Question

Just a little disclaimer beforehand: I have never studied astronomy or any exact sciences for that matter (not even IT), so I am trying to fill this gap by self-education. Astronomy is one of the areas that has captured my attention and my idea of self-education is head on applied approach. So, straight to the point - this is orbital simulation model that I am casually working on when I have time/mood. My major goal is to create complete solar system in motion and ability to plan spacecraft launches to other planets.

You are all free to pick this project up at any point and have fun experimenting!

update!!! (Nov10)

- velocity is now proper deltaV and giving additional motion now calculates sum vector of velocity
- you can place as many static objects as you like, on every time unit object in motion checks for gravity vectors from all sources (and checks for collision)
- greatly improved the performance of calculations
- a fix to account for interactive mod in matplotlib. Looks like that this is default option only for ipython. Regular python3 requires that statement explicitly.

Basically it is now possible to "launch" a spacecraft from the surface of the Earth and plot a mission to the Moon by making deltaV vector corrections via giveMotion(). Next in line is trying to implement global time variable to enable simultaneous motion e.g. Moon orbits Earth while spacecraft tries out a gravity assist maneuver.

Comments and suggestions for improvements are always welcome!

Done in Python3 with matplotlib library

```
import matplotlib.pyplot as plt
import math
plt.ion()

G = 6.673e-11  # gravity constant
gridArea = [0, 200, 0, 200]  # margins of the coordinate grid
gridScale = 1000000  # 1 unit of grid equals 1000000m or 1000km

plt.clf()  # clear plot area
plt.axis(gridArea)  # create new coordinate grid
plt.grid(b="on")  # place grid

class Object:
```

```
_instances = []
def __init__(self, name, position, radius, mass):
    self.name = name
    self.position = position
    self.radius = radius  # in grid values
    self.mass = mass
    self.placeObject()
    self.velocity = 0
    Object._instances.append(self)

def placeObject(self):
    drawObject = plt.Circle(self.position, radius=self.radius, fill=False, color="black"
    plt.gca().add_patch(drawObject)
    plt.show()

def giveMotion(self, deltaV, motionDirection, time):
    if self.velocity != 0:
        x_comp = math.sin(math.radians(self.motionDirection))*self.velocity
        y_comp = math.cos(math.radians(self.motionDirection))*self.velocity
        x_comp += math.sin(math.radians(motionDirection))*deltaV
        y_comp += math.cos(math.radians(motionDirection))*deltaV
        self.velocity = math.sqrt((x_comp**2)+(y_comp**2))

        if x_comp > 0 and y_comp > 0:  # calculate degrees depending on the coordinate c
            self.motionDirection = math.degrees(math.asin(abs(x_comp)/self.velocity))  #
        elif x_comp > 0 and y_comp < 0:
            self.motionDirection = math.degrees(math.asin(abs(y_comp)/self.velocity)) +
        elif x_comp < 0 and y_comp < 0:
            self.motionDirection = math.degrees(math.asin(abs(x_comp)/self.velocity)) +
        else:
            self.motionDirection = math.degrees(math.asin(abs(y_comp)/self.velocity)) +

    else:
        self.velocity = self.velocity + deltaV  # in m/s
        self.motionDirection = motionDirection  # degrees
    self.time = time  # in seconds
    self.vectorUpdate()

def vectorUpdate(self):
    self.placeObject()
    data = []

    for t in range(self.time):
        motionForce = self.mass * self.velocity  # F = m * v
        x_net = 0
        y_net = 0
```

```
for x in [y for y in Object._instances if y is not self]:
    distance = math.sqrt(((self.position[0]-x.position[0])**2) +
                 (self.position[1]-x.position[1])**2)
    gravityForce = G*(self.mass * x.mass)/((distance*gridScale)**2)

    x_pos = self.position[0] - x.position[0]
    y_pos = self.position[1] - x.position[1]

    if x_pos <= 0 and y_pos > 0:  # calculate degrees depending on the coordinat
        gravityDirection = math.degrees(math.asin(abs(y_pos)/distance))+90

    elif x_pos > 0 and y_pos >= 0:
        gravityDirection = math.degrees(math.asin(abs(x_pos)/distance))+180

    elif x_pos >= 0 and y_pos < 0:
        gravityDirection = math.degrees(math.asin(abs(y_pos)/distance))+270

    else:
        gravityDirection = math.degrees(math.asin(abs(x_pos)/distance))

    x_gF = gravityForce * math.sin(math.radians(gravityDirection))  # x componen
    y_gF = gravityForce * math.cos(math.radians(gravityDirection))  # y componen

    x_net += x_gF
    y_net += y_gF

x_mF = motionForce * math.sin(math.radians(self.motionDirection))
y_mF = motionForce * math.cos(math.radians(self.motionDirection))
x_net += x_mF
y_net += y_mF
netForce = math.sqrt((x_net**2)+(y_net**2))

if x_net > 0 and y_net > 0:  # calculate degrees depending on the coordinate qua
    self.motionDirection = math.degrees(math.asin(abs(x_net)/netForce))  # updat
elif x_net > 0 and y_net < 0:
    self.motionDirection = math.degrees(math.asin(abs(y_net)/netForce)) + 90
elif x_net < 0 and y_net < 0:
    self.motionDirection = math.degrees(math.asin(abs(x_net)/netForce)) + 180
else:
    self.motionDirection = math.degrees(math.asin(abs(y_net)/netForce)) + 270

self.velocity = netForce/self.mass  # update velocity
traveled = self.velocity/gridScale  # grid distance traveled per 1 sec
self.position = (self.position[0] + math.sin(math.radians(self.motionDirection))
                 self.position[1] + math.cos(math.radians(self.motionDirection))
data.append([self.position[0], self.position[1]])
```

```
            collision = 0
            for x in [y for y in Object._instances if y is not self]:
                if (self.position[0] - x.position[0])**2 + (self.position[1] - x.position[1]
                    collision = 1
                    break
            if collision != 0:
                print("Collision!")
                break

        plt.plot([x[0] for x in data], [x[1] for x in data])

Earth = Object(name="Earth", position=(50.0, 50.0), radius=6.371, mass=5.972e24)
Moon = Object(name="Moon", position=(100.0, 100.0), radius=1.737, mass = 7.347e22)  # posit
Craft = Object(name="SpaceCraft", position=(49.0, 40.0), radius=1, mass=1.0e4)

Craft.giveMotion(deltaV=8500.0, motionDirection=100, time=130000)
Craft.giveMotion(deltaV=2000.0, motionDirection=90, time=60000)
plt.show(block=True)
```

How it works

It all boils down to two things:

1. Creating object like `Earth = Object(name="Earth", position=(50.0, 50.0), radius=6.371, mass=5.972e24)` with parameters of position on grid (1 unit of grid is 1000km by default but this can be changed too), radius in grid units and mass in kg.
2. Giving object some deltaV such as `Craft.giveMotion(deltaV=8500.0, motionDirection=100, time=130000)` obviously it requires `Craft = Object(...)` to be created in the first place as mentioned in previous point. Parameters here are `deltaV` in m/s (note that for now acceleration is instantaneous), `motionDirection` is direction of deltaV in degrees (from current position imagine 360 degree circle around object, so direction is a point on that circle) and finally parameter `time` is how many seconds after the deltaV push trajectory of the object will be monitored. Subsequent `giveMotion()` start off from last position of previous `giveMotion()`.

Questions:

1. Is this a valid algorithm to calculate orbits?
2. What are the obvious improvements to be made?
3. I have been considering "timeScale" variable that will optimize calculations, as it might not be necessary to recalculate vectors and positions for every second. Any thoughts on how it should be implemented or is it generally a good idea? (loss of accuracy vs improved performance)

Basically my aim is to start a discussion on the topic and see where it leads. And, if possible, learn (or even better - teach) something new and interesting.

Feel free to experiment!

Try using:

```
Earth = Object(name="Earth", position=(50.0, 100.0), radius=6.371, mass=5.972e24)
Moon = Object(name="Moon", position=(434.0, 100.0), radius=1.737, mass = 7.347e22)
Craft = Object(name="SpaceCraft", position=(43.0, 100.0), radius=1, mass=1.0e4)

Craft.giveMotion(deltaV=10575.0, motionDirection=180, time=322000)
Craft.giveMotion(deltaV=400.0, motionDirection=180, time=50000)
```

With two burns - one prograde at Earth orbit and one retrograde at Moon orbit I achieved stable Moon orbit. Are these close to theoreticaly expected values?

Suggested exercise: Try it in 3 burns - stable Earth orbit from Earth surface, prograde burn to reach Moon, retrograde burn to stabilize orbit around Moon. Then try to minimize deltaV.

Note: I plan to update the code with extensive comments for those not familiar with python3 syntax.

Reply

Let the two bodies involved have masses m_1, m_2. Start with Newton's second law

$$F = ma$$

where a is acceleration. The gravitational force on body 2 from body 1 is given by

$$F_{21} = \frac{Gm_1m_2}{|r_{21}|^3} r_{21}$$

where r_{21} is the relative position vector for the two bodies in question. The force on body 1 from body two is of course $F_{12} = -F_{21}$ since $r_{12} = -r_{21}$. If we denote the positions by (x_1, y_1) and (x_2, y_2), then

$$r_{21} = \begin{pmatrix} x_1 - x_2 \\ y_1 - y_2 \end{pmatrix}.$$

and

$$|r| = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}.$$

Then Newton's law $a = F/m$ becomes

$$x_1''(t) = \frac{Gm_2(x_2 - x_1)}{|r|^3} \tag{1}$$

$$y_1''(t) = \frac{Gm_2(y_2 - y_1)}{|r|^3} \tag{2}$$

$$x_2''(t) = \frac{Gm_1(x_1 - x_2)}{|r|^3} \tag{3}$$

$$y_2''(t) = \frac{Gm_1(y_1 - y_2)}{|r|^3}. \tag{4}$$

Together with the initial positions and velocities, this system of ordinary differential equations (ODEs) comprises an initial value problem. The usual approach is to write this as a first-order system of 8 equations and apply a Runge-Kutta or multistep method to solve them.

If you apply something simple like forward Euler or backward Euler, you will see the Earth spiral out to infinity or in toward the sun, respectively, but that is an effect of the numerical errors. If you use a more accurate method, like the classical 4th-order Runge-Kutta method, you'll find that it stays close to a true orbit for a while but still eventually goes off to infinity. The right approach is to use a symplectic method, which will keep the Earth in the correct orbit – although its phase will still be off due to numerical errors.

For the 2-body problem it's possible to derive a simpler system by basing your coordinate system around the center of mass. But I think the formulation above is clearer if this is new to you.