Barnes-Hut N-body simulations using Python

F.Baker
7th December 2017

1 Introduction

N-body simulations are large scale physics simulations, which beget interactions between $N >> 1$ bodies. A calculation involves computing the influence of all other objects on a single object, for each object in the simulation. As such, these simulations often have a long computation time, rising with N^2. The Barnes-Hut simulation is an algorithm for 2D gravity N-body problems, decreasing computational time by making spatial approximations, dividing the simulated region into a *Quad-Tree* [1]. This allows for far away bodies from the object in consideration to be considered as a single mass, and only a single calculation to be made. This reduces the computational time to $N \log N$, producing faster simulations for large N.

2 Gravity simulation

In a gravity N-body simulation, the force acting on a single body is calculated for between each other body in the simulated region. The force vectors are then added for the body in consideration, allowing for the calculation of the acceleration, which is then used to calculate the subsequent motion of the body.

2.1 Gravitational force

Gravitational force between two bodies is given by Newton's Equation

$$\vec{F}_{1,2} = G\frac{m_1 m_2 \vec{r}_{1,2}}{|r_{1,2}^3| + \eta} \tag{1}$$

where $\vec{F}_{1,2}$ is the force experienced on mass m_1 due to mass m_2, G is the gravitational constant, and $\vec{r}_{1,2}$ is the distance between m_1 and m_2. An η parameter is introduced as an approximate 'softening factor', such that in the simulation a maximum force may be defined. The justification for this is to reduce the time-step in the integration method, and generally creates less violent motion.

3 The Barnes-Hut algorithm

The Barnes-Hut algorithm divides the simulated region into a Quad-Tree, which provides the framework for where calculations are performed.

3.1 Quad-Tree structure

The generation of the Quad-Tree involves quartering a region into square sub regions (nodes) recursively. The Python pseudo-code approach to this is

```
def divide(region):
  divide region into 4 sub_regions
  for Object in region.Objects:
    if Object.position in sub_regions.region
      sub_region.Objects.add(Object)
```

```
if sub_region.Objects > 1:
  divide(sub_region)
```

This approach is applied recursively until each node contains only one object (Figure 1).

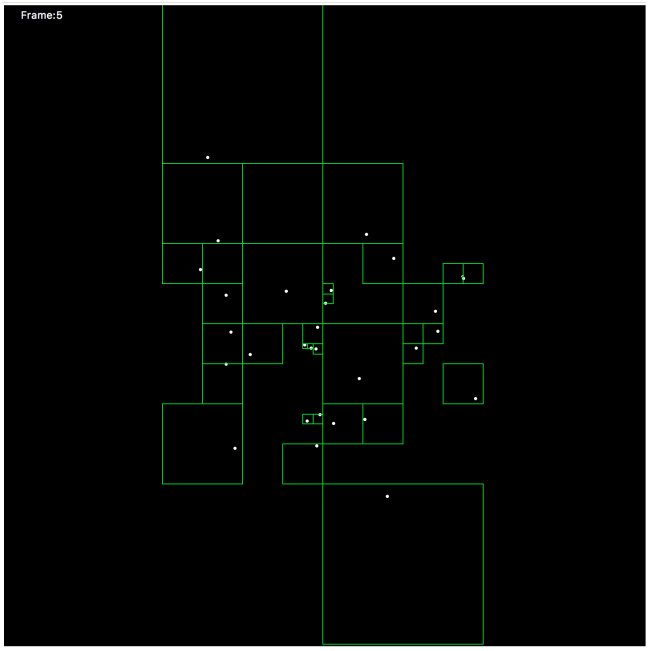

Figure 1: Program structure

.

3.2 Performing calculations

Performing a gravitational force calculation as in Equation 1 follows the Python pseudo-code procedure

```
def descend(region, Object):
  for sub_region in region:
    distance = abs( sub_region.centre_of_mass - Object.position)
    length = sub_region.side_length
    if length/distance < theta or sub_region.Objects == 1:
      return force(Region.mass, Object.mass)
    else:
      descend(region, Object)
```

The parameter `theta`, θ, is an adjustable parameter which describes the accepted ratio of $l/\vec{r}_1, 2$, where l is the side length of a region. A lower value of θ requires the node to be far away from the object under consideration, such that as $\theta \to 0$ the simulation returns to a regular N-body simulation. Higher values of θ create greater approximations and less accurate results, but can heavily reduce computational time.

4 Program Structure

The program was divided into two parts: 1) the simulator (*RunSimulation.py*) which calculates the simulation and renders the frames, 2) the GUI and image cacher. The reason for this split is due to the utilized graphics library not being able to support multi-processing across Frame objects.

4.1 Simulation

The Python simulation follows the Model-View-Controll (MVC) architecture, with a config-file behaving as the user controll (Figure 2). The slight exception is that the controll also manipulates the view section. This

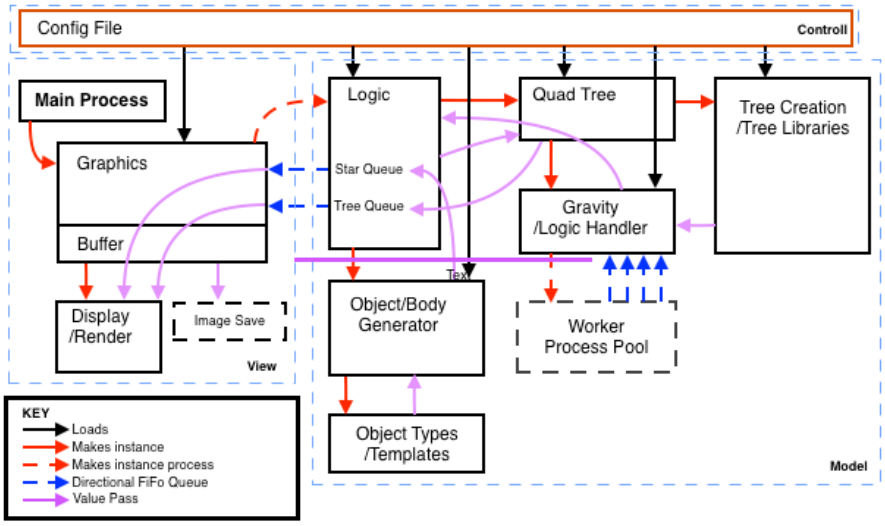

Figure 2: Program structure

.

structure optimizes the performance of the program, by making the view module essentially a listener for the model module. For low $N < 1000$, the program performs as quickly as the calculations can be done. At higher N, the render time bottlenecks performance but only slightly. Possible future implementations may include a canvas-swapping method - drawing on one canvas, whilst displaying the other - to increase performance.

4.1.1 Graphics

The graphical modules of the simulator is built on Python's Tkinter library. The graphics module consists of a file handler, responsible for creating the directory structure for storing images, a graphics handler, which streamlines retrieving values from the star and tree data queues, and lastly the display area, which creates widgets on canvas objects. The display area is built on the Tkinter Canvas classes, as well as the Python Image Library (PIL). TKinter provides the rendering in real time, seen as the direct output in a window, whereas the PIL canvas is used to save jpeg images of the renders. Unfortunately, Tkinter has no direct capturing methods, but is able to draw on screen directly, whereas PIL provides canvas capture, but cannot directly draw to screen. As such, two canvases are drawn on, one for the direct render, the other for the capture. This has the benefit of being able to save output without having to display the Quad-Tree.

4.1.2 Logic

The logic is reliant on multi-processing - dividing the computation task into chunks and calculating them in parallel. Due to Python's Global Interpreter Lock (GIL), threading was not a viable option, as the kernel rotation with threads is still series sequential. The chunks are divided as equal sections of the total gravitational objects, such that the process must only perform N/N_p iterations of the tree transversal, where N_p is the number of processes. This is slightly inefficient, as each process must have the whole tree stored in class memory, which turned out to be nearly twice as large as the gravitational object lists.

The workhorse of the logic modules is the GravityWorker module, which is the calculator process. This module performs all the time-consuming computations, as well as the calculation algorithm described in Section 3.2. Additionally, to calculate the subsequent motion of the object, the module performs a crude Euler-Cromer integration [3]:

```
def step(self, star, acc):
    star.vel += acc * self.t[1]          #self.t[1] is the time step
    star.pos += star.vel * self.t[1]
    self.stars.append(star)
```

```
self.t[0] += self.t[1]
```

This integration method was chosen primarily for speed, as the new velocity and position could be determined in two calculations.

4.2 Graphics User Interface

The Graphics User Interface (GUI) is an interface for editing config files given to the simulation, as well as visualizing simulation output. The GUI for editing config files serves primarily as a way to ensure the user does not enter values which would cause errors. In the current build, there is no value check beyond value type. This is plan for a future build, as the framework for such a condition check exists, but is not fully implemented. Type checks are performed against a `_config_check.ini`. The config template was created as a structure for the program, to help streamline development. As a consequents, some of the parameters do not actually influence the program yet and may be considered redundant.

4.2.1 Image cacher

The image cacher is designed to cache a PIL object of each image in a desired folder. It is then able to visualize the image at a much faster rate than the real time simulation, with peak speeds reaching around 10ms between frames. This is the optimal way to view the result of a simulation.

5 Simulation performance

The performance of the simulation approximately matches the expected $N \log N$ time scaling. This can be improved with tweaking of the θ parameter. A brief simulation shows $T \propto 1/\theta$, where T is the time between each frame.

Example output of a simulation of $N = 2000$ can be seen in Figure 3. Interesting to note are the 9 regions of high star density which formed from the simulation, which mimics stellar or planetary formation.

5.1 Galaxy

A later build of this simulator included the creation of galaxy templates, which involve a single very massive object at the center, representing a black hole, and stars normally distributed around the central point. An example of such a simulation can be seen in Figure 4. Interesting to note are the formation of slight spiral arms as time progresses. Especially interesting results may be observed with a single galaxy, with $size = 99$.

6 Appendix

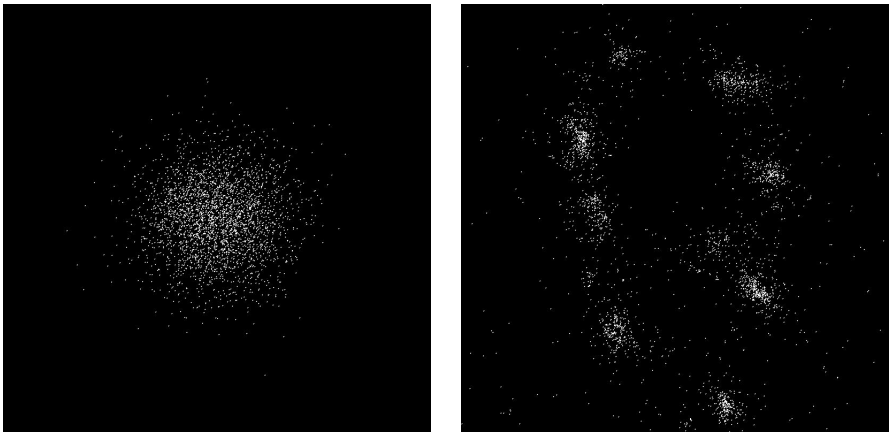

Figure 3: Simulation of normally distributed, 0-velocity stars, with $N = 2000$. Left: Frame 0. Right: Frame 178

.

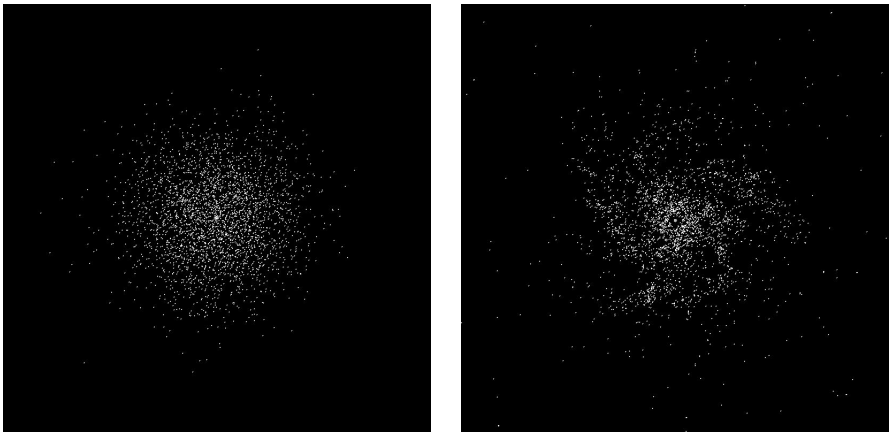

Figure 4: Simulation of a small galaxy with $N = 1000$. Left: Frame 0. Right: Frame 221.

.

7 Bibliography

References

[1] Barnes, Josh, and Piet Hut, "A Hierarchical O(N Log N) Force-Calculation Algorithm." Nature 324.6096 (1986): 446-449.

[2] Giancoli D C, "Physics for Scientists & Engineers with Modern Physics" 4th edition. Harlow: Pearson Education Limited, 2014.

[3] A. Cromer, "Stable solutions using the Euler Approximation" American Journal of Physics, 49, 455 (1981)