Boids

MPHYG001 - Python Research Programming

Sofia Qvarfort

Department of Physics and Astronomy, University College London

Student. No: SSQVA67

Email: sofia.qvarfort.15@ucl.ac.uk

27 February 2016

Abstract

This is a complementary report for the `Boids` software package which can be found at https://github.com/sqvarfort/bad-boids. `Boids` is a package that simulates the flocking behaviour of animals. In this report, we document the refactoring of a bad implementation of a Boid simulation code into a fast, readable and testable version.

I. Introduction

In 1987, Reynolds published a paper detailing how the flocking behaviour of animals, such as starlings or sardines, can be numerically simulated. The animated objects have become known as so-called 'boids'. In this report, we detail the refactoring of a bad implementation of the code found here into an efficient, readable package.

The report is structured as follows. In Section II we outline the structure of the code. In Section III, we explain the refactoring approach adopted and point out some of the changes made. Each change links to a commit logged on GitHub. In Section IV we discuss the tests written for the code and how regression testing was used throughout the refactoring process. Finally, we discuss the advantages of a refactoring approach in Section V and make some concluding remarks in Section VI

II. Code Structure

The boids are stored as `numpy` arrays containing positions and velocities. These are decided randomly and then adjusted throughout the simulation. The flocking behaviour is simulated through three separate functions: flying towards the middle, matching speed to nearby boids, and avoiding collisions with other boids.

The code has been refactored from a single update function into a class and command line interface. The class creates a `boids` object which initialises a number of boids into random initial positions and velocities. For a full breakdown of the `Boids` class and its methods, see the UML diagram in Figure 1.

The code can be accessed through the command `boids` and takes two mandatory inputs: `boids_no` (an integer) and `config.yaml` (a configuration file). The first argument decides the number of boids in the simulation, and the second argument should refer to a yaml file containing limits for the random spread of initial positions and velocities of the boids. An example of such a file can be accessed by adding the argument `-ex` when calling the code.

The output of the code is an animated scatter plot that shows the boid flock in motion.

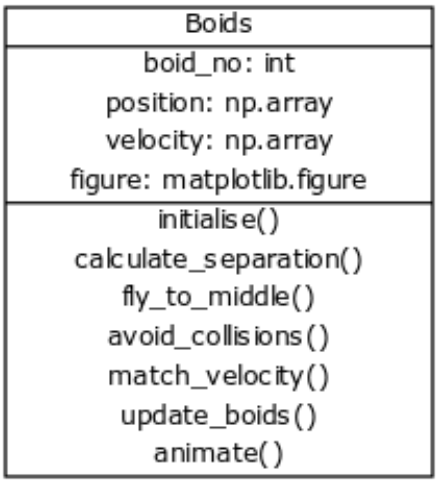

Figure 1: *UML diagram of the `Boids` class.*

III. Refactoring

The refactoring approach has been based on material provided in the course lecture notes. The initial code consisted of an inefficient algorithm with many magic numbers contained in one very large update function. This was gradually changed while still retaining the functionality of the code at each new commit. In the following section, we outline how the code was refactored and link to each commit where the change was performed. The full log of commits can be found here.

I. Identifying refactoring smells

The following smells were identified and corrected.

Replacing magic numbers This was done throughout the refactoring process, for example changing the number of Boids into an easily accessible variable `boid_no`, commit d0ecbd997414939af6c1769529e96ae970c5d931. Also, commit 6b1c41966d0c6f48eced7ea55c872305b5c5f0ca is another example of where we make sure the bounds

of the graph adapt to the input values chosen by the user, instead of being defined as arbitrary numbers.

Replacing repeated code with functions Certain calculations, like working out the separation between all boids are required for more than one of the class methods. Therefore, these calculations have been placed in a separate method, `calculate_separations()`, which is accessed once for each call of `update_boids()`. This makes each function more efficient. See commit 5119b660b00488bb33ab9dc246db279d8e020fef for these changes.

Changing variable names Variables and function names have been changed into a more readable format. The biggest change was when we replaced the iterated loops with `numpy` arrays in commit d0ecbd997414939af6c1769529e96ae970c5d931.

Replacing loops with iterators The greatest advantage brought to the new code is that of `numpy` which allows, through compact syntax, to carry out all the operations at once. This change was introduced early in commit d0ecbd997414939af6c1769529e96ae970c5d931.

Replace constants with a configuration file Similarly to replacing magic numbers, this was done in commit db8654bb3b78cbf2aad4dc9b895bcb69f8704185.

Breaking up large functions This was done in commit 267042203f6e372fc3af4206b46e16989d1301eb where we split the `update_boids()` method into several smaller methods. Each smaller method handled one instance of the flocking behaviour simulation.

Separate code concepts into files or modules Initially, the animation function was in the same file as the boid initialisation code. We moved the animation outside the class in a command-line interface file which calls the `Boids` class. See commit 979994280527631d6f5f0b41f8b2c14117ad24e0.

IV. Tests

The original update function was divided into separate methods to make it more readable, but also in order to simplify testing. Three tests have been written for this implementations.

Firstly, we have the regression test which ensures that the code functions correctly throughout the refactoring process. When changing the code structure to handle `numpy` arrays, the regression test was subsequently changed to interface with the new `Boids` class. Since the methods for velocity alteration perform the calculation in a different order, the regression test with an original tolerance of $\delta = 0.01$ fails. However, changing to $\delta = 0.06$ causes the test to pass, which indicates that the calculation stayed essentially the same. Since the change is very small, the regression test is deemed to still hold. In fact, rather than recording new fixtures, δ can be used as a measure of distance to the original code.

Secondly, we have written tests for every method in the `Boids` class. Since the motion of many boids essentially constitutes an N-body problem, we have chosen to implement tests for interactions between a count of two boids. The motion of a two-body problem is predictable, and by choosing non-random initial positions and velocities, the effect of the various functions can be easily predicted.

Finally, we have written tests for bad input to ensure that the code returns ValuError or TypeErrors when it encounters a problem, such as a negative number of boids or a non-existent configuration file.

V. Advantages of Refactoring

The most obvious advantage to refactoring the code is that the functionality of the code remains unchanged throughout the refactoring process. If a code is in continuous use, the refactoring can take place while still obtaining results from the code. Making sure that a regression test passes between every commit ensures that external uses can keep using the code. However, changes to the code might cause small deviations in the calculations, as we saw for the methods in the `Boids` class. Whether this constitutes a problem depends on the nature of the calculation in question.

The refactoring process also requires a good understanding of the original code. While this requires a time investment, it could also provide benefits in that the refactoring user can later utilise the code more efficiently.

VI. Conclusions

In this report, we have documented a refactoring process of a flocking behaviour simulation of so-called boids. We conclude that the refactoring approach is useful for maintaining the functionality of code while optimising it. Several code smells were identified and corrected through the refactoring progress. The final result is an efficient, readable and user-friendly packaged code.