Assignment 3 - Report

Rajdeep Pinge

ID - 201401103

Q1. Sliding Block Problem: Investigate (computationally) the motion of a block sliding without friction down a fixed inclined plane with different initial parameters. Derive the analytical solution for displacement, velocity, and acceleration. Compare the computational results with analytical solutions for the case when the angle of the inclined plane is 30 degree - to check the accuracy of the computational model. (Example 2.1; Marion and Thornton).

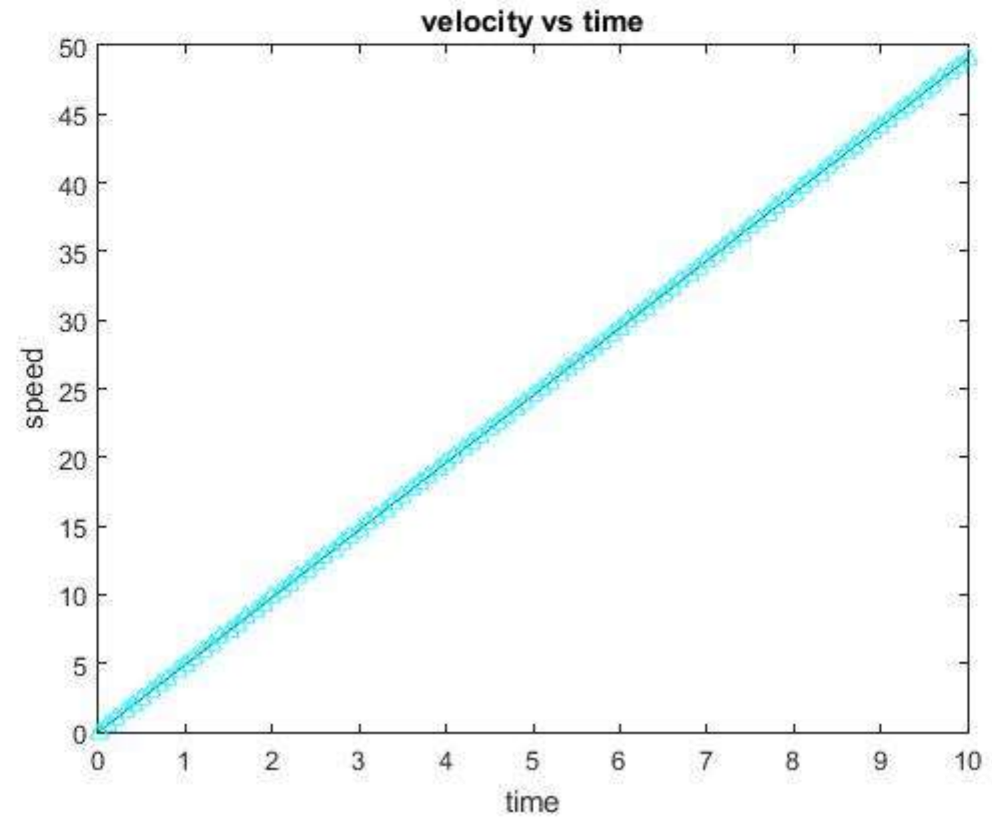
MATLAB Code

```
clear;
close all;

%defining initial values
g = 9.8;
theta_degree = 30;
theta = (pi/180) * theta_degree;    %in radians
init_vel = 0;
init_pos = 0;

global const
const = g * sin(theta);

total_time = 10;
dt = 0.1;
time = 0:dt:total_time;



% Analytical solution
acceleration = g * sin(theta);
velocity = g * sin(theta) * time + init_vel;
position = g/2 .* sin(theta) .* time .* time + init_vel .* time + init_pos;



%Computational solution
tstart = 0;
tfinal = total_time;

% set the initial conditions in the u_init column vector
u_init = zeros(2,1);
u_init(1) = init_pos; % initial position
u_init(2) = init_vel; % initial velocity

% solve using ode45
[t, u]=ode45(@q1_sliding_motion, [tstart:dt:tfinal], u_init);

% store the answer
x_pos = u(:, 1);
v_vel = u(:, 2);
```

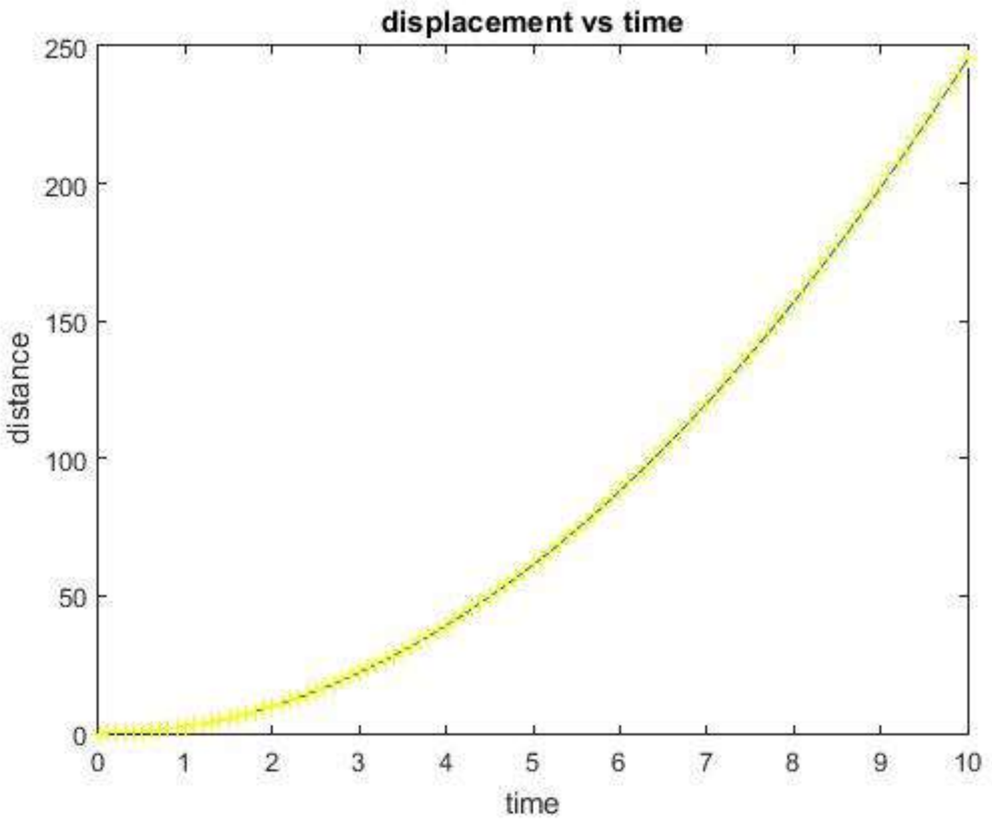
```
%plotting the graphs
plot(time, velocity)
hold on;
plot(time, v_vel,'c^')
title('velocity vs time')
xlabel('time')
ylabel('speed')

figure
plot(time, position)
hold on;
plot(time, x_pos,'y+')
title('displacement vs time')
xlabel('time')
ylabel('distance')

plot(time, position,'y+')
```

Function :

```
function F = q1_sliding_motion(t, u);

% In our case we will use:
% u(1) -> x
% u(2) -> v

% declare the globals so its value
% set in the main script can be used here
global const;

% make a zero column vector F of size of u
F = zeros(length(u), 1);

% dx/dt=v means that F(1)=u(2)
F(1) = u(2);
% dv/dt=-g * sin(theta)
F(2) = const;
```



The blue line gives the computational result while the black line gives the analytical solution. Both lines closely match indicating the correctness of the method.



Similarly yellow line gives the computational result while the black line gives the analytical result. Accuracy of the method is verified by observing the closeness of the graph.

Changing initial values:

Changing initial position does not make much of a difference because it does not affect the equation. The nature of graph remains same.

Changing initial velocity keeps the nature of velocity time graph same but changes the slope of the displacement time graph since displacement is dependent on velocity. Hence as the initial velocity increases, the distance covered by the block increases.

Changing value of theta changes both the graphs. Although the velocity-time graph remains a straight line, its slope changes. As theta increases the slope increases indicating that the particle achieves higher velocity as the inclination increases. This implies that the distance covered by the particle increases.

Q2. Introduce the effect of static friction and kinetic friction into the previous problem. Take coefficient of static friction=.4 and coefficient of kinetic friction=.3 and computationally analyze the motion for different initial angles. Report your computational observations and how the results compare with theoretical solutions (Example 2.2-2.3; Marion and Thornton).

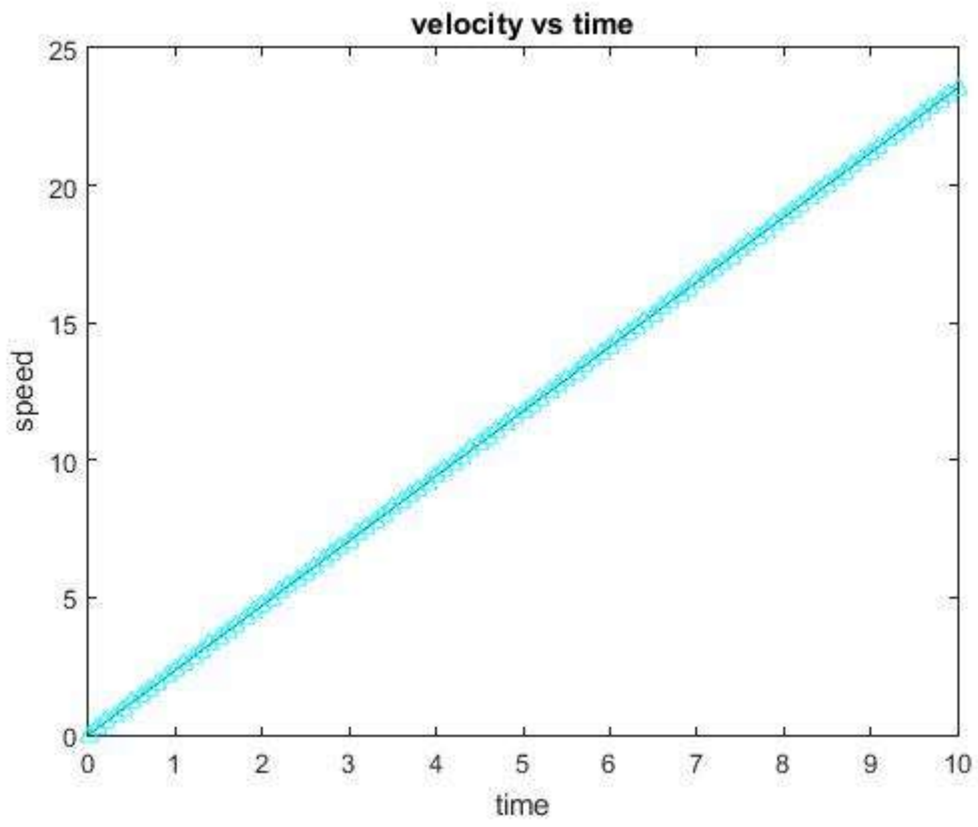
MATLAB Code

```
clear;
close all;

%defining initial values

global mu_s
g = 9.8;
mu_k=0.3;
mu_s=0.4;

global theta
theta_degree = 30;
theta = (pi/180) * theta_degree;     %in radians

global const;
const = g * (sin(theta)-(mu_k*cos(theta)));

init_vel = 10;
init_pos = 0;

total_time = 20;
dt = 0.1;
time = 0:dt:total_time;

npoints = total_time/dt + 1;



% Analytical solution
acceleration = g * (sin(theta)-(mu_k*cos(theta)));

velocity = zeros(npoints, 1);
position = zeros(npoints, 1);

velocity(1) = init_vel;
position(1) = init_pos;

for step = 2 : npoints
    velocity(step) = acceleration * dt + velocity(step-1);
    position(step) = acceleration * time(step-1) * dt + init_vel*dt +
position(step-1);
    time(step) = time(step-1) + dt;

    % the mass will start to slide only when theta > atan(mu_s) deg. Hence
the
    % following is valid only for such angles. Hence, check this condition.
    % if angle is less an the mass is initially moving then it would
eventually
    % come to rest
    if(velocity(step) <= 0)
        velocity(step) = 0;
        position(step) = position(step-1);
    end
end

%Computational solution
tstart = 0;
tfinal = total_time;

% set the initial conditions in the u_init column vector
u_init = zeros(2,1);
u_init(1) = init_pos; % initial position
u_init(2) = init_vel; % initial velocity

[t, u]=ode45(@q2_sliding_motion, [tstart:dt:tfinal], u_init);

x_pos = u(:, 1);
v_vel = u(:, 2);



%plotting the graphs
plot(time, velocity)
hold on;
plot(time, v_vel,'c^')
title('velocity vs time')
xlabel('time')
ylabel('speed')

figure
plot(time, position)
hold on;
plot(time, x_pos,'y+')
title('displacement vs time')
xlabel('time')
ylabel('distance')
```

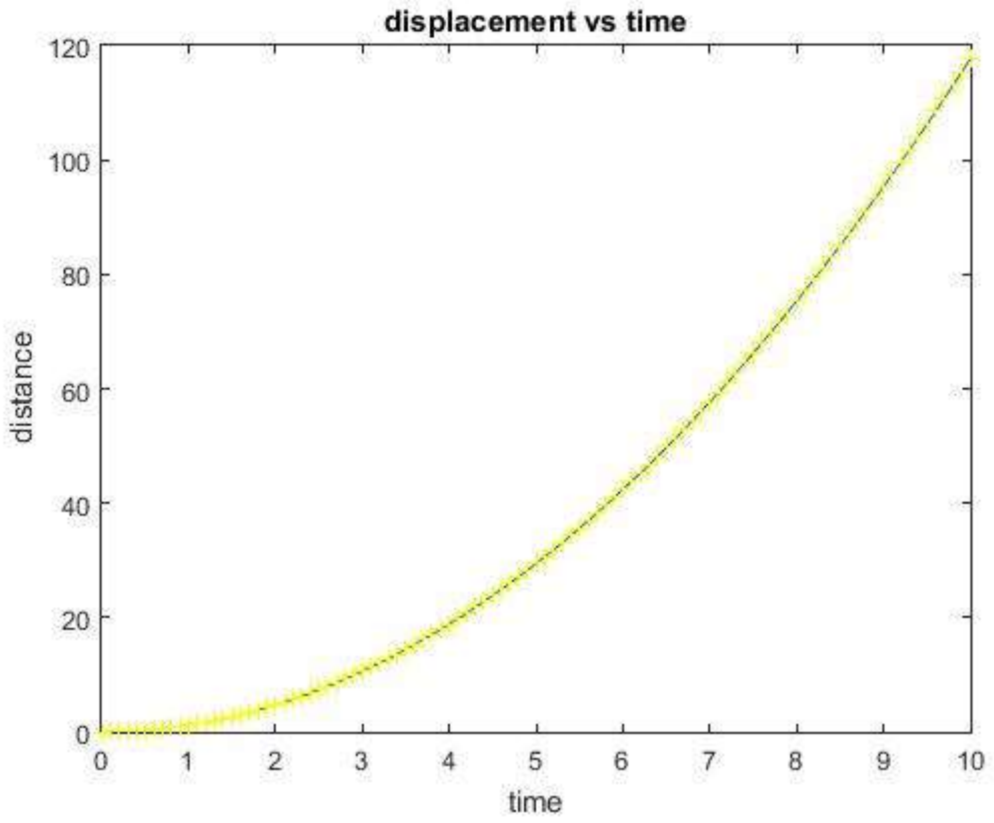
Function :

```
function F = q2_sliding_motion(t, u);

% In our case we will use:
% u(1) -> x
% u(2) -> v

% declare the globals so its value
% set in the main script can be used here
global const
global mu_s
global theta

% make a zero column vector F of size of u
F = zeros(length(u), 1);

% dx/dt=v
F(1) = u(2);
```

```
% the mass will start to slide only when theta > atan(mu_s) deg. Hence the
% motion is valid only for such angles. Hence, check this condition.
% if angle is less an the mass is initially moving then it would eventually
% come to rest. At that point stop the calculation
if(u(2) <= 0 && theta < atan(mu_s))
    return;
end

F(2) = const;
```



The blue line gives the computational result while the black line gives the analytical solution. Both lines closely match indicating the correctness of the method.



Similarly yellow line gives the computational result while the red line gives the analytical result. Accuracy of the method is verified by observing the closeness of the graph.

Changing initial values:

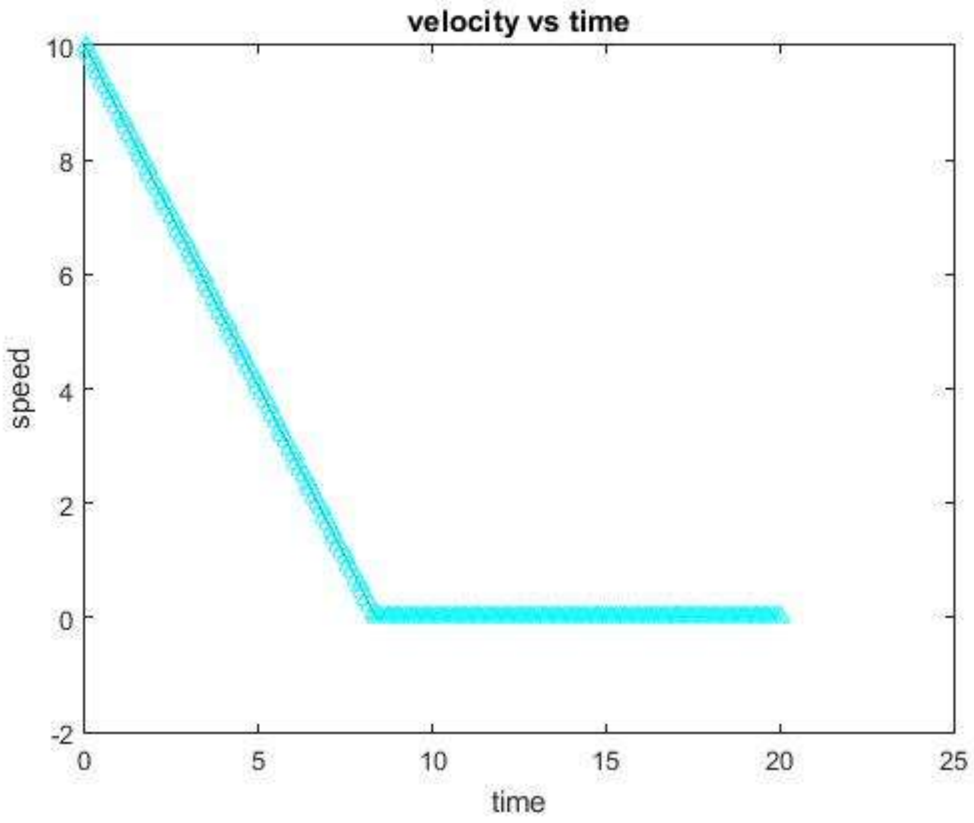
Changing initial position does not make much of a difference because it does not affect the equation. The nature of graph remains same.

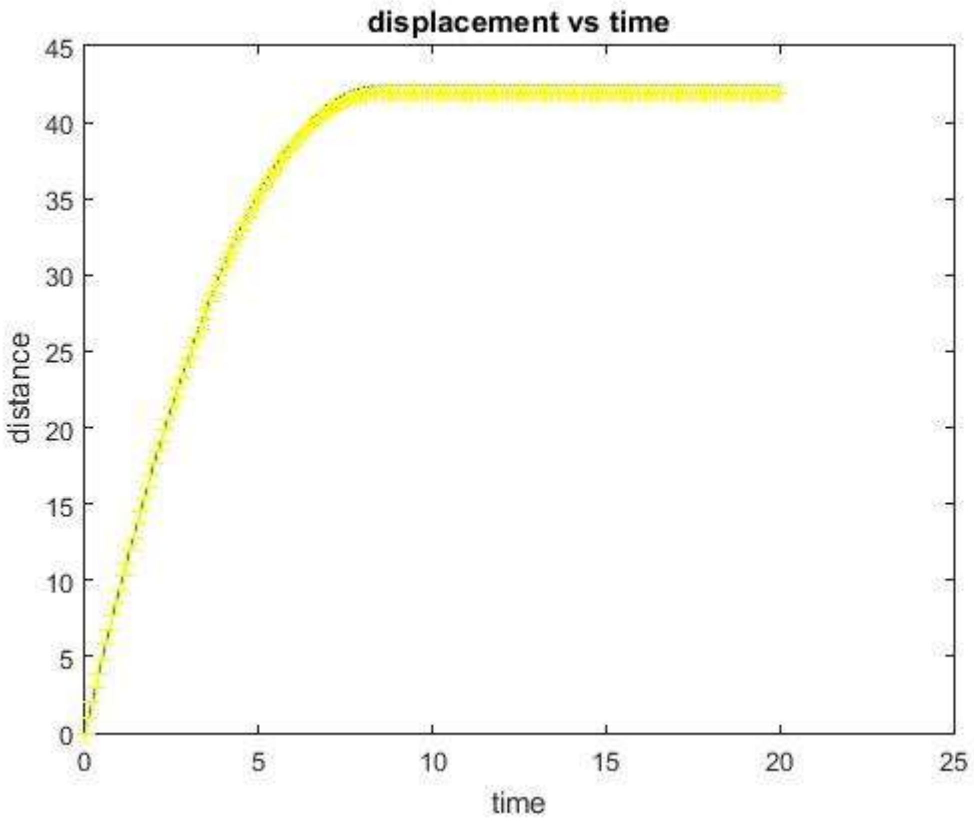
Changing initial velocity keeps the nature of velocity time graph same but changes the slope of the displacement time graph since displacement is dependent on velocity. Hence as the velocity increases, the distance covered by the block increases.

Changing value of theta changes both the graphs.

Assuming initial velocity is 0, motion is valid only if the static friction is overcome by the block. i.e. m*g*sin(theta) >= mu_s*m*g*cos(theta) it gives theta >= arc_tan(mu_s). In this case theta >= 21.8 degrees. This means that block will only move if the inclination of the plane is greater than 21.8 degrees.

If theta >= 21.8 degrees, block will always move.
If theta < 21.8 degrees,
    If initial velocity is 0, block will remain at rest.
    If initial velocity is not 0, block will initially move but will eventually come to rest.
As shown in the following graph.


velocity vs time
speed
time
10
8
6
4
2
0
-2
0
5
10
15
20
25


displacement vs time
distance
time
45
40
35
30
25
20
15
10
5
0
0
5
10
15
20
25

Q3.

A

Case 1: Gravitational force g remains constant

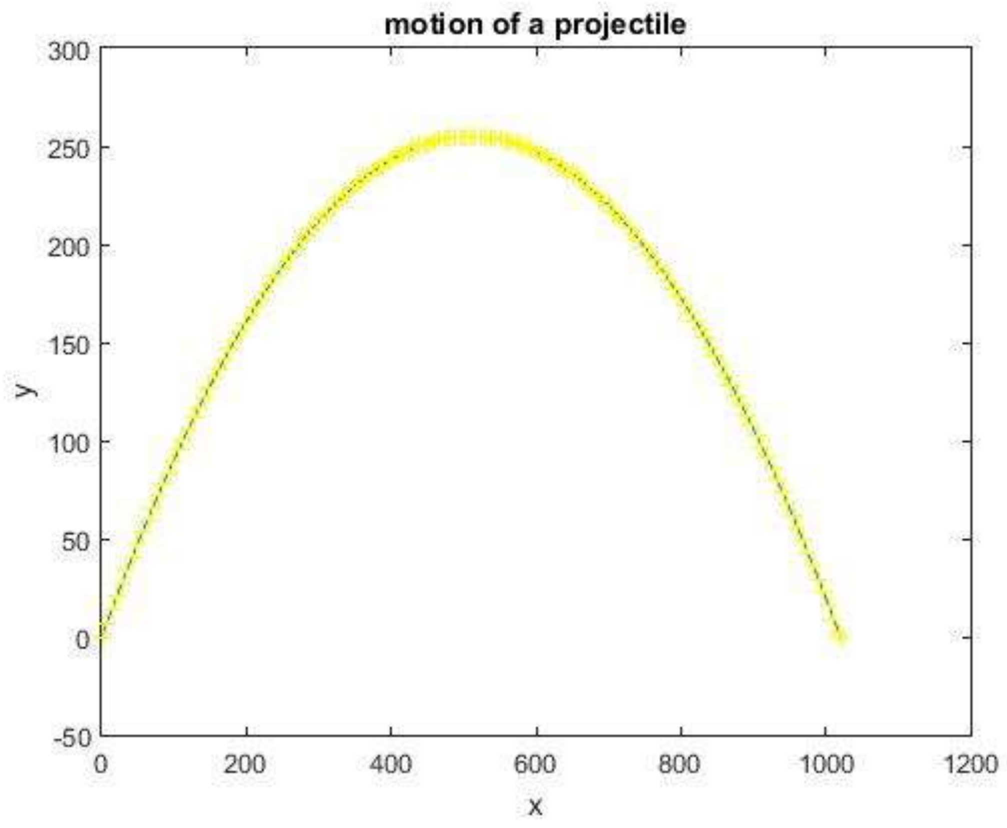
MATLAB Code

```
clear
close all;

% declaring constants and initial values
total_time=100;
dt=0.1;
npoints = total_time/dt;

global constx, g;
constx = 0;
g = 9.8;

init_pos_x = 0;
init_pos_y = 0;

theta_degree = 45;
theta = (pi/180) * theta_degree;    %in radians
init_vel = 100;

vx = init_vel * cos(theta);
vy = init_vel * sin(theta);



% Exact Solution
time_of_flight = 2*init_vel*sin(theta) / g;
time = 0:dt:time_of_flight;
x = vx * time;
y = vy * time - g/2 * time .* time;



%Computational solution
tstart = 0;
tfinal = total_time;

% set the initial conditions in the u_init column vector
u_init = zeros(4,1);
u_init(1) = init_pos_x; % initial position x-dir
u_init(2) = init_pos_y; % initial position y-dir
u_init(3) = vx; % initial velocity x-dir
u_init(4) = vy; % initial velocity y-dir

% using ODE-solver to solve the ODE
[t, u]=ode45(@q3_projectile_ideal_without_g_var, [tstart:dt:tfinal],
u_init);

x_pos = u(:, 1);
y_pos = u(:, 2);
vx_vel = u(:, 3);
```

```
vy_vel = u(:, 4);



% plotting graph
plot(x,y)
title('motion of a projectile')
xlabel('x')
ylabel('y')

hold on;
plot(x_pos, y_pos, 'y+')
title('motion of a projectile')
xlabel('x')
ylabel('y')
```

Function :

```
function F = q3_projectile_ideal_without_g_var(t, u);

% In our case we will use:
% u(1) -> x
% u(2) -> y
% u(3) -> vx
% u(4) -> vy

% declare the globals so its value
% set in the main script can be used here
global constx, g;

% make the column vector F with length equal to u
F = zeros(length(u), 1);

%if the height from ground becomes <= 0 the motion must end
if u(2) < 0
    return;
end

% Now build the elements of F
% dx/dt=vx and dy/dt = vy
F(1) = u(3);
F(2) = u(4);

% finding dvx/dt and dvy/dt
F(3) = constx;
F(4) = -g;
```



Note: Yellow trajectory is the computational result while the black trajectory is the exact solution. Both the exact solution and the computational result closely match each other.

Case 2: Gravitational force g’ changes with altitude

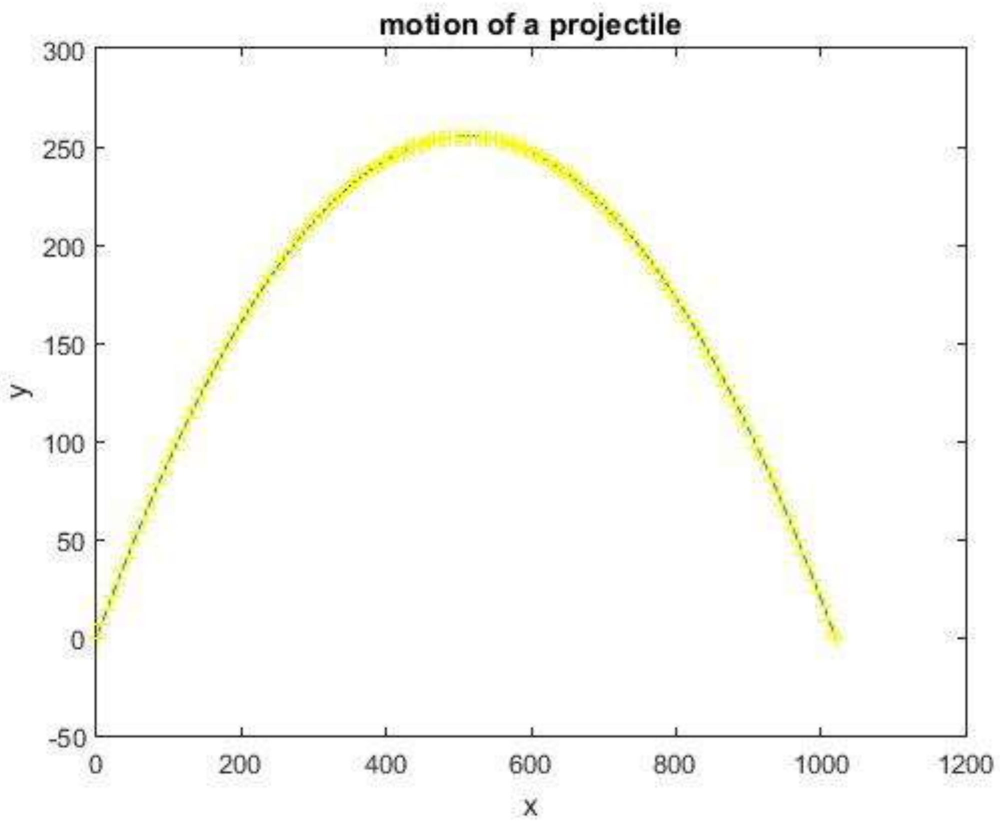
Changes in the above code

In main code

```
global R
R = 6.4e5;  % radius of earth in meters

% Exact Solution
time_of_flight = 2*init_vel*sin(theta) / g;
time = 0;
npoints = ceil(time_of_flight/dt);

x = zeros(npoints,1);
y = zeros(npoints,1);
time = zeros(npoints,1);

x(1) = 0;
y(1) = 0;
time(1) = 0;

for step = 2 : npoints
    time(step) = time(step-1) + dt;
    x(step) = vx * time(step);
    y(step) = vy * time(step) - 1/2 * g * ( R / (R+y(step-1)) )^2 *
time(step) * time(step);
end
```

In function:

```
F(4) = -g * ( (R/(u(2)+R)) ^ 2 );
```



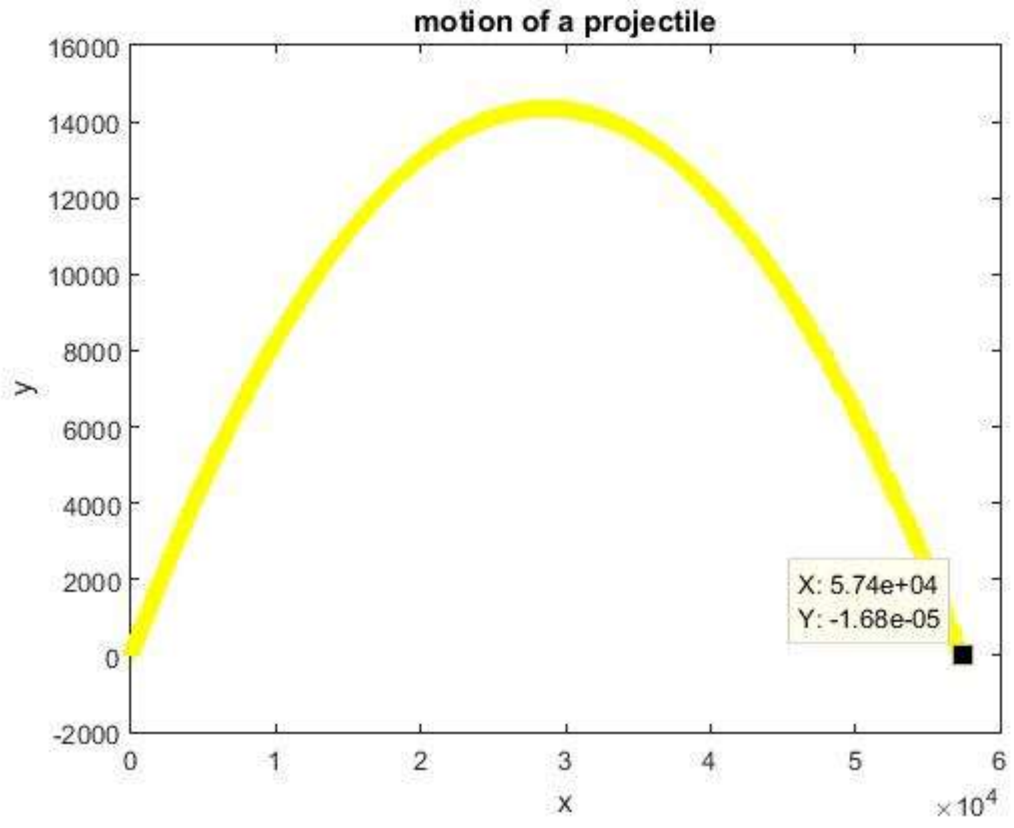
Note: Yellow trajectory is the computational result while the black trajectory is the exact solution. Both the exact solution and the computational result closely match each other.

Analysis of the above two cases: The above to cases seem similar but there is minute difference due to the changes in gravitational force. The projectile in case 2 achieves higher altitude due to less gravity as it goes higher and higher.

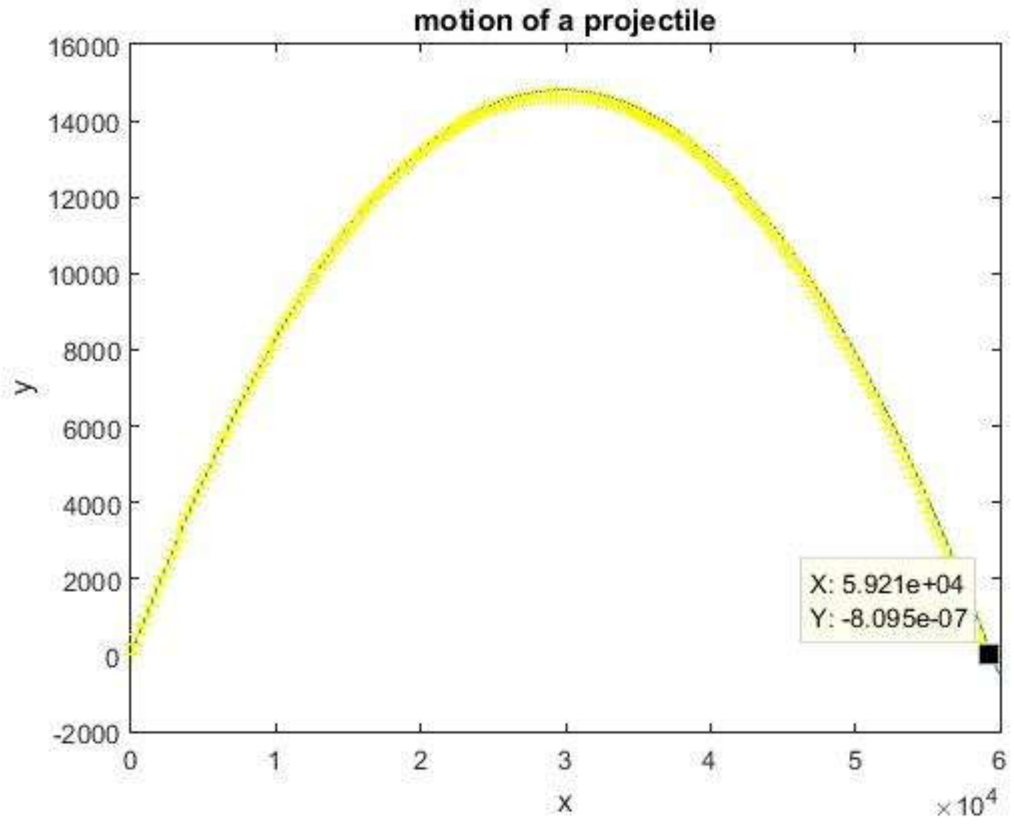
The difference becomes significant when the height goes into the kilometre range. Here it is nearly 250m, which is very less to see the effect of gravity.

The difference can be seen in the graphs below for initial velocity = 750m/s.

Graph when g is constant



Graph when g is varying



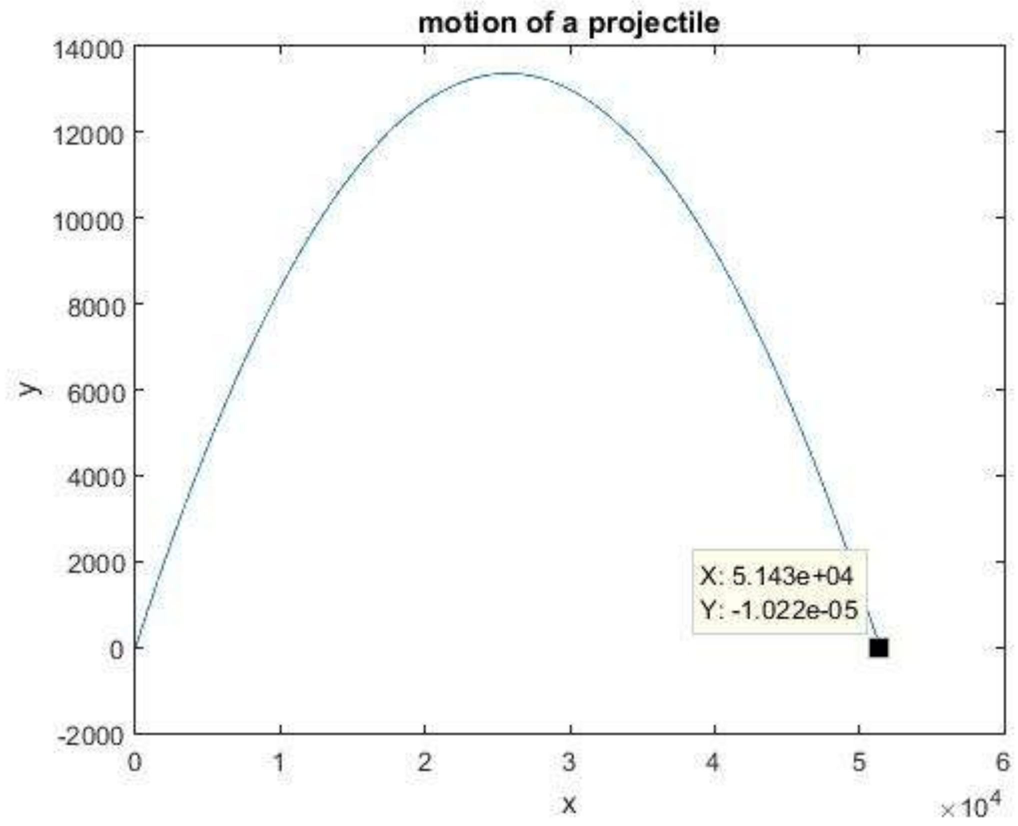
The difference between the value of x component is visible.

B part : Air Drag

Changes in the above code

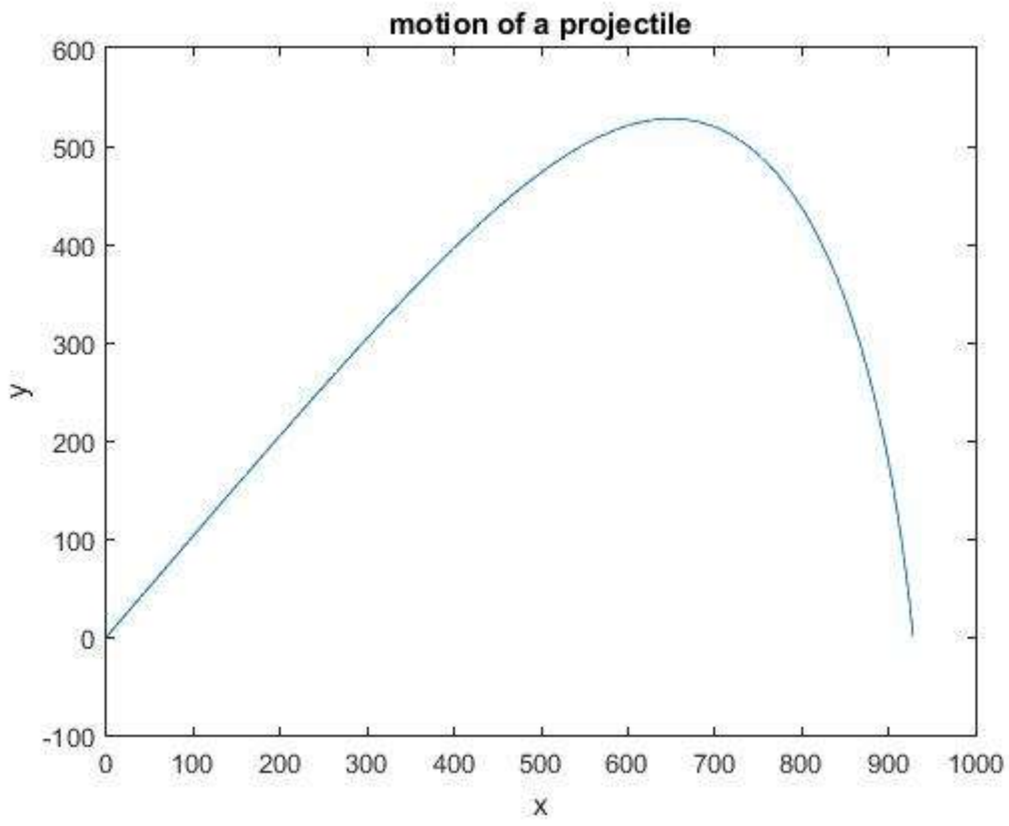
In main code:
```
global mass, B, y0;
mass = 1; %kg
B = 4e-5 * mass;
y0 = 1000;
init_vel = 750;
```

In Function:
```
F(3)  = exp(-u(2)/y0) * B/mass * u(3) * sqrt(u(3)^2+u(4)^2);
F(4)  = -g - exp(-u(2)/y0) * B / mass * u(4) * sqrt(u(3)^2 + u(4)^2);
```



Note: Here the effect of air drag cannot be seen because the magnitude of drag force is very low due to B = 4e-5 and due to the height which goes up to nearly 13km from ground level resulting in air density becoming negligible.
But still if we compare the range with the range of previous graph, we can see the effect of drag.

If we increase B to 4e-3 or neglect the term exp(-y/y0), we can clearly see the effect of the force.



Analysis: Due to drag force the velocity in x-direction constantly decreases hence the projectile covers lesser and lesser distance in x-direction. This causes the above nature of graph.

Finding angle at which the range is maximum

Angle	Range
10	1.482e4
20	2.928e4
30	4.243e4
40	5.002e4
50	5.102e4
60	4.54e4
45	5.136e4
43	5.103e4
47	5.143e4
46	5.143e4
48	5.136e4

Therefore an angle of 46 or 47 degrees gives maximum range. We know that for ideal case, the angle of 45 degrees gives maximum range but here due to the effect of air drag, the angle for maximum range changes.

C: Finding minimum velocity for different destination heights.

MATLAB Code:

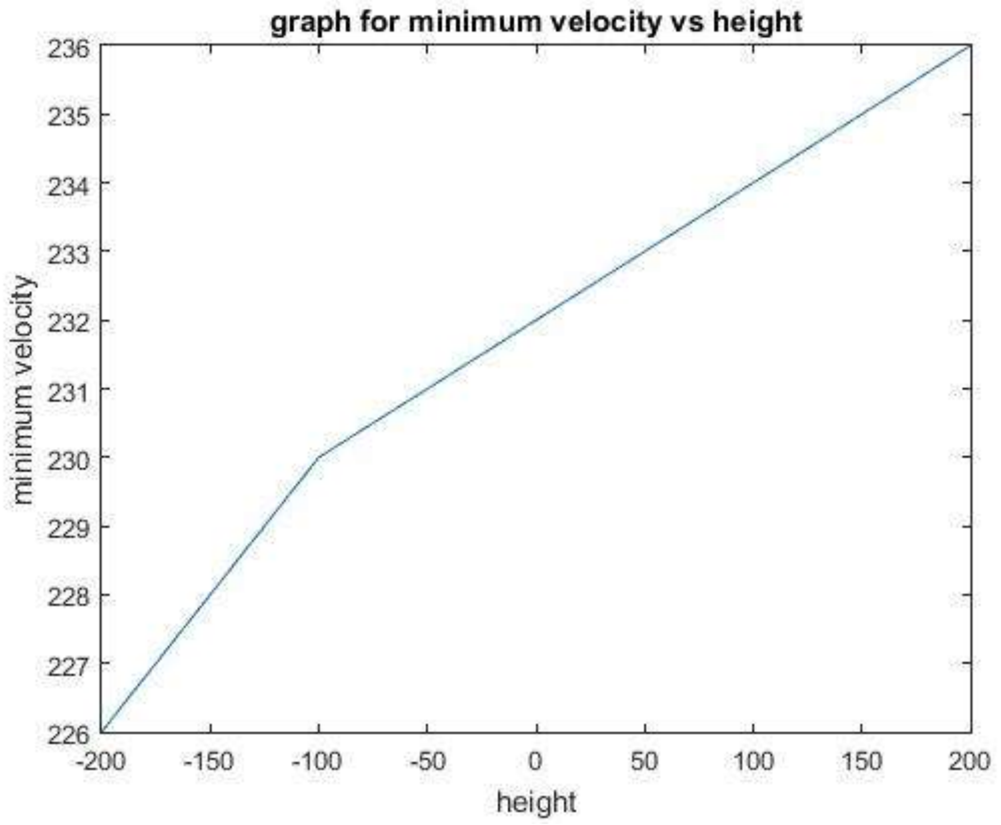
```
clear
close all;

% declaring constants and initial values
total_time=100;
dt=0.1;
npoints = total_time/dt;

global constx g mass B y0;
constx = 0;
g = 9.8;
mass = 1; %kg
B = 4e-5 * mass;
y0 = 1000;

range_x = 5e3;      % 5km
delta = 10;         % precision of 10 meters

flag = 0;

min_height = -2000;
max_height = 2000;
dh = 100;
height = [min_height : dh : max_height];

init_vel = 100;
max_vel = 600;
dv = 2;
min_vel = max_vel;
vel_arr = zeros( (max_height-min_height)/dh + 1, 1);

vx = zeros(npoints,1);
vy = zeros(npoints,1);

x = zeros(npoints,1);
y = zeros(npoints,1);
time = zeros(npoints,1);

init_theta = 0;
max_theta = 90;
d_theta = 1;

%loop
for height_y = min_height : dh : max_height
    for v = init_vel : dv : max_vel
        min_vel = max_vel;

        for theta_degree = init_theta : d_theta : max_theta
            theta = (pi/180) * theta_degree;    %in radians

            vx(1) = v * cos(theta);
            vy(1) = v * sin(theta);

            x(1) = 0;
            y(1) = 0;
            time(1) = 0;

            for step = 2 : npoints
                time(step) = time(step-1) + dt;
                x(step) = x(step-1) + vx(step-1) * dt;
                vx(step) = vx(step-1) - exp(-y(step-1)/y0) * (B / mass) *
vx(step-1) * sqrt(vx(step-1)^2 + vy(step-1)^2) * dt;
                y(step) = y(step-1) + vy(step-1) * dt;
                vy(step) = vy(step-1) - g * dt - exp(-y(step-1)/y0) * (B /
mass) * vy(step-1) * sqrt(vx(step-1)^2 + vy(step-1)^2) * dt;

                if( ((range_x - x(step))^2 + (height_y - y(step))^2) <
delta^2 )
                    if( v < min_vel )
                        min_vel = v;
                        flag = 1;
                    end
                    break;
                end
            end

            if(flag == 1)
                break;
            end
        end

        if(flag == 1)
            flag = 0;
            break;
        end
    end

    vel_arr( (height_y/dh) + 1 - (min_height/dh) ) = min_vel;
end

% plotting graph
plot(height, vel_arr)
title('graph for minimum velocity vs height')
xlabel('height')
ylabel('minimum velocity')
```

Here we have taken range to be 5km. But this must increase to effectively simulate a cannon.

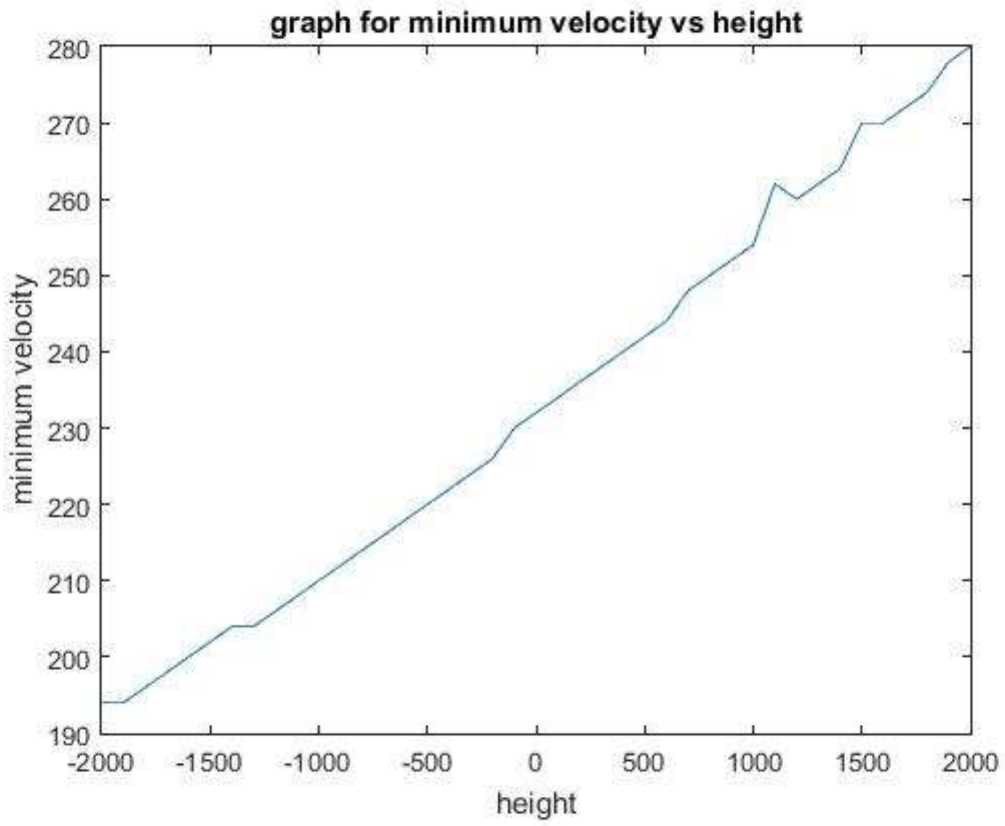
The code takes too much time to run. As the precision increases, the time to run increases.

Hence we have restricted the range to 5km, delta i.e the acceptable distance from the desired end point for which we can approximate the end of motion to be 10 meters. The minimum and maximum height have also been restricted in the range 2km below the level of projection to 2km above the level of projection.

The following graphs show some of the test cases.



This case took 399 seconds, that is approx. 6.5 mins to complete.



This graph for height from -2km to 2km, took 1280 seconds i.e. approx. 21.3 mins to complete.

The third try with even more precision didn’t complete even after 2 hours.

From the above two cases it can be inferred that the nature of the graph is linear.

The accuracy in linearity can be obtained if we simulate the code with extreme precision and on an extremely fast machine like a super-computer.

The huge amount of time taken for such a basic or preliminary problem shows the need for today’s hot topics like High Performance Computing and Optimization.

D: Effect of wind

MATLAB Code:

```
clear
close all;

% declaring constants and initial values
total_time=500;
dt=0.1;
npoints = total_time/dt;

global constx;
constx = 0;

global g;
g = 9.8;

global mass;
mass = 1; %kg

global B;
B = 4e-5 * mass;

global y0;
y0 = 1000;
```

```
init_pos_x = 0;
init_pos_y = 0;

theta_degree = 48;
theta = (pi/180) * theta_degree;     %in radians

wind_vel_x = -50;  %m/s
wind_vel_y = 0;
init_vel = 750;
vx = init_vel * cos(theta);
vy = init_vel * sin(theta);




%Computational solution
tstart = 0;
tfinal = total_time;

% set the initial conditions in the u_init column vector
u_init = zeros(7,1);
u_init(1) = init_pos_x; % initial position x-dir
u_init(2) = init_pos_y; % initial position y-dir
u_init(3) = vx; % initial velocity x-dir
u_init(4) = vy; % initial velocity y-dir
u_init(5) = init_vel;
u_init(6) = wind_vel_x;
u_init(7) = wind_vel_y;

% using ODE-solver to solve the ODE
[t, u]=ode45(@q3_b_ode_air_drag_density, [tstart:dt:tfinal], u_init);

x_pos = u(:, 1);
y_pos = u(:, 2);
vx_vel = u(:, 3);
vy_vel = u(:, 4);




% plotting graph
plot(x_pos, y_pos)
title('motion of a projectile')
xlabel('x')
ylabel('y')
```

Function:

```
function F = q3_b_ode_air_drag_density(t, u);

% In our case we will use:
% u(1) -> x
% u(2) -> y
% u(3) -> vx
% u(4) -> vy
% u(5) -> init_vel;
% u(6) -> wind_vel_x;
% u(7) -> wind_vel_y;
% declare the globals so its value
% set in the main script can be used here
global constx;
global g;
global B;
global y0;
global mass;

% make the column vector F with length equal to u
F = zeros(length(u), 1);

%if the height from ground becomes <= 0 the motion must end
if u(2) < 0
    return;
end

% Now build the elements of F
% dx/dt=vx and dy/dt = vy
F(1) = u(3);
F(2) = u(4);

% finding dvx/dt and dvy/dt
F(3) = constx - exp(-u(2)/y0) * (B / mass) * (u(3) / sqrt(u(3)^2 + u(4)^2))
* ((u(3)+u(6))^2 + (u(4)+u(7))^2); % equation of acceleration in presence
of wind
F(4) = -g - exp(-u(2)/y0) * (B / mass) * (u(4) / sqrt(u(3)^2 + u(4)^2)) *
((u(3)+u(6))^2 + (u(4)+u(7))^2);
```

Considering wind is constant at all the altitudes and since it is wind, it has constant velocity.

Comparing this with the graph in section B, we come to know that the nature of the projectile motion remains same but the total distance covered by the projectile decreases because wind is opposing the motion. If the wind is supporting the motion, the range will increase.

Similarly in y-direction, if the wind is supporting the motion, the range will increase, otherwise if it is opposing, the range will decrease.

Q4. Simple Harmonic Motion: Computationally investigate the motion of a pendulum and a springmass system as discussed in the class for damped, driven system. Draw phase plots to explain your observations. Estimate the time constant of decay for a damped system and compare the results with analytical solution.

MODIFIED: Do only for normal cases without any damping.

Motion of a Pendulum

Assumption: The motion is linear and the theta is very small.

MATLAB Code:

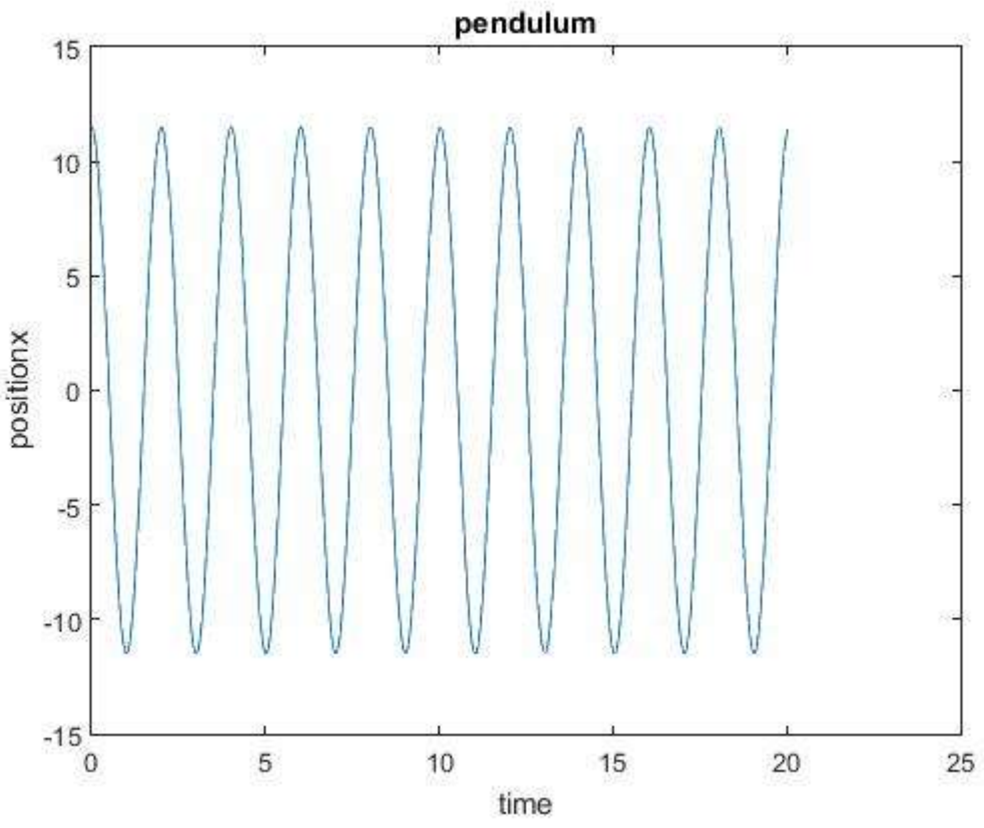
```
clear;
close all;

% declaring initial values and constants
global const;
global beta;
global b;
g=9.8;
l=1;
const=g/l;
beta=0;
b=2*l*beta;

timescale=2*pi*sqrt(l/g);
dt=timescale/100;

% set the initial and final times
tstart=0;
tfinal=10*timescale;

% set the initial conditions
u_init=zeros(2,1);
u_init(1)=.2; % initial position theta radians
u_init(2)=0; % initial velocity

% calling ode solver
[t,u]=ode45(@q4_pendulumodefunction,[tstart:dt:tfinal],u_init);

% store the solution that comes back into x and v arrays
x = pi/180 * u(:,1); % radian-->degree
v = u(:,2);



% plot graphs
plot(t,x)
title('pendulum');
xlabel('time');
ylabel('positionx');

figure
plot(x,v)
title('pendulum phase space');
```

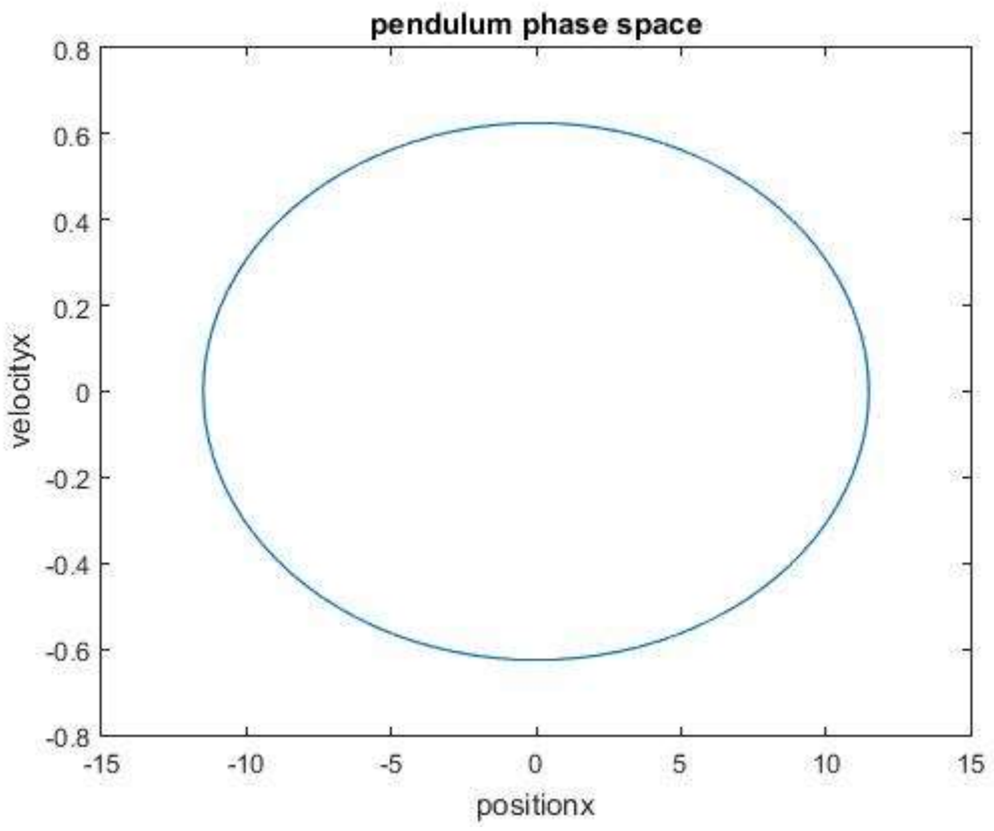
```
xlabel('positionx');
ylabel('velocityx');
```

Function:

```
function F=q4_pendulumodefunction(t,u)
% function output =name(input)
% right-hand side function for Matlab's ODE solver,

% In our case we will use:
% u(1) -> x
% u(2) -> v

% declare the globals so its value
% set in the main script can be used here
global const;
global beta;
global b;

% make the column vector F equal to length of u
F=zeros(length(u),1);

% dx/dt=v
F(1)=u(2);

% dv/dt
F(2)=-const*u(1)-b*u(2);
```

Normal motion without any damping





Note: Direction of this phase space diagram is clockwise.

Spring Mass System

MATLAB Code:

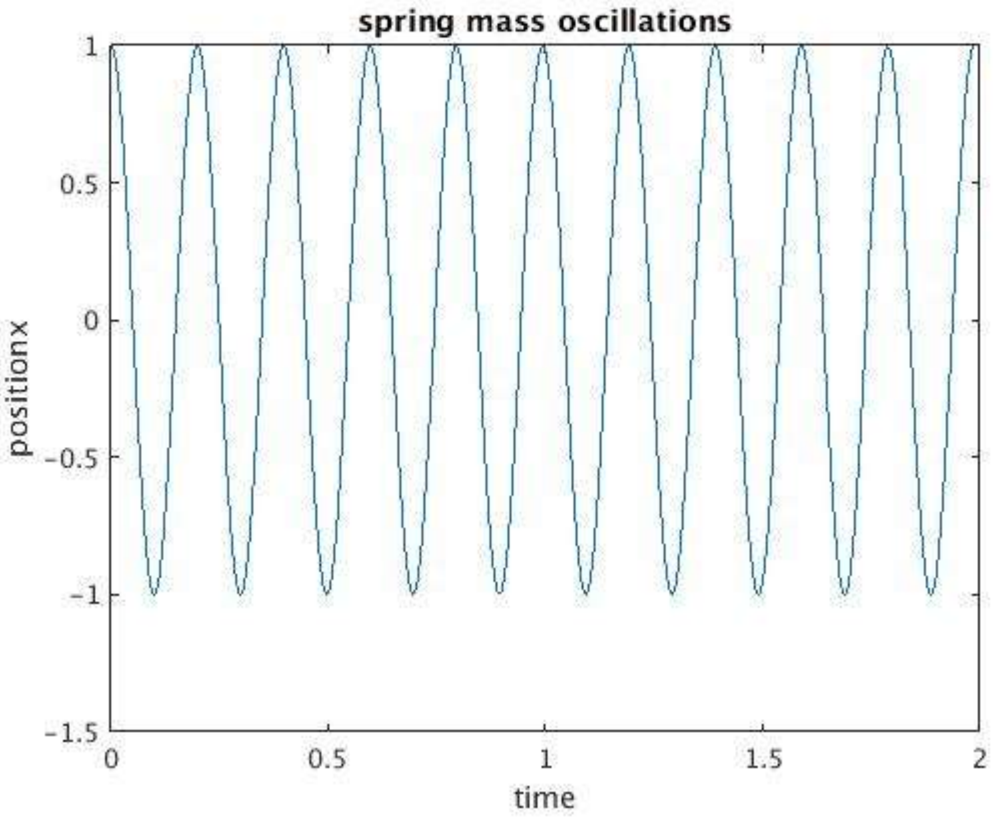
```
clear;
close all;

% declaring initial values and constants
global const;
global beta;
global b;
g=9.8;
k=1000;
m=1;

const=k/m;
beta=0;
b=2*m*beta;

timescale= 2*pi*sqrt(m/k);
dt=timescale/100;
```

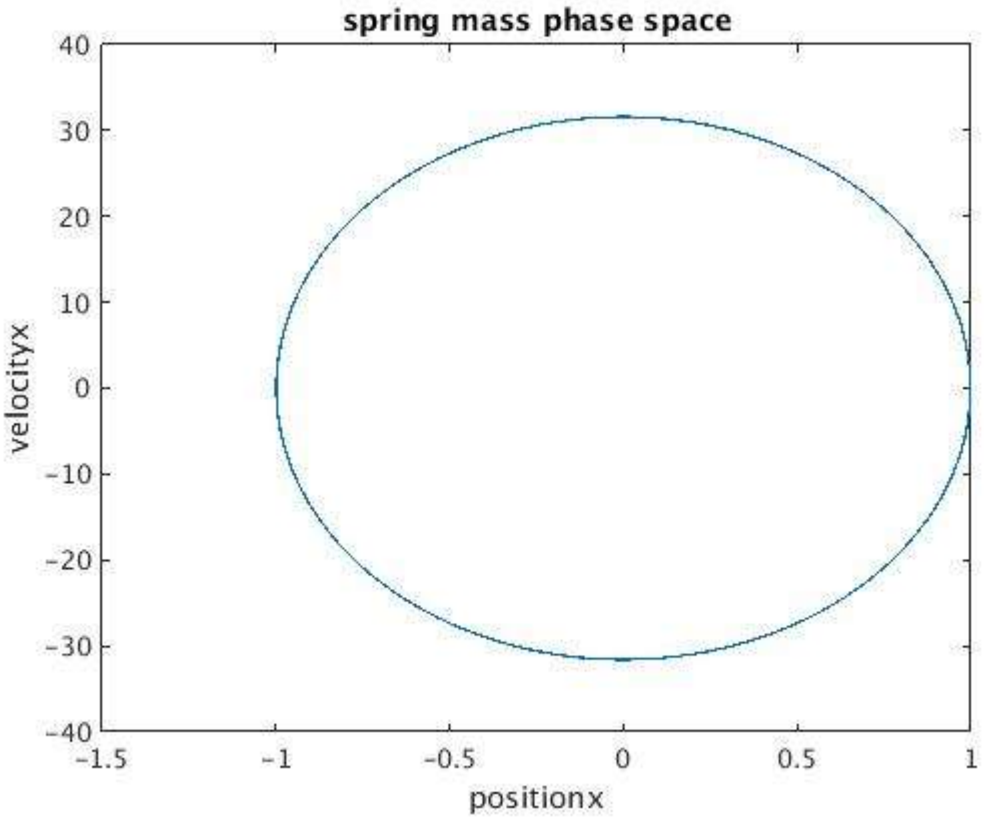
```
% set the initial and final times
tstart=0;
tfinal=10*timescale;

% set the initial conditions in the y0 column vector
u_init=zeros(2,1);
u_init(1)=.2; % initial position theta radians
u_init(2)=0; % initial velocity

% calling ode solver
[t,u]=ode45(@q4_ode_springmass,[tstart:dt:tfinal],u_init);

% store the solution that comes back into x and v arrays
x = 180/pi * u(:,1); % radian-->degree
v = u(:,2);



% plot the position vs. time
plot(t,x)
title('spring mass oscillations');
xlabel('time');
ylabel('positionx');

% make a "phase-space" plot of v vs. x
figure
plot(x,v)
title('spring mass phase space');
xlabel('positionx');
ylabel('velocityx');
```

Function:

```
function F=q4_ode_springmass(t,u)
% function output =name(input)
% right-hand side function for Matlab's ODE solver,

% In our case we will use:
% u(1) -> x
% u(2) -> v

% declare the globals so its value
% set in the main script can be used here
global const;
global beta;
global b;

% make the column vector F equal to length of u
F=zeros(length(u),1);

% dx/dt=v
F(1)=u(2);

% dv/dt
F(2)=-const*u(1)-b*u(2);
```

Normal motion without any damping





Note: Direction of this phase space diagram is clockwise.