NYU TANDON SCHOOL OF ENGINEERING

SPRING | 2016

ME-GY 7923 Robotic Gait & Manipulation

Integrated Term Project
Final Report

Dynamic Simulation of Centipede locomotion inspired by Origami Art

Angad A. Boralkar
Yoon-ho Seol

Abstract

The objective of this project is to simulate a dynamic model of centipede motion in a robotic software tool. The goal of this project is to propose a new theoretical approach for building a robotic actuator (or joint) that can mimic the motion of centipede legs when a sensory input is provided. In this proposal, it is assumed that the legs of a centipede follow a sine/cosine curve during locomotion. The objective is achieved by computing the dynamics and kinematics of centipede's legs at small intervals of time. These characteristics are then given as inputs to software based simulation model that translates the dynamics of motion into real time. In this way, it is possible to theoretically propose a new type of actuator, which can follow the motion of a centipede.

Table of Contents

INTRODUCTION 4

MATHEMATICAL BACKGROUND 5

PROGRAMMING AND CODES 07

1. First Joint/Joint 0 07

2. Next Joint 08

3. Pattern of Joints 08

4. Joints and Angles between Links at each Time Instant 09

5. Angular Velocity 10

6. Angle of each link with respect to positive X axis 12

VREP MODEL AND SIMULATION 17

1. Initialization 17

2. Running the Simulation 18

CONCLUSION & DISCUSSION 19

REFERENCES 19

Introduction

In recent times, miniature robots, which are capable of replicating the motion of animals, birds and insects, are becoming the center attraction in the robotics industry. Due to the recent advances in micro manufacturing techniques and in-depth understanding of biological insect locomotion, creation of multiplexed miniature robots is possible. Nearly the entire existing and upcoming multiple miniature legged robots take inspiration from a rigid-body hexapod system. However, another special and unique morphology found in nature is that of a centipede. Many legs and a flexible body characterize it. These facets are able to offer performance and efficiency benefits with respect to mobility, agility, stability, control and robustness.

As a preliminary step, we had to understand the dynamics and kinematics of a centipede's motion thoroughly. After researching about it, we found that the legs of a centipede follow a sine or cosine curve during locomotion. This is not the only pattern followed by the centipede but the focus for this project is based on this sine and cosine articulation. This peculiar pattern gives the insect maximum control of its body during movement. It is also an extremely effective way of locomotion as it is widely followed throughout most of the animal kingdom, including fishes and other insects like the millipede.

We used the origami approach to create a physical model of each leg of the centipede and acquired a visual understanding of it. Origami is a Japanese art form of folding paper into various shapes and figures. One of the interesting facts about origami is that it allows for a thorough and complete understanding of any dynamic motion. This is because of a perfect balance between flexibility and rigidity of the origami art when it is subjected to a certain fold. We started by creating different origami figures of a centipede's legs and applying sinusoidal inputs to each end of the paper. By doing this, we could observe the motion trajectory of the wave along the legs at any given instant of time. Taking an inspiration from this, we devised a mathematical code of this trajectory into programming software MATLAB. We used this to obtain graphs of the nature of the trajectory at different intervals of time. We could estimate the position, velocity and acceleration characteristics of each joint for one step size of the centipede. We then used robotic simulation software VREP to assign these values as inputs to a kinematic linkage of revolute joints. By running the simulation in VREP for ten seconds, we were able to replicate the motion of centipede using rigid linkages.



Figure: A picture of common centipede.

Mathematical Background

As a start point in the analysis of this project, a brief research and understanding of sinusoidal waves was conducted. For formulation of the model, each joint was considered discretely and using a function, the direction and magnitude of each position vector was derived at every instant of time.

The position vector 'p' of each joint at any instant of time 't' followed the expression:

$$pt_{joint} = \cos(\omega t + \phi)$$

where ω is the frequency of the wave and ϕ is the phase difference.

The position for two consecutive links is calculated and this is used to find the angle between them.

For each link the expression to calculate the position vector is given by:

$$link_n = pt_n - pt_{n-1}$$

where 'n' denotes the number of joints.

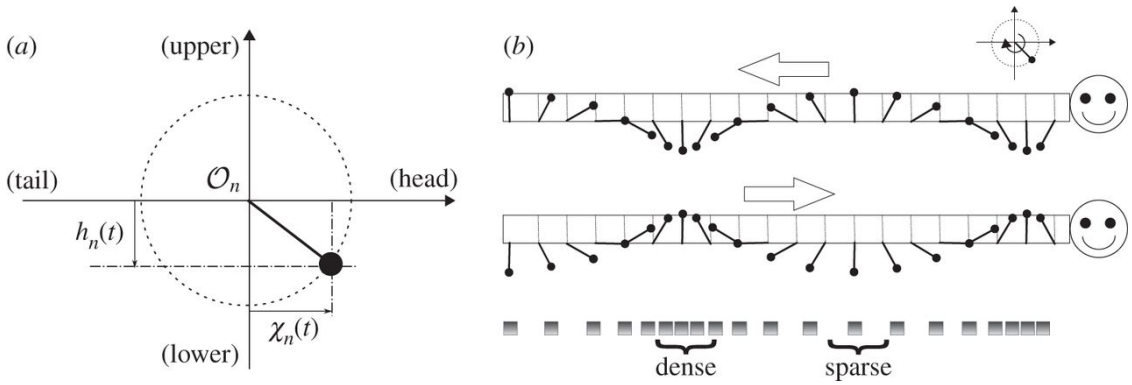
This is used to calculate the angle between two consecutive links. The angle is given by the following expression.

$$\theta_{joint} = \pm \text{acos}(link_{n-1} \bullet link_n / ([link_{n-1}] [link_n]))$$

As seen from the above expression, a cosine inverse of the dot product of two consecutive links is used to find the joint angle between them.

This joint angle is further used to calculate the velocity and acceleration of the joint. Single derivative of the joint angle gives the angular velocity and double derivative of the joint angle gives angular acceleration.

The following figure gives a better understanding of motion of legs of a centipede. With a change in the value of joint angle there is a corresponding movement of the insect in a particular direction.



Programming and Codes

The mathematical data obtained from the previous section is used to write codes in MATLAB. The codes are commented at every step to give details about each operation. This code will produce different joint angles for 'n' joints in an 'n-1' link robot.

1. First Joint/Joint 0:

About the code:

This code gives joint positions for n joints.

```
% Joint Positioning

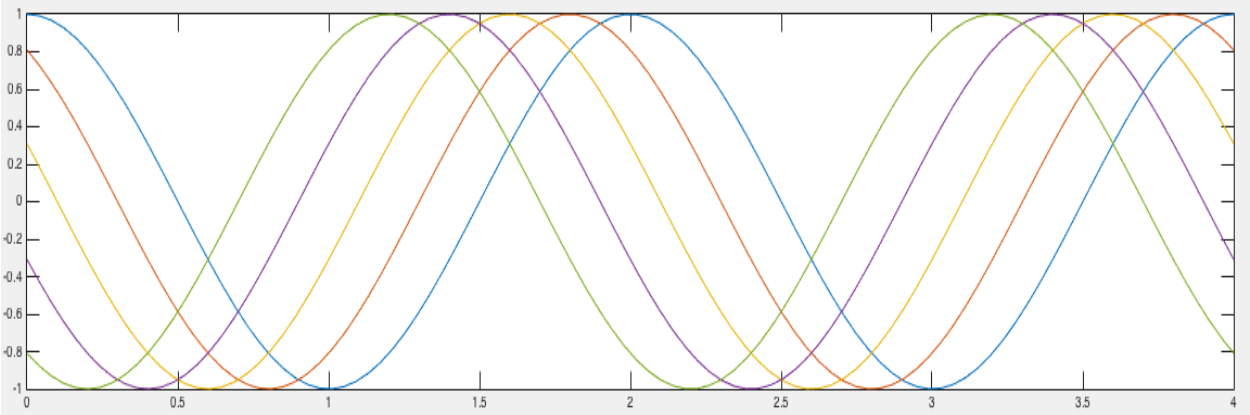
function [JointPt, LinkSize] = njoints_(vers, phase, M, n) % n >1
% [Points, Link Length]

Pt0 = complex(0, shape_(1, phase));              % Starts Matrix from x = 0
JointPt = [Pt0; 0];                              % JointSet: Col vec
LinkSize = [0;0];                                % Len Link: Row vec

for i = 2:n
    tempRefPt = JointPt(i-1);
    [JointPt(i), LinkSize(i)]  = nextjoint_(vers, phase, M, tempRefPt);
end


end
```

Results:



2. Next Joint:

About the code:

This code gives the position of next joint on the curve.

```
% Find Distance Complex While

function [pt,l] = nextjoint_(vers, phase, M, refPt)

refX = real(refPt);
ptX = refX;                                          % Starts from refPt
l = 0;                                               % starts from Link = 0

while  (l < M) && ((ptX-refX) < 2*pi)                % In case a Link goes over
a wave length
    ptX = ptX + 0.00001;
    pt = complex(ptX,shape_(vers, phase + ptX));
    l = abs((pt - refPt));                           % Magnitude(Length) of
Link
end                                                  % Escape when l = M +
0.0000?
```

3. Pattern of the Joints:

About the code:

This code recognizes the pattern of each joint.

```
function  [JointSet, LinkSize] = patternjoints_(vers, phase, M, error)

Pt0 = complex(0,cos(phase));
JointSet = [Pt0;0];
LinkSize = [0];

% Pattern
i = 1;
iniH = imag(Pt0);
newH = iniH +error+1;

while (newH > iniH +error) || (newH < iniH -error)          %until within
error
    i = i +1;
    refPt = JointSet(i-1);
    [JointSet(i), LinkSize(i)]  = nextjoint_(vers, phase, M, refPt);
    newH = imag(JointSet(i));
end

end
```

4. Joints and Angles between Links at each Time Instant:

About the code:

This code gives the position of each joint as well as the angle between consecutive links at each time interval.

```
% The time instant Joints & Angles
clear all;

%shape
vers = 1;                                       % or 2   cos(x)/ exp^(-
0.2*x)*cos(x)
wavelength = 2*pi;                              % 2*pi meter
phase = 0;                                      % or 0.25/ 0.5/ -0.25
cos/sin/-cos/-sin

% spec
M = 1;                                          % M < 2*pi
L = 0.5; pos = 0.5 -1i*L/M;                     % leg length

% tolerance
% error = 0.01;                                 % or 0.001
n = 16;                                         % 16 or 31
vel = 0.03;                                     % 0.03 m per frame
fps = 5; spf = 1/fps;                          % 10 frame per second
T = 20;  nframes = fps*T;                       % 30 seconds


%%% Find Joints
for frame = 1:nframes %seconds
    phase = frame*vel;
    [JointPt, LinkSize] = njoints_(1, phase, M, n);
    % [JointPt, LinkSize] = patternjoints_(vers, phase, M, error);
    % n = length(JointPt);
    %%% 1 frame calc
    id = 2:n;
    LinkVec = JointPt(id) - JointPt(id-1);
    LegEndPt = (   JointPt(id-1) + LinkVec(id-1) .*(pos)   );
    %%% multi frame calc
    Joints(:,frame) = JointPt;
    EndPts(:,frame) = LegEndPt;
    Angles(:,frame) = 2*pi*angle(LinkVec);
end

jX = real(Joints); jY = imag(Joints);          % joints position
eX = real(EndPts); eY = imag(EndPts);          % end pts position

disp('Link #: '),disp(n)
disp('1st frame Angles: '), disp(Angles(:,1))
for k = 1:T
 pause (spf)                                    % see every 0.1 second
 plot(jX(:,k),jY(:,k),'-.')
end
for k = 1:T
 pause (spf)                                    % see every 0.1 second
 plot(eX(:,k),eY(:,k),'o')
end
```

5. **Angular Velocity:**

About the code:

This code gives the angular velocities of each joint. The result shows the nature of these velocities.

```
% Angular velocity
clear all;
format short;
%%% Try at vel 0.003, vel 0.03, vel 1, vel 20
%%% *** T*vel >= 1 (2*pi) for joint to return to original point
%shape
vers = 1;                                   % or 2   cos(x)/ exp^(-
0.2*x)*cos(x)
wavelength = 2*pi;                          % 2*pi meter
shift = 0*(0.5*pi); % 90deg unit             % or 0/0.25/0.5/-0.25
cos/sin/-cos/-sin

% spec
M = 1;                                      % M < 2*pi
L = 0.5; pos = 0.5 -1i*L/M;                 % leg length & relative
position

% tolerance
% error = 0.01;                             % or 0.001
n = 3;                                      % 16pts; 15links;
14joints <16 or 31>
vel = 0.1*(2*pi);  % relative to 2pi      % 0.03 m per frame
fps = 20; spf = 1/fps;                      % 50miliseconds
T = 10;  nframes = fps*T;                  % 30 seconds

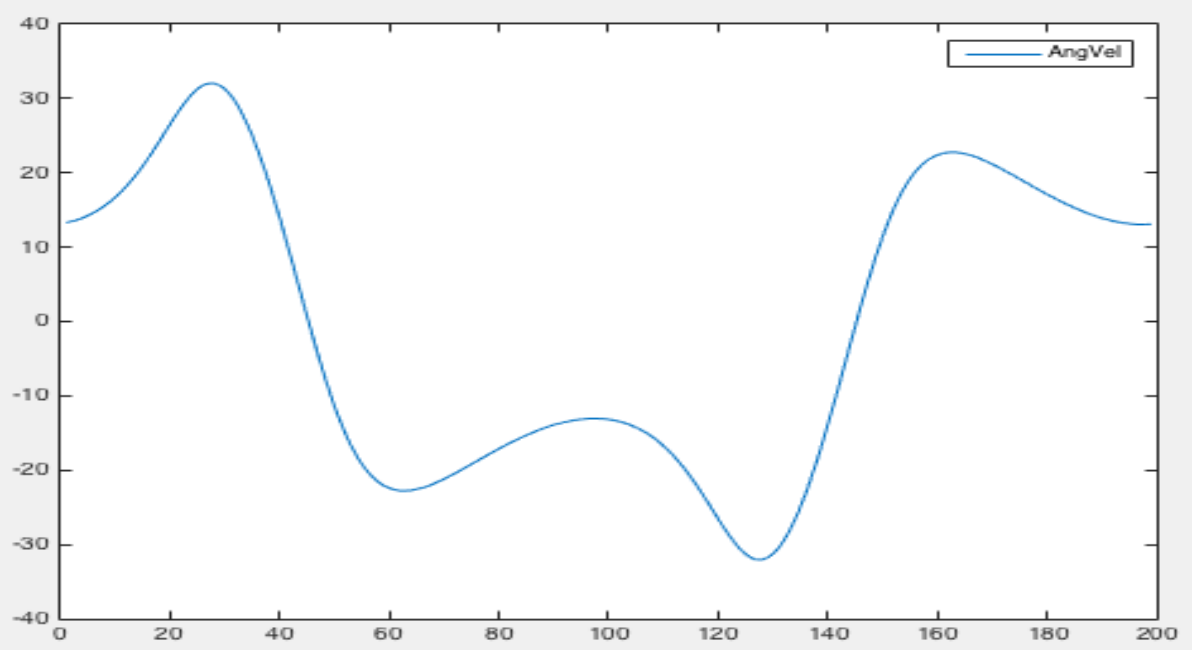

%%% Find Joints & Angles
for frame = 1:nframes %seconds
    phase = frame*spf*vel + shift;
    [JointPt, LinkSize] = njoints_(1, phase, M, n);
    % [JointPt, LinkSize] = patternjoints_(vers, phase, M, error);
    % n = length(JointPt);
    %%% 1 frame calc
    id = 2:n;
    LinkVec = JointPt(id) - JointPt(id-1);
    LegEndPt = (   JointPt(id-1) + LinkVec(id-1) .*(pos)   );
    %%% multi frame calc
    Joints(:,frame) = JointPt;
    EndPts(:,frame) = LegEndPt;
    id2 = 2:n-1;
    Angles(:,frame) = 180./pi.*( angle(LinkVec(id2))-
angle(LinkVec(id2-1)) );
end
% Joints, Links
disp('Link #: '),disp(n)
jX = real(Joints); jY = imag(Joints);         % joints position
eX = real(EndPts); eY = imag(EndPts);         % end pts position
for k = 1:nframes % for T time
 pause (spf)                                % see every 0.1 second
```

```
 plot(jX(:,k),jY(:,k),'-.')
end
for k = 1:nframes
 pause (spf)                                    % see every 0.1 second
 plot(eX(:,k),eY(:,k),'o')
end

%%% Find Angular Velocity
AngVel(:,1:nframes-1) = ( Angles(2:nframes)-Angles(1:nframes-1)
).*fps; % dividing by 50ms
% Angles
disp('1st frame Angles: '), disp(Angles(:,1:2))
% Angular Velocity - every frame
for k = 1:nframes-1 %cause 1:399
 pause (spf)                                    % every 1 second
 disp(AngVel(:,k))
end
% - every second
disp('AngVel every second')
for k = 0:T-1 %cause 1:399
 pause (1)                                    % every 1 second
 disp(AngVel(:,1+ fps*k+1))
end
% plot first joint velocity
plot(AngVel(1,:))

%maximum angle is +-46.358397693423527  (format long)
@vel:0.005*2*pi
disp([13, 'Max, Min Angle is: ']), disp(max(Angles)),
disp(min(Angles)),
disp('Max, Min Velocity is: '), disp(max(AngVel)),
disp(min(AngVel))
```

Result:



6. **Angle of each link with respect to positive X axis:**

About the code:

This code gives angle of each link with respect to the X axis. We use this data as input for the VREP model to see a dynamic simulation.

```
disp('Program started');
    % vrep=remApi('remoteApi','extApi.h'); % using the header (requires a
compiler)
    vrep=remApi('remoteApi'); % using the prototype file
(remoteApiProto.m)
    vrep.simxFinish(-1); % just in case, close all opened connections
    clientID=vrep.simxStart('127.0.0.1',19997,true,true,5000,5);

    if (clientID>-1)
        disp('Connected to remote API server');

        % enable the synchronous mode on the client:
        %vrep.simxSynchronous(clientID,true);

        % start the simulation:
        vrep.simxStartSimulation(clientID,vrep.simx_opmode_blocking);

        % Now step a few times:

        [~, handle_C1] =
vrep.simxGetObjectHandle(clientID,'Cylinder_1',vrep.simx_opmode_oneshot_wa
it);
        [~, handle_C2] =
vrep.simxGetObjectHandle(clientID,'Cylinder_2',vrep.simx_opmode_oneshot_wa
it);
        [~, handle_C3] =
vrep.simxGetObjectHandle(clientID,'Cylinder_3',vrep.simx_opmode_oneshot_wa
it);
        [~, handle_C4] =
vrep.simxGetObjectHandle(clientID,'Cylinder_4',vrep.simx_opmode_oneshot_wa
it);
        [~, handle_C5] =
vrep.simxGetObjectHandle(clientID,'Cylinder_5',vrep.simx_opmode_oneshot_wa
it);
        [~, handle_C6] =
vrep.simxGetObjectHandle(clientID,'Cylinder_6',vrep.simx_opmode_oneshot_wa
it);
        [~, handle_C7] =
vrep.simxGetObjectHandle(clientID,'Cylinder_7',vrep.simx_opmode_oneshot_wa
it);
        [~, handle_C8] =
vrep.simxGetObjectHandle(clientID,'Cylinder_8',vrep.simx_opmode_oneshot_wa
it);
        [~, handle_C9] =
vrep.simxGetObjectHandle(clientID,'Cylinder_9',vrep.simx_opmode_oneshot_wa
it);
        [~, handle_C10] =
vrep.simxGetObjectHandle(clientID,'Cylinder_10',vrep.simx_opmode_oneshot_w
ait);
        [~, handle_C11] =
vrep.simxGetObjectHandle(clientID,'Cylinder_11',vrep.simx_opmode_oneshot_w
```

```
ait);
        [~, handle_C12] =
vrep.simxGetObjectHandle(clientID,'Cylinder_12',vrep.simx_opmode_oneshot_w
ait);
        [~, handle_C13] =
vrep.simxGetObjectHandle(clientID,'Cylinder_13',vrep.simx_opmode_oneshot_w
ait);
        [~, handle_C14] =
vrep.simxGetObjectHandle(clientID,'Cylinder_14',vrep.simx_opmode_oneshot_w
ait);
        [~, handle_C15] =
vrep.simxGetObjectHandle(clientID,'Cylinder_15',vrep.simx_opmode_oneshot_w
ait);
        [~, handle_C16] =
vrep.simxGetObjectHandle(clientID,'Cylinder_16',vrep.simx_opmode_oneshot_w
ait);

        [~, handle_R1] =
vrep.simxGetObjectHandle(clientID,'Revolute_1',vrep.simx_opmode_oneshot_wa
it);
        [~, handle_R2] =
vrep.simxGetObjectHandle(clientID,'Revolute_2',vrep.simx_opmode_oneshot_wa
it);
        [~, handle_R3] =
vrep.simxGetObjectHandle(clientID,'Revolute_3',vrep.simx_opmode_oneshot_wa
it);
        [~, handle_R4] =
vrep.simxGetObjectHandle(clientID,'Revolute_4',vrep.simx_opmode_oneshot_wa
it);
        [~, handle_R5] =
vrep.simxGetObjectHandle(clientID,'Revolute_5',vrep.simx_opmode_oneshot_wa
it);
        [~, handle_R6] =
vrep.simxGetObjectHandle(clientID,'Revolute_6',vrep.simx_opmode_oneshot_wa
it);
        [~, handle_R7] =
vrep.simxGetObjectHandle(clientID,'Revolute_7',vrep.simx_opmode_oneshot_wa
it);
        [~, handle_R8] =
vrep.simxGetObjectHandle(clientID,'Revolute_8',vrep.simx_opmode_oneshot_wa
it);
        [~, handle_R9] =
vrep.simxGetObjectHandle(clientID,'Revolute_9',vrep.simx_opmode_oneshot_wa
it);
        [~, handle_R10] =
vrep.simxGetObjectHandle(clientID,'Revolute_10',vrep.simx_opmode_oneshot_w
ait);
        [~, handle_R11] =
vrep.simxGetObjectHandle(clientID,'Revolute_11',vrep.simx_opmode_oneshot_w
ait);
        [~, handle_R12] =
vrep.simxGetObjectHandle(clientID,'Revolute_12',vrep.simx_opmode_oneshot_w
ait);
        [~, handle_R13] =
vrep.simxGetObjectHandle(clientID,'Revolute_13',vrep.simx_opmode_oneshot_w
ait);
        [~, handle_R14] =
vrep.simxGetObjectHandle(clientID,'Revolute_14',vrep.simx_opmode_oneshot_w
ait);
        [~, handle_R15] =
```

```
vrep.simxGetObjectHandle(clientID,'Revolute_15',vrep.simx_opmode_oneshot_w
ait);

pause on;

for q = 0:9

    v = 16; % Number of segments of sine wave

    i = linspace(0,2*pi,v);

    deg = q*10;

    theta = deg*(pi/180);

    z = sin(i+(theta));
    plot(i,z)
    x = transpose(i);
    y = transpose(z);

    p = [x y];

    for e = 2:1:v

        for j = 1:1
            m(e,j) = x(e) - x(e-1);
            n(e,j) = y(e) - y(e-1);
        end

    end

    l = m;
    k = n;
    l(1,:) = []; %This matrix gives difference of consecutive 'x'
    k(1,:) = []; %This matrix gives difference of consecutive 'y'

    h3 = [l,k];

    h = [h3 zeros(size(h3,1),1)];




    for u = 1:(v-1)

        h1 = h(u,:);
        b = [1 0 0];

        cr = cross(h1,b);
        do = dot(h1,b);

        r2 = atand(cr/do); %This gives tan inverse in degrees
        r(u,1) = r2(3); %Form a matrix of all the angles in degrees

    end
```

```
    angle2 = 1*r;

    w = q+1;

    angle1(:,w) = [angle2];


end

angle = angle1;

pause on;

for v1 = 1:9

    pause(1.5);


vrep.simxSetJointTargetPosition(clientID,handle_R1,angle(1,v1),vrep.simx_o
pmode_oneshot);

vrep.simxSetJointTargetPosition(clientID,handle_R2,angle(2,v1),vrep.simx_o
pmode_oneshot);

vrep.simxSetJointTargetPosition(clientID,handle_R3,angle(3,v1),vrep.simx_o
pmode_oneshot);

vrep.simxSetJointTargetPosition(clientID,handle_R4,angle(4,v1),vrep.simx_o
pmode_oneshot);

vrep.simxSetJointTargetPosition(clientID,handle_R5,angle(5,v1),vrep.simx_o
pmode_oneshot);

vrep.simxSetJointTargetPosition(clientID,handle_R6,angle(6,v1),vrep.simx_o
pmode_oneshot);

vrep.simxSetJointTargetPosition(clientID,handle_R7,angle(7,v1),vrep.simx_o
pmode_oneshot);

vrep.simxSetJointTargetPosition(clientID,handle_R8,angle(8,v1),vrep.simx_o
pmode_oneshot);

vrep.simxSetJointTargetPosition(clientID,handle_R9,angle(9,v1),vrep.simx_o
pmode_oneshot);

vrep.simxSetJointTargetPosition(clientID,handle_R10,angle(10,v1),vrep.simx
_opmode_oneshot);

vrep.simxSetJointTargetPosition(clientID,handle_R11,angle(11,v1),vrep.simx
_opmode_oneshot);

vrep.simxSetJointTargetPosition(clientID,handle_R12,angle(12,v1),vrep.simx
_opmode_oneshot);

vrep.simxSetJointTargetPosition(clientID,handle_R13,angle(13,v1),vrep.simx
_opmode_oneshot);

vrep.simxSetJointTargetPosition(clientID,handle_R14,angle(14,v1),vrep.simx
_opmode_oneshot);
```

```
vrep.simxSetJointTargetPosition(clientID,handle_R15,angle(15,v1),vrep.simx
_opmode_oneshot);

end

pause off;

% stop the simulation:
vrep.simxStopSimulation(clientID,vrep.simx_opmode_blocking);

% Now close the connection to V-REP:
vrep.simxFinish(clientID);

else
disp('Failed connecting to remote API server');

end
vrep.delete(); % call the destructor!

disp('Program ended');
```

VREP Model and Simulation:

The VREP model consists of 16 joints. These joints form a 15-link robot. All the joints are considered to be revolute and the links as cylinders. This is done for simplicity of the simulation. The above MATLAB codes are used to assign Joint Parameters in terms of angle theta to the revolute joints in VREP. Software tools like 'Remote Api Bindings' are used for communication between MATLAB and VREP. The dynamic simulation model is run for ten seconds and it shows an approximate behavior of each link when every joint in the robot approaches its pre-calculated joint variable.

Initialization:

Before the simulation starts, the robot joints parameters are set to zero and the orientation is as shown below.

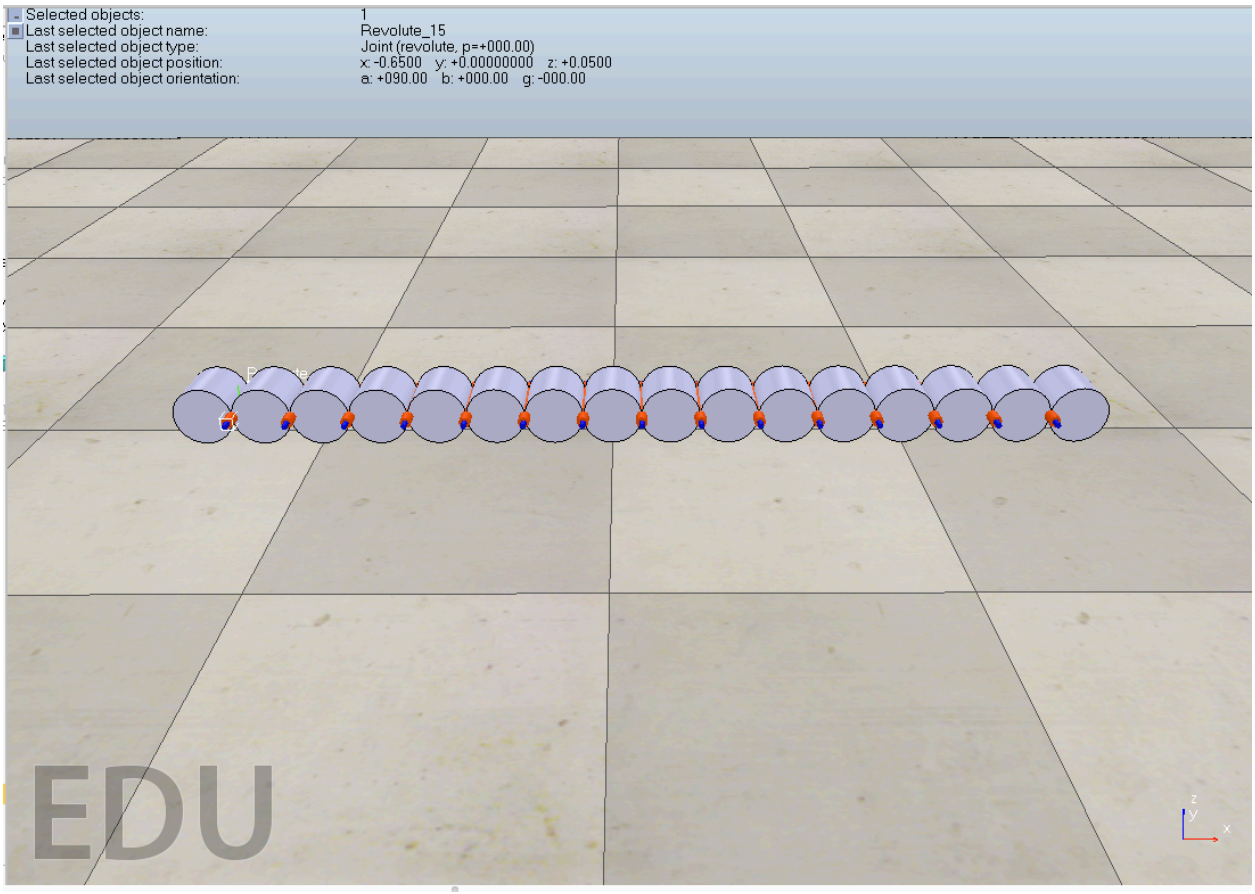
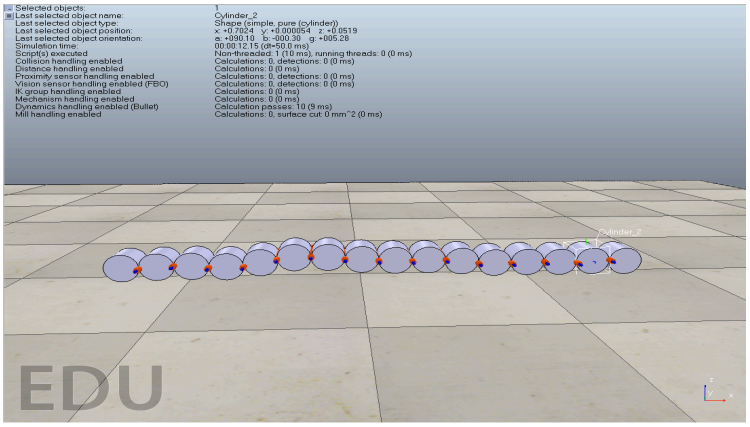


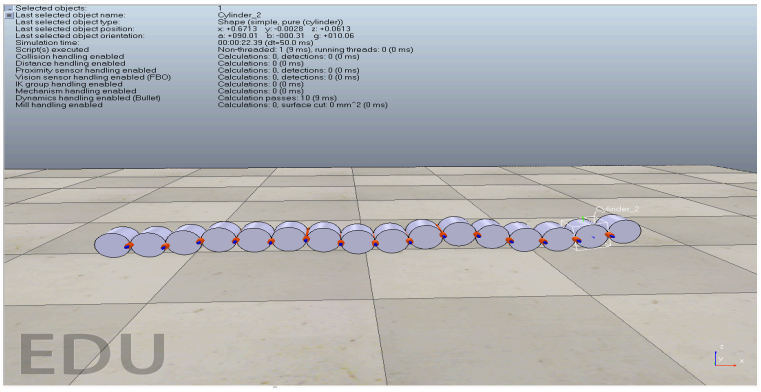
Figure: The original orientation and configuration of robot before the start of simulation.

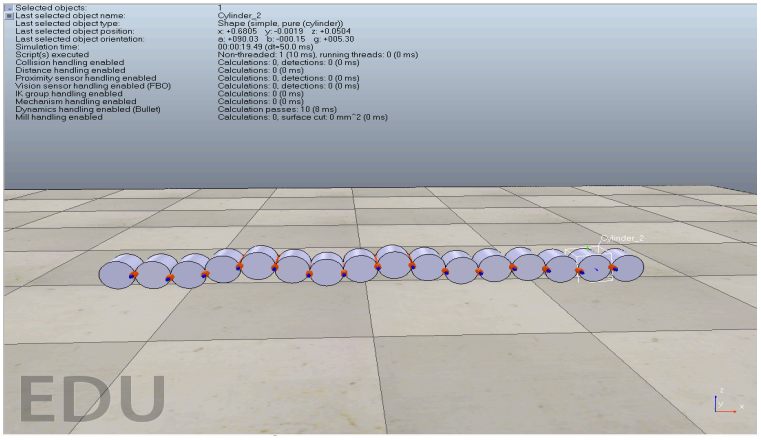
Running the Simulation:

After the simulation starts, each joint is assigned a parameter. As all the joints in this model are revolute joints, the input is given as an angle (in degrees). The parameters are updated at every second and the simulation stops once the time reaches ten seconds.

The following figures show the configuration and orientation of the robot at a certain interval of time.







Conclusion & Discussion:

The simulation proves that the robotic model is able to implement the motion of a centipede. It can be improved my dividing the model into more number of links and running more simulations at shorter intervals of time. This will allow for utmost precision and exact dynamics of locomotion.

In this way, the proposed idea is implemented and satisfactory results are obtained. As a future scope, the model can be configured in such a way to obtain 3D movement of the joints (and links) by 2D manipulation of the joints. Similar to the case observed in Origami art forms.

Another important conclusion is that, by observing the results of this project, it is theoretically possible to have a single actuator, which will follow a sine, or cosine curve when input is given to it. This is important because many of the existing biologically inspired mobile robots use revolute and prismatic joints to achieve various trajectories of locomotion. Such joints increase the complexity of the system and thereby increasing the cost. By this project, we have shown that a single actuator can be developed which will eliminate the need to use of a number of revolute (or prismatic) joints in a robot to get a sinusoidal nature of locomotion.

REFERENCES:

[1] Design and Locomotion Studies of a Miniature Centipede-Inspired Robot, Katie Lynn Hoffman, PhD. Dissertation, Harvard University, Cambridge, Massachusetts, May 2013.

[2] Semiautonomous centipede-like robot for rubble - development of an actual scale robot for rescue operation, Kazuyuki Ito, Yasunori Ishigaki, International Journal of Advanced Mechatronic Systems, Vol. 6, No. 2/3, 2015.

[3] Centipede robot for uneven terrain exploration: Design and experiment of the flexible biomimetic robot mechanism, Dooyeol Koh, Jaemin Yang, Soohyun Kim, 3rd IEEE RAS and EMBS International Conference on Biomedical Robotics and Biomechatronics (BioRob), 2010.