N-Body Simulator

```
import numpy as np
import matplotlib.pyplot as plt
import time
import numpy
import numpy.linalg
import numpy.random
import matplotlib
import matplotlib.pyplot
import matplotlib.mlab
import matplotlib.colors
import matplotlib.cm
import matplotlib.ticker
import matplotlib.patches
import matplotlib.collections
import matplotlib.path
import matplotlib.dates
import matplotlib.text
import matplotlib.lines
```

The plan:

We will write a class that can generate particle objects. This class will have methods to simulate the effect of gravity on its particles for a specified number of time steps.

To do so, we need to have many particles with different positions in 3-space and the ability to compute the following:

1. the distance to each other particle
2. the gravitational force exerted on them by all of the particles remembered in the pool
3. the new position and velocity after a time step

We will write the code to work in 3D. Each particle then has a position $\mathbf{r} = \begin{pmatrix} x \\ y \\ z \end{pmatrix}$ and a velocity $\mathbf{v} = \begin{pmatrix} v_x \\ v_y \\ v_z \end{pmatrix}$ for updates by Newton's law of gravitation:

$$\mathbf{F} = \sum_{\text{other particle other than this one } i} \frac{G m_i m_{\text{this}}}{|\mathbf{r}_i - \mathbf{r}_{\text{this}}|^2} \hat{(\mathbf{r}_i - \mathbf{r}_{\text{this}})}$$

1. Compute the force on each **body**
2. Update the position and velocity of each **body**

For accurate boundary conditions, we want the constraints to keep around L_{box} for a cubicle that would fill half in the coordinates $-L \cdots \frac{L}{2}$ and $\frac{L}{2} \cdots L$. Although, we want to be able to have the cubicle be $\begin{pmatrix} L \\ L \\ L \end{pmatrix}$ in 3-space, when computing the density, we want to have the cubicle be $\begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}$ in 3-space so that the density is not dependent on the box size. We will write the code to work in the cubicle $-L \cdots \frac{L}{2}$ and $\frac{L}{2} \cdots L$ and then when we want to find the density, we will multiply the coordinates by $\frac{1}{L}$ before we do the integrating.

For the simulation, we, for the same process, but instead of keeping the coordinates, we will the interaction of particles outside the box with those inside the box. We will do this by creating ghost particles that are copies of the original particles. We then perform the integration of motion. The function **update_positions_and_velocities()** will contain inside the grid:

1. Compute the potential **get_potential()**

The potential is calculated when it is

$$\Phi(\mathbf{r}) = -\sum_{\text{particle } i} \frac{G m_i}{|\mathbf{r} - \mathbf{r}_i|}$$

so for 3 particles, we have that

$$\Phi(\mathbf{r}) = -\frac{G m_1}{|\mathbf{r} - \mathbf{r}_1|} - \frac{G m_2}{|\mathbf{r} - \mathbf{r}_2|} - \frac{G m_3}{|\mathbf{r} - \mathbf{r}_3|}$$

Remember in terms of the density distribution we "smooth out" the masses and get rather:

$$\Phi(\mathbf{r}) = -\int \frac{G \rho(\mathbf{r}')}{|\mathbf{r} - \mathbf{r}'|} d^3\mathbf{r}'$$

This is a convolution, $\Phi_{\text{grid}} = \rho_{\text{grid}} * \frac{1}{|\mathbf{r}|}$. Hence, we can compute it with the help of Fourier transforms. In the code,

```
phi = ifftn(fftn(rho_grid)*fftn(1/(r+epsilon)))
```

Note that ρ_{grid} is $N^3 \times N^3$ since we are taking the Fourier transform of the entire box.

2. Compute the forces **get_force()**

For accurate forces, we need to compute the gradient of our potential. Our particles live in continuous inspace but we will place them onto the nodes of the surrounding cubicle. To get an estimate of the gradient at the location of any particle, we will take a weighted average of the surrounding potentials. weighting is the distance of our point to each corner of our point is easily seen to be the volume of the cube opposite to the corner. For example, the force in the x-direction is (in 2D for simplicity):

$$F_x(x, y) = F_x(0, 0) \cdot (L_x - x)(L_y - y) + F_x(L_x, 0) \cdot (x)(L_y - y) + F_x(0, L_y) \cdot (L_x - x)(y) + F_x(L_x, L_y) \cdot (x)(y)$$

Or

$$\mathbf{F}(\mathbf{r}) = \sum_{\text{corner } i} \mathbf{F}_i(\mathbf{r}_i) \cdot (L_x - |x - x_i|)(L_y - |y - y_i|)(L_z - |z - z_i|)$$

get_gradient performs the calculation. Note that it also computes the gradient accurately on if a particle is on a boundary, we take its weighted in the box like on the other side of the grid. However, we only simulate the gradient for particles with non-zero mass, so in the case that there is a particle on the boundary with zero mass, we treat it as if there is no force on it. If, and that problem is not crucial, this case must be ignored, so we leave the force at zero.

3. Move forward in time **step_forward()**

To update the positions and velocities, we are using the system:

$$\begin{aligned} \mathbf{v}(t + \Delta t) &= \mathbf{v}(t) + \mathbf{a}(t) \Delta t \\ \mathbf{r}(t + \Delta t) &= \mathbf{r}(t) + \mathbf{v}(t + \Delta t) \Delta t \end{aligned}$$

We can take a step in time using a leapfrog algorithm.

$$\begin{aligned} \mathbf{v}(t + \frac{1}{2}\Delta t) &= \mathbf{v}(t - \frac{1}{2}\Delta t) + \mathbf{a}(t) \Delta t \\ \mathbf{r}(t + \Delta t) &= \mathbf{r}(t) + \mathbf{v}(t + \frac{1}{2}\Delta t) \Delta t \end{aligned}$$

So we update our position with our current velocities, incorporate no forces, and update our velocities with those forces. We can iterate this to get the evolution of our system.

Initial conditions

For initial conditions, we can either set them to hand-in grid coordinates when initializing our system, or generate them for arbitrary locations in the "real space" and then convert them to grid coordinates.

The functions **convert_to_grid_coords** takes this conversion. Lastly **get_grid_idx**. This takes the range of the grid to be the lower and upper bounds of the grid and returns the indices of the grid that a particle would be in if it were in the specified coordinates. Finally, the initialization function, **initialize_particles()** takes in the number of particles and the box size and initializes particles randomly in the box.

Helper functions

Here are the helper functions that are called during the main class described above.

```
def get_density(ts=0):
    ax = plt.gca(projection='3d')
    ax.clear()
    ax.scatter(xs, ys, zs,
               c='r',
               s=10,
               alpha=0.5)
    ax.set_xlim(-L/2, L/2)
    ax.set_ylim(-L/2, L/2)
    ax.set_zlim(-L/2, L/2)
    ax.set_xlabel('x')
    ax.set_ylabel('y')
    ax.set_zlabel('z')
    ax.view_init(elev=20, azim=ts)
    plt.draw()
    plt.pause(.001)

for ts in range(5):
    print('iter: {}'.format(ts))
    pool = pool.update_accelerations()
    pool = pool.update_positions_and_velocities(dt=0.1)
    xs = pool.positions[:,0]
    ys = pool.positions[:,1]
    zs = pool.positions[:,2]
    print('xrms: {}'.format(np.sqrt(np.mean(xs**2))))
    print('yrms: {}'.format(np.sqrt(np.mean(ys**2))))
    print('zrms: {}'.format(np.sqrt(np.mean(zs**2))))
    print('')
    get_density(ts=ts)

xs = pool.positions[:,0]
ys = pool.positions[:,1]
zs = pool.positions[:,2]
print('xrms: {}'.format(np.sqrt(np.mean(xs**2))))
print('yrms: {}'.format(np.sqrt(np.mean(ys**2))))
print('zrms: {}'.format(np.sqrt(np.mean(zs**2))))
print('')
get_density(ts=ts)

def compute_coordinate_mesh(L=1):
    x = np.linspace(-L/2, L/2, 100)
    y = np.linspace(-L/2, L/2, 100)
    z = np.linspace(-L/2, L/2, 100)
    return np.meshgrid(x, y, z)

def plot_field(L=1, Nmesh=100):
    ax = plt.gca(projection='3d')
    ax.clear()
    ax.scatter(xs, ys, zs,
               c='r',
               s=10,
               alpha=0.5)
    ax.set_xlim(-L/2, L/2)
    ax.set_ylim(-L/2, L/2)
    ax.set_zlim(-L/2, L/2)
    ax.set_xlabel('x')
    ax.set_ylabel('y')
    ax.set_zlabel('z')
    ax.view_init(elev=20, azim=0)
    plt.draw()
    plt.pause(.001)

def get_density(ts=0):
    ax = plt.gca(projection='3d')
    ax.clear()
    ax.scatter(xs, ys, zs,
               c='r',
               s=10,
               alpha=0.5)
    ax.set_xlim(-L/2, L/2)
    ax.set_ylim(-L/2, L/2)
    ax.set_zlim(-L/2, L/2)
    ax.set_xlabel('x')
    ax.set_ylabel('y')
    ax.set_zlabel('z')
    ax.view_init(elev=20, azim=ts)
    plt.draw()
    plt.pause(.001)

def get_potential(Nmesh=100, G=1, rho=None):
    if rho is None:
        rho = self.get_density(Nmesh=Nmesh)
    print('rho shape: {}'.format(rho.shape))
    epsilon = 1e-5
    x = np.linspace(-L/2, L/2, Nmesh)
    y = np.linspace(-L/2, L/2, Nmesh)
    z = np.linspace(-L/2, L/2, Nmesh)
    print('x shape: {}'.format(x.shape))
    print('y shape: {}'.format(y.shape))
    print('z shape: {}'.format(z.shape))
    r = (x[:,numpy.newaxis,numpy.newaxis]**2 + y[numpy.newaxis,:,numpy.newaxis]**2 + z[numpy.newaxis,numpy.newaxis,:]**2)**(1/2)
    print('r shape: {}'.format(r.shape))
    print('rho shape: {}'.format(rho.shape))
    phi = ifftn(fftn(rho)*fftn(1/(r+epsilon)))
    print('phi shape: {}'.format(phi.shape))
    return phi

def get_force(self,Nmesh=100,G=1,rho=None,phi=None):
    if rho is None:
        rho = self.get_density(Nmesh=Nmesh)
    if phi is None:
        phi = self.get_potential(Nmesh=Nmesh,G=G,rho=rho)
    print('phi shape: {}'.format(phi.shape))
    gx,gy,gz = numpy.gradient(phi)
    print('gx shape: {}'.format(gx.shape))
    print('gy shape: {}'.format(gy.shape))
    print('gz shape: {}'.format(gz.shape))
    return -gx,-gy,-gz

def get_force(self,pos,Nmesh=100,G=1,rho=None,phi=None):
    if rho is None:
        rho = self.get_density(Nmesh=Nmesh)
    if phi is None:
        phi = self.get_potential(Nmesh=Nmesh,G=G,rho=rho)
    gx,gy,gz = self.get_force(Nmesh=Nmesh,G=G,rho=rho,phi=phi)
    print('gx shape: {}'.format(gx.shape))
    print('gy shape: {}'.format(gy.shape))
    print('gz shape: {}'.format(gz.shape))
    print('pos shape: {}'.format(pos.shape))
    x = np.linspace(-L/2, L/2, Nmesh)
    y = np.linspace(-L/2, L/2, Nmesh)
    z = np.linspace(-L/2, L/2, Nmesh)
    print('x shape: {}'.format(x.shape))
    print('y shape: {}'.format(y.shape))
    print('z shape: {}'.format(z.shape))
    fx = numpy.zeros(pos.shape[0])
    fy = numpy.zeros(pos.shape[0])
    fz = numpy.zeros(pos.shape[0])
    for i in range(pos.shape[0]):
        print('i: {}'.format(i))
        ix = (numpy.abs(x - pos[i,0])).argmin()
        iy = (numpy.abs(y - pos[i,1])).argmin()
        iz = (numpy.abs(z - pos[i,2])).argmin()
        print('ix shape: {}'.format(ix.shape))
        print('iy shape: {}'.format(iy.shape))
        print('iz shape: {}'.format(iz.shape))
        fx[i] = gx[ix,iy,iz]
        fy[i] = gy[ix,iy,iz]
        fz[i] = gz[ix,iy,iz]
    return fx,fy,fz

def update_accelerations(self,Nmesh=100,G=1):
    print('updating accelerations')
    pos = self.positions
    fx,fy,fz = self.get_force(pos,Nmesh=Nmesh,G=G)
    print('fx shape: {}'.format(fx.shape))
    print('fy shape: {}'.format(fy.shape))
    print('fz shape: {}'.format(fz.shape))
    ax = fx/self.masses
    ay = fy/self.masses
    az = fz/self.masses
    print('ax shape: {}'.format(ax.shape))
    print('ay shape: {}'.format(ay.shape))
    print('az shape: {}'.format(az.shape))
    self.accelerations = ax,ay,az
    print('self.accelerations[0].shape: {}'.format(self.accelerations[0].shape))
    return self

def update_positions_and_velocities(self,dt=0.1):
    print('updating positions and velocities')
    ax,ay,az = self.accelerations
    px,py,pz = self.positions
    vx,vy,vz = self.velocities
    px = px + vx*dt + ax*dt**2/2
    py = py + vy*dt + ay*dt**2/2
    pz = pz + vz*dt + az*dt**2/2
    vx = vx + ax*dt
    vy = vy + ay*dt
    vz = vz + az*dt
    print('px shape: {}'.format(px.shape))
    print('py shape: {}'.format(py.shape))
    print('pz shape: {}'.format(pz.shape))
    print('vx shape: {}'.format(vx.shape))
    print('vy shape: {}'.format(vy.shape))
    print('vz shape: {}'.format(vz.shape))
    self.positions = px,py,pz
    self.velocities = vx,vy,vz
    print('self.positions[0].shape: {}'.format(self.positions[0].shape))
    return self

def compute_coordinate_mesh(L=1):
    x = np.linspace(-L/2, L/2, 100)
    y = np.linspace(-L/2, L/2, 100)
    z = np.linspace(-L/2, L/2, 100)
    print('x shape: {}'.format(x.shape))
    print('y shape: {}'.format(y.shape))
    print('z shape: {}'.format(z.shape))
    return np.meshgrid(x, y, z)

def convert_to_grid_coords(self,L=1,Nmesh=100):
    print('converting to grid coordinates')
    px,py,pz = self.positions
    print('px shape: {}'.format(px.shape))
    print('py shape: {}'.format(py.shape))
    print('pz shape: {}'.format(pz.shape))
    ix = (px/L*Nmesh).astype(int)
    iy = (py/L*Nmesh).astype(int)
    iz = (pz/L*Nmesh).astype(int)
    print('ix shape: {}'.format(ix.shape))
    print('iy shape: {}'.format(iy.shape))
    print('iz shape: {}'.format(iz.shape))
    return ix,iy,iz

def get_grid_idx(self,L=1,Nmesh=100):
    print('getting grid indices')
    ix,iy,iz = self.convert_to_grid_coords(L=L,Nmesh=Nmesh)
    print('ix shape: {}'.format(ix.shape))
    print('iy shape: {}'.format(iy.shape))
    print('iz shape: {}'.format(iz.shape))
    return ix,iy,iz

def initialize_particles(self,N=10,L=1):
    print('initializing particles')
    px = (numpy.random.rand(N) - 0.5)*L
    py = (numpy.random.rand(N) - 0.5)*L
    pz = (numpy.random.rand(N) - 0.5)*L
    print('px shape: {}'.format(px.shape))
    print('py shape: {}'.format(py.shape))
    print('pz shape: {}'.format(pz.shape))
    self.positions = px,py,pz
    vx = numpy.zeros(px.shape[0])
    vy = numpy.zeros(px.shape[0])
    vz = numpy.zeros(px.shape[0])
    print('vx shape: {}'.format(vx.shape))
    print('vy shape: {}'.format(vy.shape))
    print('vz shape: {}'.format(vz.shape))
    self.velocities = vx,vy,vz
    m = numpy.ones(px.shape[0])
    print('m shape: {}'.format(m.shape))
    self.masses = m
    print('self.positions[0].shape: {}'.format(self.positions[0].shape))
    print('self.velocities[0].shape: {}'.format(self.velocities[0].shape))
    print('self.masses.shape: {}'.format(self.masses.shape))
    return self
```

Animate the results:

To make a movie of the system, we have to keep **plot_field** and not just let the results at each step. Do that over all of these figures and then use the command line utility **convert** to make a movie out of the .jpeg files. Use **mogrify** to make them smaller.

```
pool = pool.update_accelerations()
pool = pool.update_positions_and_velocities(dt=0.1)
xs = pool.positions[:,0]
ys = pool.positions[:,1]
zs = pool.positions[:,2]
print('xrms: {}'.format(np.sqrt(np.mean(xs**2))))
print('yrms: {}'.format(np.sqrt(np.mean(ys**2))))
print('zrms: {}'.format(np.sqrt(np.mean(zs**2))))
print('')
plot_field(L=1,Nmesh=100)

pool = pool.update_accelerations()
pool = pool.update_positions_and_velocities(dt=0.1)
xs = pool.positions[:,0]
ys = pool.positions[:,1]
zs = pool.positions[:,2]
print('xrms: {}'.format(np.sqrt(np.mean(xs**2))))
print('yrms: {}'.format(np.sqrt(np.mean(ys**2))))
print('zrms: {}'.format(np.sqrt(np.mean(zs**2))))
print('')
plot_field(L=1,Nmesh=100)
```

There's not much to see here, and we see the effect of numerical instability.

Case 2: Two particles in orbit:

```
pool = pool.initialize_particles(N=2,L=1)
pool.masses[0] = 1e3
pool.masses[1] = 1e3
pool.positions[0,0] = 0.25
pool.positions[1,0] = -0.25
pool.velocities[1,1] = 1e3
pool.velocities[0,1] = -1e3
xs = pool.positions[0,0]
ys = pool.positions[0,1]
zs = pool.positions[0,2]
pool = pool.update_accelerations()
pool = pool.update_positions_and_velocities(dt=0.1)
```

```
In [24]: plt.plot(t,
         np.sum(np.abs(potential(pos,M,G,eps) - f(t,pos,M,G,eps)), 1),
         plt.xlabel("Time", fontsize=20)
         plt.ylabel("Error", fontsize=20)
         plt.legend(["Total Energy"])
         plt.show()
```

```
Out[24]: <matplotlib.legend.Legend at 0x1a7bae4e20>
```

We see that the potential, W, has strong drop at bouncing when every particle is dumped so near the center(~bounce). So now we know that the error of the integrator is very high at bounce. We need to zoom in to see more clearly.

Implementation of RK4 integrator

Now, we implement a RK4 integrator. The integrator will calculate the derivatives based on the position of the particles at the current step. Unlike the Euler integrator, the RK4 will use the data from the previous step to calculate the derivatives. Because the integrator needs data from the previous step, we need to keep track of the previous step, so we will compute both using it. Now we have knowledge of the leading remainder term that the RK4 has. We will use it for 1000000 particles. That, let us look at a rough extrapolation to see if the error works.

```
In [25]: pos,vel = ic(N,M,G,R,seed=1)
         pos,vel,W,Wn = RK4(f,potential,t,pos,vel,M,G,eps)
         print(W,Wn)
         plt.plot(pos[:,0],pos[:,1],'.')
         plt.xlabel("x")
         plt.ylabel("y")
         plt.title("Initial Condition")
         plt.axis('equal')
         plt.show()
```

```
-0.5000000000000004 -0.5000000000000004
```



```
In [26]: tmax, dt = 1.0, 0.01
         ic = lambda N,M,G,R,seed=1: ic(N,M,G,R,seed=seed)
         integrator = RK4
         potential = potential
         plot_energy(N,M,G,R,tmax,dt,ic,integrator,potential,eps)
```

```
Run 1 in 1.1e0 seconds(!!)
  - particle time step limit: 1.1 (1%)
  - energy conservation: 0.0
```

```
In [27]: rms_E = lambda energy:np.sqrt(np.sum(energy**2))
```

```
In [28]: fig,axes = plt.subplots(2,2,sharex=True)
         ic = lambda N,M,G,R,seed=1: ic(N,M,G,R,seed=seed)
         plot_energy(N,M,G,R,tmax,dt,ic,RK4,potential,eps,ax=axes[0,0])
         plot_energy(N,M,G,R,tmax,dt,ic,leapfrog,potential,eps,ax=axes[0,1])
         axes[0,0].set_title("RK4")
         axes[0,1].set_title("Leapfrog")
         plt.suptitle("Comparison of RK4 and Leapfrog Methods")
         plt.tight_layout()
```



```
Out[28]: <matplotlib.legend.Legend at 0x1a7bae4e20>
```

There is some discrepancy between both energies. Though it seems that the RK algorithm distributes a little lot worse than the leapfrog algorithm, which suggests that the rk4 conserves energy better than the leapfrog algorithm.

Some Cool Animations

To finish, here are some satisfying animations.

```
In [29]: fig,axes = plt.subplots(2,2,sharex=True,sharey=True)
         ic = lambda N,M,G,R,seed=1: ic(N,M,G,R,seed=seed)
         plot_density(N,M,G,R,tmax,dt,ic,RK4,potential,eps,ax=axes[0,0])
         plot_density(N,M,G,R,tmax,dt,ic,leapfrog,potential,eps,ax=axes[0,1])
         plot_density(N,M,G,R,tmax,dt,ic,RK4,potential,eps,ax=axes[1,0])
         plot_density(N,M,G,R,tmax,dt,ic,leapfrog,potential,eps,ax=axes[1,1])
```


Lines: the particles start with a density inversely proportional to their distance to the center of the grid, so, some of the particles near the center get flung outward.

```
In [30]:
```