Gravitational particle Interaction Simulation

Group Members

Rishi Dinesh, Kaustubh Singh,
Nandika Damarla, Kaushik Vasan

SNU, PHY105-Project, November 2020

1 Abstract

The problem of describing the trajectories of particles under the influence of gravity is a well-known one. For simple systems of two bodies this problem can be solved analytically, but beyond that the problem must be solved numerically. The aim of this project is to create a program that helps visualise the motion of "N" bodies under the influence of an attractive Newtonian gravitational force. This is done by considering the total force on each body in three-dimensional space and considering each one's acceleration, and carrying out the displacement of the bodies in space in small changes. This approximation gives a good idea of the way the N body system evolves with time.

2 Introduction

The famous 'N-body problem' concerns itself with finding the trajectories of masses when each mass is interacting gravitationally with every other mass. This problem can be solved exactly for a two-body system by looking at the trajectory of the center of mass. For a greater number of bodies, the system becomes increasingly complex and even displays chaotic behaviour. That is, they display a great sensitivity on the initial conditions. The solutions of the equations of motion cannot be written in terms of well-known functions, and one must resort to numerical methods to solve such a problem. This project looks at the specific case of gravitationally interacting bodies in three-dimensional space and attempts to solve this problem by making certain approximations and attempting to solve the problem incrementally. The following sections explain the exact method by which the program does so. The project considers Newtonian gravity and gives a visual representation of the bodies' motion in space. For any direct contact interactions between particles this program also considers perfectly inelastic and elastic collisions.

3 Theory

Newtons law of gravitation states that for two point masses separated in space the magnitude of the force between them is inversely proportional to the square of distance between them and directly proportional to their masses. This law is stated mathematically as:

$$F = G\frac{M \times m}{|\overrightarrow{r}|^2} \tag{1}$$

Where G is the universal gravitational constant, M and m are the masses of the two bodies, r is the vector distance between them. This force acts along the r vector for each body and they are directed opposite to each other.

For spherical objects this law can be applied considering all their mass to be concentrated at their centre of mass. The code considers the force acting on each body in the set of N bodies due to every other body and gives the total force on each body.

In order to look at the trajectory and the displacement of the bodies evolving with time, the code considers time in small increments of dt for each time interval it calculates the change in the body's velocity and the body is then displaced by a small quantity $dt \times v$ This is the fundamental idea of the code.

For collisions there are two main aspects: the detection of the collision and the resulting change in velocity of the particles involved. For the first part there are two possible methods of detecting collisions often employed by computer game creators and simulation makers, static and dynamic detection. Static detection of collisions involves checking if a collision has taken place by checking whether the distance between two bodies is less than some permissible minimum (the sum of the radii of the spheres) and if there is a collision, then the necessary changes are made. Dynamic detection is a slightly more complicated process involving the pre-emptive prediction of collisions in order to avoid possible "overlaps" or other errors in the code. This project has made use of the static detection method.

For the second aspect of collisions, the change in velocity there are two situations considered: one is when the collision is elastic and the other is when it is inelastic. In case of elastic collisions, the velocity vector is resolved into two vectors: one parallel to the vector separating the two colliding bodies- this is the line along which the collision takes place- and the other is the one perpendicular to this vector. Changes in velocity only occur in the first direction and the velocity vector is changed appropriately in this direction only. The changes in the case of an elastic collision are as follows and apply to each of the three components of velocity:

$$v_1 = \frac{2mu_2 + (M - m)u_1}{M + m} \tag{2}$$

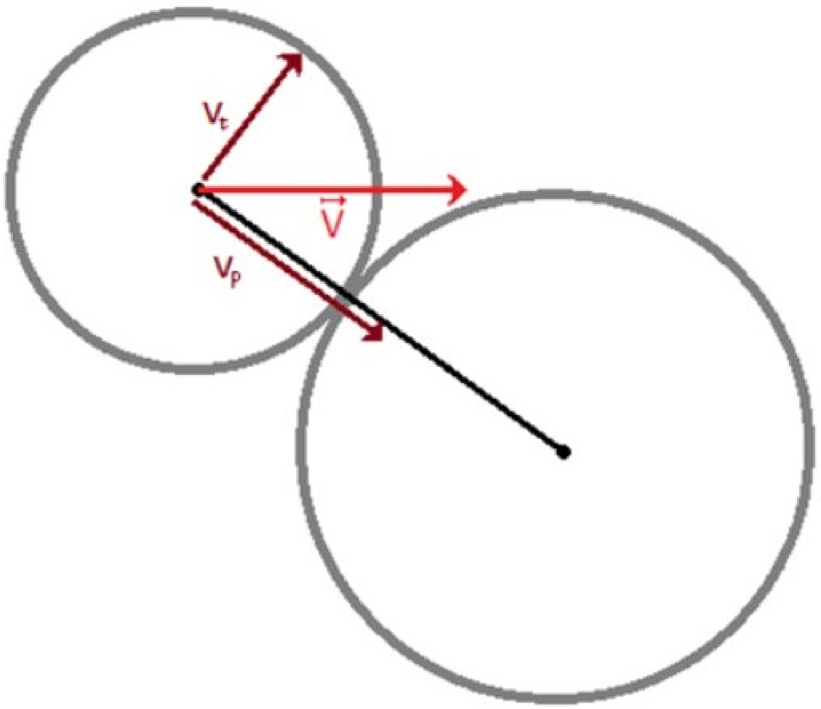
The case of inelastic collisions is far simpler. The change is made to the whole velocity vector and the two bodies merge into one and their velocity is changed as follows:

$$v = \frac{mu_2 + Mu_1}{M + m} \tag{3}$$

Finally, for the program itself: this project uses two user defined objects: Vector and Particle. They contain functions relating to the expected behaviour of the particles and vectors. The Particle object uses Vector instance variables in order to describe the velocity, position, and force acting on the bodies. Classes are blueprints describing objects which are an instance of the class. They have attributes and properties which are described by instance variables and object functions. The details of the various functions in each class are given in subsequent sections. The code uses the "vpython" module in order to display the calculated information and to visually represent the system. The vpython module can be installed like numpy, using pip.

4 Diagram

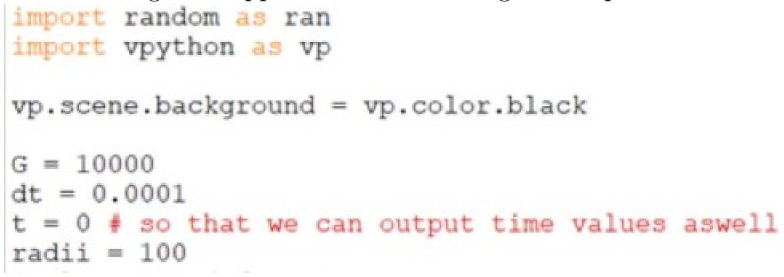
The following is a diagram showing the elastic collisions in a two-dimensional space and depicts the components of velocity that change and those that don't.



The component of the velocity that changes is the component marked V_p.

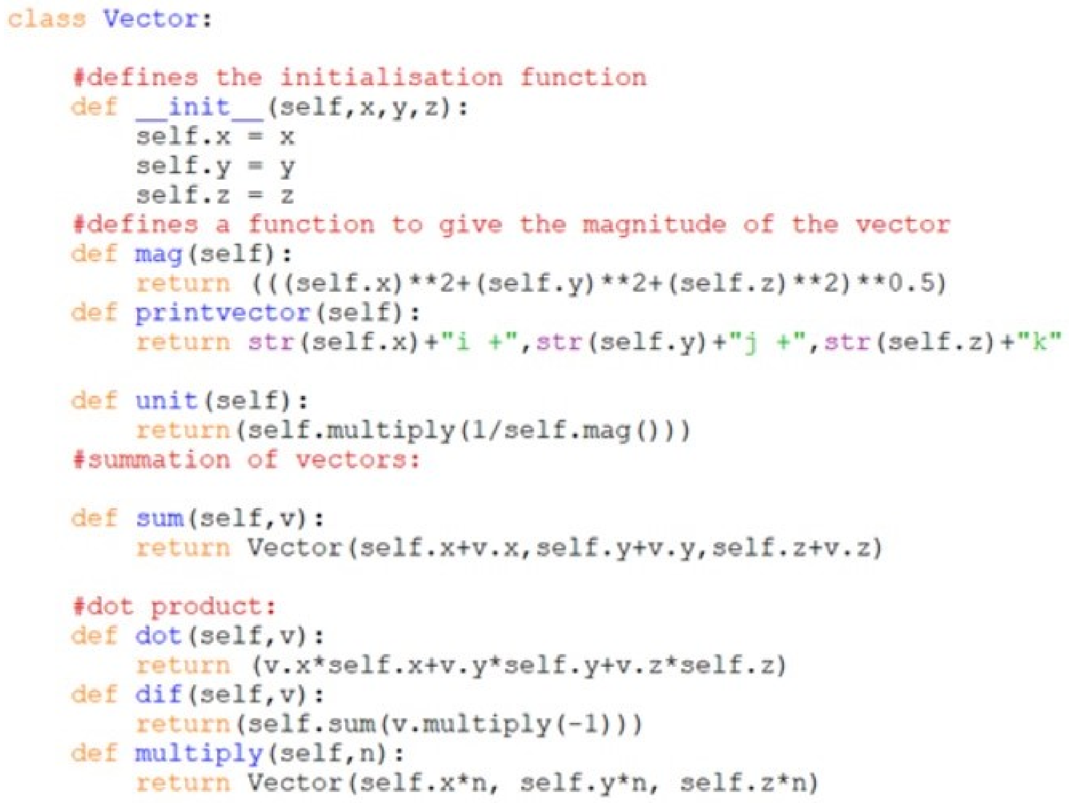
5 Code

The following are snippets of the code along with explanations of what they do.



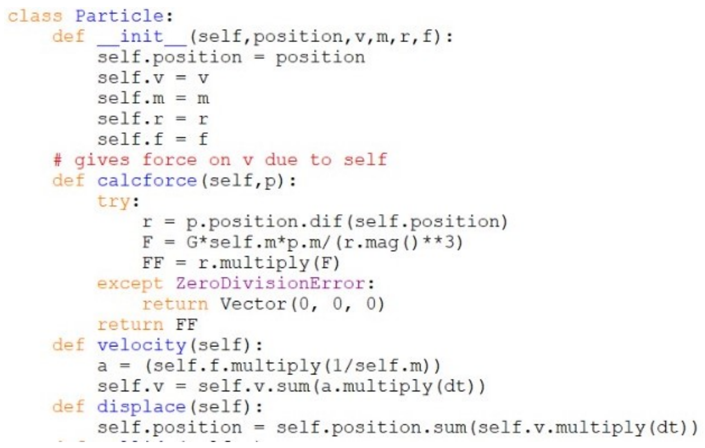
```python
import random as ran
import vpython as vp

vp.scene.background = vp.color.black

G = 10000
dt = 0.0001
t = 0 # so that we can output time values aswell
radii = 100
```

This is the opening block of code. Here the various required modules are imported and the values which are universal to the code are initialised such as the value of G and the time increment, dt, and also the radii considered for the bodies in the visualisation module, vpython.



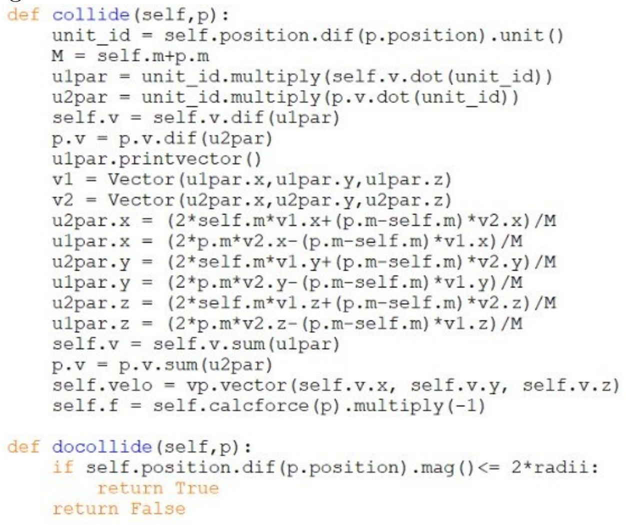
```python
class Vector:

    #defines the initialisation function
    def __init__(self,x,y,z):
        self.x = x
        self.y = y
        self.z = z
    #defines a function to give the magnitude of the vector
    def mag(self):
        return (((self.x)**2+(self.y)**2+(self.z)**2)**0.5)
    def printvector(self):
        return str(self.x)+"i +",str(self.y)+"j +",str(self.z)+"k"

    def unit(self):
        return(self.multiply(1/self.mag()))
    #summation of vectors:

    def sum(self,v):
        return Vector(self.x+v.x,self.y+v.y,self.z+v.z)

    #dot product:
    def dot(self,v):
        return (v.x*self.x+v.y*self.y+v.z*self.z)
    def dif(self,v):
        return(self.sum(v.multiply(-1)))
    def multiply(self,n):
        return Vector(self.x*n, self.y*n, self.z*n)
```

The above code is defining a class called vector which in turn acts as a blueprint for the objects, Vector. The class contains methods to return the modulus, difference, and sum of vectors, the dot product of vectors, and one that returns a unit vector in the direction of a given vector. This code makes the code easier to read and makes it more intuitive to see what certain pieces of code do and makes them reusable. This code could have been written in a completely non object oriented manner, but it would have been more difficult to read and less intuitive.



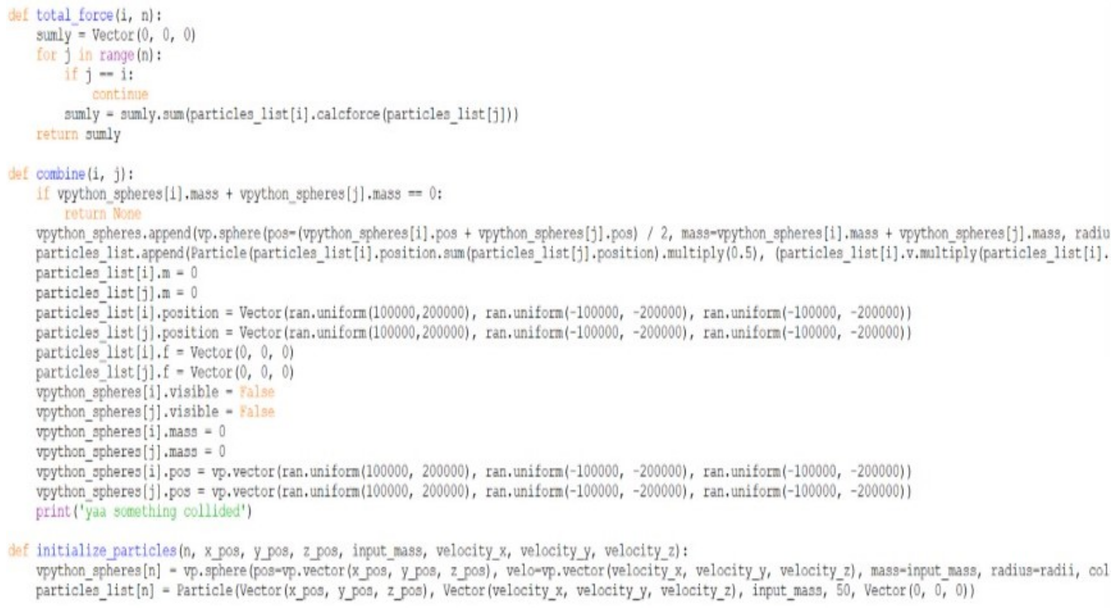
```
class Particle:
    def __init__(self,position,v,m,r,f):
        self.position = position
        self.v = v
        self.m = m
        self.r = r
        self.f = f
    # gives force on v due to self
    def calcforce(self,p):
        try:
            r = p.position.dif(self.position)
            F = G*self.m*p.m/(r.mag()**3)
            FF = r.multiply(F)
        except ZeroDivisionError:
            return Vector(0, 0, 0)
        return FF
    def velocity(self):
        a = (self.f.multiply(1/self.m))
        self.v = self.v.sum(a.multiply(dt))
    def displace(self):
        self.position = self.position.sum(self.v.multiply(dt))
```

The above is the code for the class Particle in which the objects defined represent the actual spheres that are manipulated in three-dimensional space by vpython.It has instance variables representing the mass, the force on the particles, their velocity, their position, and radii. It also contains methods that calculate the force between two particles, one that changes the velocity of a particle in accordance with the force on it and one to displace its position in accordance with that velocity. The function called *__init__* is an inbuilt syntax in python that is called every time an object of the class is initialised with some arguments.



```
def collide(self,p):
    unit_id = self.position.dif(p.position).unit()
    M = self.m+p.m
    u1par = unit_id.multiply(self.v.dot(unit_id))
    u2par = unit_id.multiply(p.v.dot(unit_id))
    self.v = self.v.dif(u1par)
    p.v = p.v.dif(u2par)
    u1par.printvector()
    v1 = Vector(u1par.x,u1par.y,u1par.z)
    v2 = Vector(u2par.x,u2par.y,u2par.z)
    u2par.x = (2*self.m*v1.x+(p.m-self.m)*v2.x)/M
    u1par.x = (2*p.m*v2.x-(p.m-self.m)*v1.x)/M
    u2par.y = (2*self.m*v1.y+(p.m-self.m)*v2.y)/M
    u1par.y = (2*p.m*v2.y-(p.m-self.m)*v1.y)/M
    u2par.z = (2*self.m*v1.z+(p.m-self.m)*v2.z)/M
    u1par.z = (2*p.m*v2.z-(p.m-self.m)*v1.z)/M
    self.v = self.v.sum(u1par)
    p.v = p.v.sum(u2par)
    self.velo = vp.vector(self.v.x, self.v.y, self.v.z)
    self.f = self.calcforce(p).multiply(-1)

def docollide(self,p):
    if self.position.dif(p.position).mag()<= 2*radii:
        return True
    return False
```

The above code is in the Particle class and one of the function checks by static detection if a collision has occurred between two particles and the other function changes the velocity of the particles as it would in the case of an elastic collision.



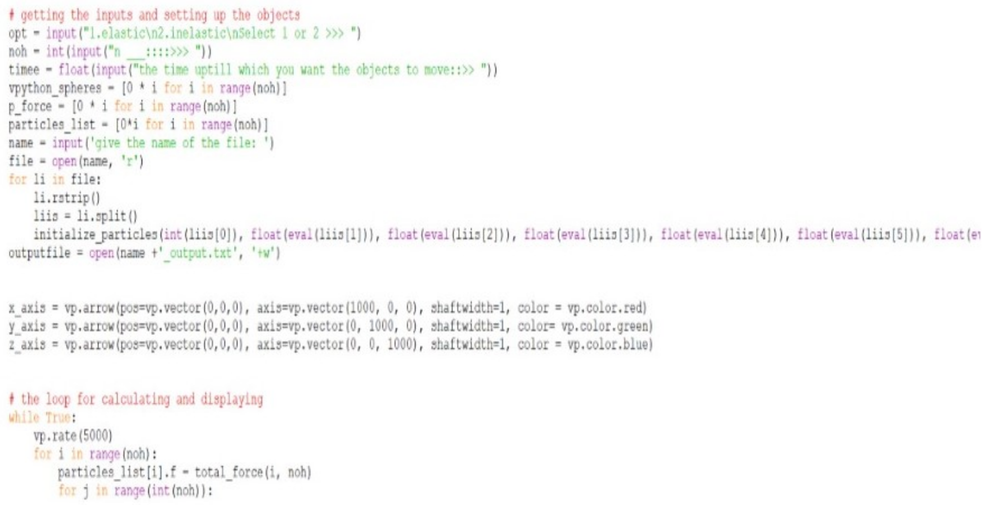
```python
def total_force(i, n):
    sumly = Vector(0, 0, 0)
    for j in range(n):
        if j == i:
            continue
        sumly = sumly.sum(particles_list[i].calcforce(particles_list[j]))
    return sumly

def combine(i, j):
    if vpython_spheres[i].mass + vpython_spheres[j].mass == 0:
        return None
    vpython_spheres.append(vp.sphere(pos=(vpython_spheres[i].pos + vpython_spheres[j].pos) / 2, mass=vpython_spheres[i].mass + vpython_spheres[j].mass, radiu
    particles_list.append(Particle(particles_list[i].position.sum(particles_list[j].position).multiply(0.5), (particles_list[i].v.multiply(particles_list[i].
    particles_list[i].m = 0
    particles_list[j].m = 0
    particles_list[i].position = Vector(ran.uniform(100000,200000), ran.uniform(-100000, -200000), ran.uniform(-100000, -200000))
    particles_list[j].position = Vector(ran.uniform(100000,200000), ran.uniform(-100000, -200000), ran.uniform(-100000, -200000))
    particles_list[i].f = Vector(0, 0, 0)
    particles_list[j].f = Vector(0, 0, 0)
    vpython_spheres[i].visible = False
    vpython_spheres[j].visible = False
    vpython_spheres[i].mass = 0
    vpython_spheres[j].mass = 0
    vpython_spheres[i].pos = vp.vector(ran.uniform(100000, 200000), ran.uniform(-100000, -200000), ran.uniform(-100000, -200000))
    vpython_spheres[j].pos = vp.vector(ran.uniform(100000, 200000), ran.uniform(-100000, -200000), ran.uniform(-100000, -200000))
    print('yaa something collided')

def initialize_particles(n, x_pos, y_pos, z_pos, input_mass, velocity_x, velocity_y, velocity_z):
    vpython_spheres[n] = vp.sphere(pos=vp.vector(x_pos, y_pos, z_pos), velo=vp.vector(velocity_x, velocity_y, velocity_z), mass=input_mass, radius=radii, col
    particles_list[n] = Particle(Vector(x_pos, y_pos, z_pos), Vector(velocity_x, velocity_y, velocity_z), input_mass, 50, Vector(0, 0, 0))
```

This code is outside of all classes and the three methods described here have three different actions. The first gives the total force acting on a particle in a list of particles at the index ‘i’ and returns it as a Vector object.

Combine is a function that calculates velocity in case of inelastic collisions and also combines the two vpython spheres objects and creates one in place of two that collide and represent the combination of two massive bodies.

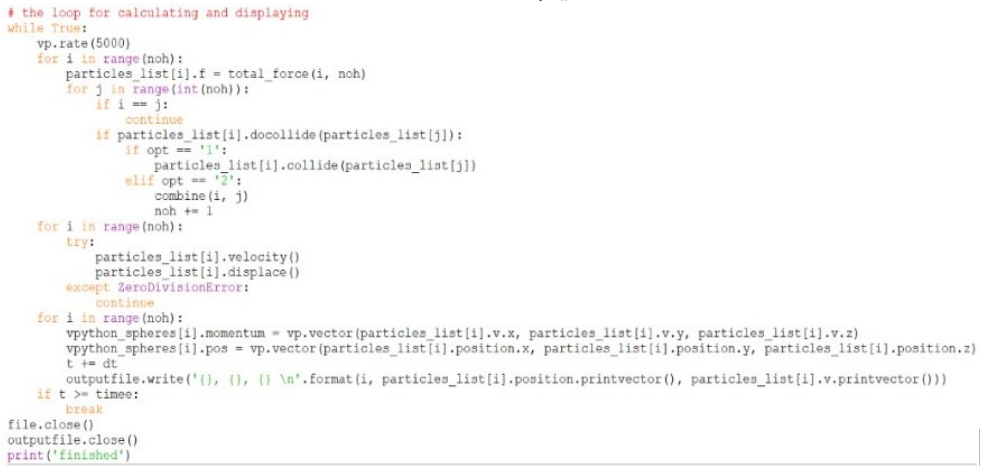
The final function is one that initialises the list of Particle objects and the vpython spheres with some desired values.

(Note: some of the lines of code go off screen and are not in the screen shot.)



```
# getting the inputs and setting up the objects
opt = input("1.elastic\n2.inelastic\nSelect 1 or 2 >>> ")
noh = int(input("n ___::::>>> "))
timee = float(input("the time uptill which you want the objects to move::>> "))
vpython_spheres = [0 * i for i in range(noh)]
p_force = [0 * i for i in range(noh)]
particles_list = [0*i for i in range(noh)]
name = input('give the name of the file: ')
file = open(name, 'r')
for li in file:
    li.rstrip()
    liis = li.split()
    initialize_particles(int(liis[0]), float(eval(liis[1])), float(eval(liis[2])), float(eval(liis[3])), float(eval(liis[4])), float(eval(liis[5])), float(e
outputfile = open(name +'_output.txt', '+w')


x_axis = vp.arrow(pos=vp.vector(0,0,0), axis=vp.vector(1000, 0, 0), shaftwidth=1, color = vp.color.red)
y_axis = vp.arrow(pos=vp.vector(0,0,0), axis=vp.vector(0, 1000, 0), shaftwidth=1, color= vp.color.green)
z_axis = vp.arrow(pos=vp.vector(0,0,0), axis=vp.vector(0, 0, 1000), shaftwidth=1, color = vp.color.blue)


# the loop for calculating and displaying
while True:
    vp.rate(5000)
    for i in range(noh):
        particles_list[i].f = total_force(i, noh)
        for j in range(int(noh)):
```

This piece of code gets necessary inputs from the user such as whether the collisions should be elastic or inelastic and how many particles are desired.

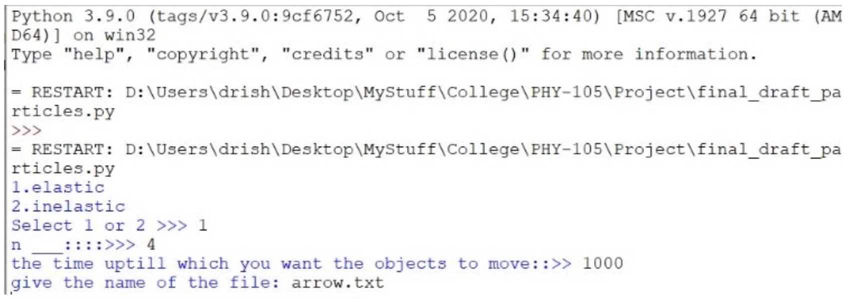


```
# the loop for calculating and displaying
while True:
    vp.rate(5000)
    for i in range(noh):
        particles_list[i].f = total_force(i, noh)
        for j in range(int(noh)):
            if i == j:
                continue
            if particles_list[i].docollide(particles_list[j]):
                if opt == '1':
                    particles_list[i].collide(particles_list[j])
                elif opt == '2':
                    combine(i, j)
                    noh += 1
    for i in range(noh):
        try:
            particles_list[i].velocity()
            particles_list[i].displace()
        except ZeroDivisionError:
            continue
    for i in range(noh):
        vpython_spheres[i].momentum = vp.vector(particles_list[i].v.x, particles_list[i].v.y, particles_list[i].v.z)
        vpython_spheres[i].pos = vp.vector(particles_list[i].position.x, particles_list[i].position.y, particles_list[i].position.z)
        t += dt
        outputfile.write('{}, {}, {} \n'.format(i, particles_list[i].position.printvector(), particles_list[i].v.printvector()))
    if t >= timee:
        break
file.close()
outputfile.close()
print('finished')
```

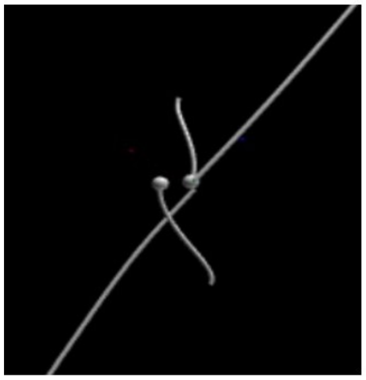
The above piece of code is the final part of the program that displays the outputs in vpython. The function vp.rate(¡r¿) makes the code change with time and the argument of the function specifies how many times in a second the loop is iterated through. This code has loops to total the force on each particle, checks if any collide and makes changes to velocity and position.

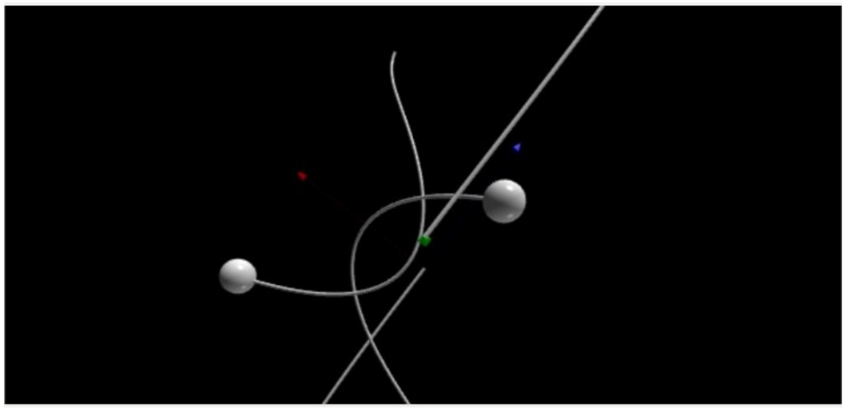
This is the end of this program

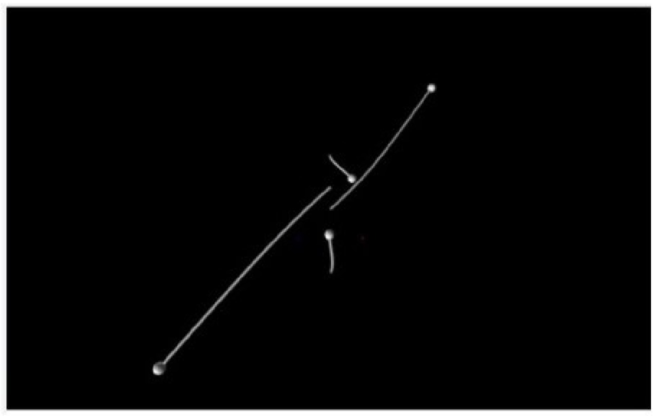
6 Outputs

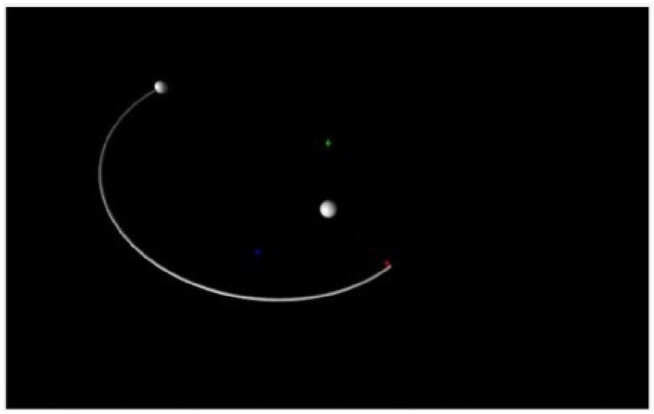


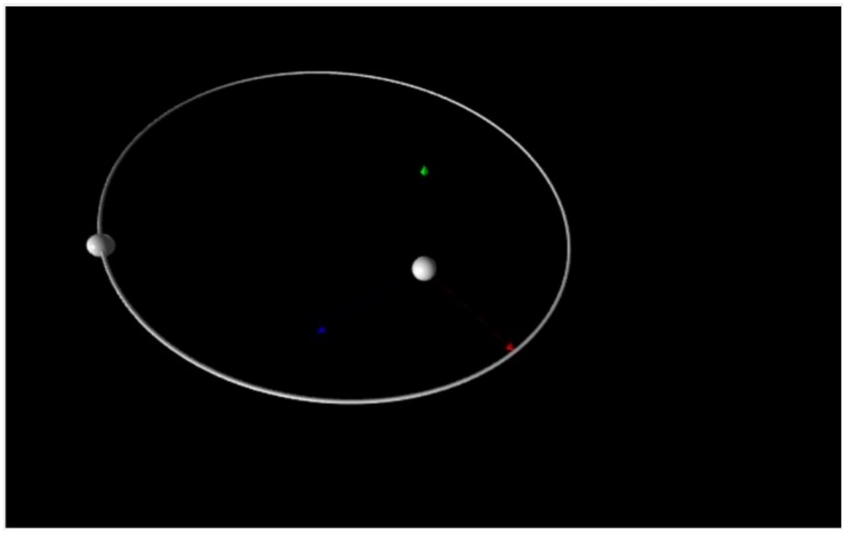
```
Python 3.9.0 (tags/v3.9.0:9cf6752, Oct  5 2020, 15:34:40) [MSC v.1927 64 bit (AM
D64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.

= RESTART: D:\Users\drish\Desktop\MyStuff\College\PHY-105\Project\final_draft_pa
rticles.py
>>>
= RESTART: D:\Users\drish\Desktop\MyStuff\College\PHY-105\Project\final_draft_pa
rticles.py
1.elastic
2.inelastic
Select 1 or 2 >>> 1
n ___::::>>> 4
the time uptill which you want the objects to move::>> 1000
give the name of the file: arrow.txt
```







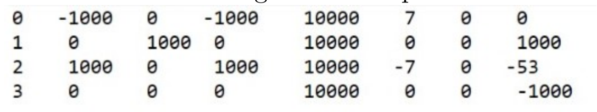




The above are screenshots of the outputs of the code. Some show a 4 body system with some initial velocities and one 2 body system in which a less massive particle revolves in an elliptical orbit around another more massive one.

7 Inputs to the Code

The code requires certain specific inputs, specifying whether collisions ought to be inelastic or elastic, the number of particles, n, the file from which to read data for the initial conditions, and the duration for which the code must continue to execute. The following is an example of how the data file ought to look.



```
0   -1000   0    -1000    10000    7    0    0
1    0      1000  0       10000    0    0    1000
2    1000   0     1000    10000   -7    0   -53
3    0      0     0       10000    0    0    -1000
```

Each row is one particle. The first column is some index, the second,third and fourth are the position coordinates, the fifth is the mass, and the last three are the components of the velocity vectors.

8 Error Analysis

The code is based in the approximation that during some small time, dt, the velocity of the bodies remains constant. However, this assumption is only valid for small values of dt and the approximation gets better and better the smaller dt gets. The limitation is that dt cannot be made too small since most computers are limited in their computing capacity and this may result in rounding errors in code, also, making dt too small increases the number of iterations of the object displacement loop required to reach a certain point in the simulation and the rate of the simulations progress decreases. This can be mitigated by increasing the rate but there is again a limit on the number of iterations a computer can do per second.

The other error is in the detection of collisions. Since the function checking for collisions does so by checking if any two particles have come within a specified distance of each other, it is possible that they would overlap to a significant degree before any collision is detected if dt is too big. This error can be avoided by using the superior dynamic detection method but for the sake of simplicity this code uses static detection. The issue with overlapping objects is that it results in problems in the vpython animation and also the force can often become extremely great and may even result in a ZeroDivisionError. These issues cause the code to behave in unexpected ways.

9 References and Citations

[1] *The data for the solar system initial conditions was gotten from this site.*
https://ssd.jpl.nasa.gov/horizons.cgi
[2] *The documentation for vpython was used extensively in order to find functions and syntax*

that performed certain desired tasks
https://www.glowscript.org/docs/VPythonDocs/index.html
[3] *The initial idea for using vpython and for the method of executing this task the idea was gotten from the following YouTube video*
https://www.youtube.com/watch?v=aCpBzdciU0o