

Nbody Simulator

Simon Tartakovsky

December 15, 2020

1 Intro

Simply put an N-body simulator needs to solve the dynamic system where N particles are connected with springs to one another. Thus this is a system with $N(N - 1)/2 \approx \mathcal{O}(N^2)$. This sounds great on paper but simulating something with $N = 100000$ becomes impossible as $N^2 = 1 \times 10^{10}$ and this is just for a single step!

Enter the magic of an FFT which allows us to compute the potential generated by any number of particles in $k \log(k)$ where k is the size of the potential generating mass matrix (or in other words the refinement of the grid one which we are solving). Typically $N \approx k$ and thus a problem where $N = 100000$ is all of a sudden within reach.

2 Implementation

If you want to skip to the results (and I respect that since code can be dry just scroll down, I wont be too butt-hurt)

2.1 Overview

The code is structured with a python class. To instantiate a simulation we simply need to call the Nbody class from the nbody_tools.py as follows:

```
1 sim = Nbody( pos = None, vel = None, mass = None, potential = r_squared, dimension = 2,  
2           BC = "wrap", grid_size = 1, grid_ref = 1, dt=0.001)
```

and then we can run a simulation using the following:

```
1 results = sim.run_sim(steps, frame_return)
```

which will return a dictionary containing the mass distribution arrays of every frame_return frames from a steps long simulation.

For the purposes of my assignment, I failed miserably at graphing 3D graphs in real time and decided to just project them down to 2D and plot the heat map there. This is achieved with the run simulation wrapper which works as follows:

```
1 def sim_wrap(nbody_name, steps, frame_rate):  
2     res = nbody_name.run_sim(steps = steps, frame_return = frame_rate)  
3     print("Energy changed by:",  
4           (np.max(res["energy"]) - np.min(res["energy"])) / np.abs(np.mean(res["energy"])) * 100,  
5           "% throughout the simulation")  
6     input("Sim has been computed, press enter to view the movie")
```

```

7 plt.clf()
8 for pic in res["density"]:
9     plt.ion()
10    while len(pic.shape) > 2:
11        pic = np.sum(pic, axis=-1)
12        plt.imshow(pic)
13        plt.show()
14        plt.pause(0.1)
15        plt.clf()
16 plt.imshow(pic)
17 plt.show()
18 print("This movie experience has been brought to you by the Nbody movie company. \n
19       Remeber to take all trash with you on the way out")

```

The testing.py file contains the test cases asked in the PDF on github

2.2 Hash

A critical part of the function of the simulator is to transition to a density map to be able to use an FFT to convolve with the potential from a point source and thus create a scalar potential. This is achieved with the hash attribute of my class:

```

1 def hash(self):
2     """
3     main position to grid mass function, uses the numba thing above to go faster
4     """
5
6     #generate the grid
7     N = int(np.ceil(1/self.ref))
8     grid_shape = [N for _ in range(self.dim)]
9     grid = np.zeros(grid_shape)
10
11    #grid indexes from coordinates
12    grid_indexii = self.pos_to_grid(self.pos)
13
14    ##something about numba changing some dependancies and it dropping
15    ##a warning which i couldnt be bothered to fix
16    with warnings.catch_warnings():
17        warnings.simplefilter("ignore")
18        #call the actual hash function
19        hash_grid = fast_hash(grid_indexii,grid_shape,self.m, grid)
20    if self.BC == "wrap":
21        #great its already circular so return
22        return hash_grid
23    elif self.BC == "hard":
24        ##we need to pad
25        pad_size = [[hash_grid.shape[i],hash_grid.shape[i]] for i in range(self.dim)]
26        return np.pad(hash_grid, pad_size)
27    #return

```

This works by first computing the grid size based on the grid refinement the user asked for and then it gets grid coordinates with the helper function:

```

1 def pos_to_grid(self, pos):
2     """
3     Simple helper that converts from coordinate to grid square index
4     """
5     return np.floor(pos/self.ref).astype(int)

```

following which it calls the numba worker function:

```

1 @jit
2 def fast_hash(grid_indexii, grid_shape, mass, grid):
3     """
4     Does a numba particle position to grid density
5     """
6     for particle_num, grid_index in enumerate(grid_indexii):
7         ##loop through all the particles:
8         grid[grid_index[0], grid_index[1]] += mass[particle_num]
9     return grid

```

which simply loops through each particle and adds its mass to the corresponding grid square in the mass distribution.

The creation of the density map is concluded by either returning it or padding it in the case where we do not want to see the wrap around nature of the FFT show up.

2.3 Potential

The potential starts by needing a template which is done with the following function:

```

1 def make_template(self):
2     """
3     A very tryhard function which generates the potential centered around a corner
4     """
5
6     #make the zeros template
7     N = int(np.ceil(1/self.ref))
8     grid_shape = [N for _ in range(self.dim)]
9     template = np.zeros(grid_shape)
10    template = np.zeros_like(self.grid, dtype=float)
11
12    #generate a basis vector (this will not handle a non-square grid)
13    #said basis vector is rolled to have zero at index zero
14    # and then positive up to half way point and negative thereafter
15    idx = np.roll(np.arange(-template.shape[0]//2, template.shape[0]//2, dtype=float),
16                  template.shape[0]//2)
17
18    #we need dimension many basis vectors
19    basis_vec = np.array([idx for _ in range(self.dim)])
20
21    #this now created the distance grid from all our basis vectors
22    dist_grid = np.array(np.meshgrid(*basis_vec))
23
24

```

```

25     #now rescale the dist grid
26     dist_grid = dist_grid/np.max(dist_grid) # * self.grid_size
27
28     #generate the template using the potential function
29     #this is greens funciton
30     #to catch the divide by zero
31     self.k = 0.1
32     with warnings.catch_warnings():
33         warnings.simplefilter("ignore")
34         template = self.pot_func(np.linalg.norm(dist_grid, axis=0, keepdims=False),self.k)
35
36     #more softeing
37     ind = [0 for i in range(self.dim)]
38     ind_n = list(ind)
39     ind_n[1] = 1
40     template[tuple(ind)] = template[tuple(ind_n)]
41     # plt.imshow(template)
42     # plt.show()
43
44     if self.BC == "wrap":
45         return template
46     elif self.BC == "hard":
47         N = template.shape[0]
48         template[N//3:2*N//3, N//3:2*N//3] = 0
49     return template

```

Which, I will be the first to admit, is needlessly complicated (but this was at the start of the adventure and I was young and full of hope). It works by generating a set of basis vectors that go from $-N/2$ to $N/2$ and then creating a mesh of arbitrary dimension. This mesh is then reduced using the `linalg.norm` and passed to the potential greens function which so happens to be simple in this case. The grid is also softened by removing the singularity at the origin and then returned or padded if the boundary condition calls for it.

2.4 Calculating the Force

This is where things went down hill... I calculate the force and update the velocity all in one go so the function which is responsible for it is this:

```

1  @njit
2  def step_vel(self):
3      """
4      This is to compute the velocity from the field. A bit harder than the last
5      We compute the vector field everywhere first and then evaluate the force individually
6
7      when the boundary condition is non-periodic this function will be doing a lotttt of extra work
8      might fix later
9      """
10     #compute a new potential and calculate the force to step velocity
11     #this function does the convolution
12     t1 = time.time()
13     potential = self.make_pot()
14     t2 = time.time()

```

```

15     #me messing with things
16     self.pot = potential
17
18     ##now get force field:
19     #we need to roll a little to the right and a little to the left and take the difference
20     #this is a gradient operation which pushes it to another dimension for the different components
21
22     #generate an array which is Ndim x grid_side^Ndim
23     derivatives = np.zeros([self.pos.shape[1]] + list(self.grid.shape))
24
25     #ok so this one is a bit of a crazy one.
26     #this opp array is gonna be a vector [grid_side^Ndim-1, grid_size^Ndim-2, ..., 1]
27     #it willl later be used for a crazy broadcasting operation
28     from functools import reduce
29     opp= np.array([reduce(lambda x,y: x*y, derivatives.shape[i:-1]) for i in range(1,
30                          len(derivatives.shape)-1)] + [1])
31
32     #get grid indexes
33     grid_pos = self.pos_to_grid(self.pos)
34     if self.BC == "hard":
35         grid_pos = grid_pos + (np.array(self.grid.shape)//3)[np.newaxis, :]
36
37     #this computes the gradient stacking the dx,dy,dz... in axis=0 of derivatives
38     t3 = time.time()
39     for dim in range(self.pos.shape[1]):
40         derivatives[dim, :] = (np.roll(potential, -1, axis=dim) - np.roll(potential, 1,
41                                  axis=dim))/(2/self.ref)
42     t4=time.time()
43     #print(t2-t1, t4-t3)
44     #now for magic
45
46     #now we flatten the first axis such that it is Ndim by however many (so flatten the coordinate)
47     der_flat = derivatives.reshape(derivatives.shape[0], -1)
48
49     #get the new flat index by dotting the grid position with the opp vector which converst to this new view
50     indexes = np.dot(grid_pos, opp.T)
51
52     #index the derivatives by the points of interest
53     ordered_der = der_flat[:,indexes]
54
55     #they happen to end up in a horizontal array which i dont like
56     ordered_der = np.transpose(ordered_der)
57
58     #update finally.....
59     self.vel = self.vel - ordered_der*self.dt /self.m[:, np.newaxis]
60
61     #save the derivatives:
62     self.derivatives = derivatives

```

Ok so lets walk through it slowly. This thing starts by computing the potential everywhere with a convolution:

```

1 def make_pot(self):
2     """
3     Generate the potential from the mass distribution by convolving
4     The BC should already be handled
5     """
6     mass_ft = sp.fft.rfftn(self.grid, workers = -1)
7     my_pot_ft = mass_ft * self.pot_template_ft
8     return sp.fft.irfftn(my_pot_ft, s = self.pot_template.shape, workers = -1)

```

using a classic convolution. (self.pot_template_ft is just the FT of the template so you did not miss much). Next we want to take this potential and get the force so we take the gradient of it! This is done really confusingly cause I kept everything parametric. In addition the derivatives I used are balanced as this removes the worry of a particle affecting itself (Potential is even so derivative is gone). The comments in the code do a decent job of explaining the numpy wizardry.

Now there is something to note. The way I am doing it is not terrible but not great. I'm computing the potential EVERYWHERE and then using the field to do lookup for individual particles to apply the force to them. This is kinda inefficient since I use mostly sparse grids and thus I am computing the force at places that do not actually mater. In any case computing the force everywhere is still big O of N but its limited by memory speed since np.roll moves stuff around in memory. My attempt to make this faster with numba failed miserably and I was not motivated enough to rewrite it in C or in cupy which would really make it fast.

The last interesting step is the forward Euler step which is part of the leap frog routine!

2.5 Position Step

Since we are on the subject of the solver lets see how we step position:

```

1 def step_pos(self, half_step = False):
2     """
3     Step the position with provisions for a euleur half step to get things going at the start
4     """
5     #compute the new positions from the velocity and then rehash the density map
6     if half_step:
7         self.pos = self.pos + self.vel*self.dt/2
8         if self.BC == "wrap":
9             #do the wrap around if particles left the grid
10            self.pos = self.pos%1 # self.grid_size
11        else:
12            #hard walls will obviously not conseerver energy
13            self.pos = np.clip(self.pos, 0, 1)
14            #now that we have new position rehash
15            self.grid = self.hash()
16            return
17        self.pos = self.pos + self.vel*self.dt
18        ##for ICrucla rBC only
19        if self.BC == "wrap":
20            #do the wrap around if particles left the grid
21            self.pos = self.pos%1 #self.grid_size
22        else:
23            #hard walls will obviously not conseerver energy

```

```

24     self.pos = np.clip(self.pos, 0, 1)
25     #now that we have new position rehash
26     self.grid = self.hash()
27     return

```

This is very simple as all we need to do is take the velocity and Euler step (or half Euler step) forward. We also carefully manage the boundary condition here which is either a wrap around or a hard wall. With the new position computed we will thus compute a new density map

2.6 Leap Frog

So to actually run the simulation we do the following:

```

1  def run_sim(self, steps = 5000, frame_return = 25):
2      """
3      Actually run the sim and return all the frames
4      """
5      import time
6      print("Running Sim, Depending on the grid size this can take a while...")
7      print("expected time to run is:", 2e-7*self.grid.size*steps, "seconds")
8
9      frames = []
10     pos = []
11     energy = []
12     t1 = time.time()
13     self.step_pos(half_step=True)
14     for step_N in range(steps):
15         #print(hello)
16         t3=time.time()
17         self.step_vel()
18         t4=time.time()
19         self.step_pos()
20         t5=time.time()
21         #print(t4-t3,t5-t4)
22         if step_N%frame_return == 0:
23             frames.append(self.grid)
24             pos.append(self.pos)
25             energy.append(self.calc_energy())
26             pass
27     t2 = time.time()
28     print("Actually took:", ((t2-t1)), "seconds")
29     return {"density": frames, "position":pos, "energy": energy}

```

What this does is first takes a half position step to offset the positions from the velocities (this isnt actually needed since our initial conditions are not real and we could have happily just pretended that we got staggered initial conditions). Then we get potential, compute velocity, compute position in a loop of length number of steps and return the states everyonce in a while for plotting

The interesting bit here is the energy calculation which we do as follows:

```

1 def calc_energy(self):
2     ##compute the energy while we are at it
3     kinetic = np.sum((sp.linalg.norm(self.vel, axis=1)**2)*self.m)
4     potential = np.sum(self.pot* self.grid)/(self.grid.size)
5     ##subtract out the self induced potential
6     ind = [0 for i in range(self.dim)]
7     potential = potential - (np.sum(self.m)*self.pot_template[tuple(ind)])
8     potential = potential*self.k
9     #print("kinetic", kinetic, "potential", potential)
10    #print(kinetic + potential)
11    return kinetic + potential

```

What this does is take the kinetic energy as $K = 1/2mv^2$ and potential is $P = \rho V$ being careful to take out the potential due to the particle itself. (I will admit that I am a little skeptical on if this is correct but it gives answers that look plausible so I will let it slide)

2.7 Plotting

2.8 Final Notes

So I am graphing by projecting down into 2D. This is kinda a shame cause its doing N work but you only get to see $N^{2/3}$ of the output. With $N = 100000$ we are loosing a factor of 50! Anyways this means that for my simulations to run in reasonable time they look a little bit pixelated but if you want to have fun switch it to 2d and then for no extra cost it will have pictures that are 50 times better.

The performance right now is limited equally by the FFT and the gradient calculation. I did not worry too much about optimizing the gradient cause no matter how much work I put in, I will only ever get a factor of 2 improvement overall.

3 Results

Here are some snippets of the results. Feel free to run testing.py in the project directory of github.com/merny93/coralbeauty to see the things generate in front of your eyes.

3.1 Particles at rest remain at rest

Spawning one particle and just waiting has the expected result of nothing changing. No surprises here.

3.2 Particles in orbit

Messing around a bit to find the right values to have particles in orbit we get particles that stay in orbit for a very long time.

3.3 Simulation with periodic boundary conditions

Generating a hundred thousand particles all of the same mass and letting them simulate gives the following results in figure 2.

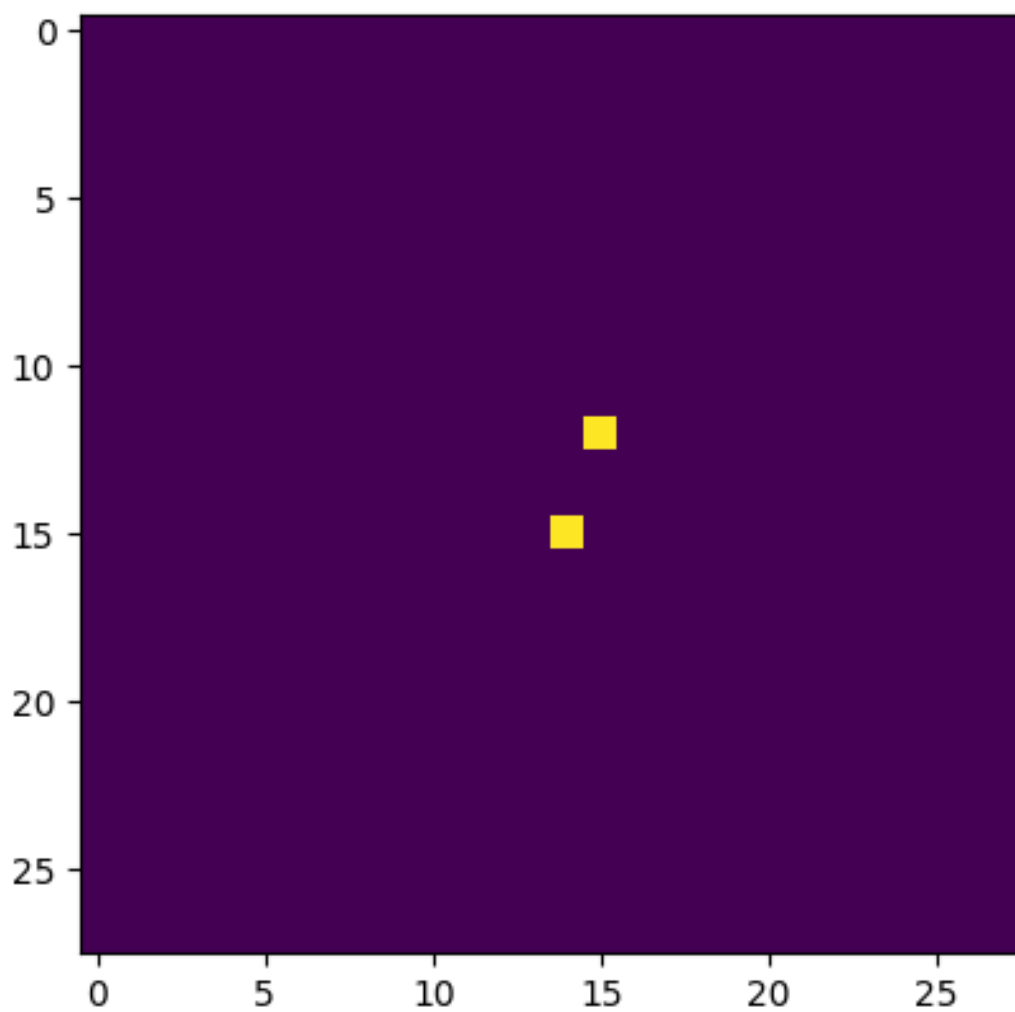


Figure 1: Still frame of orbiting particles

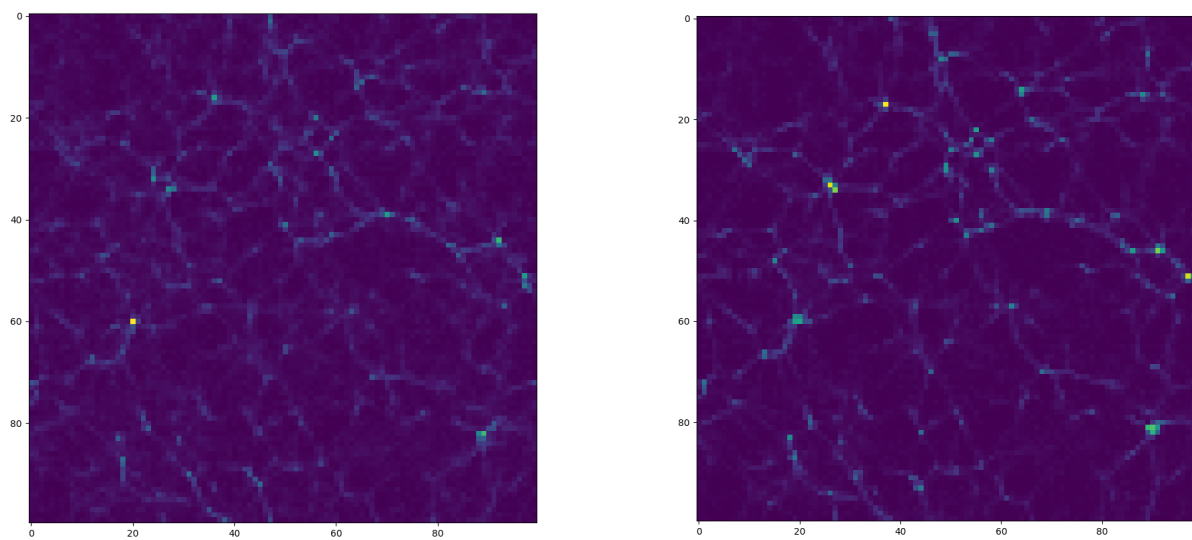


Figure 2: Uniform with circular boundary

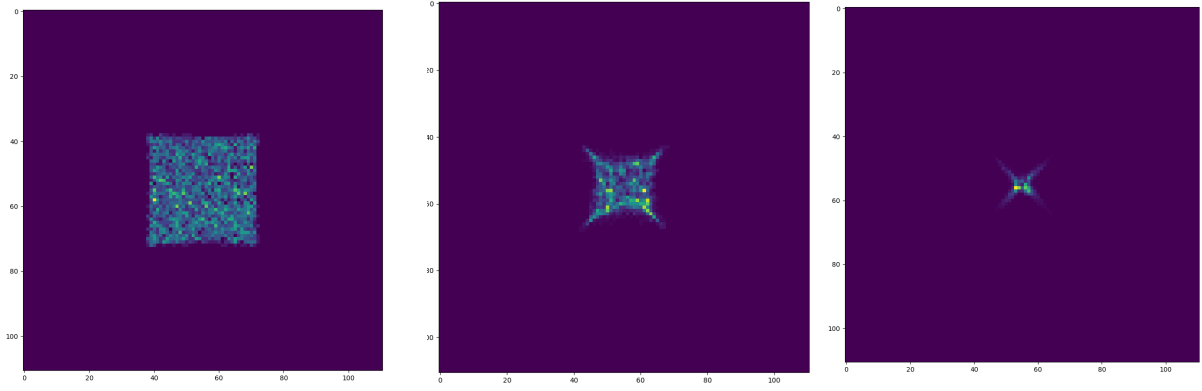


Figure 3: Uniform with hard boundary

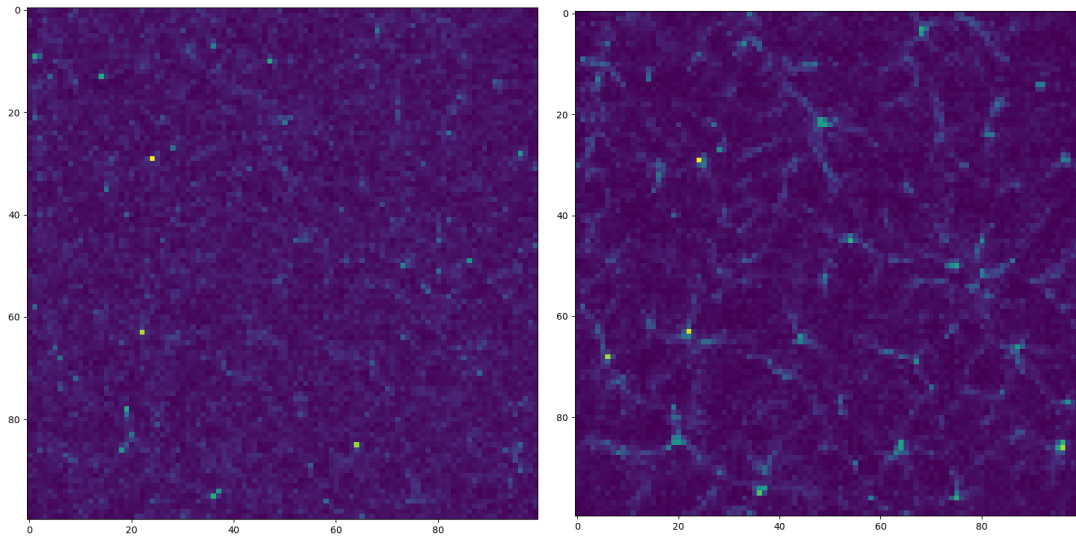


Figure 4: Universe Simulation

3.4 Simulation with hard boundary conditions

With a hard boundary it looks very different as seen in figure 3

As expected it collapse down (and then shoots back out but I didnt show that here). The "tails" that appear at the corners are a sort of proof that this simulation is actually 3d since what is happening is that the Euclidean distance from the corners to the center is larger so it takes them longer to get there!

Also the extra empty space is my way of creating padding as this prevents the wrap-around nature of the FFT from doing it's thing.

3.5 Universe mass distribution

Generating a realization of a power law distribution as was done in class we find that our universe looks about as expected as shown in figure 4

3.6 Final Thinks

Overall the energy was conserved rather well (a few percent) in all scenarios except for the hard boundary conditions where everything collapsed and softening messes with the energy conservation.

In any case hope you enjoyed and feel free to mess around with the code (it even supports arbitrary dimensionality (not sure how to interpret results but it just makes the space larger)).