HOI - Vehicle Manipulator Kinematic Control

Loc Thanh Pham
Intelligent Field Robotics Systems
Universitat de Girona
Girona, Spain
u1992429@campus.udg.edu

Atharva Anil Patwe
Intelligent Field Robotics Systems
Universitat de Girona
Girona, Spain
patweatharva@gmail.com

Samantha Caballero Gomez
Intelligent Field Robotics Systems
Universitat de Girona
Girona, Spain
u1992766@campus.udg.edu

***Abstract*—This study aims to design and implement a Task-Priority Kinematic Control algorithm to control the Kobuki Turtlebot and mobile manipulator with 6 degrees of freedom (DoF), equipped with a vacuum gripper for pick and place tasks. The system incorporates forward and inverse kinematics calculations to enable control over the Jacobian of the manipulator's end-effector. The goal is to develop a comprehensive control architecture that allows for efficient task execution.**

***Index Terms*—DoF, Stonefish, Kobuki Turtlebot, Jacobian, Task-Priority recursive control, DLS**

I. Introduction

The report outlines the hardware, software, and control architectures, discusses the forward and inverse kinematics algorithms. Furthermore, presents the implementation of the task priority control tested under different scenarios such as end-effector and mobile base positioning and orientation, joint limits, and ArUco marker-based pick-and-place operations, all established in a sequence of motions defined by a behavior tree. The results demonstrate the effectiveness and reliability of the implemented kinematic control system, highlighting its potential for practical applications. To have a broad understanding of the platform the following work will be based upon, we describe during the conceptual phase the software, hardware and control architectures.

II. Conceptual design

A. Hardware Architecture

In figure 1 we find a high-level block diagram of the robot hardware architecture, outlining the mechanical structure, sensors, actuators, and computing unit. Starting with the main component, the Turtlebot 2 with Kobuki base, powered by a lithium battery of 14.8V and equipped with a color and depth camera, wheel encoders for odometry, 3 forward bump, 3 cliff, 2 wheel drop auxiliary sensors, and a gyroscope. It is controlled by the Raspberry Pi, which handles serial communication protocols and data exchange at 115200bit/s between the computer and robot, via a USB connection. ROS commands are packed into messages and sent over, while the Raspberry reads and sends back sensor data. The Turtlebot is also equipped with a manipulator arm, the uFactory uArm Swift pro, powered with 12V from the kobuki base power outputs. It is a 4 DoF arm with a suction gripper at the end-effector for picking up objects. The main sensors used for surround-scanning and obstacle avoidance is the Intel RealSense D435i camera with a 85 degree field of view. It uses global shutter cameras for simultaneous pixel capture. The RP Lidar A2 sensor scans environment using laser triangulation.

B. Software Architecture

1) Planner: The planner is responsible for generating a task for the robot to follow in order to achieve its goal. It takes into account the robot states and the task error to define next task for the robot. A task includes task's name, desire, controller gain for each task and feed forward parameters. We create a behavior tree as a state machine for the planner. In Figure 3, planner is *behavior tree node*.

2) Controller: The controller is responsible for executing the tasks generated by the planner, while ensuring that the robot adheres to the desired behavior and dynamics. It receives feedback from the navigation system about its current state, such as position, orientation, and velocity. In this project, controller uses joints position. Based on this feedback and the desired trajectory, the controller computes control commands to adjust the robot's actuators and drive it to do the task. In this project, we use task-priority controller for each joint. In Figure 3, controller is implemented in *TP control node*.

3) Localization: Localization refers to the process of determining a robot's position and orientation within its environment. It's a critical aspect of autonomous navigation, as the robot needs to know where it is relative to its surroundings in order to perform tasks effectively. In this project, to implement the controller, we need to know joint positions of the manipulator and the turtlebot's pose. In this system, we use $\backslash$*turtlebot*$\backslash$*joint_states* topic which contains joint position of the manipulator and encoder data of the turtlebot. With the encoder data, we compute the turtlebot's pose using dead reckoning. In Figure 3, localization is implemented in *EKF node*

4) Perception: Perception refers to detect and determine object position in the environment. In this project, to implement the perception, we use aruco detector through color image. In this system, we use

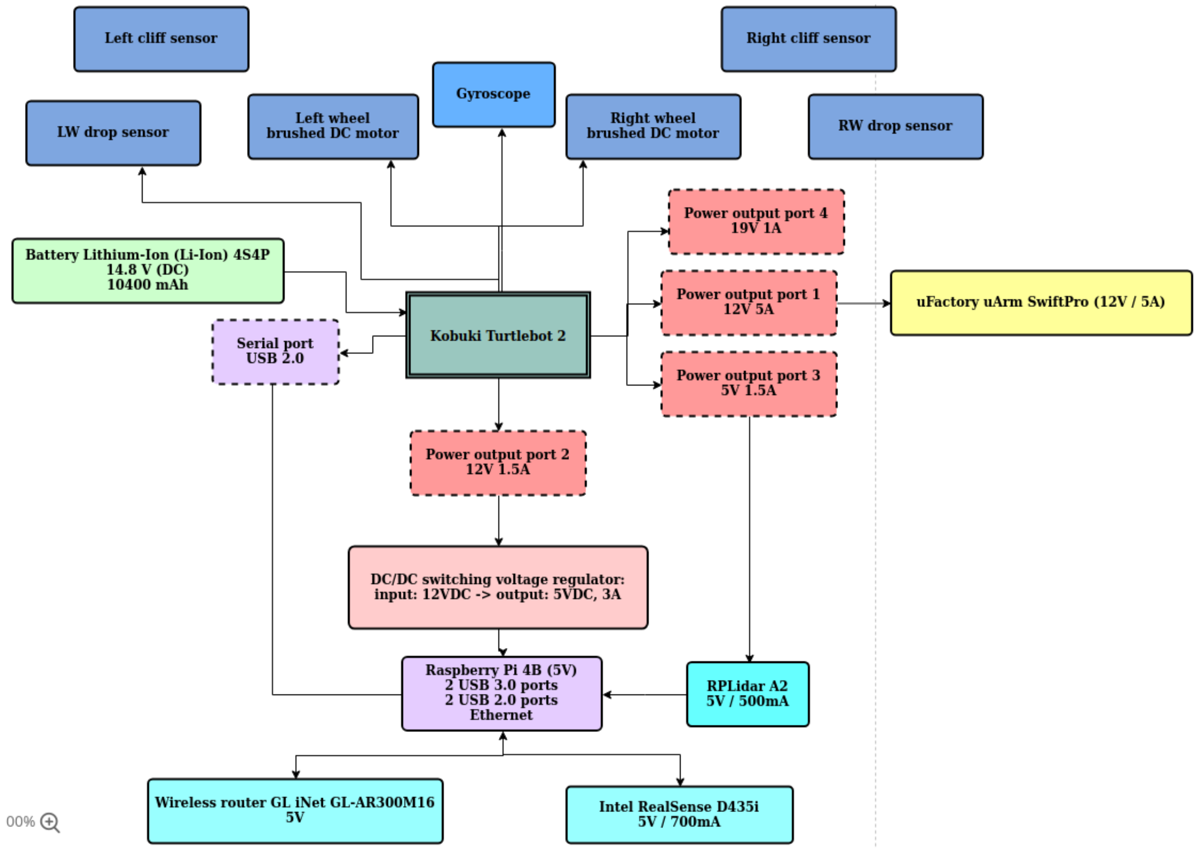
Left cliff sensor
Right cliff sensor
Gyroscope
LW drop sensor
Left wheel brushed DC motor
Right wheel brushed DC motor
RW drop sensor
Power output port 4 19V 1A
Battery Lithium-Ion (Li-Ion) 4S4P 14.8 V (DC) 10400 mAh
Power output port 1 12V 5A
uFactory uArm SwiftPro (12V / 5A)
Serial port USB 2.0
Kobuki Turtlebot 2
Power output port 3 5V 1.5A
Power output port 2 12V 1.5A
DC/DC switching voltage regulator: input: 12VDC -> output: 5VDC, 3A
Raspberry Pi 4B (5V) 2 USB 3.0 ports 2 USB 2.0 ports Ethernet
RPLidar A2 5V / 500mA
00%
Wireless router GL iNet GL-AR300M16 5V
Intel RealSense D435i 5V / 700mA

Fig. 1: Diagram of the hardware architecture

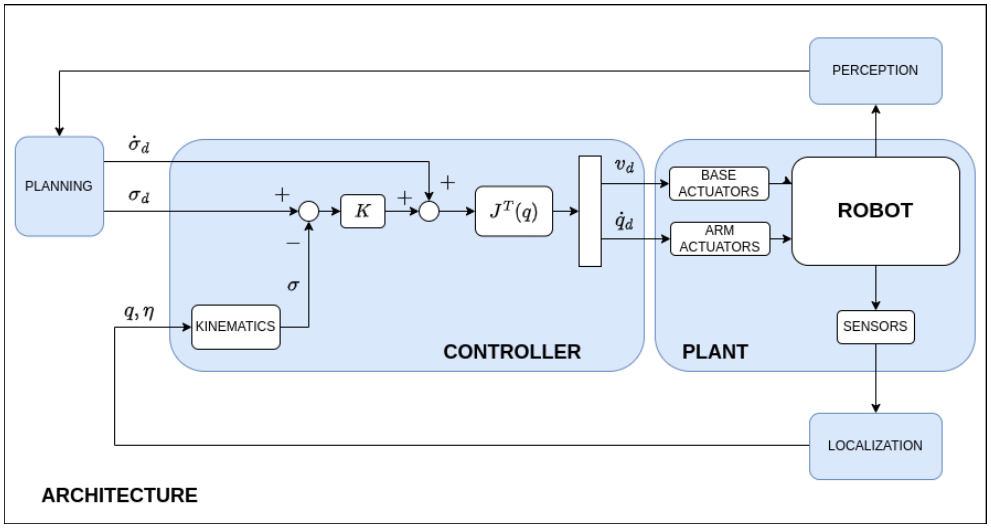

Fig. 2: Diagram of the software architecture

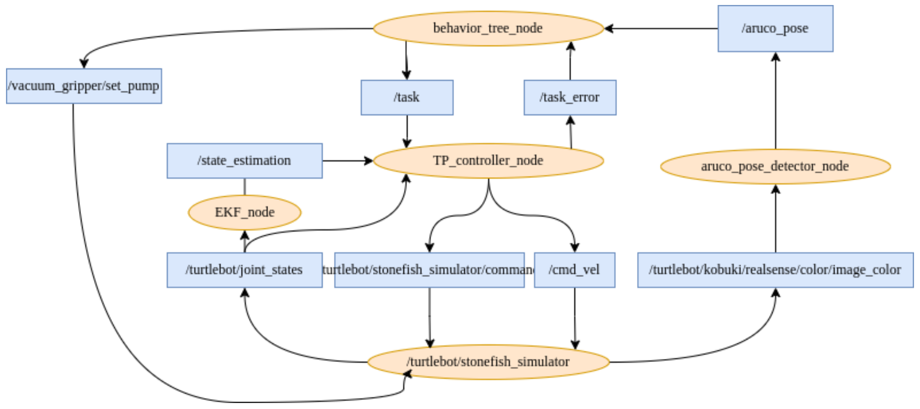

Fig. 3: Node graph

turtlebot\kobuki\realsense\color\image_color topic which contains joint position of the manipulator and encoder data of the turtlebot. With the encoder data, we compute the turtlebot's pose using dead reckoning. In Figure 3, localization is implemented in *aruco_pose_detector_node*

C. Control Architecture

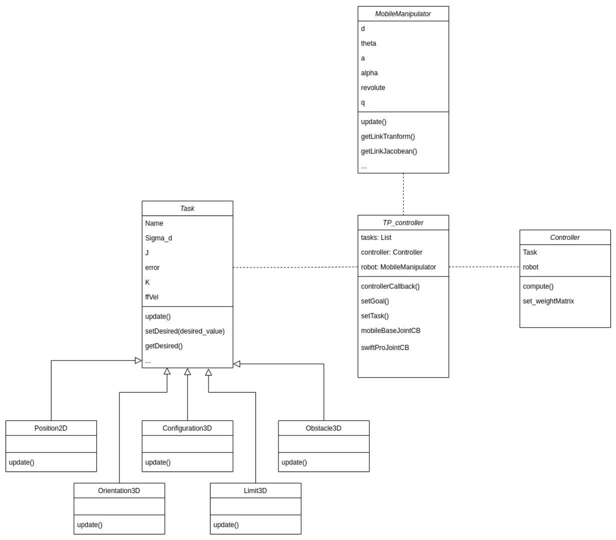

Fig. 4: Control system classes

In Figure **??**, we define classes in the *TP_control_node*. Firstly, `MobileManipulator` class includes methods computing kinematic, Jacobean of the mobile manipulator as well as methods getting and setting parameters of the robot. Secondly, *task* class compute task error from desired and robot states. Thirdly, *controller* class operates the Task-Priority controller to compute control signal for each joint of the robot.

In terms of interfacing with robot architecture, inputs, outputs, message types, it is clear to see from Figure 3. Detailed message types are showed below.

D. Kinematic Calculations

To implement a kinematic task-priority control system, we need a set of equations to determine the precise position

TABLE I: Interfacing of controller with robot architecture

Table	**Topic definition**	
	Topic	***Message Type***
1	/task	taskMsg
2	/state_estimation	Odometry
3	/turtlebot/joint_states	JointState
4	/cmd_vel	Twist
5	joint_velocity_controller/command	Float64MultiArray
6	/task_error	Float64MultiArray

and orientation of the robot's end-effector based on joint parameters. Calculating the system's forward kinematics we obtain the link between desired spatial commands and the corresponding joint actions necessary to achieve them. The Jacobian matrix, which translates changes in joint angles into changes in end-effector position and orientation, facilitates the direct control of the robot's pose andensure s that the movements are optimized according to the designated task priorities. For this 6 DoF mobile manipulator, These computations have been divided between the arm and the mobile base.

1) Forward kinematics of manipulator arm: Since this manipulator forward kinematics could not be directly calculated using the Denavit-Hartenberg method, we opted for a geometrical derivation, based on the schematic drawing provided in the documentation.

To determine the location and orientation of the end-effector with respect to the base frame, we analyze the top view of the manipulator, shown in figure 6. This projection allows to observe how the angle formed by the manipulator base joint J_1, directly affects the position of the end-effector on the XY plane. Where:

- q_1 = joint 1 revolute angle at the base
- q_4 = joint 4 revolute angle at the end-effector
- l = distance between base and end-effector frame

$$\begin{aligned} {}^{0}x_{EE} &= l\cos q_1 \\ {}^{0}y_{EE} &= l\sin q_1 \end{aligned} \tag{1}$$

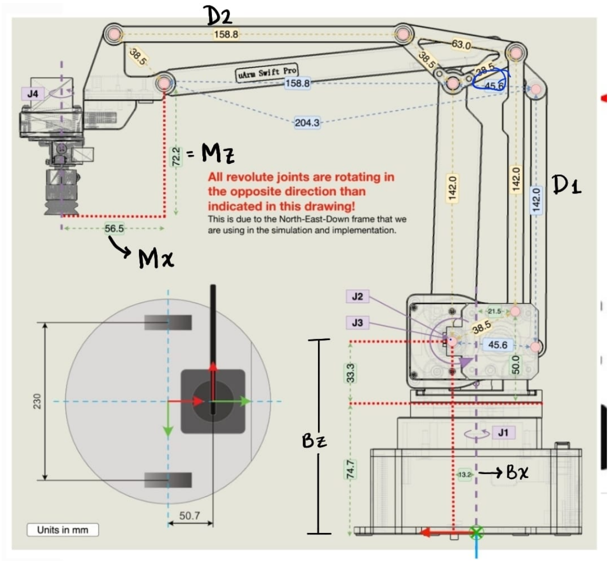
D2
158.8
63.0
38.5
uArm Swift Pro
204.3
45.6
J4
72.2
= Mz
All revolute joints are rotating in the opposite direction than indicated in this drawing!
This is due to the North-East-Down frame that we are using in the simulation and implementation.
142.0
D1
56.5
Mx
J2
J3
21.5
50.0
33.3
Bz
J1
74.7
13.2
Bx
230
50.7
Units in mm

Fig. 5: SwiftPro Manipulator schematic drawing with dimensions

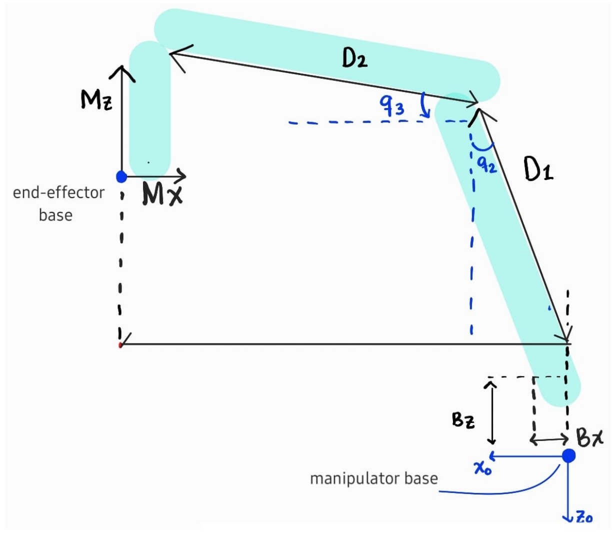

Fig. 7: Side-view on XZ of the SwiftPro manipulator establishing the position of the end-effector relative to the NED frame at the base

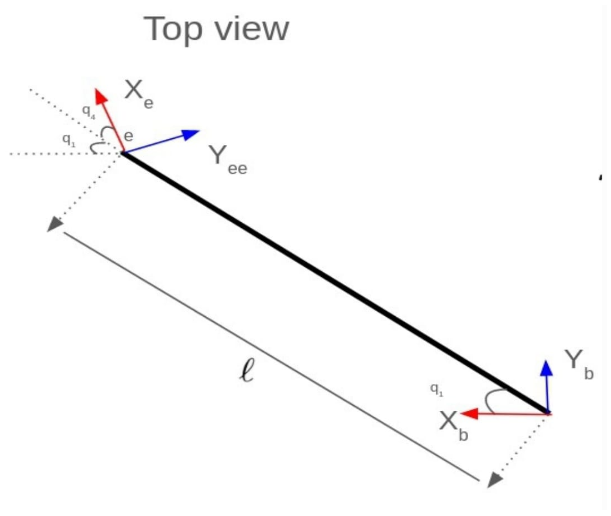

Fig. 6: Top-down view on XY plane of the SwiftPro manipulator establishing orientation relation of end-effector and the NED frame at the base.

Equations 1 below show the relation of end-effector position with respect to base angle.

This projection gives the orientation of the end-effector, which can be computed with the unit vectors going from base to end-effector frame with the same Z axis in common. The xyz components of the orientation is shown by the following equations, which will be later used to the compute the full transformation matrix.

$$\begin{aligned} x_{ee} &= x_0 \cos(\theta_1 + \theta_4) + y_0 \sin(\theta_1 + \theta_4) + z_0 \\ y_{ee} &= -x_0 \sin(\theta_1 + \theta_4) + y_0 \cos(\theta_1 + \theta_4) + z_0 \\ z_{ee} &= 0 \end{aligned} \tag{2}$$

Now that we have established a common variable l between the end-effector and the manipulator origins, we can analyse the side view to now get the Z relation between base and end-effector frames. In this projection the y-axis points towards oneself. The dimensions b_x, b_y, d_1, d_2, m_x, m_z are shown in the specification sheet in figure 5. This will give us the full translation components of the transformation matrix from base to end-effector.

$$^{0}Z_{EE} = -\left(b_z + d_1 \cos(-q_2) + d_2 \sin q_3 - m_z\right) \tag{3}$$

Using the components of the X axis shown on the lateral view of the manipulator, we can derive distance L, shown on the equation below.

$$l = b_x + d_1 \sin(-q_2) + d_2 \cos q_3 + m_x \tag{4}$$

Finally, equations 1, 2, and 3 are used to compute the full Transformation matrix of the SwiftPro manipulator base to end-effector in equation 5:

$$^{0}T_E = \begin{bmatrix} \cos(q_1+q_4) & -\sin(q_1+q_4) & 0 & 0x_{EE} \\ \sin(q_1+q_4) & \cos(q_1+q_4) & 0 & 0y_{EE} \\ 0 & 0 & 1 & 0z_{EE} \\ 1 & 0 & 0 & 1 \end{bmatrix} \tag{5}$$

Then we derive the Jacobian via differentiation of the transformation functions in equation 5. This will render the η pose of the end-effector in terms of the joint states.

$$\begin{bmatrix} x \\ y \\ z \\ \psi \end{bmatrix} = \begin{bmatrix} (b_x + d_1 \sin(-q_2) + d_2 \cos q_3 + m_x) \cos(q_1) \\ (b_x + d_1 \sin(-q_2) + d_2 \cos q_3 + m_x) \sin(q_1) \\ (-b_z - d_1 \cos(-q_2) - d_2 \sin q_3 + m_x) \\ q_1 + q_4 + \theta + \alpha \end{bmatrix} \tag{6}$$

Now, we add mobile base pose to 6. With robot pose $[\eta_x, \eta_y, \theta]$, we rewrite 6 as:

$$\begin{bmatrix} x \\ y \\ z \\ \psi \end{bmatrix} = \begin{bmatrix} l\cos(q_1+\theta+\alpha)+\eta_x+bm_x\cos(\theta) \\ l\sin(q_1+\theta+\alpha)+\eta_y+bm_x\sin(\theta) \\ -(b_z+d_1\cos(-q_2)+d_2\sin(q_3)-m_z)+bm_z \\ q_1+q_4 \end{bmatrix} \tag{7}$$

with α is angle between the mobile base heading and the manipulator heading.

Following the control law for computing the end-effector twist in cartesian in terms of the joint velocities using the system's Jacobian:

$$\dot{x}_E = J(\mathsf{q})\zeta$$

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{z} \\ \dot{\psi} \end{bmatrix} = \begin{bmatrix} \frac{\partial x}{\partial q_1} & \frac{\partial x}{\partial q_2} & \frac{\partial x}{\partial q_3} & \frac{\partial x}{\partial q_4} \\ \frac{\partial y}{\partial q_1} & \frac{\partial y}{\partial q_2} & \frac{\partial y}{\partial q_3} & \frac{\partial y}{\partial q_4} \\ \frac{\partial z}{\partial q_1} & \frac{\partial z}{\partial q_2} & \frac{\partial z}{\partial q_3} & \frac{\partial z}{\partial q_4} \\ \frac{\partial \psi}{\partial q_1} & \frac{\partial \psi}{\partial q_2} & \frac{\partial \psi}{\partial q_3} & \frac{\partial \psi}{\partial q_4} \end{bmatrix} \cdot \begin{bmatrix} \dot{q}_1 \\ \dot{q}_2 \\ \dot{q}_3 \\ \dot{q}_4 \end{bmatrix} \tag{8}$$

With:

$$\begin{aligned} \frac{\partial x}{\partial q_1} &= -l\sin(q_1+\theta+\alpha) \\ \frac{\partial y}{\partial q_1} &= l\cos(q_1+\theta+\alpha) \\ \frac{\partial z}{\partial q_1} &= 0, \frac{\partial \psi}{\partial q_1} = 1 \\ \frac{\partial x}{\partial q_2} &= -d_1\cos(q_2)\cos(q_1+\theta+\alpha) \\ \frac{\partial y}{\partial q_2} &= -d_1\cos(q_2)\sin(q_1+\theta+\alpha) \\ \frac{\partial z}{\partial q_2} &= d_1\sin(q_2) \\ \frac{\partial \psi}{\partial q_1} &= 0 \\ \frac{\partial x}{\partial q_3} &= -d_2\sin(q_3)\cos(q_1+\theta+\alpha) \\ \frac{\partial y}{\partial q_3} &= -d_2\sin(q_3)\sin(q_1+\theta+\alpha) \\ \frac{\partial z}{\partial q_3} &= -d_2\cos(q_3) \\ \frac{\partial \psi}{\partial q_3} &= 0 \\ \frac{\partial x}{\partial q_4} &= 0, \frac{\partial y}{\partial q_4} = 0 \\ \frac{\partial z}{\partial q_4} &= 0, \frac{\partial \psi}{\partial q_4} = 1 \end{aligned} \tag{9}$$

2) Forward kinematics of mobile base: For kinematics of the mobile base, we assume the base movement including 2 moving translate then rotation. Hence, we can apply DH to this component with DH table II:

	θ	**d**	**a**	α
1	90	bmz	0	90
2	0	bmx	0	-90
3	q_1-90	0	1	0

TABLE II: DH table of the mobile base

3) Kinematic of Mobile Manipulator: Combined Jacobean matrix of mobile base in II-D2 and manipulator in II-D1, we have Jacobean for mobile manipulator:

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{z} \\ \dot{\psi} \end{bmatrix} = \begin{bmatrix} \frac{\partial x}{\partial m_1} & \frac{\partial x}{\partial m_2} & \frac{\partial x}{\partial q_1} & \frac{\partial x}{\partial q_2} & \frac{\partial x}{\partial q_3} & \frac{\partial x}{\partial q_4} \\ \frac{\partial y}{\partial m_1} & \frac{\partial y}{\partial m_2} & \frac{\partial y}{\partial q_1} & \frac{\partial y}{\partial q_2} & \frac{\partial y}{\partial q_3} & \frac{\partial y}{\partial q_4} \\ \frac{\partial z}{\partial m_1} & \frac{\partial z}{\partial m_2} & \frac{\partial z}{\partial q_1} & \frac{\partial z}{\partial q_2} & \frac{\partial z}{\partial q_3} & \frac{\partial z}{\partial q_4} \\ \frac{\partial \phi}{\partial m_1} & \frac{\partial \phi}{\partial m_2} & \frac{\partial \phi}{\partial q_1} & \frac{\partial \phi}{\partial q_2} & \frac{\partial \phi}{\partial q_3} & \frac{\partial \phi}{\partial q_4} \\ \frac{\partial \theta}{\partial m_1} & \frac{\partial \theta}{\partial m_2} & \frac{\partial \theta}{\partial q_1} & \frac{\partial \theta}{\partial q_2} & \frac{\partial \theta}{\partial q_3} & \frac{\partial \theta}{\partial q_4} \\ \frac{\partial \psi}{\partial m_1} & \frac{\partial \psi}{\partial m_2} & \frac{\partial \psi}{\partial q_1} & \frac{\partial \psi}{\partial q_2} & \frac{\partial \psi}{\partial q_3} & \frac{\partial \psi}{\partial q_4} \end{bmatrix} \cdot \begin{bmatrix} \dot{m}_1 \\ \dot{m}_2 \\ \dot{q}_1 \\ \dot{q}_2 \\ \dot{q}_3 \\ \dot{q}_4 \end{bmatrix} \tag{10}$$

With m_1 and m_2 are the base joint while manipulator joint are q_1, q_2, q_3, q_4.

III. Implementation

A. Designed kinematic control system

In terms of the detailed implementation block diagram for the designed kinematic control system, we implemented only one node which is *TP_control_node* with classes are defined in 4 and ROS messages are defined in I.

B. Task-Priority redundancy resolution algorithm

After obtaining the forward kinematics of the 6 DoF mobile manipulator, a recursive task-priority algorithm is implemented to handle a sequence of motions of the joints following different tasks, each holding a certain priority of execution determined by a weighted matrix, discussed later on this document.

This algorithm ensures that higher priority tasks are given precedence, while executing also lower priority tasks in the available joint space. The recursive Task-Priority algorithm takes the previous list of tasks as an input, initialising the joint velocities as a column vector with *i* rows depending on the DoF. Following that, initialise the null-space projector *P* as an identity matrix of dimensions *i* x *i* . Where *i* is the number of DoF. The main steps of the algorithm are:

1) Iteration over tasks.
2) Update of task state.
3) Compute augmented Jacobian $\bar{J}$.

$$\zeta_i = \zeta_{i-1} + \overline{J_i^{\dagger}}(q)\left(\dot{x}_i(q) - J_i(q)\dot{\zeta}_{i-1}\right)$$

4) Compute task velocity $\dot{q}$:

$$P_i = P_{i-1} - \overline{J_i^{\dagger}}(q)\overline{J_i}(q)$$

To manage the joint limit task, a distinct method is adopted. To achieve the robot's target position without breaching joint

Algorithm 1 Task Priority Algorithm

```
Require: Desired positions/orientations for all tasks $\mathbf{d}_i$, Gain matrices $K_i$, Feedforward terms $\mathbf{f}_i$
Ensure: Joint velocities $\dot{\mathbf{q}}$
 1: Initialize $\mathbf{q} \leftarrow$ initial joint configuration
 2: Initialize $\mathbf{J}_i$ for all tasks
 3: Initialize $\mathbf{e}_i \leftarrow \mathbf{d}_i - \mathbf{x}_i(\mathbf{q})$ for all tasks
 4: for each task $i$ do
 5:     Compute task Jacobian $\mathbf{J}_i$
 6:     Compute task error $\mathbf{e}_i$
 7:     Compute task velocity $\dot{\mathbf{x}}_i = K_i\mathbf{e}_i + \mathbf{f}_i$
 8:     if $i == 1$ then
 9:         Compute $\dot{\mathbf{q}}_i$ as the solution to $\mathbf{J}_i\dot{\mathbf{q}}_i = \dot{\mathbf{x}}_i$
10:     else
11:         Compute null-space projection matrix for task $i-1$:
                $\mathbf{N}_{i-1} = \mathbf{I} - \mathbf{J}_{i-1}^{\dagger}\mathbf{J}_{i-1}$
12:         Update desired task velocity in the null space of the previous tasks:
                $\dot{\mathbf{q}}_i = \mathbf{N}_{i-1}\dot{\mathbf{q}} + \mathbf{J}_i^{\dagger}(\dot{\mathbf{x}}_i - \mathbf{J}_i\mathbf{N}_{i-1}\dot{\mathbf{q}})$
13:     end if
14:     Update joint velocities $\dot{\mathbf{q}} \leftarrow \dot{\mathbf{q}}_i$
15: end for
16: return $\dot{\mathbf{q}}$
```

limits, a Damped Least Squares (DLS) weighting strategy is implemented. This strategy adjusts the DLS weighting based on the robot's proximity to the target. When the robot is distant from the goal, the focus is on maintaining the arm's position while allowing unrestricted movement. As the robot nears its goal, the DLS weighting shifts to prioritise arm movement and reduce the overall speed of the robot. This method effectively prevents violations of joint limits and ensures the robot reaches its goal smoothly and precisely.

C. Description of the performed Tasks

During the implementation phase following tasks were defined and developed by inheriting the Base *Task* Class. All the developed child classes have two methods *update() and track_err()*. The update method updates the Jacobian and the error of the task at every iteration depending on the current configuration of the robot. This updated jacobian and error is then fed into the main Recursive Task Priority algorithm to calculate the control input for that task. The track error method saves the evolution of the error at every iteration. Depending on the objective of the task, each task was given a priority as follows:

- Joint limits
- End-effector position
- End-effector orientation
- End-effector configuration
- Mobile base orientation
- Mobile base position
- Mobile base configuration

1) Mobile Base Position: The Mobile Base position task controls the base of the mobile manipulator system. The desired value for this task is the 2D position (x, y) in the world frame. The Jacobian for this task is the first two rows of the entire 6×6 Jacobian Matrix $J(\theta)$ of the mobile base defined earlier 10.

The Jacobian matrix J is the first two rows of the entire Jacobian matrix J_{MB}:

$$J = \begin{pmatrix} J_{11} & J_{12} & \cdots & J_{1n} \\ J_{21} & J_{22} & \cdots & J_{2n} \end{pmatrix}$$

where, in our case n is 6. The task error is given by:

$$\mathbf{err} = K(\mathbf{desired} - \mathbf{position}) + \mathbf{feedforward}$$

Where, K is the gain of the task set when the task was instantiated. The feedforward term is zero as the tracking objects are not in the scope of the project. The feedforward term is given the estimated veloctiy of the object for tracking task.

2) Mobile Base Orientation: This task controls the orientation of the mobile base which has a single degree of freedom in our case i.e yaw. The task takes in the desired orientation angle in the world frame and calculates the jacobian and Error. The Jacobian matrix J is the last row of the entire Jacobian matrix J_{MB}:

$$J = \begin{pmatrix} J_{n1} & J_{n2} & \cdots & J_{nn} \end{pmatrix}$$

Where, n is 6. Additionally, The task error is given by:

$$\mathbf{err} = K \cdot \text{normalize\_angle}(\mathbf{desired} - \mathbf{orientation})$$

Where, K is the gain of the task set when the task was instantiated. The function normalize_angle ensures the angle difference is within a standard range, $[-\pi, \pi]$

3) Mobile Base Configuration: This task combines the Position and Orientation task explained in the earlier sections. The Jacobian matrix J is composed of the 0th, 1st, and 5th rows of the entire Jacobian matrix of the Mobile Base J_{MB}:

$$J = \begin{pmatrix} J_{01} & J_{02} & \cdots & J_{0n} \\ J_{11} & J_{12} & \cdots & J_{1n} \\ J_{51} & J_{52} & \cdots & J_{5n} \end{pmatrix}$$

The position error is given by:

$$\mathbf{err}_{\text{pos}} = \mathbf{desired}_{\text{pos}} - \mathbf{position}$$

The orientation error is given by:

$$\mathbf{err}_{\text{ori}} = \mathbf{desired}_{\text{ori}} - \mathbf{orientation}$$

The combined task error is:

$$\mathbf{error}_{\text{task}} = \begin{pmatrix} \|\mathbf{err}_{\text{pos}}\| \\ \mathbf{err}_{\text{ori}} \end{pmatrix}$$

The overall error is calculated as:

$$\mathbf{err} = K \begin{pmatrix} \mathbf{err}_{\text{pos}} \\ \mathbf{err}_{\text{ori}} \end{pmatrix} + \mathbf{feedforward}$$

4) EE Position: This task controls the EE position combining the all the mobile manipulator degrees of freedom. The Jacobian matrix J is the first three rows of the entire end-effector Jacobian matrix J_{EE}:

$$J = \begin{pmatrix} J_{11} & J_{12} & \cdots & J_{1n} \\ J_{21} & J_{22} & \cdots & J_{2n} \\ J_{31} & J_{32} & \cdots & J_{3n} \end{pmatrix}$$

The task error is given by:

$$\mathbf{err} = K(\mathbf{desired} - \mathbf{position}) + \mathbf{feedforward}$$

5) EE Orientation: The end effector orientation only has one degree of freedom which is the yaw of EE frame. The row controlling the yaw of the EE is the last row of the entire EE configuration jacobian. Thus, The Jacobian matrix J is the last row of the entire end-effector Jacobian matrix J_{EE}:

$$J = \begin{pmatrix} J_{n1} & J_{n2} & \cdots & J_{nn} \end{pmatrix}$$

The task error is given by:

$$\mathbf{err} = K(\mathbf{desired} - \mathbf{orientation}) + \mathbf{feedforward}$$

6) EE Configuration: This task combines the position and orientation tasks elaborated earlier. The Jacobian matrix J is composed of the 0th, 1st, 2nd, and 5th rows of the entire end-effector Jacobian matrix J_{EE}:

$$J = \begin{pmatrix} J_{01} & J_{02} & \cdots & J_{06} \\ J_{11} & J_{12} & \cdots & J_{16} \\ J_{21} & J_{22} & \cdots & J_{26} \\ J_{51} & J_{52} & \cdots & J_{56} \end{pmatrix}$$

The position error is given by:

$$\mathbf{err}_{\text{pos}} = \mathbf{desired}_{\text{pos}} - \mathbf{position}$$

The orientation error is given by:

$$\mathbf{err}_{\text{ori}} = \text{normalize\_angle}(\mathbf{desired}_{\text{ori}} - \mathbf{orientation})$$

The overall error is calculated as:

$$\mathbf{err} = K \begin{pmatrix} \mathbf{err}_{\text{pos}} \\ \mathbf{err}_{\text{ori}} \end{pmatrix} + \mathbf{feedforward}$$

7) Joint Limits Task: This task is an inequality task which does not allow a perticular joint to go past its limit. The inequality tasks, unlike normal tasks, are only activated if a given condition is not valid e.g- Joint reaching limits or EE position too close to an obstacle. .These task activations are taken care of with the help of activation functions. The activation functions are stored as a task class attribute, and the default value is given 1, meaning all the tasks by default are active. This attribute is overridden in inequality subclasses when defined. A few more methods are added to the base classes that offer access and manipulation of activation functions.

Where the 1 is at the position corresponding to the joint of the task limit.

The `Limit2D` class in the provided code defines a task for limiting the position of a specific joint within a desired range. The Jacobian matrix J for this task is defined as a row vector with a single 1 at the position corresponding to the specified joint , indicating that only this joint's position affects the task. The error in this context is fixed at a small value (0.05), scaled by the gain matrix K. This error is updated as $\mathbf{err} = K \cdot 0.05$, where K is the gain matrix.

The Jacobian matrix J for the task is:

$$J = \begin{pmatrix} 0 & 0 & \cdots & 1 & \cdots & 0 \end{pmatrix}$$

The activation function determines whether the task is active or not based on the joint position q_i and the specified thresholds. Initially, if the task is not active and the joint position exceeds the upper limit minus a threshold, the task becomes active with a state of -1. Conversely, if the joint position falls below the lower limit plus a threshold, the task becomes active with a state of 1. If the task is active and the joint position moves back within the specified thresholds, the task becomes inactive (state 0).

D. Sequence of motions task

In this task, we need to control End-Effector position from point to point. To do and test it, we subcribe */move_base_simple/goal* to get desired position from Rviz for this task.

E. Pick and Place task

In terms of pick and place task, we use a behavior tree for the planner as a state machine. In Figure 8, there are 8 stages:

1) Scan Object: the robot does base orientation task to rotate until scan an object.
2) Approach Base Object: the robot does base configuration task to approach the object while the object is still in the camera frame. Until distance between the robot and object is less than a threshold, the robot move to next behavior.
3) Approach Manipulator Object: the robot does End-Effector position task to approach the object from the top. The end-effector moves to a point which is above the object a safe distance before decrease height to

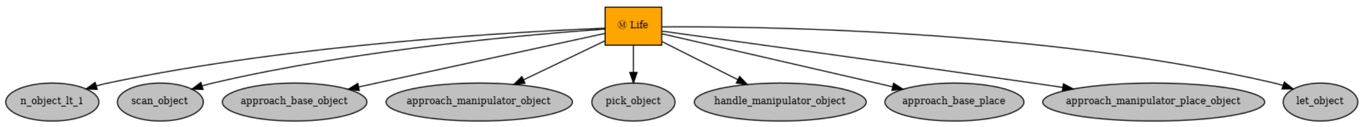

Fig. 8: Behavior tree of the pick and place task.

approach object. When end-effector height satisfy condition, the robot move to next behavior.

4) Pick Object: the robot call vaccum service to pick object.
5) Handle manipulator object: the robot lifts object up to a safe height.
6) Approach Base Place: the robot does base configuration task to go to place space.
7) Approach Manipulator Place Object: the robot does end-effector position task to place object on the groound.
8) Let Object: the robot call vaccum service to let object.

IV. Results and Discussion

A. Task performance

1) Base position task: For the implementation of this task, the base was given a desired (x, y) position in the world frame. In the figure 13, it can be observed that the robot was able achieve the base desired base position shown by blue trajectory. The error evolution is shown in the figure 14. The controller was able to converge the error to zero upon receiving a desired position. The control input calculated by the TP controller is plotted in the figure 17

2) Base orientation task: Figure 16 illustrates the error evolution in the base orientation task for various desired orientations. The control inputs generated by the controller, depicted in Figure 17, rapidly reduced the error to zero. Since this is the base orientation task, only angular input was provided by the controller to the robot.

3) Base configuration task: The base configuration task contains the Base position as well as orientation tasks. The trajectory followed by the robot is shown in figure 18. The error evolution for this task, shown in figure 19, converges to zero. The Linear and angular velocities provided by the controller are plotted in the figure 20.

4) End Effector position task: In this task, the entire mobile manipulator system was used to reach the desired EE position. The error evolution presented in figure 22, converges to zero quickly after giving the desired value. The linear velocities and angular velocities commanded by the controller are plotted in figure 23

5) End Effector configuration task: The end effector position and orientation tasks are combined here for the EE configuration. The trajectory followed by the EE is shown in figure 24 by blue with the desired position in red. The errors in the positions converge to zero as shown by the figure 25.

6) Joint Limit: Whenever the robot joint limits are breached, the joint limit tasks get activated and the task tries to bring the joint back within the limits. This behaviour can be observed in the figure 28, in which the joint limits are exceeded but the controller tried to bring them back within limits. This task ensured that the joints did not go beyond the set limits.

B. Sequence of motions task

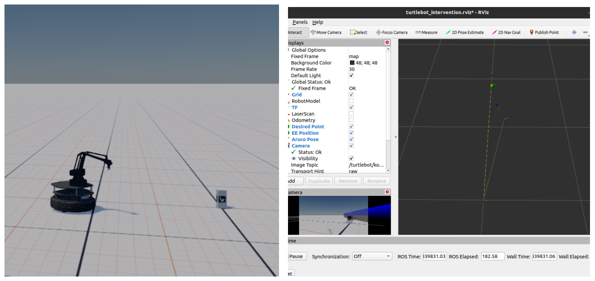

Fig. 9: Sequence of motions task in StoneFish simulation

In this task, we use goal point picked from Rviz to move the robot. Whenever user picks a new desired point from Rviz, the robot does End-effector position task to reach the desired point. In Figure 9, we plot red point is the desired point and green point is the end effector point. It is clear that end-effector reached the desired point. By this way, we can choose desired points from Rviz.

C. Pick and Place task

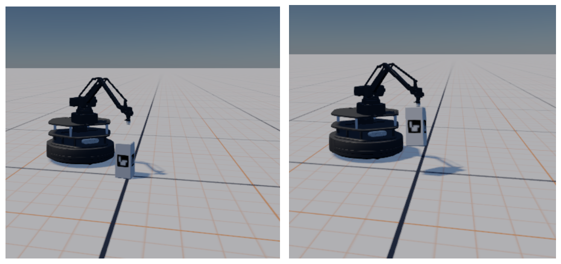

Fig. 10: Pick and Place task in StoneFish simulation

I implemented our system on the StoneFish simulation and the real turtlebot in the lab environemt.

In Figure 12, there are 4 steps. Firstly, from frame 0 to 40, the robot did base configuration task to approach object. It is clear that the robot moved close to object when task error converged to 0. Secondly, from frame 50 to 105, robot did end-effector configuration task to move to a point above the object. At frame 115, Z position error pick to 0.2 meters means the robot started to move end-effector to object from above. Thirdly, in frame 141, there is another pick in Z position error but it is position. Clearly, the robot lifted the object up.

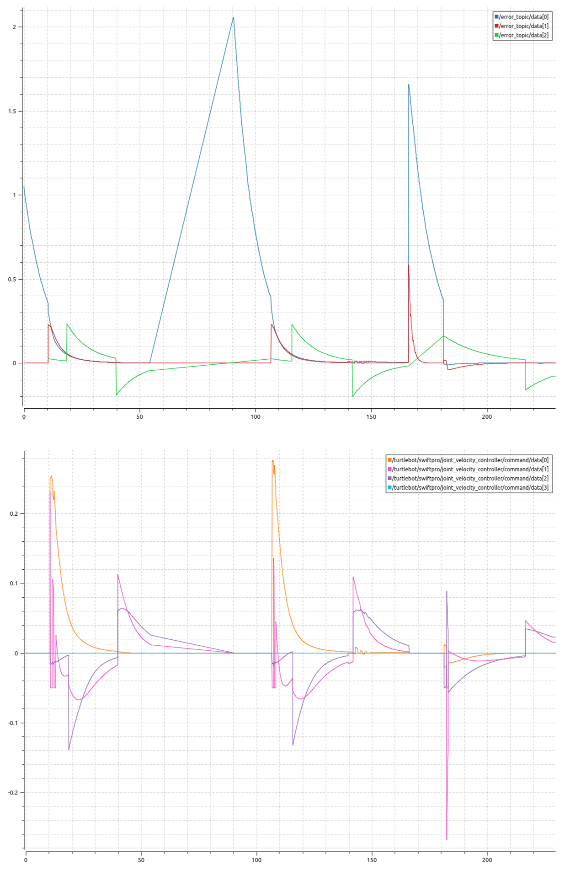

Fig. 11: Task error (First) and Manipulator control (Second) when the robot doing pick and place task. In task error figure, position error in X is blue line, Y is red line and green line is Z error.

Fourthly, frame 166 to the 180, the robot moved to placing space. After that, the robot place object on the floor. Finally, the robot moved end-effector up and finished the task.

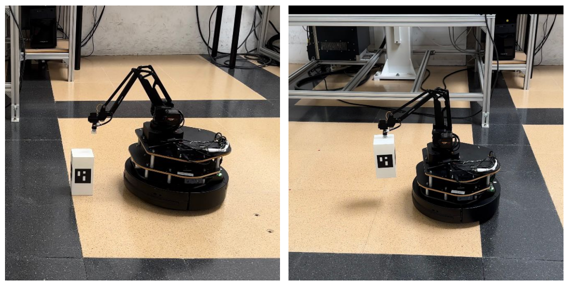

Fig. 12: Pick and Place task in real turtlebot.

Besides, we implemented our system on the real turtlebot and the robot can finished the pick and place task. However, some differences exist between simulation and reality.

- We need to tune again the control parameters. This is not due to the accuracy of the system dynamics model but to the robot's actuator dynamic.
- There exists a difference in mounting angle between the mobile base and the manipulator. While for the simulation this angle is -90 deg, for the real robot this angle is 90 deg. Therefore, we added a parameter to the robot's kinematic model: the mounting angle. When the system operates in simulation mode, this angle is set to -90 deg and vice versa. For the convenience of running the system, we provide the only parameter that needs to be set: the operating mode: SIL or HIL.

V. Conclusion

This study successfully developed a Task-Priority Kinematic Control algorithm for the Kobuki Turtlebot and a 6-DoF mobile manipulator equipped with a vacuum gripper for pick and place tasks. The integration of forward and inverse kinematics enabled precise control over the manipulator's end-effector, optimizing task execution. The implemented system demonstrated improved accuracy, robust kinematic solutions, and efficient task management. Future work will focus on expanding capabilities, incorporating advanced sensors, and applying machine learning for enhanced performance in dynamic environments. This research advances mobile manipulation, providing a robust framework for diverse robotic applications.

APPENDIX

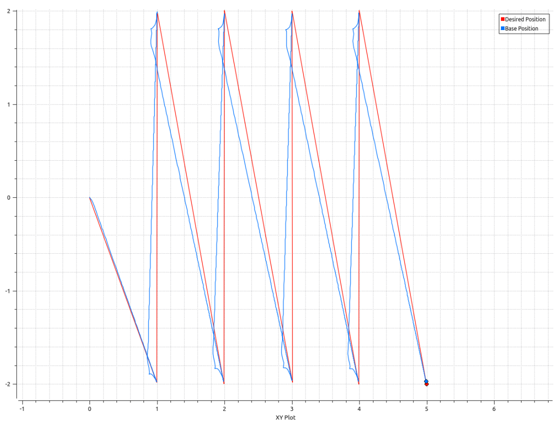

Fig. 13: Base position in XY plane with base position task (Desired position in red, Base position in blue)

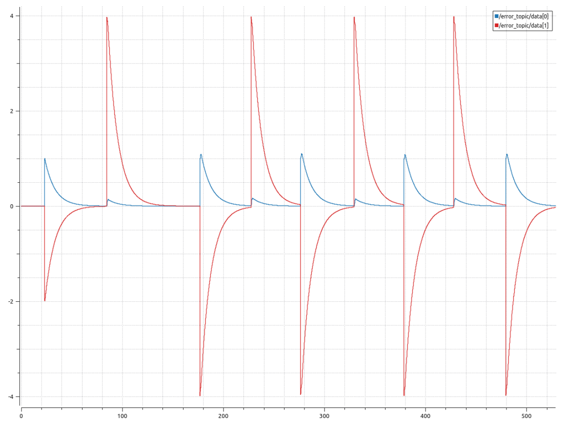

Fig. 14: Task error with base configuration task (error in X represented by blue and y by red)

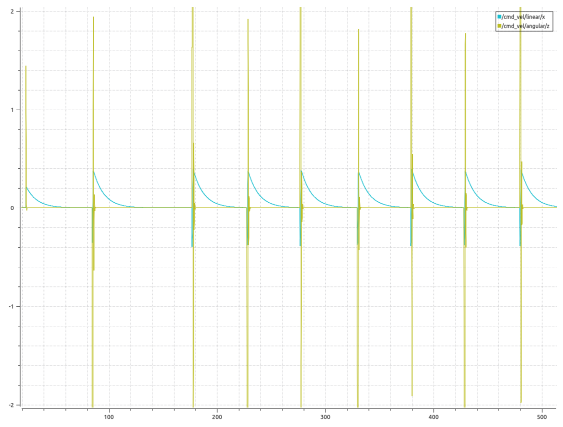

Fig. 15: Linear velocity in blue and angular velocity in yellow of the base with base position task.

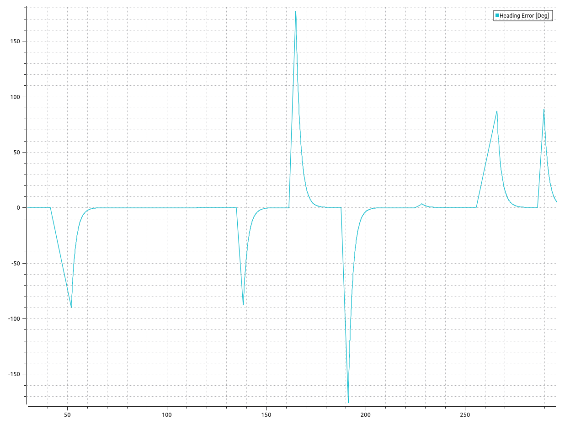

Fig. 16: Task error with base orientation task.

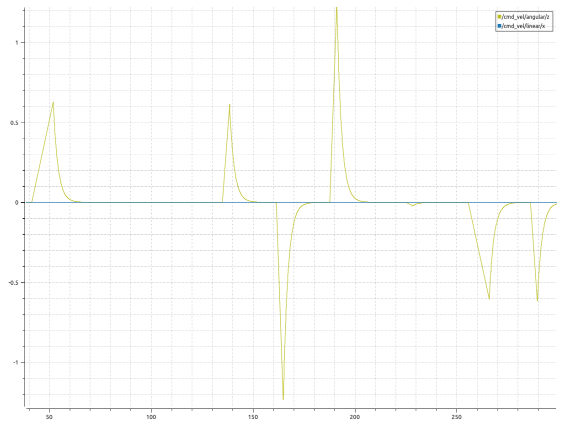

Fig. 17: Linear velocity in blue and angular velocity in yellow of the base with base orientation task.

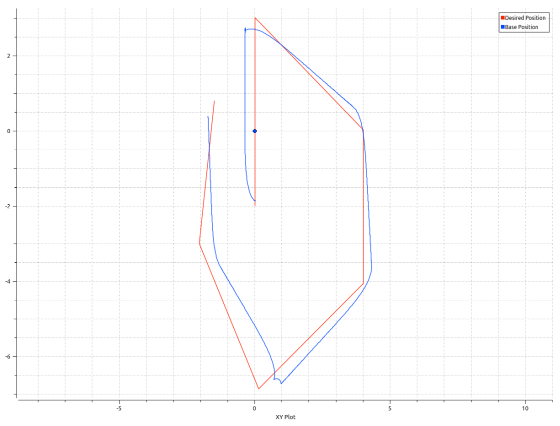

Fig. 18: Base position in XY plane with base configuration task (Desired configuration in red, Base Configuration in blue)

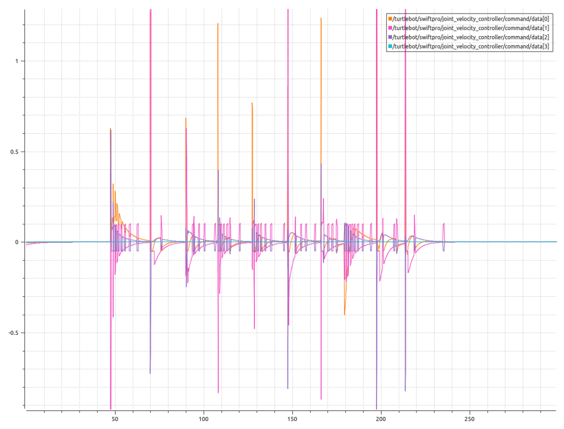

Fig. 21: Joint Velocities for End Effector Position task

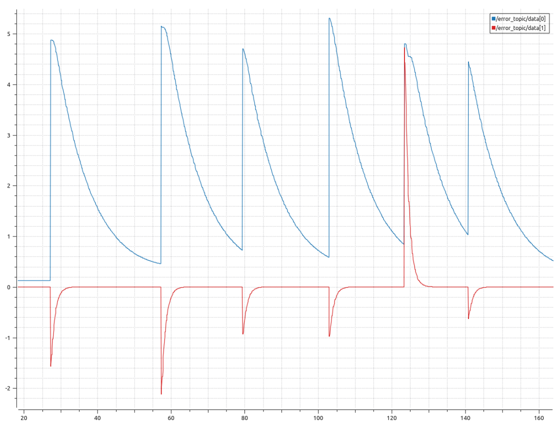

Fig. 19: Task error with base configuration task (error in X represented by blue and y by red)

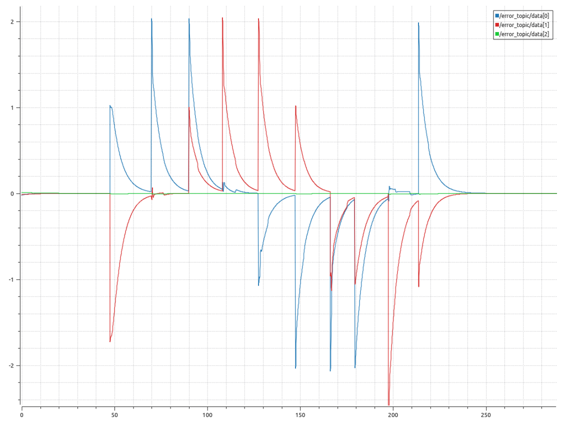

Fig. 22: Task error with End Effector position task. (Errors in x, y, and z represented by blue, red, green respectively)

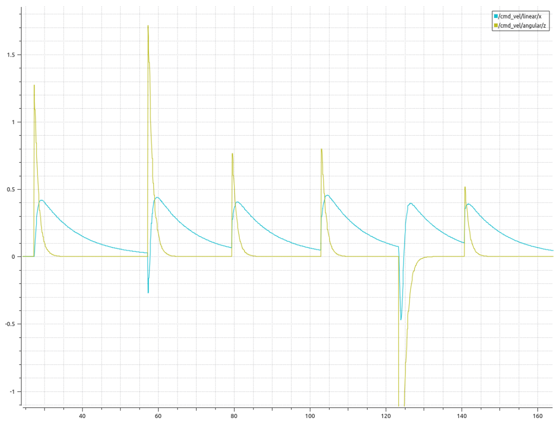

Fig. 20: Linear velocity in blue and angular velocity in yellow of the base with base configuration task.

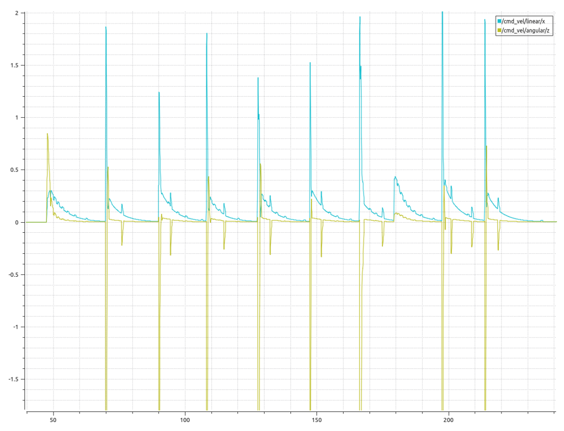

Fig. 23: Linear velocity in blue and angular velocity in yellow of the base with End Effector position task.

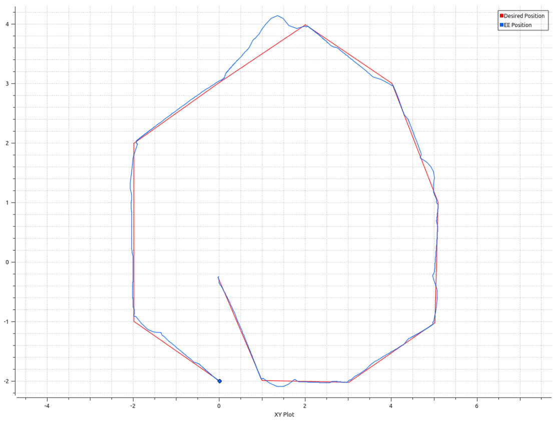

Fig. 24: End Effector position in blue Vs. Desired End Effector position in red

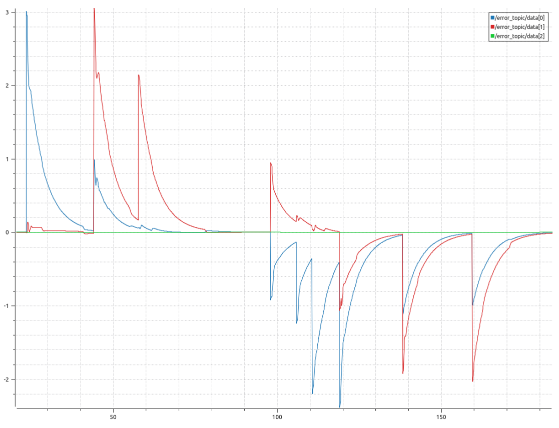

Fig. 25: Task error with End Effector configuration task.

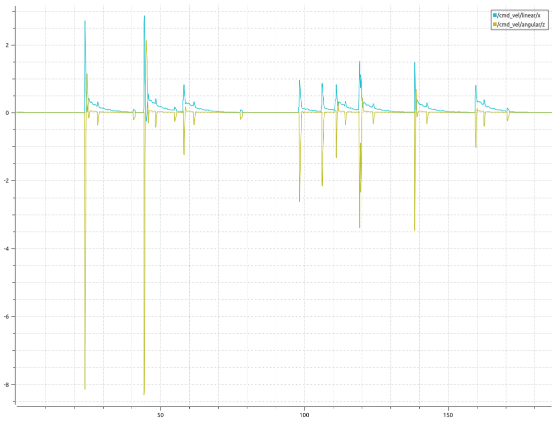

Fig. 26: Linear velocity and angular velocity of the base with End Effector position task.

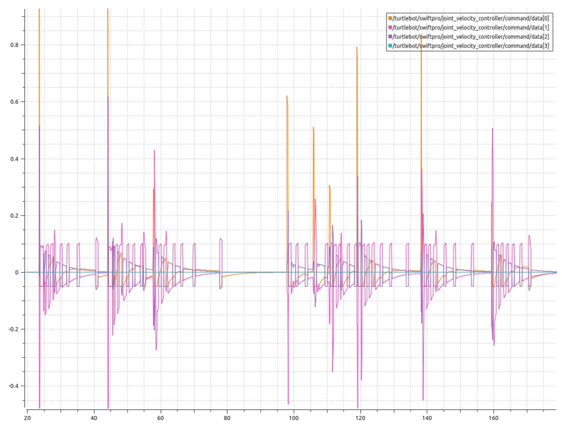

Fig. 27: Manipulator joint velocity with End Effector position task.

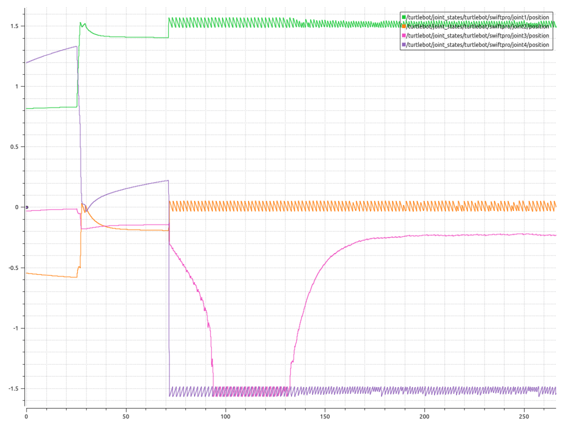

Fig. 28: Joint position with joint limit task.

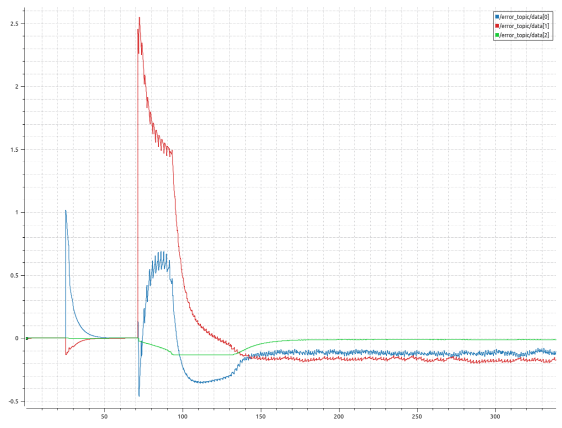

Fig. 29: Task error with joint limit task.