Simulation.py

November 12, 2017

```
In [1]: '''
        Runs a simulation to approximate the best path for a solar sail flight to Mars
        '''
        import math
        from math import sin, cos

        class Planet:
            '''
            Instantiates a planet object for easy calculation and tracking of celestial orbits
            '''
            def __init__(self, semimajor_axis, eccentricity, inclination, mean_longitude, l_pe
                '''
                Parameters up to current_time given in Keplerian Elements by E M Standish. Eac
                and the second value being the rate of change in the value with respect to tim
                The primary units for each non-self paramater are as follows: au, none, deg, d
                J2000.0. current_time is the Julian Date of closest approach. Gravitational_pa
                the planet is orbiting (sun).
                '''
                self.semimajor_axis = semimajor_axis
                self.eccentricity = eccentricity
                self.inclination = inclination
                self.l_ascending = l_ascending
                self.l_perihelion = l_perihelion
                self.mean_longitude = mean_longitude
                self.start_time = current_time
                self.mu = gravitational_parameter
                self.orbiting = orbiting
                if current_time: # In case it's a planet
                    self.location = self.GetLocation(current_time)
                    self.direction = Vectors.direction(self.GetLocation(current_time-1), self.
                else: # In case it's the sun
                    self.location = (0, 0, 0)
                    self.direction = (0, 0, 0)

            def GetLocation(self, days):
                '''
                Follows algorithm given on pages 1,2  of Keplerian Elements by E M Standish. S
```

```python
        Days is Julian Days
        '''
        time = self.NormalizeTime(days)
        self.a = (self.semimajor_axis[0] + self.semimajor_axis[1]*time)*149597870.7 #
        e = self.eccentricity[0] + self.eccentricity[1]*time
        I = self.inclination[0] + self.inclination[1]*time
        L = self.mean_longitude[0] + self.mean_longitude[1]*time
        ohm_bar = self.l_perihelion[0] + self.l_perihelion[1]*time
        OHM = self.l_ascending[0] + self.l_ascending[1]*time
        ohm = ohm_bar - OHM
        M = L - ohm_bar
        E = math.radians(self.GetEccentricAnomaly(M, e))
        I = math.radians(I)
        ohm = math.radians(ohm)
        OHM = math.radians(OHM)

        x_prime = self.a*(cos(E) - e)
        y_prime = self.a*math.sqrt(1-e**2)*sin(E)
        x = (cos(ohm)*cos(OHM)-sin(ohm)*sin(OHM)*cos(I))*x_prime
        x = x + (-sin(ohm)*cos(OHM) - cos(ohm)*sin(OHM)*cos(I))*y_prime
        y = (cos(ohm)*cos(OHM)+sin(ohm)*cos(OHM)*cos(I))*x_prime
        y = y + (-sin(ohm)*sin(OHM) + cos(ohm)*cos(OHM)*cos(I))*y_prime
        z = sin(ohm)*sin(I)*x_prime + cos(ohm)*sin(I)*y_prime
        return (x, y, z)

    def NormalizeTime(self, days):
        '''
        Since all time related units in __init__ are given by unit/centuries, this con
        a usable centuries since J2000.0 unit
        '''
        return (days - 2451545.0)/36525

    def GetEccentricAnomaly(self, M, e):
        '''
        Algorithm given by equations 8-36 and 8-37 of Keplerian Elements by E M Standi
        '''
        if M > 180:
            M -= 360
        e_star = e
        e = math.radians(e)
        incr = 1
        E = M + e_star*sin(M)
        dE = 1
        tol = 1e-6
        while abs(dE) > tol:
            dM = M - (E - e_star*sin(E))
            dE = dM/(1-e*cos(E))
            E = E + dE
```

```
        return E

    def move(self, dt = 1):
        '''
        Moves the planet through it's orbit. dt is the number of days to move ahead by
        '''
        old = self.location
        new = self.GetLocation(dt + self.start_time)
        heading = (new[0]-old[0], new[1]-old[1], new[2]-old[2])
        self.location = new
        self.direction = Vectors.direction(old, new)

    def GetForce(self, obj):
        '''
        In case of more advanced simulation, this can be finished to get the gravitati
        '''
        pass
        '''
        return (self.mu * obj.mass/(Vectors.distance(obj.location, self.location)**2),
        '''

    def GetVelocity(self):
        '''
        Returns the velocity of the planet as (magnitude, direction unit vector (rel t
        '''
        return (math.sqrt(self.orbiting.mu*(2/Vectors.magnitude(self.location) - 1/sel

def lcm(n1, n2):
    '''
    Calculates least common multiple in case that ends up being useful
    '''
    return int((n1 * n2) / math.gcd(n1, n2))

class Vectors:
    '''
    Helps to keep up with all the vector math
    '''
    @staticmethod
    def distance(v1, v2):
        temp = 0
        for i in range(len(v1)):
            temp += (v1[i] - v2[i])**2
        return math.sqrt(temp)

    @staticmethod
    def magnitude(vect):
        raw = 0
```

```
        for c in vect:
            raw += c**2
        return math.sqrt(raw)

    @staticmethod
    def unit_vector(vect):
        mag = Vectors.magnitude(vect)
        return tuple([c/mag for c in vect])

    @staticmethod
    def direction(vect1, vect2):
        raw = []
        for i in range(len(vect1)):
            raw.append(vect1[i]-vect2[i])
        mag = Vectors.magnitude(raw)

        final = []
        for c in raw:
            final.append(c/mag)
        return tuple(final)

    @staticmethod
    def dot(vect1, dir_vect):
        result = 0
        for i in range(len(vect1)):
            result += vect1[i]*dir_vect[i]
        return result

    @staticmethod
    def true_vector(mag, dir_vect):
        results = []
        if isinstance(mag, tuple):
            mag = Vectors.magnitude(mag)
        for d in dir_vect:
            results.append(mag*d)
        return tuple(results)

    @staticmethod
    def add(vect1, vect2):
        return tuple([vect1[i]+vect2[i] for i in range(len(vect1))])


class Ship:
    '''
    Class to instantiate the space ship in order to keep track of its motion
    '''
    def __init__(self, r_sail, initial_velocity, current_position, target, mass = 2000)
        self.sail_area = math.pi * r_sail**2
```

```
        self.velocity = initial_velocity
        self.location = current_position
        self.payload = mass - self.sail_area*7/1000
        self.target = target

    def move(self):
        pass


sun = Planet((0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), 0, 1.32712428e11)

earth = Planet((1.00000261, 0.00000562), (0.01671123, -0.00004392), (-0.00001531, -0.0
               (100.46457166, 35999.37244981), (102.93768193, 0.32327364), (0.0, 0.0),

mars = Planet((1.52371034, 0.00001847), (0.09339410, 0.00007882), (1.84969142, -0.0081
              (-4.55343205, 19140.30268499), (-23.94362959, 0.44441088),
              (49.55953891, -0.29257343), 2487972, 0.042828e6, sun)

'''
Setting up initial values for the spacecraft
'''
earth_velocity = earth.GetVelocity()
mars_velocity = mars.GetVelocity()
distance_e_m = Vectors.distance(earth.location, mars.location)
target = Vectors.add(Vectors.true_vector(distance_e_m, mars_velocity[1]), mars.locatio

true_escape_velocity = Vectors.true_vector(11.2, Vectors.direction(earth.location, targ

sc_velocity = Vectors.add(true_escape_velocity, Vectors.true_vector(earth_velocity[0],
print(sc_velocity)
spacecraft = Ship(10, sc_velocity, earth.location, mars)
```

```
(-8.607475024728576, 7.087991763243718, 0.8124236316338541)
```