exit

Choice: w
```

```
Generating external window...
### TIMESTEP 1 ###
Red Deaths: 1
Yellow Deaths: 0
Blue Deaths: 0
```

```
### TIMESTEP 2 ###
Red Deaths: 3
Yellow Deaths: 0
Blue Deaths: 2
```

```
### TIMESTEP 9 ###
Red Deaths: 1
Yellow Deaths: 0
Blue Deaths: 0
```

```
### TIMESTEP 10 ###
Red Deaths: 0
Yellow Deaths: 0
Blue Deaths: 0
```

```
-----
Total Red Deaths: 11
Total Yellow Deaths: 4
Total Blue Deaths: 9
```

```
Total number of deaths: 24
New Population: 176
```

```
Saved!
```

```
Would you like to do another simulation [y/n]: n
```

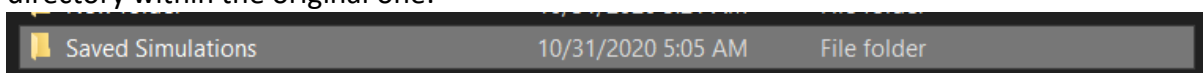
```
Leaving Planet...
```

```
Goodbye!
```

This marks the end of the simulation. It can be seen that there is a sense of respective sequence that is followed throughout the program in showing and giving the instructions on how to proceed with it, making it relatively user-friendly.


When the command for the war to take place was prescribed, the program began to display a series of outputs for each time step as explained in the user documentation, in the form of an external window. The outputs in the command window gave statistical numbers on the population deaths that occurred during a particular step and at the end, the final numbers showed 24 total deaths occurring throughout the simulation, making the population decrease from 200 to 176.

It is also noticed that due to the “y” save command given, there is now an additional directory within the original one:






COMP1005 – Fundamentals of Programming

Within this particular directory it can be seen that another new one has been made by the name of “*W_POP-200_STEPS-10*”, which is the name assigned with our particular parameters that were entered.

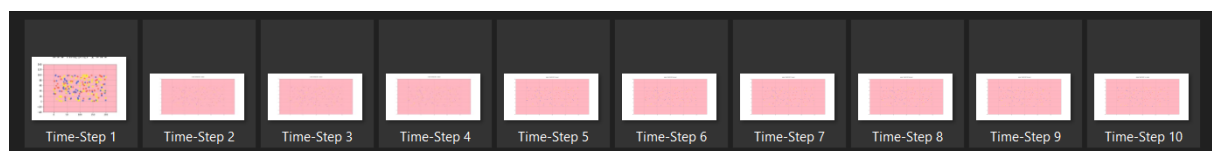
	W_POP-200_STEPS-10	10/31/2020 5:05 AM	File folder
---	--------------------	--------------------	-------------

Entering “*abc*” as the input allowed the simulation to be saved as all the three target formats as shown by the figure below. These are present within the above-mentioned directory.

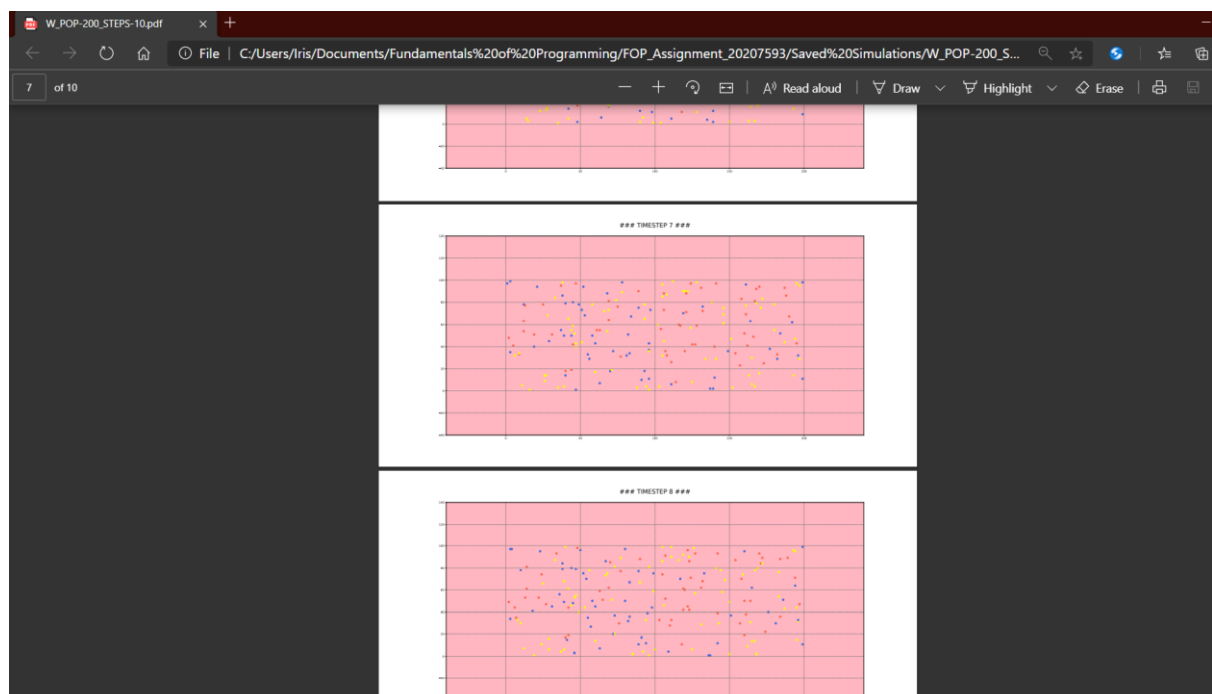
	Images	10/31/2020 5:06 AM	File folder	
	W_POP-200_STEPS-10	10/31/2020 5:06 AM	Microsoft Excel Co...	19 KB
	W_POP-200_STEPS-10	10/31/2020 5:06 AM	Microsoft Edge PD...	40 KB

The images folder contained the simulation in the form of JPG Images while the second is the simulation as a CSV file and the last being the PDF File.

The “*Images*” directory contains the following 10 images of our simulation:



The PDF File, when opened, shows each of the time steps saved in a high-quality landscape page:



COMP1005 – Fundamentals of Programming

Due to the enormous size of our population, the main lifeforms array, during the coding process, went up to the same size resulting in the CSV File to have the same criteria.

The lifeforms array for each time step were saved under the array for the previous time step making the file quite long, but still usable for data analysis and interpretation. At the end of each time step array is mentioned the death toll for each lifeform in that particular step:

402	52	75	2		
403	198	40	2		
404	128	96	1		
405	46	80	2		
406				Red Deaths: 3	
407				Yellow Deaths: 0	
408				Blue Deaths: 2	
409					
410	Time Step 3				
411	105	32	1		
412	13	49	0		
413	151	53	1		
414	11	8	1		

At the very end of the file are present the overall statistics of the entire simulation:

1925	95	12	4		
1926	152	1	1		
1927	100	77	2		
1928	174	84	0		
1929	198	48	1		
1930	52	77	2		
1931	130	98	1		
1932	45	78	2		
1933				Red Deaths: 0	
1934				Yellow Deaths: 0	
1935				Blue Deaths: 0	
1936					
1937					
1938					
1939					
1940	Total Red Deaths: 11				
1941	Total Yellow Deaths: 4				
1942	Total Blue Deaths: 9				
1943					
1944	New Population: 176				

This very same criteria and methodology can be applied to the simple simulation, reproduction and intruder functions which are bound to yield the same results as the example given above without interceding with each other.

5.0 Conclusion and Future Work

It can be said in the end that the program formulated to display the simulation of the planet was successful in terms of allowing the users gaining relevant statistics and ground values that have the potential to be used.

The formulated code involved more than one type of simulation allowing various scenarios to be made up on the planet. These were successfully transpired using various functions already described in much detail previously.

To obtain a much more reliable program an addition can be made into the current one allowing an added bundle of extra function enabling better future work.

Examples of this may include introducing a code allowing the user to re-simulate a previously run code. This could be made possible by reading the output from an already saved CSV File for that particular simulation and then using the same x, y and/or z values to plot the same co-ordinates of each lifeform. Other than this, a 3D plot of the battle, reproduction and intruder simulations could be made which were not previously coded for. The graphs plotted could be made more user interactive involving the user to practically see and note the position of each lifeform in the planet from the graphical display window. A simulation to include all three functions could be made allowing the lifeforms to be destroyed and new ones to be created including the presence of intruders. The limit for the lifeform population could be increased from 1000 whereas the planet boundaries can be restricted to a set amount for all simulations carried out irrespective of the population size.

All scenarios help show the expected life on the planet PaSciRo, enabling different simulations to be run and statistical data to be extracted from each time step. In this way the program is able to successfully carry out its function, according to the user's desires.

6.0 References

Basis of obtaining co-ordinates and plotting derived from sample code, *"main()"*.

3D Plotting – Tanya Sri. 2020. GeeksforGeeks: *"Three-dimensional Plotting in Python using Matplotlib"*.

<https://www.geeksforgeeks.org/three-dimensional-plotting-in-python-using-matplotlib/>.

Exception Handling – Vishal. 2020. PYNative: *"Check user input is a number or string in Python"*.

<https://pynative.com/python-check-user-input-is-number-or-string/>.

Indexing of array – w3resource. 2020. W3resource: *"NumPy: Search the index of a given array in another array"*.

<https://www.w3resource.com/python-exercises/numpy/python-numpy-exercise-171.php>

Maximizing the window of plotted time step – Admin. 2017. ExceptionsHub: *"How to maximize a plt.show() window using Python"*.

<https://exceptionshub.com/how-to-maximize-a-plt-show-window-using-python.html>.

Plotting a Circle – Daidalos. 2019. Open-Notebooks: *"How to plot a circle in python using matplotlib?"*

<https://moonbooks.org/Articles/How-to-plot-a-circle-in-python-using-matplotlib/>.

Re-plot in the same figure – stack overflow:

<https://stackoverflow.com/questions/35595766/matplotlib-line-magic-causes-syntaxerror-in-python-script>

Saving plots into a single PDF File – stack overflow:

<https://stackoverflow.com/questions/17788685/python-saving-multiple-figures-into-one-pdf-file>