Flocking Mechanics in Python

Joe Gibson and David Schmidt

April 6, 2018

Parameters

```
import numpy as np
import matplotlib.pyplot as plt
import mpl_toolkits.mplot3d.axes3d as p3
from scipy.spatial.distance import pdist, squareform
import matplotlib.animation as animation


#constants
radiusH=0.1
massH=1
box_length=200*radiusH

numparticles=20

groupdis=2
sepdist=0.1

weight_align=5
weight_cohesion=2
weight_seperation=1
weight_airfield=0
###########################################################################################

def airfield(x,y,z):
    return np.array([y,-x,0])
```

Class Initialization

```
class physicsvolume:
    def __init__(self,
                cornors=[-box_length/2,box_length/2,-box_length/2,box_length/2,-box_length/2,box_length/2],
                init_state=[[0,0,0,0,0,0,1],[0,0,0,0,0,0,0,1]],
                radius=radiusH):

        #things internal to the box
        self.init_state=np.asarray(init_state,dtype=float)
        self.state=self.init_state.copy()
        self.radius=radiusH
        self.time_elapsed=0
        self.cornors=cornors
        self.centerofmass=[]
        self.groupvel=[]
        self.neighbormass=0
        self.seperation=[]
        self.cohesion=[]
        self.alignment=[]
        self.aireffect=[]
```

Step Function

```
def step(self, dt):
    #move time
    self.time_elapsed +=dt

        #move particles
    self.state[:,:3]+=dt*self.state[:,3:6]

#Simple Collision Detection thing
        #finds distance between particles
        #produces an NxN array of distances between particles
    D=squareform(pdist(self.state[:,:3]))

    ###################################################
    #Flocking behavior (figure out what birds are within area of influence for each other)
    bird1,bird2 = np.where(D<groupdis)
    pairs = [[bird1[i], bird2[i]] for i in range(len(bird1)) if bird1[i]!=bird2[i]]
    plen=len(pairs)
    birdi1=np.arange(0,numparticles)
    start=0
```

Bird Properties

```
for i1 in birdi1:
    m1=self.state[i1,6]
    r1=np.array(self.state[i1,:3])
    v1=self.state[i1,3:6]
    self.centerofmass=np.array([0,0,0])
    self.seperation=np.array([0,0,0])
    self.alignment=np.array([0,0,0])
    self.cohesion=np.array([0,0,0])
    self.groupvel
    self.neighbormass=1
    for i2 in range(start,plen):
        if i1!=pairs[i2][0]:
            start=i2
            break
        elif i1==pairs[i2][1]:
            pass
        else:
        #neighbor properties
            b2=bird2[i2]
            #mass
            m2=self.state[b2,6]
            # position
            r2=np.array(self.state[b2,:3])
            # velocity
            v2=self.state[b2,3:6]
```

Flocking

```
        #neighbor calculations
            self.neighbormass=self.neighbormass+self.state[b2,6]
            #center of mass
            self.centerofmass=self.centerofmass+m2*r2

            #velocity pointing
            self.alignment=self.alignment+v2
            dis=abs(np.linalg.norm(r1-r2))
            if dis<=sepdist:
                self.seperation=self.seperation+(r2)
    #cohesion
    com=self.centerofmass/self.neighbormass
    comnorm=np.linalg.norm(com)
    if comnorm==0.0:
        self.cohesion=np.array([0,0,0])
    else:
        self.cohesion=com/comnorm
    #seperation
    sepnorm=np.linalg.norm(self.seperation)
    if sepnorm==0.0:
        self.seperation=np.array([0,0,0])
    else:
        self.seperation=self.seperation/sepnorm
    #alignment
    alignnorm=np.linalg.norm(self.alignment)
    if alignnorm==0.0:
        self.alignment=np.array([0,0,0])
    else:
        self.alignment=self.alignment/alignnorm
```

Flocking Velocities

```
#airfield calulations
self.aireffect=airfield(r1[0],r1[1],r1[2])

#assign new velocities
↪    velfull=v1+(self.alignment*weight_align-self.cohesion*weight_cohesion+self.seperation*weight_seperation+self.aireffect*we
veln=np.linalg.norm(velfull)
self.state[i1,3:6]=np.ndarray.tolist(velfull/veln)
```

Collisions

```
########################################################
    #calculate collisions
ind1,ind2 = np.where(D<2*self.radius)
unique = (ind1<ind2)
ind1 = ind1[unique]
ind2 = ind2[unique]
for i1,i2 in zip(ind1,ind2):
    m1=self.state[i1,6]
    m2=self.state[i2,6]
    # positions
    r1=self.state[i1,:3]
    r2=self.state[i2,:3]
    # velocities
    v1=self.state[i1,3:6]
    v2=self.state[i2,3:6]
    #relative location and velocities
    r_rel=r1-r2
    v_rel=v1-v2
    #momentum of com
    v_cm=(m1*v1+m2*v2)/(m1+m2)
    #collisions of spheres
    rr_rel = np.dot(r_rel,r_rel)
    vr_rel = np.dot(v_rel,r_rel)
    v_rel=2*r_rel*vr_rel / rr_rel - v_rel

    #assign new velocities
    self.state[i1,3:6]=(v_cm+v_rel*m2/(m1+m2))/np.linalg.norm((v_cm+v_rel*m2/(m1+m2)))
    self.state[i2,3:6]=(v_cm-v_rel*m1/(m1+m2))/np.linalg.norm((v_cm-v_rel*m2/(m1+m2)))
```

Boundary Conditions

```
#boundary crossing
#this just makes sure the birds don't clip through the wall
crossed_x1 = (self.state[:,0]<self.cornors[0] + self.radius)
crossed_x2 = (self.state[:,0]>self.cornors[1] - self.radius)
crossed_y1 = (self.state[:,1]<self.cornors[2] + self.radius)
crossed_y2 = (self.state[:,1]>self.cornors[3] - self.radius)
crossed_z1 = (self.state[:,2]<self.cornors[4] + self.radius)
crossed_z2 = (self.state[:,2]>self.cornors[5] - self.radius)

self.state[crossed_x1,0]=self.cornors[0] + self.radius
self.state[crossed_x2,0]=self.cornors[1] - self.radius

self.state[crossed_y1,1]=self.cornors[2]+self.radius
self.state[crossed_y2,1]=self.cornors[3]-self.radius

self.state[crossed_z1,2]=self.cornors[4]+self.radius
self.state[crossed_z2,2]=self.cornors[5]-self.radius

#makes the birds "bounce" back the way they came from
self.state[crossed_x1 | crossed_x2,3]*=-1
self.state[crossed_y1 | crossed_y2,4]*=-1
self.state[crossed_z1 | crossed_z2,5]*=-1

#renormalizes the velocity
for i in range(numparticles):
    veln=np.linalg.norm(self.state[i1,3:6])
    self.state[i1,3:6]=np.ndarray.tolist(self.state[i1,3:6]/veln)
```

Setup Simulation

```
#this creates an initial state of numparticle number of balls with an initial position velocity and mass
np.random.seed()
red=np.random.rand(numparticles,7)-0.5

#makes sure they are in the box
red[:,:3]*=box_length*.8

#sets all the masses equal
red[:,6]=massH

#says let there be a box
box = physicsvolume(init_state=red,radius=1)
dt=1./30
```

Animation

```python
fig = plt.figure()
ax= p3.Axes3D(fig)
#plots for the particles
particles, = ax.plot([],[],[],'bo',ms=1,animated=True)
# Setting the axes properties
ax.set_xlim3d([-box_length/2-1, box_length/2+1]);ax.set_xlabel('X')
ax.set_ylim3d([-box_length/2-1, box_length/2+1]);ax.set_ylabel('Y')
ax.set_zlim3d([-box_length/2-1, box_length/2+1]);ax.set_zlabel('Z')

def init():
    particles.set_data([],[])
    particles.set_3d_properties([])
    return particles,

def animate(i):
    box.step(dt)
    ms = int(fig.dpi*box.radius*fig.get_figwidth()/np.diff(ax.get_xbound())[0])
    particles.set_data(box.state[:,0],box.state[:,1])
    particles.set_3d_properties(box.state[:,2])
    particles.set_markersize(ms)

    return particles,

ani=animation.FuncAnimation(fig,animate,frames=1000,interval=1,blit=True,init_func=init)
#anim = animation.FuncAnimation(fig, animate, init_func=init, frames=200, interval=20, blit=True)

#FFwriter = animation.FFMpegWriter(fps=30)
#ani.save('flocking2.mp4', writer = FFwriter)
#plt.show()
```

Examples

Lets go see the code in action