MPHYG001: Assignment 2
Bad Boids Refactoring

Padraic Calpin
PHDCA33

Code Smells

One Large File
The initial version of the code was a single file, which contained the initialisation,update and animation routines which could be run directly. My initial commit(17cbde9) defined a `Flock` class, combing the init and update routines, and the animation routines were added in commit 7dfd3d2. This modularisation also enabled adding a command line entrypoint(281e361) and packaging the program(e68acc8).

Loop Indexing & Repetition
The initial code had several smells relating to the use of loops. Firstly, loops were integer indexed, and used that integer to access corresponding elements of different arrays. The same nested loop structure was also repeated four times; twice to perform the 'move to middle' correction for x and y, and twice more to calculate the other velocity corrections.
This was resolved by combining the positions and velocities into $2 \times N \times N$ numpy arrays, and performing calculations simultaneously using broadcasting (d9e6783, d9dfcd7, 4a5811e, 90a3659).

Repeated Code The initial code, and my initial refactoring to numpy arrays, contained repeated calls for x and y corrections. This was amended in commit 90a3659.

Large Routines
The `update_boids` function, even after replacing loops with numpy broadcasting, was the bulk of the `Flock` class definition. Each component of the velocity corrections was split into a separate method, which were then called sequentially in the new `update_boids`.

Magic Numbers & Hardcoded Constants
The initial code contained multiple hardcoded constants acting as magic numbers. The new `Flock` class loads these parameters from a descriptive config file, with the initial hardcoded values loaded as defaults (15dc7ae).

Global Variables
The animation and boid initialisation employed global variables in the initial code. I also initially made some magic numbers into global variables. These were made into properties of the `Flock` class in commits 17cbde9 and 7dfd3d2

Documentation Needed

The update and animation methods were previously slightly cryptic. Turning the simulation parameters into properties of the `Flock` class(17cbde9,7dfd3d2, 15dc7ae), and breaking up the update and animation functions into sub-routines (cef9415), helped to make this code self-documenting. Additional docstrings were added to explain the config and data structures the Constructor expects.

Reinventing the wheel

Some components of the `update_boids` function and the `Flock` constructor were simplified using sum, mean and random.uniform functions provided by the numpy library (d9e6783, d9dfcd7, 4a5811e, f090cbf).

Class Structure

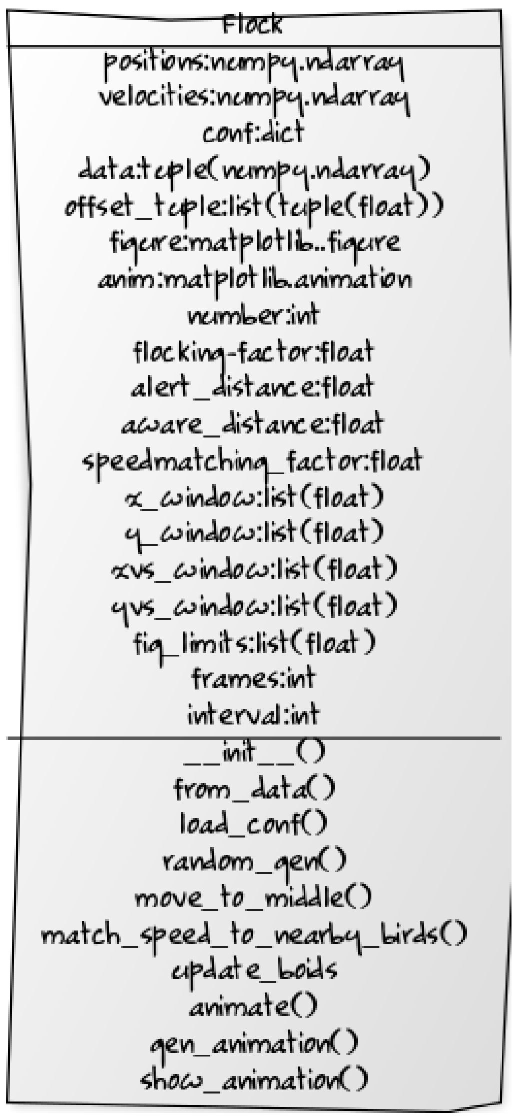

Figure 1: UML Diagram of the `Flock` class. Properties conf, data and offset_tuple are calculated properties

The Refactoring Approach

I think the refactoring approach is especially useful for quick translation of ideas into implementation, as might be required in an Agile development sprint, or when working to develop a 'Minimally Viable' version of the code. The addition of regression tests combined with Continuous Integration then allows the code to be updated safely in production, without changing the results.

One disadvantage could be that this encourages 'sloppy' design, based on the premise that it can be later refactored. There is also the possibility that regression tests allow incorrect results to propagate forward through versions, and so they cannot be used blindly; indeed, this package required a change to the regression test when refactoring altered how 'speed-matching' calculation was performed.

Problems Encountered

Already discussed above, at one point during the refactoring the regression test for `update_boids` failed. This was because of how the previous version calculated the 'speed-matching' corrections; each correction depended on the velocities of the other birds. However, as these were calculated sequentially, the corrections applied to velocities at the beginning of the list altered the calculation for birds later in the list.

When the calculation was changed to calculate all the velocity corrections simultaneously, using broadcasting, this error was corrected, but the regression test no-longer passed. This required the fixture data to be re-recorded. The change, $\mathcal{O}(0.01)$, but demonstrated the drawbacks of regression testing.

Another issue encountered was the difficulty of creating a fast, object-oriented design. In the end, I chose to optimise this code for speed, and thus there is only a single class `Flock`. A more object oriented approach might include a `Boid` class, storing the positions and velocities, which each belong to a `Flock` instance. The `Flock` could then calculate corrections by iterating over the `Boid` instances.

However, as numpy cannot create arrays of arbitrary objects, this code would be unable to take advantage of broadcasting. This could be achieved by copying the data from each boid into arrays, and then copying the updated values back into each `Boid`, but this approach is functionally the same as my `Flock` implementation.