Wall-e-simulation-ros2

Project Report

EKLAVYA MENTORSHIP PROGRAMME
At
SOCIETY OF ROBOTICS AND AUTOMATION,
VEERMATA JIJABAI TECHNOLOGICAL INSTITUTE
MUMBAI
October 2021

ACKNOWLEDGEMENT

The seniors of SRA VJTI really made this learning experience fun and extremely useful to us. The guidance that we received from our mentors helped us build a project which we couldn't have imagined of doing before this. They not only taught us concepts related to the project but also skills which would help us a long way in our engineering career. We would like to thank **SRA VJTI** for giving us this golden opportunity and our seniors who took the time and efforts to guide us. We would also like to specially thank our mentors **Gautam Agrawal** and **Anushree Sabnis** for guiding us through the entire duration of the project and helping us achieve our goal.

Aryaman Shardul
aryashardul2002@gmail.com
+917506850482

Marck Koothor
marckjk31@gmail.com
+919137828660

:

TABLE OF CONTENTS:

1. PROJECT OVERVIEW

1.1 PROJECT IDEA:

The idea of the project was to design a two wheeled robot ‘WallE’ and implement self-balancing and line-following algorithms on it. We would then simulate the bot in Gazebo.

1.2 TECHNOLOGIES AND TOOLS USED:

TECHNOLOGIES:

1. ROS2 Foxy Fitzroy : ROS (Robotics Operating System) is a collection of software frameworks for robot software development. ROS2 is the newer version of ROS. It aims to address the shortcomings in ROS. Foxy is a major milestone and is the first ROS2 release with a three year term support window.

TOOLS:

1. Gazebo: It is an open source 3-D robotics simulator. Gazebo can use multiple high-performance physics engines, such as Open Dynamics Engine(ODE), Bullet, etc with the default one being ODE. It helps us simulate our robot under different conditions which can be quite handy for testing our robot under different environments. It has many plug-ins like sensor plug-ins, world plug-ins etc which help in enhancing the bot's functionalities and creating conditions nearly as good as the real world. Here ROS2 and Gazebo(version: 11.0) have been used to simulate the WallE bot.

2. RVIZ: It is a 3-D visualization tool. It is highly configurable. Here it is used for visualizing the line the bot has to follow, through the RGB camera attached on the bot.

3. SolidWorks: SolidWorks is a solid modeling computer-aided design (CAD) and computer-aided engineering (CAE) computer program. Here it's used to design the WallE bot.

4. Autocad: AutoCAD is a commercial computer-aided design (CAD) and drafting software application. Here it's used to design the path of the world required in Gazebo.

5. Microsoft Paint: Microsoft Paint is a simple raster graphics editor. The program opens and saves files in Windows bitmap (BMP), JPEG, GIF, PNG, and single-page TIFF formats. Here it's been used to colour the PNG file required for the world in Gazebo.

2. INTRODUCTION

2.1 BASIC PROJECT DOMAINS:

The main domain of the project is simulation. The bot's actions have been simulated in Gazebo. All the different algorithms implemented on it are visualized using a simulator.

2.2 THEORY:

The designing of the robot requires a good amount of knowledge of SolidWorks. The dimensions of the bot have to be precise for it to work properly. The entire process of simulating the bot is heavily dependent on ROS2 and Gazebo. Hence, having a good knowledge of these two is an important requirement. Apart from these, knowledge of PID is very important to implement the line following and self balancing algorithms. Also, one must know either Python or C++ as the algorithms need to be written in either of these two languages.

2.3 ROS1 vs ROS2:

1. Ros or Ros1 was meant for Ubuntu, Fedora and Arch Linux. But Ros2 provides support for macOS and Windows too. This increases the accessibility of Ros2 to users with various

operating systems and makes it easy for the users to use it. This in turn, widens the Ros2 community.

2. Despite the importance of reactivity and low latency in robot control, Ros itself is *not* a real-time OS (RTOS). It is possible, however, to integrate Ros with real-time code. The lack of support for real-time systems has been addressed in the creation of Ros2, a major revision of the Ros API which will take advantage of modern libraries and technologies for core Ros functionality and add support for real-time code and embedded hardware.

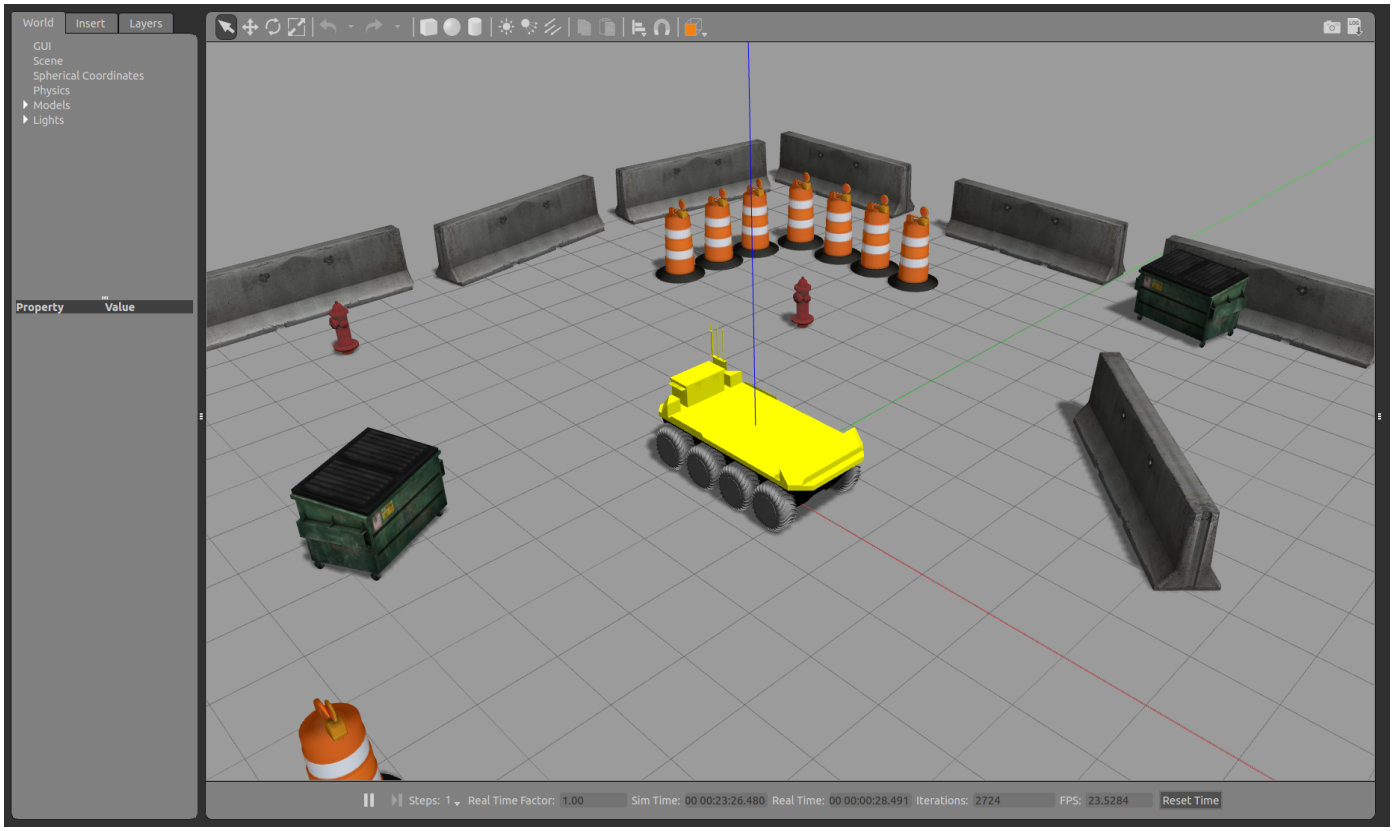
3. Ros2 also provides a bridge to Ros1 that handles bidirectional communication between the two systems. If you have an existing Ros1 application, you can start experimenting with Ros2 via the bridge and port your application incrementally according to your requirements and available resources.

There are many more new features in ROS2. We have listed the important ones above. The link for others will be provided in the references section at the end.

2.4 WHY USE GAZEBO? :

Robot simulation is an essential tool in every roboticist's toolbox. A well-designed simulator makes it possible to rapidly test algorithms, design robots, perform regression testing, and train AI system using realistic scenarios. The physics engine of

Gazebo is one of the best among the open sources softwares available out there for simulation. It is trusted and has been used in the industry for quite a long time. It is free of cost and satisfies every needs related to the simulation part of this project.



(Example of a bot being simulated in Gazebo around different obstacles.)

3. STAGES OF PROGRESS:

3.1 UNDERSTANDING ROS2 BASICS:

1. Workspace: A ROS2 workspace is a directory containing ROS2 packages. A basic ROS2 will have the following directories:

a. **Build**
b. **Install**
c. **Src**
d. **Log**

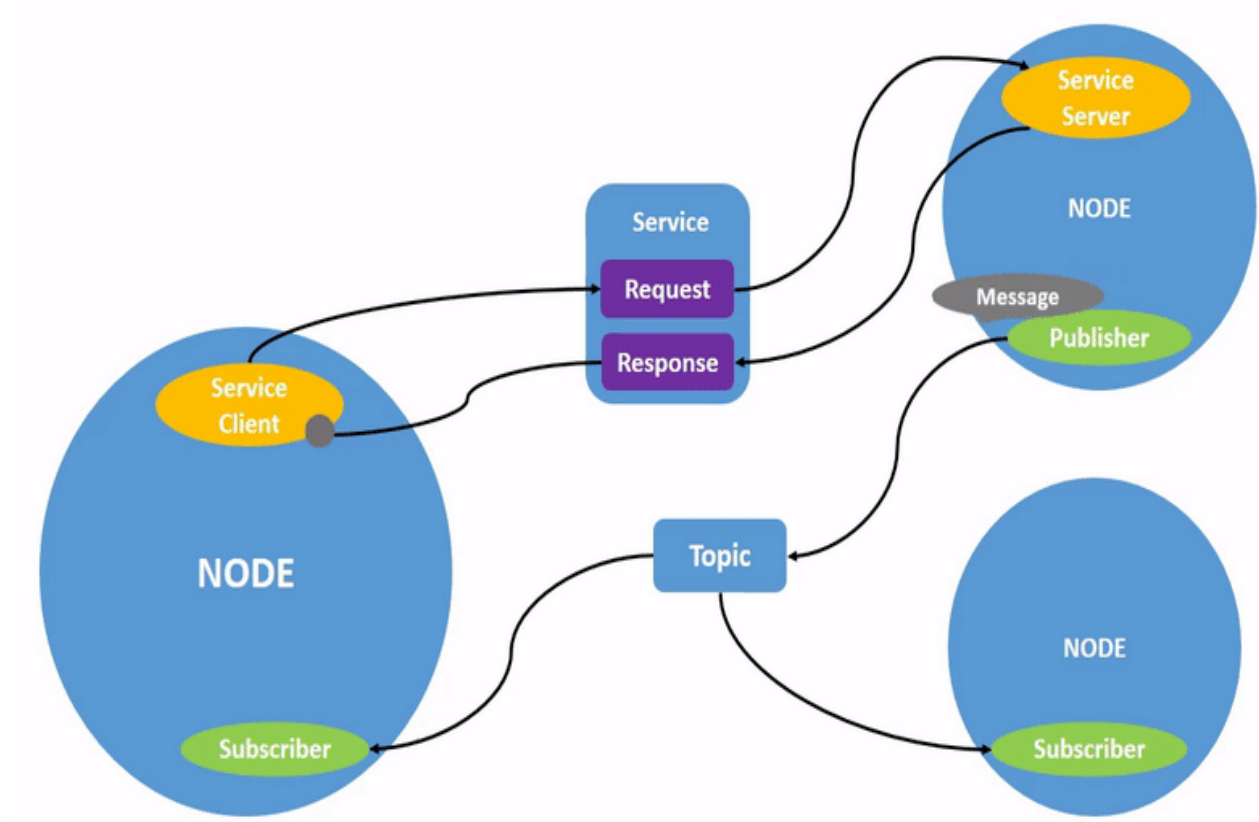
The install directory is where your workspace's setup files are, which you can use to source your overlay.

2. Colcon: Colcon is the new build tool replacing ament_tools. It is an iteration of ROS build tools. Every time you make a change in your code, you need to run the 'colcon build' command. Also, you need to source your workspace every time you open a new terminal.

3. ROS2 Package: A package can be considered a container for your ROS2 code. If you want to be able to install your code or share it with others, then you'll need it organized in a package. Package creation in ROS2 uses ament as its build system and colcon as its build tool. You can create a package using either CMake or Python, which are officially supported, though other build types do exist.

4. ROS2 node: A node is an executable file within a ROS2 package. In ROS2, a single executable (C++ program, Python program, etc.) can contain one or more nodes.

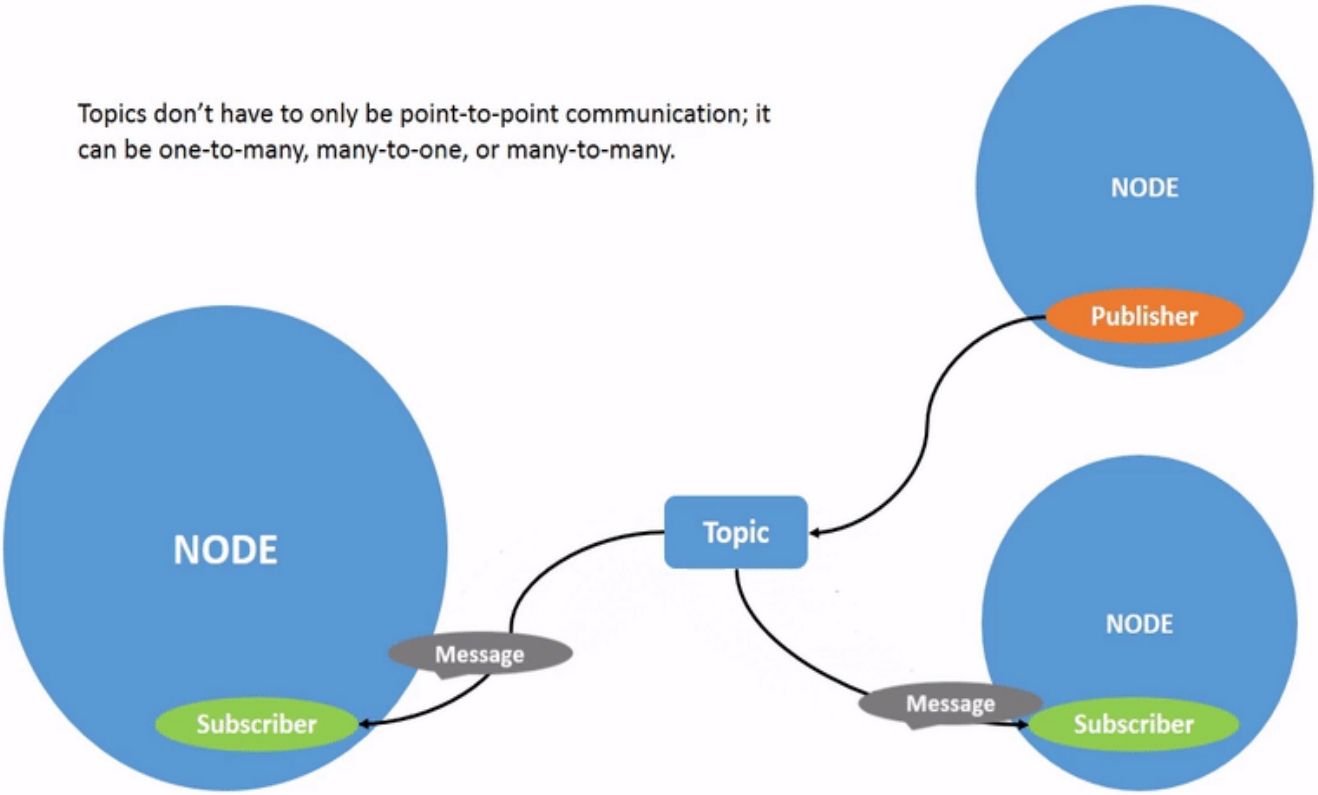
ROS2 NODES



Similarities and differences between ROS2 and ROS1 nodes.

	ROS2 node	**ROS1 node**
What is it?	An executable using ROS2 to communicate with other nodes.	An executable using ROS1 to communicate with other node.
(De)advertises itself and discovers other nodes using…	A *distributed* discovery process (which does not depend on a single node)	The ROS1 Master (a single node)
After discovery, communicates with other nodes…	Peer to peer	Peer to peer
Uses these client libraries (among others)	rclcpp = C++ client library rclpy = Python client library	roscpp = C++ client library rospy = Python client library

5. ROS2 topics: Ros2 topics are kind of a channel which help the nodes to exchange messages. A node may publish data to any number of topics and simultaneously have subscriptions to any number of topics.



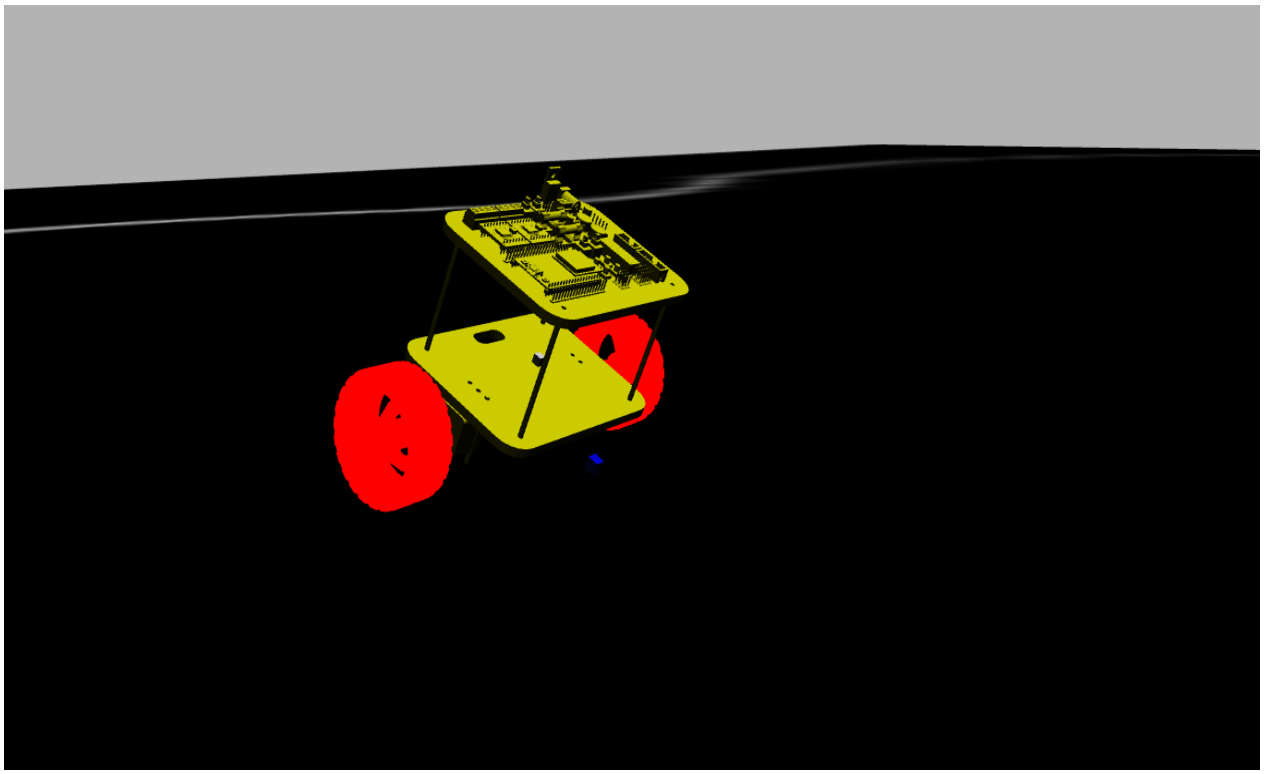
3.2 DESIGNING OF THE WallE ROBOT:

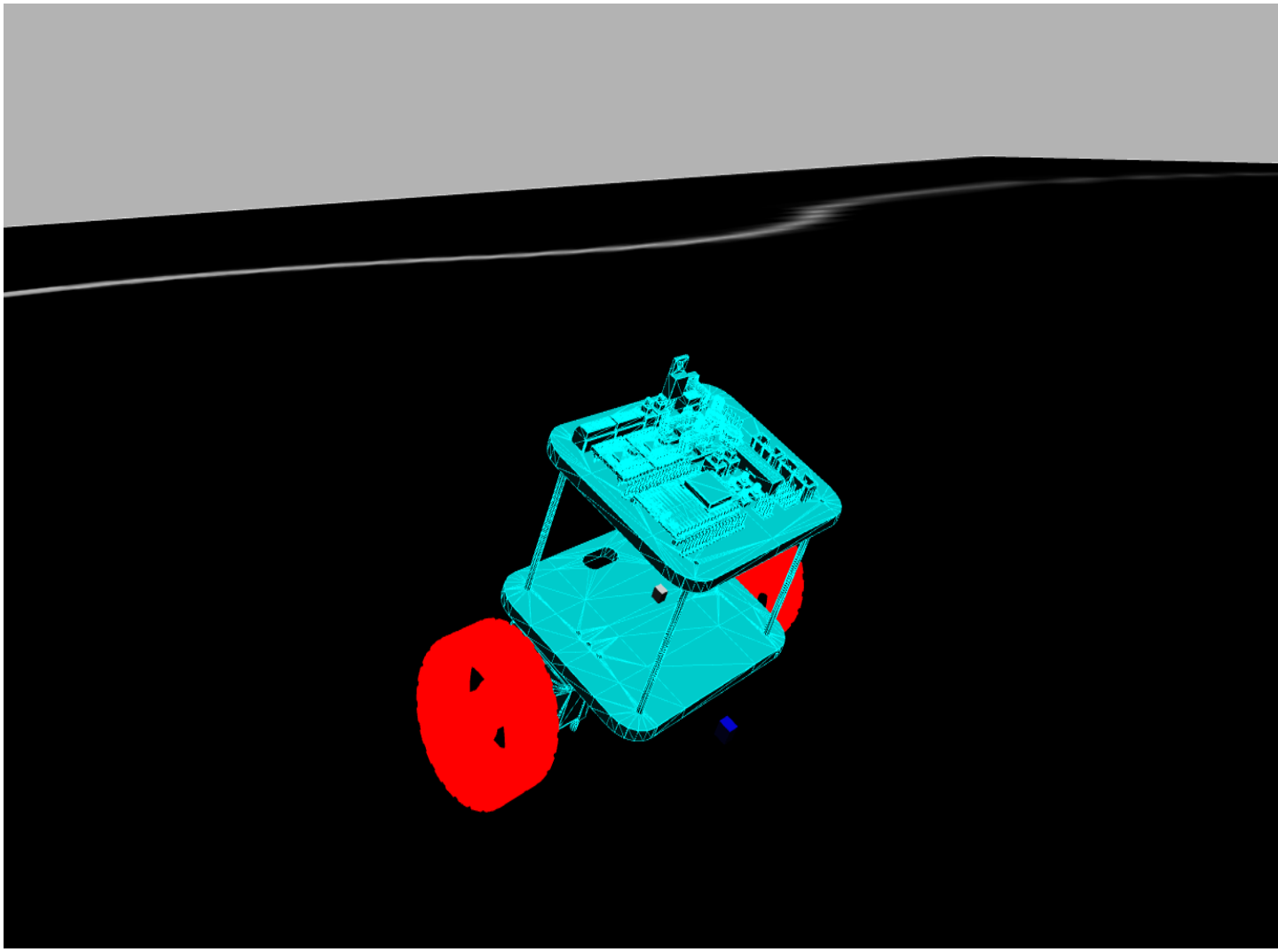
Prerequisites : A 2D sketch with proper dimensions for reference.

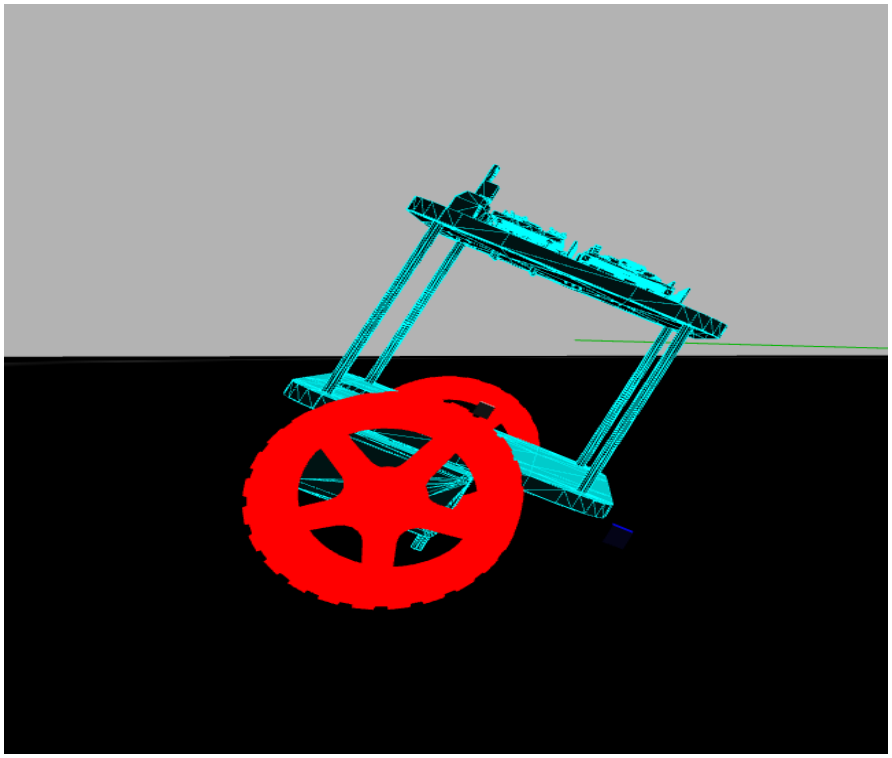
1. Start making 2D sketches of the different parts of the robot or with the sketch provided , import it into solidworks [Note : See to it that while importing , you have selected the required Unit system (MKS,CGS,INS,MMGS,etc)] and with Convert Entities tool , you can now select and make the necessary changes.
2. For 3D models, you can use the Boss-Extrude/Cut tools.

3. While Assembling all the parts , make sure that the Orientation i.e. the axes of the robot and the solidworks screen are the same.Assign proper mates between the parts.If more than one part is repeated (like in our case , the wheels)you can mirror that part simply by selecting the plane of reference.
4. **Note :** After assembly , do remember that the changes you make in the parts will also affect the final assembly.

These are some of the images of our bot.







The original colour of the WallE bot is yellow.

3.3 EXPORTING URDF:

1. To spawn our robot in gazebo, we need to export an URDF of our robot that we later convert to SDF.
2. Select the base_link (in our case - chassis) ,then the child links as the wheels or any other sub-parts, you can either keep the coordinate systems to default or assign them individually. Give proper types of joint(revolute,fixed,continuous,prismatic,etc) to the child links.
3. See to it that each and every part is selected once , be it a fixed or of a revolute joint to avoid missing them while spawning.
4. Preview and Export : Now you can export the URDF, and confirm the links and joints as well as assign values like mass,inertia,limit-velocity,limit-effort,etc or keep it as default and save it.
5. We'll now see that a folder is created that contains sub-folders : config,launch,meshes,urdf,Cmakelists.txt,package.xml,export.log,etc. The urdf folder contains a .urdf file which is then converted to sdf with the command : gz sdf -p name.urdf > name.sdf

Now we will give you a small glimpse of our sdf file

*Untitled Document 1 *Untitled Document 2

```
<sdf version='1.7'>
  <model name='walle'>
    <link name='base_link'>
      <inertial>
        <pose>-0 0 -0.0025 0 -0 0</pose>
        <mass>0.00680000</mass>
        <inertia>
          <ixx>0.000121426</ixx>
          <ixy>1.28749e-19</ixy>
          <ixz>2.00583e-22</ixz>
          <iyy>0.000118984</iyy>
          <iyz>-4.40398e-24</iyz>
          <izz>0.000240029</izz>
        </inertia>
      </inertial>
      <collision name='base_link_collision'>
        <pose>0 0 0 0 -0 0</pose>
        <geometry>
          <mesh>
            <scale>1 1 1</scale>
            <uri>meshes/base_link.STL</uri>
          </mesh>
        </geometry>
      </collision>
      <collision name='base_link_fixed_joint_lump__camera_link_collision_1'>
        <pose>0.058 0 -0.015 0 -0 0</pose>
        <geometry>
          <box>
            <size>0.007 0.007 0.007</size>
          </box>
        </geometry>
        <surface>
          <contact>
            <ode/>
          </contact>
          <friction>
            <ode/>
          </friction>
        </surface>
      </collision>
      <collision name='base_link_fixed_joint_lump__imu_link_collision_2'>
        <pose>0 0 0.01 0 -0 0</pose>
        <geometry>
          <box>
            <size>0.005 0.005 0.005</size>
          </box>
        </geometry>
        <surface>
          <contact>
            <ode/>
          </contact>
          <friction>
```

```
113         <plugin name='camera_controller' filename='libgazebo_ros_camera.so'>
114           <alwaysOn>1</alwaysOn>
115           <updateRate>100</updateRate>
116           <cameraName>rrbot/camera1</cameraName>
117           <imageTopicName>image_raw</imageTopicName>
118           <cameraInfoTopicName>camera_info</cameraInfoTopicName>
119           <frameName>camera_link</frameName>
120           <hackBaseline>0.07</hackBaseline>
121           <distortionK1>0.0</distortionK1>
122           <distortionK2>0.0</distortionK2>
123           <distortionK3>0.0</distortionK3>
124           <distortionT1>0.0</distortionT1>
125           <distortionT2>0.0</distortionT2>
126         </plugin>
127         <pose>0 0 0 0 -0 0</pose>
128       </sensor>
129       <sensor name='imu_sensor' type='imu'>
130         <plugin name='imu_plugin' filename='libgazebo_ros_imu_sensor.so'>
131           <ros>
132             <namespace>/demo</namespace>
133             <remapping>~/out:=imu</remapping>
134           </ros>
135           <initial_orientation_as_reference>0</initial_orientation_as_reference>
136         </plugin>
137         <always_on>1</always_on>
138         <update_rate>100</update_rate>
139         <visualize>1</visualize>
140         <imu>
141           <angular_velocity>
142             <x>
143               <noise type='gaussian'>
144                 <mean>0</mean>
145                 <stddev>0.0002</stddev>
146                 <bias_mean>7.5e-06</bias_mean>
147                 <bias_stddev>8e-07</bias_stddev>
148               </noise>
149             </x>
150             <y>
151               <noise type='gaussian'>
152                 <mean>0</mean>
153                 <stddev>0.0002</stddev>
154                 <bias_mean>7.5e-06</bias_mean>
155                 <bias_stddev>8e-07</bias_stddev>
156               </noise>
157             </y>
158             <z>
159               <noise type='gaussian'>
160                 <mean>0</mean>
161                 <stddev>0.0002</stddev>
162                 <bias_mean>7.5e-06</bias_mean>
163                 <bias_stddev>8e-07</bias_stddev>
164               </noise>
```

3.4 WHY IS SDF PREFERRED IN ROS2? :

In Ros2 the SpawnEntity method let's the SDF library handle the URDF conversion. So it’s better that we use a SDF file instead of an URDF or XACRO to ensure that all our tags work smoothly.

3.5 WHAT IS A WORLD FILE? :

The world file contains a description of the world to be simulated by Gazebo. It describes the layout of robots, sensors, light sources, user interface components, and so on. The world file can also be used to control some aspects of the simulation engine, such as the force of gravity or simulation time step. Gazebo world files are written in XML. So, basically a world file helps to create an environment according to the purpose of the bot.
Common tags in world file,
<model name= “ ”> </model> - It specifies the name of the model.

<visual name= “ ”> </visual> - It specifies the name of the visual element. I.e. The name of the element that we will see in the GUI.

<geometry>,<surface>,<box>,<material> are some of the other tags used to describe the world.

The world file for our project, sra.world is as follows:

```xml
<?xml version="1.0"?>
<sdf version="1.4">
  <world name="default">
    <scene>
      <ambient>0 0 0 1</ambient>
      <shadows>0</shadows>
      <grid>0</grid>
      <background>0.7 0.7 0.7 1</background>
    </scene>
    <include>
      <uri>model://sun</uri>
    </include>

    <model name="ground">
      <pose>1 2.3 -.1 0 0 0</pose>
      <static>1</static>
      <link name="ground">
        <collision name="ground_coll">
          <geometry>
            <box>
              <size>10 10 .1</size>
            </box>
          </geometry>
          <surface>
            <contact>
              <ode/>
            </contact>
          </surface>
        </collision>
        <visual name="ground_vis">
          <geometry>
            <box>
              <size>10 10 .1</size>
            </box>
          </geometry>
          <material>
            <script>
              <uri>models/course.material</uri>
              <name>course</name>
            </script>
          </material>
        </visual>
      </link>
    </model>
  </world>
</sdf>
```

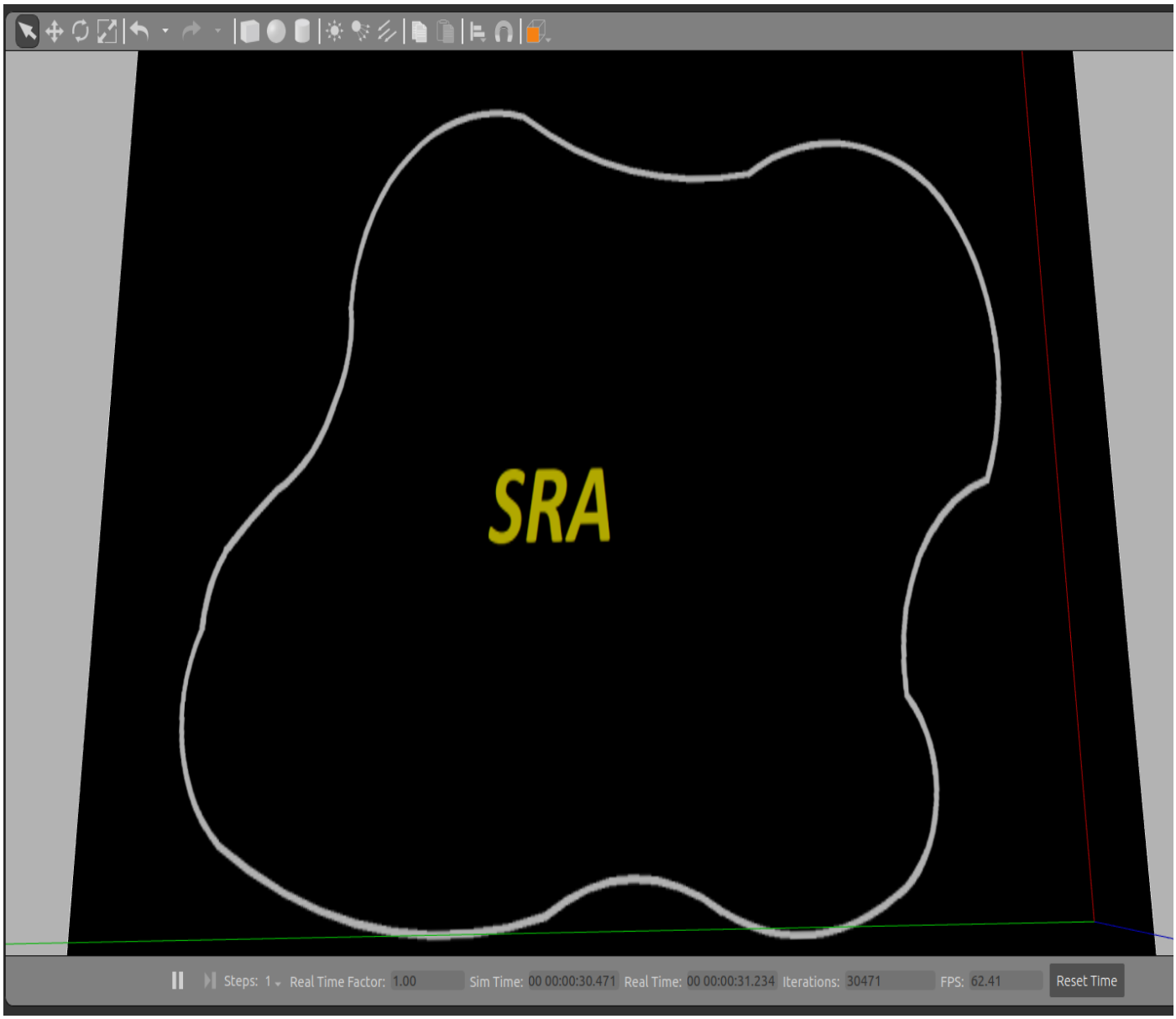
3.6 DESIGNING OF THE WORLD USING AN IMAGE:

Our current file system is something like this.

```
├ 📂config
|  └ 📜joint_names_Wall-E-urdf1.yaml          # Configuration file for joints of the bot
├ 📂launch                                   # All launch files
|  ├ 📜emptyworld.launch.py
|  ├ 📜gazebo.launch.py
|  ├ 📜gzclient.launch.py
|  ├ 📜gzserver.launch.py
|  ├ 📜line_following.launch.py          # Launch file for bot for line-following algorithm
|  ├ 📜robot_state_publisher.launch.py          # Launch file for bot for self-balancing
algorithm
|  ├ 📜self_balancing.launch.py
├ 📂meshes                                   # Meshes for different parts of bot
|  ├ 📜base_link.STL
|  ├ 📜chassis.STL
|  ├ 📜leftwheel.STL
|  └ 📜rightwheel.STL
├ 📂models
|  ├ 📜course.material                  # The file which links the world file to the png of
the world
|  └ 📜sra.png                  # The png which decides the design of the world in Gazebo
├ 📂rviz
|  └ 📜urdf_config.rviz                 # The file for rviz configuration
├ 📂src                                  # All cpp codes are stored here
|  ├ 📜line_following.cpp              # The line-following algorithm
|  ├ 📜self_balancing.cpp              # The sdf files for bot are stored here
├ 📂urdf
|  ├ 📜walle.csv
|  ├ 📜walle.urdf
|  ├ 📜walle.sdf                  # The sdf file for self-balancing bot
|  ├ 📜walle2.sdf                       # The sdf file for line-following bot
├ 📂worlds
|  └ 📜sra.world                  # The line-following path
```

```
   ├ 📜.gitignore
   ├ 📜CMakeLists.txt                    # Contains all the information regarding the
packages to be   imported
   ├ 📜LICENSE
   ├ 📜README.md
   ├ 📜export.log
   ├ 📜package.xml                       # Contains all the information regarding the
dependencies to be  imported
```

We can see that the world file, which we previously discussed about, is in the worlds folder. If we look closely at that world file again, there is an **<uri>** tag which stands for **Uniform Resource Identifier** tag. What it does here is it links the world file to a file called course.material which can be found in the models folder as shown above in the tree structure. If we look into that course.material file, it describes some additional properties for the world and it too links itself to an image sra.png which can also be found in the models folder. And the entire design of the world, how big it is, what is it's colour ,etc is based on that png. So, we here, have designed an image according to our own requirements and used it to generate a world in Gazebo. We have used autocad to draw the desired shape of the line following path and autocad also helps us to give that path width some precise dimensions. Then we've exported the plot as a png. We have used Microsoft Paint to colour the Png black and white and add text to it. This is how we designed our own custom image for our own custom world in gazebo. It might sometimes take a few tries to get your precise dimensions. But this method comes in very handy when you have very specific requirements for a world that a Gazebo tag might not be able to provide. This method gives you liberty in terms of design. Also, autocad has the scaling feature where you can scale an image if you really want to enlarge an image or make it really small.



This is the world for our project. It is made by using the method described above. It might appear small in this image because it's taken after really zooming out a lot. Also, when compared to the size of our bot, the world is actually very large.

3.7 CREATION OF LAUNCH FILES :

Launch files are very common in ROS to both users and developers. They provide a convenient way to start up multiple nodes. They also let us define different parameters like the coordinates where we want to spawn the bot. In our project, we've efficiently made use of launch files where our single launch file launches the bot, the world and gazebo for us.

But there are some major differences in launch files when we look at ROS and ROS2.
ROS uses XML to write launch files. But ROS2 uses python to write it's launch files. In ROS there is also a Python API. The problem is: no one is aware of it, and there's almost zero documentation about it. So, no one uses it. ROS2 uses python functions to write a launch file.

```python
import os

from ament_index_python.packages import get_package_share_directory
from launch import LaunchDescription
from launch.actions import ExecuteProcess
from launch_ros.actions import Node
from scripts import GazeboRosPaths


def generate_launch_description():
    model_path = GazeboRosPaths.get_paths()

    env = {
        "GAZEBO_MODEL_PATH": model_path,
    }

    sdf_prefix = get_package_share_directory("my_bot")
    sdf_file = os.path.join(sdf_prefix, "urdf", "walle.sdf")

    world_prefix = get_package_share_directory("my_bot")
    world_file = os.path.join(world_prefix, "worlds", "sra.world")

    rviz_config_prefix = get_package_share_directory("my_bot")
    rviz_config_path = os.path.join(rviz_config_prefix, 'rviz/urdf_config.rviz')
```

```
return LaunchDescription(
        [
            ExecuteProcess(
                cmd=[
                    "gazebo",
                    "-s",
                    "libgazebo_ros_init.so",
                    "-s",
                    "libgazebo_ros_factory.so",
                     world_file,
                ],
                output="screen",
                additional_env=env,
            ),
            Node(
                package="gazebo_ros",
                node_executable="spawn_entity.py",
                arguments=[
                    "-entity",
                    "walle",
                    "-x",
                    "-1",
                    "-y",
                    "0",
                    "-z",
                    ".41",
                    "-b",
                    "-file",
                    sdf_file,
                ],
            ),
```

First we import all our dependencies and then create a function. Inside it, we describe variables and assign them all the paths of the files we aim to launch. Our launch file must return a launch description object.

```
Node(
                package="robot_state_publisher",
                node_executable="robot_state_publisher",
                output="screen",
                arguments=[sdf_file],
            ),
            Node(
            package='rviz2',
            executable='rviz2',
            name='rviz2',
            output="screen",
            arguments=[rviz_config_path],
    )
        ]
    )
```

Through the return object, we launch all the things we want to.

3.8 USING GAZEBO PLUG-INS:

Gazebo plugins give your URDF models greater functionality and can tie in ROS2 messages and service calls for sensor output and motor input.
Gazebo supports several plugin types, and all of them can be connected to ROS, but only a few types can be referenced through a URDF file:

1. ModelPlugins, to provide access to the physics::Model API
2. SensorPlugins, to provide access to the sensors::Sensor API
3. VisualPlugins, to provide access to the rendering::Visual API

For our project, we will be using three plug-ins namely the ‘Differential drive’ plug-in, the ‘imu sensor’ plug-in and the ‘camera’ plug-in.

1. Differential drive plug-in: It’s a model plugin that provides a basic controller for differential drive robots in Gazebo. It is basically used to provide linear and angular velocity to our bot in Gazebo.

In our project, it helps to move the bot as described above.

2. Imu sensor plug-in: Simulates an Inertial Motion Unit sensor, the main differences from IMU (GazeboRosIMU) are: - inheritance from SensorPlugin instead of ModelPlugin, - measurements are given by gazebo ImuSensor instead of being computed by the ros plugin.

In our project, it’s used to get orientation readings based on which we calculate the pitch of the bot which is necessary for self-balancing the bot.

3. Camera plug-in: It’s a simple RGB camera used to get RGB values from the surface where the bot is facing.

In our project, it’s used to get the RGB values of the surface the bot is facing so that we can detect the line the bot needs to follow and make it follow the line.

4. IMPLEMENTING THE ALGORITHMS:

4.1 UNDERSTANDING PID :

PID is made up of three terms, namely **Proportional**, **Integral** and **Derivative**. Each of these perform different functions and are important for tuning of the bot. It is one kind of device used to control different process variables like pressure, flow, temperature, and speed in industrial applications.

Proportional Term: Proportional gives an output that is proportional to current error(e). It compares the desired or set point with the actual value or feedback process value. The resulting error is multiplied with a proportional constant to get the output.

P_term = Kp*e

Where Kp = Proportional gain,
e = error

Integral Term: Due to the limitation of P_term where there always exists an offset between the process variable and setpoint, I_term is needed, which provides necessary action to eliminate the steady-state error. It basically adds the small errors until it becomes a significant figure.

I_term = Ki*∫ e(t) dt

Where Ki = Integral gain,
∫ e(t) dt = Cumulative error

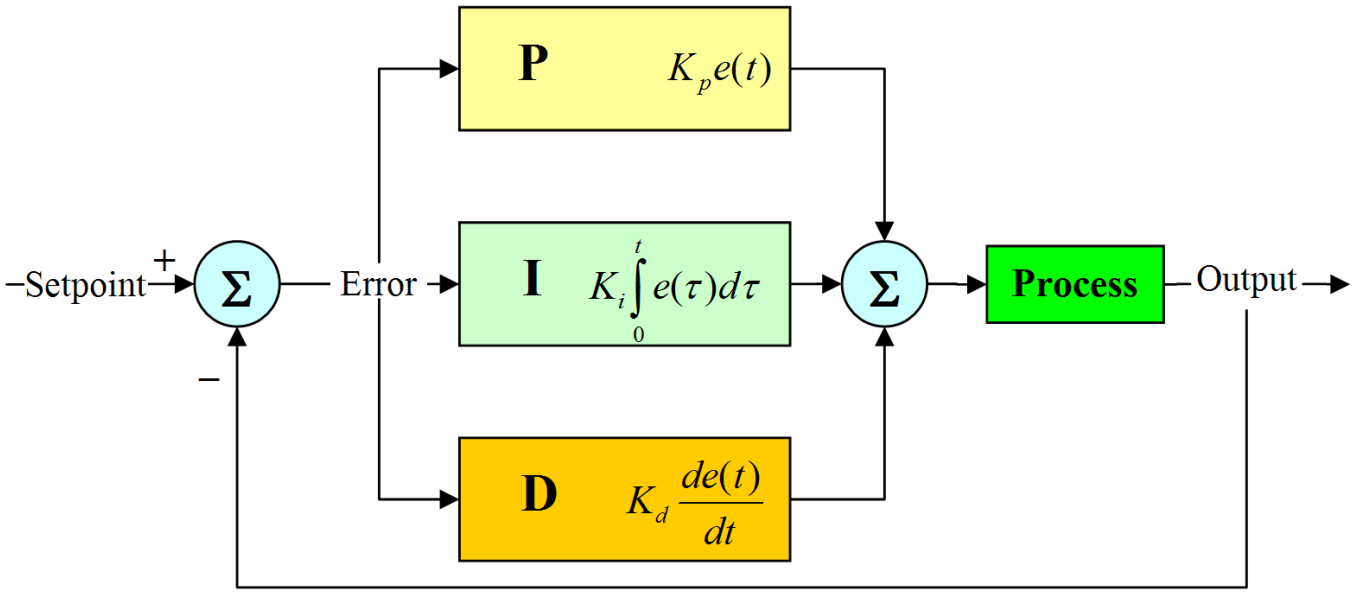
Derivative term: I_term doesn't have the capability to predict the future behavior of error. So it reacts normally once the setpoint is changed. D_term overcomes this problem by anticipating the future behavior of the error. Its output depends on the rate of change of error with respect to time, multiplied by derivative constant. It gives the kick start for the output thereby increasing system response.

D_term = Kd*(de/dt)

Where Kd = Derivative gain,
de/dt = Change in error w.r.t time

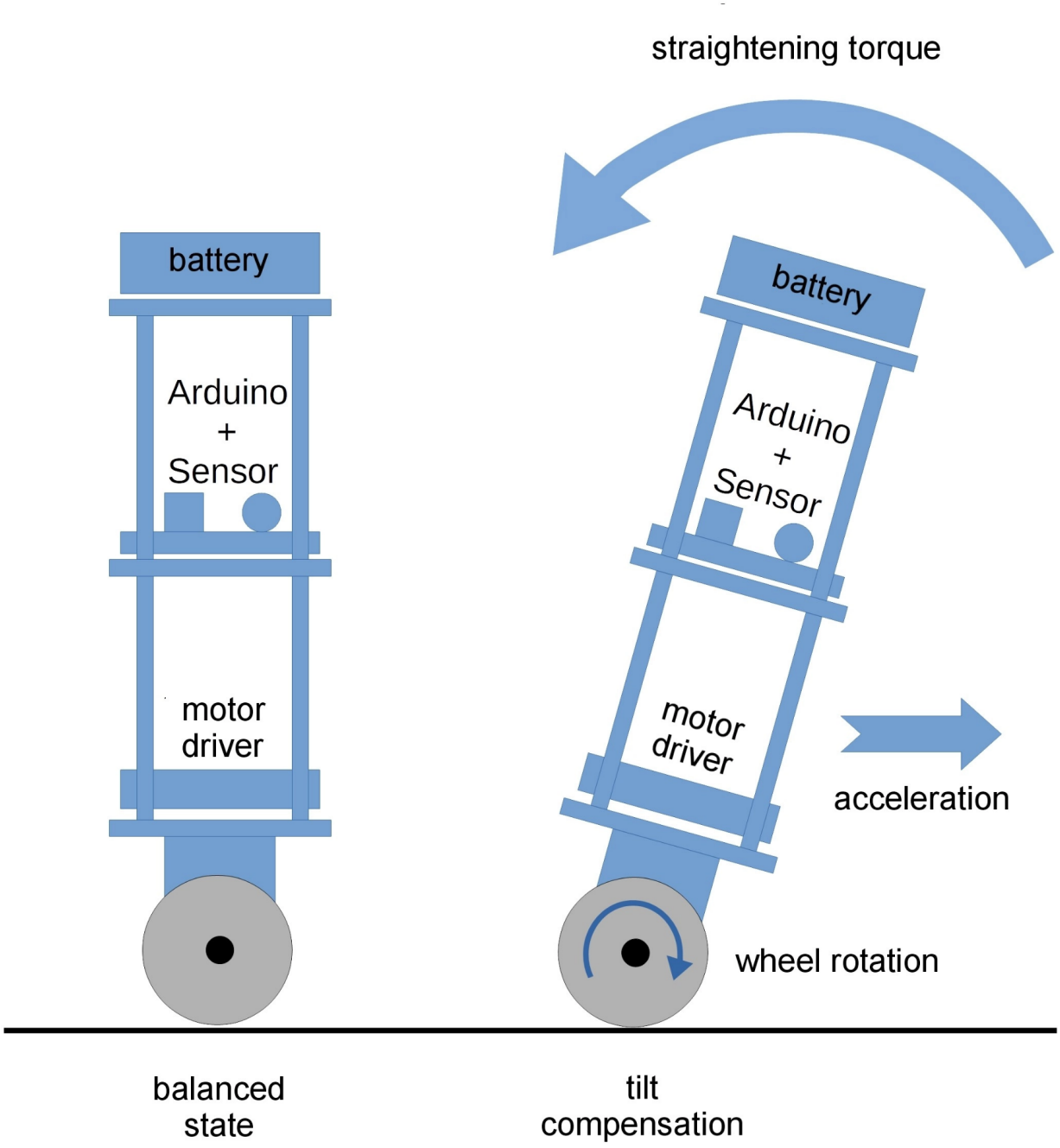
$$u(t) = K_p e(t) + K_i \int e(t)dt + K_p \frac{de}{dt}$$

Where u(t) = Correction term



4.2 SELF-BALANCING ALGORITHM :

Our bot is a two wheeled bot. So it's necessary to write an algorithm so that the bot can balance itself and prevent itself from falling.
Suppose that the bot is initially stationary. If it moves forward the front of the bot goes upwards and vice versa for when we move the bot backwards.



Explaining through small snippets of code. We have used C++ language. ROS2 C++ nodes use the concept of classes.

```
class self_balancing : public rclcpp::Node
{
  // Subscriber and publisher node
  public:
    self_balancing()
    : Node("selfbalancing")
    {
      subscription_ = this->create_subscription<sensor_msgs::msg::Imu>(
      "/demo/imu", 10, std::bind(&self_balancing::topic_callback, this, _1));

      publisher_ = this->create_publisher<geometry_msgs::msg::Twist>("/demo/cmd_vel",10);
        timer_ = this->create_wall_timer(
        500ms, std::bind(&self_balancing::timer_callback, this));
    }
```

Here we first convert the pitch angle from radians to degrees. Then we have a class which creates a node for publisher and subscriber. We take input from imu sensor and publish to cmd_vel topic.
Each of the publisher and subscriber have an individual callback function here.

```
pitch = msg->orientation.y;
     e = Convert(pitch);
     RCLCPP_INFO(this->get_logger(), "Publishing: '%lf;", e);

     // Pitch error, pitch error difference, previous pitch error, pitch rate, pitch area
     pitch_error = 5*(e-pitch_desired);
     pitch_rate = pitch_error - prev_pitch_error;
     pitch_area += pitch_error ;
     RCLCPP_INFO(this->get_logger(), "Publishing: '%lf;", pitch_area);

     // Making pitch_area zero everytime it crosses maximum or minimum value
     if(pitch_area > 5.0/1.4)
     {
       pitch_area = 0.0;
     }
     else if(pitch_area < -5.0/1.4)
     {
      pitch_area = 0.0;
     }
```

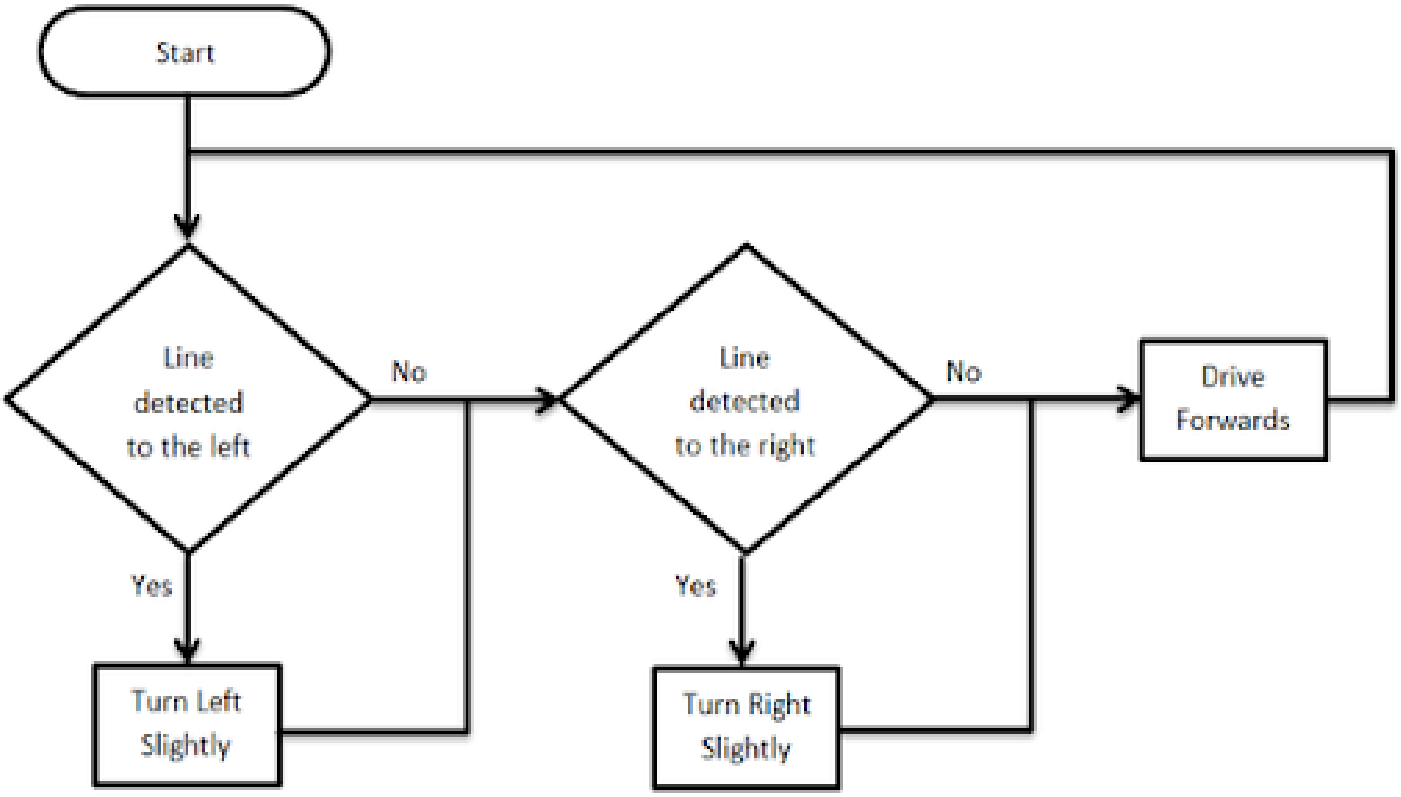
Now we take in from imu, orientation in Y direction which is the pitch for our bot. We calculate the pitch error, pitch rate, pitch area which are necessary for PID.

```
P_term = Kp*pitch_error;
     D_term = Kd*pitch_rate;
     I_term = pitch_area*Ki;
     correction = P_term + D_term + I_term;
     prev_pitch_error = pitch_error;
```

Then we calculate the P,I and D terms and the correction term as well. And finally we assign it to the linear velocity publisher. The hardest part in this is the PID tuning. It has to be done very carefully, otherwise the bot will not be able to balance itself.

4.3 LINE-FOLLOWING ALGORITHM :

Our world is black surface having a white path. So we need to ensure that the bot follows the white line.



Explaining through code snippets.

Here, similar to self-balancing, we first create a class and node for publisher and subscriber. We subscribe to the RGB camera to get the rgb values of the pixels.

```
if( r > 125.00)
        {
          if(flag == 1)
          {
            index = i+1;
            flag = 0;
          }
           count ++;
        }
      }

        if(count == 10)
        {
          if(index <= 12)
          {
            error = 16 - index - 4;
            direction = 1;
          }
          else
          {
            error = index + 4 - 17;
            direction = 2;
          }
        }
```

```
else if(count >= 5 && count < 10)
        {
          if(index < 12)
          {
            error = 12 - index;
            direction = 1;
          }
          else if(index > 21)
          {
            error = index - 21;
            direction = 2;
          }
        }
```

Here, the RGB values are stored in an array data whose size is 96. But we only consider ‘R’ values here because it’s sufficient to distinguish the white line from the black surface. After that we check for white pixels and store the index of the first pixel. We also count the number of white pixels. If the number of white pixels is 10, we check whether the first white pixel is whether to the left or right of it’s desired position. Based on that we calculate the error. Similarly, if the number of white pixels is less than 10 but greater than or equal to 5, we again check whether the first pixel is to the left or right and calculate error accordingly. After this, we calculate the other PID terms like correction, previous error,etc.

```
  // Callback function for publisher
  void timer_callback() {

   // Giving angular speed either clockwise or anti-clockwise
   if(direction == 1)
   {
     auto message = geometry_msgs::msg::Twist();
     message.linear.x = 0.2;
     message.angular.z = -ang_speed ;
     publisher_->publish(message);
   }
   else if(direction == 2)
   {
     auto message = geometry_msgs::msg::Twist();
     message.linear.x = 0.2;
     message.angular.z = ang_speed ;
     publisher_->publish(message);
   }
  }
  rclcpp::TimerBase::SharedPtr timer_;
  rclcpp::Publisher<geometry_msgs::msg::Twist>::SharedPtr publisher_;
};
```

Now we just publish the angular velocity based on whether the turn requires a clockwise or anticlockwise turn. Here the key is having the correct Kp value and the correct publishing rate of the message. The Kp value plays a huge role in scaling down the error since error here is in terms of difference in positions, so it can tend to be larger than normal errors.

4.4 SELF-BALANCING AND LINE-FOLLOWING COMBINED ALGORITHM :

We will only go through some important parts of the code to understand the concept.

```
public:
  walle()
      : Node("wall_e")
  {
    subscription_ = this->create_subscription<sensor_msgs::msg::Imu>(
        "/demo/imu", 10, std::bind(&walle::topic_callback, this, _1));

    subscription1_ = this->create_subscription<sensor_msgs::msg::Image>(
        "/camera1/image_raw", 10, std::bind(&walle::topic_callbacks, this, _1));

    publisher_ = this->create_publisher<geometry_msgs::msg::Twist>("/demo/cmd_vel", 10);
    timer_ = this->create_wall_timer(
        100ms, std::bind(&walle::timer_callback, this));
  }
```

We first create a node for subscribing to imu and rgb camera and publishing velocity to the bot.

```
if (e < 6.000000 && e > -6.000000)
    {
      balanced_state = true;
    }

    // Pitch error, pitch error difference, previous pitch error, pitch rate, pitch
    pitch_error = 5 * (e - pitch_desired);
    pitch_rate = pitch_error - prev_pitch_error;
    pitch_area += pitch_error;
```

Then we fix the range of the balanced state of the bot and then we calculate the pitch error, pitch area and other terms related to PID. After that we fix the min and max range of pitch area and calculate the correction term. Then we store the r values from rgb and count the number of red pixels.

```
if (count < 9) // If number of red pixels is less than nine
    {
      if (index <= 5)
      {
        error = 5 - index; // Checking devaition on left side
        direction = 1;
      }
      else if(index >= 18)
      {
        error = index - 18 ; // Checking devaition on right side
        direction = -1;
      }
    }
```

If the number of red pixels is less than 9, then we calculate the error.

```
if (balanced_state == false)
 {
   auto message = geometry_msgs::msg::Twist();
   message.linear.x = correction;
   message.angular.z = 0.0;
   RCLCPP_INFO(this->get_logger(), "Publishing: '%lf'", message.linear.x);
   publisher_->publish(message);
 }
```

If the bot is not balanced, then only provide linear velocity for balancing. Then calculate the PID terms for line-following.

```
else
  {
    if (direction == 1)
    {
      auto message = geometry_msgs::msg::Twist();
      message.linear.x = speed;
      message.angular.z = -ang_speed;
      publisher_->publish(message);
      RCLCPP_INFO(this->get_logger(), "Publishing ang: '%lf;", message.angular.z);

    }
    else if (direction == -1)
    {
      auto message = geometry_msgs::msg::Twist();
      message.linear.x = speed;
      message.angular.z = ang_speed;
      publisher_->publish(message);
      RCLCPP_INFO(this->get_logger(), "Publishing ang: '%lf;", message.angular.z);
    }
  }
}
```

Otherwise, provide linear and angular velocity both.

5. CHALLENGES FACED AND THE SOLUTIONS APPLIED :

5.1 CHALLENGES AND SOLUTIONS :

1. CHALLENGE : ROS2 is quite a new technology. It has quite a few differences when compared to ROS1. Also, since it's so new, it doesn't have many resources and has very little amount of projects when compared to ROS1. Also, most of the time, we had to rely only on the official documentation for information which sometimes would prove to be insufficient. Also, most of the doubts on the forums regarding ROS2 were unanswered.

SOLUTION : Try to understand from whatever is available, build your own logical deductions of what could work and keep on trying yourself.

2. CHALLENGE : The meshes of the bot wouldn't launch in gazebo. So, gazebo would tell us that the bot has been spawned but we actually couldn't see it since the meshes were not imported.

SOLUTION : After a lot of trying we realized that ROS2 doesn't accept the full address of the meshes in the uri tag. It won't spawn the meshes if you give the full address. You need to only include the folder and the name of the mesh in the address.

3. CHALLENGE : There weren't any world files which we could import directly since our world had very specific size requirements.

SOLUTION : So, we had to invest quite a lot of time in creating our world file by designing an image as described earlier.

4. CHALLENGE : The bot's dimensions weren't coming right when we designed it first.

SOLUTION : We eventually realized that there was some factor affecting the dimensions of the bot and after designing it multiple times we eventually got it right.

5. CHALLENGE : The bot, when finally spawned in Gazebo, was not stable. It used to move on it's own. Also the axes of the bot didn't match those of the world and the bot would spawn on it's back laying on the ground.

SOLUTION : We exported a huge amount of Urdfs to make the bot stable by changing various factors like mass, inertia ,etc and considering the chassis, the spacers and the PCB board as a single mesh and changing the spacers and motors a few times. Also, we got the axes correct through trial and error.

6. CHALLENGE : The imu plug-in, when tried to be incorporated in the SDF wouldn't work. It didn't generate the topic. We tried different imu plug-in syntaxes from different websites but none of them worked.

SOLUTION: Eventually, after trying many solutions for a few days, what we did was we took the imu plug-in and inserted it in a xacro file of another bot. Since xacro files are used a lot along with urdfs in ROS, we thought that it would work correctly with a xacro file. We tried to build that file but a few errors popped up which we eventually solved one by one. Then we converted the xacro file to a urdf file through the terminal. Now we took this block of code required for the plug-in from that urdf and inserted it in our bot's urdf. Then we converted that urdf to sdf again through the terminal. This is how we got the plug-in working for our bot's sdf file.

7. CHALLENGE : Our bot's chassis wouldn't lift up even after providing a huge value of forward velocity.

SOLUTION : Eventually we realized that this might be due the unsymmetrical shape of the chassis at the front and also the chassis being too heavy. So, we made the chassis symmetrical and adjusted the mass of the chassis through trial and error.

8. CHALLENGE : It was very difficult to tune our bot for self-balancing since it was very small in size. Also, our bot was very sensitive to velocity and would fall on either side in a matter of second as soon as provided with even a small amount of velocity. Also, our bot wouldn't respond fast enough to the published value of velocity which made keeping it in a balanced state extremely difficult.

SOLUTION : We had to keep on trying to tune our bot for quite a few days before it gave decent results.The key is to try and tune the Kp first, then Kd and then Ki.

9. CHALLENGE : The combining of the self-balancing and line-following was a bit tough in simulation. We couldn't control each wheel individually which created some problems. Also, we the frequencies of the two codes didn't match. And we had to change the orientation of the camera and make a new image for the world too.

SOLUTION : We did the tuning of the bot all over again for the combined code. Also we made the line quite thin and used red colour for the line since due to the self-balancing the bot sometimes faced the sky due to which the camera was giving white pixel readings for both the line as well as the sky.

6. APPLICATIONS :

6.1 INDUSTRIAL USE :

The robots in industries perform a lot of tasks, including the material handling. To automate the process of material handling, the robot has to be guided properly and using a dedicated vision system or coding the coordinates is not feasible. The simplest solution, paint colour strips on the floor for the robot to follow, this deals with the transportation of materials. The material slots can also be colour coded to allow the robot to distinguish. This is the major application of the line following robots.

6.2 USED IN RESTAURANTS :

A line following bot can be used to deliver products from point of production to point of utilisation. It can be used in a restaurant for transporting food items made by a chef to the customers' table.

6.3 ADVANTAGES OF SELF-BALANCING ROBOTS :

A two wheeled self balancing bot can cut quite a lot of the cost in designing the bot. You would require only two wheels, two sensors and a couple of the bot’s body parts and you are ready to go. Also, a two wheeled bot has much more accuracy in terms of velocity and during turns than a three or four wheeled robot. We can have more control over a two wheeled robot as compared to other robots.

7. CONCLUSION AND FUTURE WORK :

7.1 CONCLUSION AND THINGS LEARNT :

To conclude, we would like to say that over the course of the past few weeks working on this project, we have learnt a lot of concepts, lessons and skills not only related to the project but also related to engineering and life in general. We had to do a lot of research and hard work because choosing to work with ROS2 in the first place was like firing an arrow in the dark. We didn't know anything about ROS2 nor did we know whether we would be able to make such significant progress in this project or not. But thanks to the support and guidance of our mentors and seniors, we were able to make some significant progress in this project. We also learned how to research about new topics, how to debug efficiently, how to write algorithms, how to work professionally on a project and much more.

What did we learn from the project?

1. We understood how to work with ROS2.
2. We got quite a lot of experience in simulation due to the extensive use of Gazebo and a small use of Rviz in our project.
3. We used SolidWorks in our project to design our robot. So we learned quite a bit about SolidWorks and designing.

4. We also used Autocad and Microsoft paint for designing the image for our world in Gazebo. This gave us the opportunity to understand a bit about these softwares.
5. We also learned about algorithms of self-balancing and line-following. We understood how a bot actually balances itself using readings from sensors like imu and how it moves on a line.
6. We understood what PID is, studied about PID controllers, how to implement PID, the uses and importance of PID, etc.
7. Since we used C++, we came across some new and important concepts like smart pointers, their advantages and std::bind , etc.

7.2 FUTURE ASPECTS OF THE PROJECT :

- We have combined the self-balancing and line-following algorithms and implemented it too, producing results with which our mentors are satisfied for now . It travels quite a lot of distance successfully but it can't complete the path due to a small bug or error. If possible, we would like to get it corrected.

- Our long term goal is to implement the combined algorithm on the real WallE bot of SRA VJTI.

- We would also like to implement maze solving algorithms on it.

8. REFERENCES :

8.1 Useful links and Research Papers :

- Our github repository : https://github.com/Aryaman22102002/Wall-e-simulation-ros2
- My answer to why the meshes are not spawning in ROS2 : https://answers.gazebosim.org//question/26073/cannot-spawn-urdf-into-gazebo-using-ros2/
- ROS2 installation links : https://www.youtube.com/watch?v=fxRWY0j3p_U https://docs.ros.org/en/foxy/Installation/Ubuntu-Install-Debians.html
- ROS2 workspace setup : https://www.theconstructsim.com/ros2-in-5-mins-007-how-to-create-a-ros2-overlay-workspace/
- ROS2 tutorial: https://docs.ros.org/en/foxy/Tutorials.html
- https://navigation.ros.org/setup_guides/urdf/setup_urdf.html
- Plug-ins : http://gazebosim.org/tutorials?tut=ros_gzplugins#Camera
- https://github.com/sezan92/self_balancing_robot
- Gazebo installation : http://gazebosim.org/tutorials?tut=install_ubuntu
- Rviz installation : https://zoomadmin.com/HowToInstall/UbuntuPackage/rviz
- Rviz setup: https://www.youtube.com/watch?v=PjMrzIdtUOw

- https://github.com/chapulina/dolly/blob/foxy/dolly_gazebo/launch/dolly.launch.py
- https://www.wescottdesign.com/articles/pid/pidWithoutAPhd.pdf
- http://brettbeauregard.com/blog/2011/04/improving-the-beginners-pid-introduction/
- Official WallE v2.2 repository of SRA-VJTI : https://github.com/SRA-VJTI/Wall-E_v2.2
- ROS1 VS ROS2 : https://www.generationrobots.com/blog/en/ros-vs-ros2/ http://design.ros2.org/articles/changes.html