2D Vlasov-Poisson Equation by PIC Algorithm

Javier Palomares

December 14, 2012

1 The Vlasov Poisson Equation

$$\frac{\partial f}{\partial t} + \frac{\mathbf{p}}{m} \cdot \nabla f + \mathbf{F} \cdot \frac{\partial f}{\partial \mathbf{p}} = 0$$

I choose to study the **Vlasov Poisson equation** for my final project. Although we had already covered this system in class, it was only done in one dimension. I chose to extend the techniques we saw in class to two dimensions and take a look at various problems that can be solved.

The Vlasov equation describes the statistical behavior of the **probability density function** under the influence of a force $\mathbf{F}$ in the absence of particle collisions. The Vlasov equation is used in systems with number of particles too large to be modeled by deterministic methods.

The **probability density function** $f(\mathbf{r}, \mathbf{p}, t)$ is defined as the density of particles with position $\mathbf{r}$ and momentum $\mathbf{p}$ at a time t:

$$dN = f(\mathbf{r}, \mathbf{p}, t)\, d^3\mathbf{r}\, d^3\mathbf{p}$$

Combining this equation with the **Poisson equation**

$$\Delta\phi = \rho$$

Allows us to compute the central forces between particles by measuring the mass/charge density. Note that this can only be done for central forces that can be taken as the gradient of a potential. This cannot be done for non-central forces such as magnetic ones.

2 Numerical Techniques

The **PIC** (particle in cell) algorithm takes in the initial conditions of the system and diffuses the mass/charge of each particle symmetrically on a grid. Doing this allows us to quickly calculate the density of the system at all times in order to solve Poisson's equation.

I solved Poisson's equation with Fourier transforms. If we think of the density and potential as the transform of some function:

$$\phi(\mathbf{r}) = \int \Phi(\mathbf{k})\ e^{2\pi i \mathbf{k}\cdot\mathbf{r}} d^3\mathbf{k}, \quad \rho(\mathbf{r}) = \int f(\mathbf{k})\ e^{2\pi i \mathbf{k}\cdot\mathbf{r}} d^3\mathbf{k}$$

we find that

$$\phi(\mathbf{k}) = \frac{1}{(2\pi)^3}\ \frac{f(\mathbf{k})}{k^2}.$$

This allows us to find the potential by taking the fourier transform of the density, dividing by $-k^2$, and taking the inverse transform.Fourier transforms were computed using the numpy library's **FFT** function.

Once we have the potential, it's a straightfoward process to take its gradient to find the forces on particles, ($\mathbf{F} = -\nabla\phi$) and update velocities and positions using Euler methods or other integration techniques.

I chose to integrate by using **Leapfrog integration**, where the position is update by half a time step, the acceleration is calculated and used to update the velocity, and the position is once again updated by half a time step on each iteration:

$$\mathbf{r}_n = \mathbf{r}_{n-1} + \frac{dt}{2}\mathbf{v}_{n-1/2}$$

$$\mathbf{a} = \frac{\mathbf{F}}{m}$$

$$\mathbf{v}_{n+1/2} = \mathbf{v}_{n-1/2} + \mathbf{a}_n dt$$

At this stage we are able to model the behavior of a large system of particles as they interact due to the self-interacting forces such as gravity or electrical forces.

3 Simulations

After implementing the code to solve the general Vlasov Poisson equation, I took a closer look at particular some particular initial conditions and watched how they evolve.

3.1 Galactical simulation

It is believed that early in the universe, cold, slow moving particles clumped together to form celestial bodies. Various cosmological simulations,such as Eris at the UC Santa Cruz, have looked at this phenomenom. I decided to look further into this problem.

First, I set up a uniform distribution of 10000 total particles with zero velocity on a 100×100 grid with periodic boundary conditions. This starting state can be interpreted as diffuse celestial bodies uniformly distributed throughout space with no external forces. :

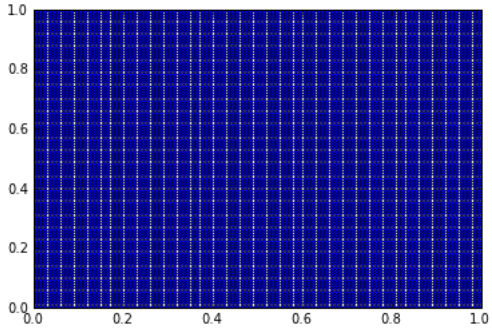

Figure 1: Initial(t=0) uniform array of 10000 particles on a grid with periodic boundaries. All particles were given an equal mass. No external forces were applied to the system, so the only forces acting on any particle within the system was the gravitational attraction to other particles. The system was then evolved in time.

This initial condition has a mean /bf correlation of -3.816×10^{-20}. This value is essentially negligible as the particles from a uniform array with no bunching.

The system was then allowed to evolve in time according to the gravitational forces, and after a long time the systems' state looked as follows:

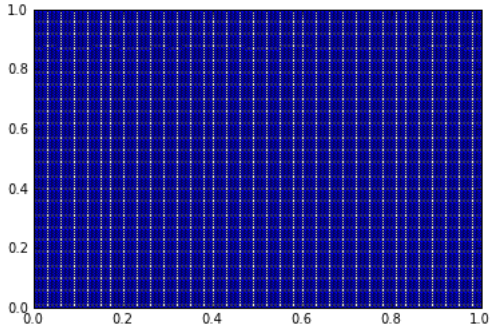

Figure 2: After evolving for a long time (t=25), the system looks just like its initial state. This is not an error, as we will see the net force on any of the particles was zero, so the particles remained stationary.

Once again, the system had a mean correlation of -3.816×10^{-20}.This is exactly the same the same conditions that we saw at t=0! Surprisingly, this does make sense: The probability density is uniform and due to the periodic boundary conditions, each particle senses an equal force in all directions, resulting in a zero net force. Thus the particles will stay in this uniform distribution for all time. We can make sense of this once more if we look at the potential, which is uniform since the density is uniform everywhere. Since the force $\mathbf{F}$ and potential ϕ are related by$\mathbf{F} = -\nabla\phi$ we see that a uniform potential leads to a zero force.

I next took a look at an almost identical situation: 10000 particles on a 100×100 grid as we saw in Figure 1. However, this time I perturbed the position of each particle by a Gaussian distribution, and now each particle had a small velocity. Moreover, this time each particle did not have an equal mass.

This resulted in the following initial state with a mean correlation of $5.47250165596 \times 10^{-07}$.

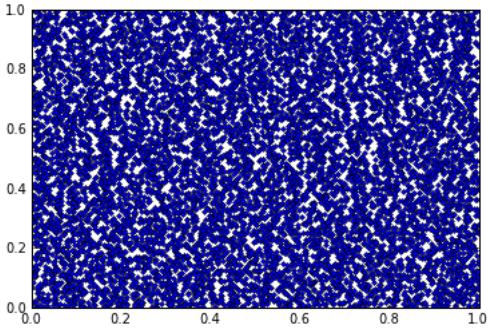

Figure 3: The perturbing distribution had a mean at zero to allow positive and negative perturbations in the $\vec{x}$ and $\vec{y}$ directions, and a standard deviation of .0001 (4 orders of magnitude smaller than the grid size). The velocities in the $\vec{x}$ and $\vec{y}$ directions were also randomly selected from this distribution.

This time we see a significantly higher initial mean correlation. This is because by pure chance, the perturbations have pushed some particles closer to each other. Thus the correlation function, which can be interpreted as a measure of how bunched up particles are, has increased. However, we still see that the particles are diffused throughout the entire grid. But if we allow this system to evolve in time, we begin to see some much more interesting results:

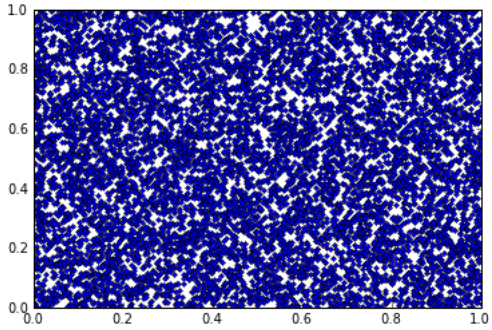

Figure 4: Time: 4.0. Mean Correlation: $1.77943674067 \times 10^{-05}$. The mean correlation has increased. The initial state did not have a uniform distribution, nor a uniform potential, so particles felt net forces causing the particles to bunch up.

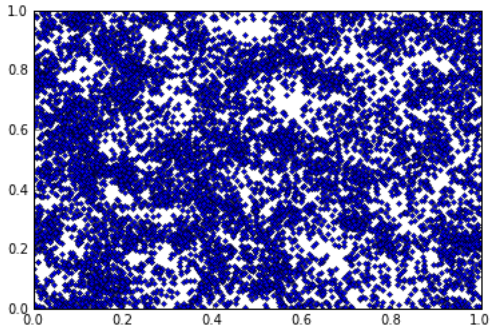

Figure 5: t: 6.0 Mean Correlation: 0.000889370887301. The correlation is significantly higher now than that of the start state. The gravitational attraction has brought particles closer to each other.

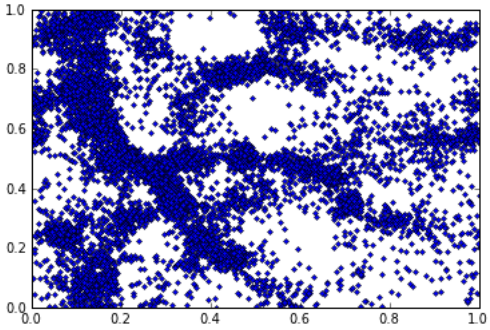

Figure 6: t: 8.0 Mean Correlation: 0.0164899352445. The correlation continues to increase. The particles no longer diffuse through the entire grid space.

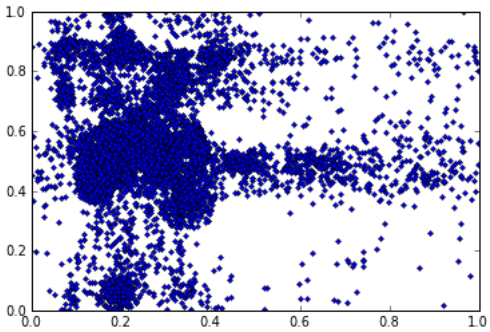

Figure 7: t: 10.0 Mean Correlation: 0.0360741148919. Particles continue to attract each other, forming a **galaxy**

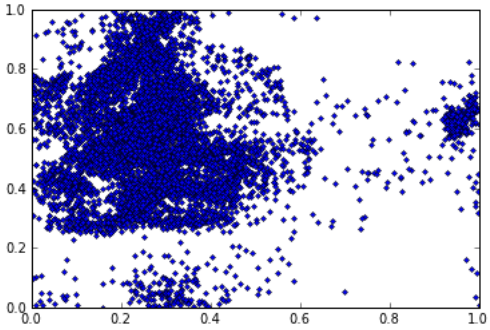

Figure 8: t: 12.0 Mean Correlation: 0.0381762366398. The particles have bunched up into a galaxy. Although not shown here, particles within a galaxy are not stationary. Although they are bound by the gravitational attraction, they have nonzero velocities and oscillate about the center of mass, much in the same way that celestial bodies within a body oscillate about the center of mass.

As we have seen, with this initial state, the particles are not static. This is because any time perturbation breaks the symmetry of the system, so the potential is now never uniform. Therefore, particles experience net forces until the form a stable galaxy. We are able to quantify the bunching of the particles by the correlation function. It also important to note that particles in the galaxy do not approach stationary states. Instead, they find stable oscillatory paths. This is all consistent with astronomical measurements, which helps to validate theories about the early universe. However, it is important to note that the number of particles in this simulation is much to small to make any claims. The Eris simulation ran 60 million particles for 8 months on a supercomputer.

3.2 Orbits

Next, I decided to take a look at particles orbiting about a fixed point. First, I set up a system of 1000 particles orbiting about a particle within a range of radius. I fixed the angular velocity of each particle to remain in a stable orbit, and gave each particle equal mass.

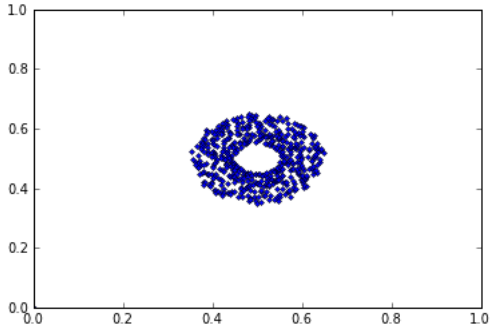

Figure 9: t: 0.0 The particles are given angular velocities in order to stay in a stable orbit about a central point at (.5,.5)

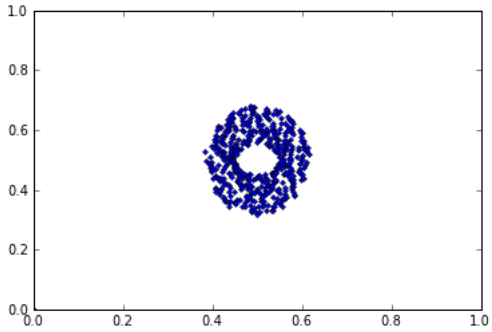

Figure 10: t: 40.0 We see that the particles stay in orbit about the central point

I next took the same initial condition and drastically altered the initial conditions by introducing a particle at the central point with a mass much more massive than the rest of the particles, leading to a much bigger force than that keeping the particles in stable orbit.

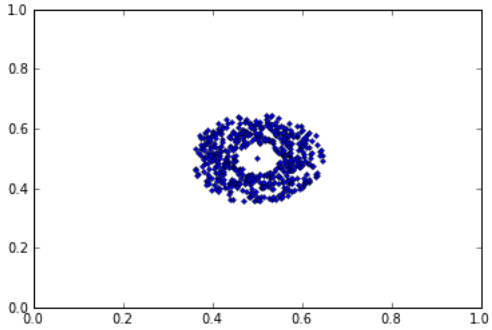

Figure 11: t: 0.0. A much more massive particle is introduced at the center of orbit, and the system is allowed to evolve

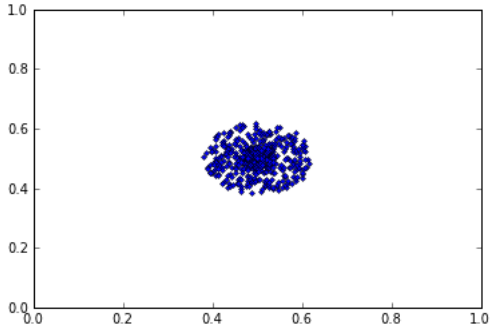

Figure 12: t: 20.0. The much more heavy particle begins to collapse the rest of the particles towards it

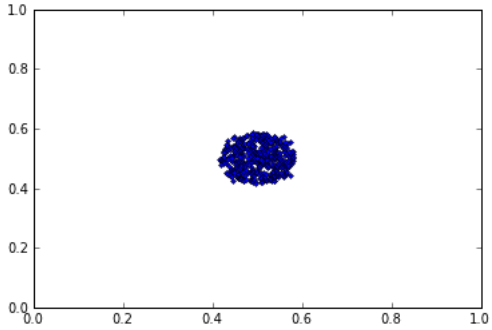

Figure 13: t: 30.0

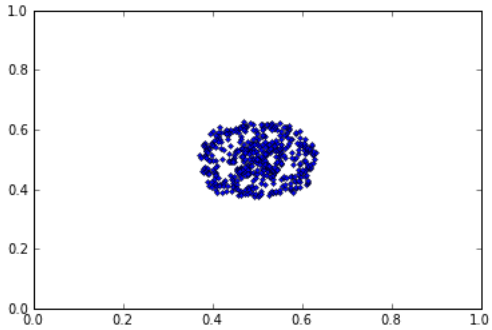

Figure 14: t: 40.0

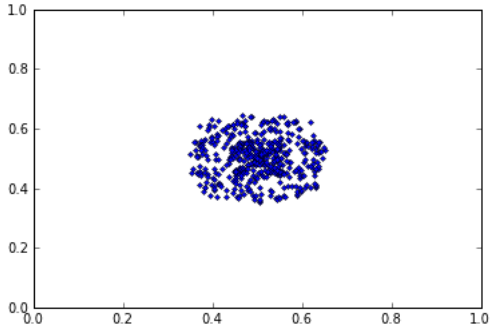

Figure 15: t: 50.0

We see that the introduction of the massive particle collapses the stability of the particles' orbits, and after a long time, they begin to oscillate linearly about the massive particle, practically ignoring the attraction due to other particles, as the attraction with the central particle is much stronger. Thus we see that if the solar mass were to suddenly increase, planets would no longer have elliptical orbits, and instead planets would oscillate to and from the sun instead of about it.

4 Conclusions

The Vlasov-Poisson equation is an effective technique for systems with large number of particles. Moreover, it is sufficiently accurate for most purposes and its computational intensity is linear to the number of particles, which makes it effective for up very large number of particles.

Although I only looked at a few cases, there are many other problems which it can solve. It can easily be changed to model electrostatics or fluid dynamics.

I take the rest of the paper as references and to post the python code I used.

5 References

http://wiki.tomabel.org
http://en.wikipedia.org/wiki/Correlation_function
http://en.wikipedia.org/wiki/Vlasov_equation
http://en.wikipedia.org/wiki/Eris_simulation Special Thanks to Professor Abel and Yao

6 Phython Code

6.1 Final_Project _Library

```
# Library for Vlasov Poisson equation in 2D # Author: Javier Palomares
import math
    from pylab import
    from numpy import
```

```
# ¡codecell¿
def getParticlePositions(Ngrid):
dx = 1.0/Ngrid
x = np.arange(0,1,dx) + .5 * dx
y = np.arange(0,1,dx) + .5 * dx
r = np.zeros((Ngrid,2))
r[:,0] = x
r[:,1] = y
return r
def getUniformParticlePositions(Ngrid):
dx = 1.0/Ngrid
x = np.arange(0,1,dx) + .5 * dx
y = np.arange(0,1,dx) + .5 * dx
r = np.zeros((Ngrid*Ngrid,2))
count = 0
for i in range(Ngrid):
for j in range(Ngrid):
r[count,0] = x[i]
r[count,1] = y[j]
count += 1
return r
def array_make_periodic(x):
x[x¿=1.] -=1.
x[x¡0.] +=1.
def scalar_make_periodic(x):
while (x¡=0.):
x += 1
while (x¿=1.):
x -= 1
return x
def getFractions(r,i,Ngrid):
dx = 1.0 /Ngrid
# square grids
dy = dx
x = r[i][0]
y = r[i][1]
# Grids around each of the particles
left = x - .5 * dx
```

```
# Periodic boundary conditions
#left = scalar_make_periodic(left);
right = left + dx
#right = scalar_make_periodic(right);
down = y - .5 *dx #down = scalar_make_periodic(down);
up = down + dx
#up = scalar_make_periodic(up);
# x index where the particle belongs
xi = math.floor(left/dx)
fracX = left/dx - xi
#xi = np.int32(left/dx)
#fracX = np.abs(right - xi * dx)/(2*dx)
# y index where the particle belongs
# y increases going down.
yi = math.floor(down/dx)
fracY = down/dx - yi
return int(xi),int(yi),fracX,fracY
def makeIndicesPeriodic(xi,yi,Ngrid):
if (xi ¿ Ngrid -1):
xi = 0
elif (xi ¡ 0):
xi = Ngrid - 1
if (yi ¿ Ngrid - 1):
yi = 0
elif (yi ¡ 0):
yi = Ngrid - 1
return xi,yi
# Deposits particles along the CIC algorithm
# Returns the density of the grid
def CIC_deposit(r,mi,Ngrid=100,periodic=1):
"""cloud in cell density estimator
"""
# The size of the grid squares
dx = 1.0/Ngrid
rho = np.zeros((Ngrid,Ngrid))
# Place each of the particles in the grid square they belongin
for i in arange(len(r) ):
xi,yi,fracX,fracY = getFractions(r,i,Ngrid)
```

```
    #Periodic boundary conditions
    xi,yi = makeIndicesPeriodic(xi,yi,Ngrid)
    x1 = xi+1
    y1 = yi+1
    x1,y1 = makeIndicesPeriodic(x1,y1,Ngrid)
    #Update the density
    mass = m[i]
    rho[yi][xi] += mass*(1. - fracX)*(1. - fracY)
    rho[yi][x1] += mass*fracX*(1. - fracY)
    rho[y1][xi] += mass*(1. - fracX)*fracY
    rho[y1][x1] += mass*fracX*fracY
    #rho -= rho.mean
    return rho
    # Returns the force at the CIC particles
    def CIC_force(r,Ngrid,m):
    dx = 1./Ngrid
    rho = CIC_deposit(r,m,Ngrid)
    Phi,fx,fy = PotFFT(rho,Ngrid)
    # the forces at the positions of the particles
    fp = np.zeros((len(r),2))
    # For each particle, need to find the grid in which each particle belongs,
and then
    # take the weighted average of each point around its neighbors
    for i in arange(len(r)):
    xi,yi,fracX,fracY = getFractions(r,i,Ngrid);
    xi,yi = makeIndicesPeriodic(xi,yi,Ngrid)
    x1 = xi+1
    y1 = yi+1
    x1,y1 = makeIndicesPeriodic(x1,y1,Ngrid)
    fp[i][0] = fx[yi][xi]*(1.-fracX)*(1.-fracY)
    fp[i][0] += fx[yi][x1]*fracX*(1.-fracY)
    fp[i][0] += fx[y1][xi]*(1.-fracX)*fracY
    fp[i][0] += fx[y1][x1]*fracX*fracY
    fp[i][1] = fy[yi][xi]*(1.-fracX)*(1.-fracY)
    fp[i][1] += fy[yi][x1]*fracX*(1.-fracY)
    fp[i][1] += fy[y1][xi]*(1.-fracX)*fracY
    fp[i][1] += fy[y1][x1]*fracX*fracY
    return fp
```

```
def PotFFT(d,N):
dx = 1./N
lphi = np.zeros((N,N),dtype=complex)
c = 1
# setup wave-vectors and its square value
kx = np.fft.fftfreq(N) # returns the wave numbers
ky = kx.copy()
kx,ky = np.meshgrid(kx,ky)
k2 = (kx*kx+ky*ky)
delta = np.fft.fft2(d) # forward transform of density
lphi = (-c/(2.*math.pi)**2 *delta*dx**2/k2)
# lphi = (-c/(2.*math.pi)**2 *delta*dx**2/sin(sqrt(k2)**2))
lphi[0,0] = 0.
fPhi = (np.fft.ifft2(lphi)).real
na = mgrid[0:N,0:N]
i = na[0]
j = na[1]
ip1 = np.remainder(na[0]+1, N)
im1 = na[0]-1
jp1 = np.remainder(na[1]+1, N)
jm1 = na[1]-1
fFx = -(fPhi[ip1,j]-fPhi[im1,j])/dx/2
fFy = -(fPhi[i,jp1]-fPhi[i,jm1])/dx/2
return fPhi, fFy, fFx
def vector_make_periodic(r):
array_make_periodic(r[:,0])
array_make_periodic(r[:,1])
# ¡codecell¿
def Evolve(r,v,m,C,Ngrid,dt,tfinal):
dx = 1./Ngrid
time = 0.
#Uniform mass
mass = m[0]
while (time ¡ tfinal):
r = r + dt/2 * v
vector_make_periodic(r)
f = CIC_force(r,Ngrid,m)
v = v + dt * f
```

```
r = r + dt/2 * v
vector_make_periodic(r)
time += dt
return r,v
def getParticleVelocities(r):
vx = .01 * sin(2. * np.pi * r[:,0])
vy = np.copy(vx)
v = np.zeros((len(r),2))
v[:,0] = vx
v[:,1] = vy
return v
def fluidVelocities(r):
v = np.zeros((len(r),2))
for i in xrange(len(r)):
# Set the y velocity to zero
v[i][1] = 0
y = r[i][1]
# x velocity is +1 if the particle is in the top of the fluid # or zero else
if (y ¿ .5):
v[i][0] = -.1
else:
v[i][0] = .1
return v
def fluidMass(r):
m = np.zeros(len(r))
mi = 1./len(r)
for i in xrange(len(r)):
y = r[i][1]
if (y ¿ .5):
m[i] = 1.99* mi
else:
m[i] = .01*mi
return m
def separateFluids(r,m):
f1x = np.zeros(0)
f1y = np.zeros(0)
f2x = np.zeros(0)
f2y = np.zeros(0)
```

```
mi = 1./len(r)
for i in xrange(len(r)):
xi = r[i][0]
yi = r[i][1]
if(m[i] ¿ mi):
f1x = np.append(f1x,xi)
f1y = np.append(f1y,yi)
else:
f2x = np.append(f2x,xi)
f2y = np.append(f2y,yi)
fluid1 = np.zeros((len(f1x),2))
fluid2 = np.zeros((len(f2x),2))
fluid1[:,0] = f1x
fluid1[:,1] = f1y
fluid2[:,0] = f2x
fluid2[:,1] = f2y
return fluid1,fluid2
def GaussianPerturbation(r):
for i in xrange(len(r)):
r[i][0] += np.random.normal(0.0,1./100)
r[i][1] += np.random.normal(0.0,1./100)
return r
def fluidPositions(Ngrid):
dx = 1.0/Ngrid
dy = .2/Ngrid
x = np.arange(0,1,dx) + .5 * dx
y = np.arange(.4,.6,dy)
r = np.zeros((len(y)*len(x),2))
count = 0
for i in xrange(len(x)):
for j in xrange(len(y)):
r[count][0] = x[i]
r[count][1] = y[j]
count += 1
return r
def uniformVelocities(r):
v = np.zeros((len(r),2))
for i in xrange(len(r)):
```

```
v[i][1] = 0
y = r[i][1]
if (y ¿ .5):
v[i][0] = -.1
else:
v[i][0] = .1
return v
# ¡codecell¿
def radial(rmin,deltar,vi,M,N):
r = np.zeros((N,2))
v = np.zeros((N,2))
m = np.zeros(N)
deltaTheta = 2. * np.pi / N
# Particles at a radial distance from the center
for i in xrange(N-1):
theta = deltaTheta * i
ri = rmin + deltar * np.random.random()
vi = (M*ri)**(-.5)
r[i][0] = .5 + ri * np.cos(theta)
r[i][1] = .5 + ri * np.sin(theta)
#v[i][0] = -ri*vi * np.sin(theta)
#v[i][1] = ri*vi * np.cos(theta)
m[i] = M
# One particle at the center much more massive than
# the rest with a zero velocity
r[N-1][0] = .5
r[N-1][1] = .5
m[N-1] = 10000*M
return r,v,m
# ¡codecell¿
# Correlation function
def correlation(r):
# the correlation function
corr = np.cov(r)
# return the mean of the correlation
meanCorr = mean(np.reshape(corr,size(corr)))
return meanCorr
```

6.2 Final_Project

```
# Vlasov Poisson Equation in 2D
    # Author: Javier Palomares
    execfile("Final_Project_Library.py")
    Ngrid = 4
    Np = Ngrid**2
    r = getUniformParticlePositions(Ngrid)
    v = uniformVelocities(r)
    Np = Ngrid*Ngrid
    m = fluidMass(r)
    rho = CIC_deposit(r,m,Ngrid)
    plot(r[:,0],r[:,1],'r.')
    C = 1
    numSteps = 180
    tFinal = 50.0
    dt = float(tFinal)/numSteps
    plot(r[:,0],r[:,1],'r.')
    xlim(0,1)
    ylim(0,1)
    for i in xrange(numSteps):
    r,v = Evolve(r,v,m,C,Ngrid,dt,dt)
    plot(r[:,0],r[:,1],'g.')
    # ¡codecell¿
    # uniform density + m[i]*gaussian - mean
    execfile("Final_Project_Library.py")
    Ngrid = 100
    Np = Ngrid**2
    C = 1.
    m = np.ones(Np)
    r = getUniformParticlePositions(Ngrid)
    v = np.zeros(np.shape(r))
    rho = CIC_deposit(r,m,Ngrid)
    #GaussianPertubations
    r = GaussianPerturbation(r)
    v = GaussianPerturbation(v)
    rho = CIC_deposit(r,m,Ngrid)
    numSteps = 25
```

```
tFinal = 25.0
dt = tFinal/numSteps
for i in xrange(numSteps):
r,v = Evolve(r,v,m,C,Ngrid,dt,dt)
if (i
figure()
rho = CIC_deposit(r,m,Ngrid)
#plt.imshow(rho)
plt.plot(r[:,0],r[:,1],'.')
corr = correlation(r)
print corr
# ¡codecell¿
execfile("Final_Project_Library.py")
Ngrid = 50
Np = Ngrid**2
r = fluidPositions(Ngrid)
m = fluidMass(r)
v = fluidVelocities(r)
rho = CIC_deposit(r,m,Ngrid)
imshow(rho)
colorbar()
rho = CIC_deposit(r,m,Ngrid)
#fluid1,fluid2 = separateFluids(r,m)
figure()
#plot(fluid1[:,0],fluid1[:,1],'r.')
#plot(fluid2[:,0],fluid2[:,1],'g.')
numSteps = 30
tFinal = 30.0
dt = tFinal/numSteps
for i in xrange(numSteps):
r,v = Evolve(r,v,m,C,Ngrid,dt,dt)
if (i
fluid1,fluid2 = separateFluids(r,m)
figure()
#plot(fluid1[:,0],fluid1[:,1],'r.')
#plot(fluid2[:,0],fluid2[:,1],'g.')
rho = CIC_deposit(r,m,Ngrid)
plt.imshow(rho)
```

```
colorbar()
# ¡codecell¿
execfile("Final_Project_Library.py")
Ngrid = 200
Np = 500
rmin = .05
deltar = .1
vi = 1.
mi = 1./Np
r,v,m = radial(rmin,deltar,vi,mi,Np)
rho = CIC_deposit(r,m,Ngrid)
plt.xlim(0,1)
plt.ylim(0,1)
plt.plot(r[:,0],r[:,1],'.')
numSteps = 40
tFinal = 40.0
C = 1.
dt = tFinal/numSteps
for i in xrange(numSteps):
r,v = Evolve(r,v,m,C,Ngrid,dt,dt)
if (i
figure()
plt.xlim(0,1)
plt.ylim(0,1)
plt.plot(r[:,0],r[:,1],'.')
```