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.

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.

0.2 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
```

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 need so only significant figures would be compared. Here is a quick rounding to signifigant 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.

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.