Semester Project Final Report

Individual: Dieu My Nguyen

Project title: Firefly Mating Strategy Agent-Based Model

Final state of system

Two questions inspired my intent to build an agent-based model: How does a female firefly maintain contact with a male she is signaling in a noisy environment? What can male fireflies do to optimize their chances at being recognized and successfully tracked?

For this project, I've created the first prototype of the firefly mating model to look at the second question, as the first question will require more extensive theoretical work that is out of the scope of this course. The 2-D flying motion of male fireflies is modeled as a correlated random walk in which they randomly choose a turning angle within a range for their next step of movement [2][3]. I've made the current agent-based model into a full OOAD framework using core OOAD concepts. A user can simply interact with a GUI to select parameters, run the model, and view the data and simulation in real time as the model runs. From this, we can see what emergent patterns come out of tweaking various model parameters.

The GUI layout made of `ipywidgets` widgets is inspired by the canonical language to create agent-based models, NetLogo [4], and by another Jupyter notebook converting NetLogo to Python [5].

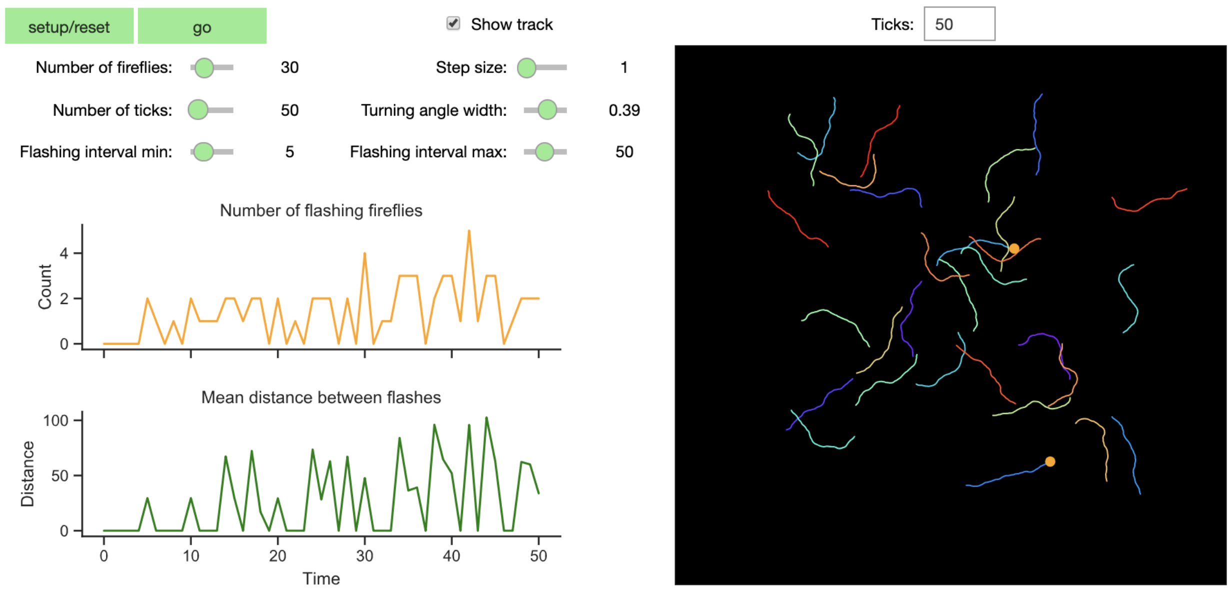
The entire code base for my model is encapsulated in four Jupyter notebooks running Python (3.6):

Firefly.ipynb: Contains the abstract base class `Firefly`, its derived class `MaleFirefly`, the factory class `MaleFireflyFactory` to generate `MaleFirefly` object, and also contains `FireflyCollection` to use the `MaleFireflyFactory` to generate and contain all male fireflies for a given simulation.

World.ipynb: Contains the class `World` that represents the entire model where fireflies behave and interact with the environment (2D space and time). This is also recording data and plotting them to update the GUI widgets. This notebook requires importing Firefly.ipynb.

GUI.ipynb: Contains the class `WidgetCollection`, in which a widget is an object of a class from the `ipywidgets` package. The widgets and the layout containing all the widgets define the GUI. This notebook also contains the class `SimulationGUI` which uses the widgets and the world to generate a NetLogo-like GUI where users can choose parameters, run the model, and watch the simulation's data plotted over time. This notebook requires importing Firefly.ipynb and World.ipynb.

Firefly Mating Strategy Agent-Based Model.ipynb: Acts as the primary user interface, with background information and references about the model and usage information to generate a GUI to run the model. This notebook requires importing World.ipynb and GUI.ipynb. The GUI looks like this and runs inside the notebook:



Show track ✓
Ticks: 50
Number of fireflies: 30
Step size: 1
Number of ticks: 50
Turning angle width: 0.39
Flashing interval min: 5
Flashing interval max: 50

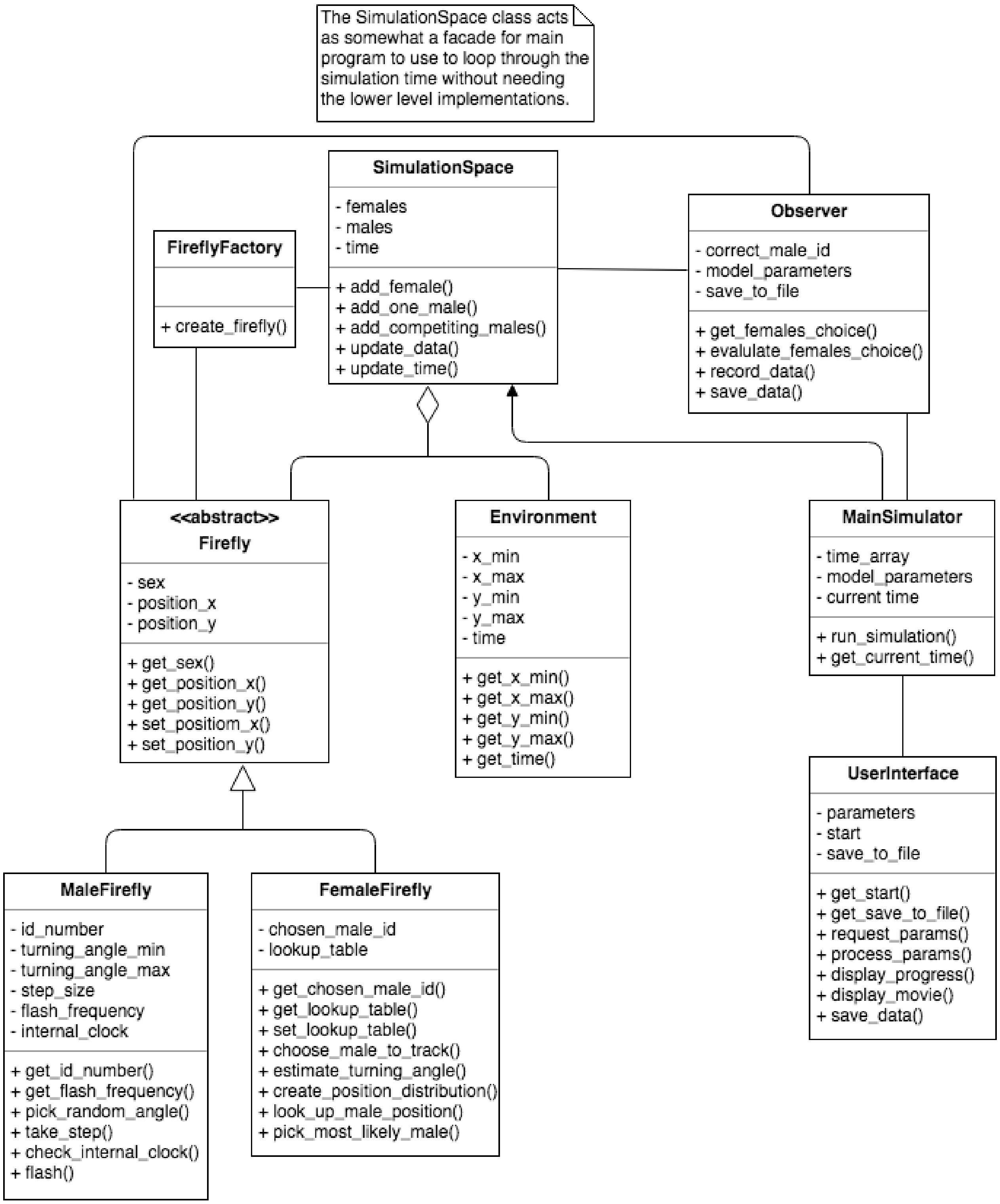
Changes from initial design

As Dr. Montgomery suggested, I kept the design within the scope of the time we had to complete this project. During the initial design, I was quite ambitious and thought I could work the entire model out. However, because some aspects of the model (e.g. the female tracking problem) require a deeper theoretical basis, I decided to focus on the object-oriented and programming aspect of my project.

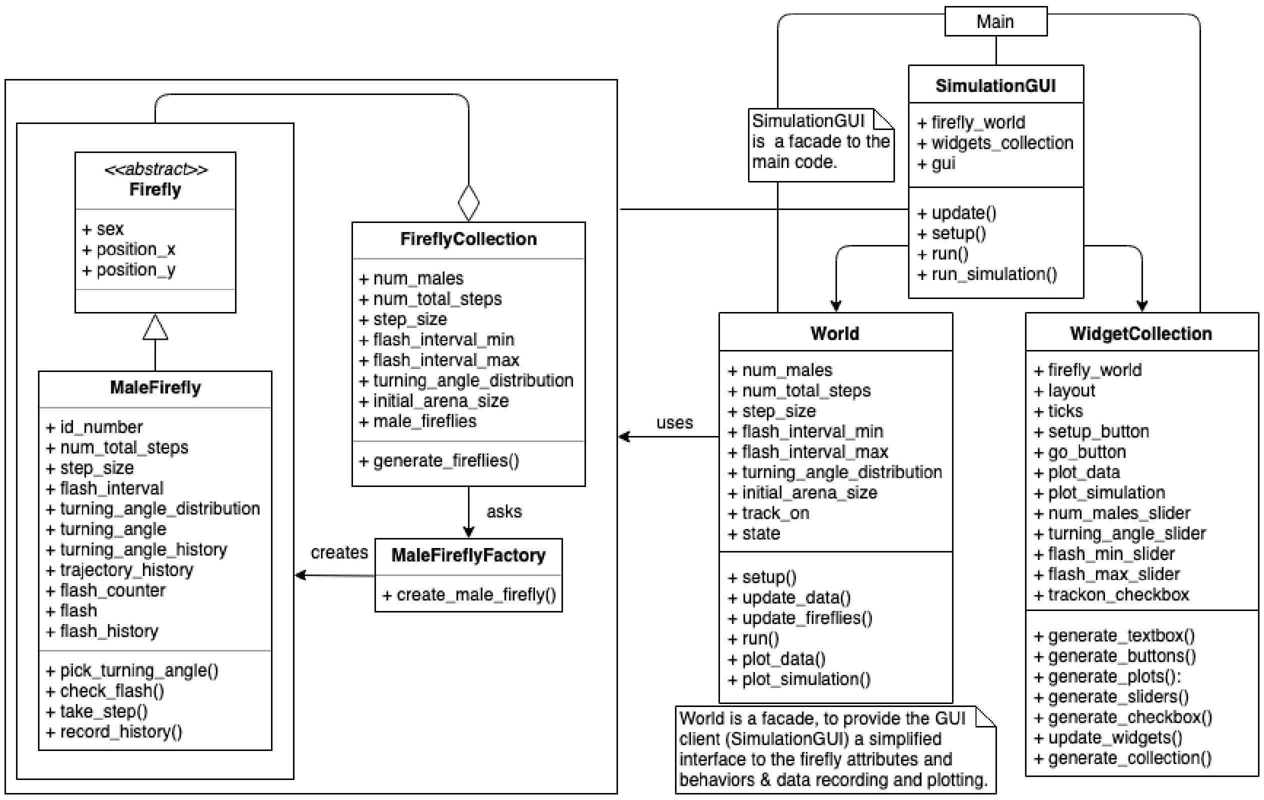
As seen below, my initial class diagram meant to depict the entire model. It was also rather messy with multiple classes that I really meant to group together (`SimulationSpace`, `Observer`, `MainSimulator`). In the final class diagram, I have a World class that contains the entire model and is used by the `SimulationGUI` to create an interactive GUI. The `Firefly` and `MaleFirefly` classes are similar, as they simply model the same attributes and methods I had thought of before. The same applies to the `MaleFireflyFactory` class and its method to create a `MaleFirefly` object. In the initial design, I did not have a way to contain all the Firefly objects created by the factory. Now I have `FireflyCollection` for that task, and it delegates to `MaleFireflyFactory` the task of generating the objects.

Originally, I did not mean to create an interactive GUI, due to not knowing how feasible it is. So I had planned to save data and videos and the user can open to view them instead. However, upon exploring the `ipywidgets` package more, I was able to create the NetLogo-like GUI for users to observe the model dynamics over time. The `WidgetCollection` class contains all the widget objects from `ipywidgets` classes. And the `SimulationGUI` uses a `WidgetCollection` object and a World object to create and run the GUI. The main program simply creates these objects and calls the GUI to display it.

Initial class diagram



Final class diagram



Please note that I decided to keep the attributes and methods public throughout the code, as in Python, there is no existence of private instance variables. Although we could use an underscore as the convention to signal to other coders to not try to access or change a variable, my code will be further developed beyond this class and most likely, users besides myself would be using the GUI or another form of user interface where all implementation details are hidden.

Design patterns in final prototype

To create a `MaleFirefly` object to put into the simulation World, I used a simple factory whose client is the `FireflyCollection`, which does not need to know the instantiation logic and only needs to pass in arguments.

The World class is a facade, to provide the GUI client (`SimulationGUI`) a simplified interface to the firefly attributes and behaviors as well as data recording and plotting.

The `SimulationGUI` class is another facade, as it hides the complicated widget and model setup, update, and run. A main program simply has to instantiate a `World` and also a `WidgetCollection` object, then pass them into the instantiation of a `SimulationGUI` object, to create a GUI right inside a Jupyter notebook.

All of these patterns and other encapsulation and abstraction concepts help my design achieve the purpose of later sharing my agent-based model with other biologists or my future self to explore different parameter sets and quickly see what patterns emerge, without needing to look through all the parts of the code.

What I learned

I learned the design and analysis and implementation are definitely iterative processes. I tried to follow my original design and class diagram, but as I began coding, I saw different and sometimes better ways to implement something, and that changes the design of the remaining aspects of the system. On the other hand, sometimes, following the original design is a wise choice. I designed certain things for a reason, and changing too much will not bring about a complete project by the deadline.

I also learned that design patterns are actually hard to implement. Going through the lecture examples of the different patterns we covered in class, I felt that each pattern really had a purpose and that I would know when to use it. However, applying patterns to my own design is not easy. The class examples first demonstrate why a piece of code is not ideal, then a pattern improves it. With my own code, I probably need more time with it, to recognize where I have cohesion and coupling issues, to be better informed where to apply certain patterns. Some simple patterns are readily available to my mind and I could see their usage from the start, such as the factory and facade patterns I described above.

Lastly, I learned that I really like the object-oriented way to life. Before this class, I was very intimidated by putting code in to classes. But now, especially with my PhD research revolving around building agent-based models, I think object-oriented is the best way to go. It makes perfect sense to abstract agents in my models into classes with attributes and behaviors, and let them interact. Also, my models are often validated as I do empirical experiments or change the theoretical basis behind a particular aspect. Object-oriented designs will help me through the maintenance headaches I have encountered. I look forward to continue using object-oriented concepts and to convert some models I have to this better programming framework.

Third-party vs Original Design/Code

Since I am designing a model for my research from scratch, I did not seek out too many existing codes as I don't imagine there are that many for firefly mating models. I made use of these common Python and Jupyter notebook packages: `abc`, `importlib`, `io`, `ipywidgets`, `ipynb`, `IPython`, `matplotlib`, `numpy`, `random`, `scipy`, `seaborn`, `sys`. One open-source Jupyter notebook I consulted to create the NetLogo-like GUI is cited and linked in [5]. This notebook taught me how to group together individual widgets to create the GUI, and how to update the data and simulation plots in real time. This notebook used an older version of `ipywidgets` that allow widgets to have individual style parameters, so I modified them to share some styling settings. My `WidgetCollection` is inspired by the code for widgets in this notebook.

References

[1] L. Buschman (2016). Field Guide to Western North American Fireflies.

[2] A. Cheung et al. (2007). Animal navigation: the difficulty of moving in a straight line. Biological Cybernetics.

[3] P.M. Kareiva and N. Shigesada (1983). Analyzing Insect Movement as a Correlated Random Walk. International Association for Ecology.

[4] U. Wilensky (1999). NetLogo (and NetLogo User Manual), Center for Connected Learning and Computer-Based Modeling, Northwestern University.

[5] D. Blank (2016). Lab 9: Ecological Models of 2016 Spring course BioCS115 Computing through Biology at Bryn Mawr College. Link