PyGravity

http://github.com/russloewe/PyGravity

Russell Loewe

7 May 2015

This is the supporting documentation for *PyGravity*

Contents

0.1 Intro

Purpose This is a gravity simulator written in Python and using PyGame as a render. This project is starting at the basics of using vector calculus and Newtons law of Gravity to calculate the paths of particles with respect to the gravity exerted by neighbouring particles. One of many end goals with this simulator is to simulate asteroids colliding into more massive asteroids, eventually becoming planets.

Context This simulator is going to be dealing with non-relativistic energies in flat space-time. That means every object can be given by a Euclidean 3-space vector. Instead of dealing with the 4 coordinates of space-time, I will assume that every object shares the same clock and thus time will be kept globally by the physics engine.

Vectors I am writing my own library for the various mathematical operations that will be need to calculate the forces. The library is going to start off with the basic vector objects. I will then add functions on vector objects such as addition, subtraction, multiplication by a scalar, normalizing, finding orthogonal vectors, and etcetera. At this point and for the rest of the application I will use Numpy to handle the large or small numbers that are going to be needed in calculations.

Objects I am also going to need to create a class object to encapsulate the particles. This class will contain the needed attributes of each particle such as the scalar mass, the vectors of position and velocity.The classes will also need to contain procedures for particle collisions. This will most likely not be need until later when I use this simulate to for colliding asteroids.

Physics After the math foundation and base object classes are established then I will need to develop a physics library to calculate physics specfic formulaes such as the force of gravity, acceleration and the net force of gravity for more than two objects.

PyGame Once I have a foundation of vector math built into my simulator I will then move on to using PyGame to plot the position of various particles and then render their movement. PyGame will be used to render the evolution of the dynamical system and also display various stats of the particles. I will also pause at this point to use PyGame to graph the gradient of gravity in the system and other fun stuff.

Chapter 1

Basic Math and Physics

1.1 The Vector Class

1.1.1 Vector Base

First thing is to have a code representation of a vector. A vector in 3-space needs to have three components. The general vector class is structured as follows:

```
class Vector(object):
    def __init__(self, x, y, z):
        self.x = x
        self.y = y
        self.z = z
        self.show = "(" + str(self.x) + ',' + str(self.y) + ',' +
            str(self.z) + ')'
```

The vector is initialized with the three components. The *__init__* method also contains the attribute *show*, which is an easy way to view the vector in string format for interactive sessions. Thus a new vector, $\vec{A}$, is created and displayed by

```
A = Vector(1.1, 1.3, 1.4)
A.show
'(1.1, 1.3, 1.4)'
```

1.1.2 Addition

Next method to add to the base vector class is the function for adding two vectors. Vector addition is going to be a static class method of the Vector Class which takes two vectors and returns a new vector.

```
    @staticmethod
    def add(A, B):
        new = Vector(A.x + B.x,
                     A.y + B.y,
                     A.z + B.z)
        return new
```

Thus to find $\vec{C} = \vec{A} + \vec{B}$, we would use this

```
A = Vector(1,1,1)
B = Vector(2,1,3)
C = Vector.add(A,B)
C.show
'(3,2,4)'
```

The test suite ensures that this addition holds true for positive and negative integers, and real numbers up to 10^{24}. It is important to note that boolean operations are only guaranteed to hold true when comparing two vector components, not necessarily a vector component and an isolated number. Example below.

```
BigA = Vector(1.3*10**24, 0, 0)
BigB = Vector(1.1*10**24, 0, 0)
Answer = Vector(2.4*10**24, 0, 0)
C = Vector.add(BigA, BigB)

C.x == Answer.x      ::True
C.x == 2.4*10**24    ::False
```

1.1.3 Subtraction

Subtraction is coded exactly like addition except that each component of $\vec{A}$ is subtracted by each cmponent of $\vec{B}$. The first function argument is the minuend and the second argument is the subtrahend. Example:

```
A = Vector(1,1,1)
B = Vector(2,2,2)
Vector.sub(A, B) = (-1, -1, -1)
Vector.sub(B, A) = (1, 1, 1)
```

1.1.4 Scalar Multiplication

Next the Vector Class needs a method for multiplying vectors by a scalar. The multiplication method will be very similar to the addition method.

```
@staticmethod
def times_scalar(a, A):
    new = Vector(A.x * a,
                 A.y * a,
                 A.z * a)
    return new
```

This method returns a new vector, thus usage is as following:

```
A = Vector(1.1, 2.2, 4.4)
a = 2.0
C = Vector.times_scalar(a, A)
C.show
'(2.2, 4.4, 8.8)'
```

For testing there were problems when working with large numbers. The computation could be correct but the answer wouldn't pass the tests because of small errors resulting from working with such large numbers. Thus rounding would be needed so only significant figures would be compared. Here is a quick rounding to significant digits function taken from *StackOverflow*[1] along with a tweak to avoid math domain errors.

```
from math import log10, floor
def round_sig(x, sig=2):
    if x == 0:
        return 0
    if x < 0:
        x = -x
        return (-1)* round(x, sig-int(floor(log10(x)))-1)
    else:
        return round(x, sig-int(floor(log10(x)))-1)
```

Then the tests look like this:

```
self.failUnless(C.x == Ans.x )
self.failUnless(round_sig(C.y, 2) == round_sig(Ans.y, 2))
```

The first line above won't hold true for large numbers with more than two significant digits. The second line will hold true since it cuts off the computer error, but preservers the significant digits.

1.1.5 Magnitude

The next function needed is finding the magnitude of a vector. The equation for the magnitude of an *Euclidean* 3-space vector is

$$\left|\vec{V}\right| = \sqrt{dx^2 + dy^2 + dz^2} \tag{1.1}$$

The code is still pretty straight forward:

```
@staticmethod
def magnitude(A):
   mag = math.sqrt(A.x**2 + A.y**2 + A.z**2)
   return mag
```

And usage is straight forward too

```
A = Vector(5, 8, 10)
B = Vector(3, 1, 2)
Vector.magnitude(A)                 # = 13.7
Vector.magnitude(B)                 # = 3.7
Vector.magnitude(Vector.sub(A,B))   # = 10.8
```

The last line gives us the distance between $\vec{A}$ and $\vec{B}$.

1.2 The Particle Class

The particle class is the object that will encapsulate the velocity, positional vectors and mass scalar of various point particles. To come will be an attribute for the radius of the particle for collisions and growth but for now we will be working with point particles.

1.2.1 Particle Base

The base class for the particle needs to include a information for where the particle is, how fast it is moving and how heavy it is. Therefore the Particle Class needs three attributes: two vectors and one scalar.

```
from Vector import Vector

class Particle(object):
    def __init__(self, P, V, m):
        self.P = P      #particle's position vector
        self.V = V      #Particle's velocity vector
        self.m = m      #particle's mass
```

1.2.2 Movement

The particle needs to be moved. For each tick of time, Δt, the particle is displaced by its velocity vector. Thus at time t, the particle a has a positional vector $\vec{P}_a(t)$ and velocity vector $\vec{V}_a(t)$. Thus if the position and velocity are know at $t = n$, then the position at $t = n + 1$ can be given by

$$\vec{P}_a(t_{n+1}) = \vec{P}_a(t_n) + \vec{V}_a(t_n) \tag{1.2}$$

Utilizing the previous vector addition function the above equation is easily given by the method function:

```
def move(self):
    new_pos = Vector.add(self.P, self.V)
    self.P = new_pos
```

The usage is really simple. When it comes time to move the particle we simply call move():

```
a = Particle(Vector(1,1,1), Vector(1,2,-3)
a.move()
print a.P.show  #display pos vector
'(2, 3, -2)'
```

The new position is then calculated using the current position and velocity of the vector.

1.2.3 Acceleration

The particle is going to be moving in gravitational fields and therefore must be able to be accelerated. At some particular time t there will be a gravitational force acting on the particle. This force will be a force vector $\vec{F}$. The acceleration will then be found by $\vec{a} = \frac{\vec{F}}{m}$, where m is the mass of the particle in question. The acceleration will not need to be saved as an attribute of the particle because it is calculated for each t_n and does not depend on the acceleration at t_{n-1}. What does need to happen is at each time interval, after the acceleration is calculated, the acceleration vector will be added to the particle's velocity vector. Therefore the velocity of particle A at $t = n + 1$ is given by

$$\vec{V}_a(t_{n+1}) = \vec{V}_a(t_n) + \vec{A}_a(t_n) \tag{1.3}$$

The particle's method function to apply acceleration closely resembles the function to move the particle, however it takes the acceleration as an argument since acceleration is not stored with the particle. The function is given by

```
def accelerate(self, A):
    new = Vector.add(self.V, A)
    self.V = new
```

Usage first requires that the acceleration acting on the particle is calculated. This will come in the next chapter when the physics calculations are formed. For now I will use explicitly define the acceleration vector.

```
Vol_vec = Vector(1, 1, 1)
Acc_vec = Vector(2, 2, -1)
p = Particle(Pos_Vec, Vol_vec, mass)
p.accelerate(Acc_vec)
print p.V.show  #display vol vector
'(3, 3, 0)'
```

1.3 Physics

This section of code is going to contain the Physics class. This class will have a list attribute for keeping track of all the particles and class methods for calculating the force of gravity in the system. The Physics class will depend on the Vector and Particle classes but will be separated for easier editing down the road.

1.3.1 Physics Base

The base class takes no arguments. It has a list as an attribute that holds the various particles in the system. Right away I'm going to add the method for adding particles to the attribute list. The implementation also depends on the Particle class.

```
from Vector import Vector
from Particle import Particle
class Physics(object):
    def __init__(self):
        self.objects = []

    def add_obj(self, obj):
        self.objects.append(obj)
```

To use first create a Physics Class object then particles can be add using the Physics Class method.

```
from libs import Vector, Particle, Physics

system = Physics()
part1 = Particle(Vector(1,1,1), (1,1,1), 5)
part2 = Particle(Vetor(5,.3,2), Vector(.2, 0, .1), 100)

system.add_obj(part1)
system.add_obj(part2)
```

1.3.2 Force of Gravity

Now we are ready to start calculating the force of gravity between two particles. This section is going to use the familiar Newtonian force of gravity,

$$F = \frac{Gm_1m_2}{R^2} \tag{1.4}$$

where,
G = Gravitational constant
m_1 = mass of particle 1
m_2 = mass of particle 2
r = distance between particle 1 and 2

The above equation only gives the magnitude of the force of gravity. The for the simulation the vector form is need. We already have the magnitude, now all we need is direction which is given by the unit vector pointing from particle 1 to particle 2.

Instead of r in the above equation let's instead take $\vec{R}$ and define it as the displacement vector :

$$\vec{R} = \vec{P_2} - \vec{P_1} \tag{1.5}$$

where P_1 and P_2 are the position vectors for both respective particles in question. Then the magnitude of $\vec{R}$ is noted as $\|\vec{R}\|$. Thus the unit vector pointing in the direction of $\vec{R}$ is $\hat{R} = \frac{\vec{R}}{\|\vec{R}\|}$. Now we can rewrite the equation 1.4 as

$$\vec{F} = \frac{Gm_1m_2}{\|\vec{R}\|^2}\frac{\vec{R}}{\|\vec{R}\|} \tag{1.6}$$

or

$$\vec{F} = \frac{Gm_1m_2}{\|\vec{R}\|^2}\hat{R} \tag{1.7}$$

The code for the force of gravity will use the functions from the Vector class for subtracting vectors, multiplying vectors by scalars and finding vector magnitude and unit vectors. The code for finding the force of gravity vector is:

```
@staticmethod
def Fg(A, B):
   G = 6.67384 * 10**(-11)                 # 1
   r = Vector.sub(A.P, B.P)                # 2
   r_hat = Vector.unit(r)                  # 3
   r_squared = Vector.magnitude(r) ** 2    # 4
   f_mag = (G*A.m*B.m)/r_squared           # 5
   f_vec = Vector.times_scalar(f_mag, r_hat) # 6
   return f_vec
```

Here is a breakdown of what each line is doing in the above function.

1 Here the newtons gravitational constant is defined. It is in metric units, so the units are in $\frac{m^3}{KgS^2}$.

2 Here the displacement vector $\vec{R}$, which goes from $\vec{A}$, to $\vec{B}$, is defined. $\vec{R}$ is found by subtracting $\vec{B}$ and $\vec{A}$ in accordance to equation 1.5 above.

3 Here we find the unit vector $\hat{R}$ is found using the Vector.unit() function.

4 Here we find $\|\vec{R}\|^2$ by using Vector.magnitude() and squaring the result. Vector.magnitude() returns a scalar so this is pretty straight forward.

5 Here the magnitude of the force of gravity is calculated. The magnitude of the force of gravity is given by equation 1.4. So we multiply the gravitational constant by both masses and divide by the distance squared which was already found in lines 1 and 4.

6 Finally the vector form of the force is found by multiplying the magnitude of the force of gravity by the unit vector pointing from $\vec{B}$ to$\vec{A}$. This can be summed up by the equation $\vec{F} = \|\vec{F}\| \times \hat{R}$.

The usage here is pretty straight forward. The force function takes two vectors and returns a thrid vector. Below is an example of finding the force vector, Force, between the two particles A and B:

```
A = Particle(Pos_vec, Vol_vec, mass)
B = Particle(Pos_vec, Vol_vec, mass)
Force = Physics.Fg(A,B)
```

1.3.3 Sum Force of Gravity

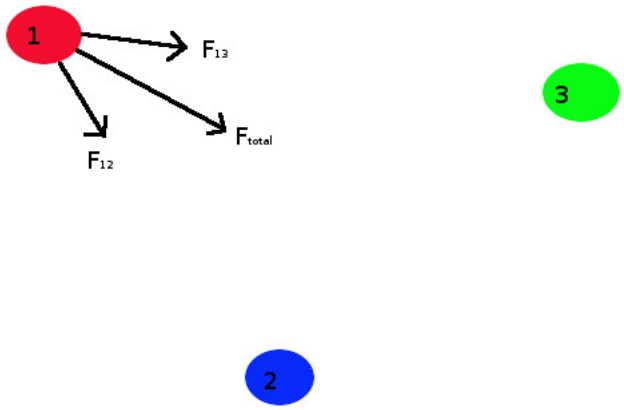
Let's consider a system of particles greater then the simple two object system. Let's call these particles P^1, P^2, P^3 up to some numbered particle P^N. Thus we have a system of particles,

$$\{P^n | \ \ n, N \in \mathbb{N}, \ \ n \leq N\}. \tag{1.8}$$

Now let's consider the force of gravity, hereby abbreviated plainly as 'force', acting on some particular particle. For notation let's say that the force between P^1 and P^2 is written as the vector $\vec{F}_{12}$, and the force between P^2 and P^1 as $\vec{F}_{21}$. The force between some arbitrary particle P^n and P^x can be written as

$$\vec{F}_{nx} \tag{1.9}$$

Now the total force acting on some particle, P^x, can be noted as $\vec{F}_{\sigma x}$.



This total force is given by the sum of all other forces acting on P^x as crudely illustrated above. Therefore $\vec{F}_{\sigma x}$ is given by

$$\vec{F}_{\sigma x} = \sum_{n=1, n \neq x}^{N} \vec{F}_{nx} \tag{1.10}$$

There is already a list of particles in our physics class. So naturally calculating the total force of gravity is going to involve picking a particle then iterating over the set of all particles less the particle in question and summing the forces between. We could also pick a particle and sum over all particles including the particle itself since $\vec{R}$ from the particle to itself will be zero anyway. Even though we are far away from even thinking about optimizations, we can skip calculating force of gravity from a particle to it's self and spare a couple CPU cycles.

So the code for finding the total force of gravity on one particle is as follows with line by line summary below.

```
def sum_Fg_one_particle(self, A):
    force_list = []                                 # 1
    for particle in self.objects:                   # 2
        if particle != A:                           # 3
            force_list.append(self.Fg(particle, A)) # 4
    f = lambda a,b: Vector.add(a,b)                 # 5
    total_force = reduce(f, force_list)             # 6
    return total_force                              # 7
```

1 Create an empty list. This list will hold the vectors for all the the forces that later will be summed.

2 Here we start iterating over all the particles stored in the physics object set.

3 This conditional makes sure we skip over calculating the force of an object to itself.

4 Here we calculate the force of gravity between the particle and one of the particles in the object class, then add the result to the list of forces we set up in line 1

5 Here we use the built in python lambda operator to make a nameless function that wraps the vector addition method from the Vector class.

6 Now use the built in function *reduce* to apply the function from line 5 to the list of forces.

7 Return the total force vector.

This function is really just a helper function for later so I will skip the implementation. It should be pretty obvious how to use it at this point.

1.3.4 Accelerate Particle

The acceleration on a particle from the force of gravity can be found using the basic $F = ma$ equation for force. If we are considering some particle A with mass m and with a total gravitational force acting on it $\vec{F}$, then the particle experiences acceleration $\vec{a}$ such that

$$\vec{a} = \frac{\vec{F}}{m}. \tag{1.11}$$

This can be easily coded as such

```
def apply_gravitational_acceleration(self, A):
    total_Fg = self.sum_Fg_one_particle(A)
    acceleration = Vector.times_scalar( (1/A.m), total_Fg)
    A.accelerate(acceleration)
```

The first line uses the sum of gravity function to find the total force of gravity. The second divides that vector by the mass of the particle according to equation 1.11. The final line uses the method function from the Particle class to apply the acceleration to the particle.

1.4 Examples

1.4.1 Ring of 4 Planets

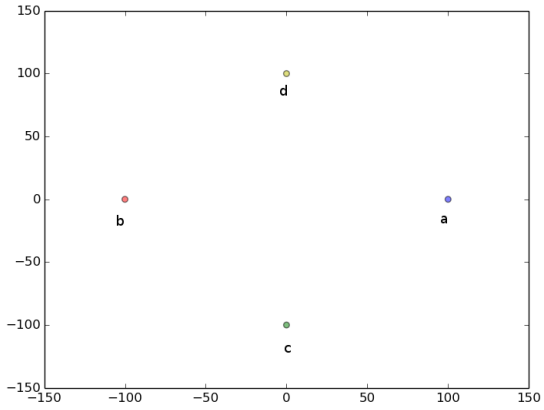
Lets create a ring of 4 planets orbiting each other.

Setup Let's suppose we have 4 particles given by the data file below:

data

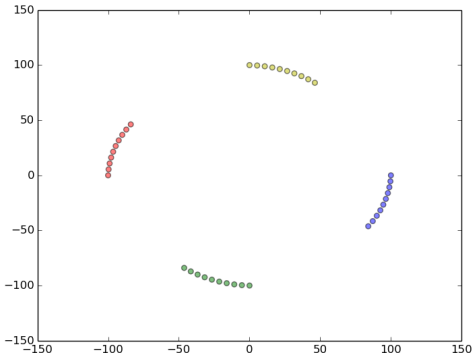
```
a:100:0:0:0:-0.09:0:1.55e10
b:-100:0:0:0:0.09:0:1.55e10
c:0:-100:0:-0.09:0:0:1.55e10
d:0:100:0:0.09:0:0:1.55e10
```

The planets form a square with velocity perpendicular to the line towards the origin.

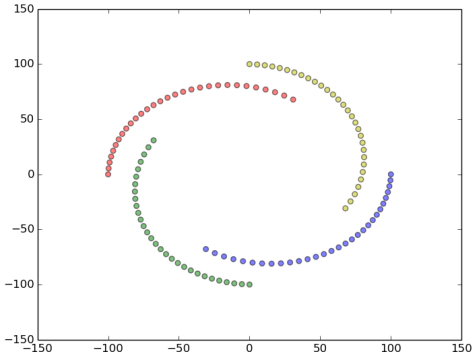


Code

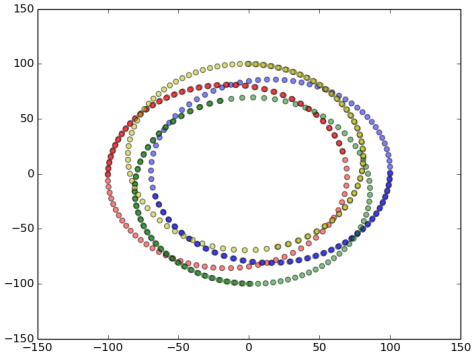
```
import numpy as np
import matplotlib.pyplot as plt
from libs import Physics, Particle, Vector, round_sig

base = Physics()
base.dimension = 3
base.prec = 100
base.timestep = 60
base.read_file('example_data.csv')

ax = []
ay = []
bx = []
by = []
cx = []
cy = []
dx = []
dy = []
for i in range(200):
    if i % 100 == 0:
        print base.objects[1].round(2)
    ax.append(base.objects[0].P[0])
    ay.append(base.objects[0].P[1])
    bx.append(base.objects[1].P[0])
    by.append(base.objects[1].P[1])
    cx.append(base.objects[2].P[0])
    cy.append(base.objects[2].P[1])
    dx.append(base.objects[3].P[0])
    dy.append(base.objects[3].P[1])

    base.step_all()

plt.scatter(ax, ay, s=10, c='b', alpha=0.5)
plt.scatter(bx, by, s=10, c='r', alpha=0.5)
plt.scatter(cx, cy, s=10, c='g', alpha=0.5)
plt.scatter(dx, dy, s=10, c='y', alpha=0.5)
plt.show()
```



10 mins



30 mins



120 mins

Bibliography

[1] indgar. *How to round a number to significant figures in Python*. `http://stackoverflow.com/questions/3410976/how-to-round-a-number-to-significant-figures-in-python`. Aug 2010.