Homework: The Barnes-Hut Algorithm
Programming for Scientists

Assignment Overview

In this assignment, we will scale up our gravity simulator to be able to handle the movements of thousands of stars over a much longer period of time. If we were to dig into the details of our `SimulateGravity` function, then we would see that by far the most time consuming part of the algorithm will be the computation of force between every pair of heavenly bodies because we will have to consider n^2 pairs, where n is the number of bodies. As n grows, this will make the algorithm very sluggish given that we need to compute the force every generation for many generations.

Reflecting on the realities of gravity gives us some hope for how we might speed up our algorithm. As you are reading this, you are under the gravitational influence of every molecule in every star in the farthest stretches of the universe; but how their movements influence you hardly crosses your mind on a daily basis. In other words, gravity is a *local* force, and so we can develop a *heuristic* in which we only compute the force of gravity between nearby objects. However, the brute force algorithm for finding all distances between points in our universe will require us to consider all pairs of points, an algorithm that will also grow as a factor of n^2.

In this assignment, we will see the benefit of using a **data structure** for organizing the n bodies in the universe so that we don't need to consider every pair of points in this universe.

The Barnes-Hut Algorithm

Clustering nearby points

Our critical insight is that if a collection of bodies are *very close to each other*, then they have comparatively little effect on a distant body and they can be grouped together. In particular, we will combine these bodies into a single body with combined mass at the **center of gravity** of the points' coordinates. (See figure below.)

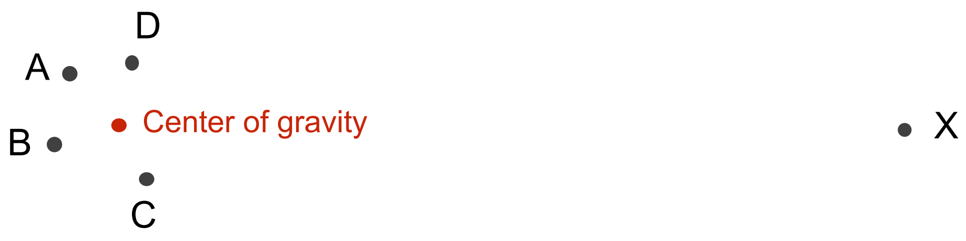


Figure 1: The four points A, B, C, and D are near each other and distant from X, so their net force acting on X will be very similar to the net force exerted by a hypothetical star with combined mass at the center of gravity of A, B, C, and D.

The center of gravity of a collection of bodies has coordinates equal to the weighted average of the points' coordinates. For example, if three stars have equal mass and the coordinates $(-4, 7)$, $(0, 5)$, and $(10, 3)$, then their center of gravity is located at

$$\left(\frac{-4+0+10}{3}, \frac{7+5+3}{3}\right) = (2,5)\,.$$

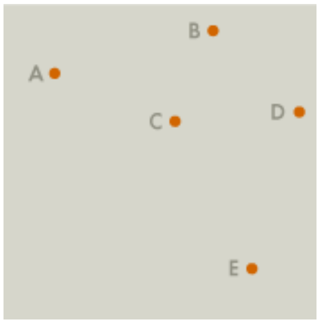
But if the same stars have masses in the ratio of $4:2:1$, then their center of mass is

$$\left(\frac{-4\cdot 4+0\cdot 2+10\cdot 1}{4+2+1}, \frac{7\cdot 4+5\cdot 2+3\cdot 1}{4+2+1}\right) = (-6/7, 41/7)\,.$$

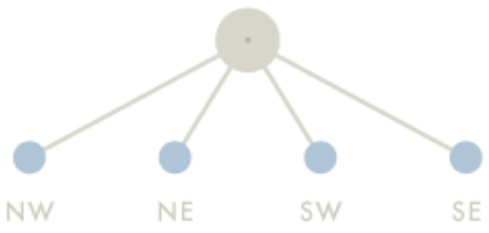
This all may seem reductive. How can we quickly find that a collection of bodies are near each other without computing the distance between every pair of bodies in the universe? And how would *adding* a body that isn't present in our universe possibly be helpful for reducing the runtime of our algorithm?

Forming a quadtree

The data structure that we will apply is called a **quadtree**. In the past, we have worked with *binary trees*, in which every node has 0, 1, or 2 children; in a quadtree, nodes can have up to four children. The quadtree of a collection of bodies is defined as follows for the hypothetical collection of five bodies with mass equal to 1 shown in the figure below.[1]

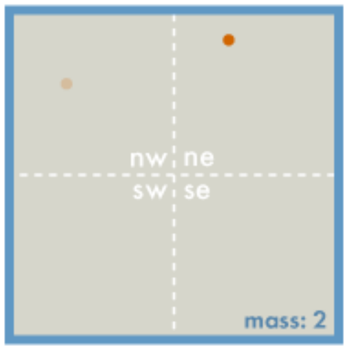


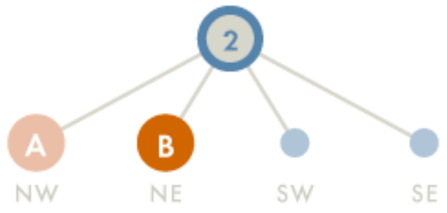
First, form a root node with four children, corresponding to each quadrant of the two-dimensional space. These nodes are labeled "NW", "NE", "SW", and "SE", but you can just think of them as children 0 through 3 of the root.



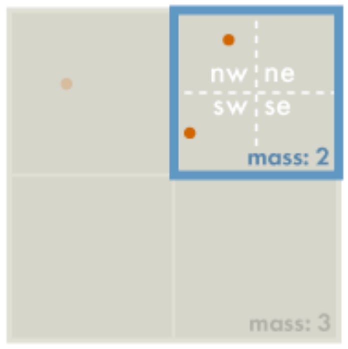
We then will build the tree by ranging through the bodies in the universe and assigning them to points. Body A occurs in the northwestern quadrant, so we will assign a pointer to A at the leftmost child of the root; body B occurs in the northeastern quadrant, so we assign a pointer to B at the second child of the root. As we assign these pointers, we create a "dummy" star at the root whose position is the center of gravity of A and B and whose mass is their combined mass (2).

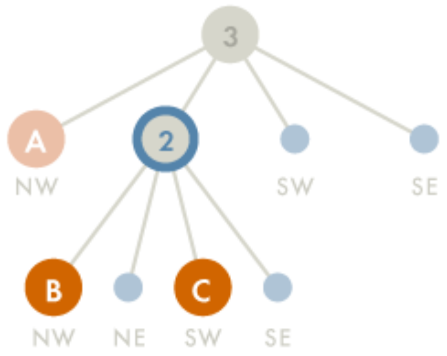
[1]Barnes-Hut figures are borrowed from `http://arborjs.org/docs/barnes-hut`.



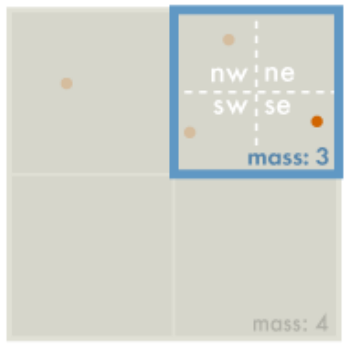


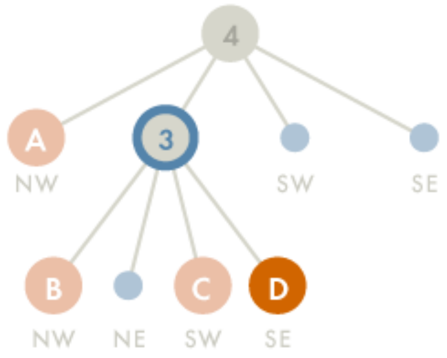
Point C also belongs to the northeast quadrant, and so rather than assign it to the node containing B, we will form four children of this node and once again divide this quadrant into four subquadrants. Within this quadrant, B belongs to the northwest subquadrant and C belongs to the southwest subquadrant, and so we assign the corresponding nodes pointers to B and C.[2]. As we add C to the tree, we update the mass and center of gravity of every internal node between the root and C. The root, for example, will have position equal to the center of gravity of A, B, and C, and mass equal to their combined mass (3).





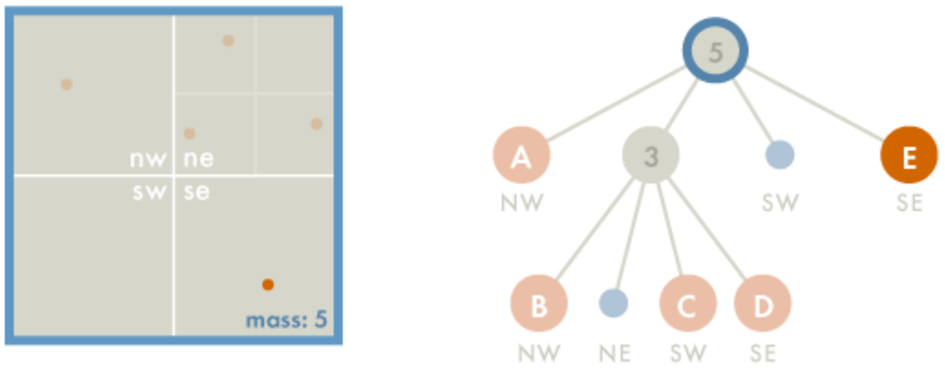
Next, body D belongs to the unoccupied southeastern subquadrant of the northeastern quadrant that we already subdivided. Therefore, we will assign this leaf node to D and update its parent to have mass 3 and the root to have mass 4.





[2]If B and C had wound up in the same subquadrant, we would continue to subdivide this quadrant until the two bodies could be assigned to distinct children.

Finally, point E is the only body in the southeastern quadrant, and so we assign this quadrant's node to point to E; the root is updated to have mass 5. The tree shown in the figure below is the final quadtree of the five bodies.

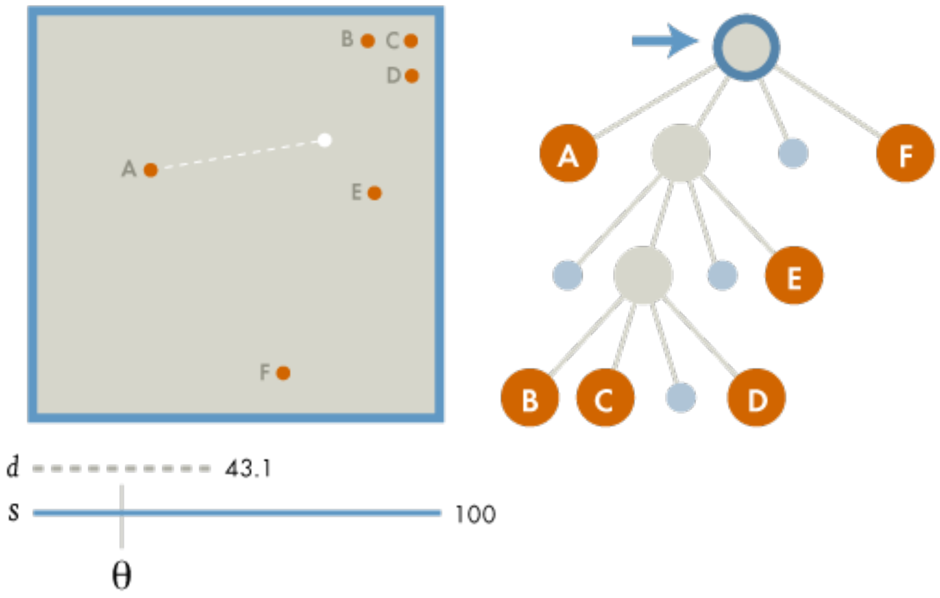


Using the quadtree to power a heuristic computing the force of gravity

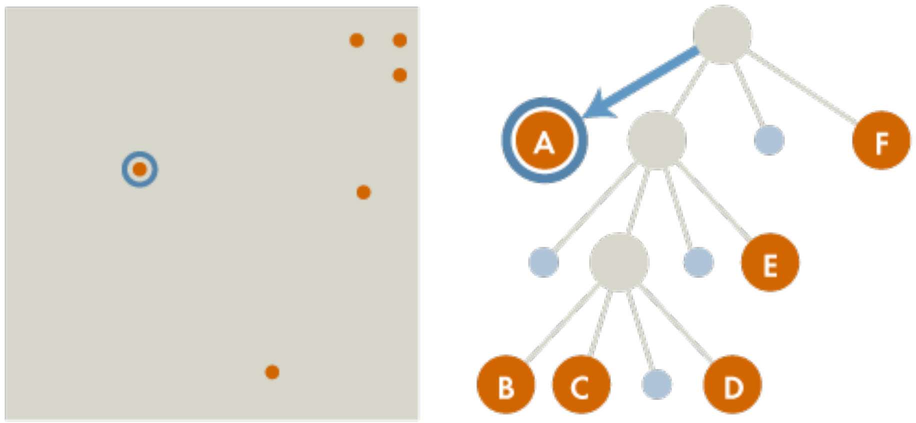
Now that we have built the quadtree, we will return to our key insight that if a collection of bodies are close to each other and distant from another object X, then their contributions to the force of gravity on this body can be combined. In terms of the quadtree, bodies are close to each other if the smallest sector (i.e., their most recent common ancestor in the quadtree) has a small width (denoted s); they are very far from X if the distance (denoted d) from their center of gravity to X is large.

We will combine s and d into a single statistic s/d: if s/d is small, then there is little individual gravitational effect; if s/d is large, then the individual gravitational effects of the bodies on X are worth considering. As a result, for a given body X, we will use a parameter θ to compare s/d against. We scan through the tree, and if we encounter an internal node for which s/d is less than θ, we do not explore this subtree further; instead, we simply consider the force of the "dummy" body occupying this node on X.

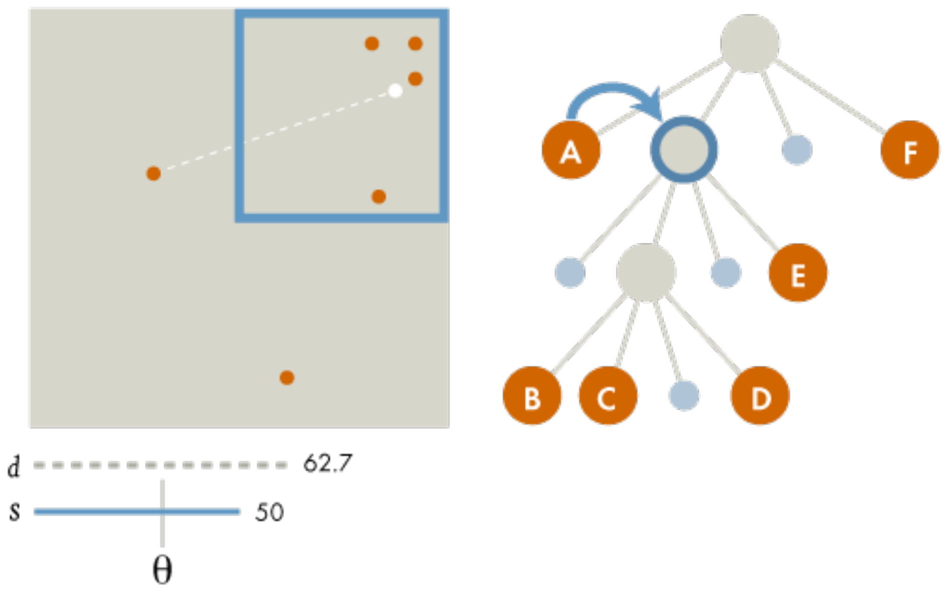
The following figures illustrate the heuristic using the quadtree of six bodies to approximate the net force of gravity acting on body A with θ equal to 0.5. We start with the root node, which has s equal to 100 and d equal to 43.1. Because $s/d > \theta$, we move on to the four children.



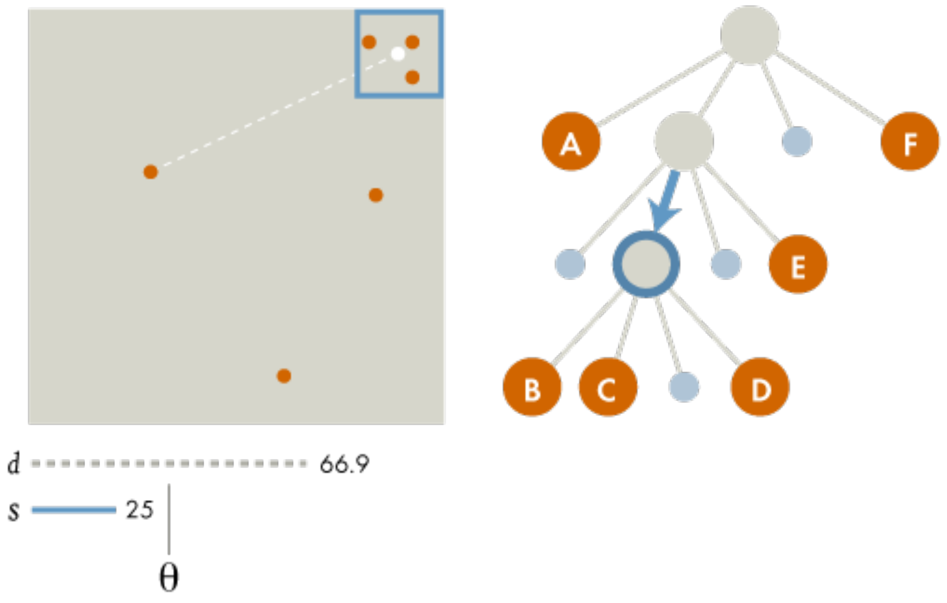
We start with the first child of the root, which points to A. This refers to A, so we skip this child.



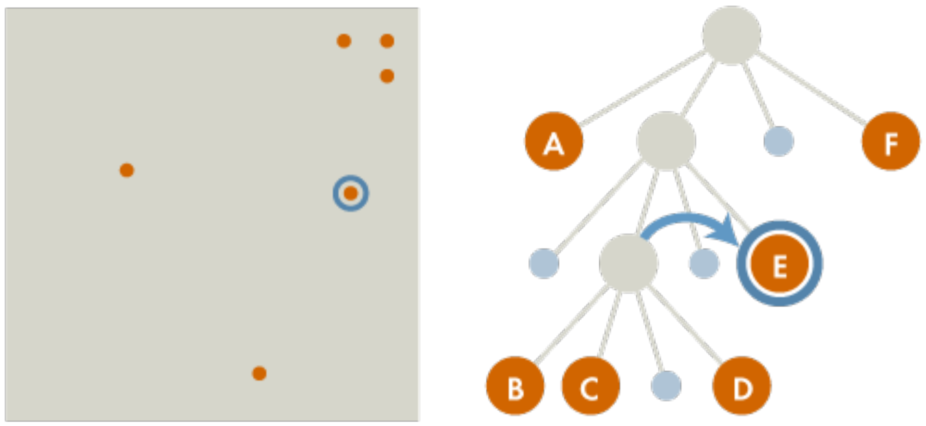
We then move to the second child of the root, which has s equal to 50 and d equal to 62.7. Because $s/d > \theta$, we explore this node's children.



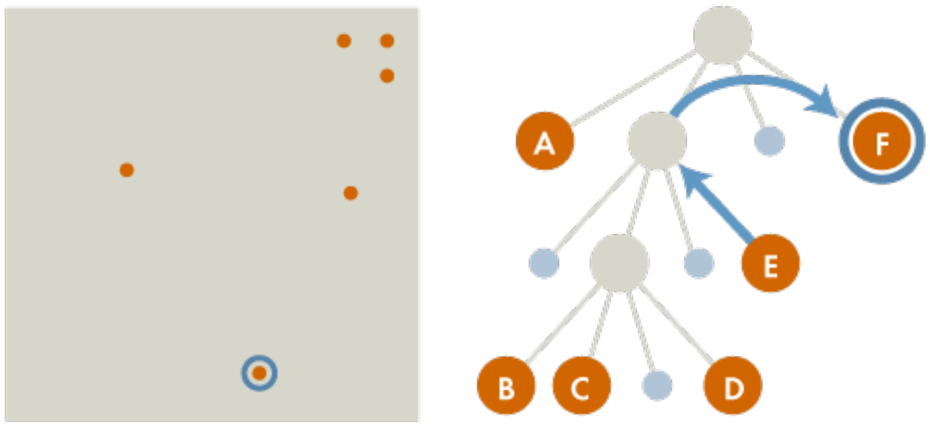
The left child of this node is blank, so we skip it and consider the second node. This node has s equal to 25 and d equal to 66.9, so we will treat the collection of bodies in the subtree rooted at this node as a single object when computing the force acting on A; this object is represented by the "dummy" body assigned to the current node.



The next node at the current level of the graph has no body assigned to it, so we skip to body E. This is a single body, and so we go ahead and add the force that E exhibits on A to the net force acting on A.



Finally, the third child of the root is empty, so we skip to the root's fourth child, which contains a single body (F). We therefore add the force that F exhibits on A to the net force acting on A.



In summary, when computing the force acting on a body b:

1. If the current node is a leaf:
    - If the leaf does not have a body assigned to it, it does not contribute to the net force on b.
    - If the leaf does have a body assigned to it, then compute the force of this body on b and add it to the net force acting on b.
2. If the current node is an internal node, compute s/d.
    - If $s/d > \theta$, then iterate on the internal node's children.
    - If $s/d \leq \theta$, then treat the internal node as a single body whose mass is equal to the sum of bodies in the subtree beneath it and whose position is the center of gravity of these bodies; this information is pre-computed when forming the quadtree. Compute the force that this dummy body exerts on b and add this force to the net force acting on b.

Every time we update the universe, we generate a quadtree, and we then use the quadtree according to the above heuristic to approximate the net force acting on each body in our universe. The resulting heuristic is called the **Barnes-Hut algorithm**.[3].

Analyzing the runtime of Barnes-Hut

Although the details of proving this are beyond the scope of this assignment, it can be demonstrated that constructing the quadtree takes on the order of $n \log n$ steps, and the same goes for using this quadtree to approximate the net forces acting on every object in our universe. As a result, the runtime of the Barnes-Hut algorithm grows analogously to some constant times $n \log n$.

The runtime of the algorithm is also heavily dependent on the choice of θ. The smaller the value of θ that we use, the more stringent the heuristic will be about when to consolidate multiple bodies into a single body, and the longer the algorithm will take.

You may be curious why it is such a big deal that we replaced an algorithm that takes on the order of n^2 steps with one that takes on the order of $n \log n$ steps. If we divide these terms, we obtain a speedup on the order of $n/\log n$, which may not seem particularly impressive. But if we are running a simulation for n equal to 10 billion stars (ten percent of the number in the Milky Way), then $\log n$ is equal to 10 and so $n/\log n$ is equal to 1 billion, a formidable improvement in runtime. Notably, the more that n grows, the larger the speedup will be; this example offers a microcosm for why we care about efficient algorithms as much today as we ever have, since the sizes of datasets are growing so quickly across many scientific fields.

Assignment

Set up

Download the `BarnesHut` template from the course code repository, and unzip it into the `src` directory inside of your `go` directory.

What you should do

In this assignment, you will expand on a Go program template to implement the Barnes-Hut algorithm. The function implementing this algorithm should take the initial positions of a collection of stars of equal mass (as well as parameters θ and *numGens*) and, after updating the stars *numGens* times according to running Barnes-Hut with its key parameter θ, generate an animated GIF illustrating the evolution of the "universe" after *numGens* steps. To get full credit, you should model the collision of two small simulated galaxies of stars.

Your program should be runnable using the command (Mac)

```
./barnes-hut command
```

where the Barnes-Hut algorithm is run on one of three datasets based on the value of `command`:

- If `command` is `jupiter`, we simulate the Jupiter moons.
- If `command` is `galaxy`, we simulate a galaxy of 500 stars spinning in place.
- If `command` is `collision`, we simulate the collision of two galaxies of 500 stars colliding.

[3]See https://www.nature.com/articles/324446a0

The third parameter choice is not that strange; after all, the Milky Way and Andromeda galaxies will collide in about four billion years. After running the Barnes-Hut algorithm, your program should produce an animated GIF of the system.

Using previous code

You are welcome to use the physics engine that we built in class as a model to start working on this. Some functions, such as the ones updating positions and velocities, will not need to change much.

Note about perfect performance

Round-off errors will accumulate because of the fact that we are updating our simulation over a series of time points, as well as error that propagates within the force computation approximations performed by the Barnes-Hut heuristic. You may also note that it is difficult to form a galaxy with stable orbits over any significant period of time; this is in part because the orbits that we see in existing galaxies have had billions of years to stabilize. It is also because forming a realistic distribution of star masses and their initial velocities based on their radius from the center of a galaxy is a difficult problem that would take us into an astrophysics course. So, you should not fret if your galaxies sometimes lose stars.

Submission

You should place all functions that you write into `functions.go`.

Please submit all of your code via Canvas along with the most interesting GIF that you produce of the collision of two galaxies. Your code will be run for the `jupiter` and `galaxy` flags but we do not expect it to be efficient enough to generate a collision of galaxies in a short period of time.

Learning outcomes

After completing this homework, you will have

- gained additional understanding of object-oriented programming, including methods and pointers;
- appreciated the importance of well-organized, readable, modular code as the size of a program grows;
- implemented a medium-scale programming project based on a code-base that we have already built.