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:

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

and then we can run a simulation using the following:

```
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:

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

```
    plt.clf()
    for pic in res["density"]:
        plt.ion()
        while len(pic.shape) > 2:
            pic = np.sum(pic, axis=-1)
        plt.imshow(pic)
        plt.show()
        plt.pause(0.1)
        plt.clf()
    plt.imshow(pic)
    plt.show()
    print("This movie experience has been brought to you by the Nbody movie company. \n
            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 scaler potential. This is achieved with the hash attribute of my class:

```
def hash(self):
    """
    main position to grid mass function, uses the numba thing above to go faster
    """

    #generate the grid
    N = int(np.ceil(1/self.ref))
    grid_shape = [N for _ in range(self.dim)]
    grid = np.zeros(grid_shape)

    #grid indexes from coordinates
    grid_indexii = self.pos_to_grid(self.pos)

    ##something about numba changing some dependancies and it dropping
    ##a warning which i couldnt be bothered to fix
    with warnings.catch_warnings():
        warnings.simplefilter("ignore")
        #call the actual hash function
        hash_grid = fast_hash(grid_indexii,grid_shape,self.m, grid)
    if self.BC == "wrap":
        #great its already circular so return
        return hash_grid
    elif self.BC == "hard":
        ##we need to pad
        pad_size = [[hash_grid.shape[i],hash_grid.shape[i]] for i in range(self.dim)]
        return np.pad(hash_grid, pad_size)
        #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:

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

following which it calls the numba worker function:

```
@njit
def fast_hash(grid_indexii, grid_shape, mass, grid):
    """
    Does a numba particle postion to grid desnity
    """
    for particle_num, grid_index in enumerate(grid_indexii):
        ##loop through all the particles:
        grid[grid_index[0], grid_index[1]] += mass[particle_num]
    return grid
```

which simply loops through each particle and adds it's 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:

```
def make_template(self):
    """
    A very tryhard function which generates the potential centered around a corner
    """

    #make the zeros template
    N = int(np.ceil(1/self.ref))
    grid_shape = [N for _ in range(self.dim)]
    template = np.zeros(grid_shape)
    template = np.zeros_like(self.grid, dtype=float)

    #generate a basis vector (this will not handle a non-square grid)
    #said basis vector is rolled to have zero at index zero
    # and then positive up to half way point and negative thereafter
    idx = np.roll(np.arange(-template.shape[0]//2, template.shape[0]//2, dtype=float),
                  template.shape[0]//2)

    #we need dimension many basis vectors
    basis_vec = np.array([idx for _ in range(self.dim)])

    #this now created the distance grid from all our basis vectors
    dist_grid = np.array(np.meshgrid(*basis_vec))


```

```
    #now rescale the dist grid
    dist_grid = dist_grid/np.max(dist_grid) # * self.grid_size

    #generate the template using the potential function
    #this is greens funciton
    #to catch the divide by zero
    self.k = 0.1
    with warnings.catch_warnings():
        warnings.simplefilter("ignore")
        template = self.pot_func(np.linalg.norm(dist_grid, axis=0, keepdims=False),self.k)

    #more softeing
    ind = [0 for i in range(self.dim)]
    ind_n = list(ind)
    ind_n[1] = 1
    template[tuple(ind)] = template[tuple(ind_n)]
    # plt.imshow(template)
    # plt.show()

    if self.BC == "wrap":
        return template
    elif self.BC == "hard":
        N = template.shape[0]
        template[N//3:2*N//3, N//3:2*N//3] = 0
        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:

```
@njit
def step_vel(self):
    """
    This is to compute the velocity from the field. A bit harder than the last
    We compute the vector field everywhere first and then evaluate the force individually

    when the boundary condition is non-periodic this function will be doing a lotttt of extra work
    might fix later
    """
    #compute a new potential and calculate the force to step velocity
    #this function does the convolution
    t1 = time.time()
    potential = self.make_pot()
    t2 = time.time()
```

```
    #me messing with things
    self.pot = potential

    ##now get force field:
    #we need to roll a little to the right and a little to the left and take the difference
    #this is a gradient operation which pushes it to another dimension for the different components

    #generate an array which is Ndim x grid_side^Ndim
    derivatives = np.zeros([self.pos.shape[1]] + list(self.grid.shape))

    #ok so this one is a bit of a crazy one.
    #this opp array is gonna be a vector [grid_side^Ndim-1, grid_size^Ndim-2, ..., 1]
    #it willl later be used for a crazy broadcasting operation
    from functools import reduce
    opp= np.array([reduce(lambda x,y: x*y, derivatives.shape[i:-1]) for i in range(1,
                   len(derivatives.shape)-1)] + [1])

    #get grid indexes
    grid_pos = self.pos_to_grid(self.pos)
    if self.BC == "hard":
        grid_pos = grid_pos + (np.array(self.grid.shape)//3)[np.newaxis, :]

    #this computes the gradiant stacking the dx,dy,dz... in axis=0 of derivatives
    t3 = time.time()
    for dim in range(self.pos.shape[1]):
        derivatives[dim, :] = (np.roll(potential, -1, axis=dim) - np.roll(potential, 1,
                               axis=dim))/(2/self.ref)
    t4=time.time()
    #print(t2-t1, t4-t3)
    #now for magic

    #now we flatten the first axis such that it is Ndim by however many (so flatten the coordinate)
    der_flat = derivatives.reshape(derivatives.shape[0], -1)

    #get the new flat index by dotting the grid position with the opp vector which converst to this new view
    indexes = np.dot(grid_pos, opp.T)

    #index the derivatives by the points of interest
    ordered_der = der_flat[:,indexes]

    #they happen to end up in a horizontal array which i dont like
    ordered_der = np.transpose(ordered_der)

    #update finally.....
    self.vel = self.vel - ordered_der*self.dt /self.m[:, np.newaxis]

    #save the derivatives:
    self.derivatives = derivatives
```

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

```
def make_pot(self):
    """
    Generate the potential from the mass distribution by convolving
    The BC should already be handled
    """
    mass_ft = sp.fft.rfftn(self.grid, workers = -1)
    my_pot_ft = mass_ft * self.pot_template_ft
    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 misserably and I was not motivated enough to rewrite it in C or in cupy which would reallly 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:

```
def step_pos(self, half_step = False):
    """
    Step the position with provisions for a euleur half step to get things going at the start
    """
    #compute the new positions from the velocity and then rehash the density map
    if half_step:
        self.pos = self.pos + self.vel*self.dt/2
        if self.BC == "wrap":
            #do the wrap around if particles left the grid
            self.pos = self.pos%1 # self.grid_size
        else:
            #hard walls will obviously not conseerver energy
            self.pos = np.clip(self.pos, 0, 1)
        #now that we have new position rehash
        self.grid = self.hash()
        return
    self.pos = self.pos + self.vel*self.dt
    ##for CIcrucla rBC only
    if self.BC == "wrap":
        #do the wrap around if particles left the grid
        self.pos = self.pos%1 #self.grid_size
    else:
        #hard walls will obviously not conseerver energy
```

```
        self.pos = np.clip(self.pos, 0, 1)
    #now that we have new position rehash
    self.grid = self.hash()
    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:

```
def run_sim(self, steps = 5000, frame_return = 25):
    """
    Actually run the sim and return all the frames
    """
    import time
    print("Running Sim, Depending on the grid size this can take a while....")
    print("expected time to run is:", 2e-7*self.grid.size*steps, "seconds")

    frames = []
    pos = []
    energy = []
    t1 = time.time()
    self.step_pos(half_step=True)
    for step_N in range(steps):
        #print(hello)
        t3=time.time()
        self.step_vel()
        t4=time.time()
        self.step_pos()
        t5=time.time()
        #print(t4-t3,t5-t4)
        if step_N%frame_return == 0:
            frames.append(self.grid)
            pos.append(self.pos)
            energy.append(self.calc_energy())
            pass
    t2 = time.time()
    print("Actually took:", ((t2-t1)), "seconds")
    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:

```
def calc_energy(self):
    ##compute the energy while we are at it
    kinetic = np.sum((sp.linalg.norm(self.vel, axis=1)**2)*self.m)
    potential = np.sum(self.pot* self.grid)/(self.grid.size)
    ##substract out the self induced potential
    ind = [0 for i in range(self.dim)]
    potential = potential - (np.sum(self.m)*self.pot_template[tuple(ind)])
    potential = potential*self.k
    #print("kinetic", kinetic, "potential", potential)
    #print(kinetic + potential)
    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 snipets 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.

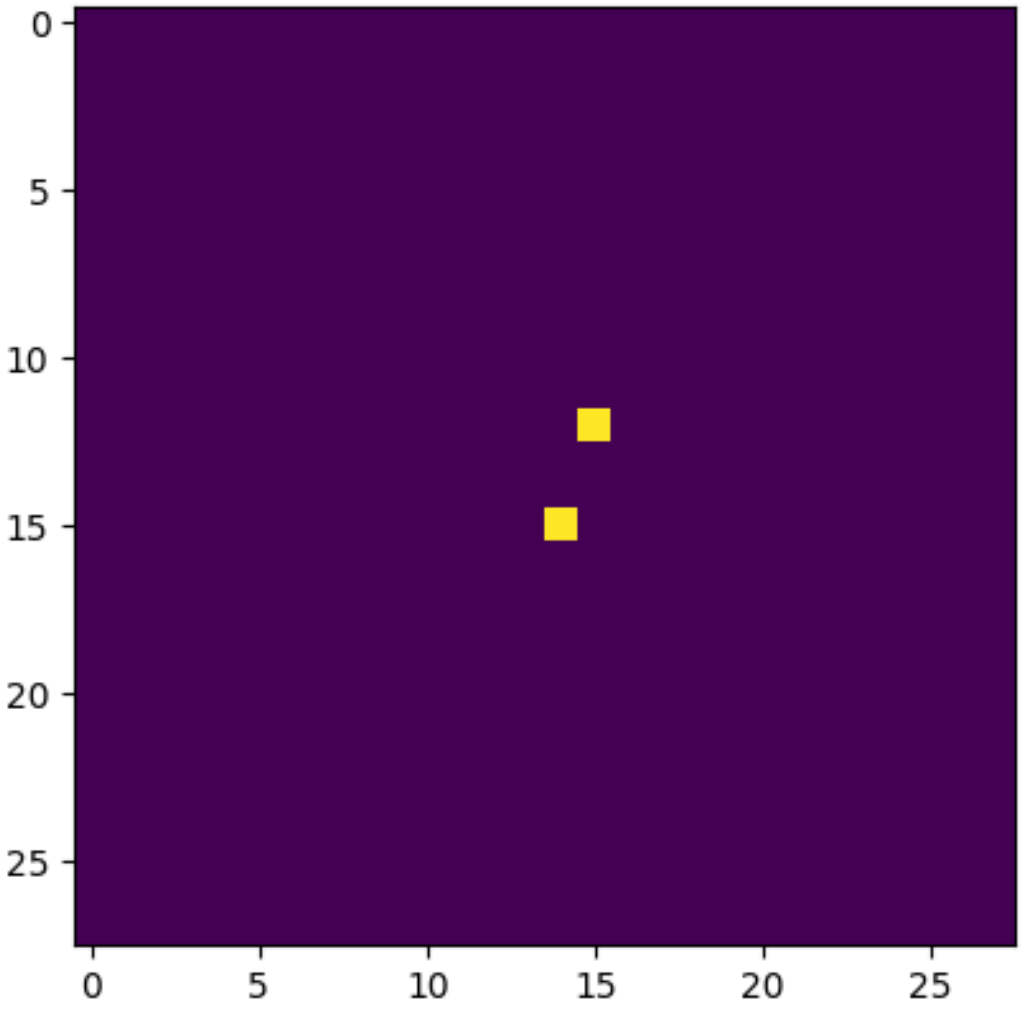

Figure 1: Still frame of orbiting particles

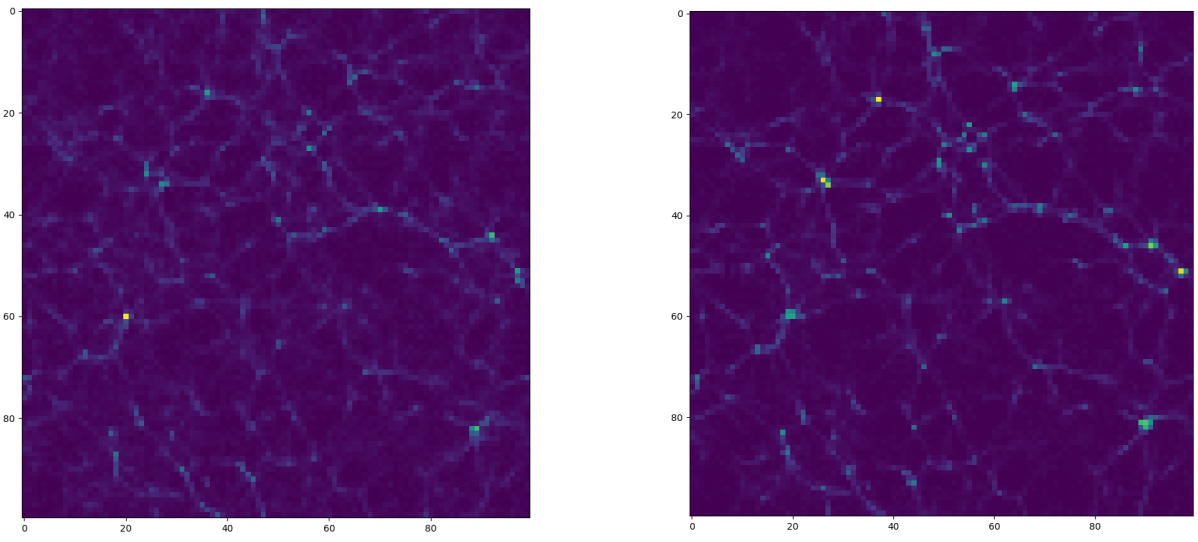

Figure 2: Uniform with circular boundary

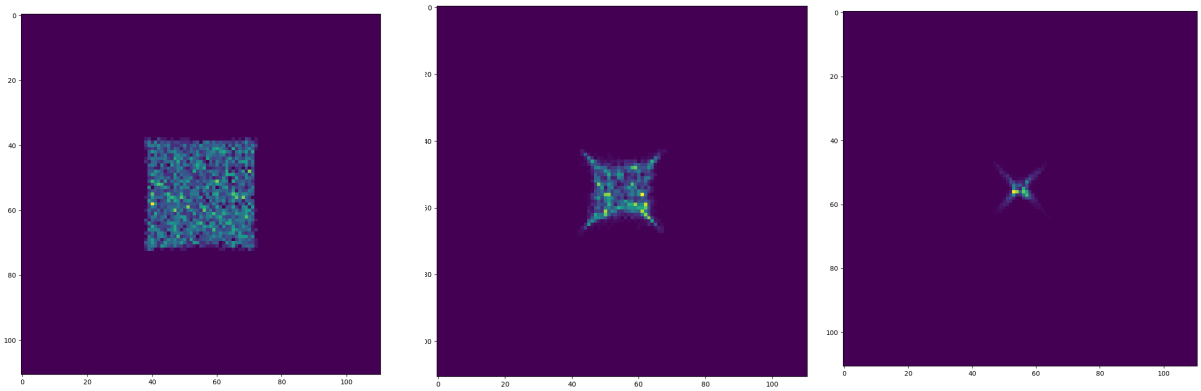

Figure 3: Uniform with hard boundary

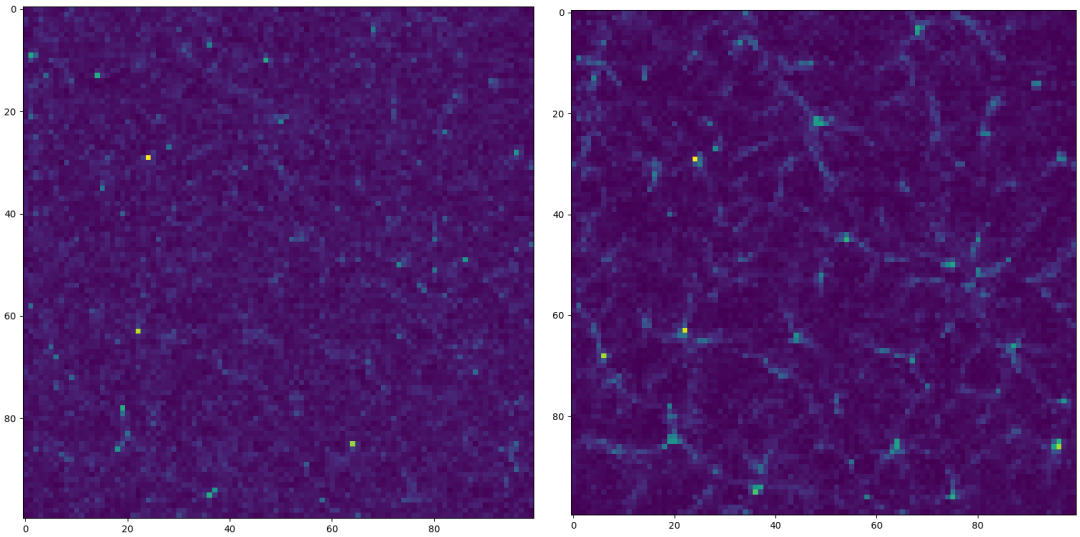

Figure 4: Universe Simulation

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