LABORATORY REPORT

To: Dr. Martha Garlick

From: Micah Runner

Subject: Confirming motility values of deer populations with chronic wasting disease using cellular automata.

Date: July 31, 2021

Introduction

The goal of this project was to introduce a method of testing motility values calculated using a set of partial differential equations that predict the path of a deer with Chronic Wasting Disease (CWD) over various landscapes. The combination of Cellular Automata and Perlin Noise allows for pseudo-random, but repeatable terrain to be created, that allow a list of calculated motility values to be simulated by our silicon friends. The model's code and documentation can be found at the *GitHub* repository.

Methodology

This section is used to give a top level understanding of the logic behind the underlying code found with the model. The two highest level parts are the World Creation and the Deer Path Modeling.

World Creation

This section covers how the world is generated and why certain choices were made when creating the world. Topics include background information of Perlin Noise, Terrain Generation, and Coloring Terrain, as gray scale models are boring to look at with the human eye. To begin with we will discuss the use of Perlin Noise.

Perlin Noise

Perlin Noise is a noise generation type commonly used within gaming programs as it allows for varying continuously generated terrain. Many of the games allow users to enter in a "seed", which is a string of numbers that control the variables used within the Perlin Noise that creates the unique worlds, such as *Minecraft*. The biggest downside to using Perlin Noise is the time complexity requirement of generating the world, which is $O(2^n)$, where n is the number of dimensions being used within the world. This means one of the slowest parts of the program will be generating the 2-dimensional world that has a large number of length and width. Ideally we would Simplex Noise, which is similar to Perlin Noise but requires less time complexity when generating noise. Due to being unsure of the legality of Simplex Noise, Simplex Noise was not used for this model.

Perlin Noise works by by creating a grid of points that create four sub-grids for each original grid. The dot-product for each of the four sub-grids is taken to find a gradient vector. Once the gradient vectors have been found, interpolation is then used to smooth out the sub-grids, as they get rounded towards the nearest grid point. This process is why the generation is slow as a 2-dimensional grid requires four gradient vectors for each grid point, thus giving $4 = 2^2 = 2^n$ or $O(2^n)$.

The following two figures show why Perlin Noise was selected for its pseudo-random generated noise patterns. Figure 1 shows randomly generated Perlin Noise with the variables used to generate the noise pattern. Figure 2 shows the same noise pattern being generated using the same variables that were randomly generated in Figure 1. This allows for tests to be conducted and verified by any user assuming that the same parameters are selected.

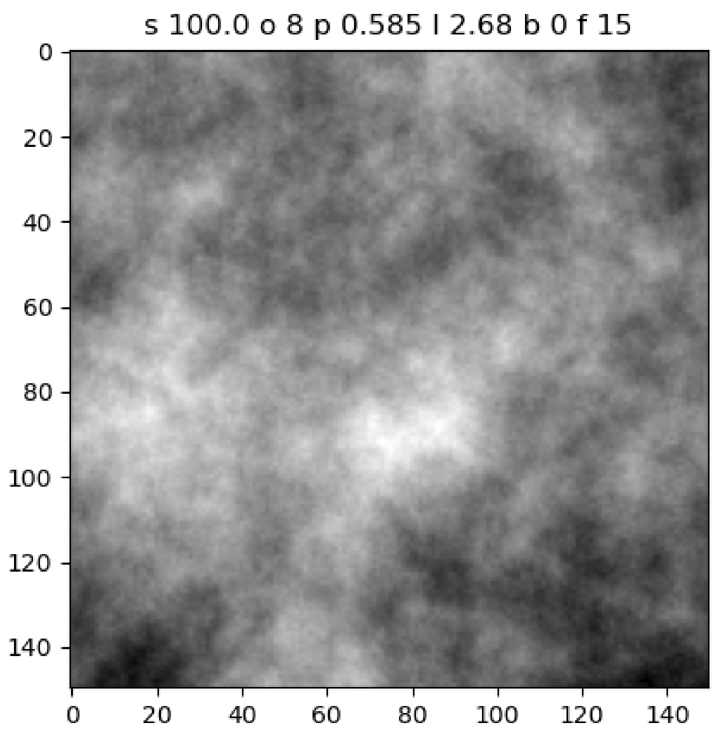


Figure 1: *Randomly generated Perlin Noise.*

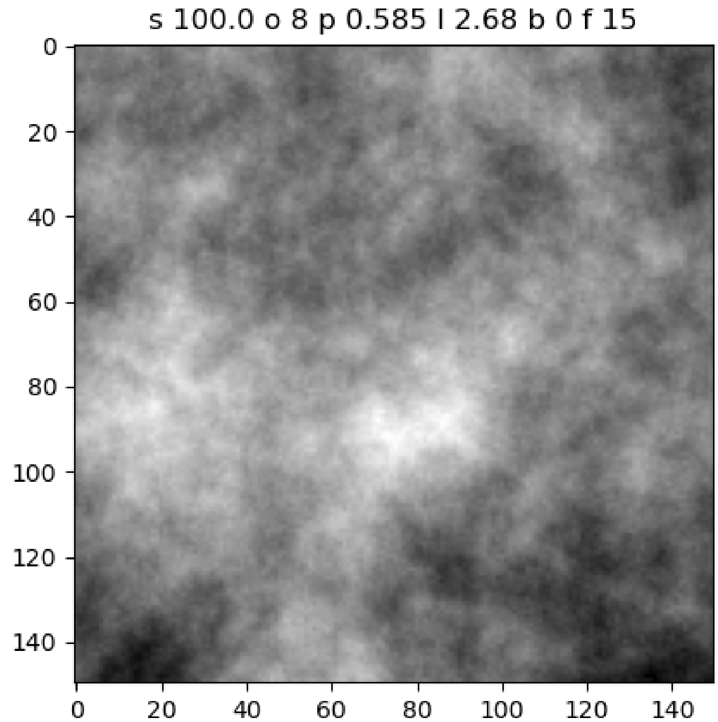


Figure 2: *Repeated generated Perlin Noise using the same parameters that were originally selected.*

Terrain Generation

Perlin Noise as just noise is not extremely useful as it is smoothed noise, which is why we decided to smooth the noise further into terrain chunks of values that are specified by the user. This allows for the worlds to be more consistent to real terrain. The terrain chunks are created by using a locking system that rounds values to the nearest locking value. For example, if the user gave the following locking values [-0.6, -0.1, 0.1, 0.6] and the Perlin Noise values were [-0.9, -.59, 0, 1]. -0.9 would be changed to -0.6 as it is less than the locking value of -0.6. -0.59 is greater than the -0.6 locking value but less than -0.1, so it would turn into -0.1. 0 would be turned into 0.1 and 1 would not be changed to 0.6 as it is not less than 0.6. The user should always have their last locking-value to 1 to stop this from occurring.The final Perlin Noise world array would be [-0.6, -0.1, 0.1, 1].

The figure below shows the locking-system applied to the Perlin Noise in Figure 1. For this example we used a 15 feature locking-system that ranged from values of -0.867 to 1. The terrain that is now composed of 15 different values is then saved to preform all mathematical equations within the cellular automata simulation.

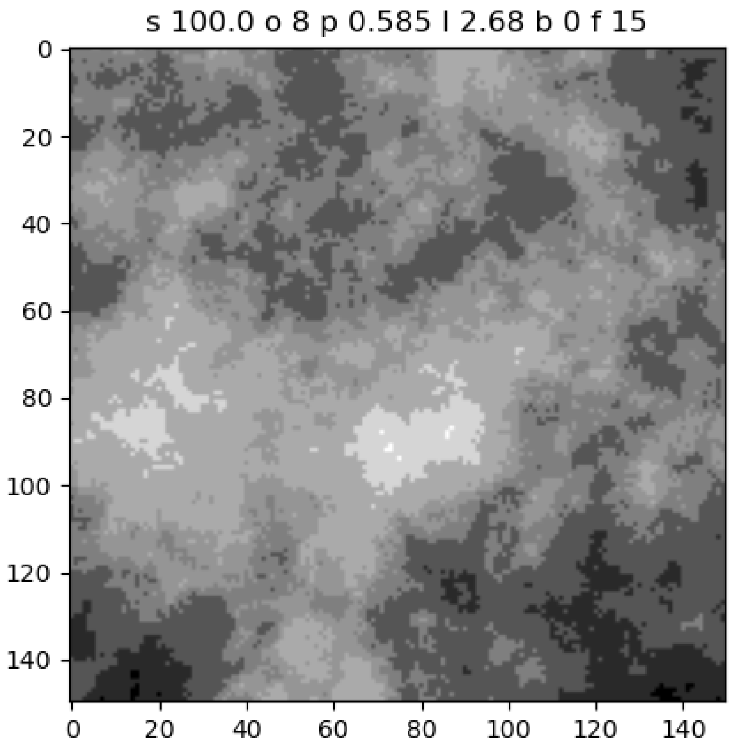


Figure 3: *Fifteen feature locking-system generating terrain features from the Perlin Noise generated in the Figure 1.*

Coloring Terrain

We decided to go a step above just generating terrain by adding color to the world by using RGBA. This provides all the colors of RGB but with the ability to change the a transparency of the RGB pixels, which is used when (tracking the path, make link to that section of paper) of the deer within the CA model. The color generation relies on two different arrays provided by the user.

The first being an array of RGB integer values ranging from 0-255, that they wish to use within the model. The total amount of The color array will then be changed to RGBA values that are floating point values from 0 to 1, with either the alpha, 'A', set to either 0.1 or 1 depending on how they want the model to track the path.

The second being an array of lock-system values that are to be used to smooth the Perlin Noise further. These values should range from -1 to 1 as Perlin Noise will generate values from -1 to 1.

Once these arrays have been supplied, the Perlin Noise will only contain the values of the lock-system values. This creates a pseudo-random terrain, which then receives a RGBA value and results in an output found in the figures below.

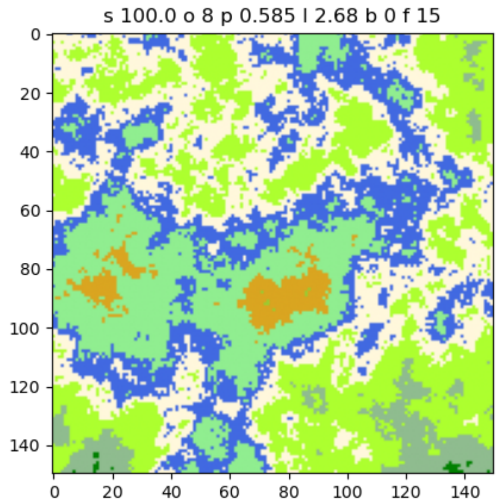


Figure 4: *Fifteen feature color terrain from the Perlin Noise generated in the Figure 1.*

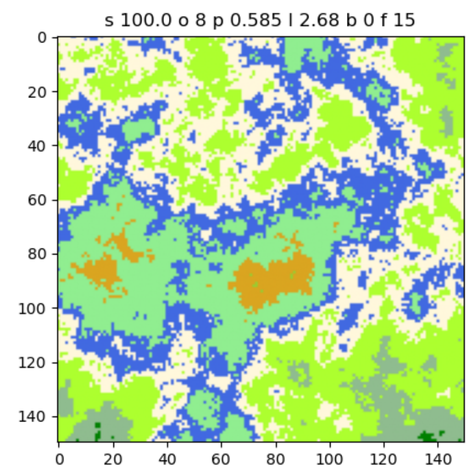


Figure 5: *Fifteen feature color terrain from the Perlin Noise generated in the Figure 2.*

Deer Path Modeling

Cellular Automata

Cellular Automata was originally selected for this project as recent Z-transformations have allowed for continuous solutions to a select few partial differential equations such as the heat equation, the wave equation, and Laplace's equation. This provides a relatively accurate solution to approximating a PDE, while also being relativity computationally cheap. Unfortunately, we could not transform this problem over to be solvable via cellular automata. However, we did come up a method of testing terrain motility values generated by a differential equation solution made through coffee and coffee alone.

Cellular Automata at its core is a grid of points, that represent a state, and a set of rules. This results in a simplistic model, but can result in complex output such as Conway's game of life, which has its own game within one of the possible outcomes. For our model, we decided that each point within the grid would either be a terrain type or the deer itself would become the point until the next iteration. Unlike most cellular automata models, this model does not have a rule set that spreads and creates, but rather determines a best path.

Rule Set

The rule set for this model has the following two parts. Moore Neighborhood, for movement between each point on the grid, and Path Selection, which is a simple mathematical approach to simulate how a deer would select a path given different terrain features. All terrain features have a motility value that was calculated via partial differential equations, which the deer will tend towards the lowest motility values. To add a bit of a random walk the normal distribution was included to the motility. We will first explain what the Moore Neighborhood and why it was selected, then we will go in depth on the Path Selection process.

Moore Neighborhood

The Moore Neighborhood is one of the two typical ways of having points interacting with each other. The Moore Neighborhood shown in the figure below, allows the middle point to interact with not just the top, left, right, and bottom points, as is the case with Von Neumann Neighborhood, but with the corners to make a complete square around the middle point. This is how the deer moves between points within the model, as a deer in the real world can move in any direction that is physically possible. Originally, we intended to only use this method for selecting the where to move the deer, but found the results were to random for an animal that can process terrain and make movements based off the terrain.

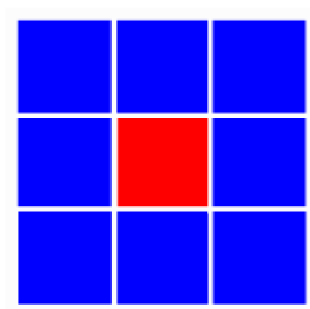


Figure 6: *Image showing the square that is allows the middle point to interact with the remaining points, otherwise known as the Moore Neighborhood.*

Path Selection

The Path Selection process is where most of the modeling magic is done as the generating a world is relatively easy to achieve. As stated above, we originally tried only using the Moore Neighborhood to determine where the deer would move with the equation below:

$$pos_{\text{next}} = \begin{cases} x_i, & \text{IF } \mathrm{x_i} < \text{ave(x) + normal, where \{x:= Moore Neighborhood, \quad normal:= normal distribution\}} \\ pos_{\text{current}}, & \text{Other} \end{cases}$$

This implementation resulted in random paths for the deer as it would leave terrain features with low motility for a terrain feature of high motility. After two random motions forward, it would be too far to return the back into the low motility terrain feature.

We decided to implement a new method of looking three points ahead and turn the Moore Neighborhood points into the average of the other future possible points. Figure 7 shows the an approximate overlapping view of the total viewing square that is being used of path selection.

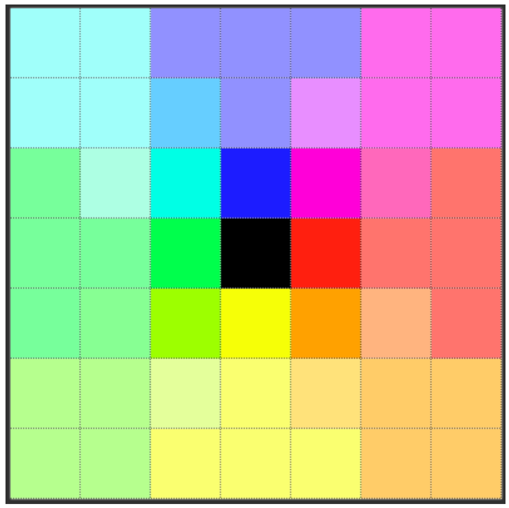


Figure 7: *Image showing the Moore Neighborhood with an expanded view for more accurate deer path selection.*

The figure below shows which points, or squares, are being used to calculate the top right square of the Moore Neighborhood by averaging the future possible squares. The dark purple square becomes the average of the motility values of the lighter purple arrow shaped squares.

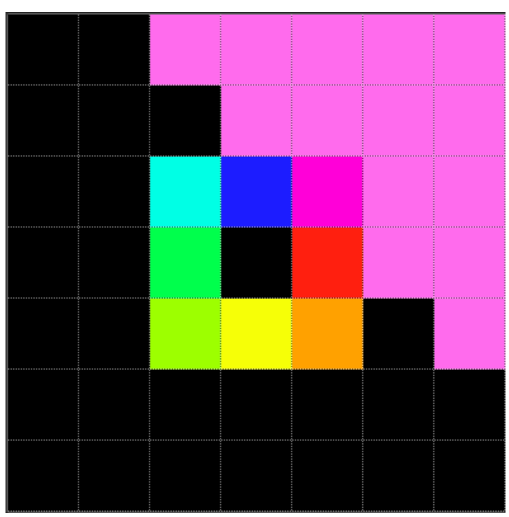


Figure 8: *Image showing the Moore Neighborhood focusing on the top right square.*

Figure 9 shows the averaging of the top middle square, shown in blue, by averaging the top two rows, shown in light blue. The choice of taking the entire two rows instead of a v-shape, was due to the corners of the viewing square all having 12 points to average.

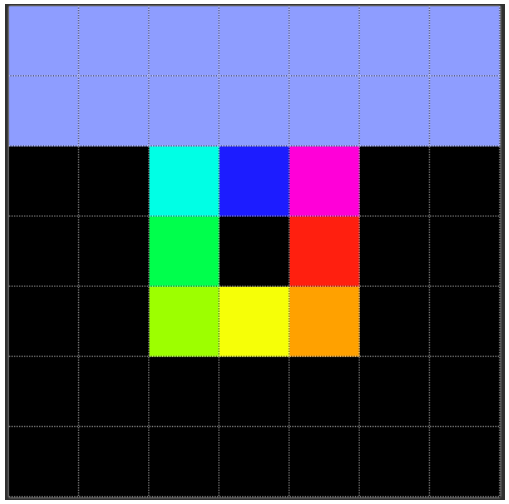


Figure 9: *Image showing the Moore Neighborhood focusing on the top middle square.*

Figure 10 shows the averaging of the top left middle square (in cyan) by averaging the top left arrow of squares in

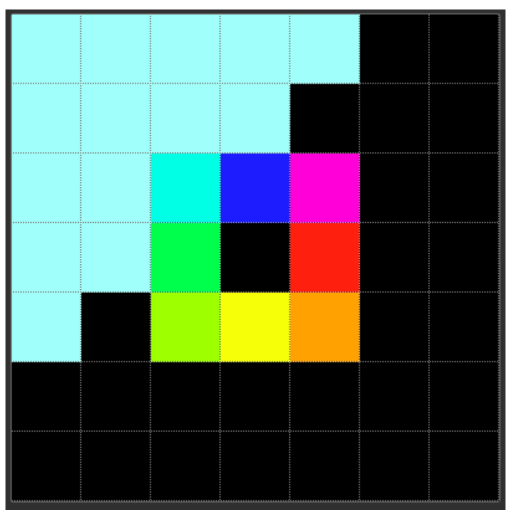


Figure 10: *Image showing the Moore Neighborhood focusing on the top left square.*

The following figures are of the remaining methods of averaging the future possible paths. The left middle, bottom middle, and right middle squares also follow the same amount of total squares shown in Figure 9, which adds in extra squares that are not exactly in a linear line as with the corner cases.

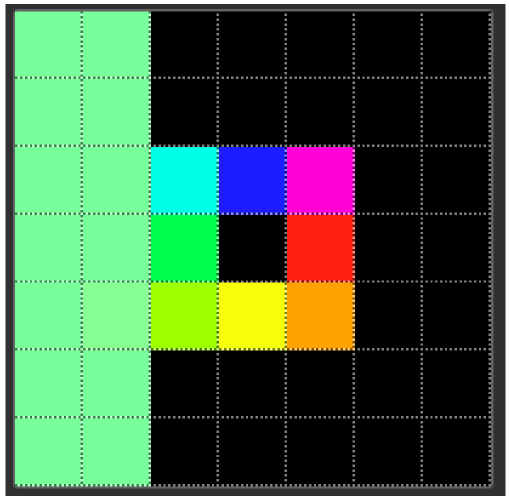


Figure 11: *Image showing the Moore Neighborhood focusing on the middle left square.*

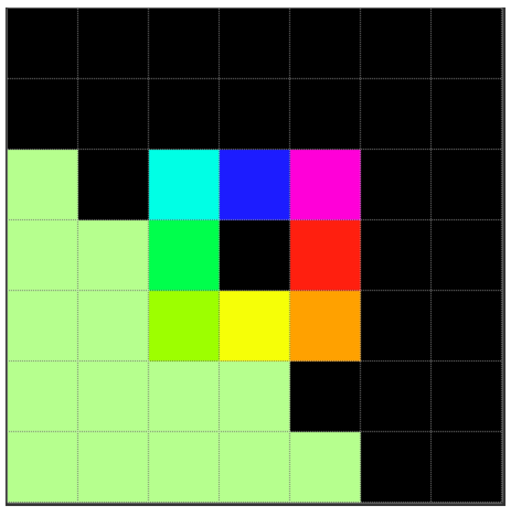


Figure 12: *Image showing the Moore Neighborhood focusing on the back left square.*

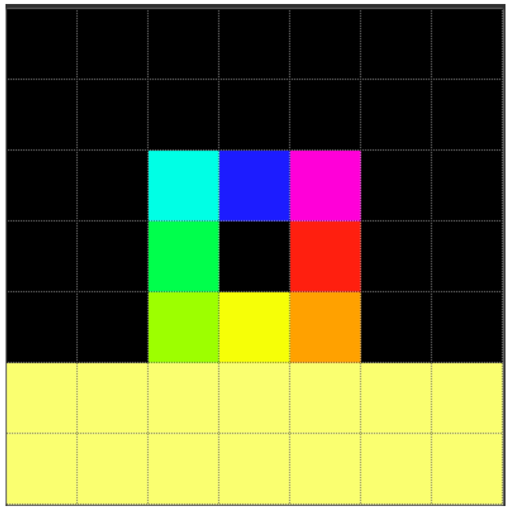


Figure 13: *Image showing the Moore Neighborhood focusing on the back middle square.*

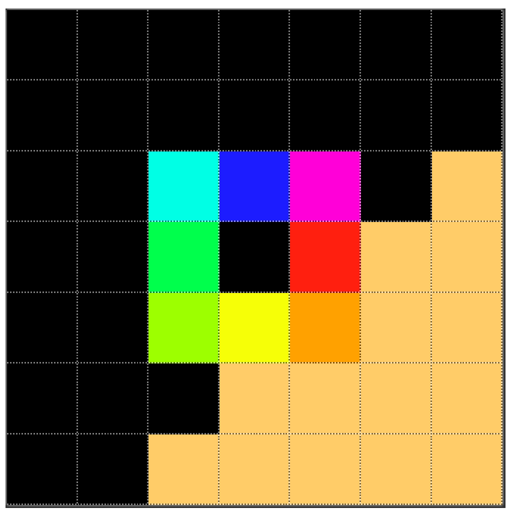


Figure 14: *Image showing the Moore Neighborhood focusing on the back right square.*

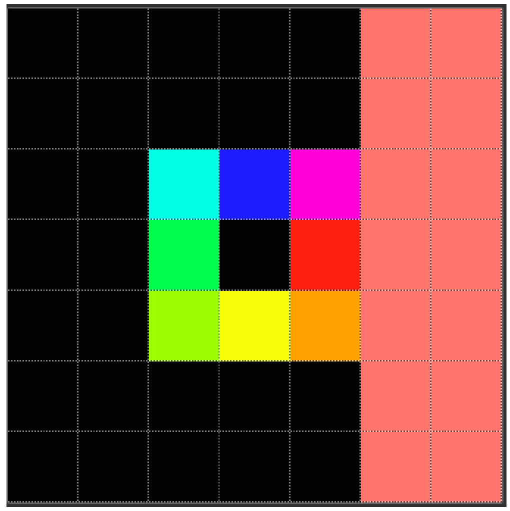


Figure 15: *Image showing the Moore Neighborhood focusing on the middle right square.*

Path Tracking

The deer's path is tracked via two different methods, one of which visually shows the user and the other keeps a detailed list. We will first discuss the visual method.

The first method is shown via two different images after total amount of iterations have been ran throughout the Cellular Automata. The path is tracked by adjusting the alpha value (transparency of the RGBA pixel) every time the deer occupies the square. This means that the pixel will become more transparent the more the deer returns that to that pixels location within the grid.

Figure 16 shows the final iteration of the simulated world and the deer's path through the terrain. This version decreases the transparency after each time the deer occupies the square, which results in the square getting darker. The second visual is shown in Figure 17, which is the same as Figure 16, but with all unused pixels set to be a black pixel. This allows the more transparent and less occupied pixels to be more observable as opposed to just setting all unused pixels to being completely transparent.

The second method is an excel output file that is determined by the user. This provides the user with the terrain type name, the motility value associated with the terrain type, and the axis position in relation to the generated world (shown in Figure 18). The library that handles the conversion from an array to the excel file shows array indexing on the left hand column of the file. To our knowledge there is no way to remove this column.

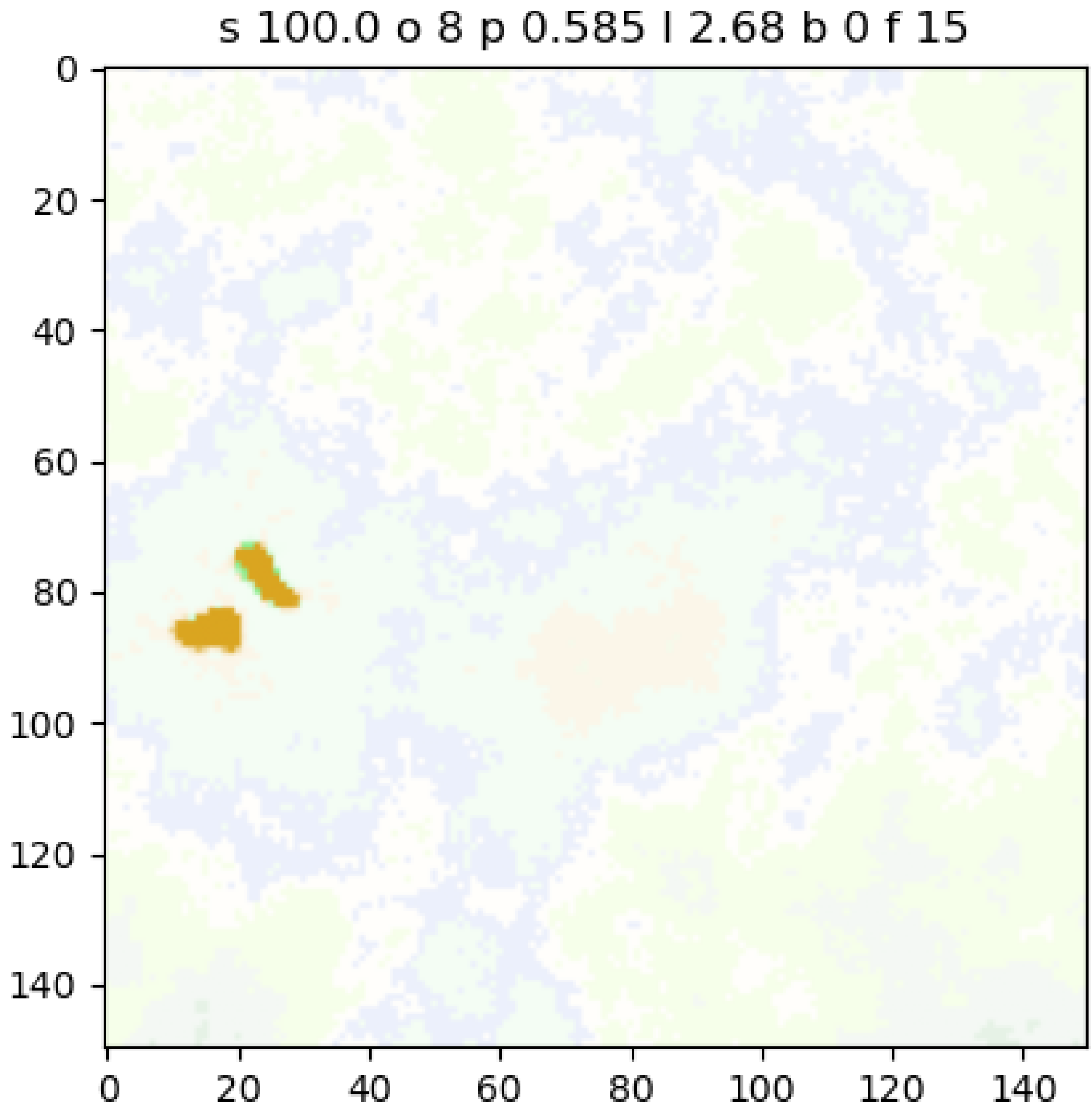


Figure 16: *Image showing the last iteration of the deer's path using the same world in Figure 1.*

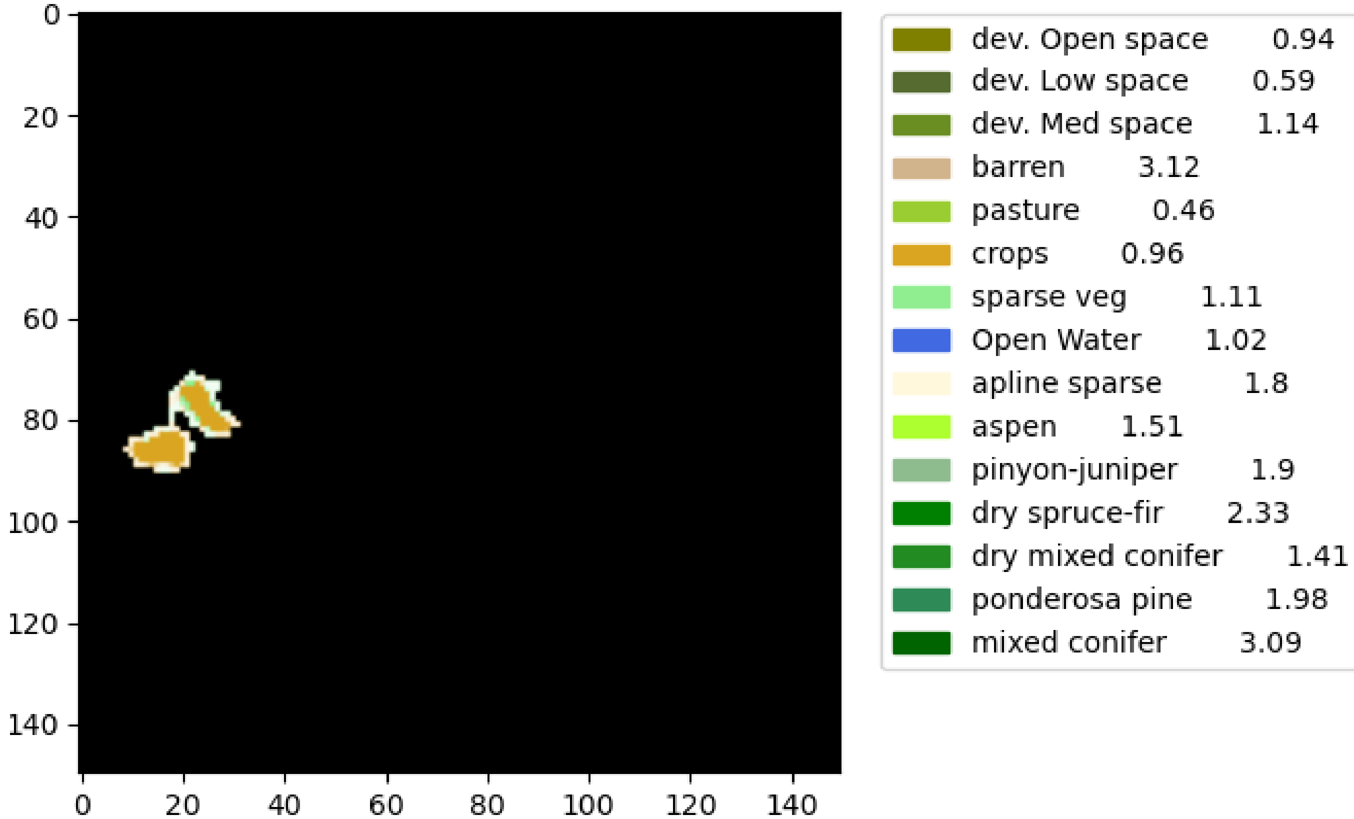


Figure 17: *Image showing the deer's path as in Figure 16, but with all other untouched squares set to black.*

	A	B	C	D
1		**Terrain**	**Motility**	**Axis Position**
2	**0**	crops	0.96	[86, 21]
3	**1**	crops	0.96	[86, 21]
4	**2**	sparse veg	1.11	[85, 22]
5	**3**	sparse veg	1.11	[86, 22]
6	**4**	crops	0.96	[87, 21]
7	**5**	crops	0.96	[87, 20]
8	**6**	crops	0.96	[86, 19]
9	**7**	crops	0.96	[85, 18]
10	**8**	crops	0.96	[84, 18]
11	**9**	crops	0.96	[85, 17]
12	**10**	crops	0.96	[86, 16]
13	**11**	crops	0.96	[87, 15]
14	**12**	crops	0.96	[86, 15]
15	**13**	crops	0.96	[85, 16]
16	**14**	crops	0.96	[85, 17]
17	**15**	crops	0.96	[86, 18]
18	**16**	crops	0.96	[85, 17]
19	**17**	crops	0.96	[85, 18]
20	**18**	crops	0.96	[86, 19]
21	**19**	crops	0.96	[85, 18]
22	**20**	crops	0.96	[86, 18]
23	**21**	crops	0.96	[85, 19]
24	**22**	crops	0.96	[86, 18]
25	**23**	crops	0.96	[87, 17]

Figure 18: *Image showing the deer's path in an output excel file.*

Usage

This section covers the base usage case to operate the preset defaults of the model all the way up to running the model to its full capabilities as well as some potential code hacking if time permits.

Default Case

We will talk about how to setup the program to run the preset defaults, which can occur if the user does not setup there more advance use cases for the program.

The first set is to install all the needed libraries that are used within the model, which runs *Python* 3.8. This model uses the following libraries: *numpy* version 1.20.2, *pandas* version 1.2.5, *matplotlib* version 3.3.4, and noise version 1.2.2. This model was created using the *Python Conda* environment within the *PyCharm* IDE. Appendix A shows all of the libraries that are installed within the *PyCharm* IDE, but the majority of them should not be required to run the program. If running *PyCharm* the IDE will ask if the user if they would like to install the needed library to the project, which means the user should install them if they want to run the model.

The second step is understanding which functions to call in the correct creation order. The figure below shows the flow chart of correct function calling. The flow chart can also be found within the *Viso* file in the *GitHub* repository. We recommend using the **CaDeerMotiltiy_testing** file as your editing file, as it provides how to import the class file.

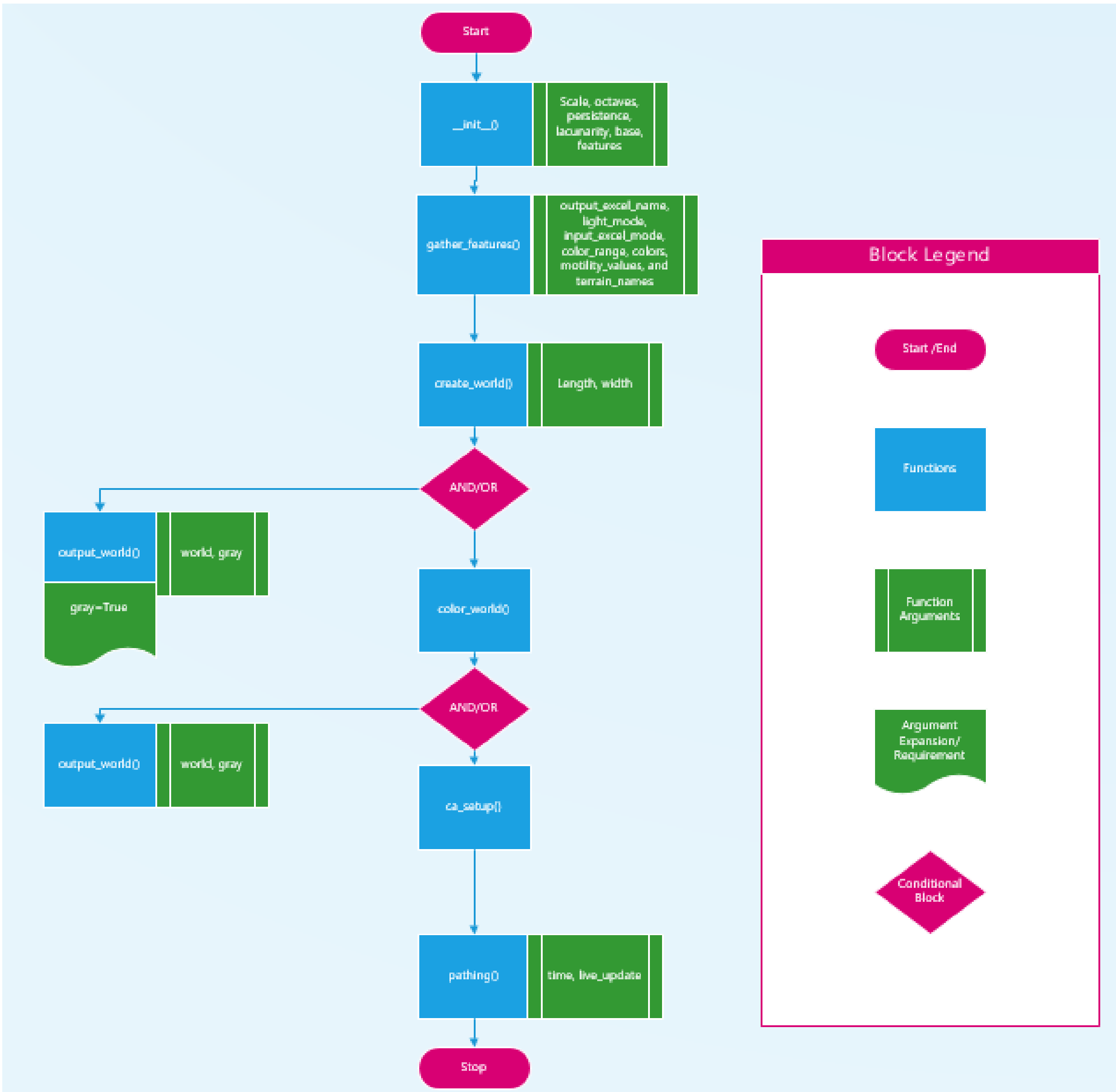


Figure 19: *Image showing the flow chart of the correct function calling.*

Class Initialization

Class Initialization begins by assigning a variable, say **deer**, to the class name *CaDeer*, which would be: deer = CaDeer() This can occur as every initialization variable has it's own default values. We will talk more in depth on what those variables do in a more advance section. The variable **deer** is now the entire class structure of the model, which means all other function calls and change of individual variables can be accessed by: deer.function/variable

Gather Features Class Function Call

We now need to gather the output file name for the excel output file that contains the deer's path throughout the generated world. This is achieved via: deer.gather_features("test_output") Notice how this time the function needed a variable to be passed into it for the function to work. The output excel name needs to have a string variable for the function and future functions to work. Again all other variable controls will be explained in the function and variable section of the report.

Create World Class Function Call

The next required function is the function that uses Perlin Noise to create a pseudo-random world. The required function call is: deer.create_world()

Using Class Variables

This is an optional step but shows the user how to access variables stored within the class structure. We will output the Perlin Noise using the class output function and class variable that holds the Perlin Noise. This function call will be as follows: deer.output_world(deer.world, True)

Color World Class Function Call

This is the next required function call unless the user wants to do some class variable hacking, which will talked about later on how to speed up the program by skipping generation and just running the path simulation section of the model. Notice that this function does not have any inputs as it relies on all class variables. This function call is simply: deer.color_world()

Optional Output Viewing

These next two function calls are used to show the RGBA world in all of its glory and a gray scale version of the RGBA world, which is used for all of the deer path movement math. The first output is for the RGBA: deer.output_world(deer.world_color) and the gray scale: deer.output_world(deer.ca_world, True)

Path Class Function Call

This is the last required step, which runs the simulation of the deer as it has a random location within the world that was generated a few steps ago. This function call requires an integer variable that controls how many iterations the simulation will run for as simulating time. The function call is: deer.pathing(10000) This will give a decent amount of time for the simulation to run, but the simulation is limited to five features. See the **Conclusion** section to see the output, but remember everything is randomized between runs.

Advance Case

The advance case is how to use the model to its fullest using all the extra variables. For more in depth view on the variables check the *Viso* flow chart, which has all of the required functions and variables. If the user is using PyCharm they can hover their cursor over the function, which will pull up the hidden variable comments, which provides what type the variable is as well as a description of what the variable controls within the program. The figure below shows an example of *PyCharm* function cursor hover. This section will cover all the variables used within the required functions and the output.

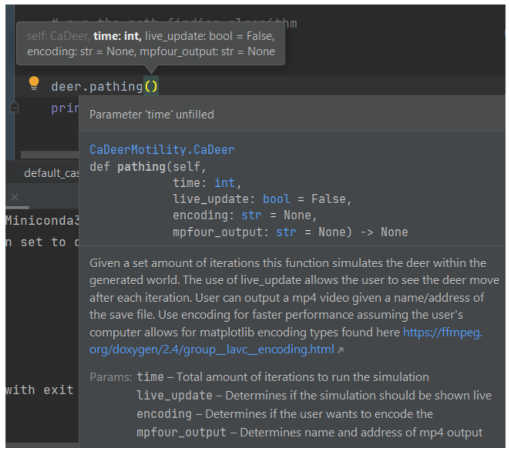


Figure 20: *Image of the class commented code showing what the function does and the variable types and uses by hovering the cursor over a function call in PyCharm.*

__init__

The function __init__ is used to initialize the class. This function is used when the class is being assigned to a variable. This function supplies users input to setup the Perlin Noise generation variables as well as check the class variable **features** to see if it is less than 5. If found to be true than the default of 5 features will be assigned. *Note! features* is used a checking variable in all other functions, which means that not matching it with the total number of names, motility values, RGB values, and color locking variables will result in a default mode.

- scale
  - Controls the viewing scale of the generated Perlin Noise. Default is 100.0
  - *float, optional*
- octaves
  - Number of features that will appear within the distinct float steps generated within the Perlin Noise. Default is 6.
  - *int, optional*
- persistence
  - Number of repeating characteristics within the Perlin Noise. Default is None, which will be changed to a random float between (0.2, 0.6).
  - *float, optional*
- lacunarity

  - Level of detail found within the Perlin Noise. Default is None, which will be changed to a random float between (2.2, 3).
  - *float, optional*

- base
  - Determines starting point of the map. Default is None, which will be changed to an integer between (2, 25).
  - *int, optional*

- features
  - Number of features to color and use in model. Must match the number of elements within the colors, color_range, feature_list, and names arrays. Default is 5, and will be used for all other functions to run the cellular automata.
  - *int, optional*

gather_features

This function collects many of the needed variables used for creating the world as well as variables used for running the pathing function. This function requires one variable(**output_excel_name**) to be passed into the function for it to operate. This is required to output the excel file name the user wishes to store path data. The other variables are optional but allow the user to switch between having pixels start mostly transparent or not transparent, read from an input excel file, give the color locking values, RGB values, motility values, and terrain names. This function will prioritize the excel input file over inputting arrays into the last four variables.

- output_excel_name
  - Used to name the output excel file name.
  - *string*

- light_mode
  - Used to determine if the RGBA will either get less transparent or more transparent.
  - *bool, optional*

- input_excel_name
  - File path/name of the excel file that provides the color_range, colors, motility_values, and terrain names. Must match the number of features for each variable.
  - *string, optional*

- color_range
  - Holds the values to determine cutoff values for RGBA. Must match the number of features.
  - *ndArray, optional*

- colors

- Holds the RGB values that the user wishes to use to color the world. Must be a ndArray composed of 3-int ndArrays. Example ndArray([[0,0,0], [255,255,255]]). Must match the number of features and ints must be range from 0-255 not 0-1.
- *ndArray, optional*

- motility_values
  - Holds the motility values of each terrain type, which will be used when the simulated deer makes movement choices. Must match number of features
  - *ndArray, optional*
- terrain_names
  - Holds the names of the terrain that are used within the simulation. Must match the number of features.
  - *list, optional*

create_world

This function takes in two optional variables, **length** and **width**, that determine the size of the world. It then generates Perlin Noise in the size and shape of the world.

NOTE! This section can be skipped if the user provides their own ndArray of values and setting it equal to the class variable **world**. The user will then have to set the **length** and **width** variables to match the ndArray size. This will reduce the most amount of time for the program as the Perlin Noise takes the longest amount of time.

- length
  - Sets the length of the world.
  - *int, optional*
- width
  - Sets the width of the world.
  - *int, optional*

output_world

This function outputs an ndArray of floats or a ndArray of RGBA to the user. The ndArray is required for the function to work. The optional variable is **gray** which allows the user to use gray-scale. This should not used with the RGBA values.

- world
  - Array of RGBA values that represent the world.
  - *ndArray*
- gray
  - Determines if the world should be shown in gray scale.

    - *bool, optional*

color_world

This function is used to create the RGBA world as well as terrain map for the pathing to use for mathematical purposes.

This section can be skipped if the user sets the class variables **world_color** to a ndArray of RGBA values and **ca_world** to a ndArray of only values that are used within **color_range**. This skipping does not speed up the program by much, but is possible.

pathing

This function is the meat and bones of the model as it tracks and models the deer's behavior and its pathing throughout the simulated world. This function requires a number of iterations to be passed in by the user via the variable **time** as it represents time within the simulation. The other variables allow the user to watch, in real-time, the deer's path, enter an encoding type if they have hardware that can increase encoding times of a mp4 file, or input a name for the output of the mp4 file.

Note! Check list of encoding that can be used, or just leave **encoding** blank if you are not sure. Having the mp4 output file and live update options both selected will result in only the mp4 output, as this speeds up the simulation process, and then in-turn the mp4 output. Lastly, the mp4 output requires a lot of time and a lot of memory.

- time
    - Total amount of iterations to run the simulation.
    - *int*
- live_update
    - Determines if the simulation should be shown live.
    - *bool, optional*
- encoding
    - Determines if the user wants to encode the mp4 with a specific set of encoding instructions.
    - *string, optional*
- mpfour_output
    - Determines name and address of mp4 output.
    - *string, optional*

Advance Case Function

The following can be found within the included testing file found in Appendix B.1 under the function **advance_case**. This shows how to properly fill in all function variables and includes how where to set starting positions for the deer on a world before the pathing function begins.

Hacking Case

This is section of code is included within the testing file found in Appendix B.1 under the function **hacking**. This part includes how to input without the use of an excel file as well as placement to bypass the Perlin Noise generation, but the user will have to know how to input a custom world into the specified variable as well as account for the length and width of the custom world.

Conclusion

The mixture of Cellular Automata and Perlin Noise world generation worked decently well as the Perlin Noise generation adds in pseudo-random terrain generation that can be recreated by any other user given the same generation parameters. The use extra viewing range in the Cellular Automata allowed for a less random walk through the terrain in the pathing simulation, which now results in a path closer to a path the real deer are selecting. The user may reverse the motility values, by selecting random motility values with terrain and then put those values through the simulation to observe pathing. If the pathing is more accurate, they can then use those new motility values within the partial differential equations that gave way to the original list of motility values. Below are two of the outputs from running the testing code provided in Appendix B.1.

One downside of using Perlin Noise and the locking color/terrain range is that the user will have to spend time figuring out either the order that terrains should go in with respect to color locking range system, or what color locking range values work best for the terrain they want to generate. This has an upside as putting in a little bit of extra time can result in very realistic terrain generations.

Default Case Output

The following is the output of the default case results, which only has five features. The pathing did decently well as it journeyed from the top of the map from terrain that had a high motility with a little bit of random movement. It then down to the bottom left of the world, which had a low motility value and stayed most of its time there. This is what we expect to happen with a good run. Ten-thousand iterations is a minimum iterations that should be ran as it gives enough time to show a path that a deer would take in real life. This is shown in Figures 24 and 25.

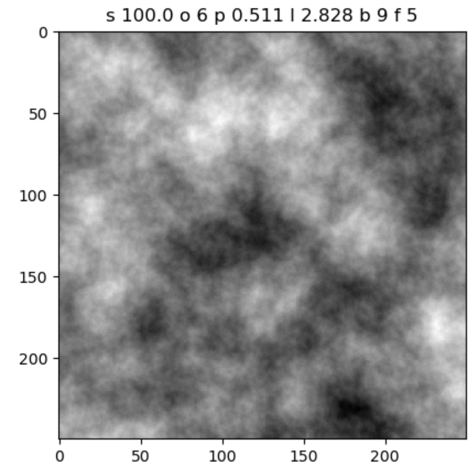


Figure 21: *Image of the Perlin Noise generated with the example usage for the Default Case.*

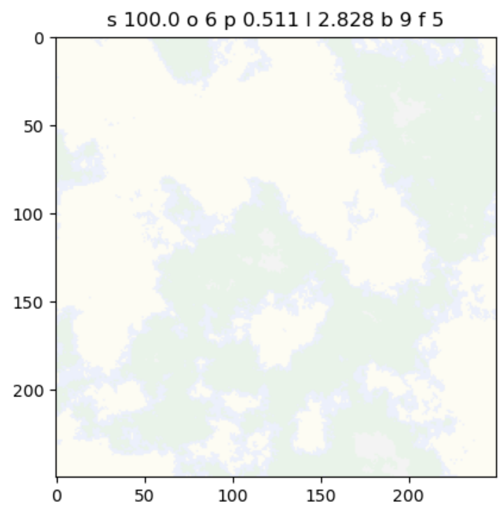


Figure 22: *Image of the RGBA world with the example usage for the Default Case.*

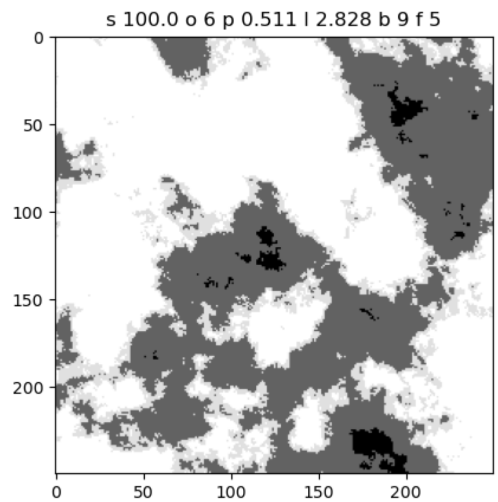


Figure 23: *Image of the RGBA world with the example usage for the Default Case.*

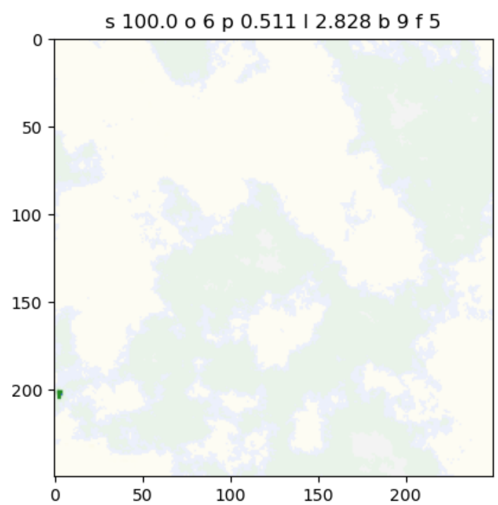


Figure 24: *Image of the final pathing used in the example usage for the Default Case.*

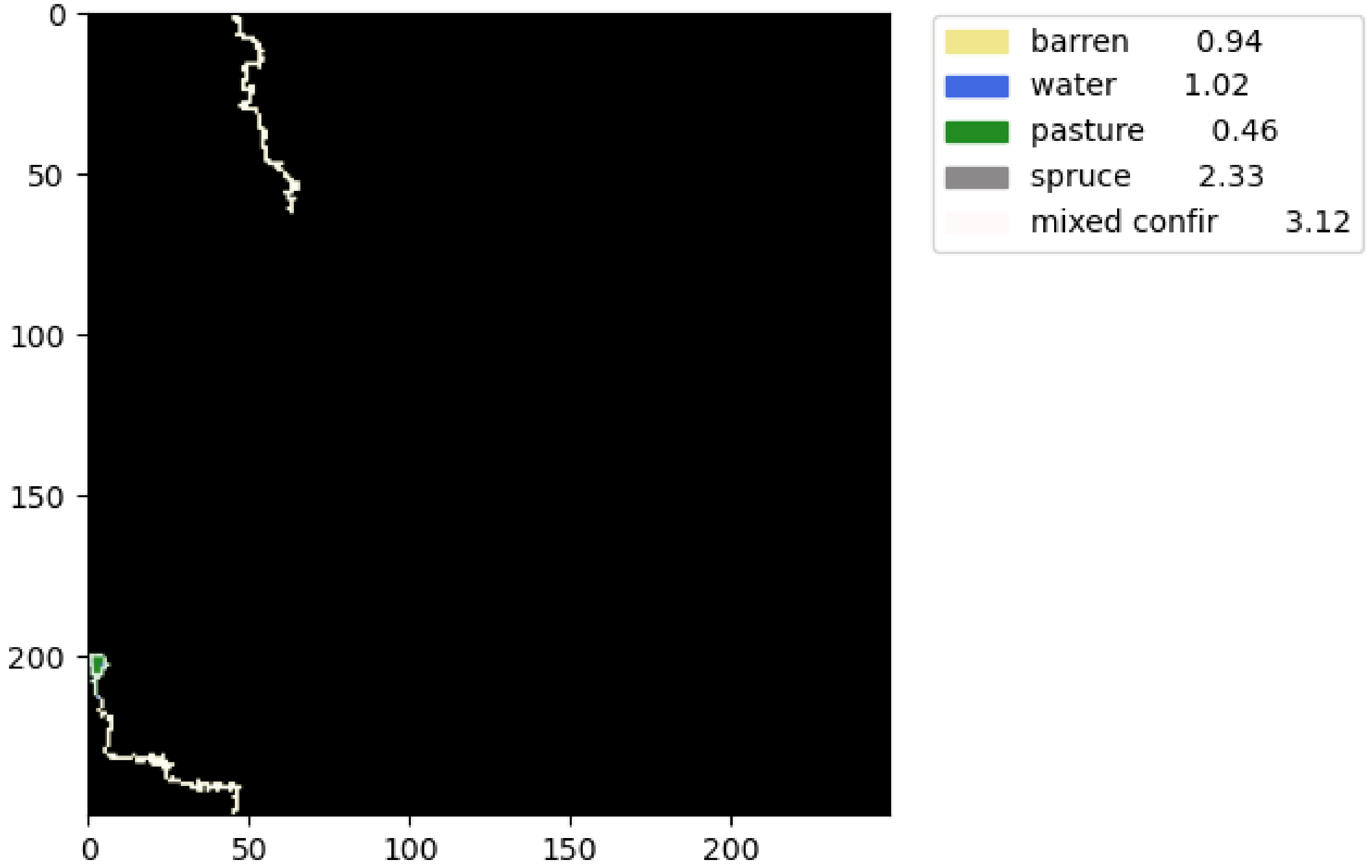


Figure 25: *Image of the path in better contrast used in the example usage for the Default Case.*

Advance Case Output

The following is the output from using the Advance Case Output function, which happens to be a randomly placed deer on the world created in Figure 1. The water appears to be a bigger problem than what is actually happening. The model is still correct according to the motility values of the water and the surrounding land motility values, as the water has a lower motility than the land motility values.

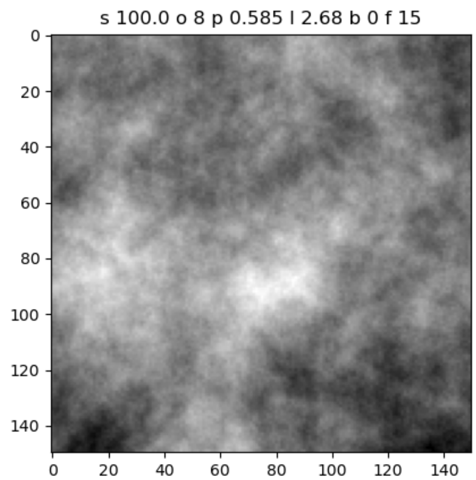


Figure 26: *Image of the Perlin Noise generated with the example usage for the Advance Case.*

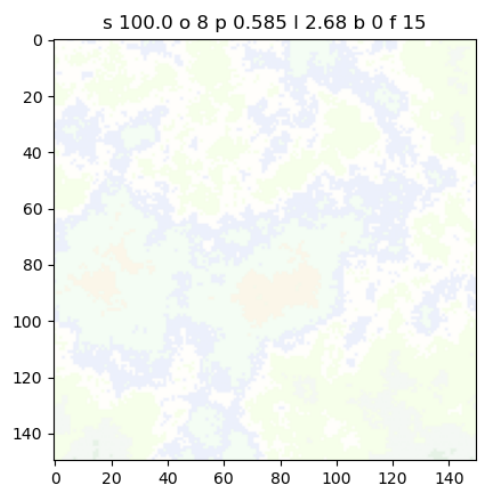


Figure 27: *Image of the RGBA world with the example usage for the Advance Case.*

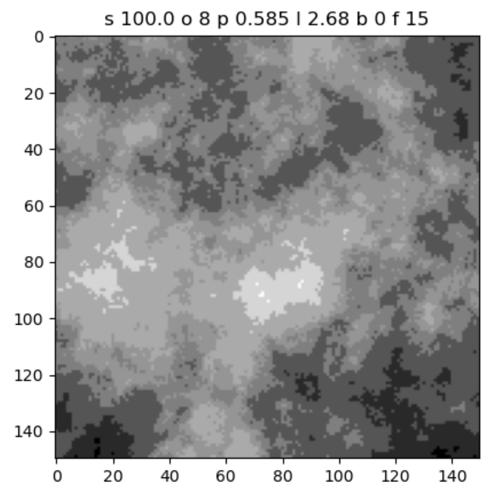


Figure 28: *Image of the RGBA world with the example usage for the Advance Case.*

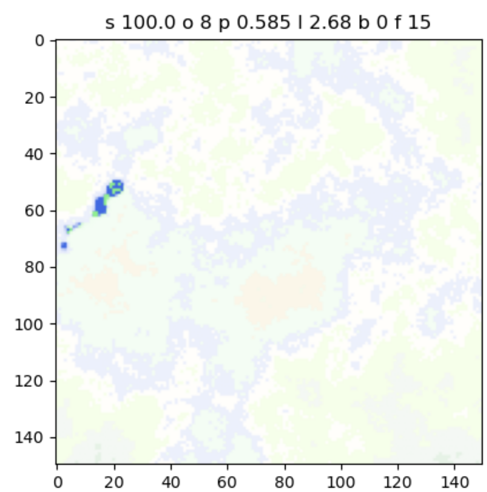


Figure 29: *Image of the final pathing used in the example usage for the Advance Case.*

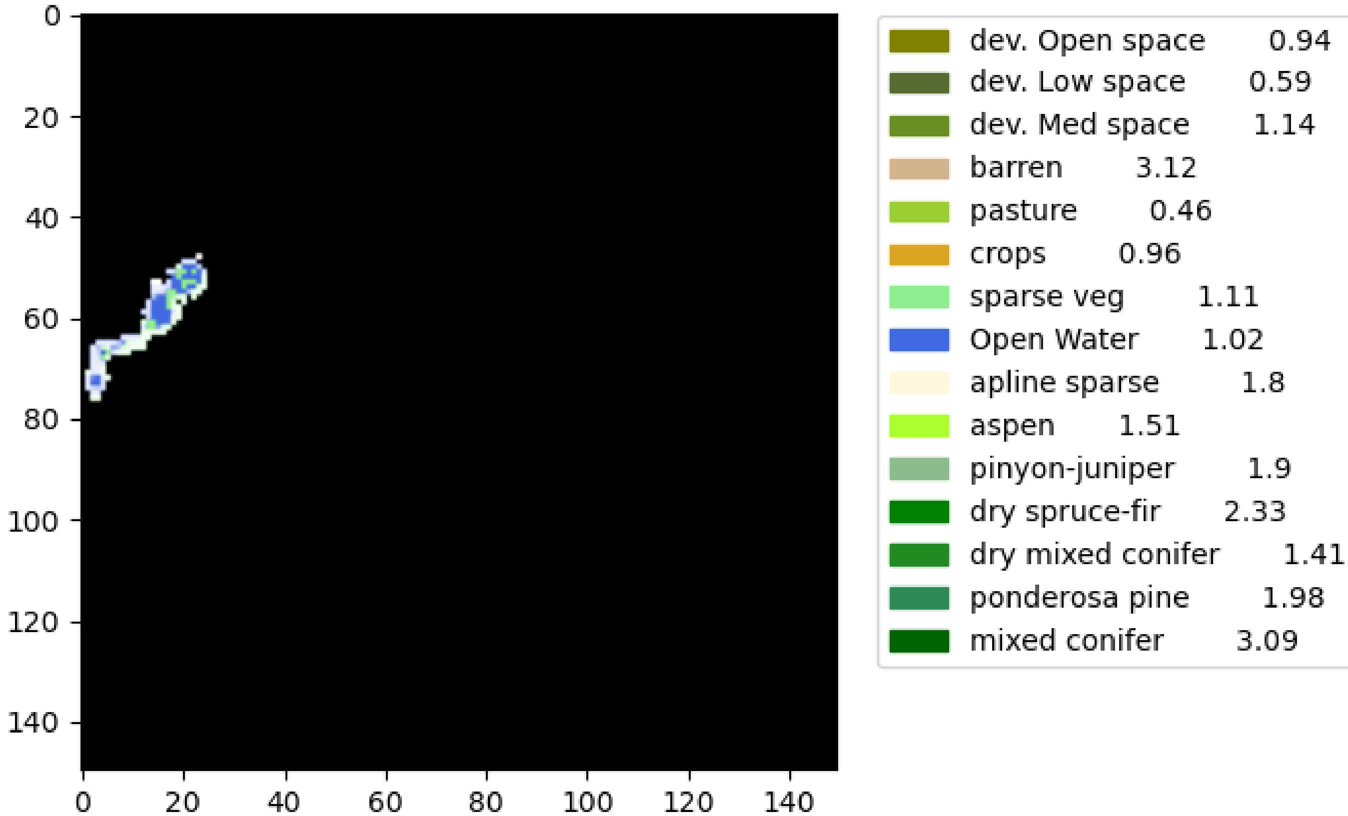
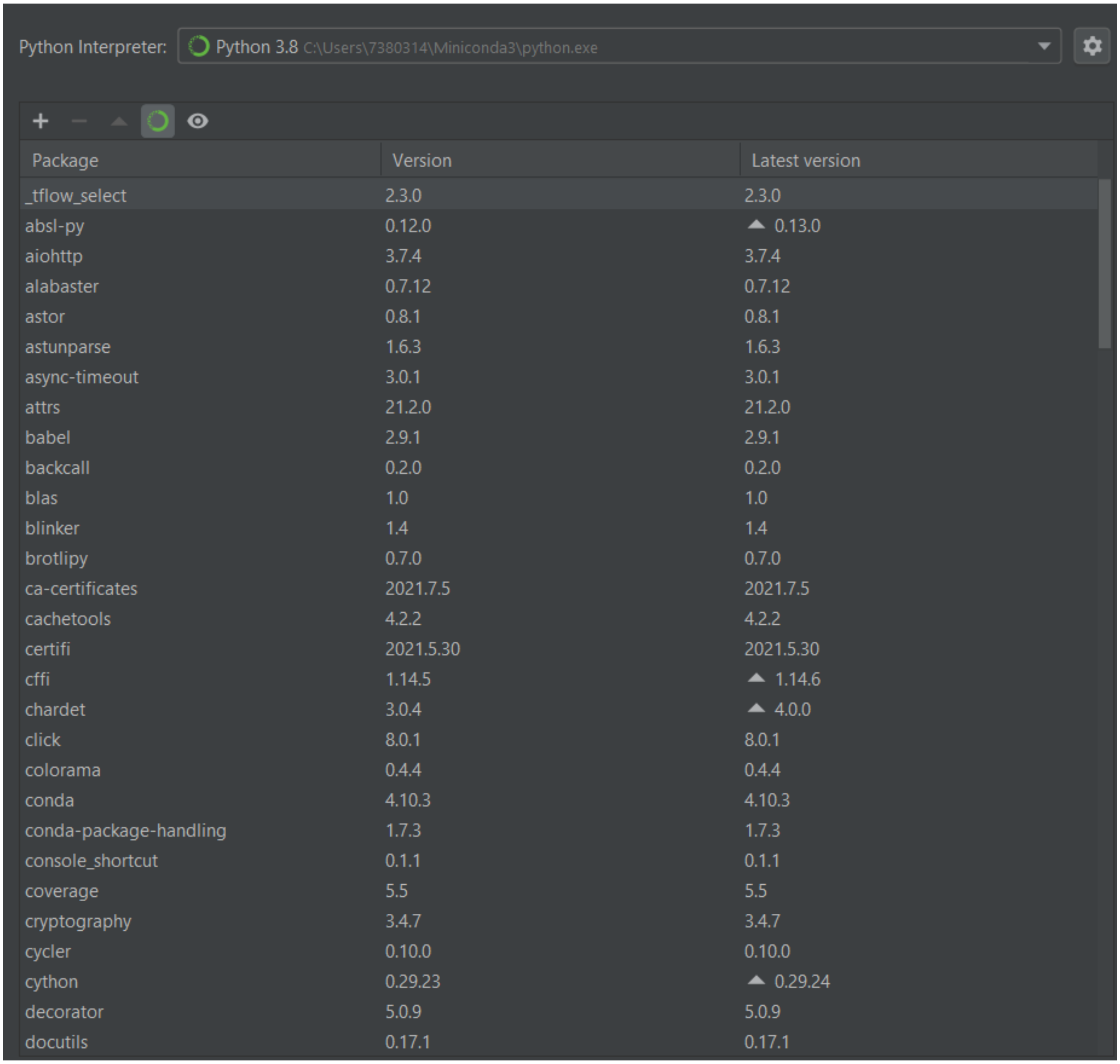


Figure 30: *Image of the path in better contrast used in the example usage for the Advance Case.*

Appendices

A Library Version History



Python Interpreter: Python 3.8 C:\Users\7380314\Miniconda3\python.exe

Package	Version	Latest version
_tflow_select	2.3.0	2.3.0
absl-py	0.12.0	▲ 0.13.0
aiohttp	3.7.4	3.7.4
alabaster	0.7.12	0.7.12
astor	0.8.1	0.8.1
astunparse	1.6.3	1.6.3
async-timeout	3.0.1	3.0.1
attrs	21.2.0	21.2.0
babel	2.9.1	2.9.1
backcall	0.2.0	0.2.0
blas	1.0	1.0
blinker	1.4	1.4
brotlipy	0.7.0	0.7.0
ca-certificates	2021.7.5	2021.7.5
cachetools	4.2.2	4.2.2
certifi	2021.5.30	2021.5.30
cffi	1.14.5	▲ 1.14.6
chardet	3.0.4	▲ 4.0.0
click	8.0.1	8.0.1
colorama	0.4.4	0.4.4
conda	4.10.3	4.10.3
conda-package-handling	1.7.3	1.7.3
console_shortcut	0.1.1	0.1.1
coverage	5.5	5.5
cryptography	3.4.7	3.4.7
cycler	0.10.0	0.10.0
cython	0.29.23	▲ 0.29.24
decorator	5.0.9	5.0.9
docutils	0.17.1	0.17.1

Package	Version
et_xmlfile	1.1.0
ffmpeg	4.2.2
freetype	2.10.4
gast	0.4.0
google-auth	1.31.0
google-auth-oauthlib	0.4.2
google-pasta	0.2.0
grpcio	1.36.1
h5py	2.10.0
hdf5	1.10.4
icc_rt	2019.0.0
icu	58.2
idna	2.10
imagesize	1.2.0
importlib-metadata	3.10.0
intel-openmp	2021.2.0
ipython	7.22.0
ipython_genutils	0.2.0
jdcal	1.4.1
jedi	0.17.0
jinja2	3.0.1
jpeg	9b
keras-applications	1.0.8
keras-preprocessing	1.1.2
kiwisolver	1.3.1
libpng	1.6.37
libprotobuf	3.14.0
libtiff	4.2.0
llvmlite	0.36.0

Package	Version
lz4-c	1.9.3
markdown	3.3.4
markupsafe	2.0.1
matplotlib	3.3.4
matplotlib-base	3.3.4
menuinst	1.4.16
mkl	2021.2.0
mkl-service	2.3.0
mkl_fft	1.3.0
mkl_random	1.2.1
multidict	5.1.0
noise	1.2.2
numba	0.53.1
numpy	1.20.2
numpy-base	1.20.2
oauthlib	3.1.0
olefile	0.46
openpyxl	3.0.7
openssl	1.1.1k
opt_einsum	3.3.0
packaging	21.0
pandas	1.2.5
parso	0.8.2
pickleshare	0.7.5
pillow	8.2.0
pip	21.1.2
powershell_shortcut	0.0.1
prompt-toolkit	3.0.17
protobuf	3.14.0

Package	Version
pyasn1	0.4.8
pyasn1-modules	0.2.8
pycosat	0.6.3
pycparser	2.20
pygments	2.9.0
pyjwt	2.1.0
pyopenssl	20.0.1
pyparsing	2.4.7
pyqt	5.9.2
pyreadline	2.1
pysocks	1.7.1
python	3.8.3
python-dateutil	2.8.1
pytz	2021.1
pywin32	227
qt	5.9.7
requests	2.25.1
requests-oauthlib	1.3.0
rsa	4.7.2
ruamel_yaml	0.15.100
scipy	1.6.2
setuptools	57.0.0
sip	4.19.13
six	1.16.0
snowballstemmer	2.1.0
sphinx	4.0.2
sphinxcontrib-applehelp	1.0.2
sphinxcontrib-devhelp	1.0.2
sphinxcontrib-htmlhelp	2.0.0

Package	Version
sphinxcontrib-jsmath	1.0.1
sphinxcontrib-qthelp	1.0.3
sphinxcontrib-serializinghtml	1.1.5
sqlite	3.35.4
tensorboard	2.4.0
tensorboard-plugin-wit	1.6.0
tensorflow	2.3.0
tensorflow-base	2.3.0
tensorflow-estimator	2.5.0
termcolor	1.1.0
tk	8.6.10
tornado	6.1
tqdm	4.59.0
traitlets	5.0.5
typing-extensions	3.7.4.3
typing_extensions	3.7.4.3
urllib3	1.26.4
vc	14.2
vs2015_runtime	14.27.29016
wcwidth	0.2.5
werkzeug	1.0.1
wheel	0.36.2
win_inet_pton	1.1.0
wincertstore	0.2
wrapt	1.12.1
xlrd	2.0.1
xz	5.2.5
yaml	0.2.5
yarl	1.6.3

zipp	3.4.1
zlib	1.2.11
zstd	1.4.9

B Source Code

B.1 CaDeerMotility testing

```python
import timeit
import numpy as np
from CaDeerMotility import CaDeer


def main():
    default_case()
    advance_case()
    hacking()


def default_case():
    # class initialization
    deer = CaDeer()

    # class function call, gather features
    deer.gather_features("test_output")

    # create the world, default size is 250 by 250
    deer.create_world()

    # optional call to output the world in black and white, while using a class variable
    deer.output_world(deer.world, True)

    # creates the color version of the world
    deer.color_world()

    # outputting the world in color after applying the color range values
    deer.output_world(deer.world_color)

    # outputting the grayscale version of the color world
    deer.output_world(deer.ca_world, True)

    # run the path finding algorithm
    deer.pathing(10000)

    print("Done, with Default Case")


def advance_case():
    # size of the world
    x, y = 150, 150

    # scaling of the world
    scale = 100.0

    # number of octaves
    octaves = 8

```

```python
    # creating and initializing the Cellar Autonoma of deer
    deer = CaDeer(scale=scale, octaves=octaves, persistence=0.585, lacunarity=2.68, base=0, features=15)

    # collect the features from Excel
    deer.gather_features("test_output", light_mode=False, color_range=None, input_excel_name="test_input.xlsx",
                         colors=None, motility_values=None, terrain_names=None)

    # creating the world given the sizes and feature list
    deer.create_world(length=x, width=y)

    # outputting the perlin noise world
    deer.output_world(world=deer.world, gray=True)

    # creates the color version of the world
    deer.color_world()

    # outputting the world in color after applying the color range values
    deer.output_world(world=deer.world_color, gray=False)

    # outputting the grayscale version of the color world
    deer.output_world(world=deer.ca_world, gray=True)

    # remove comments to specify a starting point, x and y are opposite of the built in cursor within the plot output
    # deer.starting_pos_x = 71
    # deer.starting_pos_y = 24

    # showing the CA pathing of the deer showing Live update and recording the path taken
    deer.pathing(time=100000, live_update=False, encoding=None, mpfour_output=None)

    print("Done with Advance Case")


def hacking():
    # size of the world
    x, y = 150, 150

    # scaling of the world
    scale = 100.0

    # number of octaves
    octaves = 8

    # trying w/ 15 features
    names = ['Open space', 'Low Space', 'Med Space', 'barren', 'pasture', 'crops', 'sparse veg', 'open water',
             'apline sparse', 'aspen', 'juniper', 'dry spruce', 'dry mixed', 'pine', 'mixed conifer']

    # 15 feature motility values
    motility_values = np.array(
        [0.94, 0.59, 1.14, 3.12, 0.46, 0.96, 1.11, 1.02, 1.80, 1.51, 1.90, 2.33, 1.41, 1.98, 3.09])

    # 15 feature color set
```

```python
    colors = np.array([[128, 128, 0], [85, 107, 47], [107, 142, 35], [210, 180, 140], [154, 205, 50], [218, 165, 32],
                       [144, 238, 144], [65, 105, 225], [255, 248, 220], [173, 255, 47], [143, 188, 143], [0, 128, 0],
                       [34, 139, 34], [46, 139, 87], [0, 100, 0]])

    # 15 feature color range
    color_range = np.array([-0.867, -0.733, -0.6, -0.467, -0.333, -0.2, -0.067, 0, 0.067, 0.2, 0.333, 0.467, 0.733,
                            0.867, 1])

    # creating and initializing the Cellar Autonoma of deer
    deer = CaDeer(scale=scale, octaves=octaves, persistence=0.585, lacunarity=2.68, base=0, features=15)

    # collect the features without the use of an excel file, because we know how to code
    deer.gather_features("test_output", light_mode=False, color_range=color_range,
                         colors=colors, motility_values=motility_values, terrain_names=names)

    # user could save or create a world to use by setting the following comment with a correct ndArray (you fill in)
    # deer.world = (some ndArray)
    # self.x = (length of ndArray)
    # self.y = (length of ndArray)
    # and comment out the deer.create_world()
    # creating the world given the sizes and feature list
    deer.create_world(length=x, width=y)

    # outputting the perlin noise world
    deer.output_world(world=deer.world, gray=True)

    # creates the color version of the world
    deer.color_world()

    # outputting the world in color after applying the color range values
    deer.output_world(world=deer.world_color, gray=False)

    # outputting the grayscale version of the color world
    deer.output_world(world=deer.ca_world, gray=True)

    # specify a starting point, x and y are opposite of the built in cursor within the plot output
    deer.starting_pos_x = 71
    deer.starting_pos_y = 24

    # showing the CA pathing of the deer showing Live update and recording the path taken
    deer.pathing(time=20000, live_update=False, encoding=None, mpfour_output="test_output_two")

    print("Done")


if __name__ == "__main__":
    main()
```

B.2 CaDeerMotility

```python
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import noise
from matplotlib import cm
import matplotlib.patches as mpatches
import matplotlib.animation as animation


class CaDeer(object):
    """This is a Cellular Automata Class that is be used to check motility values of deer. The use of Perlin Noise
        allows for repeatable, but random enough generated worlds to check a minimum of 5 motility values in the form of
        features within the Perlin Noise.
        :class:`CaDeer`

        :param scale: Controls the viewing scale of the generated Perlin Noise. Default is 100.0
        :type scale: float, optional
        :param octaves: Number of features that will appear within the distinct float steps generated within
        the Perlin Noise. Default is 6.
        :type octaves: int, optional
        :param persistence: Number of repeating characteristics within the Perlin Noise. Default is None, which will be
        changed to a random float between (0.2, 0.6).
        :type persistence: float, optional
        :param lacunarity: Level of detail found within the Perlin Noise. Default is None, which will be changed to a
        random float between (2.2, 3).
        :type lacunarity: float, optional
        :param base: Determines starting point of the map. Default is None, which will be changed to an integer between
        (2, 25).
        :type base: int, optional
        :param features: Number of features to color and use in model. Must match the number of elements within the
        colors, color_range, feature_list, and names arrays. Default is 5, and will be used for all other functions
        to run the cellular automata.
        :type features: int, optional
    """

    def __init__(self, scale=100.0, octaves=6, persistence=None, lacunarity=None, base=None, features=5):
        """
        Constructor method
        """

        if features > 5:
            self.features = features
        else:
            print("Features have been set to default, please enter a number of features greater than 5")
            self.features = 5

```

```
        self.scale = scale
        self.octaves = octaves
        self.persistence = persistence
        self.lacunarity = lacunarity
        self.base = base
        self.starting_pos_x = None
        self.starting_pos_y = None
        # used to get corrected grayscale
        color = cm.get_cmap(’gray’)
        self.cmap = color.reversed()

    def excel_read(self, input_excel_file_name):
        """Used to gather names, motility values, colors, and color range from an excel file that is passed in by the
        user. Values are checked to make certain that there are no NaN’s which occur from having one or more of the
        columns that does not match the length of the other columns.

        :param input_excel_file_name: File path of the Excel File
        :type input_excel_file_name: string
        """

        xl = pd.ExcelFile(input_excel_file_name)
        default = False
        # get the data frame
        df = xl.parse(0, skiprows=0)
        data = df.values

        # gather names
        self.names = data[:, 0].tolist()
        if any(type(x) is float for x in self.names):
            default = True
            print("Names column does not match the length of the other or contains a float. The default values have been set.")

        # gather motility
        self.motility_values = data[:, 1]
        if np.isnan(np.sum(self.motility_values)):
            default = True
            print("Motility values column does not match the length of the other, default values have been set.")

        # gather RGB colors and
        colors = data[:, 2]
        if any(type(x) is float for x in colors):
            default = True
            print("Colors column does not match the length of the other, default values have been set.")
        else:
            colors = np.asarray([colors[i].replace(’ ’, ’’) for i in range(colors.size)])
            colors = np.asarray([np.fromstring(colors[i], dtype=int, sep=’,’) for i in range(colors.size)])
            self.colors = colors

        # get the color range
        self.color_range = data[:, 3]
        if np.isnan(np.sum(self.color_range)):
            default = True
```

```
            print("Color range column does not match the length of the other, default values have been set.")

        if default:
            self.default()

    def excel_write(self, path_taken, excel_output_name, motilities_taken, axis_position):
        """Outputs pathing data that the deer took into an excel sheet showing the terrain name and the motility
        assigned to the terrain.

            :param path_taken: Terrain path that was taken by the deer through the simulation.
            :type path_taken: List
            :param excel_output_name: Name of the Excel file the user wishes to output data from the simulation.
            :type excel_output_name: str
            :param motilities_taken: Motility values of each terrain feature that was taken by the deer throughout the
            simulation.
            :type motilities_taken: List
            :param axis_position: List of axis positions.
            :type axis_position: List
            """

        df = pd.DataFrame({'Terrain': path_taken, 'Motility': motilities_taken, 'Axis Position': axis_position})
        writer = pd.ExcelWriter(excel_output_name + '.xlsx')
        df.to_excel(writer)
        writer.save()

    def ca_setup(self):
        """Used to setup the starting position of the deer within the simulated world.
        """

        # starting values must not be on the edges of the world... for now
        if self.starting_pos_x is None:
            self.starting_pos_x = np.random.randint(1, self.length - 1)
        if self.starting_pos_y is None:
            self.starting_pos_y = np.random.randint(1, self.width - 1)

        # set current position to starting position
        self.current_pos_x = self.starting_pos_x
        self.current_pos_y = self.starting_pos_y

    def color_append(self, pathing=True):
        """Returns an array of strings that hold the values of colors used within the world and to be used by the
        matplotlib legend. User can add in a brown path by setting pathing=True.

        :param pathing: Determines if the deer path should be added to the legend list
        :type pathing: bool, optional
        :return: A string array of the colors used within the world.
        :rtype: ndArray
        """

        if pathing:
            # add color of deer
```

```python
            color = np.vstack([self.colors, [255 / 255, 0 / 255, 255 / 255, 1]])
            # delete the alpha out as we do not need it for the main blocks
            color = np.delete(color, 3, axis=1)
        else:
            color = np.delete(self.colors, 3, axis=1)
        # convert from rgb decimal to rgb hex
        colors = ['#{:02x}{:02x}{:02x}'.format(int(color[i][0] * 255), int(255 * color[i][1]), int(255 * color[i][2]))
                  for i in range(color.shape[0])]

        # return appending colors array
        return colors

    def color_world(self):
        """ Used to create the RGBA world as well as color range values of the world.
        """

        # create RGB from size of the heat map of the world
        self.world_color = np.zeros(self.world.shape + (4,))
        # loop through and create a CA model from the original world
        self.ca_world = np.zeros(self.world.shape)

        # create RGB from heat map
        for i in range(self.length):
            for j in range(self.width):
                # creates array of indices that meet the condition
                k = np.where(self.world[i][j] < self.color_range)
                # use index
                self.ca_world[i][j] = self.color_range[k[0][0]]
                self.world_color[i][j] = self.colors[k[0][0]]

    def create_world(self, length=250, width=250):
        """ Creates the Perlin Noise given user dimensions.

        :param length: Sets the length of the world
        :type length: int, optional
        :param width: Sets the width of the world
        :type width: int, optional
        """

        # number of pixels of the world for a set length
        self.length = length
        # number of pixels of the world for a set width
        self.width = width

        # check to see if we need to fill in with random values
        if self.persistence is None:
            self.persistence = np.random.uniform(0.2, 0.6)
        if self.lacunarity is None:
            self.lacunarity = np.random.uniform(2.2, 3)
        if self.base is None:
            self.base = np.random.randint(0, 25)

        # create the world array
        self.world = np.zeros((self.length, self.width))

        # use perlin noise to generate random world
        for i in range(self.length):
            for j in range(self.width):
```

```
                self.world[i][j] = noise.pnoise2(i / self.scale, j / self.scale, self.octaves, self.persistence,
                                                 self.lacunarity, self.length, self.width, self.base)

    def default(self):
        """ Provides the default values for the features, colors, color_range, motility_values, and names. Default
        settings create a world with 5 features using the barren, water, pasture, spruce, and mixed confir values.
        """

        # default of 5 features
        self.features = 5

        # default of 5 colors
        self.colors = [[240, 230, 140], [65, 105, 225], [34, 139, 34], [139, 137, 137], [255, 250, 250]]

        # default of 5 color ranges
        self.color_range = [-0.05, 0, 0.2, 0.36, 1]

        # default of 5 motility values
        self.motility_values = [0.94, 1.02, 0.46, 2.33, 3.12]

        # default of 5 names
        self.names = ['barren', 'water', 'pasture', 'spruce', 'mixed confir']

    def gather_features(self, output_excel_name, light_mode=False, input_excel_name=None, color_range=None, colors=None,
                        motility_values=None, terrain_names=None):
        """ This function gathers all needed values used for coloring in the world as well as running the simulation.
            User must provide the name of the output name of the excel file. All other choices must match the number of
            features that was used in class creation.

            :param output_excel_name: Used to name the output excel file name.
            :type output_excel_name: string
            :param light_mode: Used to determine if the RGBA will either get less transparent or more transparent.
            :type light_mode: bool, optional
            :param input_excel_name: File path/name of the excel file that provides the color_range, colors,
            motility_values, and terrain names. Must match the number of features for each variable.
            :type input_excel_name: string, optional
            :param color_range: Holds the values to determine cutoff values for RGBA. Must match the number of features.
            :type color_range: ndArray, optional
            :param colors: Holds the RGB values that the user wishes to use to color the world. Must be a ndArray
            composed of 3-int ndArrays. Example ndArray([[0,0,0], [255,255,255]]). Must match the number of
            features and ints must be range from 0-255 not 0-1.
            :type colors: ndArray, optional
            :param motility_values: Holds the motility values of each terrain type, which will be used when the
            simulated deer makes movement choices. Must match number of features
```

```
            :type motility_values: ndArray, optional
            :param terrain_names: Holds the names of the terrain that are used within the simulation. Must match the
            number of features.
            :type terrain_names: list, optional
            """

        # save string name for later
        self.output_excel_name = output_excel_name

        # save for later functions choices
        self.light_mode = light_mode

        if input_excel_name is not None:
            self.excel_read(input_excel_name)
        else:

            if color_range is None:
                self.default()
            else:
                self.color_range = color_range
                if len(self.color_range) != self.features:
                    print("Default values of 5 features has been selected. "
                          "Please enter number of color range that matches the number of features.")
                    self.default()

            if colors is None:
                self.default()
            else:
                self.colors = colors
                if self.colors.shape[0] != self.features:
                    print("Default values of 5 features has been selected. "
                          "PLease enter number of colors that matches the number of features.")
                    self.default()

            if motility_values is None:
                self.default()
            else:
                self.motility_values = motility_values
                if self.motility_values.size != self.features:
                    print("Motility values have been set to default. Please enter motility values that match the "
                          "number of features.")
                    self.default()

            if terrain_names is None:
                self.default()
            else:
                self.names = terrain_names
                if len(self.names) != self.features:
                    print("Default values have been set. Please enter an equal amount of names to features.")
                    self.default()

        self.rgb_to_rgba()
        self.create_dictionary()

```

```python
    def rgb_to_rgba(self):
        """ Used to turn the user inputted RGB values, between 0-255 into RGBA values between 0-1.
        """

        colors = np.divide(self.colors, 255)

        if self.light_mode:
            apparent_value = np.ones([self.features, 1])
        else:
            apparent_value = np.add(np.zeros([self.features, 1]), 0.1)

        # add either 1's or 0's to the color array to make it RGBA
        self.colors = np.hstack((colors, apparent_value))

    def create_dictionary(self):
        """ Used to create dictionaries to increase search time within the simulation.
        """

        # create dictionary to speed up the program, 1500^2 resulted in a 77% speedup
        dict_index = dict(zip(self.color_range, self.motility_values))
        self.motility_dictionary = {float(key): dict_index[key] for key in dict_index}

        dict_index = dict(zip(self.color_range, self.names))
        self.names_dictionary = {float(key): dict_index[key] for key in dict_index}

    def moore_neighborhood(self, square):
        """ Uses a moore neighborhood to determine which new position to move the deer based off of the average of the
        values found within the moore neighborhood. Each outer square of the moore neighborhood is checked against the
        current lowest motility value plus a random normal distribution value.

        :param square: Moore neighborhood array of floats, must be a 3x3 ndArray.
        :type square: ndArray
        """

        # default for current motility
        current_motility = 100.0
        self.next_position_y = 0
        self.next_position_x = 0

        average = np.average(square)

        for i in range(len(square)):
            for j in range(len(square[0])):
                # compare options based off being less than average, while excluding the curr_pos
                check_motility = square[i][j]

                # check to see if the new motility is less than the average
                # ignore the current position in the middle of the grid
                if not (i == 1 and j == 1):
                    # add randomness to increase movement
                    if check_motility < average + np.random.normal():
                        if check_motility < current_motility:
                            current_motility = check_motility
                            self.next_position_x = i
                            self.next_position_y = j
```

```python

        # update for the fact that position is in the middle of the grid
        self.next_position_x -= 1
        self.next_position_y -= 1

    def view_finder(self, square):
        """ This function simulates the viewing of the deer as it uses an extended moore neighborhood to help find the
        best choice of movement. Each of the moore neighborhood values will be the average of the surrounding values,
        which is then passed into the moore_neighborhood function to find the next position.

        :param square: Extended moore neighborhood ndArray of floats, must be 7 by 7.
        :type square: ndArray

        """

        # Python memory hack making me do this :'/
        square = np.copy(square)

        # convert from the heat map values to the respective motility values
        for i in range(len(square[0])):
            for j in range(len(square[0])):
                square[i][j] = self.motility_dictionary[square[i][j]]

        # front view
        front_view = np.sum(square[0:2, 0:7])
        square[2][3] = front_view / 14

        # front left view
        front_left_view = np.sum(square[0, 0:5]) + np.sum(square[1, 0:4]) + np.sum(square[2, 0:2]) + np.sum(
            square[3, 0:2]) + square[4][0]
        square[2][2] = front_left_view / 14

        # left view
        left_view = np.sum(square[0:7, 0:2])
        square[3][2] = left_view / 14

        # front right view
        front_right_view = np.sum(square[0, 2:7]) + np.sum(square[1, 3:7]) + np.sum(square[2, 5:7]) + np.sum(
            square[3, 5:7]) + square[4][6]
        square[2][4] = front_right_view / 14

        # right view
        right_view = np.sum(square[0:7, 5:7])
        square[3][4] = right_view / 14

        # back right view
        back_right_view = np.sum(square[6, 2:7]) + np.sum(square[5, 3:7]) + np.sum(square[4, 5:7]) + np.sum(
            square[3, 5:7]) + square[2][6]
        square[4][4] = back_right_view / 14

        # back left view
        back_left_view = np.sum(square[6, 0:5]) + np.sum(square[5, 0:4]) + np.sum(square[4, 0:2]) + np.sum(
```

```
            square[3, 0:2]) + square[2][0]
        square[4][2] = back_left_view / 14

        # back view
        back_view = np.sum(square[5:7, 0:7])
        square[4][3] = back_view / 14

        # used to move
        movement = square[2:5, 2:5]

        self.moore_neighborhood(movement)

    def live_updater(self, buffer, t, colors, motility, prev_pos_x, prev_pos_y):
        """ Provides the ability to update the matplotlib output given the current position of the deer. Calls the
        alpha change function to update the current value of the RGBA pixel position.

        :param buffer: Copy of the RGBA world used to show current position of the deer.
        :type buffer: ndArray
        :param t: Current time or iteration of the simulation.
        :type t: int
        :param colors: Color array with the deer being added into the simulation.
        :type colors: ndArray
        :param motility: Motility values being used within the world.
        :type motility: ndArray
        :param prev_pos_x: Previous position of the deer on the x-axis
        :type prev_pos_x: int
        :param prev_pos_y: Previous position of the deer on the y-axis
        :type prev_pos_y: int

        """
        # needed for video
        plt.clf()

        if self.current_pos_x == self.width * -1 - 1:
            self.current_pos_x = np.remainder(self.current_pos_x, self.width)

        if self.current_pos_y == self.length * -1 - 1:
            self.current_pos_y = np.remainder(self.current_pos_y, self.length)

        # buffer
        buffer[prev_pos_x][prev_pos_y] = self.world_color[prev_pos_x][prev_pos_y]

        # use pink for current position [255, 0, 255]
        buffer[self.current_pos_x][self.current_pos_y] = [255 / 255, 0 / 255, 255 / 255, 1]

        # add in location and iteration/time passed
        string = "t {} \n (x,y): ({},{})\n s {} o {} p {} l {} b {} f {}".format(t,
                                                                                   np.
remainder(self.current_pos_x,

        self.width),
                                                                                   np.
remainder(self.current_pos_y,

        self.length), self.scale,
                                                                                   self.
```

```
octaves,
                                                                                     np.round(self.persistence, 3),
                                                                                     np.round(self.lacunarity, 3),
                                                                                     self.base, self.features
                                                                                     )

        # plot the updated time/iteration and current coordinates of the deer
        plt.title(string)

        # plot the buffer world image
        plt.imshow(buffer)

        # create the legend box with names of each color as well the as the motility values
        patches = [mpatches.Patch(color=colors[i], label='{:^5} {:>10}'.format(self.names[i], motility[i])) for i in
                   range(len(self.names))]

        # plot the legend box
        plt.legend(handles=patches, bbox_to_anchor=(1.05, 0.0, 0.3, 1), loc=2, borderaxespad=0.1)  # , mode='expand')

        # show the image
        plt.show()

        # use to determine the time between each iteration
        plt.pause(0.3)

    def pathing(self, time, live_update=False, encoding=None, mpfour_output=None):
        """ Given a set amount of iterations this function simulates the deer within the generated world. The use of
        live_update allows the user to see the deer move after each iteration. User can output a mp4 video given a
        name/address of the save file. Use encoding for faster performance assuming the user's computer allows for
        matplotlib encoding types found here https://ffmpeg.org/doxygen/2.4/group__lavc__encoding.html

        :param time: Total amount of iterations to run the simulation
        :type time: int
        :param live_update: Determines if the simulation should be shown live
        :type live_update: bool, optional
        :param encoding: Determines if the user wants to encode the mp4 with a specific set of encoding instructions.
        :type encoding: string, optional
        :param mpfour_output: Determines name and address of mp4 output
        :type mpfour_output: string, optional
        """

        # get starting positions
        self.ca_setup()

        # stop user from slowing the process of mp4 output
        if mpfour_output is not None:
            live_update = False

```

```
        # make a copy of the world
        buffer = np.copy(self.world_color)

        # holds the x,y coordinates of the path taken by the deer
        path_taken = []

        # clear up strings by adding path and deer
        motility = self.string_names()

        # clear up colors by adding path and deer
        colors = self.color_append()

        # set the next position to 0
        self.next_position_x = 0
        self.next_position_y = 0

        # previous position
        prev_pos_x = self.current_pos_x
        prev_pos_y = self.current_pos_y

        # used to start the video format
        if live_update:
            plt.show()
            plt.ion()
            plt.figure()

        if mpfour_output is not None:
            self.ims = []
            writer = animation.FFMpegWriter(fps=30, codec=encoding)
            fig = plt.figure()

        # loop through the iterations known as time
        for t in range(time):
            # output the deer on the colored world
            if live_update:
                self.live_updater(buffer=buffer, t=t, colors=colors, motility=motility,
                                  prev_pos_x=prev_pos_x, prev_pos_y=prev_pos_y)

            if mpfour_output is not None:
                self.mp4(buffer=buffer, t=t, colors=colors, motility=motility,
                         prev_pos_x=prev_pos_x, prev_pos_y=prev_pos_y)

            # add previous position of the deer to a list
            path_taken.append([prev_pos_x, prev_pos_y])

            # update the RGBA world with current position of the world
            self.world_color[prev_pos_x][prev_pos_y] = self.alpha_change(self.world_color[prev_pos_x][prev_pos_y])

            # find the square that the deer is considering based off of the current position
            square_choice = self.edge_check(x=self.current_pos_x, y=self.current_pos_y)

            # use Moore neighborhood to select the next position
            self.view_finder(square_choice)

            # previous position
```

```python
            prev_pos_x = self.current_pos_x
            prev_pos_y = self.current_pos_y

            # update current position to future position
            self.current_pos_x += self.next_position_x
            self.current_pos_y += self.next_position_y

            self.current_pos_x = np.remainder(self.current_pos_x, self.length)
            self.current_pos_y = np.remainder(self.current_pos_y, self.width)
            print("\rPathing: {:.2f} ".format(t / time * 100), end="")

        print("\rPathing: 100%")

        if live_update:
            # used to end the video format
            plt.ioff()
            plt.close()

        if mpfour_output is not None:
            ani = animation.ArtistAnimation(fig, self.ims, interval=500, blit=True)
            ani.save(mpfour_output + '.mp4', writer=writer)

        self.output_world(self.world_color)
        self.path_map()

        terrain_path = []
        motilities_taken = []

        for i in range(len(path_taken)):
            x = path_taken[i][0]
            y = path_taken[i][1]
            terrain_path.append(self.names_dictionary[self.ca_world[x][y]])
            motilities_taken.append(self.motility_dictionary[self.ca_world[x][y]])

        self.excel_write(path_taken=terrain_path, excel_output_name=self.output_excel_name,
                         motilities_taken=motilities_taken, axis_position=path_taken)

    def string_names(self):
        """ Appends the deer to the name array and returns a list of strings of the motility values.

        :return motility: List of strings of the motility values
        :rtype motility: list

        """

        # add in the path and deer to the name list
        self.names += ['deer']

        # converts to a list of strings for motility values
        motility = list(map(str, self.motility_values))

        # add in NA for deer and path, as they do not have their own values on the value map
        motility += ['NA']

        # return the motility string
        return motility
```

```
    def output_world(self, world, gray=False):
        """ Outputs the generated world with the option of having it in a gray scale.

        :param world: Array of RGBA values that represent the world
        :type world: ndArray
        :param gray: Determines if the world should be shown in gray scale.
        :type gray: bool, optional

        """

        plt.figure()

        # add in location and iteration/time passed
        string = "s {} o {} p {} l {} b {} f {}".format(self.scale, self.octaves, np.round(self.persistence, 3),
                                                        np.round(self.lacunarity, 3), self.base, self.features)

        # plot the updated time/iteration and current coordinates of the deer
        plt.title(string)

        if gray is True:
            print("Grayscale")
            plt.imshow(world, cmap=self.cmap)
        else:
            print("RGBA")
            plt.imshow(world)
        plt.show()

    def alpha_change(self, pixel):
        """ Changes the current pixel by increasing or decreasing the alpha value depending if light mode has been
        activated. Returns the changed pixel value

        :param pixel: Current pixel that needs to be adjusted as a ndArray with 4 float elements
        :type pixel: ndArray
        :return rgba: Updated pixel value as a ndArray with 4 float elements
        :rtype rgba: ndArray

        """

        # check light mode
        if self.light_mode:
            rgba = [pixel[0], pixel[1], pixel[2], 0.95 * pixel[3]]
        else:
            if pixel[3] <= 0.95:
                rgba = [pixel[0], pixel[1], pixel[2], 1.05 * pixel[3]]
            else:
                rgba = [pixel[0], pixel[1], pixel[2], 1]

        return rgba

    def path_map(self):
        """ Shows the path taken by the simulated deer to the user.
        """

        self.color_append(False)
```

```python

        for i in range(self.length):
            for j in range(self.width):

                change_apperent = self.world_color[i][j]

                if self.light_mode:
                    if change_apperent[3] == 1:
                        self.world_color[i][j] = [0, 0, 0, 1]
                else:
                    if change_apperent[3] <= 0.1:
                        self.world_color[i][j] = [0, 0, 0, 1]

        plt.figure()

        colors = self.color_append(False)

        # create the legend box with names of each color as well the as the motility values
        patches = [mpatches.Patch(color=colors[i], label='{:^5} {:>10}'.format(self.names[i], self.motility_values[i]))
                   for i in
                   range(len(self.colors))]

        # plot the legend box
        plt.legend(handles=patches, bbox_to_anchor=(1.05, 0.0, 0.3, 1), loc=2, borderaxespad=0.1)  # , mode='expand')

        plt.imshow(self.world_color)

        plt.show()

    def edge_check(self, x, y):
        """ Determines which values of the world to use within the 7 by 7 viewing array given the current x and y
        positions of the deer.

        :param x: Current x position of the deer
        :type x: int
        :param y: Current y position of the deer
        :type y: Current y position of the deer
        :return square: 7 by 7 extended moore neighborhood with the current deer position index at the middle of the
        array
        :rtype square: ndArray

        """

        # check to see if indices are outside of the array
        if x + 4 > self.length and y + 4 > self.width:
            square = self.bottom_right_corner(x, y)
        elif x - 3 < 0 and y - 3 < 0:
            square = self.upper_left_corner(x, y)
        elif x + 4 > self.length and y - 3 < 0:
            square = self.bottom_left_corner(x, y)
        elif x - 3 < 0 and y + 4 > self.width:
            square = self.upper_right_corner(x, y)
        elif x + 4 > self.length:
            square = self.bottom(x, y)
```

```python
        elif x - 3 < 0:
            square = self.upper(x, y)
        elif y + 4 > self.width:
            square = self.right(x, y)
        elif y - 3 < 0:
            square = self.left(x, y)
        else:
            square = self.ca_world[x - 3:x + 4, y - 3:y + 4]

        return square

    def right(self, x, y):
        """ Finds the overlap given the current position is on the right side of the world array. Returns the 7 by 7
        array used by the moore neighborhood.

        :param x: Current x position of the deer
        :type x: int
        :param y: Current y position of the deer
        :type y: int
        :return : 7 by 7 extended moore neighborhood with the current deer position index at the middle of the
        array
        :rtype : ndArray

        """
        # get the left half
        left_half = self.ca_world[x - 3:x + 4, y - 3:self.length]
        # get the right half
        right_half = self.ca_world[x - 3:x + 4, 0:np.remainder(y + 4, self.length)]
        # return the total
        return np.hstack((left_half, right_half))

    def left(self, x, y):
        """ Finds the overlap given the current position is on the left side of the world array. Returns the 7 by 7
        array used by the moore neighborhood.

        :param x: Current x position of the deer
        :type x: int
        :param y: Current y position of the deer
        :type y: int
        :return : 7 by 7 extended moore neighborhood with the current deer position index at the middle of the
        array
        :rtype : ndArray
        """

        # get the right half
        right_half = self.ca_world[x - 3:x + 4, 0:y + 4]
        # get the left half
        left_half = self.ca_world[x - 3:x + 4, np.remainder(y - 3, self.length):self.length]
        # return the total
        return np.hstack((left_half, right_half))

    def upper(self, x, y):
        """ Finds the overlap given the current position is on the top side of the world array. Returns the 7 by 7
```

```
            array used by the moore neighborhood.

        :param x: Current x position of the deer
        :type x: int
        :param y: Current y position of the deer
        :type y: int
        :return : 7 by 7 extended moore neighborhood with the current deer position index at the middle of the
        array
        :rtype : ndArray
        """

        # get lower half
        lower_half = self.ca_world[0:x + 4, y - 3:y + 4]
        # get the upper half
        upper_half = self.ca_world[np.remainder(x - 3, self.width):self.width, y - 3:y + 4]
        # return the total
        return np.vstack((upper_half, lower_half))

    def bottom(self, x, y):
        """ Finds the overlap given the current position is on the bottom side of the world array. Returns the 7 by 7
        array used by the moore neighborhood.

        :param x: Current x position of the deer
        :type x: int
        :param y: Current y position of the deer
        :type y: int
        :return : 7 by 7 extended moore neighborhood with the current deer position index at the middle of the
        array
        :rtype : ndArray
        """

        # gather upper half
        upper_half = self.ca_world[x - 3:self.width, y - 3:y + 4]
        # gather lower half
        lower_half = self.ca_world[0:np.remainder(x + 4, self.width), y - 3:y + 4]
        # return the upper and lower
        return np.vstack((upper_half, lower_half))

    def upper_right_corner(self, x, y):
        """ Finds the overlap given the current position is on the upper right corner of the world array.
        Returns the 7 by 7 array used by the moore neighborhood.

        :param x: Current x position of the deer
        :type x: int
        :param y: Current y position of the deer
        :type y: int
        :return : 7 by 7 extended moore neighborhood with the current deer position index at the middle of the
        array
        :rtype : ndArray
        """

        # get lower right quarter
        lower_right_quarter = self.ca_world[np.remainder(x - 3, self.width):self.width
```

```
    ,
                              0:np.remainder(y + 4, self.length)]
        # get upper right quarter
        upper_right_quarter = self.ca_world[0:x + 4, 0:np.remainder(y + 4, self.length
    )]
        # right side
        right_half = np.vstack((upper_right_quarter, lower_right_quarter))
        # lower left quarter
        lower_left_quarter = self.ca_world[0:x + 4, y - 3:self.length]
        # upper left quarter
        upper_left_quarter = self.ca_world[np.remainder(x - 3, self.width):self.width,
     y - 3:self.length]
        # left side
        left_half = np.vstack((upper_left_quarter, lower_left_quarter))
        # combine the sides together
        return np.hstack((left_half, right_half))

    def bottom_left_corner(self, x, y):
        """ Finds the overlap given the current position is on the bottom left corner
    of the world array.
        Returns the 7 by 7 array used by the moore neighborhood.

        :param x: Current x position of the deer
        :type x: int
        :param y: Current y position of the deer
        :type y: int
        :return : 7 by 7 extended moore neighborhood with the current deer position
    index at the middle of the
        array
        :rtype : ndArray
        """

        # get the upper right quarter
        upper_right_quarter = self.ca_world[x - 3:self.width, 0:y + 4]
        # get the bottom right quarter
        lower_right_quarter = self.ca_world[0:np.remainder(x + 4, self.width), 0:y +
    4]
        # right half
        right_half = np.vstack((upper_right_quarter, lower_right_quarter))
        # bottom left quarter
        lower_left_quarter = self.ca_world[0:np.remainder(x + 4, self.width),
                             np.remainder(y - 3, self.length): self.length]
        # get the upper left quarter
        upper_left_quarter = self.ca_world[x - 3:self.width, np.remainder(y - 3, self.
    length):self.length]
        # create the left half
        left_half = np.vstack((upper_left_quarter, lower_left_quarter))
        # completed 7 by 7 square
        return np.hstack((left_half, right_half))

    def bottom_right_corner(self, x, y):
        """ Finds the overlap given the current position is on the bottom right corner
     of the world array.
        Returns the 7 by 7 array used by the moore neighborhood.

        :param x: Current x position of the deer
        :type x: int
        :param y: Current y position of the deer
        :type y: int
```

```
        :return : 7 by 7 extended moore neighborhood with the current deer position index at the middle of the
        array
        :rtype : ndArray
        """

        # get the upper left quarter
        upper_left_quarter = self.ca_world[x - 3:self.width, y - 3:self.length]
        # bottom left quarter
        lower_left_quarter = self.ca_world[0:np.remainder(x + 4, self.width), y - 3:self.length]
        # create the left half
        left_half = np.vstack((upper_left_quarter, lower_left_quarter))
        # get the upper right quarter
        upper_right_quarter = self.ca_world[x - 3:self.width, 0:np.remainder(y + 4, self.length)]
        # get the bottom right quarter
        lower_right_quarter = self.ca_world[0:np.remainder(x + 4, self.width), 0:np.remainder(y + 4, self.length)]
        # right half
        right_half = np.vstack((upper_right_quarter, lower_right_quarter))
        # completed 7 by 7 square
        return np.hstack((left_half, right_half))

    def upper_left_corner(self, x, y):
        """ Finds the overlap given the current position is on the upper left corner of the world array.
        Returns the 7 by 7 array used by the moore neighborhood.

        :param x: Current x position of the deer
        :type x: int
        :param y: Current y position of the deer
        :type y: int
        :return : 7 by 7 extended moore neighborhood with the current deer position index at the middle of the
        array
        :rtype : ndArray
        """

        # get lower right quarter
        lower_right_quarter = self.ca_world[0:x + 4, 0:y + 4]
        # get upper right quarter
        upper_right_quarter = self.ca_world[np.remainder(x - 3, self.width):self.width, 0:y + 4]
        # right side
        right_half = np.vstack((upper_right_quarter, lower_right_quarter))
        # lower left quarter
        lower_left_quarter = self.ca_world[0:x + 4, np.remainder(y - 3, self.length):self.length]
        # upper left quarter
        upper_left_quarter = self.ca_world[np.remainder(x - 3, self.width):self.width,
                                           np.remainder(y - 3, self.length):self.length]
        # left side
        left_half = np.vstack((upper_left_quarter, lower_left_quarter))
        # combine the sides together
        return np.hstack((left_half, right_half))

    def break_it(self):
        """ Used to test all positions of the world using the edge check function.
```

```
    Writes to a file "breakit.txt".
        """

        # open a file to check output checks
        file1 = open("breakit.txt", "w")
        for i in range(self.width):
            for j in range(self.length):
                check_grid = self.edge_check(x=i, y=j)
                file1.write("({}, {} \n)".format(i, j))
                if check_grid.shape[0] != 7:
                    file1.write("shape[0] != 7 \n")
                if check_grid.shape[1] != 7:
                    file1.write("shape[1] != 7 \n")

        file1.close()

    def mp4(self, buffer, t, colors, motility, prev_pos_x, prev_pos_y):

        if self.current_pos_x == self.width * -1 - 1:
            self.current_pos_x = np.remainder(self.current_pos_x, self.width)

        if self.current_pos_y == self.length * -1 - 1:
            self.current_pos_y = np.remainder(self.current_pos_y, self.length)

        # buffer
        buffer[prev_pos_x][prev_pos_y] = self.world_color[prev_pos_x][prev_pos_y]

        # use pink for current position [255, 0, 255]
        buffer[self.current_pos_x][self.current_pos_y] = [255 / 255, 0 / 255, 255 /
    255, 1]

        # add in location and iteration/time passed
        string = "t {} \n (x,y): ({},{})\n s {} o {} p {} l {} b {} f {}".format(t,
                                                                               np.
    remainder(self.current_pos_x,

            self.width),
                                                                               np.
    remainder(self.current_pos_y,

            self.length), self.scale,
                                                                               self.
    octaves,
                                                                               np.
    round(self.persistence, 3),
                                                                               np.
    round(self.lacunarity, 3),
                                                                               self.
    base, self.features
                                                                               )

        # plot the updated time/iteration and current coordinates of the deer
        plt.title(string)

        # create the legend box with names of each color as well the as the motility
    values
        patches = [mpatches.Patch(color=colors[i], label='{:^5} {:>10}'.format(self.
    names[i], motility[i])) for i in
                  range(len(self.names))]
```

```python

    # plot the legend box
    plt.legend(handles=patches, bbox_to_anchor=(1.05, 0.0, 0.3, 1), loc=2,
borderaxespad=0.1)  # , mode='expand')

    # plot the buffer world image
    im = plt.imshow(buffer)

    # add to image array
    self.ims.append([im])
```