VEERMATA JIJABAI TECHNOLOGICAL INSTITUTE

Internship Duration

2 June 2023 to 31 July 2023

Report by:

Gaurav Kumar

201050053

Trajectory Genration and Tracing with a Non-Holonomic Drive robot

Design and implement a trajectory tracing controller (using PID) for a mecanum wheel robot that enables it to accurately follow a predefined trajectory .

Under the Guide :

Dr. Nagamanikandan Govindan

Robotics Research Centre

IIIT -Hyderabad

Contents

Acknowledgement

I would like to take this opportunity to express my heartfelt gratitude to all those who have supported me throughout my internship journey.

I am immensely grateful to Veermata Jijabai Technological Institute, Mumbai, for providing me with the opportunity to undertake this internship. I would like to extend my sincere appreciation to the Saad Hashmi and Tejal Bedmutha for their constant motivation and support.

My profound thanks go to my guide, Dr. Nagamanikandan sir and all my seniors for their invaluable mentorship, inspiration, and affection during my training. Their unwavering support and encouragement have been instrumental in shaping my career.

I also want to acknowledge the immense contribution of my professors, Prof. Dr. Suranjana Gangopadhyay, Asst. Prof. Aniket Gajbhiye, Prof. Arvind Bhongade (HOD Textile Engineering Department), and other staff members. They have been a constant source of support and have helped me to understand my capabilities and bring out my highest potential.

Lastly, I would like to express my appreciation to my friends for their direct and indirect help, which has been pivotal in helping me successfully complete my internship.

Sincerely,
Gaurav Kumar
Third Year

Abstract

This report presents a comprehensive study on trajectory generation and tracing techniques for a non-holonomic drive robot, specifically focusing on a mecanum wheel robot. The ability of a robot to accurately follow predefined trajectories is crucial for various applications, ranging from autonomous navigation in industrial settings to mobile robotics in complex environments. Achieving accurate trajectory tracing involves challenges posed by the inherent non-holonomic constraints of certain robot platforms, such as mecanum wheel robots.

The first focus of this study delves into the fundamental concepts of trajectory generation. Different trajectory representation methods, including parametric equations and waypoints, are examined in the context of non-holonomic constraints. The report then explores trajectory planning algorithms that take into account the kinematic and dynamic constraints of the mecanum wheel robot. These algorithms enable the generation of feasible trajectories that consider the robot's non-holonomic nature and its capability to move in any direction without changing its orientation.

The second focus of this report is on the implementation of a trajectory tracing controller using the Proportional-Integral-Derivative (PID) control scheme. PID control is a widely used technique for maintaining a desired trajectory by adjusting control inputs based on the error between the robot's current state and the desired trajectory. The paper discusses the design considerations for the PID controller in the context of mecanum wheel robots, highlighting the challenges posed by the robot's omnidirectional movement and the need to manage both translational and rotational velocities.

To validate the proposed concepts, a series of simulation and real-world experiments were conducted using a physical mecanum wheel robot. The results demonstrate the effectiveness of the trajectory generation techniques in accommodating non-holonomic constraints, and the PID-based trajectory tracing controller's ability to accurately follow predefined trajectories. The experiments also reveal insights into the controller tuning process and the trade-offs between accuracy, stability, and responsiveness in trajectory tracking.

Introduction

Design and implement a trajectory tracing controller for a **Mecanum Wheel Robot (ALOG)** that enables it to accurately follow a predefined trajectory consisting of waypoints **(Positions parameterised with Time)**. The controller should utilize odometry data to calculate control signals for the robot's linear and angular velocities using a PID controller. The objective is to ensure smooth and stable motion while accurately tracing the specified trajectory. Thorough testing and potential safety features should be incorporated to guarantee the controller's reliability and robustness in real or simulated robot environments.

The trajectory planner or generator, if necessary, will be responsible for creating feasible and smooth trajectories that the robot can follow without encountering any undesirable movements or jerks.Testing of the trajectory tracing controller will be conducted in simulated environment first and then on an actual mecanum wheel robot. Rigorous testing will ensure that the controller performs accurately, providing stable and precise motion following the predefined trajectory.

Ultimately, the successful implementation of this trajectory tracing controller will allow the mecanum wheel robot to autonomously follow complex paths with high accuracy, making it suitable for various applications, including automation, robotics research, and autonomous vehicles. The main objectives of the controller are as follows:

1. **Trajectory Following:** The mecanum drive robot should smoothly and precisely follow the specified trajectory, maintaining the desired positions at each waypoint in accordance to velocity profile .

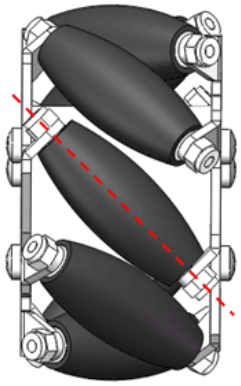
2. **Odometry-Based Control:** The controller should use odometry data to obtain the robot's current position (x, y) and orientation (yaw) to compute the necessary control inputs.
3. **PID Control:** Implement a PID control algorithm to calculate the linear and angular velocities required to steer the robot towards each waypoint accurately. The PID controller will use the error between the current

position and orientation and the desired position and orientation as feedback to adjust the robot's motion.

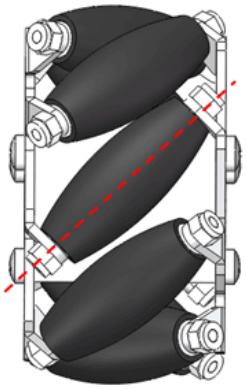
4. **Smooth Trajectory Generation:** If not already provided, generate smooth and feasible trajectories that connect the waypoints, avoiding abrupt changes in velocity and direction.
5. **Robustness and Safety:** Incorporate safety features to handle unexpected scenarios and ensure the robot's robust performance during trajectory tracing, preventing collisions and maintaining stability.
6. **Simulation and Real-World Testing:** Thoroughly test the trajectory tracing controller in simulated environments and, if applicable, on a physical mecanum drive robot to validate its accuracy and reliability.

Kinematic Modeling of 4 Wheeled Mecanum Drive

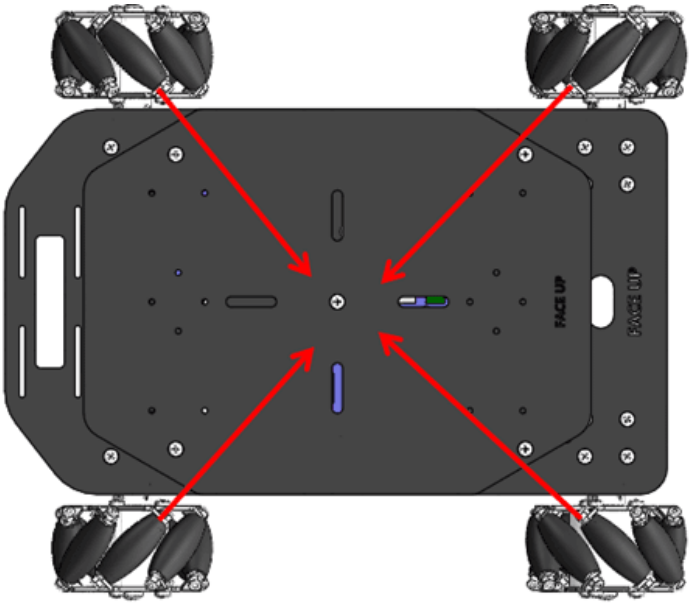
The Mecanum wheel is one design for a wheel which can move a vehicle in any direction. It is a conventional wheel with a series of rollers attached to its circumference. These rollers each have an axis of rotation at 45° to the plane of the wheel and at 45° to a line through the centre of the roller parallel to the axis of rotation of the wheel. There are two types of Mecunum wheels, left-handed and right-handed Mecunum wheel; the difference between them is the orientation of rollers. For left-handed Mecunum wheel, as shown below, rollers are orientated from lower right to upper left. Rollers for right-handed wheels are installed in the opposite way.



Left Mecanum Wheel

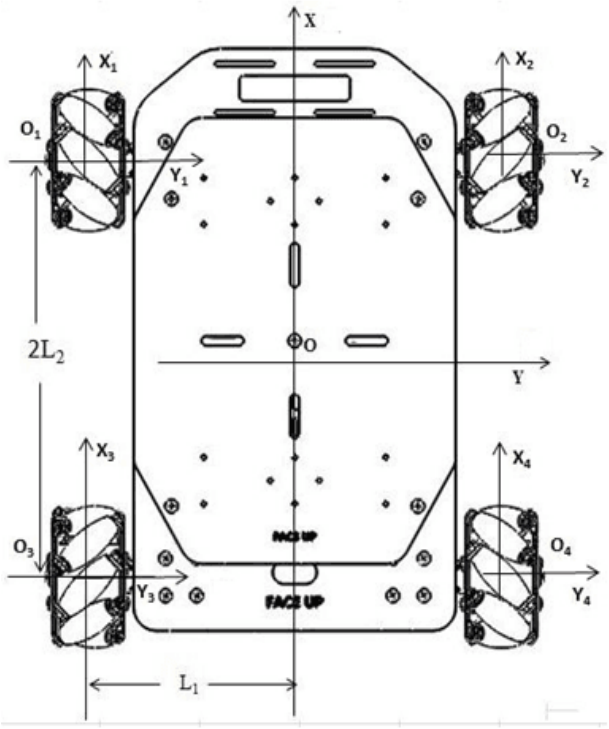


Right Mecanum Wheel

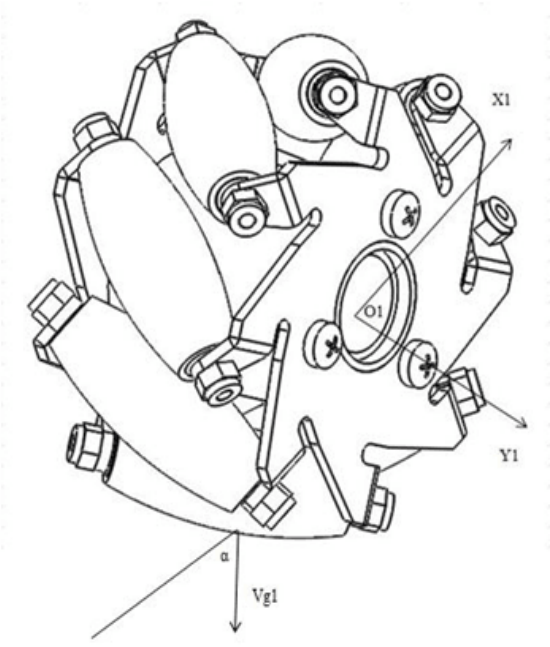


A correct configuration requires each of four wheels is set in the way as shown below, where the rotation axis of each wheel's top roller points to the center of the platform as shown in above figure . Please notice that all the dynamic analysis are based on this configuration.

The angled peripheral rollers translate a portion of the force in the rotational direction of the wheel to a force normal to the wheel direction. Depending on each individual wheel direction and speed, the resulting combination of all these forces produce a total force vector in any desired direction. Let the radius of the wheel to be R, the angular velocity of four wheels to be ω1, ω2, ω3, ω4, the speeds of rollers on each wheel to be v_g1, vg2, vg3 and vg4, and the speed of platform in x-direction, y direction and angular velocity to be vx, v_y, and ω0. The global coordinate origins at O, the center of the platform, and local coordinate systems at each wheel have origin of O1, O2, O3 and O4. The distance from the mid of platform to the mid of wheel is L1, and L2 is for the distance between mid of the platform to the wheel's axis of rolling. α is the angle of the rollers: 45º in this case.



Global Coordinate System



Local Coordinate System

❖ **Inverse kinematics Equations for Drive Control**

In the global coordinate, the speed at the center of wheel 1, O1, is

$$v_{o1x} = v_x - \omega_0 * L_1 \quad (1)$$

$$v_{o1y} = v_y - \omega_0 * L_2 \quad (2)$$

While in the local coordinate at wheel 1, the speed of O1 is

$$v_{o1x} = -v_{g1} * \cos\alpha + \omega_1 * R \quad (3)$$

$$v_{o1y} = v_{g1} * \sin\alpha \quad (4)$$

Combine equation (1) ~ (4), we have

$$v_x - \omega_0 * L_1 = -v_{g1} * \cos\alpha + \omega_1 * R \quad (5)$$

$$v_y - \omega_0 * L_2 = v_{g1} * \sin\alpha \quad (6)$$

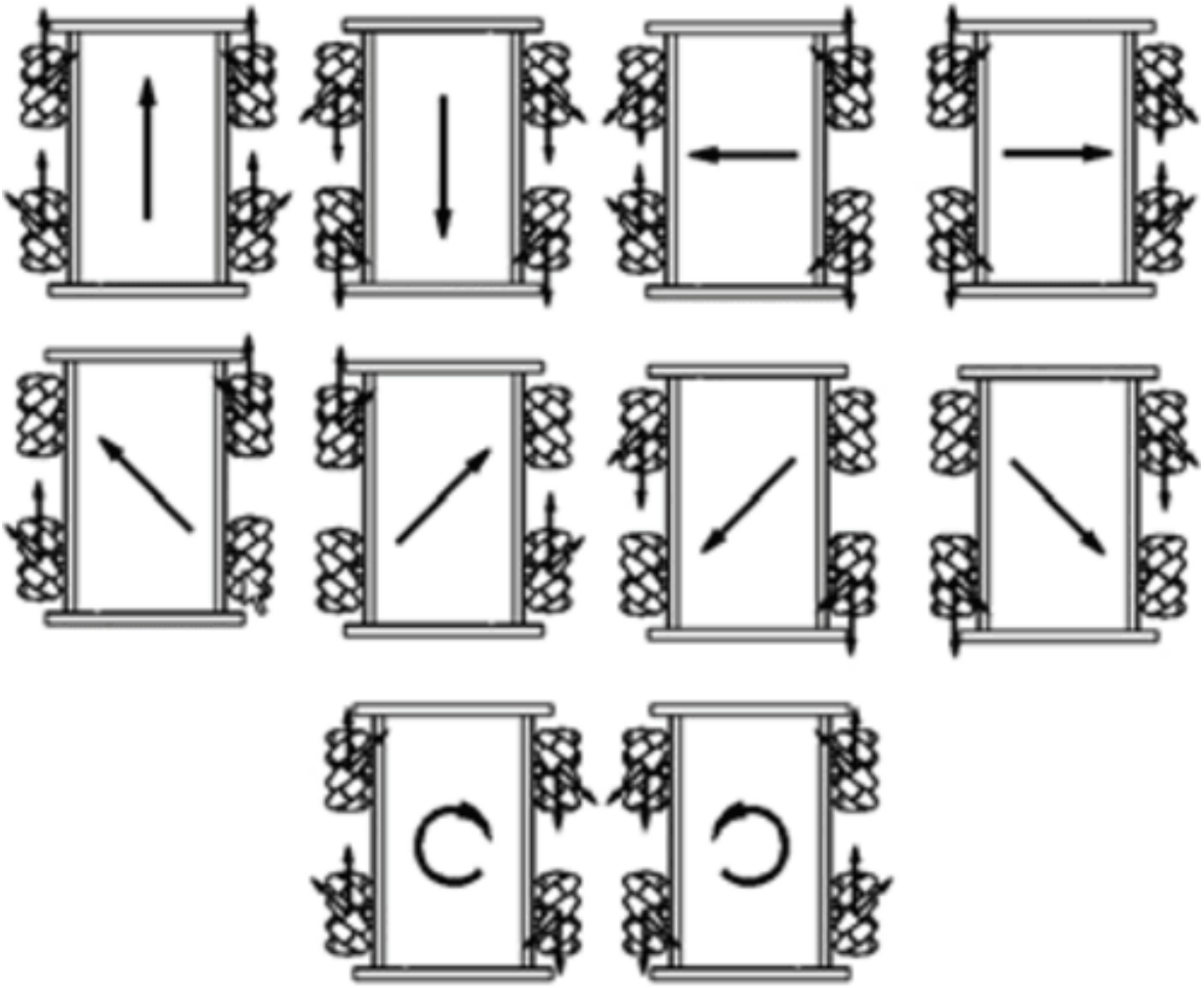
Solving (5) and (6), then the angular velocity of wheel 1 is

$$\omega_1 = \frac{1}{R}\begin{bmatrix} 1 & \frac{1}{\tan\alpha} & -(L_1 + \frac{L_2}{\tan\alpha}) \end{bmatrix}\begin{bmatrix} v_x \\ v_y \\ \omega_0 \end{bmatrix} \quad (7)$$

Similarly, the velocity of other 3 wheels can be calculated as

$$\begin{bmatrix} \omega_1 \\ \omega_2 \\ \omega_3 \\ \omega_4 \end{bmatrix} = \frac{1}{R}\begin{bmatrix} 1 & 1 & -(L_1 + L_2) \\ 1 & -1 & (L_1 + L_2) \\ 1 & -1 & -(L_1 + L_2) \\ 1 & 1 & (L_1 + L_2) \end{bmatrix}\begin{bmatrix} v_x \\ v_y \\ \omega_0 \end{bmatrix} \quad (8)$$

The equation (8) shows the relationship between the rotation speeds of the wheels and the movement of the platform. Theoretically, the platform can move in any direction by a proper combination of the four wheels'angular velocity. In fact, For this platform, The most common used movements are quite limited. Here we give out a simplified working principle of the platform. If you are disgusted with the numbers or equations, just ignore the dynamic analysis section, and read the figure below.



Mecanum Wheel Working Principle (Simplified)

Moving all four wheels in the same direction causes forward or backward movement, running the wheels on one side in the opposite direction to those on the other side causes rotation of the vehicle, and running the wheels on one

diagonal in the opposite direction to those on the other diagonal causes sideways movement.

❖ **Forward Kinematics Equations for Odometry calculations**

Mobile robot longitudinal velocity:

$$V_x(t) = (\omega_1 + \omega_2 + \omega_3 + \omega_4) \cdot \frac{R}{4}$$

Mobile robot transverse velocity:

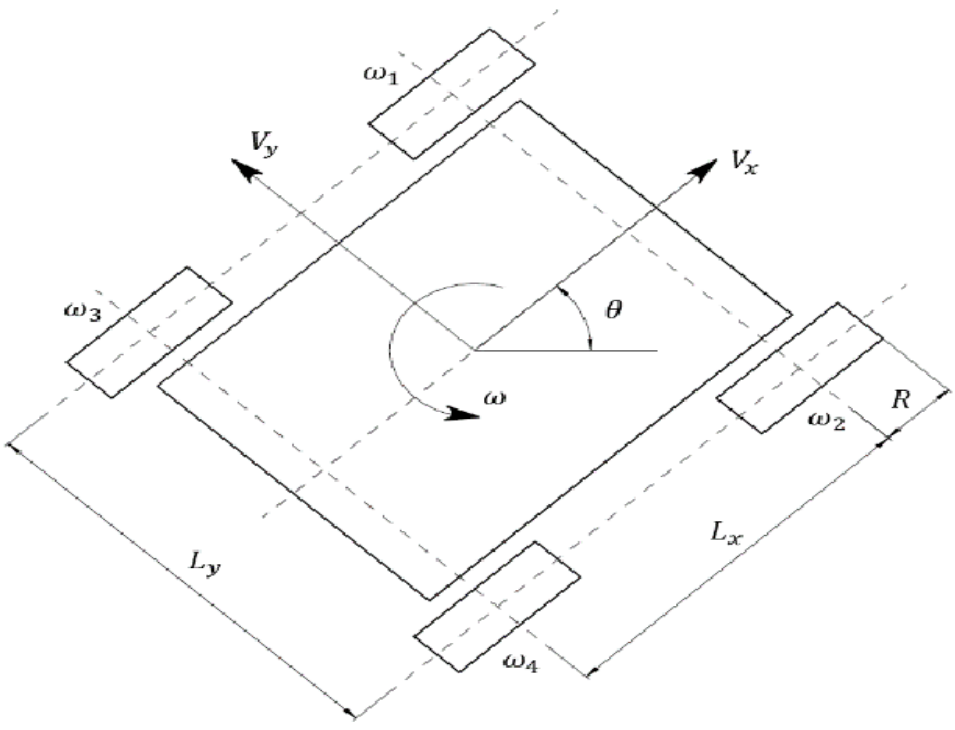
$$V_y(t) = (-\omega_1 + \omega_2 + \omega_3 - \omega_4) \cdot \frac{R}{4}$$

Mobile robot full velocity:

$$V_{full} = \sqrt{{V_x}^2 + {V_y}^2}$$

The direction of the mobile robot's motion vector:

$$\theta = \frac{1}{\tan\left(\frac{V_y}{V_x}\right)}$$



Trajectory generation

During robot motion, the robot controller is provided with a steady stream of goal positions and velocities to track. **This specification of the robot position as a function of time is called a trajectory.** In some cases, the trajectory is completely specified by the task – for example, the end-effector may be required to track a known moving object. In other cases, as when the task is simply to move from one position to another in a given time, we have freedom to design the trajectory to meet these constraints. This is the domain of trajectory planning. The trajectory should be a sufficiently smooth function of time, and it should respect any given limits on joint velocities, accelerations, or torques.

In this, we consider a trajectory as the combination of a **path**, a purely geometric description of the sequence of configurations achieved by the robot, and a **time scaling**, which specifies the times when those configurations are reached.

A path **θ(s)** maps a scalar path parameter s, assumed to be 0 at the start of the path and 1 at the end, to a point in the robot's configuration space Θ,

$$\theta : [0, 1] \rightarrow \Theta.$$

As s increases from 0 to 1, the robot moves along the path. Sometimes s is taken to be time and is allowed to vary from time s = 0 to the total motion time s = T , but it is often useful to separate the role of thegeometric path parameter s from the time parameter t. A time scaling **s(t)** assigns a value s to each time

$$t \in [0, T], s : [0, T] \rightarrow [0, 1].$$

$$\dot{\theta} = \frac{d\theta}{ds}\dot{s},$$
$$\ddot{\theta} = \frac{d\theta}{ds}\ddot{s} + \frac{d^2\theta}{ds^2}\dot{s}^2.$$

Together, a path and a time scaling define a trajectory θ(s(t)), or θ(t) for short.To ensure that the robot's acceleration (and therefore dynamics) is well defined,each of **θ(s)** and **s(t)** must be **twice differentiable**.

The planning modalities for trajectories may be quite different:

- **Point-to-Point**
- **With Pre-defined Path**

For planning a desired trajectory, it is necessary to specify two aspects:

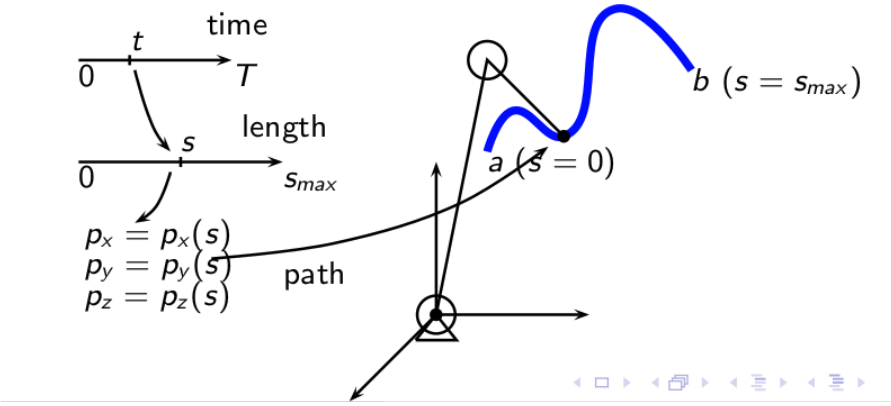
- **Geometric Path**
- **Motion Law**

with constraints on the continuity (smoothness) of the trajectory and on its time-derivatives up to a given degree.The geometric path can be defined in the work-space or in the joint-space. Usually, it is expressed in a parametric form as

p = p(s) work-space

q = q(σ) joint-space

The parameter s (σ) is defined as a function of time, and in this manner the motion law **s = s(t) (σ = σ(t))** is obtained.



Examples of geometric paths: (in the work space) linear, circular or parabolic segments or, more in general, tracts of analytical functions.

In the joint space, geometric paths are obtained by assigning initial/final (and, in case, also intermediate) values for the joint variables, along with the desired motion law. Concerning the motion law, it is necessary to specify continuos functions up to a given order of derivations (often at least first and second order, i.e. velocity and acceleration).Usually, polynomial functions a of proper degree n are employed:

$$s(t) = a_0 + a_1 t + a_2 t^2 + \ldots + a_n t^n$$

In this manner, a “smooth” interpolation of the points defining the geometric path is achieved.

The algorithm that computes a function q(t) interpolating the given points is characterized by the following features:

- Trajectories must be **computationally efficient**
- The position and velocity profiles (at least) must be **continuos functions of time**
- **Undesired effects** (such as non regular curvatures) must be minimized or completely avoided.

In the following discussion, a single joint is considered which can later be extrapolated to 2 Dimentions i.e X and Y.

In the most simple cases, (a segment of) a trajectory is specified by assigning **initial** and **final conditions on: time (duration)**, **position**, **velocity**,**acceleration, ...** Then, the problem is to determine a function

q = q(t)

so that those conditions are satisfied. This is a boundary condition problem, that can be easily solved by considering polynomial functions such as:

$$q(t) = a_0 + a_1 t + a_2 t^2 + \ldots + a_n t^n$$

The **degree n (3, 5, ...)** of the polynomial depends on the number of boundary conditions that must be verified and on the desired “smoothness” of the trajectory.

❖ Third-Order Polynomial Trajectories

Given an initial and a final instant t i , t f , a (segment of a) trajectory may be specified by assigning initial and final conditions:

- Initial position and velocity **q i** , **q̇ i**
- Final position and velocity **q f** , **q̇ f**

There are four boundary conditions, and therefore a polynomial of degree 3 (atleast) must be considered

$$q(t) = a_0 + a_1 t + a_2 t^2 + a_3 t^3 \qquad (1)$$

where the four parameters $a0$, $a1$, $a2$, $a3$ must be defined so that the boundary conditions are satisfied.From the boundary conditions, it follows that

$$\begin{aligned} q(t_i) &= a_0 + a_1 t_i + a_2 t_i^2 + a_3 t_i^3 &&= q_i \\ \dot{q}(t_i) &= a_1 + 2a_2 t_i + 3a_3 t_i^2 &&= \dot{q}_i \\ q(t_f) &= a_0 + a_1 t_f + a_2 t_f^2 + a_3 t_f^3 &&= q_f \\ \dot{q}(t_f) &= a_1 + 2a_2 t_f + 3a_3 t_f^2 &&= \dot{q}_f \end{aligned} \qquad (2)$$

In order to solve these equations, let us assume for the moment that t i = 0. Therefore:

$$a_0 = q_i \qquad (3)$$

$$a_1 = \dot{q}_i \qquad (4)$$

$$a_2 = \frac{-3(q_i - q_f) - (2\dot{q}_i + \dot{q}_f)t_f}{t_f^2} \qquad (5)$$

$$a_3 = \frac{2(q_i - q_f) + (\dot{q}_i + \dot{q}_f)t_f}{t_f^3} \qquad (6)$$

The results obtained with the polynomial (1) and the coefficients (3)-(6) can be generalized to the case in which $t_i \neq 0$ One obtains:

$$q(t) = a_0 + a_1(t - t_i) + a_2(t - t_i)^2 + a_3(t - t_i)^3 \qquad t_i \leq t \leq t_f$$

with coefficients

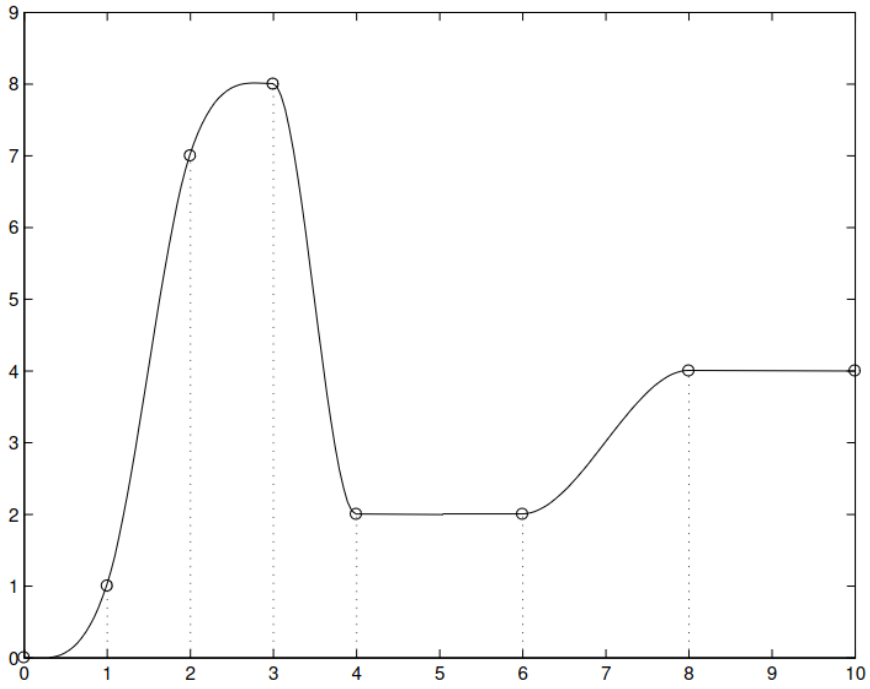
$$\begin{aligned} a_0 &= q_i \\ a_1 &= \dot{q}_i \\ a_2 &= \frac{-3(q_i - q_f) - (2\dot{q}_i + \dot{q}_f)(t_f - t_i)}{(t_f - t_i)^2} \\ a_3 &= \frac{2(q_i - q_f) + (\dot{q}_i + \dot{q}_f)(t_f - t_i)}{(t_f - t_i)^3} \end{aligned}$$

In this manner, it is very simple to plan a trajectory passing through a sequence of intermediate points.

The trajectory is divided in n segments, each of them defined by:

- Initial and Final point q_k and q_{k+1}
- Initial and Final instant t_k and t_{k+1}
- Initial and Final velocity $\dot{q}_k$ and $\dot{q}_{k+1}$

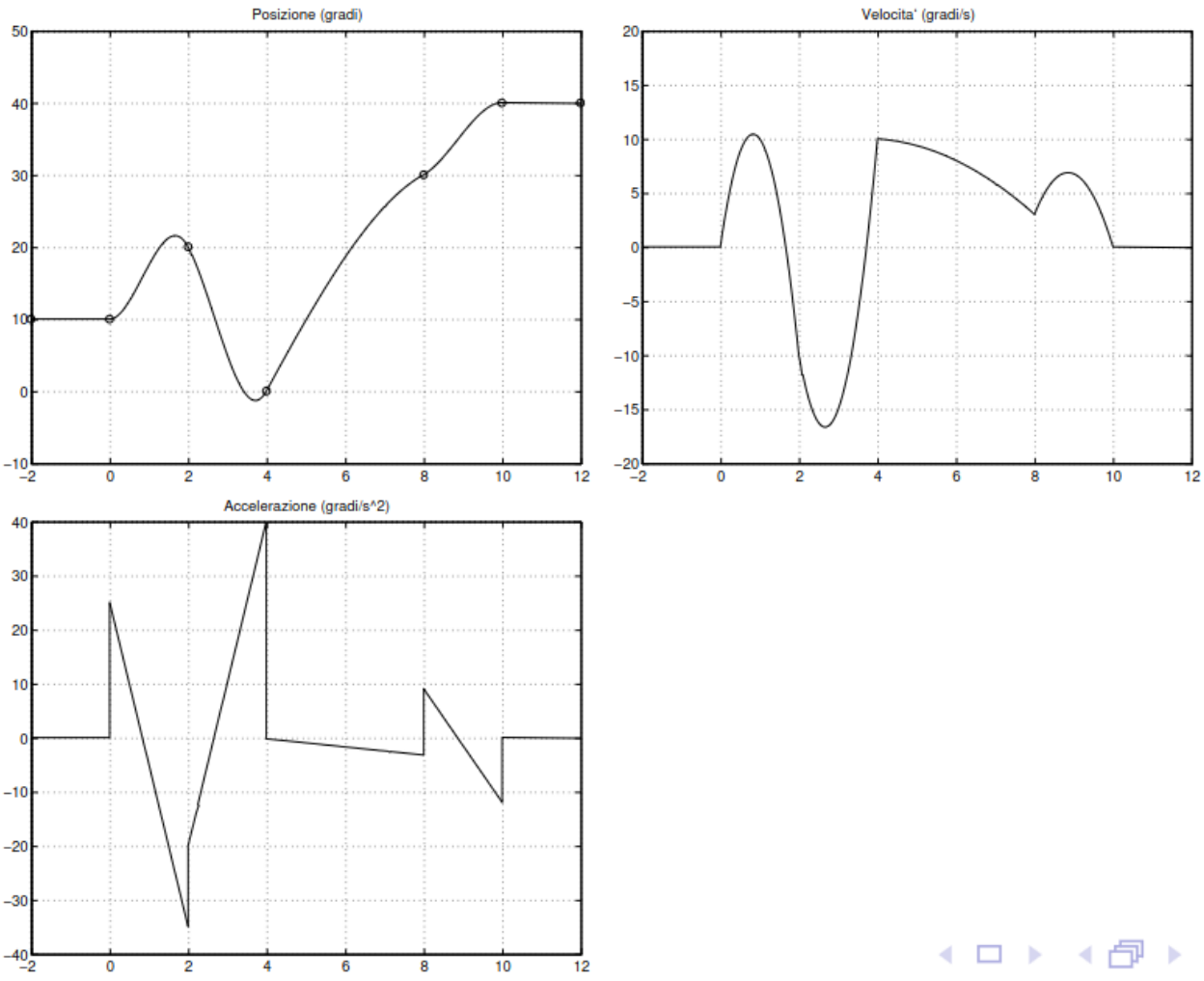
k = 0, . . . , n − 1.
The above relationships are then adopted for each of these segments.



For Example ,Position, velocity and acceleration profiles with:

$t_0 = 0$	$t_1 = 2$	$t_2 = 4$	$t_3 = 8$	$t_4 = 10$
$q_0 = 10^o$	$q_1 = 20^o$	$q_2 = 0^o$	$q_3 = 30^o$	$q_4 = 40^o$
$\dot{q}_0 = 0^o/s$	$\dot{q}_1 = -10^o/s$	$\dot{q}_2 = 20^o/s$	$\dot{q}_3 = 3^o/s$	$\dot{q}_4 = 0^o/s$

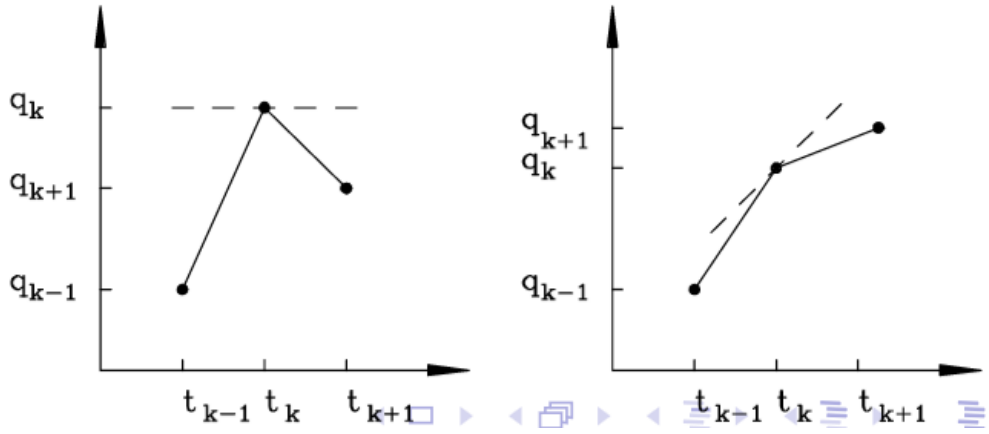
Graph for Position,Velocity and Acceleration Profiles are given below :



Often, a trajectory is assigned by specifying a sequence of desired points **(via-points)** without indication on the velocity in these points.In these cases, the "**most suitable**" values for the **velocities must be automatically computed.** This assignment is quite simple with heuristic rules such as:

$$
\begin{aligned}
\dot{q}_1 &= 0; \\
\dot{q}_k &= \begin{cases} 0 & \text{sign}(v_k) \neq \text{sign}(v_{k+1}) \\ \frac{1}{2}(v_k + v_{k+1}) & \text{sign}(v_k) = \text{sign}(v_{k+1}) \end{cases} \\
\dot{q}_n &= 0
\end{aligned}
$$

Being V_k the 'slope' of the tract [t_{k-1} – t_k].



From the above examples, it may be noticed that both the **position and velocity profiles are continuous functions of time**.This is **not true for the acceleration,** that presents therefore discontinuities among different segments. Moreover, it is not possible to specify for this signal suitable initial/final values in each segment. In many applications, these aspects do not constitute a problem, being the trajectories "smooth" enough.And to have more smooth trajectories we can go for higher polynomial interpolation functions like Quantic polynomial or for other higher polynomial functions.

There are many other methods for interpolating a set of via points. For example, **B-spline interpolatio**n is popular. In B-spline interpolation, the path may **not pass exactly through the via points**, but the path is guaranteed to be confined to the convex hull of the via points.This can be important to ensure that joint limits or workspace obstacles are respected.

❖ **Pre-defined Sinusoidal Trajectories**

So consider,

$$X(t) = A_1 cos(w_1 t + \phi_1)$$
$$Y(t) = A_2 sin(w_2 t + \phi_2)$$

These parametric equations represent a 2D curve that combines a sinusoidal motion along the x-axis and another sinusoidal motion along the y-axis. The shape of this curve depends on the values of the parameters A_1 , A_2 , w_1 , w_2 , ϕ_1 and ϕ_2.

- $X(t) = A_1 cos(w_1 t + \phi_1)$: This equation represents the x-coordinate of the points on the curve at time t. It is a cosine function that varies with time t, scaled by the amplitude A1, and shifted by the phase angle phi1.
- $Y(t) = A_2 sin(w_2 t + \phi_2)$: This equation represents the y-coordinate of the points on the curve at time t. It is a sine function that varies with time t, scaled by the amplitude A2, and shifted by the phase angle phi2.

By varying the values of A_1 , A_2 ,w_1 , w_2 , ϕ_1 and ϕ_2, you can create a wide range of 2D curves with different shapes, sizes, and patterns.
For Example:

1. Circle:
    - Equation:

$$X(t) = Acos(wt)$$
$$Y(t) = Asin(wt)$$

    - Shape: A circle with radius A.
2. Cycloid:
    - Equation:

$$X(t) = A(t - sin(t))$$
$$Y(t) = A(1 - cos(t))$$

    - Shape: The path traced by a point on the rim of a rolling circle.
3. Figure Eight:
    - Equation:

$$X(t) = Asin(wt)$$
$$Y(t) = Asin(2wt)$$

    - Shape: A double loop resembling the number "8".
4. Ellipse:
    - Equation:

$$X(t) = A_1 cos(wt)$$
$$Y(t) = A_2 sin(wt)$$

    - Shape: An elongated or squished circle.
5. Spiral:

- Equation:

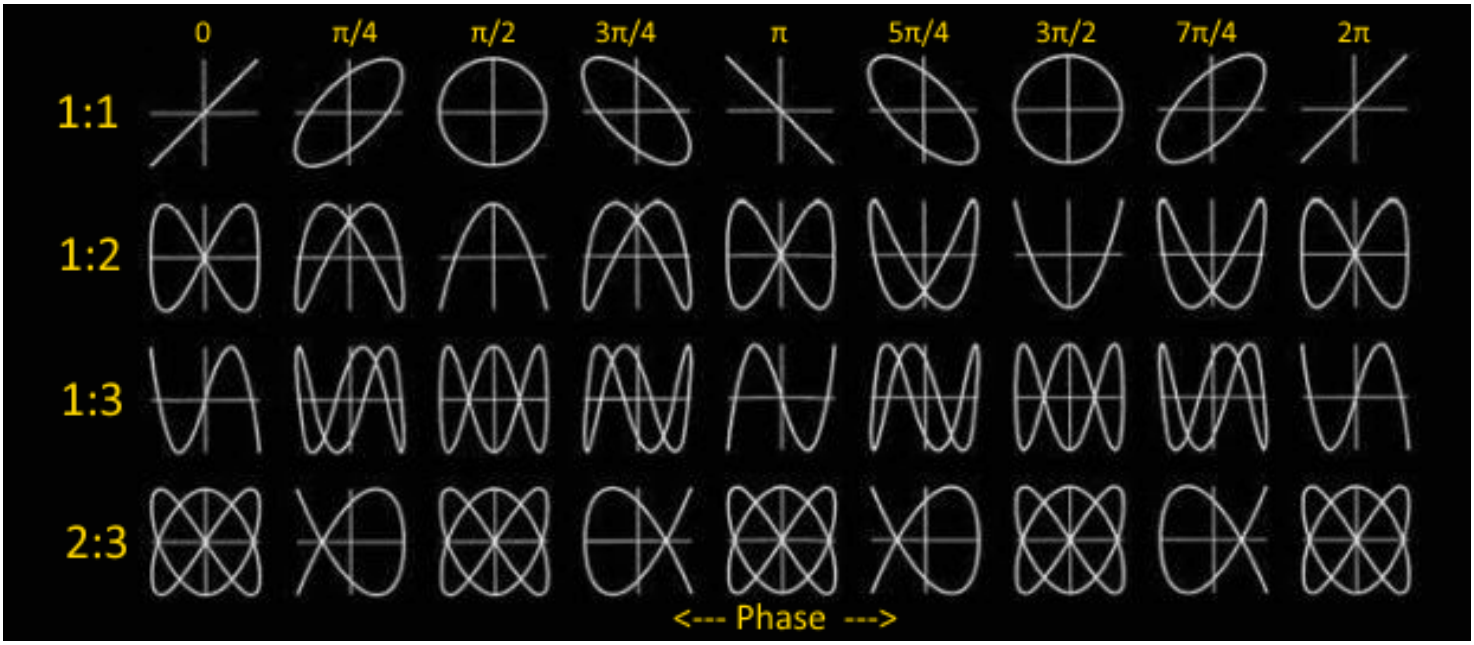
$$X(t) = A_1 cos(w_1 t + \phi_1)t$$
$$Y(t) = A_2 sin(w_2 t + \phi_2)t$$

- Shape: A spiral trajectory that grows or shrinks over time.

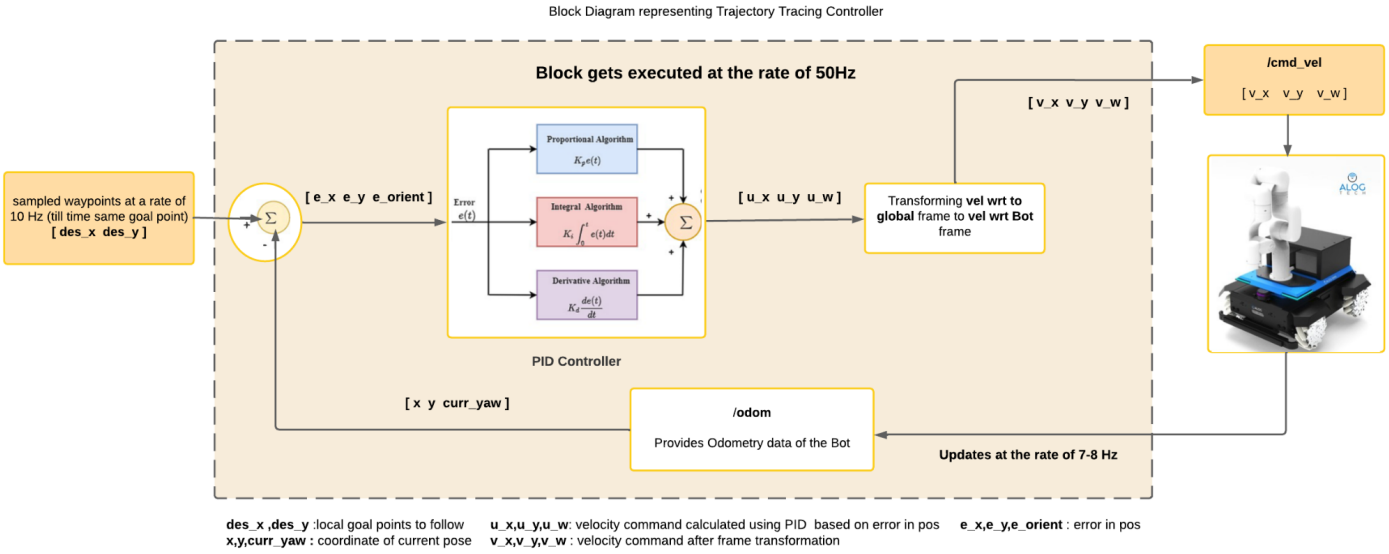
6. Lissajous Curve:
   - Equation:

$$X(t) = A_1 cos(w_1 t + \phi_1)$$
$$Y(t) = A_2 sin(w_2 t + \phi_2)$$

   - Shape: A complex pattern formed by the interaction of two perpendicular harmonic motions.

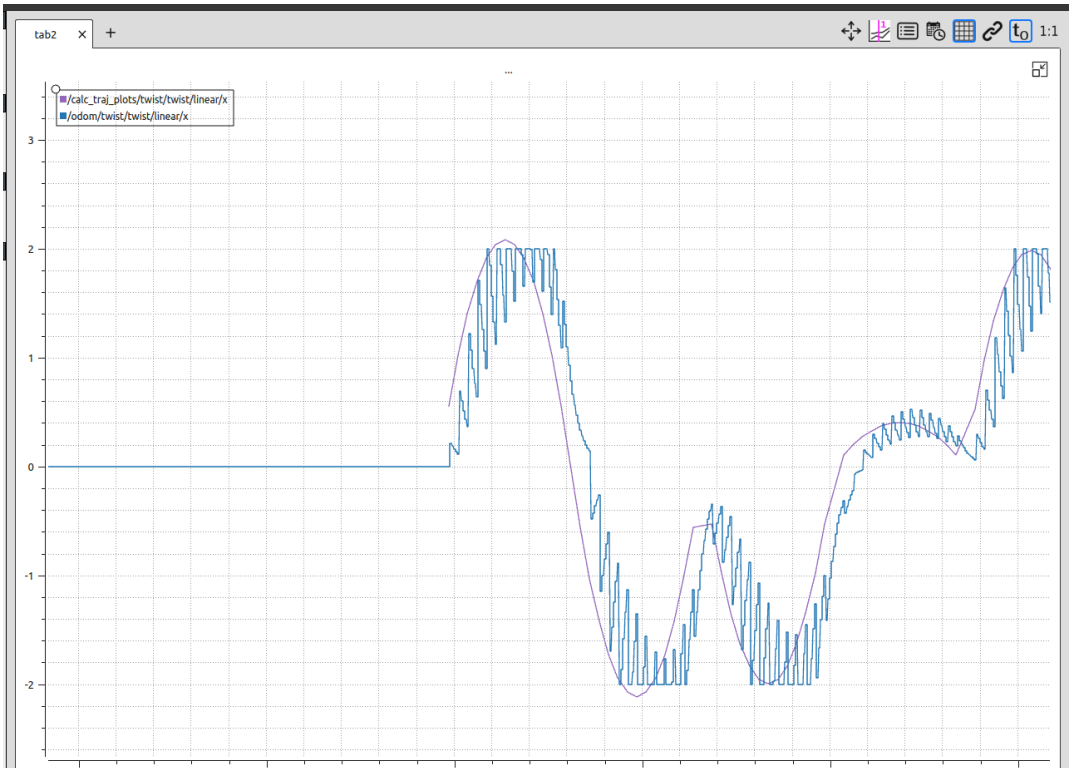


Controller Implementation

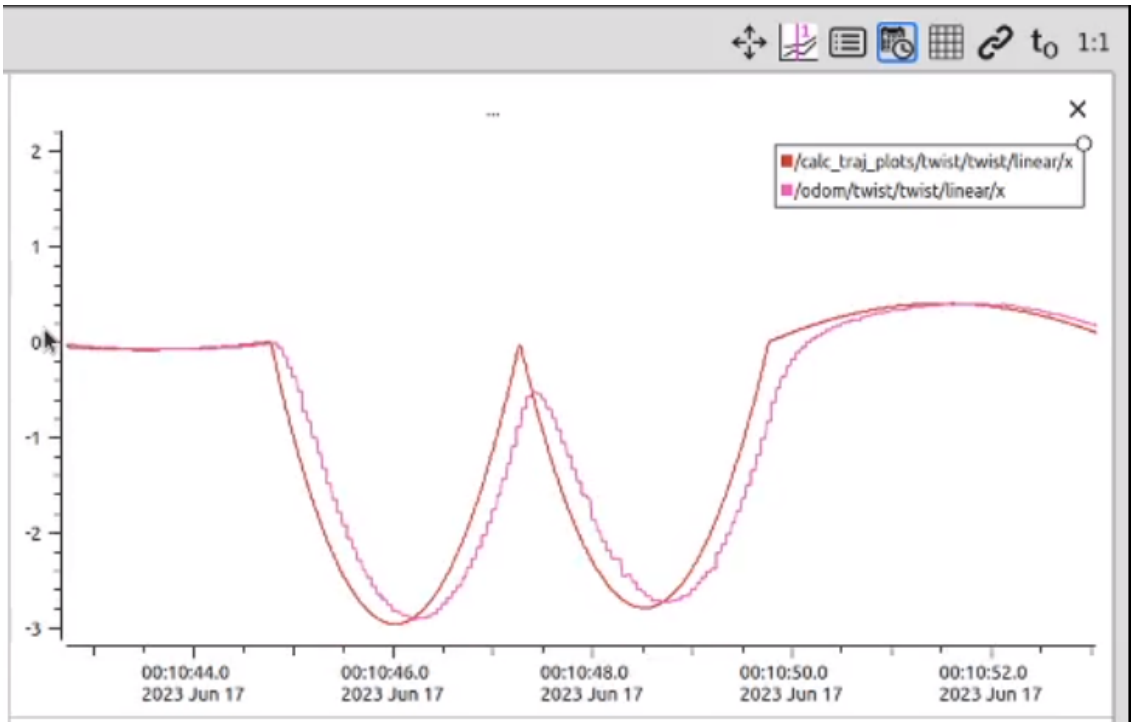


The presented block diagram illustrates the real-world implementation of the controller on both hardware and simulation platforms for a Mecanum drive robot. The rates mentioned in the diagram are specifically tailored for the hardware implementation. However, in the simulation environment, these calculations are performed at significantly higher rates, aiming for faster processing.

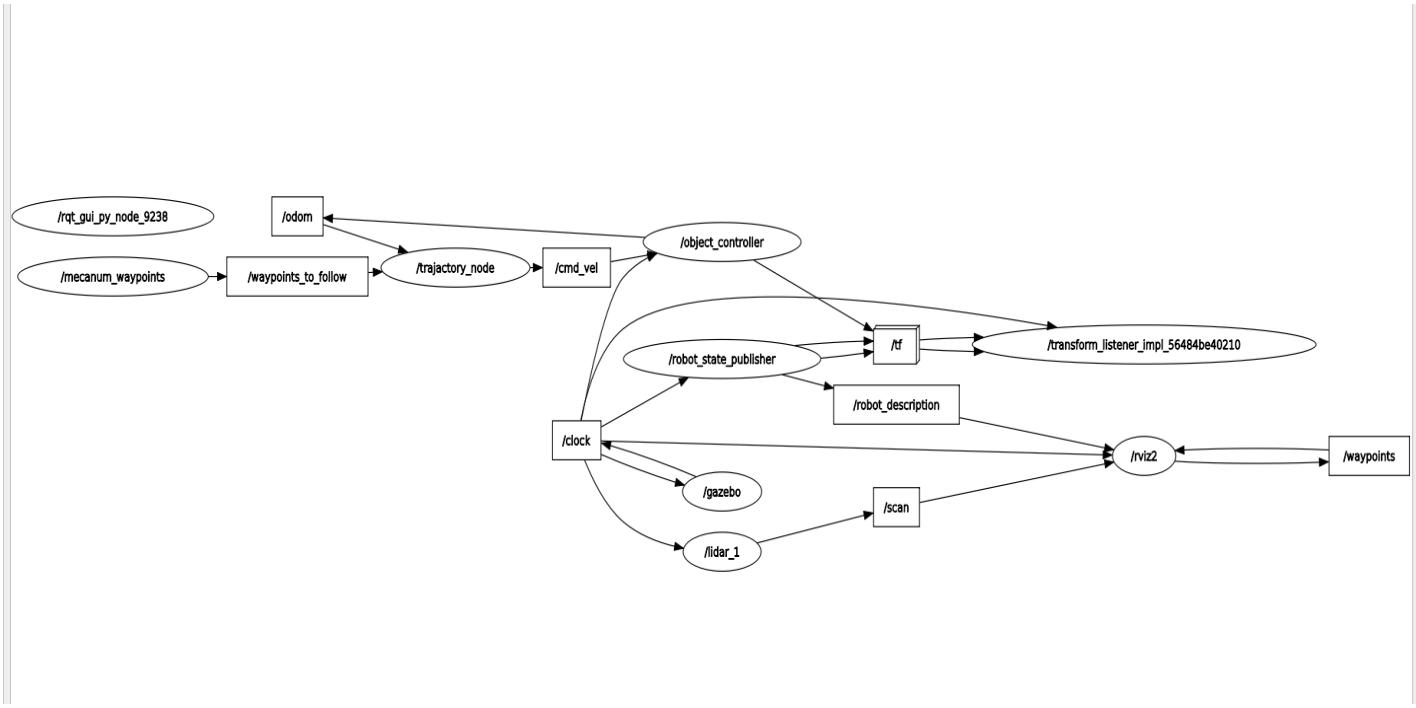
It is important to note that the hardware implementation is constrained by the limitations of the physical sensors and the actuation speed of the robot's motors, resulting in lower rates compared to simulation. The higher rates in simulation are chosen to achieve optimal and smooth tracing of the velocity profile, as well as to minimize the lag between the actual position of the robot and its expected position, which are defined by the trajectory constraints. These higher rates in simulation enable more precise control and provide better accuracy in following the desired trajectory.



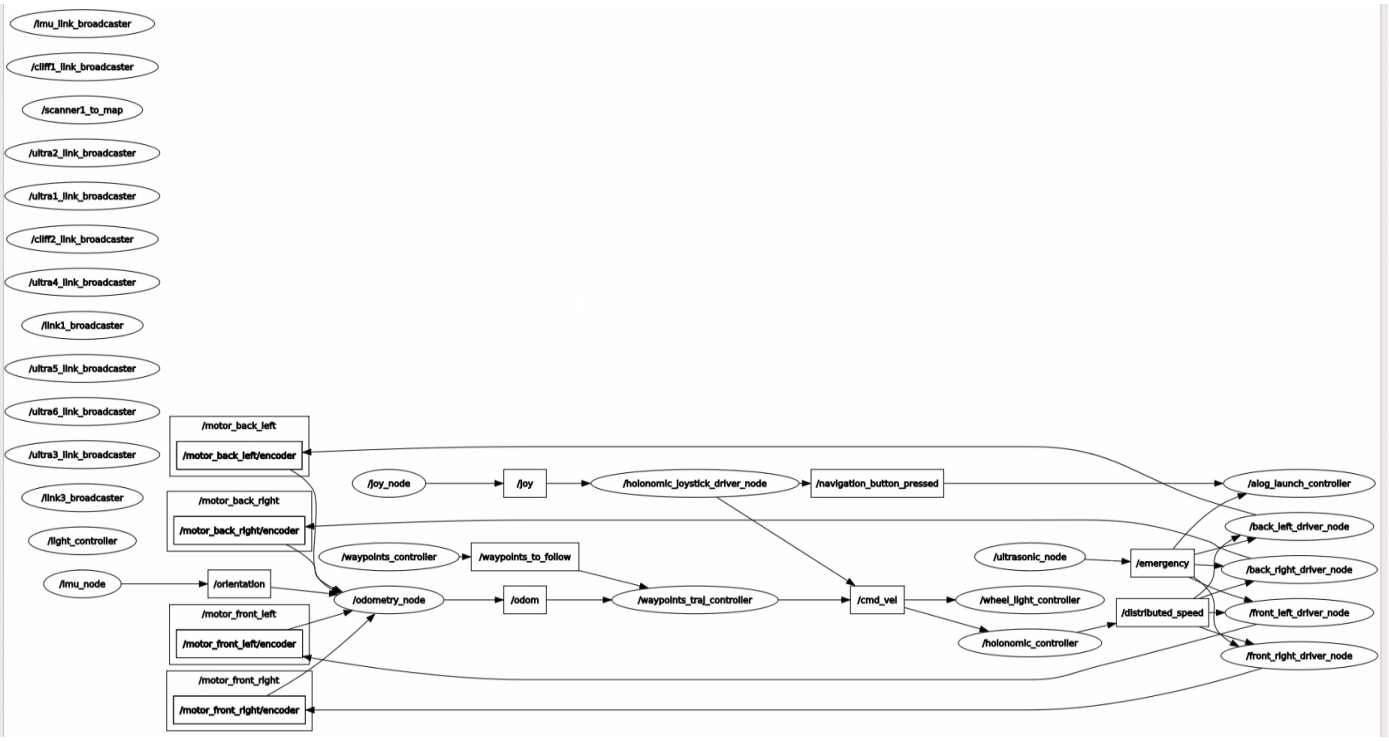
Velocity in x direction with lower sample rates



Velocity in x direction with higher sample rates



rqt graph showing the simulation implementation of controller



rqt graph showing the hardware implementation of controller

- In Simulation part , **/mecanum_waypoints** is the node which is responsible for trajectory generation and sampling the points i.e des_x and des_y ,see block diagram for clarity .And **/trajectory_node** is the node which is performing all the calculations inside the block represented in dashed lines.For more details on the nodes check Codebase Description section.
- In Hardware part , **/waypoints_controller** is the node which is responsible for trajectory generation and sampling the points i.e des_x and des_y ,see block diagram for clarity .And **/waypoints_traj_controller** is the node which is performing all the calculations inside the block represented in dashed lines.For more details on the nodes check Codebase Description section.

Codebase Description

❖ Simulation and Experimenting with Gazebo :

For simulation we used a ROS2 FOXY (supports Python 3 and higher versions) and Gazebo version 11 . 11 which requires Ubuntu 20.04 LTS . Details of package used is provided below-

simulation_ws

1. build
2. install
3. log
4. src :
    1) mecanum_drive : Package that has all the files for control of mecanum drive i.e all controller and visualisation related files.
        - ➔ launch :
            - control.launch.py
            - go_to_goal.launch.py
        - ➔ mecanum_drive :
            - __init__.py
            - mecanum_controller.py
            - mecanum_traj_controller.py
            - mecanum_waypoints_param.py
            - mecanum_waypoints.py
            - quat_to_euler.py
        - ➔ resource
        - ➔ test
        - ➔ package.xml : Make sure you have added(above <test_depend> tag) below mentioned in package.xml of package.

```
<depend>example_interfaces<depend/>
 <depend>numpy<depend/>
 <depend>math<depend/>
 <depend>nav_msgs<depend/>
```

        - ➔ setup.cfg
        - ➔ setup.py : Make sure your setup.py should look like the below code snippt .

```
from setuptools import setup
import os
from glob import glob
package_name = 'mecanum_drive'
```

```
setup(
   name=package_name,
   version='0.0.0',
   packages=[package_name],
   data_files=[

('share/ament_index/resource_index/packages',
           ['resource/' + package_name]),
       ('share/' + package_name, ['package.xml']),
       (os.path.join('share', package_name),
glob('launch/*.launch.py'))
   ],
   install_requires=['setuptools'],
   zip_safe=True,
   maintainer='manish',
   maintainer_email='manish@todo.todo',
   description='TODO: Package description',
   license='TODO: License declaration',
   tests_require=['pytest'],
   entry_points={
       'console_scripts': [
       "mecanum_controller =
mecanum_drive.mecanum_controller:main",
       "quat_to_euler =
mecanum_drive.quat_to_euler:main",
       "mecanum_controller_new =
mecanum_drive.mecanum_controller_new:main",
       "mecanum_waypoints =
mecanum_drive.mecanum_waypoints:main",
       "mecanum_traj_controller =
mecanum_drive.mecanum_traj_controller:main",
       "mecanum_waypoints_param =
mecanum_drive.mecanum_waypoints_param:main"
       ],
   },
)
```

2) neo_simulation2 : Package that has simulation,rviz visuallisation and model description files ,mesh files all that stuff necessary to represent our mecanum bot .

- ➔ components:
    - sensors
- ➔ configs :
    1) mp_400
        - mapping.yaml
        - navigation.yaml
    2) mp_500
        - mapping.yaml
        - navigation.yaml
    3) mpo_500
        - mapping.yaml
        - navigation.yaml
    4) mpo_700
        - mapping.yaml
        - navigation.yaml
- ➔ launch :
    - mapping.launch.py
    - navigation.launch.py
    - simulation.launch.py
- ➔ maps :
    - aws.pgm
    - aws.yaml
    - neo_track1.pgm
    - neo_trac1.yaml
    - neo_workshop.pgm
    - neo_workshop.yaml
- ➔ models : contains stl files of sensors used and models used to build the world.
- ➔ neo_simulation2
- ➔ resource
- ➔ robots :
    1) mp_400
        - meshes
        - mp_400.urdf
    2) mp_500
        - meshes
        - mp_500.urdf
    3) mpo_500
        - meshes
        - mpo_500.urdf : This file contains the description of a mecnum drive robot with which we are concerned.It contains declaration of all plugins we have used.For mecanum drive we have used planar_move_plugin which is used to make a

planer objects move by providing velocities directly in terms of vx,vy and w unlike other plugins which commands actuations to the wheels and then drive moves.We did so because there is no std plugin available for it.

4) mpo_700
    - meshes
    - mpo_700.urdf

➔ test

➔ worlds
- aws.world
- empty_world.world
- neo_track1.world
- neo_track2.world
- neo_workshop.world

➔ CMakeLists.txt : Your CMake file should be like these

```
cmake_minimum_required(VERSION 3.5)
project(neo_simulation2)

find_package(ament_cmake REQUIRED)
find_package(nav2_bringup REQUIRED)
find_package(navigation2 REQUIRED)

install(DIRECTORY launch
 configs
 maps
 worlds
 robots
 components
 models
 DESTINATION share/${PROJECT_NAME})

ament_package()
```

➔ package.xml : Make sure you have added below code in <export> tag of package.xml of package.

```
<build_depend>nav2_bringup</build_depend>
<export>
   <build_type>ament_cmake</build_type>
```

```
  <gazebo_ros gazebo_model_path="${prefix}/.." />
  <gazebo_ros gazebo_model_path="${prefix}/models"
/>
 </export>
```

And <buildtool_depend> tag should have

```
<buildtool_depend>ament_cmake</buildtool_depend>
```

➔ README.md
➔ setup.cfg
➔ setup.py

- ***mecanum_controller.py*** :

The provided Python code represents a ROS (Robot Operating System) node for controlling a mecanum-wheeled robot a basic go to goal algorithm. The node subscribes to two topics, "/orient_euler" and "/odom", to receive information about the robot's orientation and odometry data, respectively. It also publishes commands to the "/cmd_vel" topic to control the robot's motion.

Description of the code:

1. Importing necessary libraries and ROS messages:
    - The script starts by importing required libraries, including rclpy for creating the ROS node, and various ROS message types like Twist, Odometry, Vector3, and Point.
2. The `ControlNode` class:
    - The main functionality is encapsulated in the `ControlNode` class, which is derived from the `rclpy.node.Node` class. This class serves as the ROS node for controlling the mecanum-wheeled robot.
    - It initializes various control parameters and the desired path for the robot to follow.
    - The `get_orient` method is a callback function that receives the robot's current yaw orientation in radians from the "**/orient_euler**" topic.
    - The `turtlebot3_callback` method is a callback function that receives odometry data from the "**/odom**" topic and performs control calculations to guide the robot along the specified path.
3. Control algorithm:
    - The control algorithm used here is a combination of position control and orientation control.
    - The desired position (x, y) is taken from the predefined path, and the difference between the desired and current positions (e_x and e_y) is computed.

    - The difference between the desired orientation (based on the path) and the current orientation (yaw) is computed as e_orient.
    - The controller uses **Proportional-Integral-Derivative (PID)** control for both orientation and position control.
    - The computed control signals (u_w, u_x, and u_y) are adjusted to keep them within specific limits.
    - The control signals are then transformed from the global frame to the body frame of the robot (v_x and v_y) based on the current yaw angle.
    - The final control commands for the robot's linear and angular velocities are published to the "**/cmd_vel**" topic.
4. Main function:
    - The main function initializes the ROS node, creates an instance of the `ControlNode`, and enters the ROS spin loop to process incoming messages and publish control commands.

To summarize, the provided code implements a ROS node to control a mecanum-wheeled robot and guides it along a predefined path using a **PID-based control algorithm** for both **position and orientation control**. The robot's motion commands are published to the "**/cmd_vel**" topic, allowing other nodes or the robot itself to execute the motion.

- ***quat_to_euler.py*** :

The provided Python code represents another ROS node responsible for converting quaternion orientation information from the "**/odom**" topic into Euler angles (roll, pitch, and yaw) and then publishing the Euler angles to the "**/orient_euler**" topic.

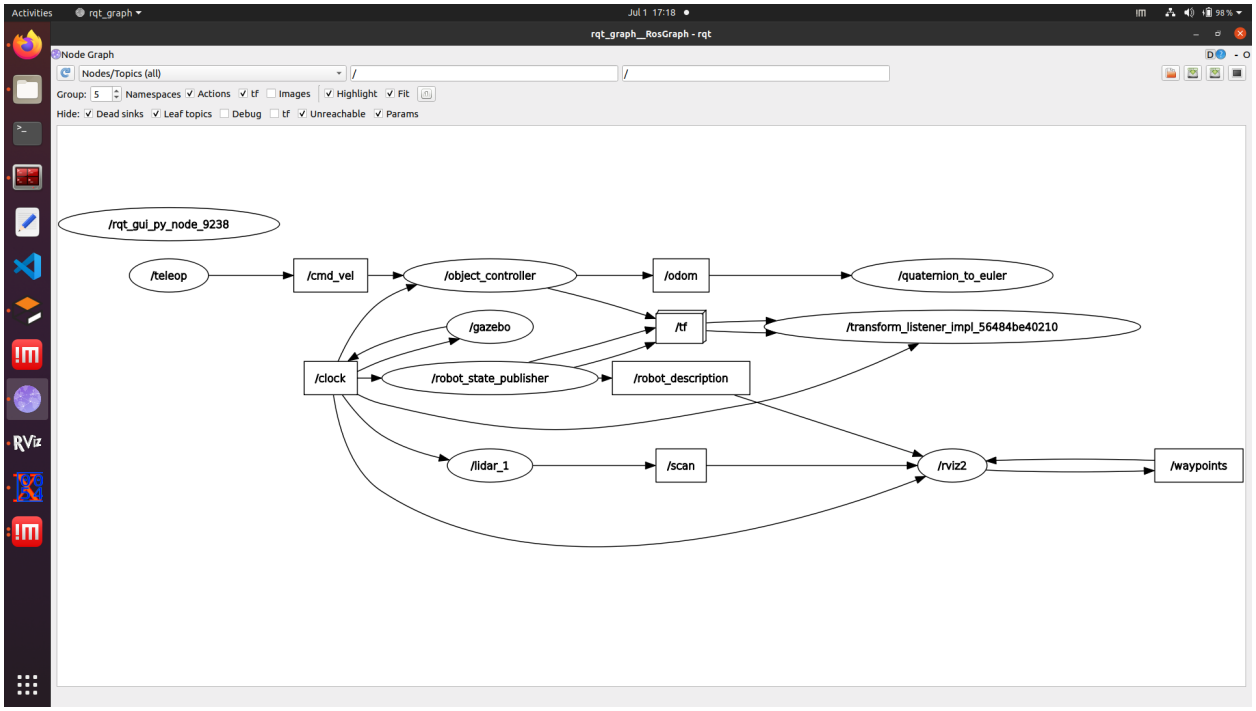
Description of the code:

1. Importing necessary libraries and ROS messages:
    - The script starts by importing required libraries, including rclpy for creating the ROS node, and various ROS message types like Vector3 and Odometry.
2. The `QuaternionToEulerNode` class:
    - The main functionality is encapsulated in the `QuaternionToEulerNode` class, which is derived from the `rclpy.node.Node` class. This class serves as the ROS node for **converting quaternion to Euler angles** and publishing the Euler angles.
    - It initializes a publisher for publishing the Euler angles to the "**/orient_euler**" topic and a subscriber for receiving odometry data from the "**/odom**" topic.
    - The `callback_get_eular` method is a callback function that receives odometry data containing the quaternion orientation from the "**/odom**" topic.
    - It then performs the conversion from quaternion to Euler angles (roll, pitch, and yaw) using mathematical formulas.

- The Euler angles are then packaged into a `Vector3` message and published to the "**/orient_euler**" topic.

3. Quaternion to Euler conversion:
   - The code implements the conversion from quaternion to Euler angles using the formulas described in the comments.
   - The formulas for roll (roll_x), pitch (pitch_y), and yaw (yaw_z) are derived from the **quaternion components (x, y, z, w)** provided by the "**/odom**" topic.
   - The calculated Euler angles are then published to the "**/orient_euler**" topic.
4. Main function:
   - The main function initializes the ROS node, creates an instance of the `QuaternionToEulerNode`, and enters the ROS spin loop to process incoming odometry data and publish the Euler angles.

To summarize, the provided code implements a ROS node responsible for converting quaternion orientation information from the "**/odom**" topic into Euler angles and publishing them to the "**/orient_euler**" topic. This node can be useful for visualizing the orientation of the robot in the Euler angle representation, which is more human-readable and easier to work with than quaternions.



rqt_graph after runing quat_to_euler.py file

- ***mecanum_waypoints.py :***

The provided Python code represents another ROS node responsible for generating polynomial trajectories between waypoints and publishing the trajectory points to the

"**/waypoints_to_follow**" topic. Additionally, it publishes some trajectory-related data to the "**/calc_traj_plots**" topic for **visualization purposes**.

Description of the code:

1. Importing necessary libraries and ROS messages:
    - The script starts by importing required libraries, including rclpy for creating the ROS node, and various ROS message types like Vector3 and Odometry.
2. The `WaypointsNode` class:
    - The main functionality is encapsulated in the `WaypointsNode` class, which is derived from the `rclpy.node.Node` class. This class serves as the ROS node for generating polynomial trajectories between waypoints and publishing trajectory points.
    - It initializes publishers for publishing the waypoints trajectory points to the "**/waypoints_to_follow**" topic and for publishing some trajectory-related data to the "**/calc_traj_plots**" topic for visualization.
    - It also sets some initial values and creates a timer to execute the `publish_waypoints_method` at a regular interval.
3. `publish_waypoints_method`:
    - This method generates and publishes trajectory points for the mecanum robot to follow. It **implements polynomial trajectory planning between waypoints**.
    - The waypoints are defined as `[x, y, T]` where (x, y) is the goal position, and T is the desired time to reach that goal.
    - The method iterates through each waypoint and generates a 3rd-order polynomial trajectory to smoothly move from the current waypoint to the next waypoint.
    - It calculates the trajectory points for a specific time interval and publishes them to the "**/waypoints_to_follow**" topic.
    - Additionally, it updates the robot's position and velocity data in an Odometry message and publishes this message to the "**/calc_traj_plots**" topic for visualization purposes.
4. `move_to_point`:
    - This method is used to generate the polynomial trajectory between two waypoints. It **calculates the coefficients of the 3rd-order polynomial** for both the x and y axes using the given start and end points, start and end velocities, and the **desired time (T)** to complete the trajectory.
5. Main function:
    - The main function initializes the ROS node, creates an instance of the `WaypointsNode`, and enters the ROS spin loop to continuously execute the `publish_waypoints_method`.

To summarize, the provided code implements a ROS node responsible for generating and publishing polynomial trajectories between waypoints for a mecanum robot to follow. The

trajectory points are published to the "**/waypoints_to_follow**" topic, and trajectory-related data is published to the "**/calc_traj_plots**" topic for visualization purposes. This node allows the robot to smoothly move between predefined waypoints using polynomial trajectory planning.

- ***mecanum_waypoints_param.py :***

The provided Python code represents a ROS node that **generates a sinusoidal trajectory for a mecanum robot to follow**. The trajectory is **defined in polar coordinates**, and the resulting (x, y) position and velocity values are published to the "**/waypoints_to_follow**" topic. Additionally, trajectory-related data is published to the "**/calc_traj_plots**" topic for visualization purposes.

Description of the code:

1. Importing necessary libraries and ROS messages:
    - The script starts by importing required libraries, including rclpy for creating the ROS node, and various ROS message types like Vector3 and Odometry.
2. The `WaypointsNode` class:
    - The main functionality is encapsulated in the `WaypointsNode` class, which is derived from the `rclpy.node.Node` class. This class serves as the ROS node for generating the sinusoidal trajectory and publishing the trajectory points.
    - It initializes publishers for publishing the trajectory points to the "**/waypoints_to_follow**" topic and for publishing some trajectory-related data to the **"/calc_traj_plots**" topic for visualization.
    - It also sets some initial values and creates a timer to execute the `publish_waypoints_method` at a regular interval.
3. `publish_waypoints_method`:
    - This method generates and publishes the sinusoidal trajectory points for the mecanum robot to follow. The trajectory is defined in polar coordinates as:
        - x_t = A1 * cos(omega1 * t + phi1)
        - y_t = A2 * sin(omega2 * t + phi2)
        - x_t_dot = -A1 * omega1 * sin(omega1 * t + phi1)
        - y_t_dot = A2 * omega2 * cos(omega2 * t + phi2)
    - The method samples the trajectory points in a specific time interval (T) and publishes them to the "**/waypoints_to_follow**" topic.Last sampled point will be published till the time new point is sampled according to T value specified.
    - Additionally, it updates the robot's position and velocity data in an Odometry message and publishes this message to the "**/calc_traj_plots**" topic for **visualization purposes**.
4. Main function:

- The main function initializes the ROS node, creates an instance of the `WaypointsNode`, and enters the ROS spin loop to continuously execute the `publish_waypoints_method`.

To summarize, the provided code implements a ROS node responsible for generating and publishing a sinusoidal trajectory for a mecanum robot to follow. The trajectory points are published to the "**/waypoints_to_follow**" topic, and trajectory-related data is published to the "**/calc_traj_plots**" topic for visualization purposes. This node allows the robot to follow a **sinusoidal path** as defined by the polar coordinates **(A1, omega1, phi1)** and **(A2, omega2, phi2)**.

- ***mecanum_traj_controller.py :***

The provided Python code represents a ROS node responsible for controlling the motion of a mecanum robot using a PID controller to follow a given trajectory defined by continuous stream of local waypoints available via **"/waypoints_to_follow"** topic.

Description of the code:

1. Importing necessary libraries and ROS messages:
   - The script starts by importing required libraries, including rclpy for creating the ROS node, and various ROS message types like Twist, Vector3, and Odometry.
2. The `TrajectoryNode` class:
   - The main functionality is encapsulated in the `TrajectoryNode` class, which is derived from the `rclpy.node.Node` class. This class serves as the ROS node for controlling the mecanum robot's motion based on a **PID controller**.
   - It initializes various variables and gains related to the PID controller and sets up publishers and subscribers.
3. The `quat_to_euler` method:
   - This method converts the quaternion orientation of the robot (received from the "**/odom**" topic) to Euler angles, specifically the yaw angle (in radians). The yaw angle is essential for orientation control.
4. The `odometry_data_callback` method:
   - This method gets executed every time the "**/odom**" topic is updated, and it updates the current position and velocity of the robot.
5. The `go_to_goal` method:
   - This method is called every time a new waypoint is received on the "**/waypoints_to_follow**" topic. It calculates the errors in position and orientation between the current robot pose and the desired waypoint.
   - A PID controller is used to calculate the control signals (linear and angular velocities) needed to reach the desired waypoint. The PID gains are defined as class attributes.

    - The calculated velocities are transformed from the global frame to the body frame and published to the "**/cmd_vel**" topic, which controls the robot's motion.
6. Main function:
    - The main function initializes the ROS node, creates an instance of the `TrajectoryNode`, and enters the ROS spin loop to continuously execute the control logic.

Overall, the provided code demonstrates how to control a mecanum robot using a PID controller to follow a trajectory defined by waypoints. The robot's motion is controlled based on the **errors in position and orientation with respect to the desired waypoints**, and the calculated control signals are published to the "**/cmd_vel**" topic. The PID gains can be adjusted to achieve the desired performance of the robot's motion control.

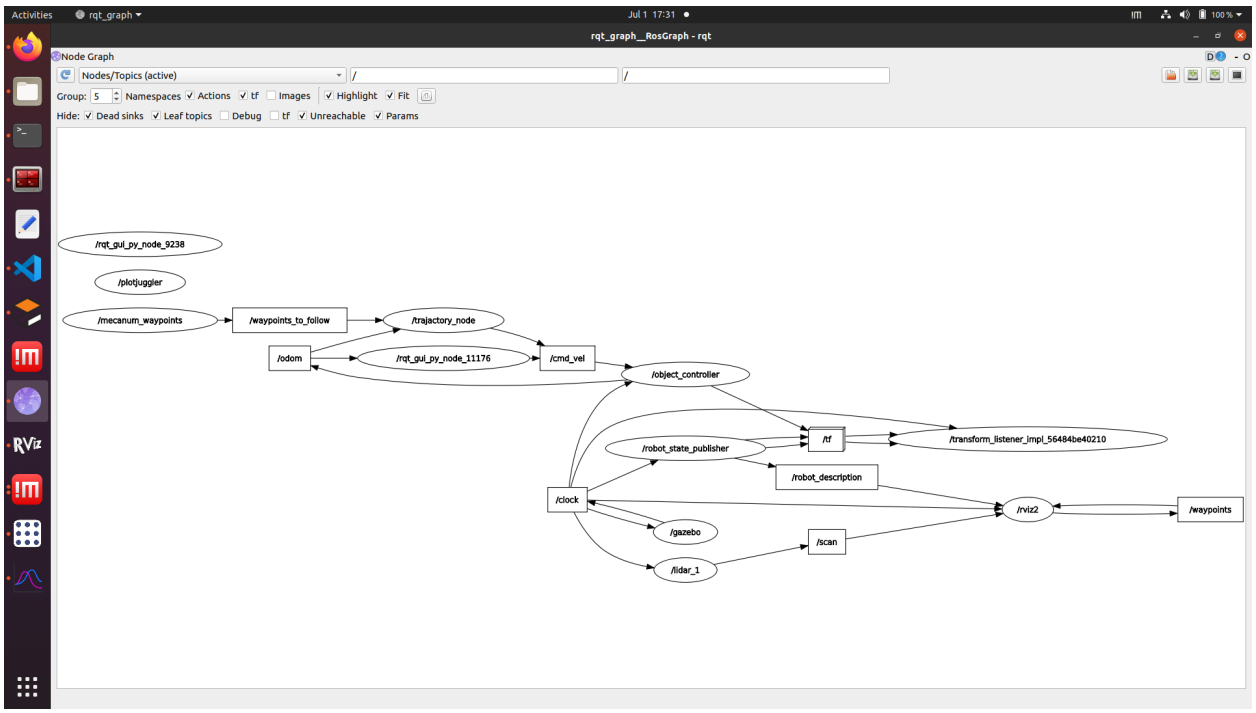
- ***control.launch.py*** :

The provided Python code is a Launch file for launching two ROS nodes: "**mecanum_waypoints_param**" or "**mecanum_waypoints**" and "**mecanum_traj_controller**". This Launch file will be used with the ROS2 launch system to start these nodes simultaneously.

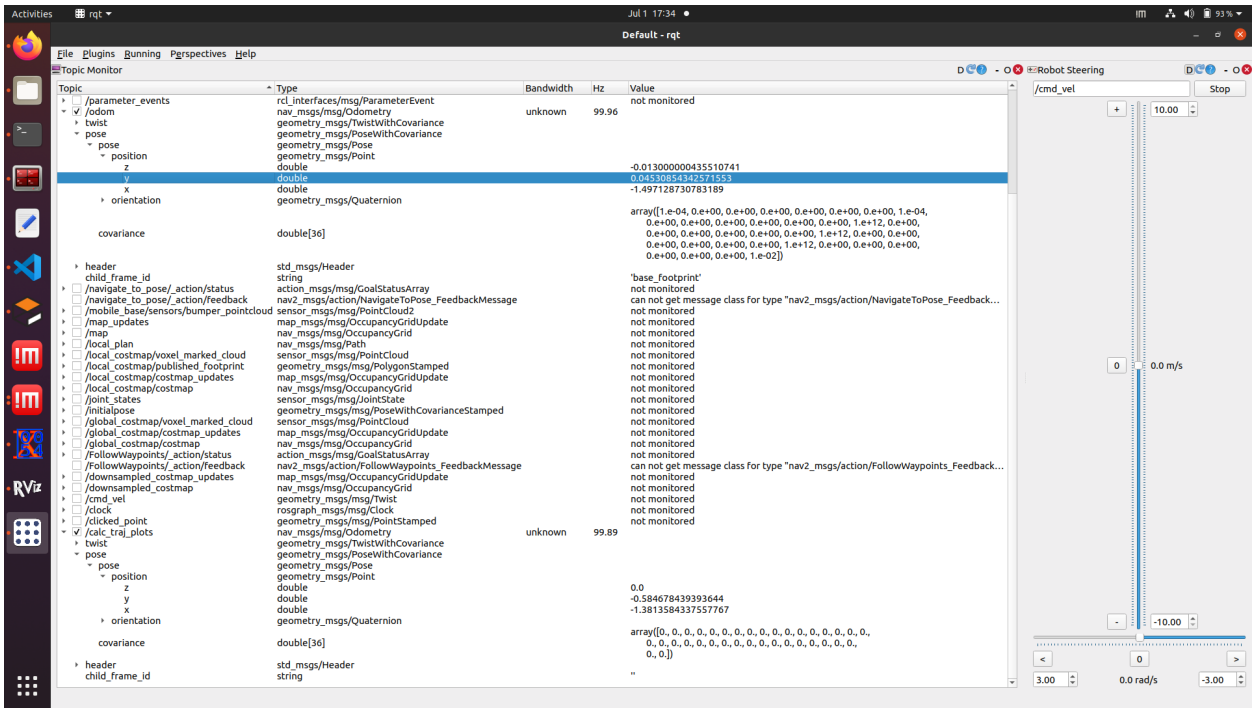
Description of the Launch file:

1. Importing necessary libraries:
    - The Launch file imports the `LaunchDescription` class and the `Node` action from the `launch` and `launch_ros.actions` modules, respectively.
2. The `generate_launch_description` function:
    - This function creates and returns a `LaunchDescription` object containing the configuration for launching the two nodes: "**mecanum_waypoints_param**" or "**mecanum_waypoints**"and "**mecanum_traj_controller**".
3. Adding nodes to the LaunchDescription:
    - Two instances of the `Node` action are created to represent the two ROS nodes that need to be launched: "**mecanum_waypoints_param**" or "**mecanum_waypoints**" and "**mecanum_traj_controller**".
    - For each node, the package name and the executable name are specified using the `package` and `executable` arguments, respectively.
    - These nodes will be launched with the specified names using this Launch file.
4. Adding nodes to the LaunchDescription object:
    - The two nodes created earlier are added to the `LaunchDescription` object `ld` using the `add_action` method. This includes the nodes in the Launch file's launch sequence.
5. Return the LaunchDescription object:
    - Finally, the function returns the `LaunchDescription` object `ld` containing the configuration for launching the two nodes.

Overall, this Launch file will launch the two ROS nodes, **"mecanum_waypoints_param"** or "**mecanum_waypoints**" and "**mecanum_traj_controller**", using the ROS2 launch system.



rqt_graph after launching control.launch.py



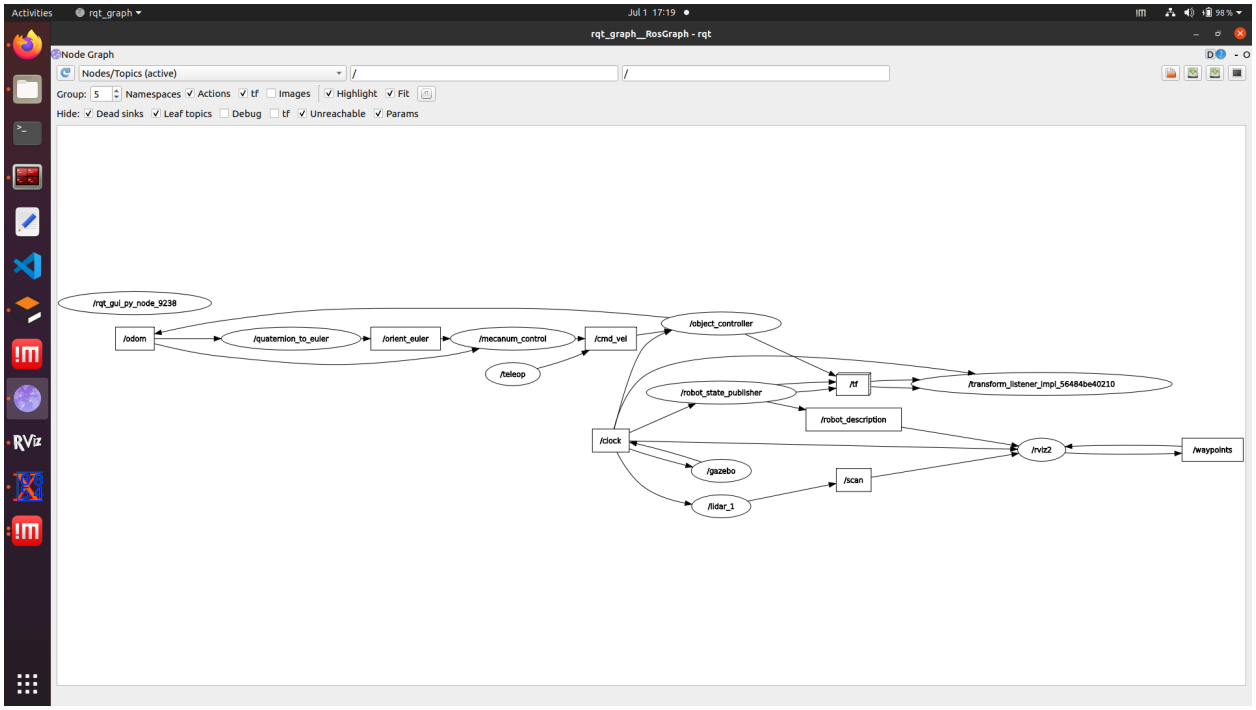
rqt topic monitor showing active topics after launch of control.launch.py

- ***go_to_goal.launch.py :***

The provided Python code is a Launch file for launching two additional ROS nodes: "**quat_to_euler**" and "**mecanum_controller**". These nodes will be used to control a mecanum drive robot in order to track goal points.

Description of the Launch file:

1. Importing necessary libraries:
    - The Launch file imports the `LaunchDescription` class and the `Node` action from the `launch` and `launch_ros.actions` modules, respectively.
2. The `generate_launch_description` function:
    - This function creates and returns a `LaunchDescription` object containing the configuration for launching the two new nodes: "**quat_to_euler**" and "**mecanum_controller**".
3. Adding nodes to the LaunchDescription:
    - Two instances of the `Node` action are created to represent the two new ROS nodes that need to be launched: "**quat_to_euler**" and "**mecanum_controller**".
    - For each node, the package name and the executable name are specified using the `package` and `executable` arguments, respectively.
4. Adding nodes to the LaunchDescription object:
    - The two new nodes created earlier are added to the existing `LaunchDescription` object `ld` using the `add_action` method. This includes the new nodes in the Launch file's launch sequence.
5. Return the LaunchDescription object:
    - Finally, the function returns the `LaunchDescription` object `ld` containing the configuration for launching the two nodes: "**quat_to_euler**" and "**mecanum_controller**".



rqt_graph after go_to_goal.launch.py

- ***simulation.launch.py :***

The provided Python code is a Launch file for launching a simulation of the Neobotix MPO-500 robot in Gazebo. It launches various nodes and sets up the necessary configurations to simulate the robot.

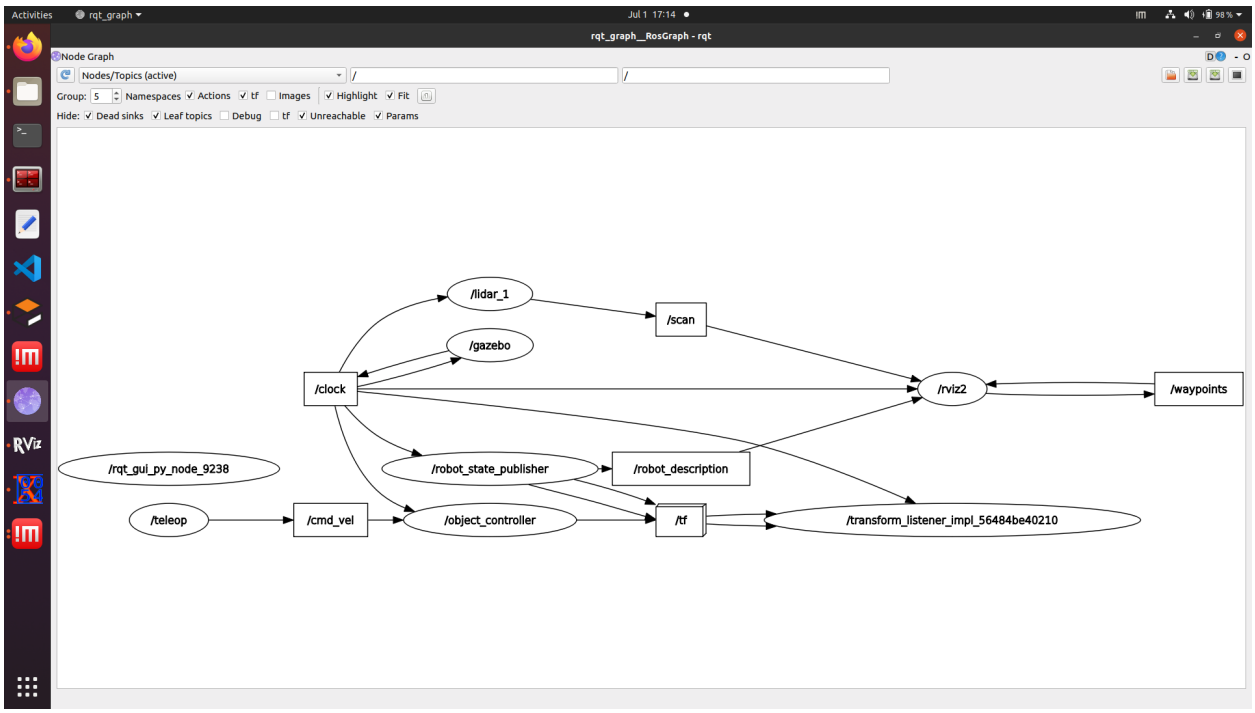
Description of the Launch file:

1. Importing necessary libraries:
    - The Launch file imports the required libraries and modules for launching the simulation.
2. The `generate_launch_description` function:
    - This function creates and returns a `LaunchDescription` object containing the configuration for launching the simulation.
3. Setting default values:
    - The default world path and whether to use simulation time are specified using `default_world_path` and `use_sim_time`, respectively.
4. Declaring launch arguments:
    - The launch arguments `use_sim_time` and `robot_dir` are declared using the `DeclareLaunchArgument` action.
    - `use_sim_time` is used to control whether to use simulation time or not.
    - `robot_dir` specifies the directory of the robot URDF file.
5. Setting up the simulation environment:

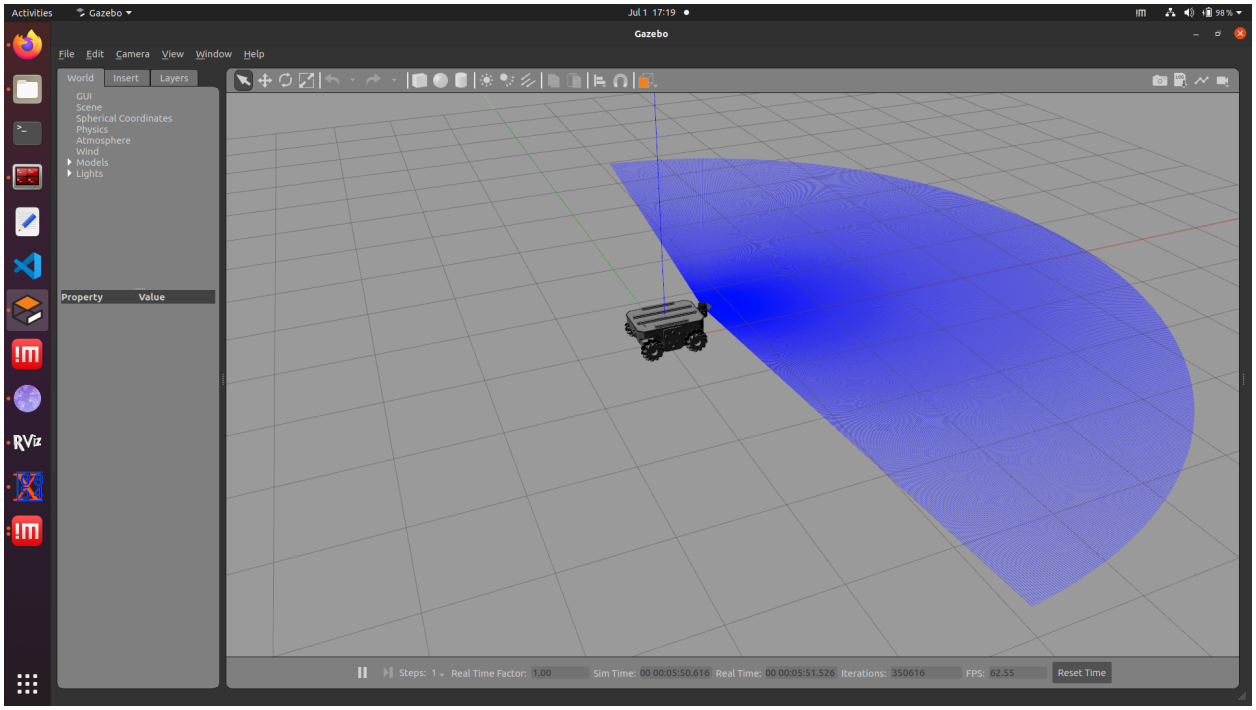
- The `spawn_entity` Node is used to spawn the robot entity in Gazebo. It specifies the package, executable, and arguments required to spawn the robot.

6. Launching **robot_state_publisher**:
   - The `start_robot_state_publisher_cmd` Node launches the `robot_state_publisher` with the specified parameters and arguments. It publishes the robot's joint states.
7. Launching **teleop_twist_keyboard**:
   - The `teleop` Node launches the `teleop_twist_keyboard` package, which provides keyboard control for the robot's motion.
8. Launching **Gazebo**:
   - The `gazebo` LaunchDescription includes the Gazebo launch file provided by the `gazebo_ros` package. It launches Gazebo with the specified world file.
9. Launching **RViz**:
   - The `rviz` Node launches RViz with the specified arguments and parameters. It loads the RViz configuration file and sets the `use_sim_time` parameter.
10. Return the LaunchDescription object:
    - Finally, the function returns the `LaunchDescription` object containing the configuration for launching the simulation.

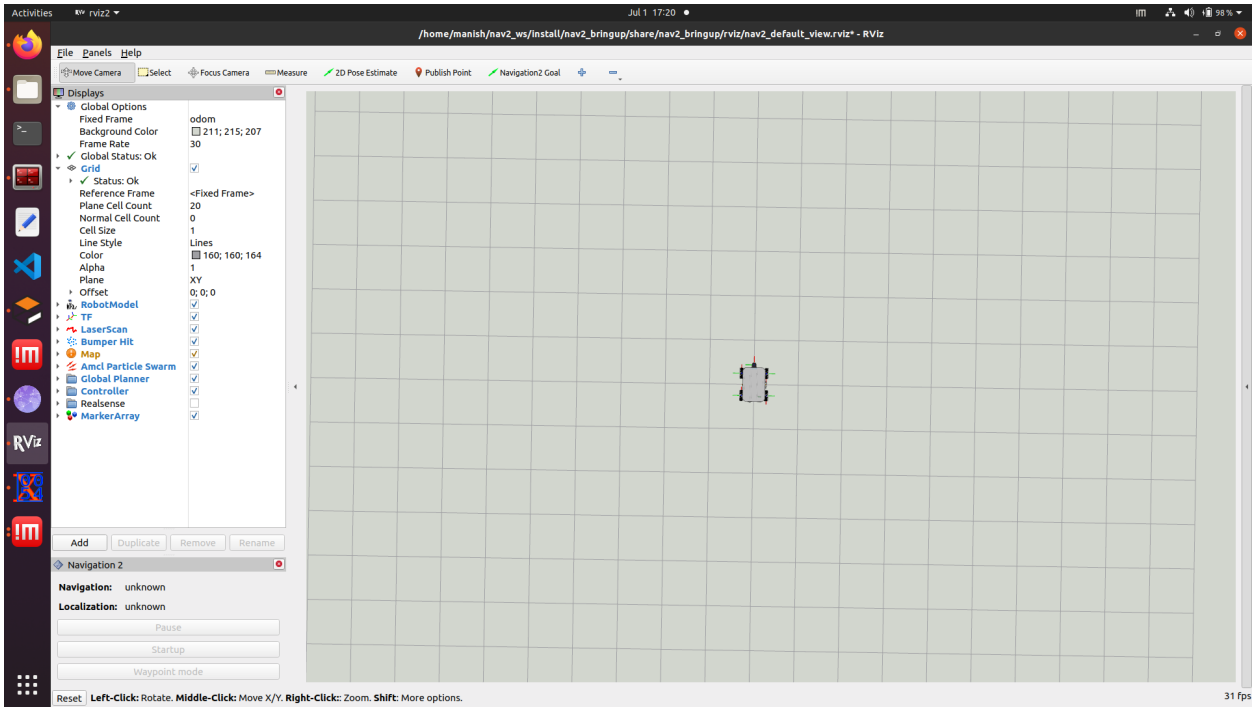
Overall, this Launch file sets up the simulation environment for the **Neobotix MPO-500** robot in Gazebo(which is the mecanum drive model). It spawns the robot, launches the `robot_state_publisher`, provides teleop control, and opens **RViz for visualization**.



rqt_graph after launching simulation.launch.py



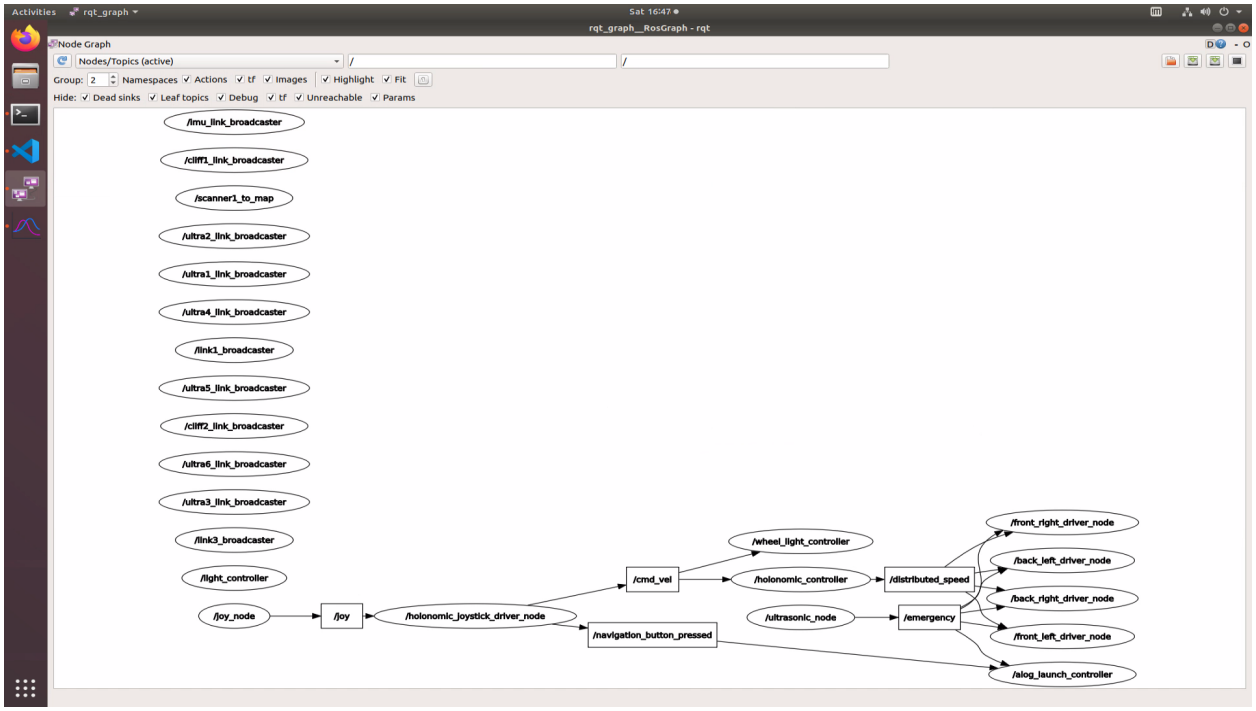
After spawning robot in gazebo using launch file



After launch of simulation.launch.py robot spawned in rviz2

❖ On hardware System (ALOG) :

It is a complete system well integrated with **Realsense camera** ,**RP LIDAR** ,**LED**s for indications **Stepper Motor with encoders** mounted ,**Leadshine Servo Motor Driver**,**IMU** and processing powers of **Intel NUC**.It has packages specifically build for it which are not customizable or nor we can rebuilt by making some changes into it.It has resource packages for autonomous navigation and planning.The system has Ubuntu 18.04 LTS and everything which is integrated and designed for ROS MELODIC (supports Python 2 only) .



rqt_graph when ALOG starts

For Documentation of its Packages go to Alog_Doc_and_description .

- ***go_to_goal.py*** :

The provided code is a Python script for controlling a mobile robot to follow a predefined set of goals in a 2D environment using odometry feedback. The robot's control inputs (linear and angular velocities) are calculated based on the desired path and its current odometry information.

Description of the code:

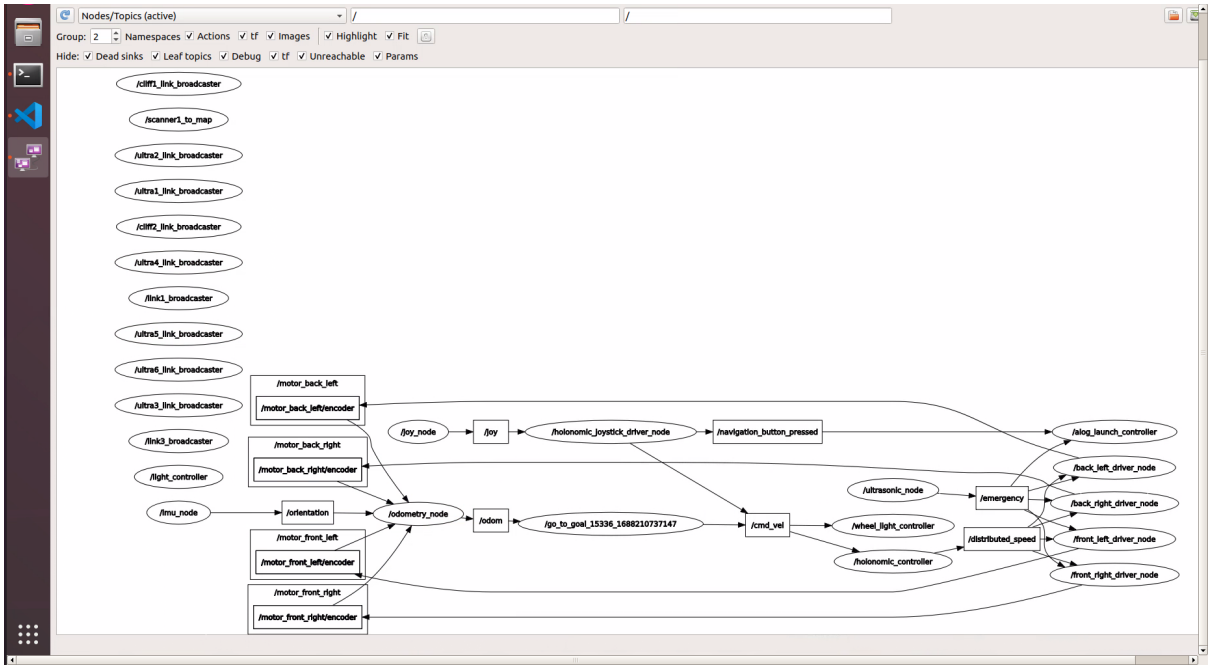
1. Importing necessary libraries:
    - `rospy`: Python client library for ROS (Robot Operating System).
    - `Twist`, `Vector3`: Messages for linear and angular velocities and vector representation.

- `Odometry`: Message for odometry information (position and orientation).

2. Class `Data`:
   - This class defines variables and parameters for the robot's control and path.
   - It sets up default values for PID (Proportional-Integral-Derivative) gains for angular and linear velocity control.
   - `path`: A list of waypoints that represents the robot's predefined path in the environment.
   - `publish_`: A ROS publisher to publish the robot's control commands (Twist messages).
   - `publish_euler`: A ROS publisher to publish the robot's orientation in Euler angles (yaw).
3. `node_start()` function:
   - This function initializes the ROS node and sets up the data object.
   - It subscribes to the **`/odom`** topic to receive odometry information (robot's position and orientation).
   - The function enters a ROS spin loop to keep the node running.
4. `quat_to_euler()` function:
   - This function converts a quaternion representation of orientation to Euler angles.
   - It is used to extract the yaw angle (z-axis rotation) from the robot's orientation quaternion.
5. `odometry_data_callback(odom)` function:
   - This callback function is executed every time new odometry data is received on the **`/odom`** topic.
   - It calculates the error in orientation (yaw), error in x and y position from the desired path.
   - The **PID controller is used** to calculate control inputs for angular and linear velocities based on the error values.
   - The calculated velocities are then transformed from global coordinates to robot body frame using trigonometric functions.
   - The final control commands (linear and angular velocities) are published to the **/cmd_vel** topic.
6. `main(args=None)` function:
   - This is the main entry point of the script.
   - It initializes the ROS node, prints a message indicating the node has started, and calls `node_start()`.
7. `if __name__ == "__main__":`
   - This condition checks if the script is being run directly (not imported as a module).
   - If true, it calls the `main()` function to start the robot controller.

Overall, the code establishes a ROS node that subscribes to odometry data, performs path following using a PID controller, and publishes control commands to drive the robot along the

predefined path. The robot's orientation in Euler angles is also published for visualization purposes. It provides a basic example of path-following control for a mobile robot using odometry feedback in a ROS environment.



rqt_graph after launching go_to_goal.py and odom_custom.launch

- ***waypoints_controller.py*** **:**

The provided Python script is designed to generate local waypoints for a Mecanum robot, forming either an infinity (∞) symbol or a circle trajectory and much more depending upon the values of parameters. The script utilizes ROS (Robot Operating System) to communicate with other nodes and control the robot's motion.

Description of the code:

1. `node_start()` Function**:**
    - This function is the main function of the script responsible for controlling the Mecanum waypoints.
    - It initializes ROS node, publishers for waypoints and trajectory plots.
    - The function generates Mecanum wheel trajectories in the shape of an infinity (∞) symbol or a circle.
    - It calculates the position and velocity coordinates of the Mecanum robot.
    - The calculated values are published t**o /waypoints_to_follow** and **/calc_traj_plots** topics.
    - The function runs in a loop until ROS is shut down, updating the position and velocity values at a rate of 100 Hz.

- The interval in which the msg is published on "**/waypoints_to_follow**" topic is specified by the delay using `rospy.sleep()` function .

2. Initialization:

- Import necessary libraries and ROS messages (`rospy`, `Vector3`, `Odometry`).
- Initialize ROS node and create publishers for **/waypoints_to_follow** and **/calc_traj_plots** topics.
- Set up initial values for the waypoints and time.

3. Waypoint Generation:

- The function generates waypoints for Mecanum robot to follow in the shape of an infinity (∞) symbol or a circle.
- It uses trigonometric functions (`cos` and `sin`) to calculate the x, y positions of the waypoints.
- The shape of the trajectory (infinity or circle) can be selected by uncommenting the respective block and commenting the other.

4. Velocity Calculation:

- The function calculates the x, y velocity components of the Mecanum robot at each waypoint.
- It uses the derivatives of the x, y position equations to find the velocity components.

5. Publishing Waypoints and Trajectory Plots:

- The calculated waypoints and velocity values are published to the **/waypoints_to_follow** and **/calc_traj_plots** topics, respectively.
- These published values can be used by other nodes to control the Mecanum robot's motion.

6. Node Execution:

- The `main()` function initializes the ROS node and calls `node_start()` to start the Mecanum waypoint generation.
- It prints messages to indicate the start and end of node execution.

Note: Make sure to set the appropriate values for the shape (infinity or circle and much more) and adjust the trajectory parameters (**A1, A2, omega1, omega2, phi1, phi2)** based on the desired trajectory pattern and robot behavior.

- ***waypoints_controller_traj_gen.py*** :

The provided Python2 code represents a ROS node for generating robot's trajectory using a 3rd-order polynomial trajectory generator, and the velocity control is implemented to ensure smooth movement between waypoints.

Description of the code:

1. Importing necessary libraries and ROS messages:
    - The script starts by importing required ROS libraries, including `rospy`, and necessary ROS message types from the `geometry_msgs` and `nav_msgs` packages.
2. The Data class:
    - The `Data` class is defined to store essential data related to waypoints, such as start points, target points, and published waypoints. It also contains publishers for **/waypoints_to_follow** and **/calc_traj_plots** topics.
3. Function `node_start:`
    - The `node_start` function is the main entry point for the ROS node.
    - It initializes the ROS node and sets up the necessary variables and waypoints for the robot to traverse.
    - The function enters a loop, where the robot's trajectory is controlled by calling the `move_to_point` function.
4. Function `move_to_point:`
    - The `move_to_point` function generates a polynomial trajectory and moves the robot to the current waypoint.
    - It calculates coefficients of the 3rd-order polynomial trajectory using `polynomial_time_scaling_3rd_order` function.
    - It then performs position and velocity updates in a loop to move the robot smoothly along the generated trajectory until it reaches the target waypoint.
    - The local goal points are published to **/waypoints_to_follow**, and trajectory points are published to **/calc_traj_plots**.
    - The interval in which the msg is published on "**/waypoints_to_follow**" topic is specified by the delay using `rospy.sleep()` function .
5. Function `polynomial_time_scaling_3rd_order`:
    - This function computes the coefficients of a 3rd-order polynomial trajectory based on the given start and end points, start and end velocities, and the desired time (`T`) to complete the trajectory.

- The function returns the coefficients **A and B** for the polynomial trajectory in the x and y directions, respectively.

6.Function `main`:

- The `main` function initializes the ROS node with the name '**mecanum_waypoints**' and calls `node_start` to start the waypoint following process.

- ***waypoints_traj_controller.py*** :

The provided Python script is a ROS (Robot Operating System) node named "**mecanum_traj_controller**" designed to control a mecanum-wheel robot's trajectory and navigation towards given local waypoints. It subscribes to two topics: "**/waypoints_to_follow**" to receive the desired waypoints and "**/odom**" to obtain the robot's odometry data, which includes position, velocity, and orientation.

Description of the code:

1. Importing necessary libraries:
   - The script starts by importing required libraries and modules, including `rospy`, which is the Python client library for ROS, and messages from ROS message packages, such as `Twist`, `Vector3`, `Point`, and `Odometry`.
2. Class Data:
   - This class is defined to store relevant data variables and parameters used in the mecanum trajectory controller.
   - It includes variables to hold the robot's current position (x, y), velocity (vel_x, vel_y), and current yaw (orientation).
   - The class also contains **PID gains for controlling errors in X, Y, and orientation** directions, as well as variables to store integral and derivative terms used in PID control.
   - Additionally, the class initializes a publisher (`publish_`) to send velocity commands to the robot.
3. Function node_start():
   - This function is the main entry point of the script and is called to start the mecanum trajectory controller node.
   - It initializes the `Data` class to manage data and control parameters.
   - The function subscribes to two ROS topics: "**/waypoints_to_follow**" to receive desired waypoints and "**/odom**" to obtain odometry data.
   - The function enters the ROS spin loop (`rospy.spin()`) to continuously process incoming messages and control the robot's motion.
4. Function quat_to_euler(x, y, z, w):

    - This function is a helper function to convert a quaternion (used for orientation) to Euler angles (roll_x, pitch_y, yaw_z).
    - The function takes four arguments (x, y, z, w) representing the quaternion values and calculates the corresponding yaw angle (yaw_z) using trigonometric functions.
5. Function odometry_data_callback(odom):
    - This function is executed every time an "**odom" message** is received.
    - It extracts the robot's current position (x, y), linear velocities (vel_x, vel_y), and orientation (yaw_z) from the received "odom" message.
    - The extracted values are stored in the `Data` class variables for later use in the trajectory controller.
6. Function go_to_goal(waypoint):
    - This function is called whenever a new waypoint is received from the "**/waypoints_to_follow**" topic.
    - It calculates errors (**e_x, e_y, e_orient**) between the current position and the desired waypoint.
    - The function then uses PID control to calculate control inputs (**u_w, u_x, u_y**) to drive the robot towards the desired waypoint.
    - The control inputs are converted from the global frame to the robot's body frame to account for the robot's orientation.
    - The calculated velocities (**v_x, v_y, v_w**) are published to the "**/cmd_vel**" topic to control the robot's motion.
    - The function also handles integral and derivative terms for **PID control to minimize overshoot and steady-state errors.**
7. Function main(args=None):
    - This function initializes the ROS node with the name "**mecanum_traj_controller**" and sets it to run anonymously.
    - It then calls the "**node_start()**" function to start the mecanum trajectory controller node.

Overall, the Python script acts as a ROS node responsible for controlling a mecanum-wheel robot's trajectory towards given waypoints using a PID control algorithm to **adjust the robot's linear and angular velocities** based on the **error between the current position and the desired waypoints**.

- ***traj_trac.launch :***

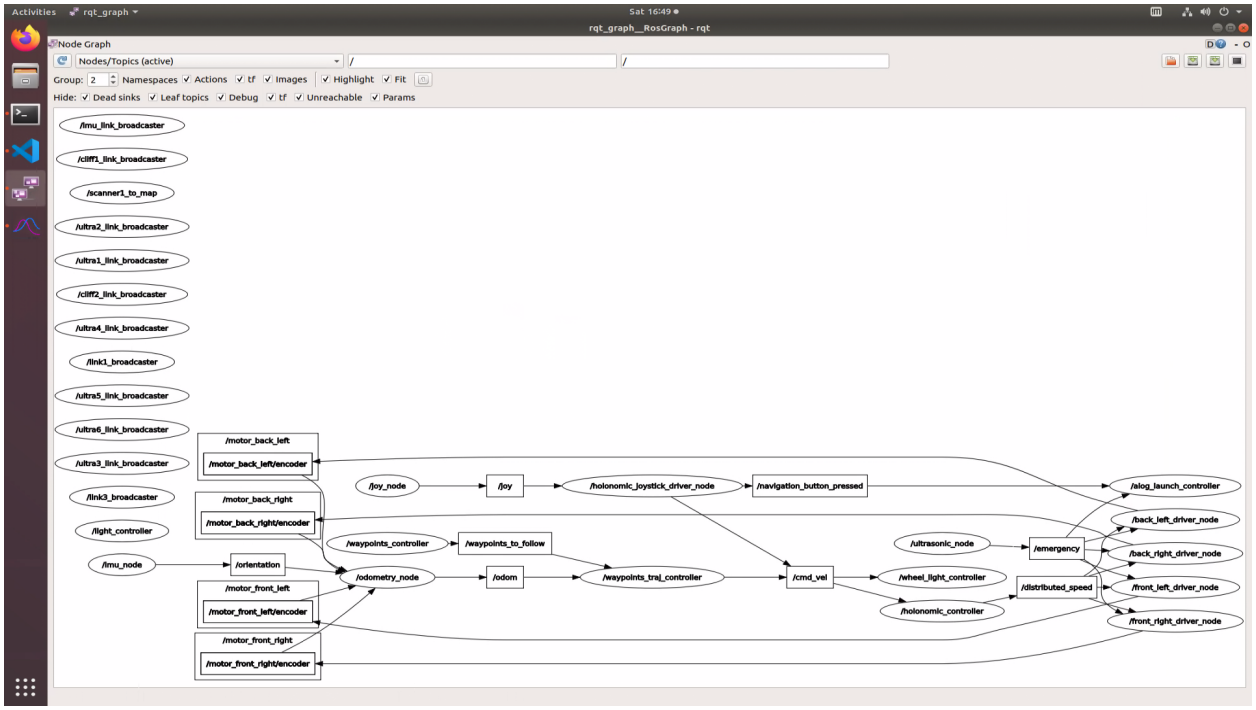
The provided launch file is used to launch several ROS nodes such as **waypoints_controller** node or **waypoints_controller_traj_gen** node and **waypoints_traj_controller** node.

Description of the code:

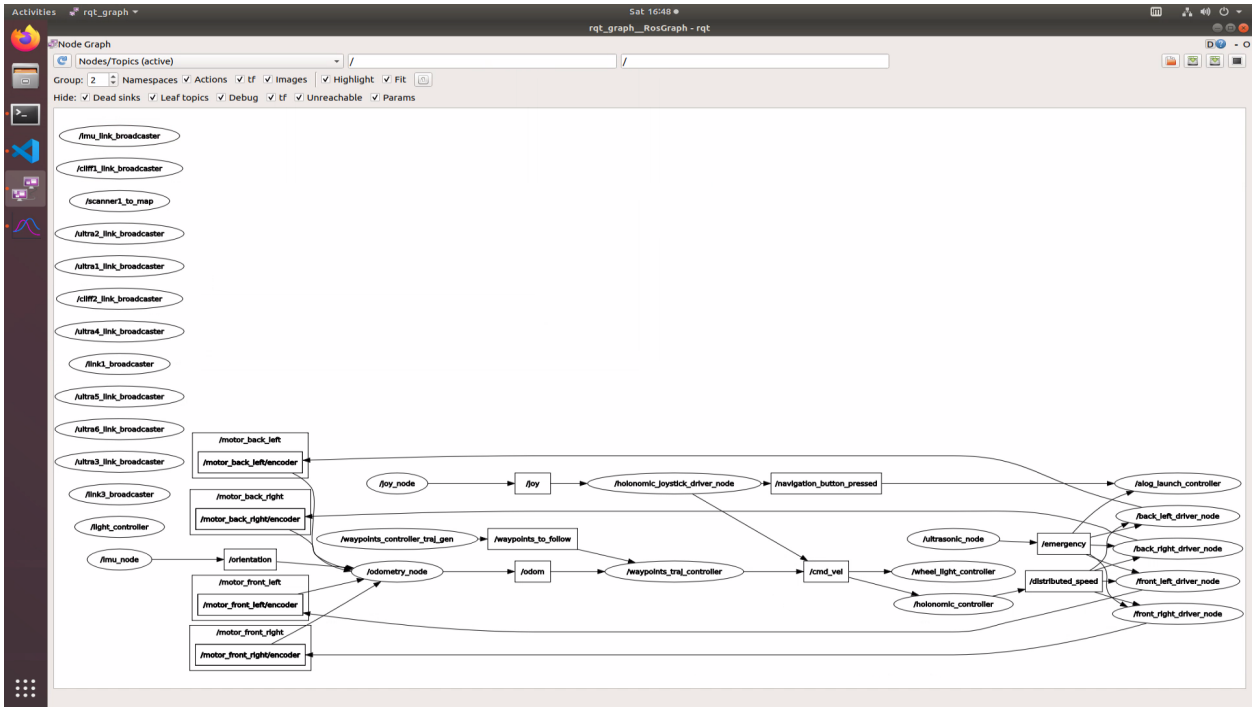
1. Launch Odometry Node (Commented Out):
    - The **holonomic_odometry_node** is commented out in the launch file, so it won't be launched when the file is executed.

    - This node is responsible for providing odometry information, which includes the robot's position, orientation, and velocity.
2. Launch IMU Node (Commented Out):
    - The **imu.py** node from the `sensors_package` is commented out in the launch file, so it won't be launched when the file is executed.
    - This node is responsible for interfacing with the Inertial Measurement Unit (IMU) sensor and providing IMU data such as orientation, angular velocity, and linear acceleration.
3. Launch Waypoints Controller Node:
    - This section launches the **waypoints_controller.py** node from the `alog_control` package.
    - The node is given the name **waypoints_controller**, and its output will be displayed on the screen (`output="screen"`).
    - The **waypoints_controller** node is responsible for controlling the robot's movement to follow a series of predefined waypoints. It likely uses the robot's odometry and/or IMU data to calculate the required control commands.
4. Launch Waypoints Trajectory Controller Node:
    - This section launches the **waypoints_traj_controller.py** node from the `alog_control` package.
    - The node is given the name **waypoints_traj_controller,** and its output will be displayed on the screen (`output="screen"`).
    - The **waypoints_traj_controlle**`r` node is likely responsible for generating trajectories based on the predefined waypoints and sending appropriate control commands to the robot to follow those trajectories accurately.

The code has two nodes commented out **(holonomic_odometry_node** and **imu.py**), which means they are not launched when the file is executed. When these two nodes are launched along with other nodes specified in the launch file will result in defination of origin and global frame from thereof.



rqt_graph after launching traj_trac.launch with waypoints_controller.py



rqt_graph after launching traj_trac.launch with waypoints_controller_traj_gen.py

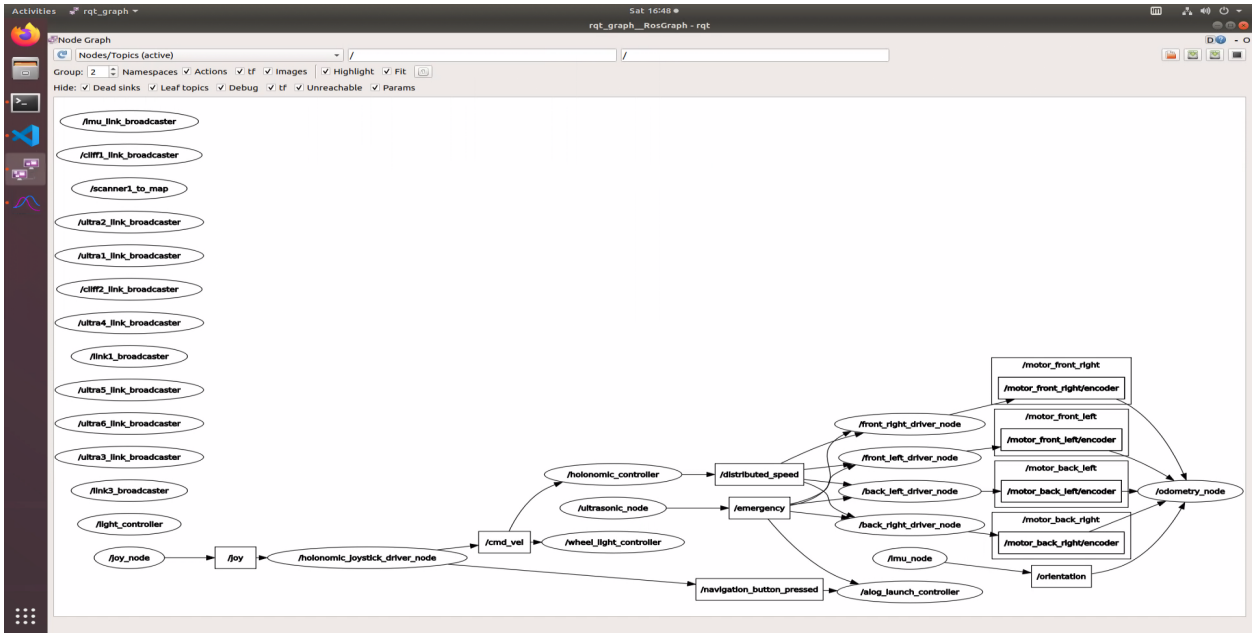
- ***odom_custom.launch:***

The provided modified launch file includes the launch of two ROS nodes, the `holonomic_odometry_node` from the `holonomic_controller` package, and the `imu.py` node from the `sensors_package`.

Description of the code:

1. Launch Odometry Node:
    - The **holonomic_odometry_node** is launched from the **holonomic_controller** package.
    - The node is given the name **odometry_node**, and its output will be displayed on the screen (`output="screen"`).
    - This node is responsible for providing odometry information, including the robot's position, orientation, and velocity. It is essential for tracking the robot's motion.
2. Launch IMU Node:
    - The **imu.py** node is launched from the `sensors_package`.
    - The node is given the name **imu_node**, and its output will be displayed on the screen (`output="screen"`).
    - This node is responsible for interfacing with the Inertial Measurement Unit (IMU) sensor and providing IMU data, such as orientation, angular velocity, and linear acceleration. IMU data is crucial for understanding the robot's orientation and motion.

Note: The `<include>` tag, which was present in the original launch file, has been commented out in the modified launch file. This means that the **hardware.launch** file, which launches other hardware-related nodes,related to navigation and planning .



rqt_graph after launching odom_custom.launch

Difficulties Encountered and their Solutions

The project goal was to generate a trajectory through a set of waypoints and make the robot with mecanum wheels attached to it to follow the trajectory generated. To achieve it we can divide the task into two parts -

❖ **Virtual simulation -**

- For the development of our system, we had chosen to go with ROS2 FOXY for this project. At first, we looked for the models for a mechanum bot and looked if any urdf or mesh files were available for us to run the bot in simulation, we came across a Urdf file for mpo500(foxy). We started with spawning the bot in the open world and writing code to make the bot move in a particular direction and velocity. While searching for the trajectory generation of mecanum bot we didn't find any pre-implemented setup for the project we had been assigned.
- Soon after we achieved the task of spawning the mpo500 in an empty environment we went to write the code for the trajectory generation for the bot, we came across these resources -[1] trajectory generation code can be found above with detailed explanation. We faced some problems in the simulation let's discuss them in detail -
    - The /cmd_vel velocity was published to the bot resulting in the movement of the body frame only, i.e the bot was moving without the rotation of wheels imagine it like the bot if floating and moving. To address this issue we needed to publish the /cmd_vel to a motor driver which could convert the bot velocity to wheel rotational motion. Hence for that, we collected these resources- [2]. Though this wasn't implemented as the hardware we were going to work on already had this integration in place.

- After the simulation was accomplished on the ROS2 foxy. when the time came for the implementation on the hardware we faced the problem of having to switch from ROS2 foxy to ROS1 Melodic, upgrading the bot wasn't possible for the moment because first of all there were too many

complications in the low-level control and also many packages didn't have the support for ROS2, not to mention the amount of time that would have taken to migrate the whole hardware system. So we made the decision of downgrading our code to ROS1 Melodic and also switching to python2 for the same.

- While writing our code for goal to goal, our code didn't use to work always. The problem was when the bot reached the goal location it didn't use to stop, the reason for that was when we reached the goal location the exact mentioned goal point was never published due to the publishing rate, take for eg if the goal location is (1,1) the goal wasn't reached because Odom rate used to miss the point 1 by say 1.9 which rounds off to two so to solve this problem we created a buffer around the goal locations which would help us to stop the bot at a given range.

❖ **Physical implementation on the actual hardware -**

- Getting started with the hardware you can actually find the code documentation over here[3] This covers the main integration and working of the bot with the ros and there is some very insightful documentation that will help with onboarding with the hardware.
- After testing the code for the trajectory following we were facing some unexpected issues, in the simulation the bot was working just fine though in real life the bot was performing in an unexpected manner. we had given it a goal to reach but then it started moving slowly and then speeded up constantly whereas in the simulation and by the code logic it must have reached the goal within 5 seconds but the bot didn't stop and continued to move at a constantly increasing speed and continued even after reaching it's desired goal location you can have a look at the video over here [4].
- The reason for this kind of behavior was found out to be because of the difference in the velocity publishing, updating, and new odom subscribing rate. Actually, the robot has a velocity updating rate of 7hz and we were publishing velocity at 50hz, also the Odom that we were subscribed to was updating at a rate of 10hz hence making this whole system a mess.
- To fix it we synchronized the code efficiently and also added rospy.sleep at the velocity publishing part of the code.
- After successful implementation of the goal to goal programme we were ready to move ahead with the implementation of via points trajectory

following we initially were implementing basic cubic polynomial trajectory generation following which we started with the implementation of via point trajectory following with via point velocity.

- **Trajectory Approach and Types-**

For the trajectory generation a lot of different options are available, in our research there we came across a few really good trajectory generation methods. While generation of trajectory we usually define a polynomial equation usually cubic or quintic. The difference between the two is the degrees of polynomial and the type of trajectory it generates. For different cubic trajectories you can refer this [5] .

Cubic and quintic polynomials are both types of polynomial functions, but they have different degrees and, therefore, different numbers of turning points. This leads to distinct trajectories when these polynomials are graphed.

Cubic Polynomial:
A cubic polynomial is a polynomial function of degree 3, which means it has the form:
$f(x) = ax^3 + bx^2 + cx + d$, where a, b, c, and d are coefficients.
The graph of a cubic polynomial can have up to two turning points. Depending on the values of the coefficients, these turning points can be local minima or maxima. The trajectory of a cubic polynomial can take various shapes, such as a "U" shape, an "S" shape, or a combination of both. It can also exhibit symmetry or asymmetry, depending on the coefficients involved.

Quintic Polynomial:
A quintic polynomial is a polynomial function of degree 5, which means it has the form:
$f(x) = ax^5 + bx^4 + cx^3 + dx^2 + ex + f$, where a, b, c, d, e, and f are coefficients.
The graph of a quintic polynomial can have up to four turning points. Similarly to the cubic polynomial, these turning points can be local minima or maxima. However, with the higher degree, quintic polynomials tend to have more complex trajectories. They can exhibit multiple loops, more intricate patterns, and greater overall variability.

Initially we used Cubic Polynomial trajectory generation approach for trajectory generation, yet in further course we used Cubic Polynomial trajectory generation with via point velocities. Let's discuss in more detail about the trajectory genration by cubic polynomial.

To generate a trajectory using cubic polynomials with way-points, you can use a technique called cubic spline interpolation. Cubic spline interpolation is commonly used to create smooth curves that pass through specified points.Here's a step-by-step process to generate a trajectory using cubic polynomials and way-points:

1. Define your way-points: Determine the points through which you want your trajectory to pass. Each way-point should have an associated x-coordinate and y-coordinate along with a time parameter specifying the time it's supposed to take to reach to the given point.
   Format - (x,y,T)
2. Calculate the coefficients: For each pair of adjacent way-points (P1 and P2), calculate the coefficients of the cubic polynomial that passes through those points. You will have one cubic polynomial between each pair of adjacent way-points.

   To find the coefficients a, b, c, and d, you need to set up a system of equations using the way-points' coordinates and derivative conditions.

   a. The first equation ensures that the polynomial passes through the first way-point (P1):
   $p1 \ = \ 0 \, at \, t \ = \ 0$ hence,
   For polynomial, $f(t) \ = \ at^3 \ + \ bt^2 + \ ct \ + \ d$
   $f(p1) \ = \ f(0) \ = \ a(0)^3 \ + \ b(0)^2 + \ c(0) \ + \ d \ = \ 0$

   b. The second equation ensures that the polynomial passes through the second way-point (P2):
   Here, $p2 \ = \ x \, at \, t \ = \ x$
   Hence, $f(p1) \ = \ f(x) \ = \ a(x)^3 \ + \ b(x)^2 + \ c(x) \ + \ d \ = \ x$
   We get, $f(p1) \ = \ ax^3 + \ bx^2 + \ cx \ + \ d$

c. The third equation enforces smoothness by requiring the first derivative to be continuous at the way-point:
$At\ t = 0,\ f'(0) = 0$
Also at the final point, $At\ t = x,\ f'(x) = 0$

d. The fourth equation enforces smoothness by requiring the second derivative to be continuous at the way-point(applicabe only for quintic polynomial):
$At\ t = 0,\ f''(0) = 0$
Also at the final point, $At\ t = x,\ f''(x) = 0$

3. Express the cubic polynomial: The general form of a cubic polynomial is $f(t) = at^3 + bt^2 + ct + d$
4. To find the coefficients a, b, c, and d, you need to solve a system of equations using the way-points' coordinates.

Ensure smoothness: To ensure smoothness at the way-points, you need to enforce continuity conditions between adjacent polynomials. This is typically done by requiring that the first and second derivatives match at each way-point.

5. Evaluate the trajectory: Once you have the coefficients for each cubic polynomial, you can evaluate the trajectory by calculating the y-coordinate for a given x-coordinate. You can sample the trajectory at regular intervals or at specific points of interest.

References

Trajectory generation [1],[4],[5]-

For different types of splines that can help with trajectory generation -

1. All curves in summary(yt)[5]
2. Trajectory generation with n-via points for a mobile robot(yt)[1]
3. Trajectory Generation[1]
4. Linorobot_mecanum , linorobot2 [4]
5. Trajectory generation via points velocity not specified

Mechanum wheel control [2] -

Resources for conversion bot velocity to wheel rotational motion -

1. Drive Kinematics: Skid Steer & Mecanum (ROS Twist included) this has the mathematical logic for **IK & FK.**
2. C++ code for it
3. linorobot/src/lino_base.cpp
4. How to Use Mecanum Wheels
5. ROS2 important Commands

Hardware Resouces and work flow [3]-

This contains the rosouces related to hardware and everything related to it that I found helpful.

1. Hardware source code documentation

Introduction to Autonomous Mobile Robots book

Modern Robotics book by Kevin Lynch

Getting Ready for ROS Part 8 Simulating with Gazebo Articulated Robotics

Simulation model of ROS2 Mecanum drive

Videos and Screenshots