PyGravity

http://github.com/russloewe/PyGravity

Russell Loewe

7 May 2015

This is the supporting documentation for *PyGravity*

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.

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

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)'
```

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
```

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)
```

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].

```
from math import log10, floor

def round_sig(x, sig=2):
   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.

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.

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      #particles position vector
        self.V = V      #Particles velocity vector
        self.m = m      #particles mass
```

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.

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 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 first 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)'
```

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.