# Dexter Watkins  Propulsion - Homework 1

## Problem Definition

Calculate radius, velocity, and acceleration as a function of time for a spacecraft moving radially (and only radially) away from the earth. Use a numerical method to solve this problem.  Examine the case with and without onboard propulsion

## Specified Properties and Assumptions

1. The vehicle is initially moving at escape velocity (11.2 km/s)
2. The vehicle starts at earth's surface (R0 = 6400 km)
3. Non-conservative forces can be ignored (earth's atmosphere)
4. The vehicle's mass is 400kg, 80kg of which is fuel
5. If onboard propulsion is used, fuel consumption is 3.33e-6 kg/s
6. If onboard propulsion is used, 70 mN of thrust is generated.
7. The vehicle's flight can be modeled as a 2-body problem

## Governing Equations

The equation of motion for the flight vehicle is determined by the following system of equations:

$$r'' = -g0 * R0^2/r^2 + Thrust/(m(t))$$

$$m' = -3.33 * 10^-6$$

For easier use, the system of equations is rewritten as a system of purely first order equations, as follows:

$$y_1' = y_2$$

$$y_2' = -g0 * R0^2/r^2 + Thrust/(y_3)$$

$$y_3' = -3.33 * 10^-6$$

Where $y_1$ is radius, $y_2$ is velocity, and $y_3$ is mass.

## Problem Setup

A custom matlab class named "Vehicle" is used to encapsulate vehicle properties such as fuel mass, fuel consumption rate, thrust produced, and the vehicle's state vector. While overkill for this particular problem, this custom class will provide a good foundation for future assignments, and more importantly my own research. As such, this class will be further developed and refined as the semester continues. For further documentation, see the section "Vehicle Class Description".

The system of differential equations will be solved using matlab's built in ode45 solver. The ode45 solver gets it name from the 4th and 5th order runge kutta numerical techniques it uses. Ode45 first estimates a time step and approximates a solution using the 4th order method. The 5th order method is then evaluated and the difference between the two is taken to be the method's error. This error is compared to the user-specified error tolerance (default 0.1%). If the error is too large, a smaller time step is chosen and the solution re-evaluated.

In order to use ode45, a function must be defined that calculates the system of equations in the following way:

$y'_1 = f_1(t, y) \ldots y'_n = f_n(t, y)$

As a system of equations that satisfies this criterion has already been developed, all that is needed are initial conditions and the start and end times.

The following code initializes the unit and coordinate system, celestial constands, and vehicle properties. Two vehicles corresponding to the two cases evaluated are then instantiated.

Define units and coordinate system

```
units = 'Metric';
CS = 'Spherical';
```

Define celestial body constants

```
g0 = 9.81; %m/s^2
R0 = 6400000; %m
```

Define vehicle properties

```
vehicleMass = 400; %kg
fuelMass = 80; %kg
dmdt = -3.33e-6; %kg/sec
Thrust = 0.07; %N
```

Construct vehicle state vector

```
r = R0; %m initial radius
escapeVelocity = sqrt(2*g0*R0^2/r); %m/s
```

```
drdt = escapeVelocity; %initial radial velocity m/s
sv = [0 0 r 0 0 drdt 0 0 0]; %state vector [pos vel accel]
```

Instantiate vehicles

```
baseVehicle = Vehicle(vehicleMass, 0, 0, sv, 0, CS, units); %base w/o Thrust
thrustVehicle = Vehicle(vehicleMass, fuelMass,...
    dmdt, sv, Thrust, CS, units); %vehicle w/Thrust
```

## Solver Execution

After defining a time span over which to solve the problem, the ode45 solver is ready to be run. The homework1_propagate() defined within the vehicle class is called, which contains the ode45 solver setup.

Define/Calculate Time Spans

```
t0 = 0; %s initial time
tf = fuelMass/abs(dmdt); %s final time when fuel
```

Run ode45 solver

```
baseSolution = baseVehicle.homework1_propagate([t0 tf]);
thrustSolution = thrustVehicle.homework1_propagate([t0 tf]);
```

Plot and discuss results

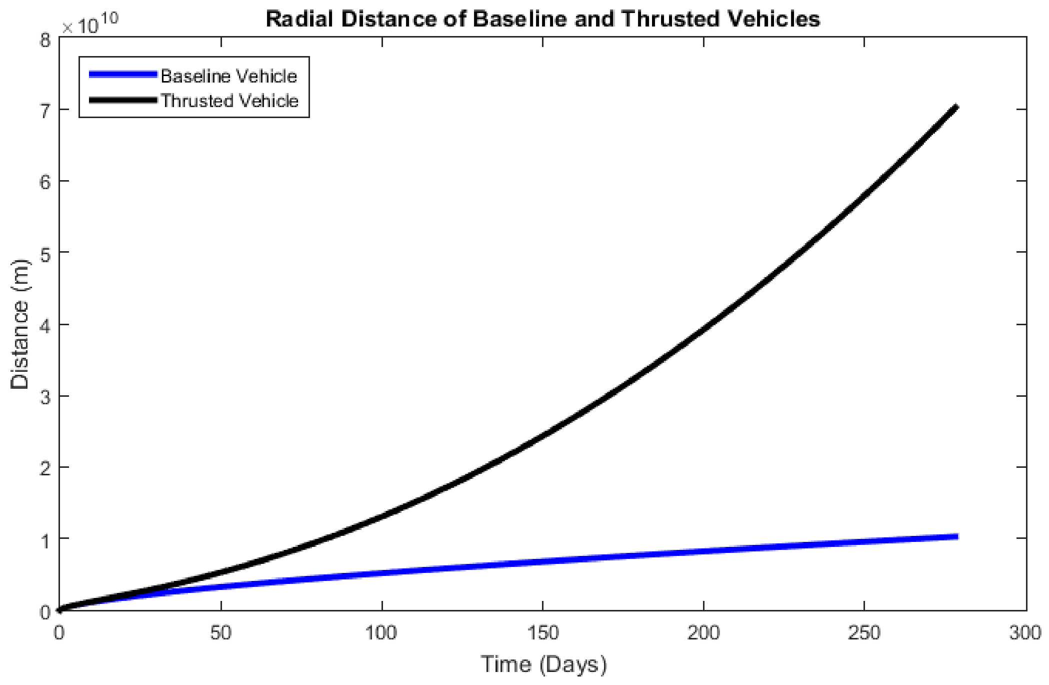
```
%set line width
lw = 3;
```

## Position Analysis

While the baseline vehicle appears to be asymptotically approaching some value, it is actually unbounded and will increase to infinity.This is due to the fact that the vehicle was initially moving at escape velocity, meaning that it's velocity will only be zero at infinity.

The thrusted vehicle, on the other hand, will clearly continue its outward trajectory indefinitely, as earth's gravity is quickly reduced to a value so small as to be considered negligible compared to thrust.

```
posfig = figure(1);
plot(baseSolution.time, baseSolution.position, 'b',...
    thrustSolution.time, thrustSolution.position, 'k','LineWidth',lw);
posfig.Units = 'normalized';
posfig.Position = [0.05 0.5 0.42 0.42];
title('Radial Distance of Baseline and Thrusted Vehicles');
ylabel('Distance (m)');
xlabel('Time (Days)');
legend('Baseline Vehicle', 'Thrusted Vehicle', 'Location', 'NorthWest');
```

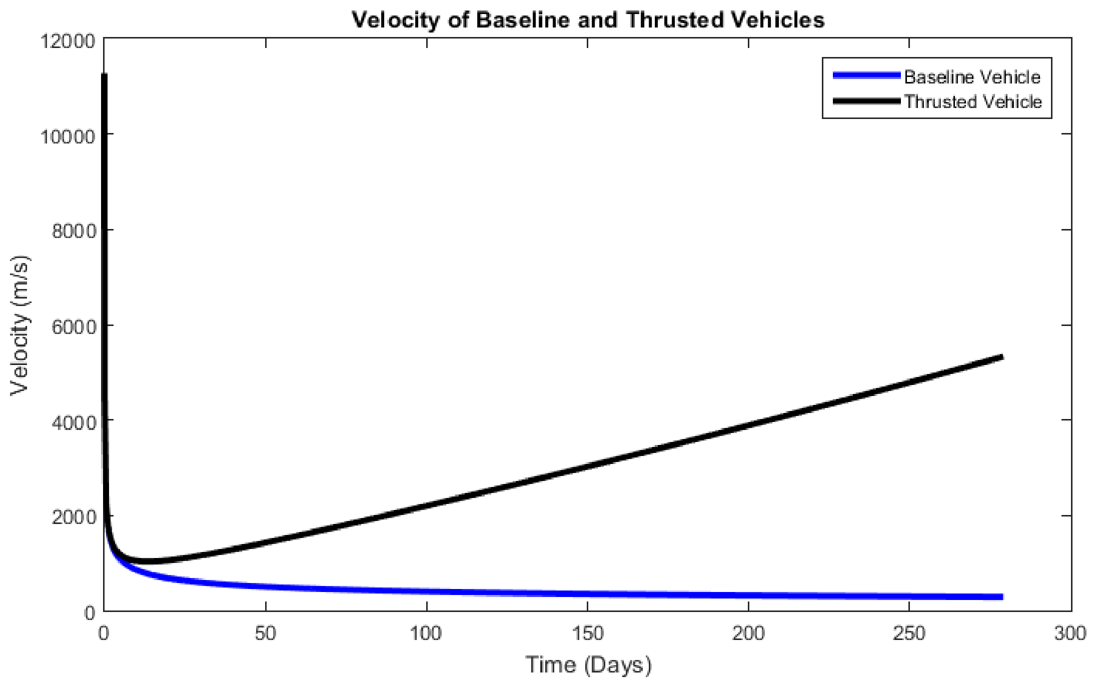


## Velocity Analysis

Again, the velocity of the baseline vehicle approaches zero, but since it was initially going at escape velocity, it will only be zero at infinity

The thrusted vehicle's velocity initially decays in the same way as the unthrusted, since earth's gravity is initially much greater than its thruster. But as radius increases, earth's gravity decrease quadratically and the vehicle's thrust sooner overtakes it and starts to acclerate the vehicle.

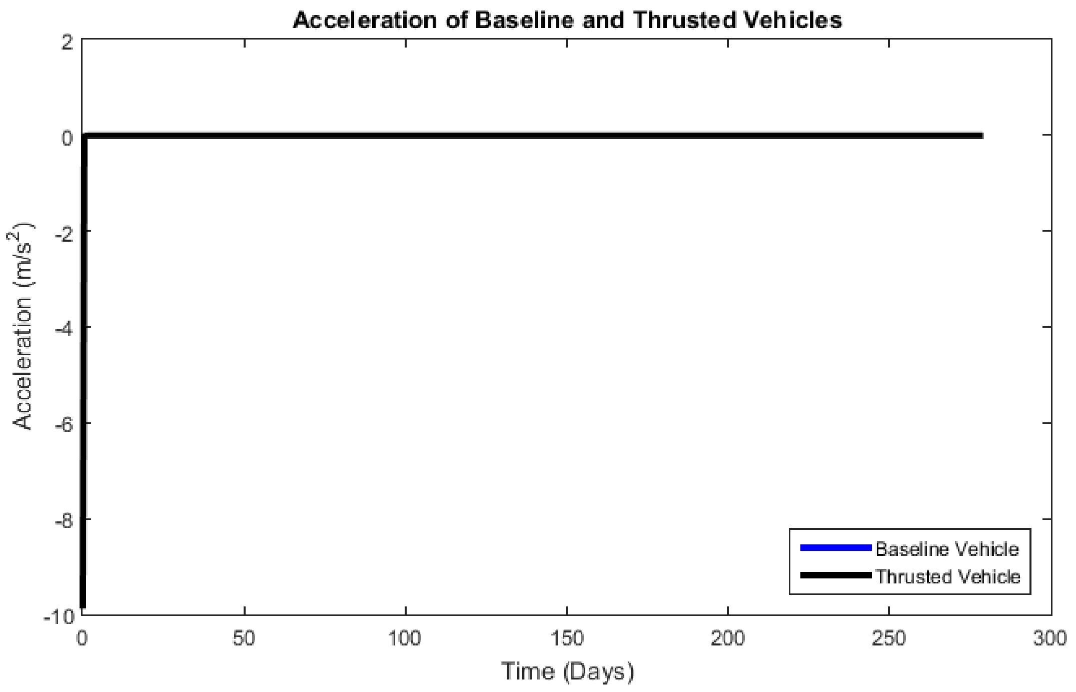
```
velfig = figure(2);
plot(baseSolution.time, baseSolution.velocity, 'b',...
    thrustSolution.time, thrustSolution.velocity, 'k','LineWidth',lw);
velfig.Units = 'normalized';
velfig.Position = [0.55 0.5 0.42 0.42];
title('Velocity of Baseline and Thrusted Vehicles');
ylabel('Velocity (m/s)');
xlabel('Time (Days)');
legend('Baseline Vehicle', 'Thrusted Vehicle');
```



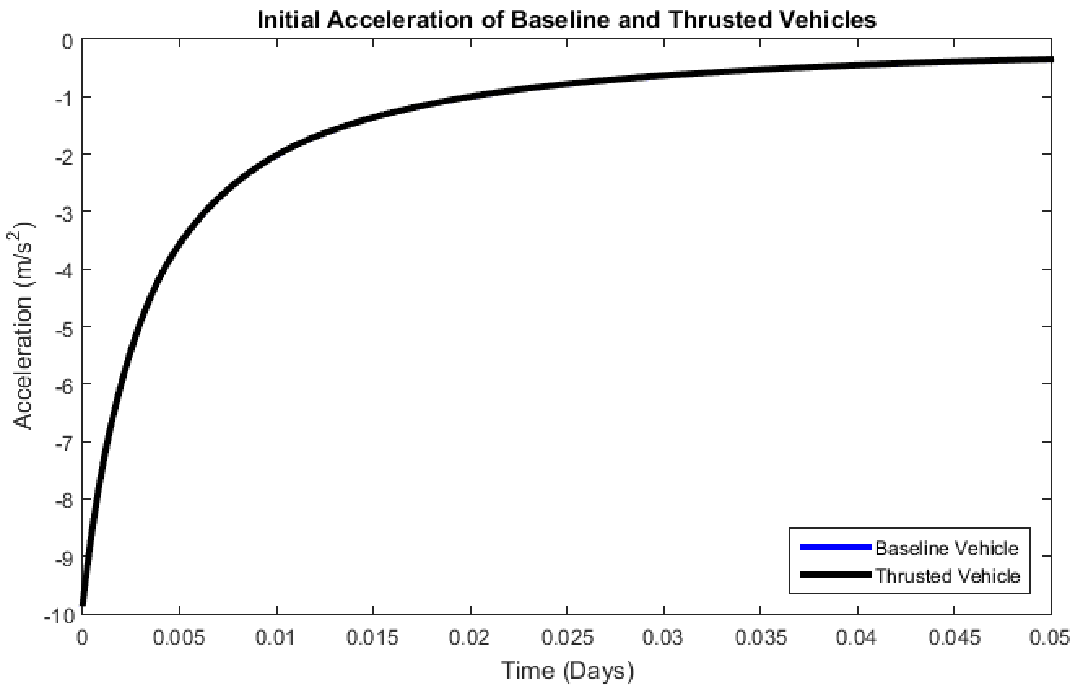
## Acceleration Analysis

As can be seen in the following three graphs, the vehicle's acceleration is initially dominated by earth's presence, before leveling out to nearly zero in the base case or nearly 70mn/vehicle_mass in the thrusted case. In the early time stages, the baseline and thrusted vehicle's acceleration is almost indistinguishable.

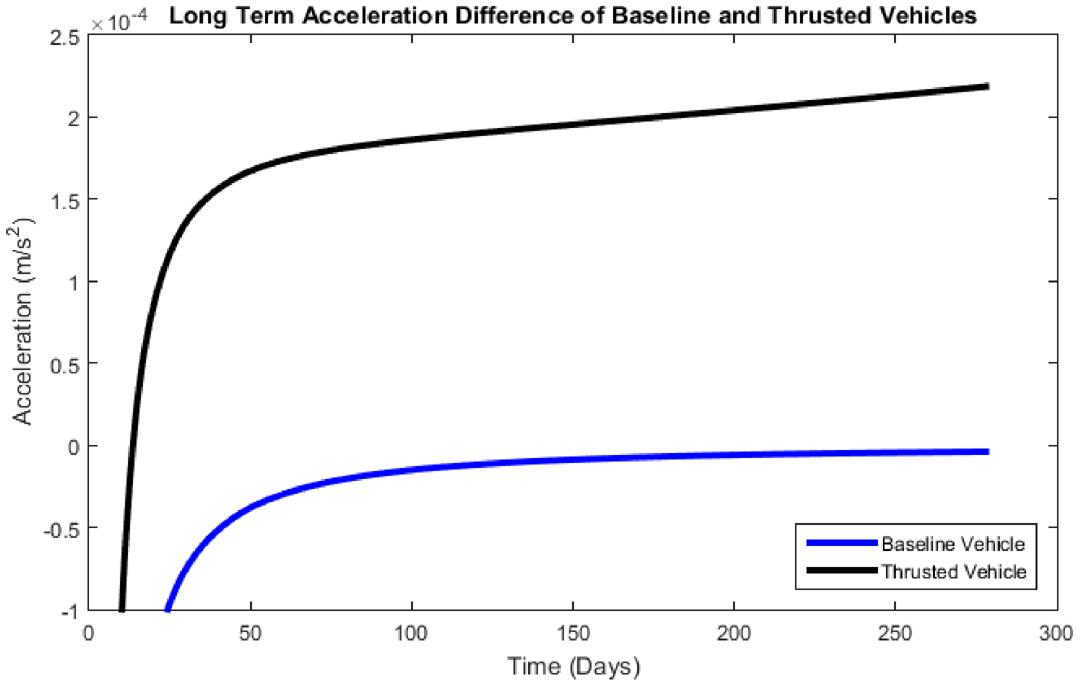
```
accelfig = figure(3);
plot(baseSolution.time, baseSolution.acceleration, 'b',...
    thrustSolution.time, thrustSolution.acceleration, 'k','LineWidth',lw);
accelfig.Units = 'normalized';
accelfig.Position = [0.05 0.02 0.40 0.40];
title('Acceleration of Baseline and Thrusted Vehicles');
ylabel('Acceleration (m/s^2)');
xlabel('Time (Days)');
legend('Baseline Vehicle', 'Thrusted Vehicle', 'Location', 'SouthEast');

accelfig_init = figure(4);
plot(baseSolution.time, baseSolution.acceleration, 'b',...
    thrustSolution.time, thrustSolution.acceleration, 'k','LineWidth',lw);
accelfig_init.Units = 'normalized';
accelfig_init.Position = [0.05 0.02 0.40 0.40];
accelfig_init.CurrentAxes.XLim = [0 0.05];
title('Initial Acceleration of Baseline and Thrusted Vehicles');
ylabel('Acceleration (m/s^2)');
xlabel('Time (Days)');
legend('Baseline Vehicle', 'Thrusted Vehicle', 'Location', 'SouthEast');

accelfig_end = figure(5);
plot(baseSolution.time, baseSolution.acceleration, 'b',...
    thrustSolution.time, thrustSolution.acceleration, 'k','LineWidth',lw);
accelfig_end.Units = 'normalized';
accelfig_end.Position = [0.05 0.02 0.40 0.40];
accelfig_end.CurrentAxes.YLim = [-1e-4 2.5e-4];
title('Long Term Acceleration Difference of Baseline and Thrusted Vehicles');
ylabel('Acceleration (m/s^2)');
xlabel('Time (Days)');
legend('Baseline Vehicle', 'Thrusted Vehicle', 'Location', 'SouthEast');
```



Acceleration of Baseline and Thrusted Vehicles
Acceleration (m/s²)
Time (Days)
2
0
-2
-4
-6
-8
-10
0
50
100
150
200
250
300
Baseline Vehicle
Thrusted Vehicle



Initial Acceleration of Baseline and Thrusted Vehicles
Acceleration (m/s²)
Time (Days)
0
-1
-2
-3
-4
-5
-6
-7
-8
-9
-10
0
0.005
0.01
0.015
0.02
0.025
0.03
0.035
0.04
0.045
0.05
Baseline Vehicle
Thrusted Vehicle

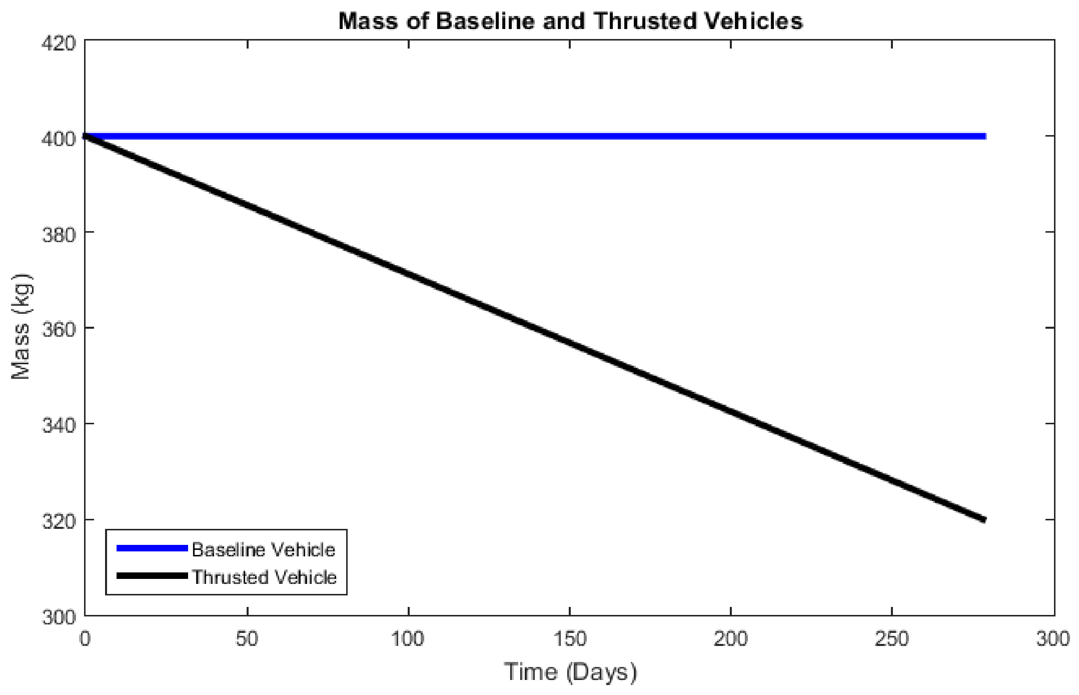

Long Term Acceleration Difference of Baseline and Thrusted Vehicles
×10^-4
2.5
2
1.5
1
0.5
0
-0.5
-1
Acceleration (m/s^2)
0
50
100
150
200
250
300
Time (Days)
Baseline Vehicle
Thrusted Vehicle

## Mass Analysis

Mass is constant with time for the base case, since there is no propulsion and therefore no fuel consumption

Mass decreases linearly for the propulsion case, as defined in the problem setup

```
massfig = figure(6);
plot(baseSolution.time, baseSolution.mass, 'b',...
    thrustSolution.time, thrustSolution.mass, 'k','LineWidth',lw);
massfig.Units = 'normalized';
massfig.Position = [0.55 0.02 0.40 0.40];
massfig.CurrentAxes.YLim = [300 420];
title('Mass of Baseline and Thrusted Vehicles');
ylabel('Mass (kg)');
xlabel('Time (Days)');
legend('Baseline Vehicle', 'Thrusted Vehicle', 'Location', 'SouthWest');
```



## Vehicle Class

```
%The following is a matlab class defining a vehicle that contains vehicle
%properties, the ability to switch between english and metric units, the
%ability to switch between cartesian and spherical coordinates, and will
%contain built in functions that will aid in Propulsion assignments and
%personal research.


classdef Vehicle

    properties(Constant, Access = private)
        ConversionFactors = [.3048,...  %ft/m
            2.20462,... %lbm/kg
            0.224809];    %lbf/N
        UnitConversionIndex = {'Mass', 2;...   %index of proper conversion factor
            'FuelMass', 2;...
            'FuelMassDot', 2;...
            'Sv',1;...
            'Thrust', 3;...
            }
    end

    properties
        Mass %total mass of vehicle
        FuelMass
        FuelMassDot %fuel consumption rate
        Sv %1x9 3D state vector [position velocity acceleration]
        Thrust %thrust generated by vehicle
    end

    properties(Dependent)
        Units %Unit system used (Metric, English)
        CoordinateSystem %Coordinate system used (Cartesian, Spherical)
        %        g %earth's gravitational acceleration felt by vehicle
    end

    properties(Access = private)
        PrivateUnits = 'Metric';
        PrivateCoordinateSystem = 'Cartesian';
    end

    methods
```

```
        %       Ctor*******************************************************
        function obj = Vehicle(mass,fuelmass,fuelmassdot,sv,Thrust,CS,Units)
            if nargin > 0
                switch(CS)
                    case 'Cartesian'
                    case 'Spherical'
                        obj.PrivateCoordinateSystem = CS;
                    otherwise
                        error('Vehicle:InvalidCoordinateSystem',...
                            'Coordinate System ''%s'' is not supported',...
                            CS);
                end
                switch(Units)
                    case 'Metric'
                    case 'English'
                        obj.PrivateUnits = Units;
                    otherwise
                        error('Vehicle:InvalidUnits',...
                            'Units ''%s'' is not supported', Units);
                end
                obj.Mass = mass;
                obj.FuelMass = fuelmass;
                obj.FuelMassDot = fuelmassdot;
                obj.Sv = sv;
                obj.Thrust = Thrust;
            else
                obj.Mass = 0;
                obj.FuelMass = 0;
                obj.FuelMassDot = 0;
                obj.Sv = [0 0 0 0 0 0 0 0 0];
                obj.Thrust = 0;
            end
        end
        %       /Ctor*****************************************************

        %       Get Functions*********************************************
        function units = get.Units(obj)
            units = obj.PrivateUnits;
        end

        function coordinatesystem = get.CoordinateSystem(obj)
            coordinatesystem = obj.PrivateCoordinateSystem;
        end
        %       /Get Functions********************************************

        %       Set Functions*********************************************
        function obj = set.Units(obj,newUnits)
            switch(newUnits)
                case 'Metric'
                    if ~strcmp(obj.PrivateUnits, newUnits)
                        obj.PrivateUnits = newUnits;
                        obj =convertPropertyUnits(obj,1./obj.ConversionFactors);
                    end
                case 'English'
                    if ~strcmp(obj.PrivateUnits, newUnits)
                        obj.PrivateUnits = newUnits;
                        obj = convertPropertyUnits(obj,obj.ConversionFactors);
                    end
                otherwise
                    error('Vehicle:InvalidUnits',...
                        'Units ''%s'' is not supported', newUnits);
            end
        end

        function obj = set.CoordinateSystem(obj,newCS)
            switch(newCS)
                case 'Cartesian'
                    if ~strcmp(obj.PrivateCoordinateSystem, newCS)
                        obj = convertPropertyCoordinates(obj,newCS);
                        obj.PrivateCoordinateSystem = newCS;
                    end
                case 'Spherical'
                    if ~strcmp(obj.PrivateCoordinateSystem, newCS)
                        obj = convertPropertyCoordinates(obj,newCS);
                        obj.PrivateCoordinateSystem = newCS;
                    end

                otherwise
                    error('Vehicle:InvalidCoordinateSystem',...
                        'Coordinate System ''%s'' is not supported',...
                        newCS);
            end
        end

        function obj = set.Mass(obj,mass)
            if mass<0
                error('Mass must be non-negative');
            else
                obj.Mass = mass;
            end
        end

        function obj = set.FuelMass(obj,mass)
            if mass<0
                error('Mass must be non-negative');
            else
                obj.FuelMass = mass;
            end
        end

        function obj = set.Sv(obj,sv)
            if ~isequal(size(sv),[1,9])
                error('Vehicle:InvalidValue',...
                    'State vector must be 1x9 vector');
            else
                obj.Sv = sv;
            end
        end

        function obj = set.Thrust(obj,T)
            if isnumeric(T)
                obj.Thrust = T;
            else
                error('Vehicle:InvalidValue',...
                    'Thrust value must be numeric');
            end
        end
        %       /Set Functions********************************************

        %       Additional Functions**************************************
```

## Vehicle Homework 1 Function Definition

```
        function struct = homework1_propagate(obj,tspan)

            % Define celestial body constants
            g0 = 9.81; %m/s^2
            R0 = 6400000; %m

           y0 = zeros(3,1);
           y0(1) = obj.Sv(3);
           y0(2) = obj.Sv(6);
           y0(3) = obj.Mass;

            sol = ode45(@(t,y) hw1_prop_eq(t,y,g0,R0,...
                obj.Thrust,obj.FuelMassDot),...
                tspan,y0);


            xint1 = tspan(1):1:600;
            xint2 = 660:60:tspan(2);
            xint = [xint1 xint2];
            [y,yp] = deval(sol, xint);
            struct.position = y(1,:);
            struct.velocity = y(2,:);
            struct.acceleration = yp(2,:);
            struct.mass = y(3,:);
            struct.time = xint/(24*3600);

            %define ode function to pass to ode45
            function dy = hw1_prop_eq(~,y,g0,R0,Thrust,fuelMassDot)
                dy = zeros(2,1);
                dy(1) = y(2);
                dy(2) = -g0*R0^2/y(1)^2 + Thrust/(y(3));
                dy(3) = fuelMassDot;
            end
        end
        %       /Additional Functions*************************************
    end %/methods

    %       Helper Functions******************************************
    methods(Access = private)
        function obj = convertPropertyUnits(obj,cf)
            for prop=obj.UnitConversionIndex'
                if ~strcmp(prop{1}, 'Sv')
                    obj.(prop{1}) = obj.(prop{1})*cf(prop{2});
                else
                    if strcmp(obj.PrivateCoordinateSystem, 'Cartesian')
                        obj.(prop{1}) = obj.(prop{1})*cf(prop{2});
                    elseif strcmp(obj.PrivateCoordinateSystem, 'Spherical')
```

```
                        sv = obj.Sv;
                        obj.Sv(3) = sv(3)*cf(prop{2});
                        obj.Sv(6) = sv(6)*cf(prop{2});
                        obj.Sv(9) = sv(9)*cf(prop{2});
                    end
                end
            end
        end

        function obj = convertPropertyCoordinates(obj,newCS)
            sv = obj.Sv;
            switch(obj.PrivateCoordinateSystem)
                case 'Cartesian'
                    if strcmp(newCS, 'Spherical')
                        [az,el,r] = cart2sph(sv(1),sv(2),sv(3));
                        [vaz,vel,vr] = cart2sph(sv(4),sv(5),sv(6));
                        [aaz, ael, ar] = cart2sph(sv(7),sv(8),sv(9));
                        obj.Sv = [az,el,r,vaz,vel,vr,aaz,ael,ar];
                    end
                case 'Spherical'
                    if strcmp(newCS, 'Cartesian')
                        [x,y,z] = sph2cart(sv(1),sv(2),sv(3));
                        [vx,vy,vz] = sph2cart(sv(4),sv(5),sv(6));
                        [ax,ay,az] = sph2cart(sv(7),sv(8),sv(9));
                        obj.Sv = [x,y,z,vx,vy,vz,ax,ay,az];
                    end
            end
        end
    end

end
```