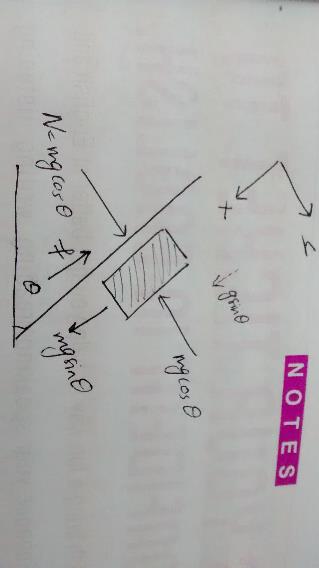
Name : Omkar Damle

ID : 201401114

CS 201 - ASSIGNMENT 3

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).

This is a very simple problem which can be solved analytically too.



Here friction is absent. Hence f = 0.

On solving we get,

Acceleration a =g\*sin(theta)

Velocity v = g\*sin(theta)\*t + initial\_vel

displacement x = g\*sin(theta)\*t\*t/2 + initial\_vel\*t + initial\_pos

Computational results :

Note : Bold green is computational graph and black is exact graph

1. For theta = 30 degrees , initial\_pos = initial\_vel = 0





The result is as expected and the analytical and computational solutions match.

1. Theta = 30 degrees , initial\_pos = 10m , initial\_vel = -20 m/s







Here the block first travels up the inclined plane due to its initial velocity and then eventually starts travelling downwards with acceleration g\*sin(theta). The computational and the exact solution match.

Code:

1. Q1.m

clear;close all;

global cnst initial\_vel;

g=9.8;

theta= 30 \* pi / 180;

cnst=g\*sin(theta);

initial\_vel = -20;

initial\_pos = 10;

timescale=1;

dt=timescale/100;

% set the initial and final times

tstart=0;

tfinal=10\*timescale;

% set the initial conditions in the u0 column vector

u0=zeros(2,1);

u0(1)=initial\_pos; % initial position;

u0(2)=initial\_vel; % initial velocity

[t,u]=ode45(@odeFunctionq1,[tstart:dt:tfinal],u0);

% store the solution that comes back into x and v arrays

x=u(:,1);

v=u(:,2);

x\_exact = (cnst/2).\*t.\*t + initial\_vel.\*t + initial\_pos;

v\_exact = cnst.\*t + initial\_vel;

% plot the position vs. time

plot(t,x,'g^',t,x\_exact)

title('Position vs. Time')

figure;

plot(t,v,'g^' , t , v\_exact)

title('Velocity vs Time')

figure;

plot(x,v, 'g^' , x\_exact, v\_exact)

title('phase-space plot of v vs. x')

1. odeFunctionq1.m

function F = odeFunctionq1( 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 cnst initial\_vel;

% make the column vector F filled with zeros

F=zeros(length(u),1);

% Now build the elements of F

%

% so the equation dx/dt=v means that F(1)=u(2)

F(1) = u(2) ;

% so the equation dv/dt=g\*sin(theta)

F(2)=cnst;

end

2. 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)

Angle of static friction = tan-1(0.4) = 21.80 degrees.

Angle of kinetic friction = tan-1(0.3) = 16.70 degrees.

If friction is taken into account , interesting cases can be observed.

Firstly if the angle of static friction is more than the angle of the plane than the block remains still and does not move. This is because the static frictional force is more than the gravitational force component along the plane.

If the inclination of the plane is more than the angle of static friction, then the block begins to slide and accelerated with an acceleration of a = **g\*sin(theta) – uk\*g\*cos(theta)**

Hence upon integration we get

x = 0.5\*a\*t^2

vx = a\*t

The following graphs illustrate that case :



Now suppose we give an initial velocity of -10 m/s. An interesting case arises wherein the block first moves upwards with deceleration a = **g\*sin(theta) + uk\*g\*cos(theta)**

then stops and then starts moving downwards with acceleration

a = **g\*sin(theta) – uk\*g\*cos(theta)**

Computationally we can verify the above analysis :







Note: The angle of inclination of the plane was taken to be 30 degrees.

**3. Projectile motion : Cannon shell / missile problem**

a) If we ignore the effects of air drag and air density, we can easily compute the exact solution.

The exact solution for x and y is as follows:

x\_exact = initv \* cos(theta).\*t;

y\_exact = initv \* sin(theta).\* t - 0.5 \* g .\* t .\* t;

In x-direction we have constant velocity and in y-direction we have constant acceleration of g in downward direction.

The graph of y vs x is a perfect parabola and the computational and the exact results match.



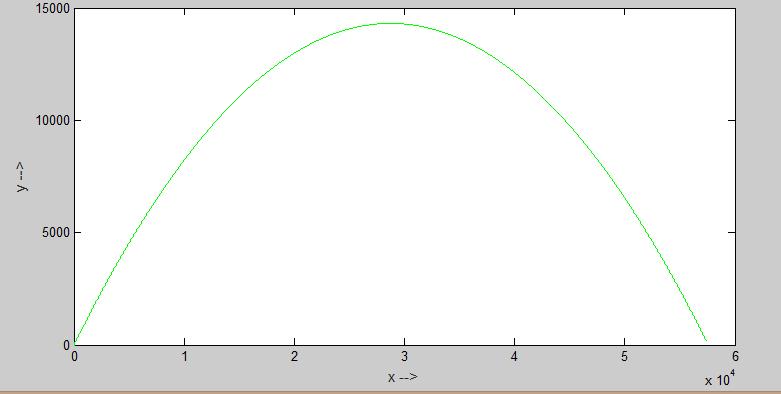
The above graph was taken for an angle of 45 degrees. It can be easily shown that the range is maximum for a given initial velocity for an angle of 45 degrees since the range formula is:

R = u\*u\* sin (2\*theta) / g .

Now if we consider the variation of g with altitude i.e g decreases as altitude increases, we can expect the time of flight and maximum height to increase. The formula used for dependence of g with altitude is:

g\_actual = g\*(1-(2\*y/R)) where R is the radius of the Earth = 6400 kms

The plot is as follows:



Note that the maximum height reached increases from **14.28 kms to 14.31kms** whereas the range remains approximately same at **57.4 kms**.

Matlab code:

1. Q3.m

clear all;

close all;

global g R constant;

g = 9.8;

R = 6400e3;

dt=.01;

initial\_x\_pos = 0;

initial\_y\_pos =0;

initv=750;

constant=4e-5;

g=9.8;

thetadegree=45;

theta=.0174\*thetadegree;

% time given so that projectile reaches ground

totaltime= 2\*initv\*sin(theta)/g;

starttime = 0;

%ux0 should contain initial conditions for x and vx

u0 = zeros(4,1);

u0(1) = initial\_x\_pos;

u0(2) = initv\*cos(theta);

u0(3) = initial\_y\_pos;

u0(4) = initv\*sin(theta);

[t,u]=ode45(@odeFunctionQ3 , [starttime:dt:totaltime] , u0);

x = u (: , 1);

y = u (: , 3);

% now we will calculate exact solutions

x\_exact = initv \* cos(theta) .\*t;

y\_exact = initv \* sin(theta).\* t - 0.5 \* g .\* t .\* t;

plot(x,y, 'g', x\_exact,y\_exact , 'r')

xlabel('x -->')

ylabel('y -->')

title('Ideal case without considering effect of air drag and air density - computationally-black and exact - green')

1. **odeFunctionQ3.m**

function F = odeFunctionQ3(t,u)

% F(1) = dx/dt

% F(2) = dvx/dt

% F(3) = dy/dt

% F(4) = dvy/dt

global g R;

F=zeros(length(u),1);

F(1) = u(2) ;

F(2) = 0;

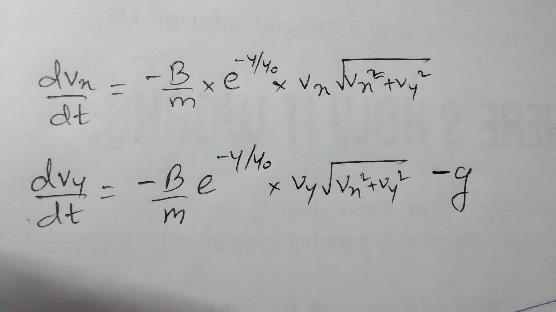
F(3) = u(4) ;

F(4) = -1 \* g \* (1 - (2\*u(3)/R));

% F(4) = -1\*g;

end

b) If we include effect of air resistance , then the equations change as follows :



Now if we perform the experiment for different firing angles we get the following observations :

|  |  |  |
| --- | --- | --- |
|  | MAX HEIGHT(kms) | RANGE(kms) |
| Ideal case angle = 45 | 14.28 | 57.4 |
| Considering effect of altitude on g , angle = 45 | 14.31 | 57.4 |
| With Air drag angle =10 | 0.726 | 14.77 |
| With Air drag angle =30 | 6.181 | 42.33 |
| With Air drag angle =45 | 12.83 | 51.35 |
| With Air drag angle =47 | 13.77 | 51.43 |
| With Air drag angle =50 | 15.18 | 51.05 |
| With Air drag angle =60 | 19.63 | 45.56 |
|  |  |  |

**So we can observe that the maximum range occurs for an angle of approximately 47 degrees which is slightly more than 45 degrees. This is an interesting observation and you must note that the effect of air drag decreases exponentially as height increases and hence for projectiles which travel a few kilometers into the air, the air drag is negligible.**

**Also if we had considered air drag without considering air density, then the result gives an maximum range for an angle of around 37 degrees. Here the effect of air drag plays an important role. So if we are considering a ball thrown into the air, the maximum range will be achieved for an angle close to 37 degrees as air density won’t change much.**

Matlab code –

1. Q3.m

clear all;

close all;

global g R constant;

g = 9.8;

R = 6400e3;

dt=.01;

initial\_x\_pos = 0;

initial\_y\_pos =0;

initv=750;

constant=4e-5;

g=9.8;

thetadegree=45;

theta=.0174\*thetadegree;

% time given so that projectile reaches ground

totaltime= 2\*initv\*sin(theta)/g;

starttime = 0;

%ux0 should contain initial conditions for x and vx

u0 = zeros(4,1);

u0(1) = initial\_x\_pos;

u0(2) = initv\*cos(theta);

u0(3) = initial\_y\_pos;

u0(4) = initv\*sin(theta);

[t,u]=ode45(@odeFunctionQ3b , [starttime:dt:totaltime] , u0);

x = u (: , 1);

y = u (: , 3);

plot(x,y, 'g')

xlabel('x -->')

ylabel('y -->')

title('Ideal case without considering effect of air drag and air density - computationally-black and exact - green')

2.odeFunctionQ3b.m

function F = odeFunctionQ3b(t,u)

% F(1) = dx/dt

% F(2) = dvx/dt

% F(3) = dy/dt

% F(4) = dvy/dt

global g R constant; % constant is b/m

F=zeros(length(u),1);

F(1) = u(2) ;

F(2) = - constant \* u(2) \*exp(-u(3) / 1000)\* sqrt (u(2)\*u(2) + u(4)\*u(4));

F(3) = u(4) ;

F(4) = - constant \* u(4) \*exp(-u(3) / 1000)\* sqrt (u(2)\*u(2) + u(4)\*u(4)) - g;

end

c)

The approach used is simple. To find the minimum velocity to hit a target at position (xt,yt) we use 2 for loops. The outer for loop is for the velocity required and the inner one is for the firing angle chosen.

**It is beneficial to take the outer for loop to be of velocity because once we find a suitable velocity we can directly break out of the 2 loops and move on to the next target position because it will be the minimum velocity**.



This problem is computationally challenging as in total we require 4 for loops and the computations are of the order of 10^8 .

If we increase the accuracy and hit a target at 10kms distance then we get the following graphs :



Also we can plot the firing angle corresponding to the minimum velocity required.



This question takes a lot of computations to solve. If we want accurate straight line graph for the 50 kms. target we need to increase the accuracy. If we do so it takes **5-6 hours** to solve. Hence I solved the problem for 10 kms target and varied the altitude from -2kms to 2kms. This took approximately **14 minutes** to compute.

**The observation is that the minimum velocity required to hit the target increases linearly with altitude.**

**The firing angle also increase linearly though the graph is not an exact straight line.**

A lot of optimizations were done in the code to reduce the number of computations(the optimisations have been labeled in the comments in the code ) -

1. A flag was used to detect if the projectile is within 20 m of the target. If yes we have found the minimum velocity and we need not go further. Hence we break out of the for loops using the flag and proceed to next position of target.
2. We know by intuition that if the altitude increases the minimum velocity required also increases. So instead of starting the loop from 290m/s, we can start it close to the minimum velocity calculated for the previous few altitudes of the target.
3. We can do an approximate analysis of what velocities and angles actually hit the target. And hence vary the range of the velocity and firing angle loops accordingly. For Eg for the target at 10kms distance and altitude ranging from -2 to 2 kms, I varied the firing velocity from 290 to 370 m/s and the angle from 30 to 55 degrees.

Matlab code :

clear all;

close all;

g = 9.8;

initial\_x\_pos = 0;

initial\_y\_pos =0;

constant=4e-5; % B/m

% time given so that projectile reaches ground

totaltime= 2\*600\*sin(pi/3)/g;

starttime = 0;

dt = 0.125 ;

npoints = round((totaltime - starttime) / dt) ; % approx points - very important parameter .

flag = 0 ;

xtarget =10000 ;

max\_ytarget\_ht = 2000;

min\_ytarget\_ht = -2000;

dh = 100 ; % increment in y target ht

npoints\_min\_vel = (max\_ytarget\_ht - min\_ytarget\_ht)/dh + 1;

min\_vel\_array = zeros(npoints\_min\_vel , 1);

firing\_angle\_array = zeros(npoints\_min\_vel , 1);

array\_index = 1 ;

vel\_lower\_limit = -1;

%let us measure the time required for our computations

tic

for ytarget = min\_ytarget\_ht : dh : max\_ytarget\_ht

min\_vel\_array(array\_index) = 1000; % check this assumption

flag =0; % optimisation 1

if(array\_index - 5 > 0)

vel\_lower\_limit = min\_vel\_array(array\_index - 5) ; % optimisation 2

else

vel\_lower\_limit = 290;

end

for v\_init = vel\_lower\_limit : 0.25 : 370 % assume this is the physical limitation

if( flag == 1 )

break;

end

for theta\_degrees = 30 : 0.5 : 55 % assume this is the physical limitation - optimisation 3

x = zeros (npoints);

y = zeros (npoints);

x(1) = initial\_x\_pos;

y(1) = initial\_y\_pos;

vx = v\_init \* cos(theta\_degrees\*0.01745);

vy = v\_init \* sin(theta\_degrees\*0.01745);

%now we simulate the motion of the projectile and break the

%loop if we are near the target

for step=2:npoints

x(step)=x(step-1)+vx\*dt;

y(step)=y(step-1)+vy\*dt;

drag=constant\*sqrt(vx\*vx+vy\*vy)\*exp(-y(step)/1000);

vx=vx-drag\*vx\*dt;

vy=vy-drag\*vy\*dt-g\*dt;

% break if we are within 1m radius of the target.

% use distance formula

if(sqrt(power(x(step)-xtarget,2) + power(y(step)-ytarget,2)) < 20 )

if(v\_init < min\_vel\_array(array\_index))

min\_vel\_array(array\_index) = v\_init ;

firing\_angle\_array(array\_index) = theta\_degrees ;

flag =1;

break; %update the minimum velocity

end

end

if((y(step) + 100 < ytarget) && x(step) > 25000) % break if you can't hit also make sure you don't break when you are climbing !!

break

end

end

if(flag == 1)

break; % we have got the minimum velocity ...why go further !

end

end

end

array\_index = array\_index +1 ;

end

time\_req = toc

plot(min\_ytarget\_ht : dh : max\_ytarget\_ht , min\_vel\_array)

grid on ;

xlabel('altitude of target in metres -->')

ylabel('minimum velocity required in m/s -->')

title('Finding the minimum velocity required to reach a target 10kms away at varying altitudes');

figure ;

plot(min\_ytarget\_ht : dh : max\_ytarget\_ht , firing\_angle\_array)

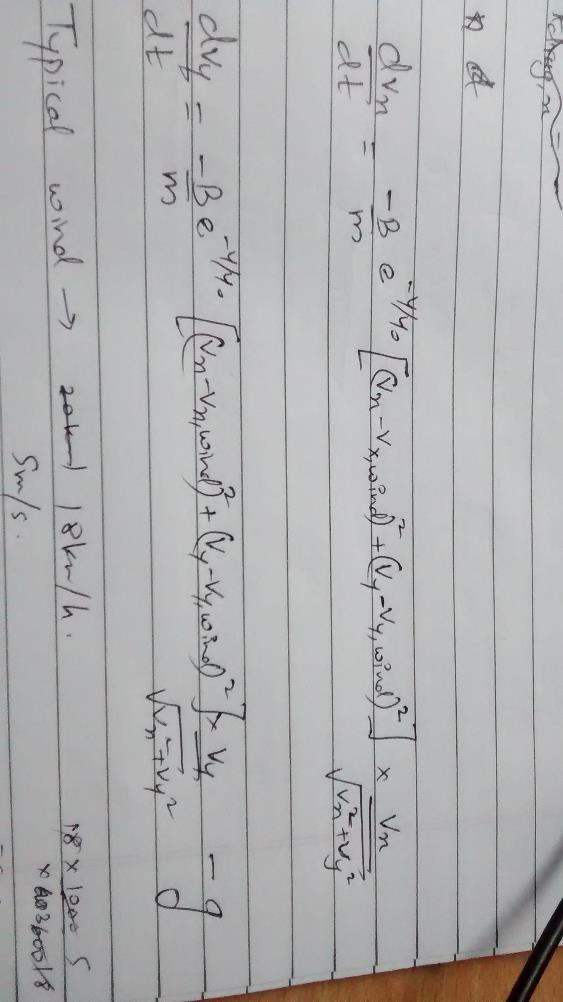
grid on ;

xlabel('altitude of target in metres -->')

ylabel('firing angle of minimum velocity in degrees -->')

title('Finding the firing angle of minimum velocity to reach a target 10kms away at varying altitudes');

d) If we take the effect of wind into consideration the only change required is that we need to consider the relative velocity of cannon ball wrt air to calculate the drag force. The equations are changed as follows:

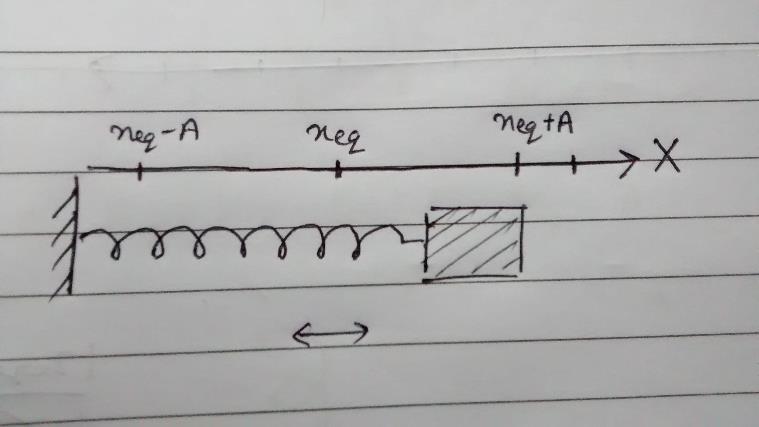
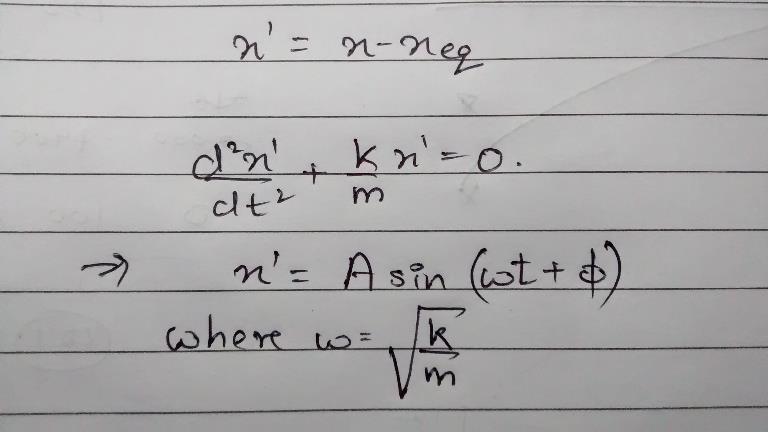


If the wind is in the x-direction of the projectile the range will increase by some amount. However if the wind is in opposite direction of the projectile the range decreases by some amount.

Q4. Simple Harmonic Motion: Computationally investigate the motion of a pendulum and a spring mass system as discussed in the class for damped, driven system. Draw phase plots to explain your observations.

Spring mass system –

For a spring mass system the equation is as stated below :

So we can see that the mass oscillates with SHM about the equilibrium point.

Matlab Code :

clear all;

close all;

k = 1; % spring constant

mass = 1 ;

timescale=2\*pi\*sqrt(mass/k);

initial\_disp = 10;

initial\_vel = 0;

dt=timescale/1000;

simulationt=10\*timescale;

npoints=round(simulationt/dt);

displacement=zeros(npoints,1);

velocity=zeros(npoints,1);

time=zeros(npoints,1);

displacement(1) = initial\_disp;

velocity(1) = initial\_vel;

for step=1:npoints-1

velocity(step+1)=velocity(step)-(k/mass)\*displacement(step)\*dt;

displacement(step+1)=displacement(step)+velocity(step + 1)\*dt;

time(step+1)=time(step)+dt;

end

plot(time,displacement)

xlabel('time in s')

ylabel('displacement in m/s')

grid on;

figure

plot(time,velocity)

xlabel('time in s')

ylabel('velocity in m/s')

grid on;

figure

plot(displacement ,velocity)

xlabel('displacement in m')

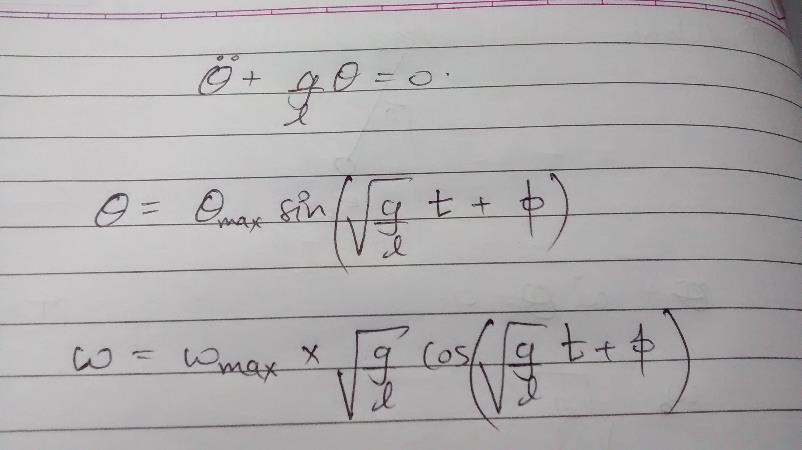
ylabel('velocity in m/s')

grid on;

Note that we have used the Euler Cromer method because the Euler method doesn’t work for SHM.

Pendulum –

The equation of a simple pendulum can be obtained using the angle with the vertical and the angular velocity. We can prove that the motion is similar to SHM and can be analyzed similarly.





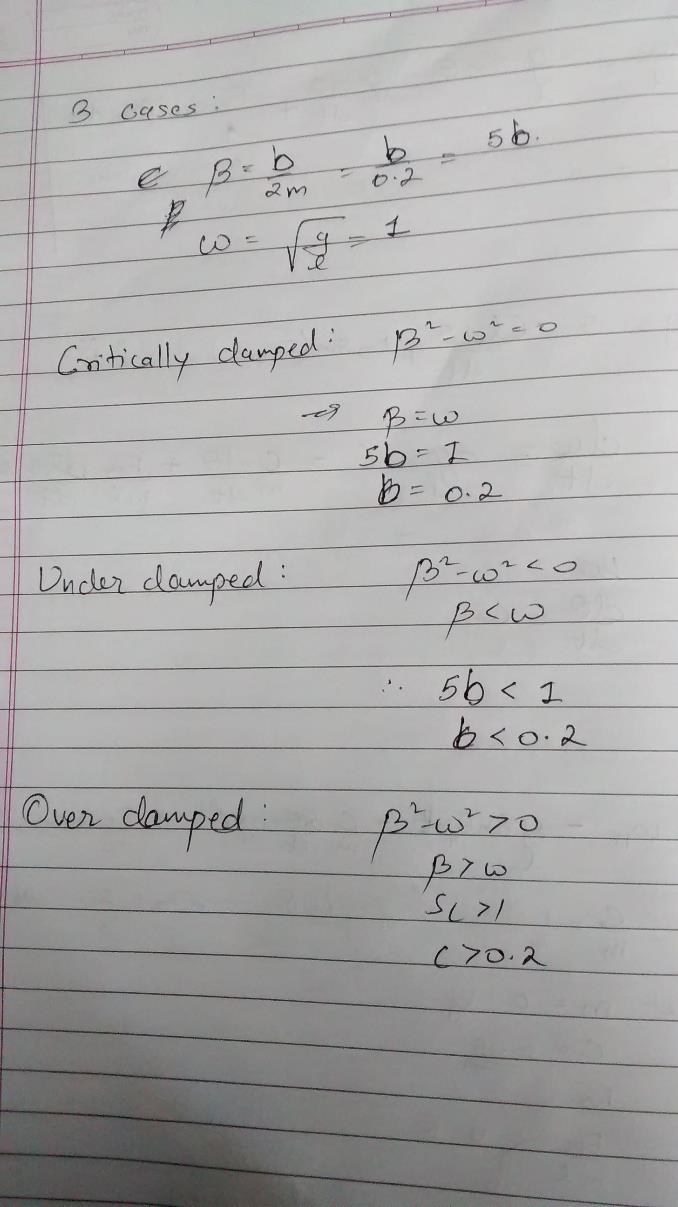


So we expect the Euler method to give the correct solution in which the oscillations go on forever and the amplitude of oscillations remains the same. However its surprising to note that the Euler method doesn’t give the correct solution for this problem. If we use that method the amplitude increases which is impossible as there is no other source of energy to the pendulam.

To get the correct answer we use **the Euler-Cromer method** in which there is a slight change while calculating theta(step + 1).

Theta(step + 1 ) = theta(step ) + angular\_vel (step + 1) \* dt ;



If we include the damping factor, we can observe three cases : 

Also the values of the constants used are:

Mass of pendulum(m) = 0.1 Kg

Drag force constant (b) = 0.2 (for critical damping)

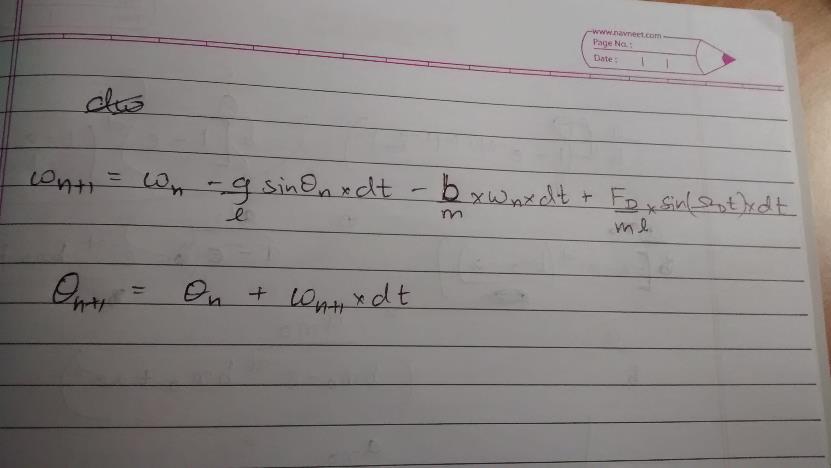
Length = 9.8 m

g = 9.8 m/s^2

Driving force amplitude (Fd) = 0 N

Hence the natural angular frequency of oscillation is 1 .

The equations used are :



1. Under Damping : The oscillations die down gradually. (b = 0.01)







1. Critical Damping : No oscillations are seen. Instead the angle theta goes to 0 almost immediately.(b = 0.2)







1. Over Damping : The angle goes down to 0 exponentially. (b = 1 )







Now if we give a driving force with its own driving frequency then, we find that for small values of the force , oscillations of driver frequency are seen. For large values of force, there is chaos.

Driver angular frequency = 0.66

Amplitude of driver force = 0.5 N







Driver angular frequency = 0.66

Amplitude of driver force = 1.5 N







Matlab code :

clear all;

close all;

g=9.8;

length=9.8;

timescale=2\*pi\*sqrt(length/g);

mass = 0.1;

C = 0.05 % C is the drag constant Fdamp = C\*length\*omega

Fd = 1.5 % this is the driving force

driver\_freq = 0.66; % its the angular frequency

dt=timescale/1000;

simulationt=10\*timescale;

npoints=round(simulationt/dt);

theta=zeros(npoints,1);

omega=zeros(npoints,1);

time=zeros(npoints,1);

theta\_degrees = 10 ;

theta(1)= theta\_degrees \* pi / 180;

%for step=1:npoints-1

%omega(step+1)=omega(step)-(g/length)\*theta(step)\*dt;

%theta(step+1)=theta(step)+omega(step +1)\*dt;

%time(step+1)=time(step)+dt;

%end

%the following code is for simulating damp driven system

for step=1:npoints-1

omega( step + 1) = omega (step) - (g/length)\*sin(theta(step))\*dt - (C/mass)\*omega(step)\*dt + (Fd/(mass\*length))\*sin(driver\_freq\*time(step))\*dt;

theta(step+1)=theta(step)+omega(step + 1)\*dt;

if theta(step + 1) > pi

theta (step +1) = theta (step +1) - 2\*pi;

elseif theta(step + 1) < -pi

theta (step +1) = theta (step +1) + 2\*pi;

end

time(step+1)=time(step)+dt;

end

theta=57.5\*theta;

plot(time,theta)

xlabel('time')

ylabel('theta in degrees')

grid on;

figure

plot(time,omega)

xlabel('time')

ylabel('angular velocity')

grid on;

figure

plot(theta ,omega)

xlabel('theta')

ylabel('angular velocity')

grid on;

Conclusion of the report :

The first 2 problems were pretty straight forward and could even be solved analytically and be rectified. The 3rd problem was the one which took much time to solve computationally.