ENPM 662 - FINAL PROJECT

SNAKE ROBOT

DECEMBER 9, 2020

AUTHORS :
ANUBHAV PARAS (116905909)
PREYASH RAJESHKUMAR PARIKH (117303698)

Contents

Abstract

Snake Robots have many degrees of freedom, which makes them extremely versatile and complex to control. This report presents a modular snake robot, its electronic architecture and control. Inspired by biological snakes, snake robots move using cyclic motions called gaits. These cyclic motions directly control the snake robot's internal degrees of freedom which causes a net motion. Each mode of the robot is controlled by a sinusoidal oscillator with four parameters: amplitude, frequency, phase, and offset.

Introduction

A Snake robot is a biomorphic hyper-redundant robot that resembles a biological snake. There are various snake robots which are constructed by chaining together a number of independent links. This redundancy makes them resistant to failure, because they can continue to operate even if parts of their body are destroyed. The redundancy in configurations gives them the technical name: hyper redundant robots.

Motivation

People have tried to make technology that can coordinate with their internal degrees of freedom to perform various locomotive gaits, but snake robots can be highly articulated and robust that can pass through compact places where legged or wheeled robots cannot reach.

Snake Robots have many degrees of freedom, which make them extremely versatile and complex to control. We propose to develop a modular snake robot that can use its multiple degrees of freedom and thread through constrained locations and go beyond the capabilities of conventional legged and wheeled robots.

Multiple actuated joints give them superior ability to flex, reach, and approach a huge volume in its workspace with an infinite number of configurations. This redundancy in configurations gives them the technical name: hyper redundant robots. In terms of application, snake robots can be used for defense and surveillance and find their place in many places and terrains such as forests, trees, deserts and water.

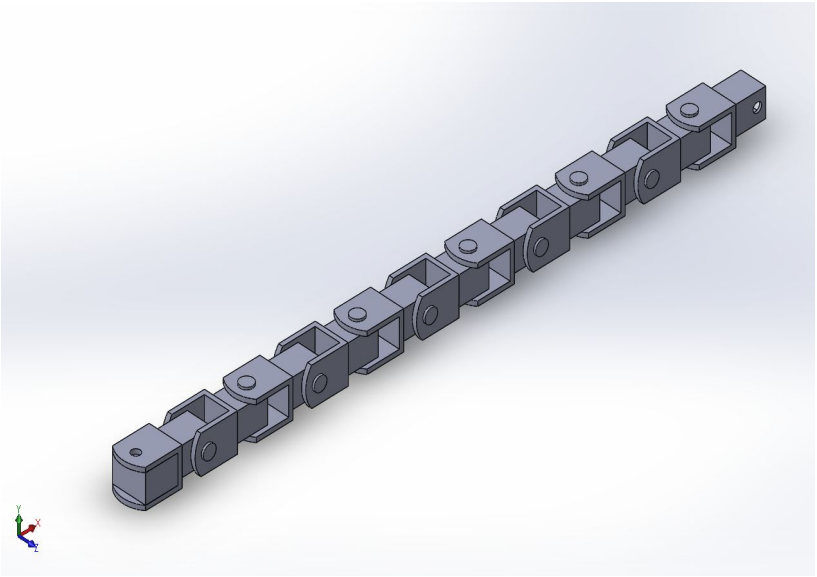
A much more compact design can make them available to be used in the medical domain too. This makes snake robots versatile, achieving movements and gaits not just limited to crawling, climbing and swimming.

ROBOT DESIGN AND DESCRIPTION

We propose a prototype that can basically perform various types of snake movements.
→ Our robot can be made up of 10 links which are connected with the help of a rod in between them.
→ Among them half of the modules can move in vertical direction and remaining in horizontal direction.
→ Sensors: The ‘head’ link can onboard a camera and/or a lidar sensor.
→ Material:

- Lightweight and good tensile strength are the primary criteria for the selection of material of the link.
- Few materials that can be tried are: Aluminum, Nylon and Delrin (polyoxymethylene plastic. The density of delrin is 1.41 g/cm^3 (almost half of that of aluminum).
- The material used should also be able to provide necessary friction for proper movement of the robot.

The proposed snake robot with 11 links has a set of two links from which one is rotated by 90 and are interconnected through a rod. Purpose of the rod is just to interlock the facility and provide necessary movement. The base link or the head of the robot contains a small box on which the camera can be mounted. There are 10 motors which can be placed between the gaps in the links. The joints consist of Fixed joints and Revolute joints which are assigned alternatively beginning with a Revolute joint. They are placed alternatively so that we can have our desired movements. The joints are controlled with electric motors.
The isometric view of the snake robot and the links are shown below.

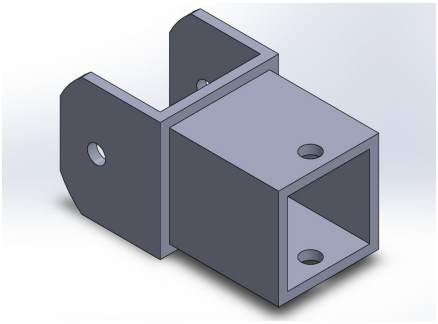


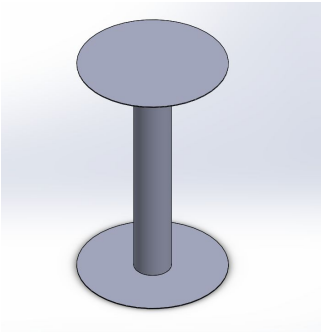
Complete Model

Robot Parameters are as follow :-

Link and rod for the model is shown below

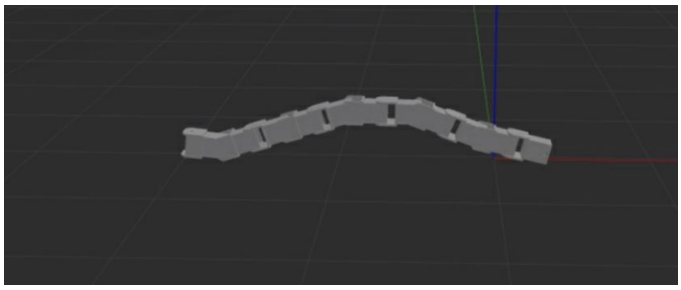
Parameter	Value
Link total length	11 inch
Link front width	5.5 inch
Rod disc diameter	4 inch
Rod disc thickness	0.02 inch





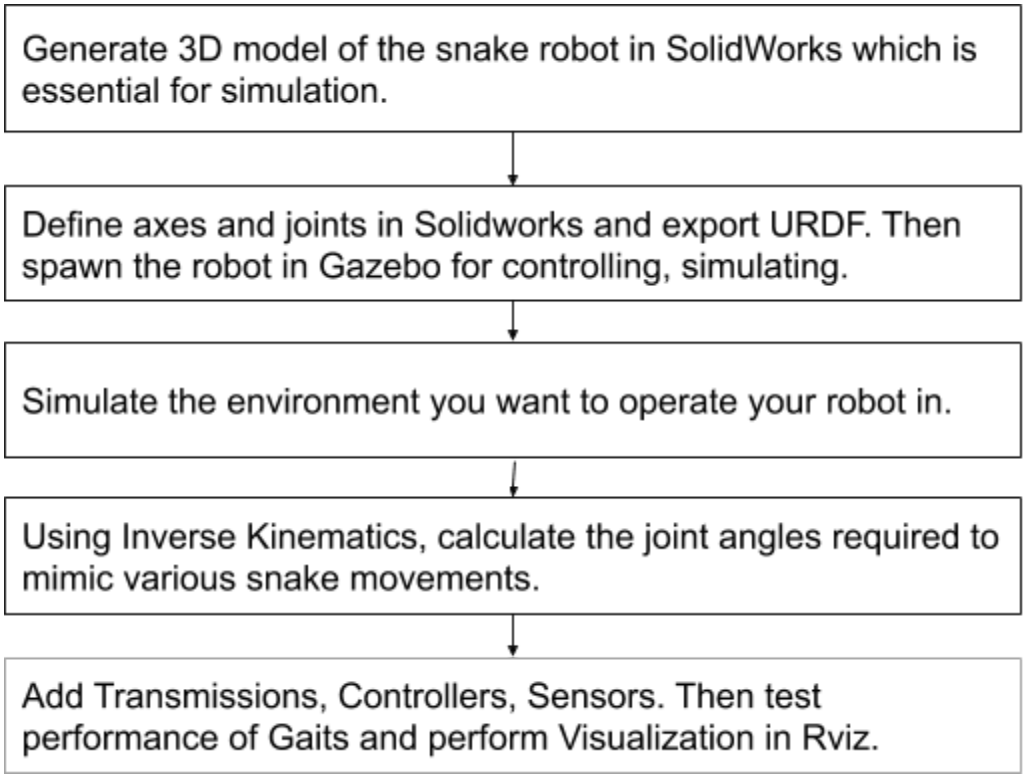
Appropriateness

This robot design is appropriate since snake robots are highly articulated and robust that can pass through compact places where legged or wheeled robots cannot reach. The design we are going to implement is similar to the assembly of links which are positioned in alternate horizontal and vertical orientation which will allow the movement of some links in horizontal and that of some in vertical direction, giving a sinusoidal movement in both the directions. Each mode of the robot can be controlled by a sinusoidal oscillator with four parameters: amplitude, frequency, phase, and offset. This movement will simulate a snake-like motion and hence achieving one of our goals.



Scope of Study

Below is a complete list of concepts we have studied. They are mentioned in the same order as we have followed to accomplish this task:



Assumptions

We consider the following assumptions while designing our robot and simulating it in the environment :

1. All links are rigid, and joints are considered ideal.
2. All the even numbered links and joints possess the same geometrical and inertial Properties.
3. All the odd numbered links and joints possess the same geometrical and inertial Properties.
4. There is no friction between joints.
5. There is enough friction between robot and any surface to make the bot move and perform any motion.
6. There is no backlash in motors.
7. Quasi-static motion.
8. Motors have access to any required torque.
9. Velocity constraints and dynamic properties are not considered.
10. Robots are free of defects at any point of time.
11. All joints have only a single degree-of-freedom.
12. Power loss/consumption is not taken into account.

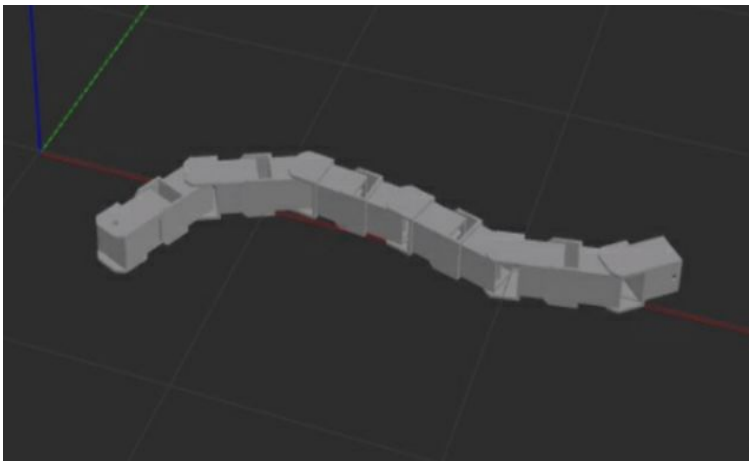
GAITS

Gait is the pattern of movement of the limbs of animals and/or humans during locomotion. It refers to propulsion across a solid substrate by generating reactive forces against it. In our scenario,in order to control the snake's many degrees of freedom we have implemented gaits. These gaits rely on pre-defined undulations that are passed through the length of the snake and we have used parameterized sine waves. Our snake robot is a combination of joints which are alternatively oriented in lateral and dorsal planes. The skeleton of this project is that the gaits consists of separate parameterized sine waves that propagate through the lateral and dorsal joints.

In this project, we have implemented various snake terrestrial locomotion :

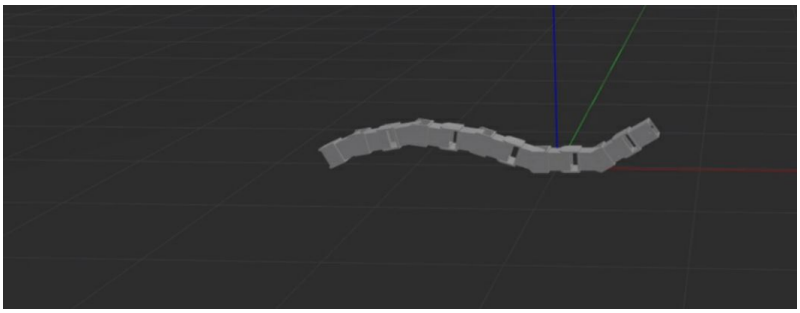
Lateral Undulation

This movement is the common serpentine locomotion. Here, the waves of lateral bending are propagated along the body from head to tail. As the snake moves forward, each point along its body follows along the path established by the head and neck. Here, dorsal are activated sequentially throughout the body.



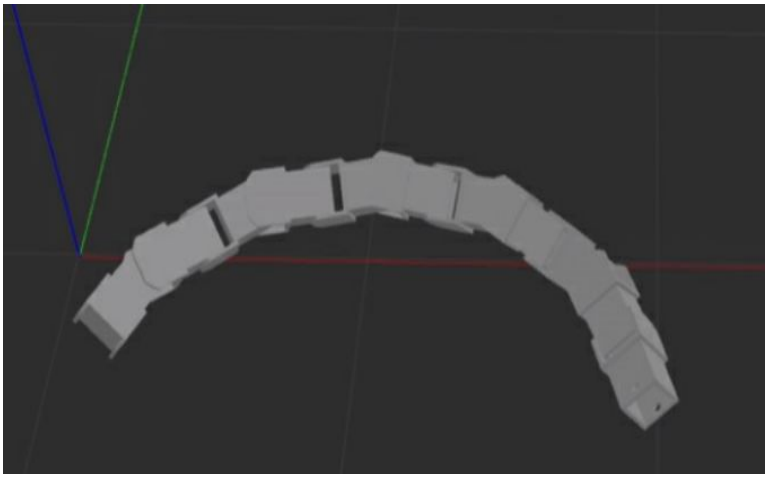
Linear Progression

Also called as simple forward motion. In this movement, the snake doesn't bend its body, in fact the snake flexes the body. In this type of locomotion, sine waves are passed through the length of the robot which makes it propel in forward or backward direction. This movement is usually used when the robot needs to fit in tight shapes[2].



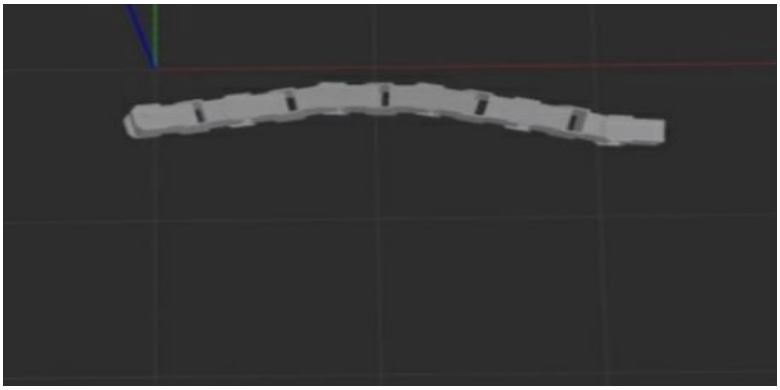
Rolling

This movement makes the robot move in a sideway direction. First the snake curves lightly in ‘C’ shape then rolls sideways. Here, the snake irregularly bends the body and the tail presses vertically on the surface at different points and when the body slips on the surface, it pushes down with enough force to move the center of mass in a quasi-regular pattern [3].



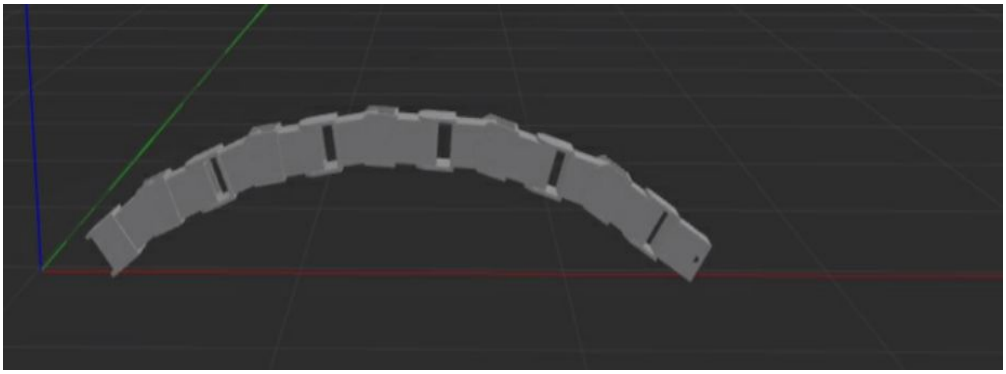
Sidewinding

Lateral undulation will show a shape similar to sine waves with straight parts having negative and positive slope. In sidewinding, undulation is done both laterally and vertically and the head is perpendicular to the direction of movement[5].



Caterpillar Motion

This is also called Rectilinear locomotion. It's a movement in a straight line.Here, the belly scales are alternately lifted slightly from the ground and pulled forward, and then pulled downward and backward[3]. Since, the body remains in touch with ground,thus the body is pulled forward and the cycle repeats.



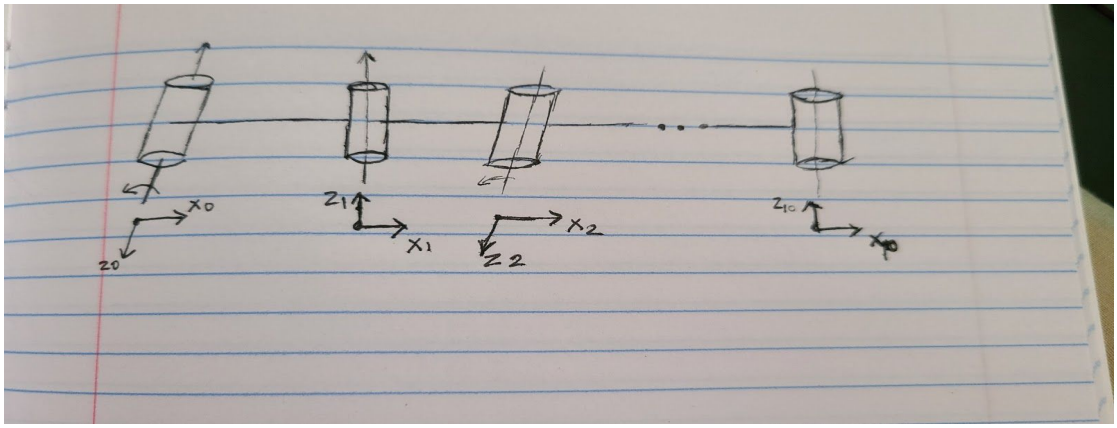
Corkscrewing

A type of gait movement in which the snake robot is in a spiral form of its body, and when these spirals are propagated back through its body, the snake propels forward. This motion is particularly for traveling forwards or backwards in the presence of obstacles, when linear progression cannot easily propel the robot.

Various movements in detail are shown through projects ahead in the report.

Forward and Inverse Kinematics:

- **DH Parameter table:**



Frames

	θ_n	α_n	d_n	r_n
0 - 1	θ	$\Pi/2$	0	l_1
1-2	θ	$\Pi/2$	0	l_2
2-3	θ	$\Pi/2$	0	l_3
3-4	θ	$\Pi/2$	0	l_4
4-5	θ	$\Pi/2$	0	l_5
5-6	θ	$\Pi/2$	0	l_6
6-7	θ	$\Pi/2$	0	l_7
7-8	θ	$\Pi/2$	0	l_8
8-9	θ	$\Pi/2$	0	l_9
9-10	θ	$\Pi/2$	0	l_{10}
10-11	θ	$\Pi/2$	0	l_{11}

- The DH parameter is quite simple. From the figure, the adjacent link is rotated by 90 degrees and the alternate links have the same orientation. Thus,the DH table is generic.

The equations for the forward kinematics are given by:

$$ {}^{n-1}T_n = \begin{pmatrix} \cos\theta_n & -\sin\theta_n \cos\alpha_n & \sin\theta_n \sin\alpha_n & a_n \cos\theta_n \\ \sin\theta_n & \cos\theta_n \cos\alpha_n & -\cos\theta_n \sin\alpha_n & a_n \sin\theta_n \\ 0 & \sin\alpha_n & \cos\alpha_n & d_n \\ 0 & 0 & 0 & 1 \end{pmatrix} $$

$$ \text{and , } {}^{0}T_{11} = \prod_{n=1}^{11} T_n^{n-1} $$

where, θ_n, α_n, d_n and a_n are obtained from the DH table.
We can assume the tail of the bot to be the base-link and the head to be the end-effector. This way we can get the position and the orientation of the head wrt to the base frame using the transformation matrix obtained from the DH table.

Inverse kinematics and Snake gait design:

- Being a hyper-redundant robot, there can be infinitely many solutions for the joint angles to achieve different locomotion gaits.
- So to mimic the snake like motion we implemented gaits based on sinusoidal curves.
- A periodic set of joint angles were given to the model to perform various types of locomotions.
- The joint angles for any n^{th} motor/joint at time-step t are given by:

$$ angle(n,t) = \begin{cases} A_x * \sin(\omega_x t + n * \delta_x), where\ n = even \\ A_y * \sin(\omega_y t + n * \delta_y + \emptyset), where\ n = odd \end{cases} $$

  where,
  - n = motor/actuator/joint number/id,
  - A_x, A_y = amplitude in horizontal and vertical planes respectively,
  - ω_x, ω_y = temporal frequencies,
  - δ_x, δ_y = spatial frequencies,
  - ϕ = phase displacement between horizontal and vertical waves.
- A_x, A_y control the extent of motion (displacement) of the links in the horizontal and vertical planes respectively,
- Temporal frequencies control the speed of the motion of the bot. The higher the value the more distance the bot will be able to cover in a given amount of time.
- Spatial frequencies control the phase difference between each link.
- With a combination of these values the snake bot is allowed to have different periodic displacements along the planes at varying speeds due to which the links are able to

exert a force on the ground to push the ground and move forward due to the friction between the surfaces.
- The different combinations of these values to achieve various gait are discussed in the further section.

SOFTWARE TOOLS FOR SIMULATION

We primarily used three softwares to achieve the task of Modelling, Simulation and Visualization for the proposed Snake Robot. For the same, we used Solidworks, Gazebo and Rviz respectively. Gazebo is preferred over Vrep, mainly due to its ability to accurately and efficiently simulate in complex indoor and outdoor environments with high-quality graphics, and convenient programmatic and graphical interfaces. Rviz is used for visualization purposes.

Validation and Results:

- Once the model was exported from Solidworks, we checked the correctness of the URDF file before using that in our XACRO file.
- Various scripts corresponding to various gaits were written to publish the joint angles to the snake bot.
- The motion was observed and compared with the expected snake-like and non snake-like motions and hence, perceptually validated.
- For launching the simulation in gazebo, the following configurations were done:
    - Modified the **snakebot.urdf**:
        - for adding a dummy link and joint for the world frame and the base link,
        - to set the effort (set to 1000) and velocity limits (set to 0.1),
        - To set the limits of the joint angles (set to -3.14 to 3.14 rad)
        - to modify the moment of inertia values if the bot is unstable while launching in gazebo.
    - Created a **snakebot.xacro** file:
        - to import the main snakebot.urdf file,
        - to add transmissions for the joint actuators,
        - to add a ros-control plugin.
    - Defined joint controller configurations in **joint_controllers.yaml** file
    - Created a launch file to spawn:
        - the joint controller nodes,
        - the urdf model in gazebo and rviz,
        - the joint state publisher.

The configuration files are as follows:

- Effort, velocity and joint angle limits in the snakebot.urdf file. Same values were set for all the revolute joints.

```
<joint
  name="rod_one_joint"
  type="revolute">
  <origin
    xyz="0.2159 0 0"
    rpy="3.1416 0 0" />
  <parent
    link="base_link" />
  <child
    link="rod_one" />
  <axis
    xyz="0 1 0" />
  <limit
    lower="-3.14"
    upper="3.14"
    effort="1000"
    velocity="0.1" />
</joint>
```

- Moment of inertia and mass in snakebot.urdf file:

```
<link
  name="link_one">
  <inertial>
    <origin
      xyz="0.108134816869862 2.4999999991976E-07 0.03175025"
      rpy="0 0 0" />
    <mass
      value="0.372498080839688" />
    <inertia
      ixx="0.40127571298329475"
      ixy="-8.22466335437065E-19"
      ixz="1.64509583289318E-19"
      iyy="0.40265498440151005"
      iyz="9.61291128826174E-22"
      izz="0.20234553145484644" />
  </inertial>
```

- **Snakebot.xacro**

```
<?xml version="1.0"?>
<!-- Name your robot here -->
<robot name ="snakebot" xmlns:xacro="http://wiki.ros.org/xacro">

    <xacro:include filename="$(find snakebot)/urdf/snakebot.urdf"/>

    <!-- Add your gazebo sensors here -->
    <xacro:macro name="joint_transmission" params="joint_num">
        <transmission name="joint_{joint_num}_trans">
            <type>transmission_interface/SimpleTransmission</type>
            <joint name="rod_${joint_num}_joint">
                <hardwareInterface>hardware_interface/EffortJointInterface</hardwareInterface>
            </joint>
            <actuator name="motor_${joint_num}">
                <hardwareInterface>hardware_interface/EffortJointInterface</hardwareInterface>
                <mechanicalReduction>1</mechanicalReduction>
            </actuator>
        </transmission>
    </xacro:macro>


    <joint_transmission joint_num="one"/>
    <!--<joint_transmission joint_num="two"/> -->
    <joint_transmission joint_num="three"/>
    <joint_transmission joint_num="four"/>
    <joint_transmission joint_num="five"/>
    <joint_transmission joint_num="six"/>
    <joint_transmission joint_num="seven"/>
    <joint_transmission joint_num="eight"/>
    <joint_transmission joint_num="nine"/>
    <joint_transmission joint_num="ten"/>
    <joint_transmission joint_num="eleven"/>

    <!--  Gazebo plugin for control here is already added for you -->
    <gazebo>
        <plugin name="gazebo_ros_control" filename="libgazebo_ros_control.so">
            <robotNamespace>/snakebot</robotNamespace>
            <robotSimType>gazebo_ros_control/DefaultRobotHWSim</robotSimType>
            <legacyModeNS>true</legacyModeNS>
        </plugin>

        <plugin name="imu_plugin" filename="libgazebo_ros_imu.so">
            <alwaysOn>true</alwaysOn>
            <bodyName>base_link</bodyName>
            <topicName>imu</topicName>
            <serviceName>imu_service</serviceName>
            <gaussianNoise>0.0</gaussianNoise>
            <updateRate>20.0</updateRate>
        </plugin>
    </gazebo>

</robot>
```

- **joint_controller.yaml:**

```yaml
snakebot:
    joint_state_controller:
        type: joint_state_controller/JointStateController
        publish_rate: 20

    joint_1_position_controller:
        type: effort_controllers/JointPositionController
        joint: rod_one_joint
        pid: {p: 2000.0, i: 50, d: 100.0}
        #pid: {p: 100.0, i: 30.0, d: 40.0}

    joint_2_position_controller:
        type: effort_controllers/JointPositionController
        joint: rod_three_joint #rod_two_joint   #rod_joint_two is not missing. typo in naming second joint. second joint named rod_three_joint
        pid: {p: 2000.0, i: 50, d: 100.0}
        #pid: {p: 100.0, i: 30.0, d: 40.0}

    joint_3_position_controller:
        type: effort_controllers/JointPositionController
        joint: rod_four_joint #rod_three_joint
        pid: {p: 2000.0, i: 50, d: 100.0}
        #pid: {p: 2000.0, i: 50, d: 100.0}

    joint_4_position_controller:
        type: effort_controllers/JointPositionController
        joint: rod_five_joint #rod_four_joint
        pid: {p: 2000.0, i: 50, d: 100.0}
        #pid: {p: 100.0, i: 30.0, d: 40.0}

    joint_5_position_controller:
        type: effort_controllers/JointPositionController
        joint: rod_six_joint #rod_five_joint
        pid: {p: 2000.0, i: 50, d: 100.0}
        #pid: {p: 100.0, i: 30.0, d: 40.0}

    joint_6_position_controller:
        type: effort_controllers/JointPositionController
        joint: rod_seven_joint #rod_six_joint
        pid: {p: 2000.0, i: 50, d: 100.0}
        #pid: {p: 100.0, i: 30.0, d: 40.0}

    joint_7_position_controller:
        type: effort_controllers/JointPositionController
        joint: rod_eight_joint #rod_seven_joint
        pid: {p: 2000.0, i: 50, d: 100.0}
        #pid: {p: 100.0, i: 30.0, d: 40.0}

    joint_8_position_controller:
        type: effort_controllers/JointPositionController
        joint: rod_nine_joint #rod_eight_joint
        pid: {p: 2000.0, i: 50, d: 100.0}
        #pid: {p: 100.0, i: 30.0, d: 40.0}

    joint_9_position_controller:
        type: effort_controllers/JointPositionController
        joint: rod_ten_joint #rod_nine_joint
        pid: {p: 2000.0, i: 50, d: 100.0}
        #pid: {p: 100.0, i: 30.0, d: 40.0}

    joint_10_position_controller:
        type: effort_controllers/JointPositionController
        joint: rod_eleven_joint #rod_ten_joint
        pid: {p: 2000.0, i: 50, d: 100.0}
        #pid: {p: 100.0, i: 30.0, d: 40.0}
```

- **snakebot.launch**:

```
<launch>

  <include file="$(find gazebo_ros)/launch/empty_world.launch">
    <!--<arg name="world_name" value="$(find snakebot)/worlds/competition_arena.world"/>-->
  </include>

  <param name= "robot_description" command="$(find xacro)/xacro --inorder $(find snakebot)/urdf/snakebot.xacro "/>

  <rosparam file="$(find snakebot)/config/joint_controller.yaml" command="load"/>

  <node name="spawn_model" pkg="gazebo_ros" type="spawn_model" args="-urdf -param robot_description -model snakebot" output="screen"/>

  <!-- Controller spawner: Starts all the defined controllers with their configs.
       Just add your controller names defined in your config file into the args tag below   -->
  <node name="controller_spwaner" pkg="controller_manager" type="spawner" respawn="false"
    output="screen" ns="/snakebot" args="joint_state_controller
                                         joint_1_position_controller
                                         joint_2_position_controller
                                         joint_3_position_controller
                                         joint_4_position_controller
                                         joint_5_position_controller
                                         joint_6_position_controller
                                         joint_7_position_controller
                                         joint_8_position_controller
                                         joint_9_position_controller
                                         joint_10_position_controller"/>

  <node name="tf_footprint_base" pkg="tf" type="static_transform_publisher" args="0 0 0 0 0 0 base_link base_footprint 40" />

  <node name="joint_state_publisher" pkg="joint_state_publisher" type="joint_state_publisher">
  </node>

  <node name="robot_state_publisher" pkg="robot_state_publisher" type="robot_state_publisher" />

  <node name="rviz" pkg="rviz" type="rviz" args="-d $(find snakebot)/urdf/snakebot.rviz" />

  <!-- <node name="rqt_publisher" pkg="rqt_publisher" type="rqt_publisher" /> -->

</launch>
```

The following section shows how the scripts are implemented to perform the various gaits.

1) ***Publisher for publishing the joint angles:***

```
def get_publisher_nodes(num_joints):
    joint_publishers = []
    for i in range(num_joints):
        pub_node_name = '/snakebot/joint_{}_position_controller/command'.format(i+1)
        pub_node = rospy.Publisher(pub_node_name, Float64, queue_size=1000)
        joint_publishers.append(pub_node)

    return joint_publishers


if __name__=="__main__":

    rospy.init_node('snakebot_linear_progression', anonymous=True)

    num_joints = 10
    joint_publishers = get_publisher_nodes(num_joints=num_joints)
    gait = LinearProgessionGait(publishers=joint_publishers, num_joints=num_joints)


    rate = rospy.Rate(10)
    time_begin = rospy.Time.now()
    while not rospy.is_shutdown():
        time = (rospy.Time.now() - time_begin).to_sec()
        gait.update_and_publish_angles(t=time)

        #rospy.spin()
        rate.sleep()
```

This is a publisher for forward motion of the snakebot.

1) Initialize a ros node.
2) Initialize the ten publisher nodes corresponding to the ten actuators.
3) Instantiate the particular gait class (here, LinearProgressionGait). One can replace this with any other class instance (RollingGait, CaterpillarGait, and so on). We will be discussing such classes further in this section.
4) Calculate the elapsed time and feed it to the method to update the joint angles.

2) <u>Gaits:</u>

The scripts for the gaits calculated the joint angles as per the sine equation discussed above.
By having different combinations of the amplitudes, frequencies and the phases, we achieve different motions.
A base gait class was defined as the abstract class that was implemented by the concrete gait classes. The abstract methods that each class need to implement is the **get_angle_params()** and **update_and_publish_angles()**

Gait.py

```
class Gait:

    def get_angle_params(self, num_joints):
        pass

    def update_and_publish_angles(self, t):
        pass
```

Concrete gait classes: For each concrete gait implementation classes we define the following parameters:

- **a_hor** - amplitude in horizontal plane (A_x)
- **a_ver** - amplitude in vertical plane (A_y)
- **w_hor**, **w_ver** - temporal frequencies (ω_x, ω_y)
- **ph_hor**, **ph_ver** - spatial frequencies (δ_x, δ_y)
- **axis_lag_hor**, **axis_lag_ver** - Φ i.e the lag between horizontal and vertical plane sine waves

1) ***Linear Progression:***

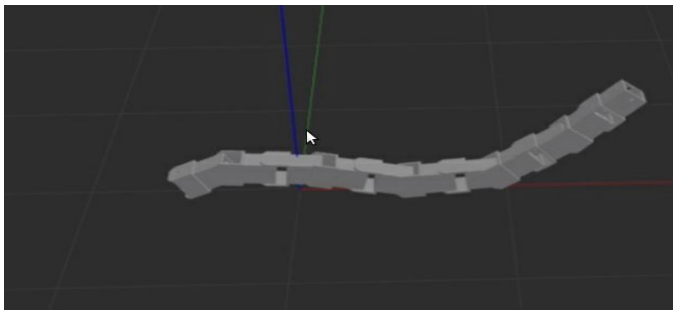
```
class LinearProgessionGait(Gait):
    def __init__(self, publishers, num_joints=10):
        self.num_joints = num_joints
        self.amplitudes, self.omegas, self.phases, self.axis_lags = self.get_angle_params(num_joints)
        self.publishers = publishers

    def get_angle_params(self, num_joints):
        amp_reduction_factor_hor = 0.6 #1.5 #0.4
        amp_reduction_factor_ver = 0.6 #1.5 #0.1

        a_hor = rad(0) * amp_reduction_factor_hor
        a_ver = rad(60) * amp_reduction_factor_ver
        amplitudes = [a_hor, a_ver]

        w_hor = rad(150)
        w_ver = rad(150)
        omegas = [w_hor, w_ver]

        ph_hor = rad(0)
        ph_ver = rad(120)
        phases = [ph_hor, ph_ver]

        axis_lag_hor = rad(0)
        axis_lag_ver = rad(0)
        axis_lags = [axis_lag_hor, axis_lag_ver]

        return amplitudes, omegas, phases, axis_lags


    def update_and_publish_angles(self, t):

        for i in range(self.num_joints):
            n = i+1
            a = self.amplitudes[n % 2]
            w = self.omegas[n % 2]
            ph = self.phases[n % 2]
            lag = self.axis_lags[n % 2]

            link_num = i // 2   # to calculate the ith horizontal/vertical link number

            angle = (a * sin(w*t + ph*n + lag))*math.pow(-1, link_num)
            rospy.loginfo('Angle = {} for joint = {}'.format(angle, i+1))
            self.publishers[i].publish(angle)
```

Here,

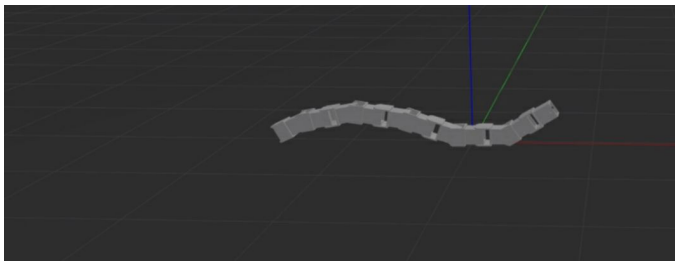
a_hor = 0 rad	a_ver = 60° (π/3 rad)
w_hor = 150° (5π/6 rad)	w_ver = 150° (5π/6 rad)
ph_hor = 0 rad	ph_ver = 120° (2π/3 rad)
axis_lag_hor = 0 rad	axis_lag_ver = 0 rad

We observed that the snakebot was moving forward and the transitions were as follows:

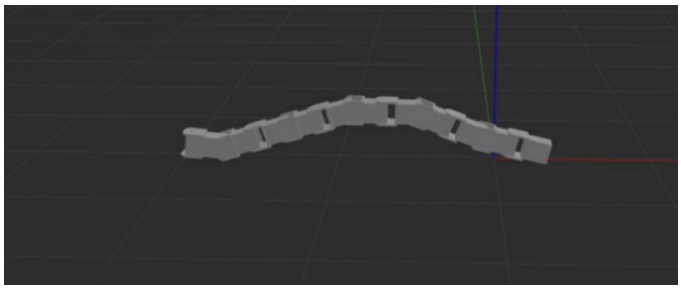
Frame 1:



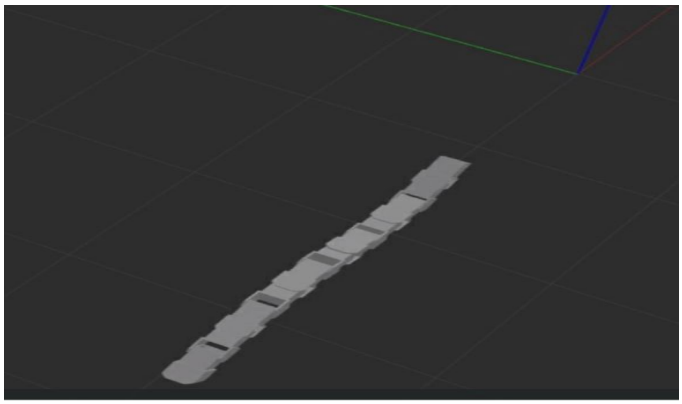
Frame 2:



Frame 3:



Frame 4:



Another thing to observe is that the links are only having vertical displacements and no displacement in the horizontal plane. This conforms with the amplitude values that were given.

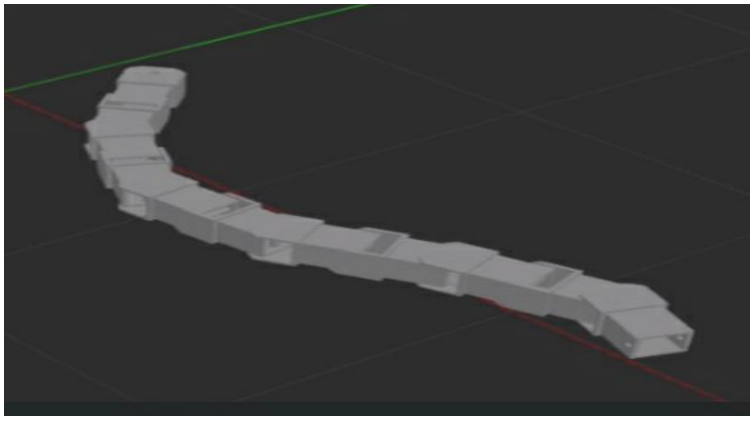
2) ***Lateral Undulation:***

```
class LateralUndulationGait(Gait):
    def __init__(self, publishers, num_joints=10):
        self.num_joints = num_joints
        self.amplitudes, self.omegas, self.phases, self.axis_lags = self.get_angle_params(num_joints)
        self.publishers = publishers

    def get_angle_params(self, num_joints):
        amp_reduction_factor_hor = 0.7 #1.5 #0.4
        amp_reduction_factor_ver = 0.7 #1.5 #0.1

        a_hor = rad(60) * amp_reduction_factor_hor
        a_ver = rad(0) * amp_reduction_factor_ver
        amplitudes = [a_hor, a_ver]

        w_hor = rad(150)
        w_ver = rad(150)
        omegas = [w_hor, w_ver]

        ph_hor = rad(120)
        ph_ver = rad(0)
        phases = [ph_hor, ph_ver]

        axis_lag_hor = rad(0)
        axis_lag_ver = rad(0)
        axis_lags = [axis_lag_hor, axis_lag_ver]

        return amplitudes, omegas, phases, axis_lags


    def update_and_publish_angles(self, t):

        for i in range(self.num_joints):
            n = i+1
            a = self.amplitudes[n % 2]
            w = self.omegas[n % 2]
            ph = self.phases[n % 2]
            lag = self.axis_lags[n % 2]

            link_num = i // 2   # to calculate the ith horizontal/vertical link number

            angle = (a * sin(w*t + ph*n + lag))*math.pow(-1, link_num)
            rospy.loginfo('Angle = {} for joint = {}'.format(angle, i+1))
            self.publishers[i].publish(angle)
```

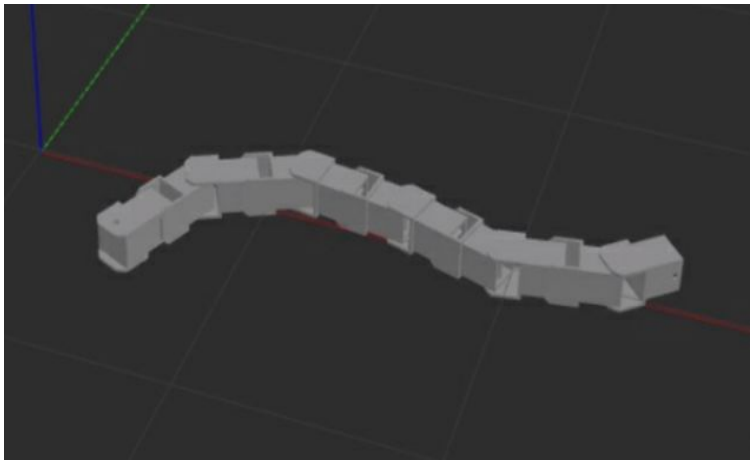
a_hor = 60° (π/3 rad)	a_ver = 0 rad
w_hor = 150° (5π/6 rad)	w_ver = 150° (5π/6 rad)
ph_hor = 120° (2π/3 rad)	ph_ver = 0 rad
axis_lag_hor = 0 rad	axis_lag_ver = 0 rad

The transition frames are:

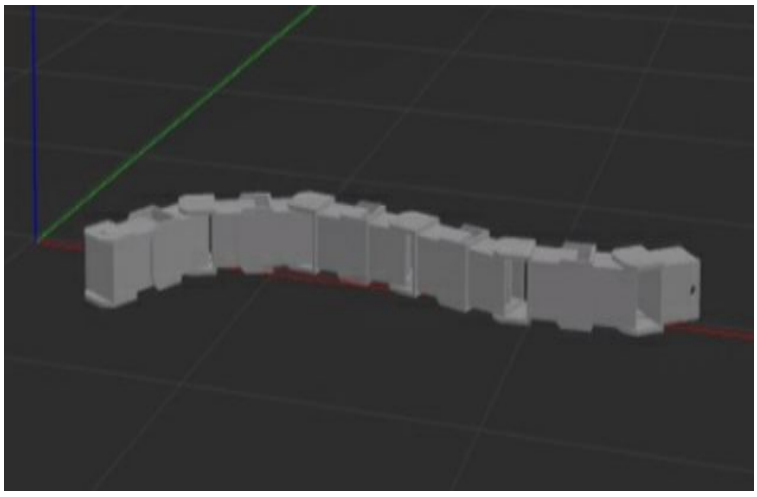
Frame 1:



Frame 2:



Frame 3: No displacement of the links in the vertical plane.



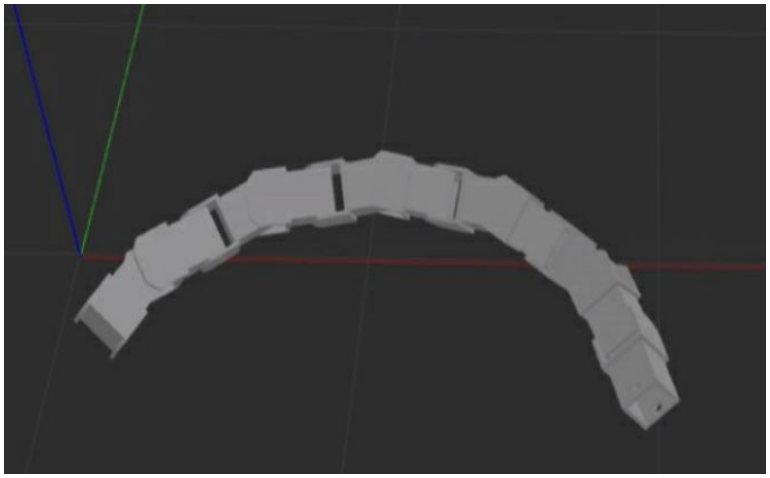
3) ***Rolling:***

```python
sin = math.sin
rad = math.radians
class RollingGait(Gait):
    def __init__(self, publishers, num_joints=10):
        self.num_joints = num_joints
        self.amplitudes, self.omegas, self.phases, self.axis_lags = self.get_angle_params(num_joints)
        self.publishers = publishers

    def get_angle_params(self, num_joints):

        amp_reduction_factor_hor = 0.4 #1.5 #0.4
        amp_reduction_factor_ver = 0.4 #1.5 #0.1

        a_hor = rad(60) * amp_reduction_factor_hor
        a_ver = rad(60) * amp_reduction_factor_ver
        amplitudes = [a_hor, a_ver]

        w_hor = rad(150)
        w_ver = rad(150)
        omegas = [w_hor, w_ver]

        ph_hor = rad(90)
        ph_ver = rad(90)
        phases = [ph_hor, ph_ver]

        axis_lag_hor = rad(0)
        axis_lag_ver = rad(30)
        axis_lags = [axis_lag_hor, axis_lag_ver]

        return amplitudes, omegas, phases, axis_lags


    def update_and_publish_angles(self, t):

        for i in range(self.num_joints):
            n = i+1
            a = self.amplitudes[n % 2]
            w = self.omegas[n % 2]
            ph = self.phases[n % 2]
            lag = self.axis_lags[n % 2]

            link_num = i // 2   # to calculate the ith horizontal/vertical link number

            angle = (a * sin(w*t + ph*n + lag))*math.pow(-1, link_num)
            rospy.loginfo('Angle = {} for joint = {}'.format(angle, i+1))
            self.publishers[i].publish(angle)
```

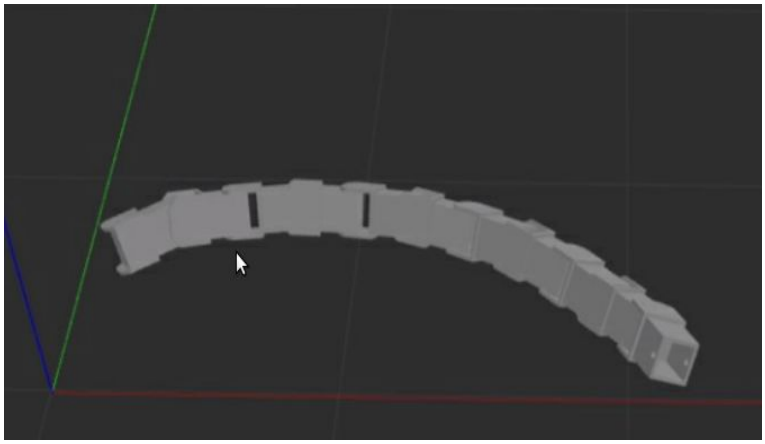
a_hor = 60° (π/3 rad)	a_ver = 60° (π/3 rad)
w_hor = 150° (5π/6 rad)	w_ver = 150° (5π/6 rad)
ph_hor = 90° (π/2 rad)	ph_ver = 90° (π/2 rad)
axis_lag_hor = 0 rad	axis_lag_ver = 30° (π/6 rad)

The bot was observed taking a C-shape and was rolling along its own longitudinal axis.

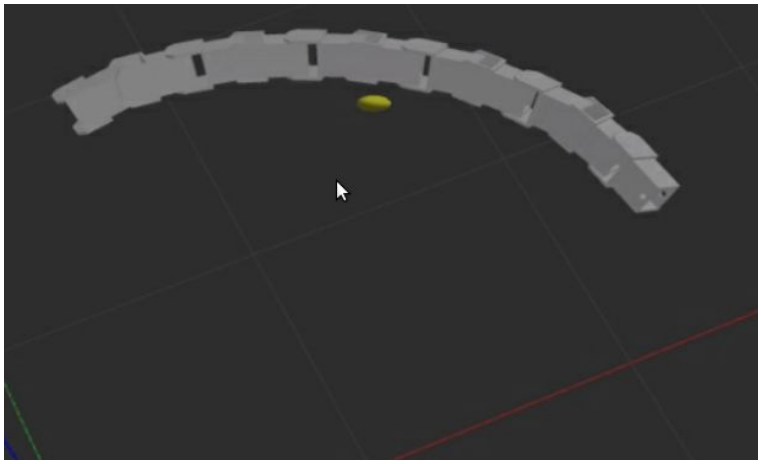
Frame 1: C-shape



Frame 2:



Frame 3:



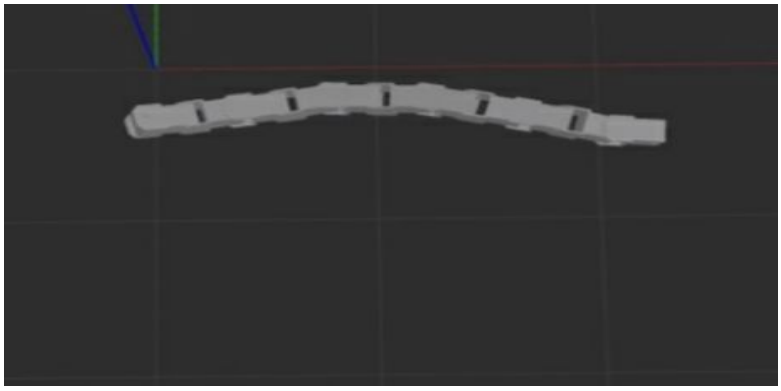
4) ***Sidewinding:***

```python
sin = math.sin
rad = math.radians
class SideWindingGait(Gait):
    def __init__(self, publishers, num_joints=10):
        self.num_joints = num_joints
        self.amplitudes, self.omegas, self.phases, self.axis_lags = self.get_angle_params(num_joints)
        self.publishers = publishers

    def get_angle_params(self, num_joints):

        amp_reduction_factor_hor = 0.5 #1.5 #0.4
        amp_reduction_factor_ver = 0.5 #1.5 #0.1

        a_hor = rad(30) * amp_reduction_factor_hor
        a_ver = rad(30) * amp_reduction_factor_ver
        amplitudes = [a_hor, a_ver]

        w_hor = rad(150)
        w_ver = rad(150)
        omegas = [w_hor, w_ver]

        ph_hor = rad(120)
        ph_ver = rad(120)
        phases = [ph_hor, ph_ver]

        axis_lag_hor = rad(0)
        axis_lag_ver = rad(0)
        axis_lags = [axis_lag_hor, axis_lag_ver]

        return amplitudes, omegas, phases, axis_lags


    def update_and_publish_angles(self, t):

        for i in range(self.num_joints):
            n = i+1
            a = self.amplitudes[n % 2]
            w = self.omegas[n % 2]
            ph = self.phases[n % 2]
            lag = self.axis_lags[n % 2]

            link_num = i // 2   #to calculate the ith horizontal/vertical link number

            angle = (a * sin(w*t + ph*n + lag))*math.pow(-1, link_num)
            rospy.loginfo('Angle = {} for joint = {}'.format(angle, i+1))
            self.publishers[i].publish(angle)
```

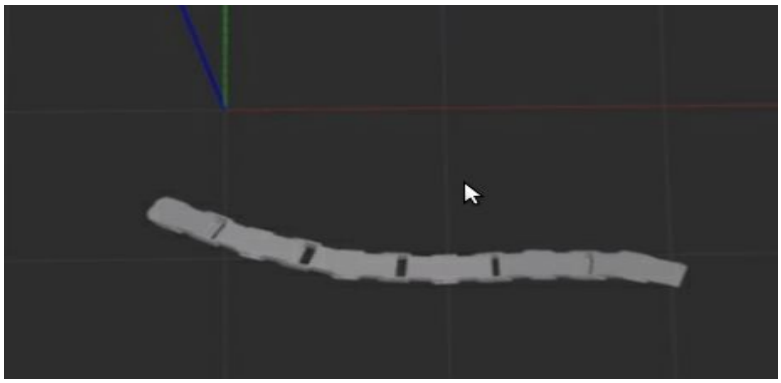
a_hor = 30° (π/6 rad)	a_ver = 30° (π/6 rad)
w_hor = 150° (5π/6 rad)	w_ver = 150° (5π/6 rad)
ph_hor = 120° (2π/3 rad)	ph_ver = 120° (2π/3 rad)
axis_lag_hor = 0 rad	axis_lag_ver = 0 rad

The bot was observed moving towards its side. The forces exerted by links moving towards the left side of the snakebot made it push the ground and moved to its side.

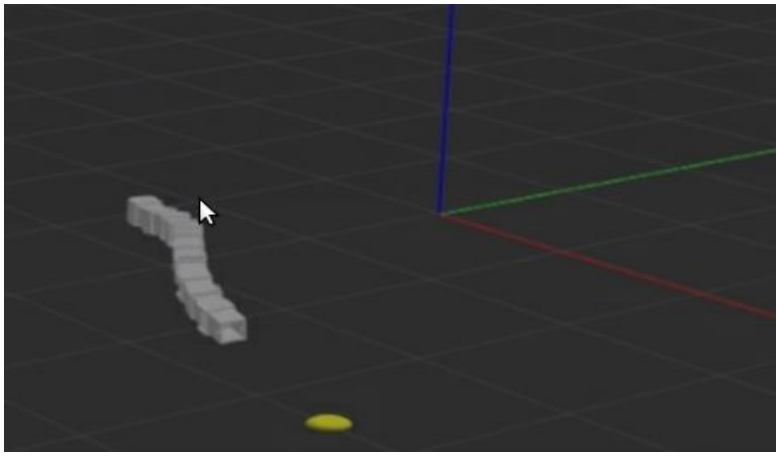
Frame 1:



Frame 2:



Frame 3:



Link displacement both in vertical and horizontal planes.

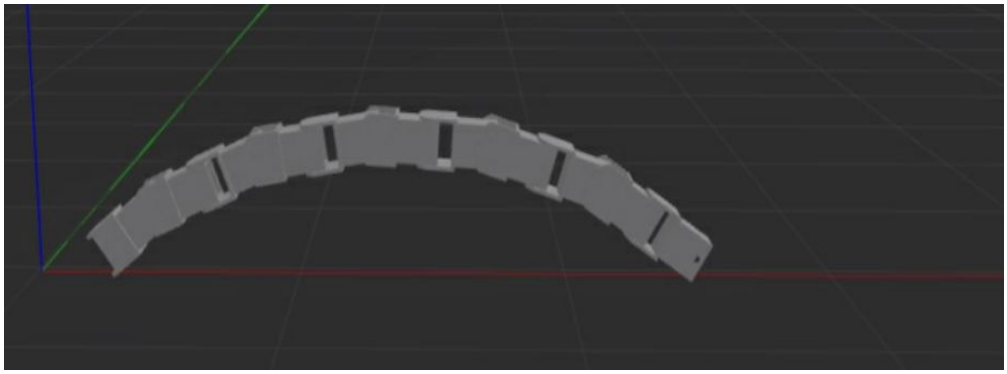
5) ***Caterpillar Motion:***

```
class CaterpillarGait(Gait):
    def __init__(self, publishers, num_joints=10):
        self.num_joints = num_joints
        self.amplitudes, self.omegas, self.phases = self.get_angle_params(num_joints)
        self.publishers = publishers

    def get_angle_params(self, num_joints):
        amp_reduction_factor = 0.3
        a_ver = rad(60) * amp_reduction_factor
        a_hor = rad(0) * amp_reduction_factor
        amplitudes = [a_hor, a_ver]

        w_ver = rad(150)
        w_hor = rad(0)
        w_o = rad(30)
        omegas = [w_hor, w_ver, w_o]

        ph_ver = rad(120)
        ph_hor = rad(0)
        phases = [ph_hor, ph_ver]

        return amplitudes, omegas, phases


    def update_and_publish_angles(self, t):

        for i in range(self.num_joints):
            a = self.amplitudes[(i+1)%2]
            w = self.omegas[(i+1)%2]
            ph = rad(self.phases[(i+1)%2])

            w_o = self.omegas[2]

            angle = min(a*sin(w*t + ph + w_o), 0)
            rospy.loginfo('Angle = {} for joint = {}'.format(angle, i+1))
            self.publishers[i].publish(angle)
```

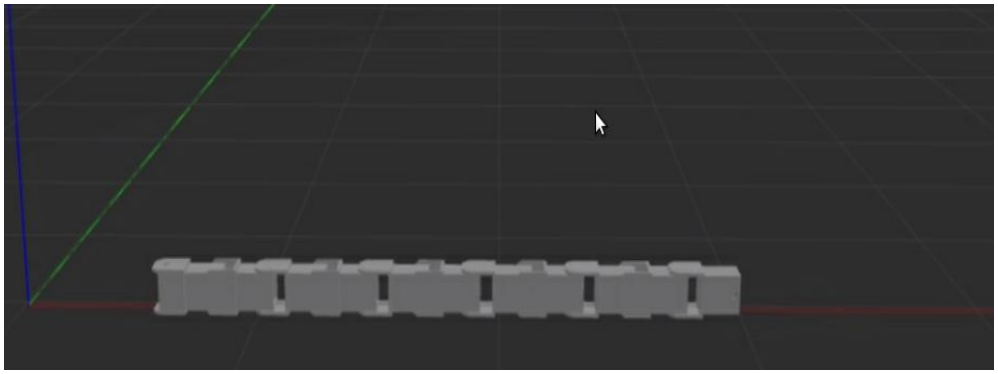
a_hor = 0 rad	a_ver = 60° (π/3 rad)
w_hor = 0 rad	w_ver = 150° (5π/6 rad)
ph_hor = 0 rad	ph_ver = 120° (2π/3 rad)
axis_lag_hor = 0 rad	axis_lag_ver = 30 rad

The bot was observed moving like a caterpillar by shrinking and expanding its body alternately.

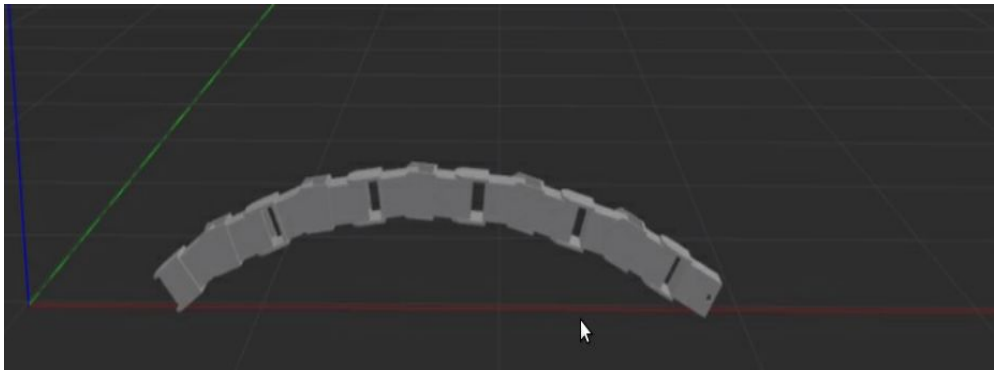
Frame 1:



Frame 2:



Frame 3:



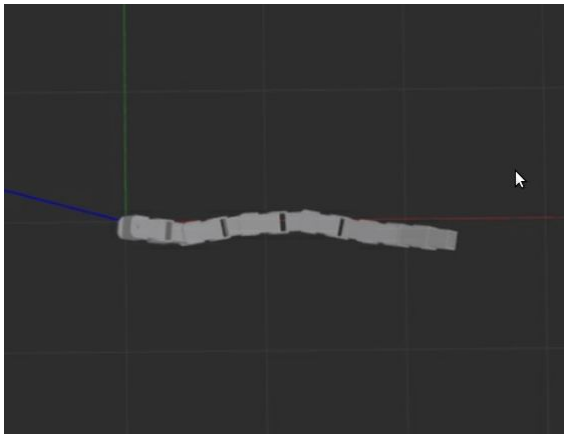
6) ***Rotation along axis:***

```
sin = math.sin
rad = math.radians
class RotationAlongAxis(Gait):
    def __init__(self, publishers, num_joints=10):
        self.num_joints = num_joints
        self.amplitudes, self.omegas, self.phases, self.axis_lags = self.get_angle_params(num_joints)
        self.publishers = publishers

    def get_angle_params(self, num_joints):

        amp_reduction_factor_hor = 0.2 #1.5 #0.4
        amp_reduction_factor_ver = 0.2 #1.5 #0.1

        a_hor = rad(90) * amp_reduction_factor_hor
        a_ver = rad(90) * amp_reduction_factor_ver
        amplitudes = [a_hor, a_ver]

        w_hor = rad(180)
        w_ver = rad(180)
        omegas = [w_hor, w_ver]

        ph_hor = rad(90)
        ph_ver = rad(0)
        phases = [ph_hor, ph_ver]

        axis_lag_hor = rad(0)
        axis_lag_ver = rad(0)
        axis_lags = [axis_lag_hor, axis_lag_ver]

        return amplitudes, omegas, phases, axis_lags


    def update_and_publish_angles(self, t):

        for i in range(self.num_joints):
            n = i+1
            a = self.amplitudes[n % 2]
            w = self.omegas[n % 2]
            ph = self.phases[n % 2]
            lag = self.axis_lags[n % 2]

            link_num = i // 2   # to calculate the ith horizontal/vertical link number

            angle = (a * sin(w*t + ph*link_num + lag))*math.pow(1, link_num)
            rospy.loginfo('Angle = {} for joint = {}'.format(angle, i+1))
            self.publishers[i].publish(angle)
```

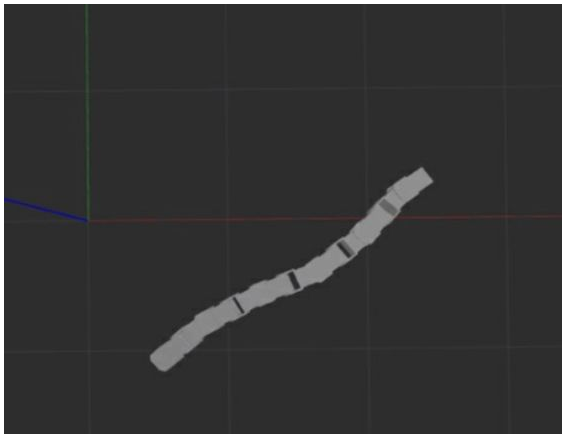
a_hor = 90° (π/2 rad)	a_ver = 90° (π/2 rad)
w_hor = 180° (π rad)	w_ver = 180° (π rad)
ph_hor = 90° (π/2 rad)	ph_ver = 0 rad
axis_lag_hor = 0 rad	axis_lag_ver = 0 rad

The bot was observed moving turning two one of its sides and was rotating about the vertical axis passing through its center.

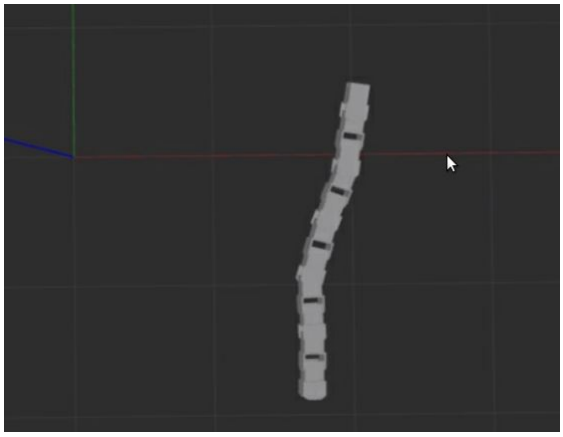
Frame 1:



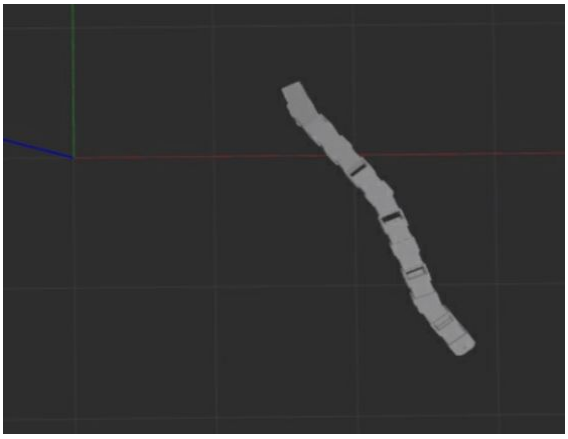
Frame 2:



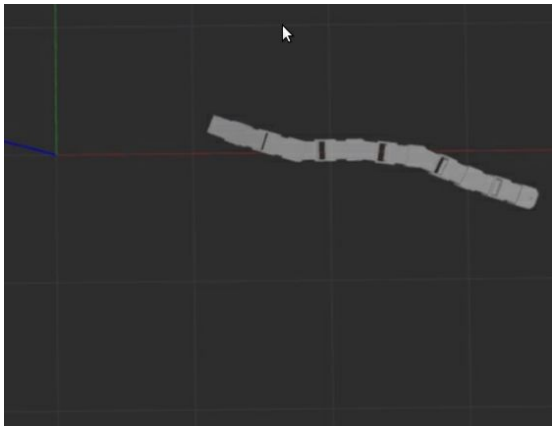
Frame 3:



Frame 4:



Frame 5:



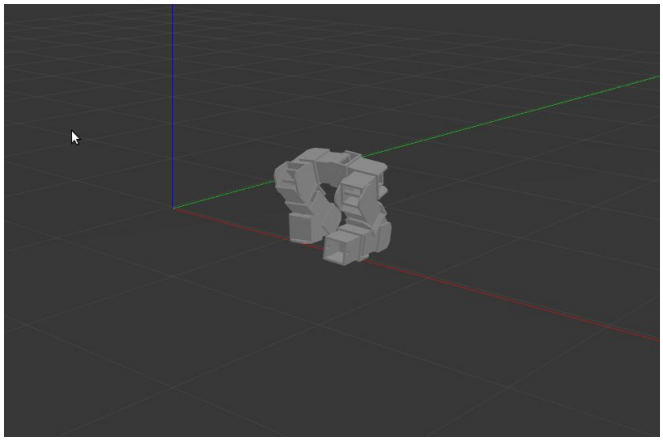
7) ***Corkscrew:***

```python
class CorkScrewGait(Gait):
    def __init__(self, publishers, num_joints=10):
        self.num_joints = num_joints
        self.amplitudes, self.omegas, self.phases, self.axis_lags = self.get_angle_params(num_joints)
        self.publishers = publishers

    def get_angle_params(self, num_joints):

        amp_reduction_factor_hor = 0.3 #1.5 #0.4
        amp_reduction_factor_ver = 0.3 #1.5 #0.1

        a_hor = rad(60) * amp_reduction_factor_hor
        a_ver = rad(30) * amp_reduction_factor_ver
        amplitudes = [a_hor, a_ver]

        w_hor = rad(150)
        w_ver = rad(75)
        omegas = [w_hor, w_ver]

        ph_hor = rad(120)
        ph_ver = rad(120)
        phases = [ph_hor, ph_ver]

        axis_lag_hor = rad(0)
        axis_lag_ver = rad(0)
        axis_lags = [axis_lag_hor, axis_lag_ver]

        return amplitudes, omegas, phases, axis_lags


    def update_and_publish_angles(self, t):

        for i in range(self.num_joints):
            n = i+1
            a = self.amplitudes[n % 2]
            w = self.omegas[n % 2]
            ph = self.phases[n % 2]
            lag = self.axis_lags[n % 2]

            link_num = i // 2   # to calculate the ith horizontal/vertical link number

            angle = (a * sin(w*t + ph*link_num + lag))*math.pow(-1, link_num)
            rospy.loginfo('Angle = {} for joint = {}'.format(angle, i+1))
            self.publishers[i].publish(angle)
```

a_hor = 60° (π/3 rad)	a_ver = 30° (π/6 rad)
w_hor = 150° (5π/6 rad)	w_ver = 75° (5π/12 rad)
ph_hor = 120° (2π/3 rad)	ph_ver =120° (2π/3 rad)
axis_lag_hor = 0 rad	axis_lag_ver = 0 rad

Ambitious Goal Implementation:

We know that this model has high DOF and hence is able to perform more complex and dexterous gaits and motions. With this idea, we tried to **transform** this snake bot model to a **bipedal** robot. This opens the scope for a lot of possibilities of transforming this same model into other various types of shapes to perform appropriate, complex and difficult motions.
The figure below shows the bipedal version of our robot.



Methodology to transform into a bipedal:
The objective is to have a modular mechanism which can transform from a snake configuration into a walking configuration without any detaching and attaching modules.
This can be broken down into two gaits:
1) Transformation gait
2) Walking gait
In this report we have implemented the transformation gait.

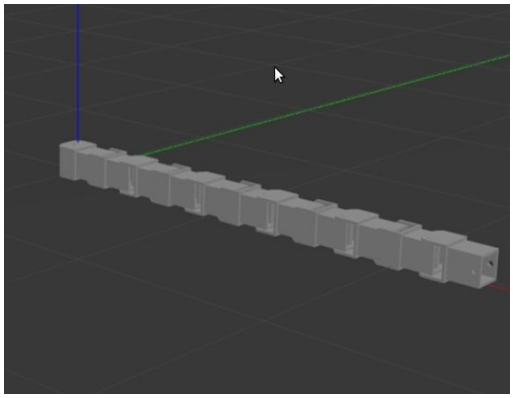
- This transformation was designed using key-frames interpolation based method.
- A key-frame consists of a set joint angles at a particular time-stamp.
- To ensure the stability of the robot the ground projection of the center of mass (CoM) of the robot must lie within the support polygon of the robot.
- Every key-frame must satisfy this criteria
- Also, the criteria must be satisfied while transitioning between the frames.

Keyframe interpolation:

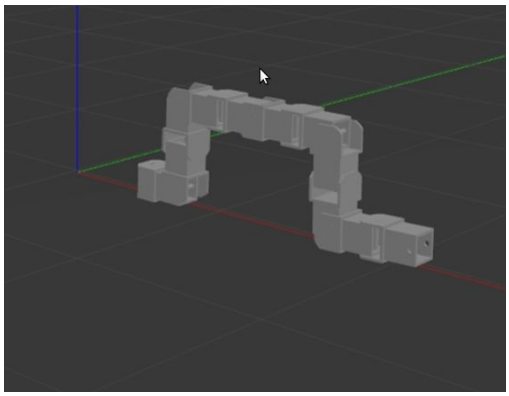
- For key-frame-1, all the joint angles are set to zero and an open chain is formed
- In key-frame-2, the bot is made to take an open rectangular shape followed by a pentagon shape.
- These transformations are in 2D plane and the angles can be calculated using inverse kinematics and the transitions can be made using linear interpolation.
- For further transitions more number of intermediate key-frames were taken to approximate better transition curves.

Key frames:

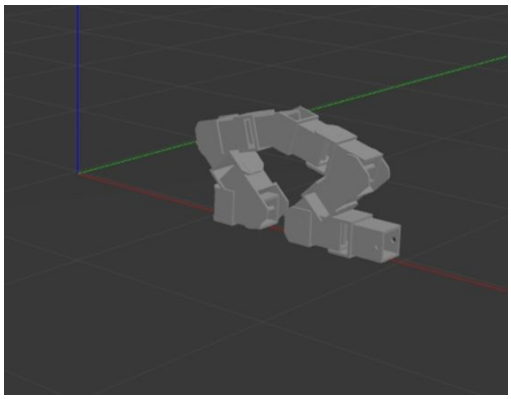
Frame 1:



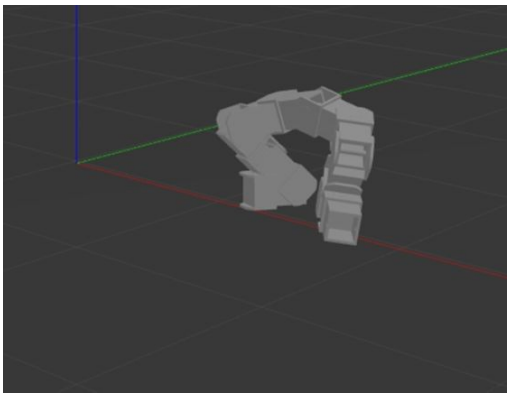
Frame 2:



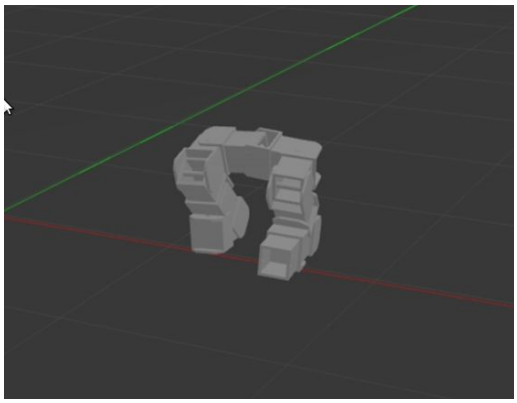
Frame 3:



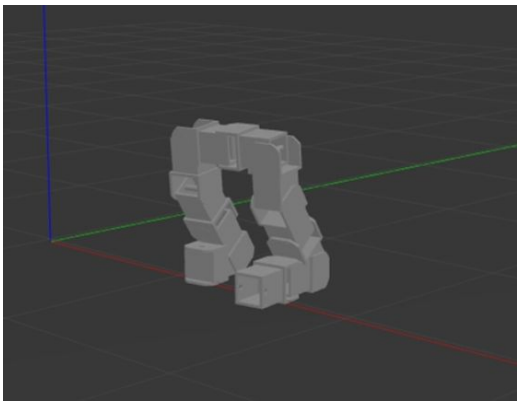
Frame 4:



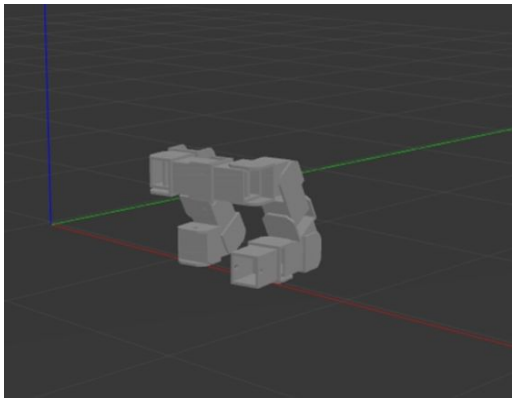
Frame 5:



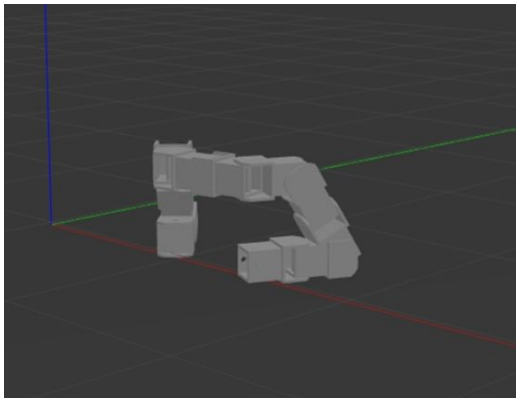
Frame 6:



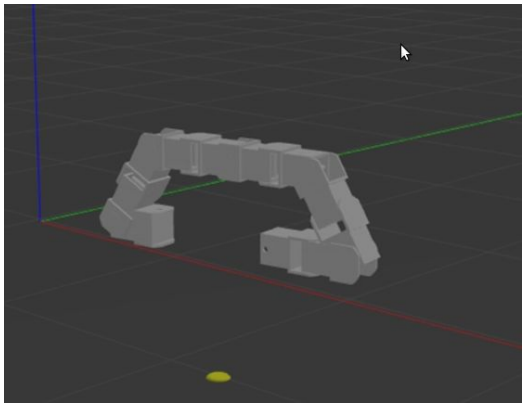
Frame 7:



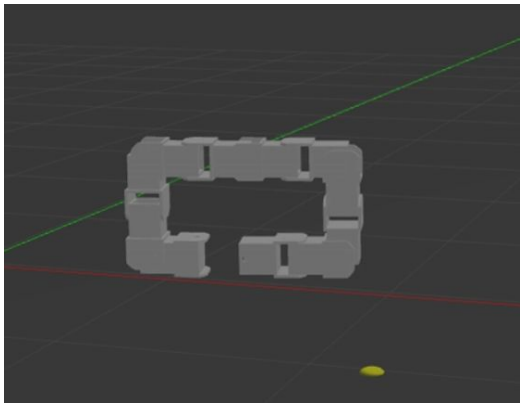
Frame 8:



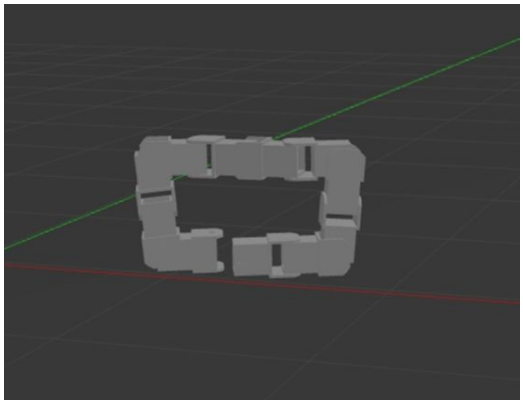
Frame 9:



Frame 10:



Frame 11:



Script for the transformation gait:

```
class TransformerGait(Gait):
    def __init__(self, publishers, num_joints=10):
        self.num_joints = num_joints
        self.publishers = publishers
        self.key_point = 0


    def update_and_publish_angles(self, t):
        # key-frames - set of joint angles as per different time-stamps
        angle_matrix = [
            [90,  0,       -90,    0,    0,     0,  -90,    0,    90,      0],

            [120,  0,      -110,   0,   -30,    0,  -110,   0,    120,     0],
            [135,  0,      -120,   0,   -30,    0,  -120,   0,    135,     0],

            [135,  0,      -120,  -60,  -30,   -60, -120,   0,    135,     0],
            [135,  0,      -120,  -90,  -10,  -90, -120,   0,    135,     0],
            [135,  0,      -120,  -90,   0,    -90, -120,   0,    135,     0],

            ## biped is formed
            ## moving on to make it a closed polygon
            [125,  0,      -80,   -90,   0,    -90,  -80,   0,    125,     0],
            [125,  0,      -60,   -90,   0,    -90,  -60,   0,    125,     0],
            [125,  0,      -20,   -90,   0,    -90,  -20,   0,    125,     0],
            [120,  0,       20,   -90,   0,    -90,   20,   0,    120,     0],
            [120,  0,       40,   -90,   0,    -90,   40,   0,    120,     0],
            [120,  0,       60,   -90,   0,    -90,   60,   0,    120,     0],

            #renctangle shape done
            #over to making the legs more closer
            [120,  0,       60,    0,    0,    0,    60,   0,    120,     0],

            [90,  0,        90,    0,    0,    0,    90,   0,    90,      0],
            [85,  0,        95,    20,   0,   -20,   95,   0,    85,      0]
        ]

        global count  #initial value of count = 0
        if (t >= TRANSIT_TIME*count):  # transition time between two key-frames: t_transit = 4 sec
            count+=1
            self.key_point += 1
            self.key_point = min(len(angle_matrix)-1, self.key_point)
        angles = angle_matrix[self.key_point]

        for i in range(self.num_joints):
            angle = rad(angles[i])
            rospy.loginfo('Angle = {} for joint = {}'.format(angle, i+1))
            self.publishers[i].publish(angle)
```

The ability to transform between snake and multiple walking configurations means that this model has a multitude of locomotion strategies.
For walking gait IK can be calculated using forward kinematics approach.

- Use all the possible ranges of the joint angles to get x,y,z,roll,pitch and yaw and store it in a database(DB).
- For any position and orientation the joint angles can be queried from the DB using the neared pose to joint angle mapping.

Discussions and Conclusions

We believe that, not only the proposed study has successfully achieved various Snake movements but, we achieved our ambitious goal of implementing a Bipedal robot from Snake robot.The model was successfully built on Solidworks and imported as a URDF file into Gazebo environment. For the study, inverse kinematics and forward kinematics is successfully validated. Also, validation for every snake robot movement is achieved.
Link to the video:
All the gaits
Transformer Bipedal

Replicability

The study can be divided into two portions:

1) **Model Creation**
2) **Gazebo Simulation**

- Model Creation: Create a model with same specifications and design with same number of links and joints. To create an assembly, create individual parts of rod, rotated link, link and head link and box. Then create an assembly with a box and rotated link and save this assembly file as a part. In the new assembly file, first import the file which is saved as a part file and this is going to be a base link. Keep on adding all the parts required to build your model. Then with the function of “mate” in Solidworks. Mate all the parts as per the given design. After your mating is completed, use a parallel mate to mate all the alternate links together. By doing so, you avoid chances of your model to be in the air in the Gazebo world. Afterwards, assign a fixed and revolute joint to alternate joints present in the model, starting with a revolute joint in generation of a URDF step. The next step is to export URDF. At this stage,again assign revolute and fixed joints to each and every joints present. The next step is to add a dummy link in the URDF model. The last step is to open the URDF and check the URDF file before spawning into Gazebo.

Add this portion in URDF file below your “ robot name”

```
<link
 name="dummy_link">
</link>
<joint
 name="dummy_joint"
 type="fixed">
 <parent
  link="dummy_link"/>
 <child
  link="base_link"/>
```

Command to check URDF file: check_urdf <filename.urdf>

- Gazebo Simulation:

The following instructions can be followed to run the gazebo simulation.

1) The URDF file imported from Solidworks needs to be modified. The modification steps and the values have been discussed in the validation section.
2) Starting with ROS:
    a) create a workspace and a src folder in it: **$ mkdir snakebot_ws/src**
    b) $ cd snakebot_ws/src
    c) Create a package inside the src folder with the dependencies required:
        i) catkin_create_pkg snakebot roscpp rospy urdf joint_state_publisher joint_state_controller controller_manager tf sensor_msgs std_msgs
        ii) Make sure that the package name mentioned in the urdf file and the one that is being created is the same, else make the appropriate change.
    d) $ source devel/setup.bash
    e) $ roscd snakebot/
    f) Copy the **urdf, meshes, launch, config, textures** folder inside the new package (snakebot)
    g) Create a **snakebot.xacro** file in the urdf folder as discussed in the validation section.
    h) Create a **joint_controller.yaml** file inside the **config** folder
    i) Create **snakebot.launch** file inside the **launch** folder to launch the ros nodes.
    j) Write the **scripts** inside a **scripts** folder to publish the joint angles for simulation. The different scripts and their simulation results have been discussed in the validation and result section.
    k) Add the script name in the CMakelist.txt file inside the snakebot package:
    Example:

```
catkin_install_python(PROGRAMS
  scripts/forward.py
  scripts/transformer.py
  scripts/corkscrew.py
  scripts/caterpillar.py
  scripts/sidewinding.py
  scripts/linear_progression.py
  scripts/rolling.py
  scripts/lateral_undulation.py
  scripts/rotation_along_axis.py
  DESTINATION ${CATKIN_PACKAGE_BIN_DESTINATION}
```

l) Make the scripts publisher scripts executable before running them as nodes: Example:
   - Go to the **scripts** folder.
   - run $ chmod +x linear_progression.py

m) To run the simulation:
   i) Open a new terminal.
   ii) ***$ cd ~/snakebot_ws***
   iii) ***$ catkin_make***
   iv) ***$ source devel/setup.bash***
   v) $ ***roslaunch snakebot snakebot.launch***
   vi) This will spawn the model in gazebo
   vii) Open another terminal.
   viii) ***$ cd ~/snakebot_ws***
   ix) ***$ source devel/setup.bash***
   x) To run any gait (example linear progression)
   $ rosrun snakebot linear_progression.py

Complete code base:
Link to the complete code:
Link to the complete model: https://github.com/Preyash18/Snake-Robot

Scope for Future Work

→ Performing more gaits like:
- Right turn, Left turn
- **Tree climbing** - the links should be light-weighted, more friction between the surface, some feedback system will be required to measure the grasp on the tree.
- **Swimming** - fully enclosed casing required
- **Moving the bipedal** - We did not implement the walking gaits using this model. But there is a scope of extending this model to transform this into a biped. We can assume the foot, in contact with the ground, as the base and the other foot as the manipulator.
- More **dexterous shapes** and motions.

- The motions implemented can be **fine tuned**.
- Mounting **camera/lidar sensors** for surveillance, obstacle avoidance and path planning tasks.

REFERENCES :

1) https://en.wikipedia.org/wiki/Gait
2) http://biorobotics.ri.cmu.edu/research/gaitResearch.php
3) https://userweb.ucs.louisiana.edu/~brm2286/locomotn.htm
4) http://biorobotics.ri.cmu.edu/projects/modsnake/gaits/gaits.html
5) https://en.wikipedia.org/wiki/Sidewinding
6) https://www.ivlabs.in/uploads/1/3/0/6/130645069/rebis.pdf