A kinematic Control System for a Mobile Manipulator, based on the Task-Priority Redundancy Resolution Algorithm

Moses Ebere, *Member, IEEE,* Joseph Adeola, *Member, IEEE,* and Preeti Verma, *Member, IEEE,*

Abstract—This report describes a kinematic control system derived and implemented on a differential-drive robot (Kobuki Turtlebot 2), fitted with a 4 DOF manipulator (uFactory uArm Swift Pro). The system is predicated on the task-priority redundancy resolution algorithm. The implementation is done both in simulation and on the real hardware. Additionally, the developments made are integrated with perception, localization, mapping, planning, and exploration modules.

Index Terms—Task-priority, Kinematic Control, Task Jacobians.

1 Introduction

Mobile manipulation has become a major topic of interest in the robotics community in recent years. There is a plethora of applications that can benefit from the advantages of mobile manipulation. Such applications include material handling, search-and-rescue, inventory management, etc. Therefore, there has been increased traction in research along this line.

1.1 Project Objective

The primary objective of this research is to demonstrate the feasibility and performance of a kinematic control system that is underpinned by the task priority redundancy resolution algorithm in uniquely prepared simulation environments and on the real mobile manipulator. Through the implementation and evaluation of the proposed methodology, valuable insights and results will significantly contribute to the understanding and advancement of mobile manipulation.

2 Conceptual Design

This section begins with a close examination of the hardware architecture of the mobile manipulator, followed by an overview of the software architecture in the simulation setup and the proposed functional block diagram of the kinematic control system.

2.1 Hardware Architecture

The table below collates the most important components of the hardware setup.

- *Moses Ebere, Joseph Adeola, and Preeti Verma are Erasmus Mundus Master's students in Intelligent Field Robotic Systems at the University of Girona, Spain.*
E-mail: moseschukaebere@gmail.com, adeolajosepholoruntoba@gmail.com, preeti8600verma@gmail.com

Manuscript submitted on June 12, 2023.

	Component
1	Kobuki Turtlebot 2
2	Raspberry Pi 4B
3	uFactory uArm SwiftPro
4	Intel RealSense D435i
5	RPLidar A2
6	Wireless router GL iNet GL-AR300M16
7	DC/DC switching voltage regulator: input: 12VDC - output: 5VDC, 3A
8	Battery Lithium-Ion (Li-Ion) 4S4P

TABLE 1: Hardware Components

A high-level block diagram of the robot's hardware architecture is included in the appendix (see fig. 22).

2.2 Software Architecture

Included in the appendix are block diagrams (see fig. 23 and 24) presenting the software architecture of the simulation setup prepared for the robotic platform, including the utilized nodes, topics, and services. The first diagram gives an intuitive view of the ROS interconnections.

2.3 Kinematic Control System

The designed kinematic control system is encapsulated in fig. 25. It is a detailed functional block diagram including all necessary components of the control system, interfacing with robot architecture, inputs, outputs, and message types.

3 Kinematics and Task-priority Formulations

In this section, a symbolic derivation of the forward and inverse kinematics of the experimental system is carried out. This is done in two separate formulations - derivations for the uFactory uArm SwiftPro manipulator only and derivations for the mobile base and the manipulator combined. In the latter portion of this section, the task priority (TP) redundancy resolution algorithm is introduced alongside the different tasks considered in this project.

3.1 Kinematics

The generalized coordinates of the mobile manipulator system can be defined as follows:

$$\mathbf{q} = [\boldsymbol{\eta}\, q]^{\mathsf{T}} \tag{1}$$

where η is the generalized coordinates vector of the Kobuki Turtlebot 2 (pose vector) and q is the generalized coordinate vector of the 4DOF uArm (configuration vector):

$$\boldsymbol{\eta} = [x\ y\ \theta]^{\mathsf{T}} \tag{2}$$

$$\mathbf{q} = [q_1\ q_2\ q_3\ q_4]^{\mathsf{T}} \tag{3}$$

3.1.1 Manipulator-Only Derivation

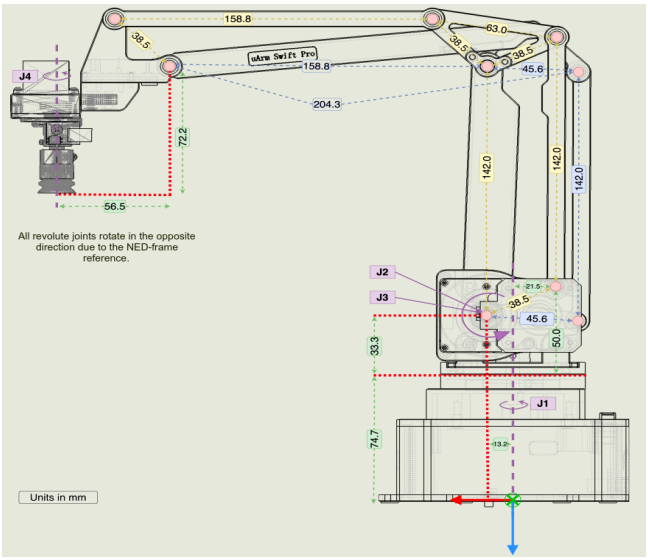
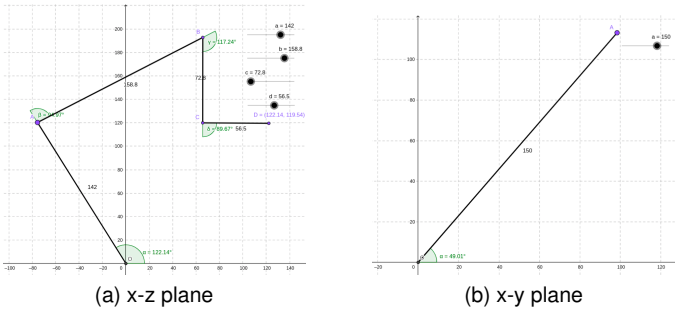


Fig. 1: Dimensions of the Robotic Arm

The uArm Swift Pro (as seen in fig. 1) is based on a parallel mechanism; therefore, the kinematics were derived geometrically using trigonometry. The frame of reference used is the North East Down (NED) N-frame and it is placed at the center of the arm's base which is at an offset of 50.7mm along the x-axis of the base. The cartesian position of the end-effector $[{}^{A}E_x\ {}^{A}E_y\ {}^{A}E_z]$ in the aforementioned frame is obtained as in the figures 2a & b.



(a) x-z plane (b) x-y plane

Fig. 2: (a), (b) Forward Kinematics from Geometry.

The resulting equations from the above geometry are:

$${}^{A}E^{*} = 158.8cos(q_3) - 142sin(q_2) + 56.5 + 13.2 \tag{4}$$

$${}^{A}E_x = E^{*}cos(q_1) \tag{5}$$

$${}^{A}E_y = E^{*}sin(q_1) \tag{6}$$

$${}^{A}E_z = 72.2 - 108 - 142cos(q_2) - 158.8sin(q_3) \tag{7}$$

where ${}^{A}E^{*}$ is the distance between the reference frame and the end-effector, irrespective of the configuration of the arm. The forward kinematics of other degrees of freedom are included in the appendix. The constants are in millimeters, so they should be converted to meters before the calculation. Also, ${}^{A}X$ means that the values are in the frame of the arm's base; ${}^{W}X$ corresponds to the world frame.

The mobile manipulator Jacobian relates the joint velocities (quasi-velocities) $\boldsymbol{\zeta} = [\boldsymbol{\nu}^{\boldsymbol{T}}\ \dot{\boldsymbol{q}}^{\boldsymbol{T}}]^{\mathsf{T}}$ to the end-effector velocities $\dot{\eta}_E$

$$\dot{\eta}_E = J(q)\zeta \tag{8}$$

For the arm (i.e., with $\boldsymbol{\zeta} = \dot{\boldsymbol{q}}$), the Jacobian is obtained by taking derivates of eq. (5), (6), and (7) with respect to the configuration vector in eq. (3).

$$\mathbf{J}_i(q) = \begin{bmatrix} \frac{\partial E_x}{\partial q_1} & \frac{\partial E_x}{\partial q_2} & \frac{\partial E_x}{\partial q_3} & \frac{\partial E_x}{\partial q_4} \\ \frac{\partial E_y}{\partial q_1} & \frac{\partial E_y}{\partial q_2} & \frac{\partial E_y}{\partial q_3} & \frac{\partial E_y}{\partial q_4} \\ \frac{\partial E_z}{\partial q_1} & \frac{\partial E_z}{\partial q_2} & \frac{\partial E_z}{\partial q_3} & \frac{\partial E_z}{\partial q_4} \\ 0 & 0 & -1 & 0 \\ 0 & -1 & 0 & 0 \\ 1 & 0 & 0 & 1 \end{bmatrix} \tag{9}$$

3.1.2 Full System Derivation

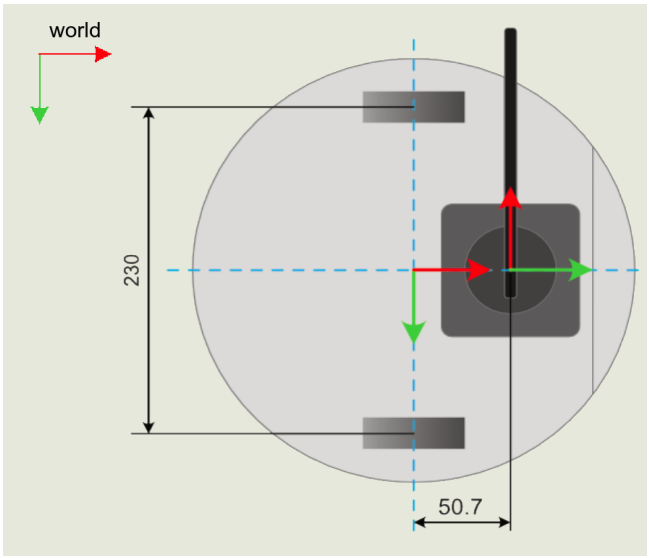


Fig. 3: Relation between Frames

This derivation is seen as an extension of the results in the previous case. Here, the world frame is set and all calculations are done relative to this frame. Figure 3 gives a symbolic representation of this. The cartesian position of the end-effector in this world frame is given by the following equations.

$${}^{W}E_x = {}^{A}E^{*}sin(q_1 + \theta) + 50.7cos(\theta) + x \tag{10}$$

$$^{W}E_y = -^{A}E^{*}cos(q_1+\theta) + 50.7sin(\theta) + y \quad (11)$$

$$^{W}E_z = 72.2 - 108 - 142cos(q_2) - 158.8sin(q_3) - 198 \quad (12)$$

Taking derivatives with respect to the configuration vector as previously done (eq. (9)), the Jacobian for the arm in the world frame can be obtained. The constants are in millimeters, so they should be converted to meters before the calculation.

To integrate the mobile base (i.e., the Kobuki Turtlebot 2) with the arm, the base could be considered as a part of the manipulator. This way, two extra degrees of freedom which correspond to the linear and angular velocities of the entire system can be generated. To do this, the base is modeled as a revolute and a prismatic joint using the Denavit-Hartenberg (DH) algorithm. The resulting DH parameters are as follows:

DOF	θ	d	a	α
1	$-\pi/2$	-0.198	0.0	$-\pi/2$
2	0.0	0.0507	0.0	$\pi/2$

TABLE 2: Denavit-Hartenberg Table

Note: The units in the above table are in millimeters. 0.0507 is the offset of the base of the arm from the center of the Kobuki base and -0.198 is the height of the Kobuki from the ground in the NED-frame.

The final transformation from the world to the end-effector frame is given by:

$$\begin{aligned} ^{W}T_E &= ^{W}T_R\ ^{R}T_A\ ^{A}T_E \\ &= \begin{bmatrix} c\theta & -s\theta & 0 & x_R \\ s\theta & c\theta & 0 & y_R \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} c(\frac{-\pi}{2}) & -s(\frac{-\pi}{2}) & 0 & 0.0507 \\ s(\frac{-\pi}{2}) & c(\frac{-\pi}{2}) & 0 & 0 \\ 0 & 0 & 1 & -0.198 \\ 0 & 0 & 0 & 1 \end{bmatrix} \\ &\quad \begin{bmatrix} 1 & 0 & 0 & ^{A}E_x \\ 0 & 1 & 0 & ^{A}E_y \\ 0 & 0 & 1 & ^{A}E_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \end{aligned} \quad (13)$$

where $c\theta$ and $s\theta$ correspond to the cosine and sine of θ respectively.

The part of the system Jacobian corresponding to the base (i.e., for the first two DOFs in the overall quasi-velocities ζ) is obtained algorithmically.

Algorithm 1 Recursive Jacobian Computation

Require: DH parameters $d, \theta, a, \alpha, \rho \in \Re^n$
Ensure: Jacobian matrix $J(q) = [J_1, J_2]^T \in \Re^{6xn}$
1: Compute: $T_i{}^{i-1} = DH(d_i, \theta_i, a_i, \alpha_i),\ i = 1...n$
2: Compute: $T_i = T_i{}^{0} = \prod_{k=1}^{i} T_k{}^{k-1},\ i = 1...n$
3: Initialize: $z_0 = [0\ 0\ 1]^T, O_0 = [0\ 0\ 0]^T$
4: **for** $i \in 1..n$ **do**
5: $J_i = \begin{bmatrix} \rho_i z_{i-1} X(O_n - O_{i-1}) + (1-\rho_i) z_{i-1} \\ \rho_i z_{i-1} \end{bmatrix}$
6: **end for**
7: **return** J

where ρ_i is 0 for prismatic joints and 1 for revolute joints while z_i and O_i and z-axis and origin of the respective frames. The result of the above formulation is concatenated with the world-referenced Jacobian formulation for the manipulator.

3.2 Task Priority Control

A task is a column vector that is defined using the coordinates introduced in eq. (1). It is represented symbolically by σ and is usually sub-scripted with a letter defining the type of task. Some examples of tasks explored include; end-effector position, end-effector orientation, end-effector configuration, base orientation, joint position, joint limit, and obstacle avoidance. The TP algorithm enables a robot to execute multiple tasks with different priorities using its null space. Higher-priority tasks are executed first and the errors that arise from their execution are corrected before the execution of lower-priority tasks.

3.2.1 Equality Tasks

Of the 7 tasks listed above, the first 5 are considered equality tasks. This is because they are used for regulation or tracking purposes; therefore, whenever they are defined, they are always kept active. These tasks are properly defined with their variables (σ_i), errors ($\tilde{\sigma}_i$), and corresponding Jacobians ($J(q)$) in table 3.

3.2.2 Set-based Tasks

Tasks within this category are mostly implemented for safety reasons, and as such, it is required that they are placed at the top of the task hierarchy [5]. Two set-based (inequality) tasks are implemented in this project - joint limit and obstacle avoidance. In both cases, the task is only triggered when the task variable falls within a certain activation threshold. The parameters of the joint limit task are outlined below.

$$\sigma_{li} = \sigma_{li}(q) = q_i \quad (14)$$

$$Safe\ set,\ \delta_{li} = [q_{i,min}, q_{i,max}] \quad (15)$$

$$Error,\ \tilde{\sigma_i} = 1 \quad (16)$$

$$J_i = [0, 0, ..., 1, ..., 0] \in \Re^{1xn} \quad (17)$$

$$a_{li}(q) = \begin{Bmatrix} -1,\ a_{li} = 0 \wedge q_i \geq q_{i,max} - \alpha_{li} \\ 1,\ a_{li} = 0 \wedge q_i \leq q_{i,min} + \alpha_{li} \\ 0,\ a_{li} = -1 \wedge q_i \leq q_{i,max} - \delta_{li} \\ 0,\ a_{li} = 1 \wedge q_i \geq q_{i,min} - \delta_{li} \end{Bmatrix} \quad (18)$$

In the above equations, δ_{li} and α_{li} are the deactivation and activation thresholds, respectively.

In the case of obstacle avoidance, this is important to keep a specific joint away from a region of interest. For instance, during a manipulation task such as pick-and-place, keeping the base of the Kobuki Turtlebot away from the object of interest is key. The following equations define the obstacle avoidance task.

$$\sigma_r = \sigma_r(q) = |\eta_{1,ee}(q) - P| \quad (19)$$

$$Safe\ set,\ \delta_r = \Re^3\{\eta_1 : \eta_1 - P| \leq r\} \quad (20)$$

Task	Task Definition
Position	$\sigma_p = \sigma_p(\mathbf{q}) \in \mathbb{R}^{3\times 1}$ $\sigma_p = \eta_1 = \begin{bmatrix} x \\ y \\ z \end{bmatrix}$ $\quad \tilde{\sigma}_p = \eta_{1,d} - \eta_1$ Linear velocity part of the end-effector Jacobian (3 top rows) $J_p = J_p(\mathbf{q}) = J_v(\mathbf{q}) \in \mathbb{R}^{3\times n}$
Orientation	$\sigma_o = \sigma_o(\mathbf{q}) \in \mathbb{R}^{3\times 1}$ $\tilde{\sigma}_o = w\epsilon_d - w_d\epsilon - \epsilon \times \epsilon_d$ Angular velocity part of the end-effector Jacobian (3 bottom rows) $J_o = J_o(\mathbf{q}) = J_\omega(\mathbf{q}) \in \mathbb{R}^{3\times n}$
Configuration	$\sigma_c = \sigma_c(\mathbf{q}) \in \mathbb{R}^{6\times 1}$ $\tilde{\sigma}_c = \begin{bmatrix} \eta_{1,d} - \eta_1 \\ w\epsilon_d - w_d\epsilon - \epsilon \times \epsilon_d \end{bmatrix}$ Full end-effector Jacobian $J_c = J_c(\mathbf{q}) = J(\mathbf{q}) \in \mathbb{R}^{6\times n}$
Joint Position	$\sigma_{ji} = \sigma_{ji}(\mathbf{q}) \in \mathbb{R}$ $\tilde{\sigma}_{ji} = \sigma_{ji,d} - \sigma_{ji}$ Single-entry row matrix 0 1 ... i ... n $J_{ji} = [0,0,\ldots,1,\ldots,0] \in \mathbb{R}^{1\times n}$

TABLE 3: Equality Tasks - [1]

$$Error,\ \tilde{\sigma}_i = \frac{\eta_{1,ee}(q) - P}{|\eta_{1,ee}(q) - P|} \tag{21}$$

$$J_r = J_r(q) = J_v(q) \in \Re^{exn} \tag{22}$$

$$a_{li}(q) = \begin{Bmatrix} 1,\ a_r = 0 \wedge |\eta_{1,ee}(q) - P| \leq r_\alpha \\ 0,\ a_r = 1 \wedge |\eta_{1,ee}(q) - P| \geq r_\delta \end{Bmatrix} \tag{23}$$

In the above equations, r_δ and r_α are the deactivation and activation thresholds, respectively. Additionally, in set-based tasks, it is imperative that the deactivation threshold is greater than the activation threshold to avoid chatter.

Having defined the tasks, it is necessary to have a way of integrating them in an algorithmic manner. For this, the Recursive Task Priority Redundancy Resolution algorithm as seen in [2] and [1] was implemented. This is summarized in the following pseudocode.

Algorithm 2 Recursive Task Priority Redundancy Resolution

Require: A list of tasks
Ensure: Quasi-velocities $\zeta_k \in \Re^n$
Initialise: $\zeta_0 = 0^n, P_0 = I^{nxn}$
for $i \in 1..k$ **do**
  if $a_i(q) \neq 0$ **then**
    $\bar{J}_i(q) \leftarrow J_i(q)P_{i-1}$
    $\zeta_i \leftarrow \zeta_{i-1} + \bar{J}_i^\dagger(q)(a_i(q)\dot{x}_i(q) - J_i(q)\zeta_{i-1})$
    $P_i \leftarrow P_{i-1} + \bar{J}_i^\dagger(q)\bar{J}_i(q)$
  else
    $\zeta_i \leftarrow \zeta_{i-1}, P_i \leftarrow P_{i-1}$
  end if
end for
return ζ_k

In the above algorithm, P is the null space projector which is used to find the set of joint velocities that does not produce any velocity in the current task space. Therefore, it allows a robot to perform secondary tasks that have no influence whatsoever on primary tasks. The base formula for P is:

$$P = (I - J_i{}^\dagger(q)J(q) \tag{24}$$

When the secondary task cannot be fulfilled without preventing the primary task from being realized, the matrix $\bar{J}_i^\dagger(q)$ becomes singular, thus generating the same problems occurring at kinematic singularities, i.e., high joint velocities and oscillations (algorithmic singularity). During such singularities, priority could be inverted; therefore, the TUNED damped-least squares (DLS) method was used for the Jacobian inversion required to compute $\bar{J}_i^\dagger(q)$. [2]. On the other hand, the DLS method does not preserve the null space of the Jacobian matrix. This implies that when it is used, some joint velocities that are supposed to be in the null space may be projected out of it, leading to unwanted motion in the null space. This directly leads to task violation in the task priority algorithm.

4 Implementation

Following the requisite system design and algorithmic developments, the control system was implemented on the mobile manipulator in simulation and the necessary validation was performed in real experiments with the mobile manipulator. In the following subsections, all developments are made using the Robot Operating System (ROS) interface as the middleware framework. This is done for ease of implementation in simulation, and, more importantly, on the hardware which is also based on ROS.

4.1 Overview

The final goal of this project is the Implementation of a pick, transport, and place operation, using a sequencer, for a prior known approximate location of pick and place locations, utilizing visual feedback from ARUCO marker detection and dead reckoning for navigation. To achieve this, subgoals were constructed to incrementally build complexity and for intermediate results before the final goal. In the remaining subsections, the different parts that constitute holistic development will be discussed.

4.2 Implementation Block

Several ROS nodes, python scripts, services, messages, etc., were used for implementing the kinematic control system.

These are documented in several UML and doxygen diagrams in the ROS package for this project and in the appendix of this report (see figures 27 and 28).

4.3 Choice of a Task Planning Strategy

A behavior tree, implemented as a major ROS node defines the full system integration. Behavior trees are hierarchical tree structures, where each node signifies a stand-alone action, condition, or decision point. The tree is built from its root and uses 'branches' to connect intermediate nodes called control flow nodes and leaf nodes. The whole structure defines a sequence of actions, conditions, and tasks to be taken, considered, and executed. This sequencer was chosen because of its simplicity, scalability, traceability, and high level of intuitiveness. The py_trees library was used to generate and manage the behavior tree and implemented behaviors. [3].

4.4 TASK D*: Motion between Desired Cartesian End-Effector Positions

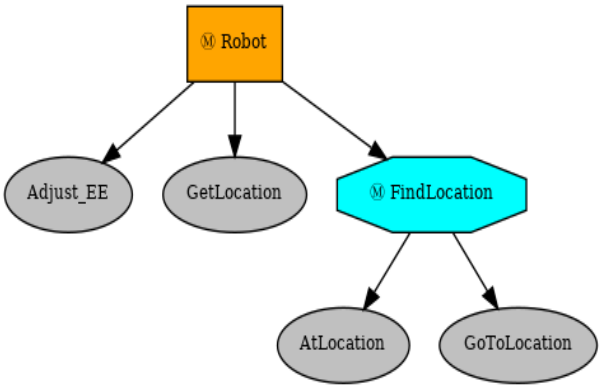
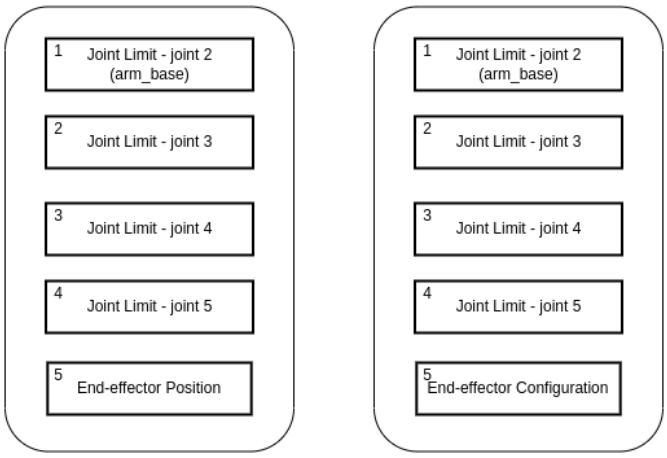


Fig. 4: Behavior Tree for Tasks D* and D



(a) Task D* (b) Task D

Fig. 5: (a), (b) Some Task Hierarchies. The joint limits, like other set-based tasks, are usually given precedence for safety reasons.

This implementation combines joint limits tasks of each DOF of the robotic arm and an end-effector position task as the lowest priority task. Joint limits tasks are given precedence for safety reasons and to prevent the arm from getting 'stuck'. The FIGURE BELOW depicts the behavior tree with the implementation decorators and leaf nodes.

The generic behaviors implemented are summarized below.

- **Adjust_EE** - This uses the arm-only task priority node to move the end-effector to what's considered as the 'home' position, i.e., approximately aligned with the x-axis of the base and pointing outwards.
- **GetLocation** - This behavior primarily reads the predefined desired EE-position from a Python dictionary and writes each location to the blackboard before popping that location from the aforementioned dictionary. The reading/writing process is done at a rate of 1 location per full tick of the behavior. This behavior returns 'FAILURE' if the dictionary is empty and 'SUCCESS' if 'location' was successfully set.
- **AtLocation** - As the name implies, this behavior checks if the end-effector is at the desired location. It does this by subscribing to the odometry topic of the full robot and its joint_states, extracting the robot's current pose, using the forward kinematics to find the EE's current pose, then comparing it with the location it reads from the blackboard. This comparison is done within an error threshold. The behavior returns 'SUCCESS' if the end-effector is within the threshold; otherwise, it returns 'FAILURE'.
- **GoToLocation** - This behavior is designed in an agnostic manner such that it triggers the execution of the task priority of the full mobile manipulator by sending a location it reads from the blackboard via a custom ROS service and uses a ROS subscriber to keep track of the TP error to know when the desired location has been reached. The ROS subscriber and service are initialized in the setup function of the behavior.
- **FindLocation** - This is a selector decorator (not a behavior) that is executed until the first child returns 'SUCCESS'. It's a different implementation of the class 'if' statement in programming.

4.5 Task D: Motion between Desired Cartesian End-Effector Configurations

This uses the same behavior tree as the previous case, but an end-effector configuration task replaces the EE-position task in the task hierarchy (see fig. 5b).

4.6 Task E: Pick an Object

This task is done using the arm-only task priority node as it guarantees manipulation while having the assurance that the Kobuki base will not move throughout its execution. For this, the desired locations ought to be known in the frame of the base of the arm.

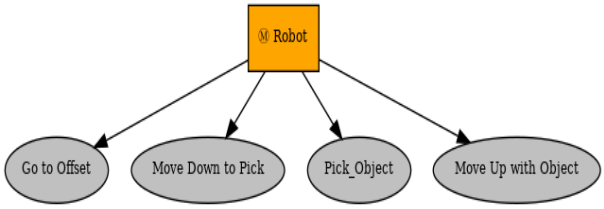


Fig. 6: Behavior Tree for Task E

The following are the behaviors in the leaf nodes of the above tree.

- **Go To Offset** - This moves the end-effector to a clearance position vertically above the object to be picked.
- **Move Down to Pick** - As the name implies, this moves the EE to the top-center of the object.
- **Pick_Object** - This uses the boolean ROS service for the vacuum gripper to turn on the vacuum.
- **Move Up with Object** - This simply does the reverse of the 'Move Down to Pick' behavior, except that it does so while lifting the object.

The behaviors, 'Go To Offset', 'Move Down to Pick', and 'Move Up with Object' are not unique behaviors; rather, they are all based on the generic behavior, 'GoToLocation'. Seemingly minor approaches like this make the behavior tree a lot more scalable.

4.7 Task F: Place an Object on the Robot's Platform

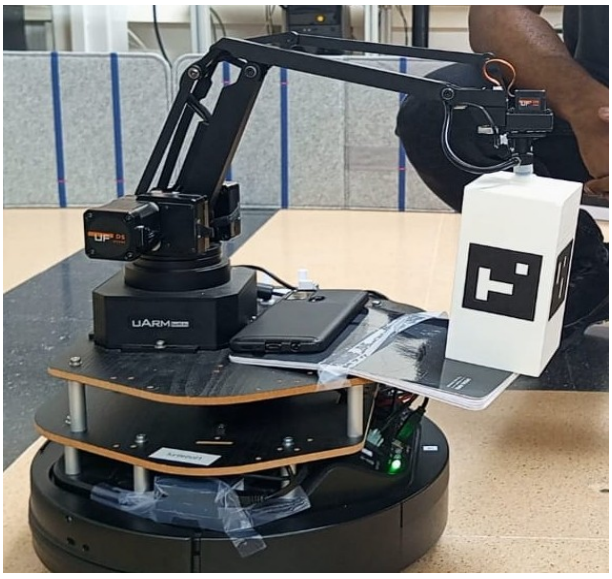


Fig. 7: Place Object on Robot's Platform

Similar to task E, this task is done using the arm-only task priority node. The sequence of implementing this task is:

- Move the end-effector to an offset vertically above the object to be picked.
- Move the EE down to the top-center of the object.
- Turn on the vacuum and latch onto the object with the suction cup.
- Move the EE with the object to an offset vertically above the picking position.
- Turn the EE to the platform while holding the object.
- Move the EE down for the object to come in contact with the platform
- Turn off the vacuum to place the object on the platform.
- Move the EE to an offset vertically above the placing position.

This employs generic behaviors from previous sub-sections such as 'Adjust_EE' and 'Pick_Object'. See fig. 9a for the behavior tree for this task.

4.8 Task G: Pick-Transport-Place with Dead-Reckoning

For this, an extra ROS node - the dead-reckoning (DR) node - is required. In the simulation, a ground truth odometry sensor is placed on the robot and the topic is subscribed to and used for DR. In the real implementation, the joint states of the robot are subscribed to. From this topic, the left and right wheel angular velocities are obtained. These along with the radius of the wheels and the distance between the two wheels are used to compute the linear and angular velocities of the full system. Then, the results are used to update the state of the robot - $[x\ y\ \theta]^T$. This translates to a simple kinematic model for a differential drive robot.

$$\begin{aligned} x_k &= x_{k-1} + V\cos\theta_{k-1}\Delta t \\ y_k &= y_{k-1} + V\sin\theta_{k-1}\Delta t \\ \theta_k &= \theta_{k-1} + W\Delta t \end{aligned} \tag{25}$$

where V and W are linear and angular velocities respectively.

The sequence of implementing this task is:

- Move the EE to the home position.
- Read the location (known apriori) of the object to be picked from the blackboard.
- Go to the EE location in the vicinity of the object using the full robot's TP node.
- Overwrite the blackboard location with the precise picking location.
- Find the location and move there with the EE
- Turn on the vacuum and latch onto the object with the suction cup.
- Set the offset location vertically above the picking position.
- Activate solution weighting using a service to the TP node.
- Lift the object to the offset
- Deactivate solution weighting using a service to the TP node.
- Set the placing location on the blackboard
- Move the full robot to the placing vicinity
- Move the EE to the placing offset
- Move down to place the object.
- Turn off the vacuum and release the object.
- Move the EE to an offset vertically above the placing position.
- Use the full robot's TP to move the robot back to the start position.

This employs generic behaviors from previous sub-sections such as 'Adjust_EE', 'Pick_Object', 'Pick_Object', 'GoToLocation', 'SendFlag', 'AtLocation', and 'SetPickLocation'.

The additional generic behaviors used here are described below:

- **Place_Object** - This uses the boolean ROS service for the vacuum gripper to turn off the vacuum.
- **SendFlag** - This also uses a boolean ROS service to communicate with the TP node and switch a flag that controls whether or not solution weighting is applied.
- **SetBlackboardVariable** - This is a default py_trees behavior for overwriting blackboard variables.

In this implementation, it was discovered that the robot's base tends to hit the object to be picked during the picking/placing process using the TP algorithm for the whole system. To mitigate this, solution weighting was applied. This will be discussed in a subsequent sub-section on practical modifications.

See fig. 9b for the behavior tree for this task.

4.9 Pick-Transport-Place with Dead-Reckoning and ArUco Marker Detection

In the last task, the locations of the objects to be picked were known as apriori. In this task, however, the realsense camera attached to the Kobuki base will be used to detect ArUco markers that are attached to the objects. This is done based on a ROS ArUco detection node that subscribes to the relevant camera topics and robot's odometry topic, estimates the pose of the marker using OpenCV functions, transforms the poses to the world, and publishes them on a dedicated topic. Two extra generic behaviors were written to account for the ArUco detection aspect.

- **Detect_ArUco** - This behavior subscribes to the ArUco marker detection node and writes the pose to the blackboard. It also flips a detection flag variable on the blackboard to 'True' before returning 'SUCCESS'.
- **ArUco_Offset** - Since the detected point is the center of the ArUco marker, this behavior provides the necessary offset for the end-effector. It overwrites the blackboard variable set by the 'Detect_ArUco' behavior.

Additionally, as an improvement to the implementation of task G, obstacle avoidance was used in place of solution weighting for critical aspects like pick and place. The resulting task hierarchy is found in fig. 8.

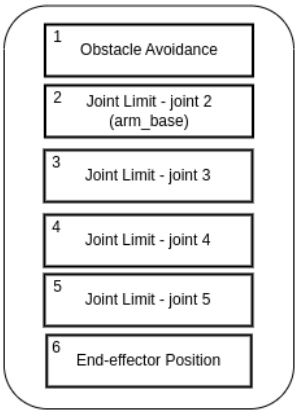


Fig. 8: Task Hierarchy for Task H

See fig. 9c for the behavior tree for this task.

4.10 Practical Modifications

4.10.1 Velocity Scaling

The task priority algorithm inherently has no information about the velocity limits of the robot under consideration; therefore, it is important to include this for safety reasons. In this project, the maximum velocities for all degrees of freedom were set to 1m/s. So, the velocity algorithm implemented is as follows:

Algorithm 3 Velocity Scaling

Require: Maximum velocities: $\zeta_{i,max}\ for\ i = 1...n(DOF)$
Ensure: Scaled velocities
1: s = $\max_{i \in 1...n}(\zeta_i/\zeta_{i,max})$
2: **if** $s > 1$ **then**
3: **return** ζ/s
4: **else**
5: **return** ζ
6: **end if**
7: **return** ζ_k

4.10.2 Solution Weighting

In some cases, being able to channel velocities only to the arm for tasks like picking and placing is a topic of interest. This is where **solution weighting** comes in. This allows the TP algorithm to emphasize the use of certain degrees of freedom over others. For example:

- If the EE desired position is far from the base, it would be better to focus on moving more of the base and less of the arm until the robot is closer to the goal, before channeling more velocities to the arm.
- On the other hand, during manipulation tasks, it would be beneficial to keep the base static and move only the joints of the arm. To implement solution weighting, a diagonal matrix of weights is defined such that higher weights mean that the corresponding DOFs will be less preferred. With this, *the TP algorithm could be modified by replacing instances of the Jacobian transpose with* $W^{-1}.J_i{}^T(q)$ *in both the pseudoinverse and DLS methods of Jacobian inversion.*

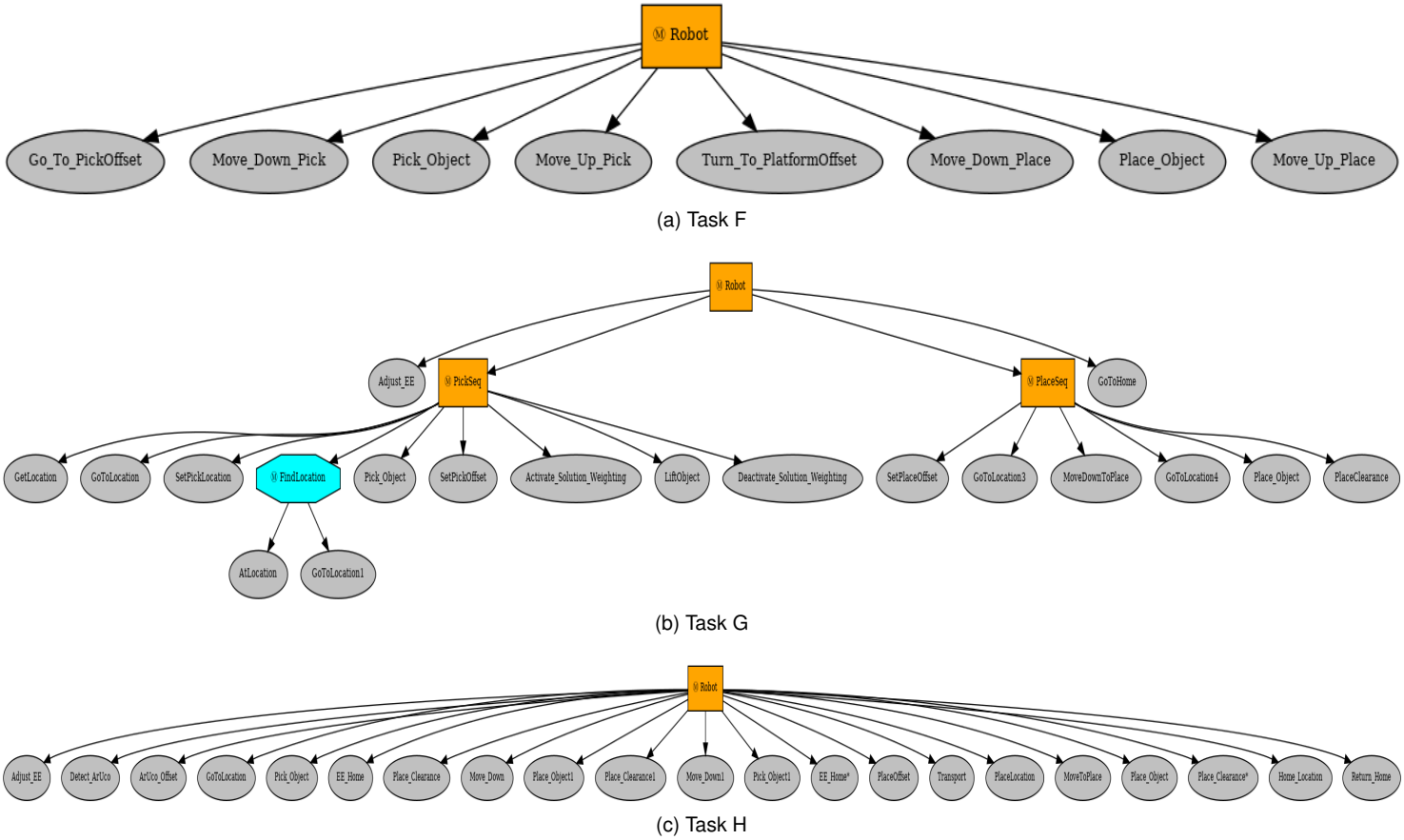


Fig. 9: (a), (b), (c) Behavior Trees for Different Tasks. The tree increases in size as the complexity of the task increases.

5 RESULTS

In the foregoing subsections, three major plots are used to show the results of the implementations of the tasks. A summary of the format and content of the plots is as follows:

1) **Evolution of control errors over time** - In this plot, the positions of each joint of the arm are plotted. The joint limits for each of these are also plotted using dashed lines. Finally, the norm of the error of the end-effector's position (x,y,z) is plotted. For task H, the distance of the end-effector from the obstacle of interest is also plotted since obstacle avoidance was implemented for this task.
2) **Evolution of joint velocities over time** - this simply plots the joint velocities generated by the TP algorithm against time. For tasks like pick and place without moving where only the task priority of the arm was used, this plot contains only 4 parameters. In other cases, it contains 6 parameters that correspond to the degrees of freedom of the base and the arm.
3) **Motion of the mobile base and the end-effector in the X-Y plane** - This plot is generated by extracting the EE's pose from the forward kinematics and the pose of the Kobuki base from the relevant variables (that interface with the joint states) and plotting them on the X-Y plane.

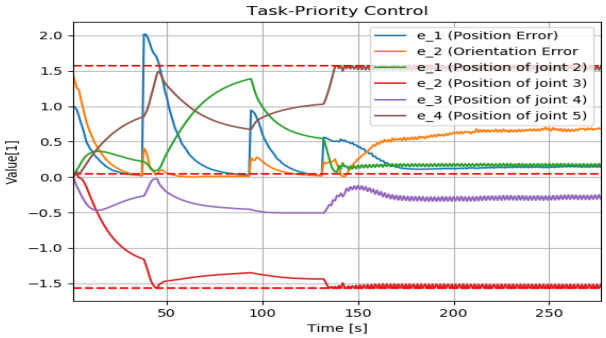
5.1 Simulation

5.1.1 Motion between Desired Cartesian End-effector Configurations

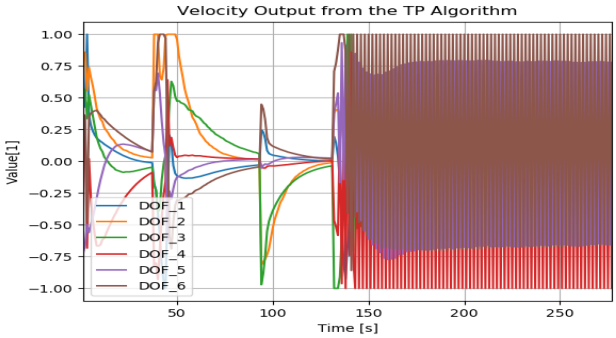
This task was implemented on the robot a number of times and the video of this is found in the slides. However, it proved to be rather challenging because of the nature of the configuration task. That is, the errors in position and orientation are simultaneously being reduced. This causes the robot to 'struggle' in some set points that are seemingly easy to reach. A similar case was also noticed in simulation and this is evident in the results presented in figure 10. End effector configurations (x, y, z, θ) of $[2.05, 0.15, -0.18, 0]$, $[1.05, -0.15, -0.25, \pi/24]$, $[3.0, 0.3, -0.2, \pi/12]$, $[2.3, -0.35, -0.137, \pi/16]$ were defined and the task hierarchy in 5 was used with the tree in 4. The first three configurations were achieved without triggering the joint limits as seen in the first 150s in figure 10a; however, in the final configuration task, the position error go down to almost zero, but the orientation error does not. The reason is because of the constant triggering of the joint limits of joints 3 and 5 (i.e., q2 and q4 of the arm). The velocity outputs also tell a similar story (figure 10b. The trajectories of the base and the end-effector are found in figure 10c.

5.1.2 Motion between Desired Cartesian End-effector Positions

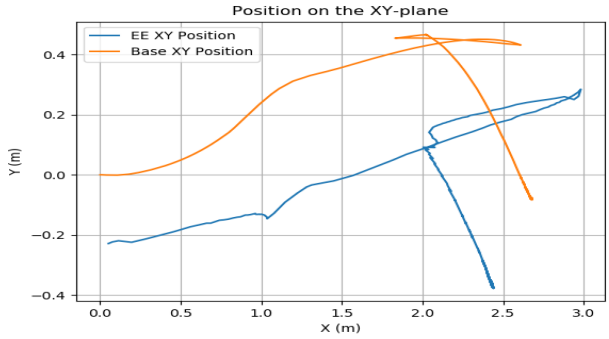
In this task, 4 desired end-effector positions were set in the field of the blackboard of the behavior tree. Plots 11a and



(a) Evolution of Control Errors Over Time

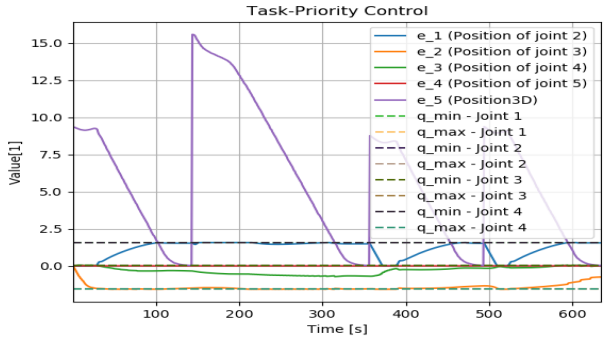


(b) Evolution of Joint Velocities Over Time



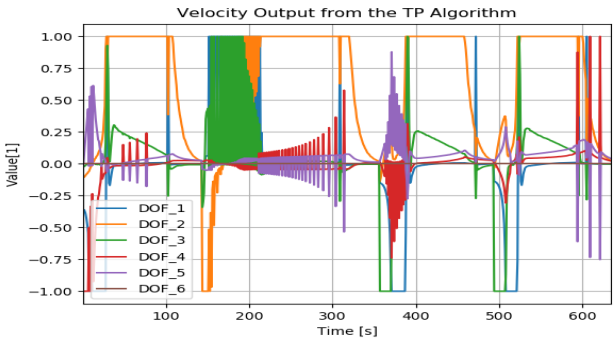
(c) Motion of the End-Effector in the X-Y plane

Fig. 10: (a), (b), (c) Plots for Task D in Simulation.

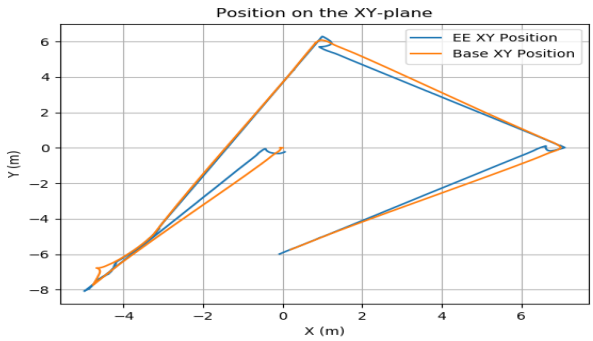


(a) Evolution of Control Errors Over Time

(b) Evolution of Control Errors Over Time

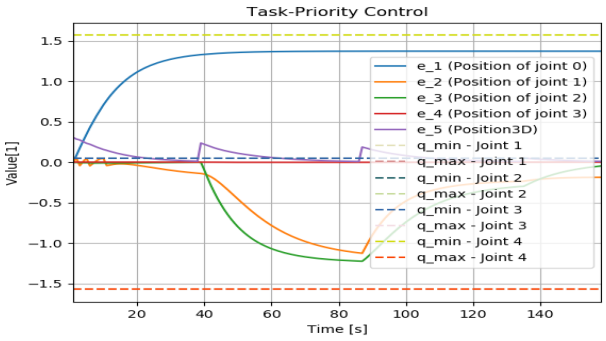


(c) Evolution of Joint Velocities Over Time

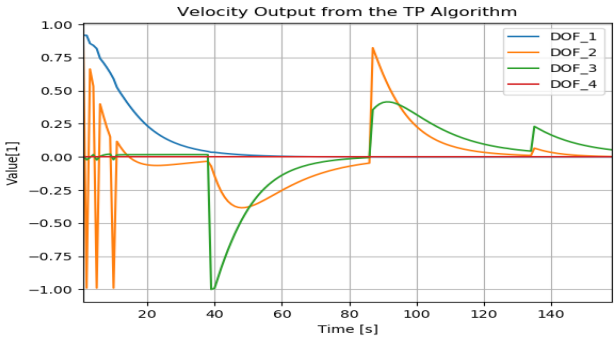


(d) Motion of the Mobile Base and the End-Effector in the X-Y plane

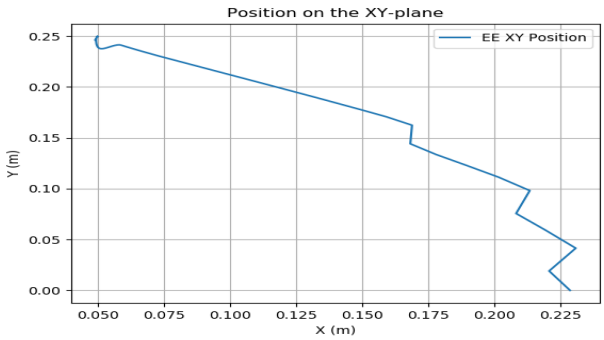
Fig. 11: (a), (b), (c) Plots for Task D* in Simulation.



(a) Evolution of Control Errors Over Time



(b) Evolution of Joint Velocities Over Time



(c) Motion of the End-Effector in the X-Y plane

Fig. 12: (a), (b), (c) Plots for Task E in Simulation.

11b show that the error (in purple) always converges to zero as each sub-task is achieved. The task priority behavior ends the sub-task when all the errors in the sub-task go below 0.01. In these same plots, it is observed, as expected, all joints stay within their limits. Joints 2 and 3 (which are the first and second joints of the arm from its base) are seen to test the limits more often than other joints while joint 4 does this towards the end of the run. It is worth noting that joint 5 (of the end-effector) remains at zero throughout the run because this is a position, not a configuration or orientation task. In plot 11c, it is quite apparent that solution weighting is not applied because there's preference given to any degree of freedom. The presence of velocity scaling is obvious, however, as no DOF is allowed to exceed 1m/s. In plot 11d), we observe a mostly linear trajectory of the base and end-effector in the X-Y plane. This is an expected result because the damped-least squares method was used in the null-space projector and pseudo-inverse for other Jacobian inversions. Additionally, since the joint limits were not triggered in this task (see plots 11a), the trajectory is smooth.

5.1.3 *Pick an Object*

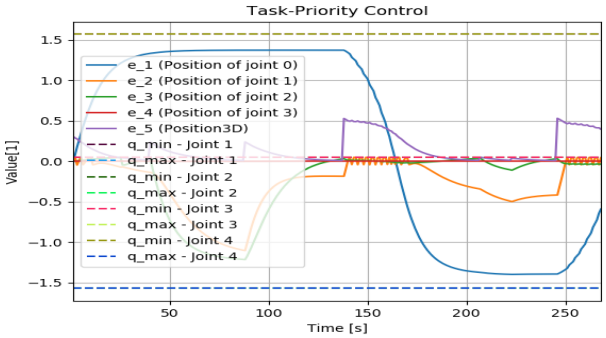
In plot 12a, the first joint of the arm starts from the zero position and increases rapidly before maintaining a relatively constant position. This is because the arm is moved from the zero-start and subsequent motions to pick the object are simply translational along the z-axis of the arm's base. As expected, the error in the EE's position always converges to zero when each sub-task is achieved. Following the above explanation, it is no surprise that in plot 12b, the velocity output of the arm's first joint starts at a max velocity of 1m/s (due to velocity scaling) and goes down to zero as that DOF is not required for the rest of the task. The third joint switches between positive and negative velocities in a pattern because the EE is lowered to pick the object and lifted after picking. In plot 12c, the trajectory of the EE has 3 jerks which exactly correspond to the joint limits that were triggered in DOF_2 in plot 12a (the orange line between 0 - 20 seconds).

5.1.4 *Place an Object on the Robot's Platform*

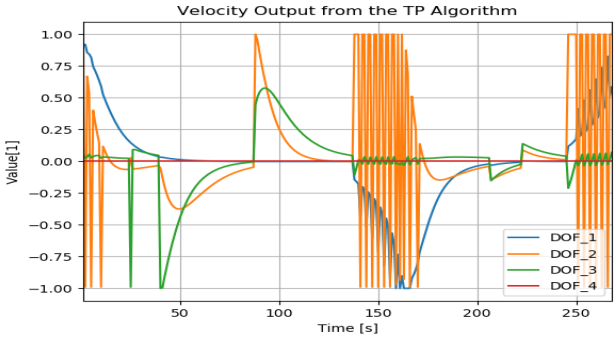
The first noticeable pattern in plot 13a is the position of joint 0 (the arm's first joint). Since this task involves picking an object from the front of the robot and placing it on the platform towards the back of the robot, joint 0 goes from positive to negative. This can also be deduced from the EE trajectory in plot 13c which is not smooth due to the joint limits that are triggered and directly affect the EE's position. In plot 13b, joint 1 (i.e., DOF_2) outputs velocities that seem to alternate between -1 and +1, especially around 150s. This is also related to the joint limits.

5.1.5 *Pick-Transport-Place with Dead-Reckoning*

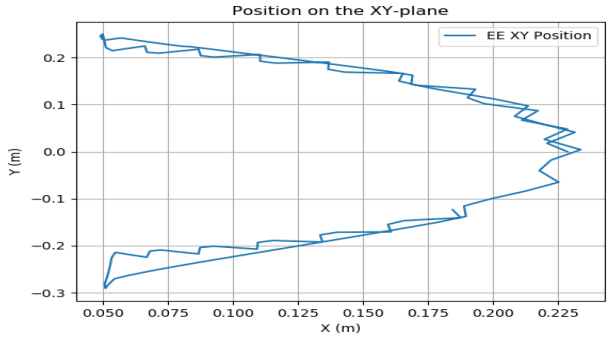
This task was dependent on an EE-position task, therefore, the EE-joint (DOF_6) is hardly used (see plots 14a and 14c).



(a) Evolution of Control Errors Over Time

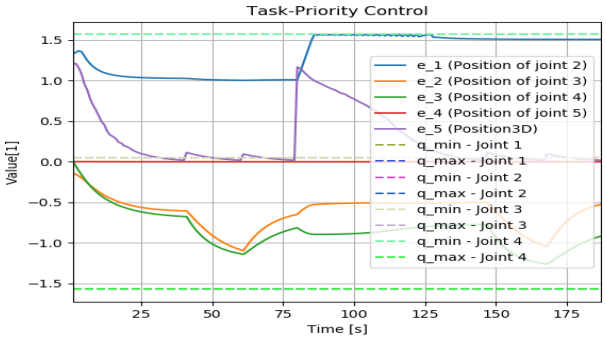


(b) Evolution of Joint Velocities Over Time

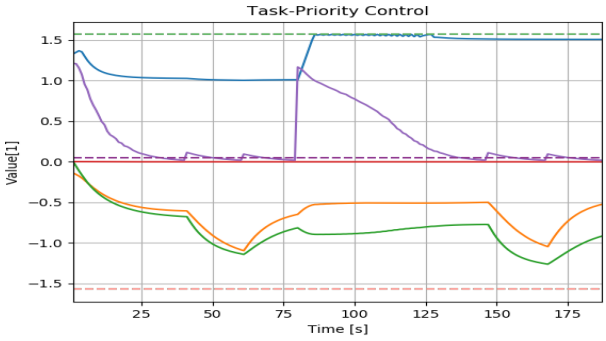


(c) Motion of the End-Effector in the X-Y plane

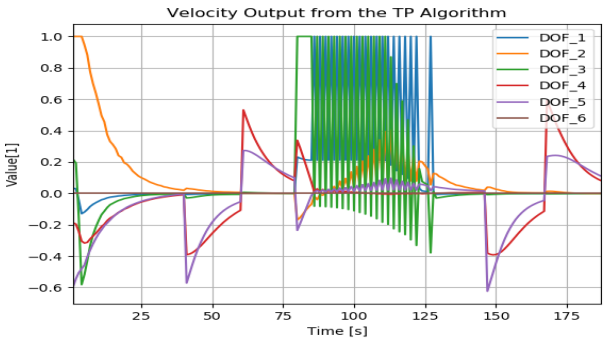
Fig. 13: (a), (b), (c) Plots for Task F in Simulation.



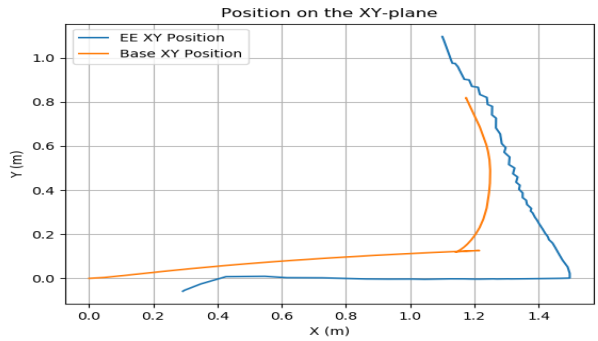
(a) Evolution of Control Errors Over Time



(b) Evolution of Control Errors Over Time

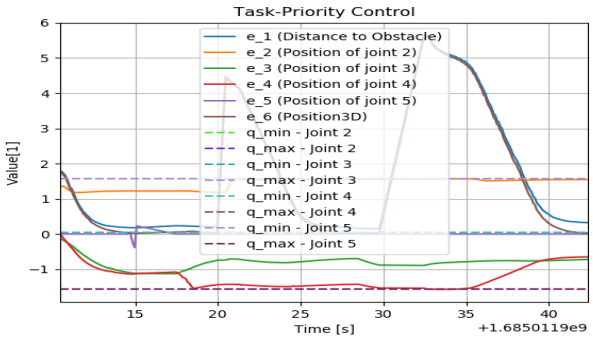


(c) Evolution of Joint Velocities Over Time

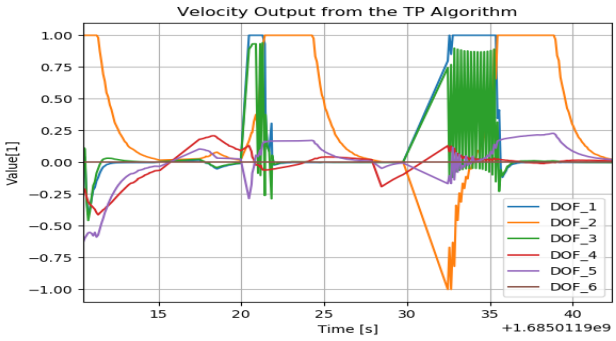


(d) Motion of the Mobile Base and End-Effector in the X-Y plane

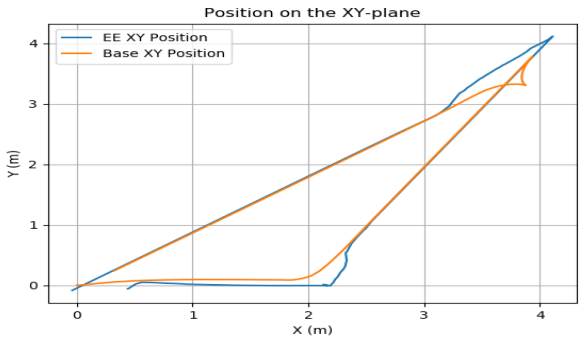
Fig. 14: (a), (b), (c) Plots for Task G in Simulation.



(a) Evolution of Control Errors Over Time



(b) Evolution of Joint Velocities Over Time



(c) Motion of the Mobile Base and End-Effector in the X-Y plane

Fig. 15: (a), (b), (c) Plots for Task H in Simulation.

5.1.6 Pick-Transport-Place with Dead-Reckoning and ArUco Detection

These X-Y trajectories in plot 15c show that the task was completed by returning the robot to the position from which it started. In the regions where the EE and the base follow almost the same trajectory, it is because it was aligned almost perfectly with the x-axis of the Kobuki base. In this task, obstacle avoidance was implemented and this is evident in plot 15a. The distance to the obstacle closely follows the error in the EE's position because both use the norm-2 distance. They noticeably differ when the EE manipulates the object and the base remains outside the activation threshold.

5.2 Experiments

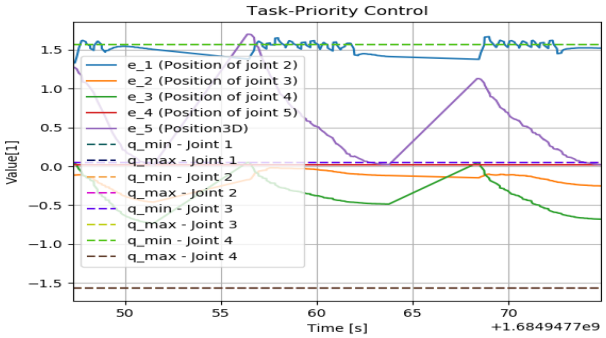
For the plots of experimental implementations, rospy time was used; therefore, some plots are not exactly as they should be. Using rospy time meant that there was a time difference while the behavior tree switched behaviors. So, such regions appear as an upward-sloping straight line. For example, in the evolution of control errors, the end-effector error appears to slowly rise, which is contrary to what is expected. The error should abruptly go high (not linearly) as the desired end-effector pose changes before converging to zero as the task is achieved. This issue was only discovered after concluding the experiments, so it was only corrected in the simulation results.

The plots from the experimental implementations are noticeably similar to those obtained in the simulation. This is a testament to the fact that the simulation environment and software capture most of what is obtainable in the real world. In the real world, however, we were unable to experiment with desired end-effector positions that were far from the start position of the robot to examine how the task priority would perform in such cases. We opine that the performance should be quite similar except that irregularities in the terrain amongst other external factors would play a major role in the results.

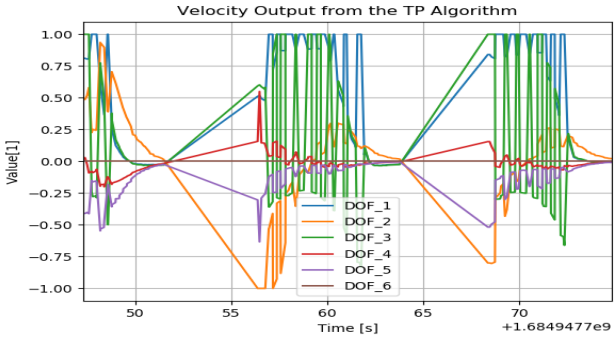
5.3 Integration Task

The integration task, or challenge is to explore an area while picking up certain objects and placing them in predefined locations. The kinematic control system developed in this project is primarily focused on the pick-and-place aspects of the integration task. Our approach to this task is as summarized below:

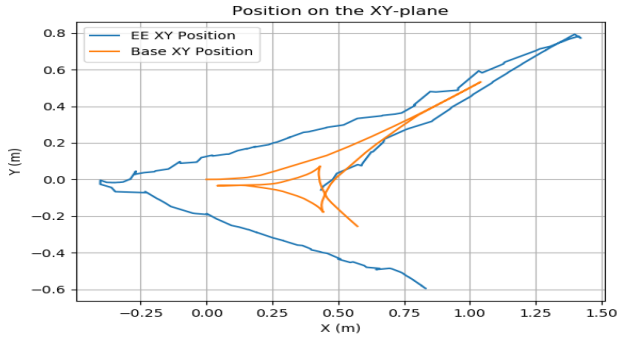
1) The task begins with an adjustment of the end-effector using the task priority node for the arm. The purpose of this is to move it to a safe position to prevent unwanted collisions with nearby objects during the exploration phase or occlusion of the camera during the aruco marker detection phase.
2) The exploration node is activated alongside the SLAM node and the robot starts exploring its environment.
3) During exploration, the aruco detection node is also kept running to detect any objects in the environment.



(a) Evolution of Control Errors Over Time

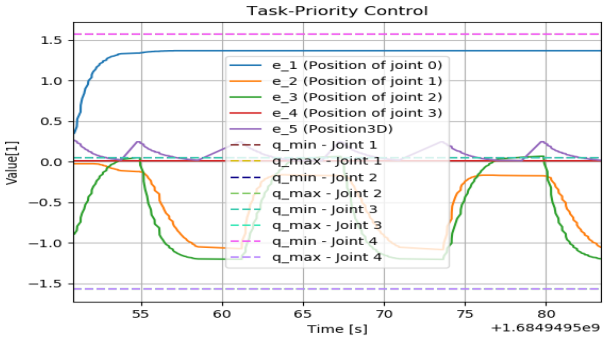


(b) Evolution of Joint Velocities Over Time

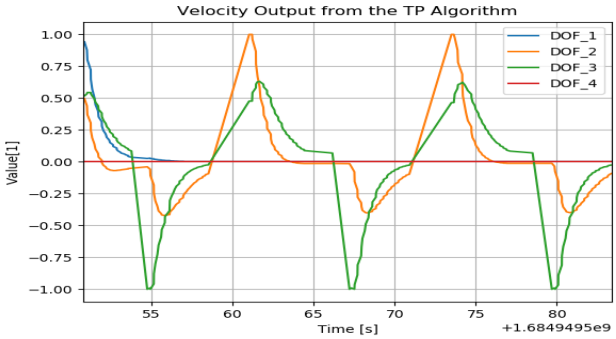


(c) Motion of the Mobile Base and the End-Effector in the X-Y plane

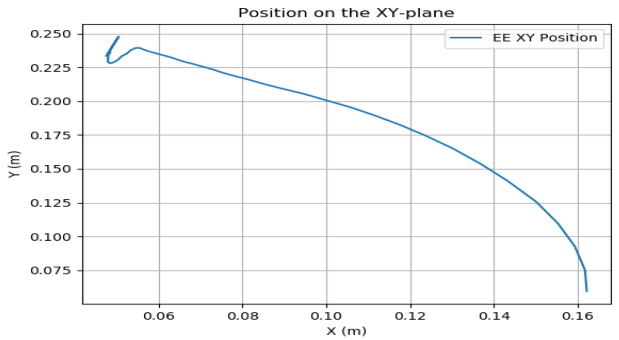
Fig. 16: (a), (b), (c) Plots for Task D* on the Real Robot.



(a) Evolution of Control Errors Over Time

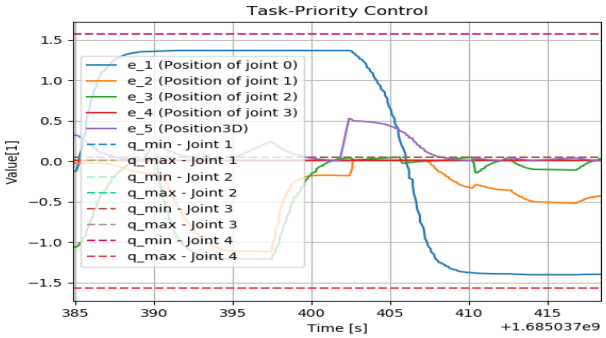


(b) Evolution of Joint Velocities Over Time

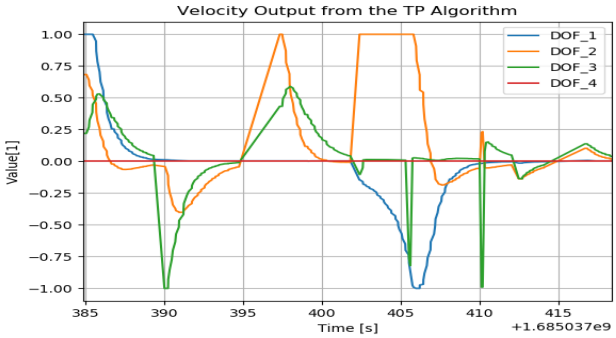


(c) Motion of the End-Effector in the X-Y plane

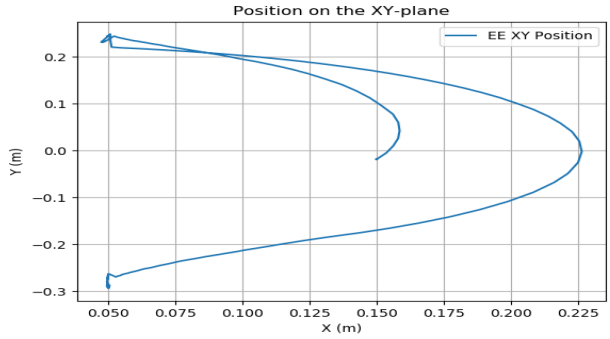
Fig. 17: (a), (b), (c) Plots for Task E on the Real Robot.



(a) Evolution of Control Errors Over Time

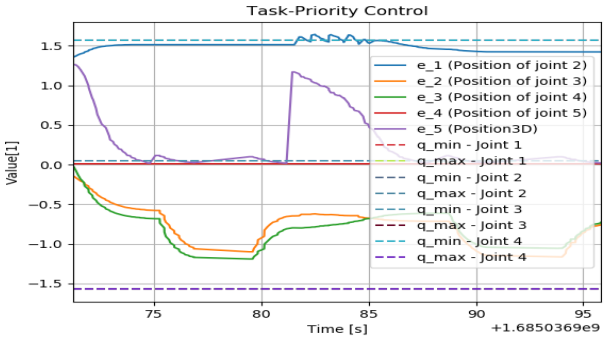


(b) Evolution of Joint Velocities Over Time

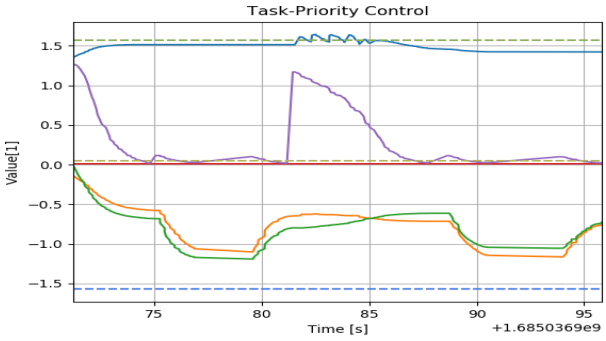


(c) Motion of the End-Effector in the X-Y plane

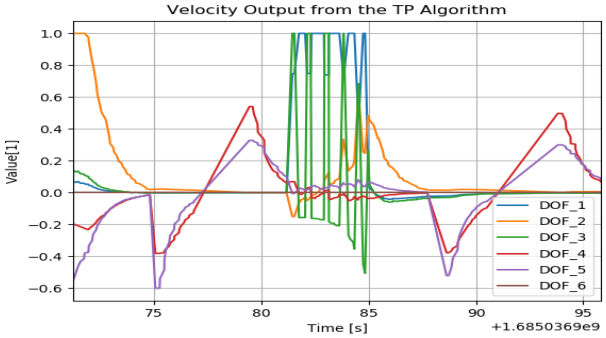
Fig. 18: (a), (b), (c) Plots for Task F on the Real Robot.



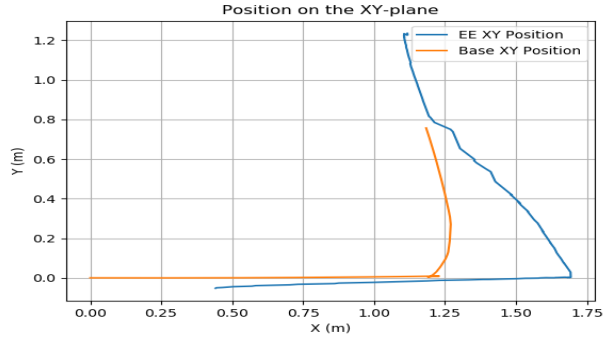
(a) Evolution of Control Errors Over Time



(b) Evolution of Control Errors Over Time

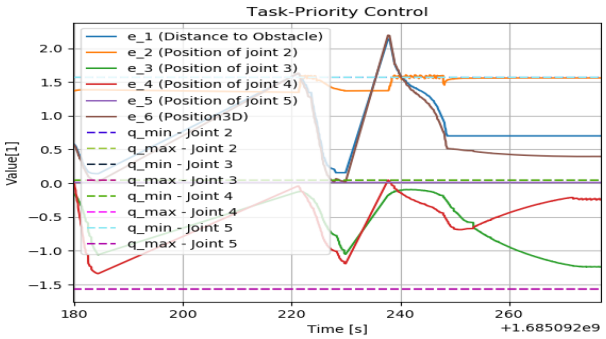


(c) Evolution of Joint Velocities Over Time

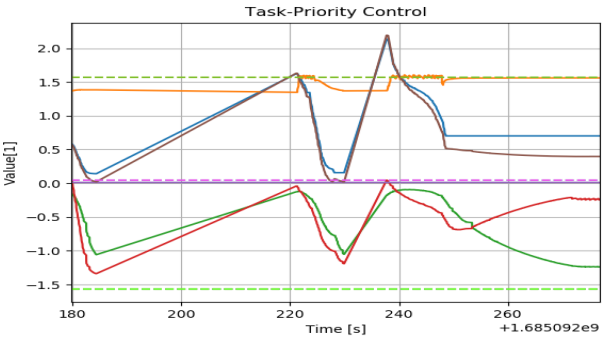


(d) Motion of the Mobile Base and End-Effector in the X-Y plane

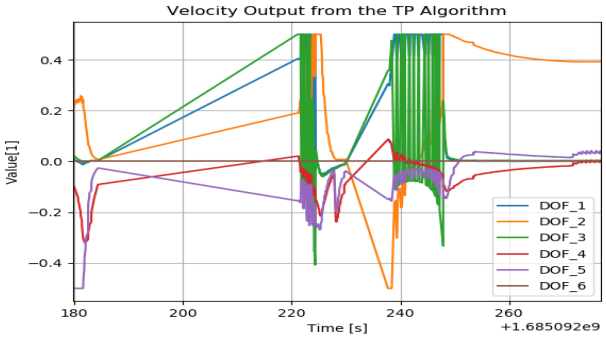
Fig. 19: (a), (b), (c) Plots for Task G on the Real Robot.



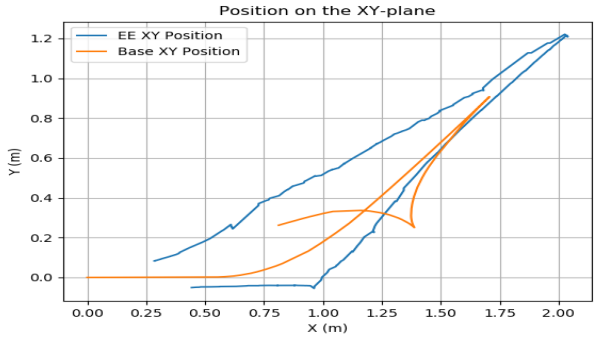
(a) Evolution of Control Errors Over Time



(b) Evolution of Control Errors Over Time



(c) Evolution of Joint Velocities Over Time



(d) Motion of the Mobile Base and End-Effector in the X-Y plane

Fig. 20: (a), (b), (c) Plots for Task H on the Real Robot.

4) Once an object of interest (i.e., having an aruco marker) is detected, the exploration is halted and an obstacle-free path is planned to a region close to the object (a region we term *pick vicinity*) using the planning node.
5) The controller node (based on the Dynamic Window Approach [4]) is then called upon to move the robot to the pick vicinity.
6) Once there, a second detection of the aruco marker is taken to get a more precise end-effector position for the task priority to take over.
7) This point is post-processed to get the top-center of the object, then the full robot's task priority node is used to move into position to pick the object while running obstacle avoidance the highest-priority task.
8) With the arm's TP node and a call to the vacuum's service, the object is picked and moved to the robot's platform.
9) At this point, steps 4b and 5 are repeated but with the designated placing position.
10) Once the robot is in the placing vicinity, the task priority is used to retrieve the object from the robot's platform and place it in the aforementioned placing position.
11) The task is complete for one object and the arm is returned to a safe position in readiness to keep exploring if there are still unexplored regions.

The final behavior tree of the integration task is found in figure 21. The behaviors that integrate the intervention hands-on are encircled with green boxes, those of hands-on perception are encircled with pink boxes, and those of hands-on planning are encircled with blue boxes. Throughout the autonomous task, pose-based ICP SLAM is used, so it is not included as a behavior.

6 Conclusion

This research project focused on designing a kinematic control system for the Kobuki Turtlebot2 robot that is fitted with the uFactory uArm Swift Pro to enable it to perform set tasks like picking, transporting, placing, etc. The developed architecture incorporated recursive task-priority implementations for the arm and for the full robot. Different tasks like end-effector position, orientation, configuration, pick, place, pick-transport-place with dead-reckoning, and pick-transport-place with dead-reckoning and ArUco marker detection were implemented and sufficiently tested in simulation and on the hardware. The relevant results have been documented in this report as well as in the videos and presentation submitted. The results demonstrated the versatility and applicability of the task-priority redundancy resolution algorithm in both real-world scenarios and simulation environments while also showing its limitations for relatively strict tasks like configuration tasks, especially in the presence of joint limits.

For future work, the focus will be on coming up with a derivative of the task priority algorithm that handles tasks in a more adaptive method. For example, when a

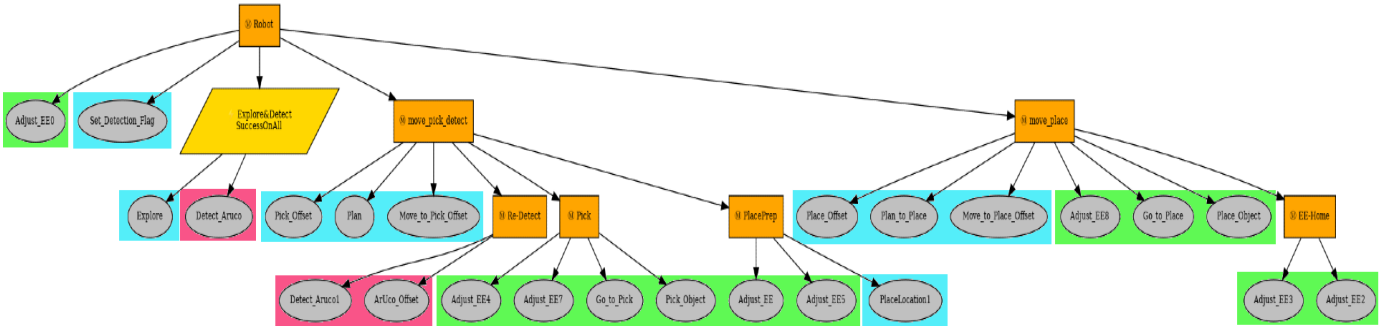


Fig. 21: Behavior Tree for the Integration Task

configuration task is defined, it may be better to violate the orientation in the early stages of fulfilling the task in favor of the desired position and correct the orientation error later on, and vice versa. Although this is already possible by splitting the configuration task into independent position and orientation tasks, it would be worth looking into doing this without having to split up the tasks.

APPENDIX A
FORWARD KINEMATICS OF OTHER DOFS

The equations position of the third DOF of the arm $q1$ are:

$$L_3 = 158.8cos(q_3) - 142sin(q_2) + 13.2 \quad (26)$$

$$^{A}L_{3,x} = L_3cos(q_1) \quad (27)$$

$$^{A}L_{3,y} = L_3sin(q_1) \quad (28)$$

$$^{A}L_{3,z} = -108 - 142cos(q_2) - 158.8sin(q_3) \quad (29)$$

$$^{W}L_{3,x} = L_3sin(q_1 + \theta) + 50.7cos(\theta) + x \quad (30)$$

$$^{W}L_{3,y} = -L_3cos(q_1 + \theta) + 50.7sin(\theta) + y \quad (31)$$

$$^{W}L_{3,z} = -108 - 142cos(q_2) - 158.8sin(q_3) - 198 \quad (32)$$

The equations of the position of the second DOF of the arm $q1$ are:

$$L_2 = -142sin(q_2) + 13.2 \quad (33)$$

$$^{A}L_{2,x} = L_2cos(q_1) \quad (34)$$

$$^{A}L_{2,y} = L_2sin(q_1) \quad (35)$$

$$^{A}L_{2,z} = -108 - 142cos(q_2) \quad (36)$$

$$^{W}L_{2,x} = L_2sin(q_1 + \theta) + 50.7cos(\theta) + x \quad (37)$$

$$^{W}L_{2,y} = -L_2cos(q_1 + \theta) + 50.7sin(\theta) + y \quad (38)$$

$$^{W}L_{2,z} = -108 - 142cos(q_2) - 198 \quad (39)$$

The equations of the position of the first DOF of the arm $q1$ are:

$$L_1 = 13.2 \quad (40)$$

$$^{A}L_{1,x} = L_2cos(q_1) \quad (41)$$

$$^{A}L_{1,y} = L_2sin(q_1) \quad (42)$$

$$^{A}L_{1,z} = -108 \quad (43)$$

$$^{W}L_{1,x} = L_1sin(q_1 + \theta) + 50.7cos(\theta) + x \quad (44)$$

$$^{W}L_{1,y} = -L_1cos(q_1 + \theta) + 50.7sin(\theta) + y \quad (45)$$

$$^{W}L_{1,z} = -108 - 198 \quad (46)$$

All constants in the above derivations are in millimeters; therefore, they ought to be converted into meters before calculating the results. ^{A}X - values in the frame of the arm's base; ^{W}X - values in the world frame.

APPENDIX B
VIDEO LINKS

B.1 Implementation of project tasks in simulation

- **Link for End-effector Positions**: Implementation in Simulation
- **Link for Pick-Transport-Place**: Implementation in Simulation
- **Link for Joint Limits**: Implementation in Simulation
- **Link for Joint Limits with arm base, End-Effector Configuration task**: Implementation in Simulation

B.2 Implementation on turtlebot2 in real-world environment

- Link for **implementation of move to point operation, video**: Task D*
- Link for **implementation of simple pick and place operation**: Simple pick and place
- Link for **implementation of a pick, transport, and place operation, using dead reckoning for navigation, video**: Pick, transport, and place operation using dead-reckoning
  Link of the **implementation of a pick, transport, and place operation, utilising visual feedback from ArUco marker detection and dead reckoning for navigation, video**: Pick, transport, and place operation using visual feedback from ArUco marker and dead-reckoning
- Link for **Rosbags for some tasks implemented in the real world**: Tasks F, G, and H

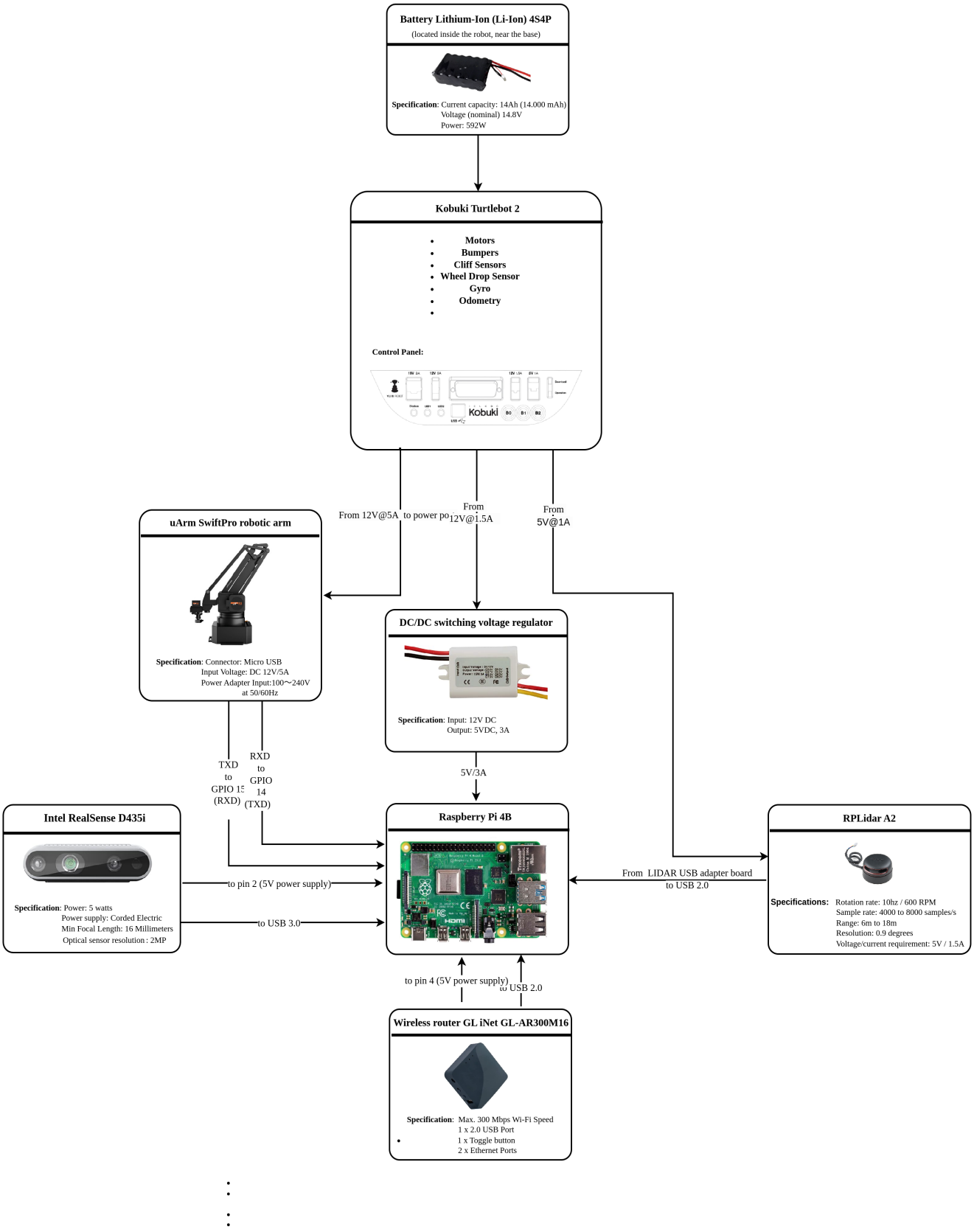


Fig. 22: Hardware Architecture

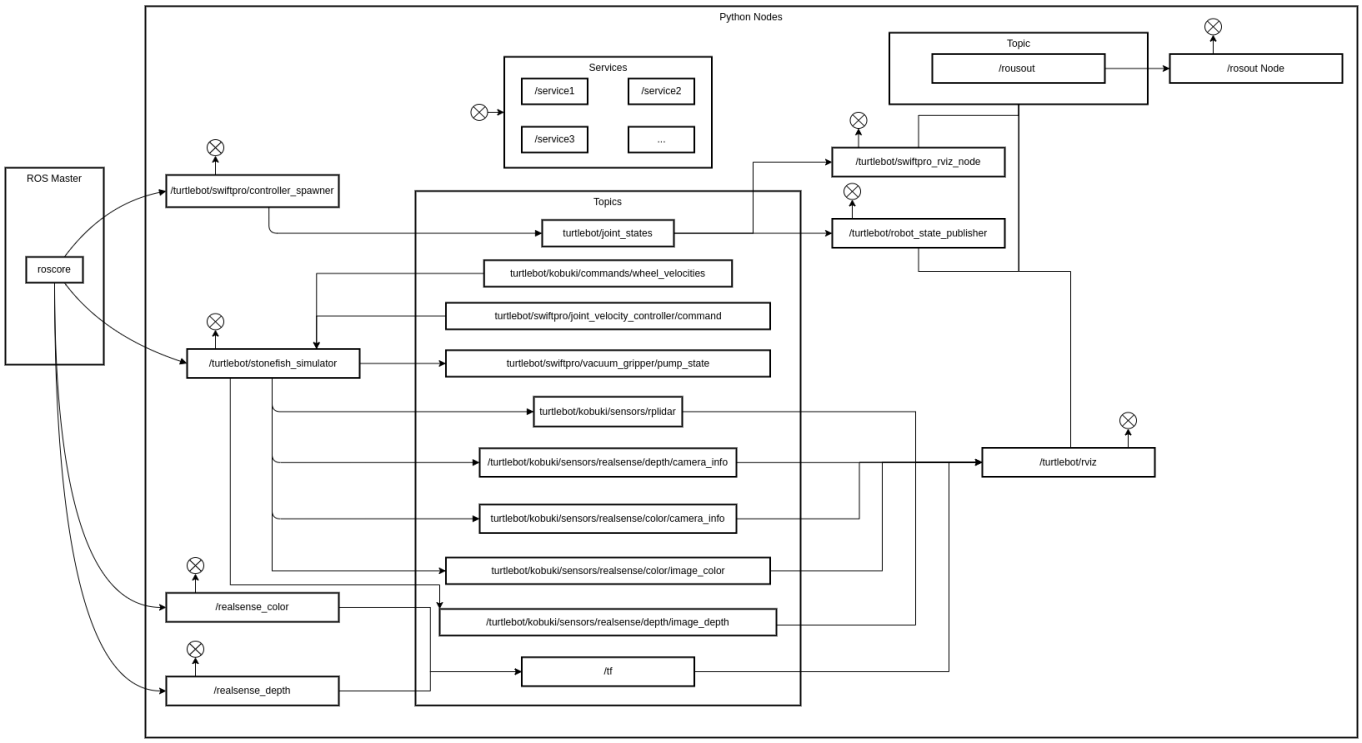


Fig. 23: Simulation Software Architecture - Nodes and Topics

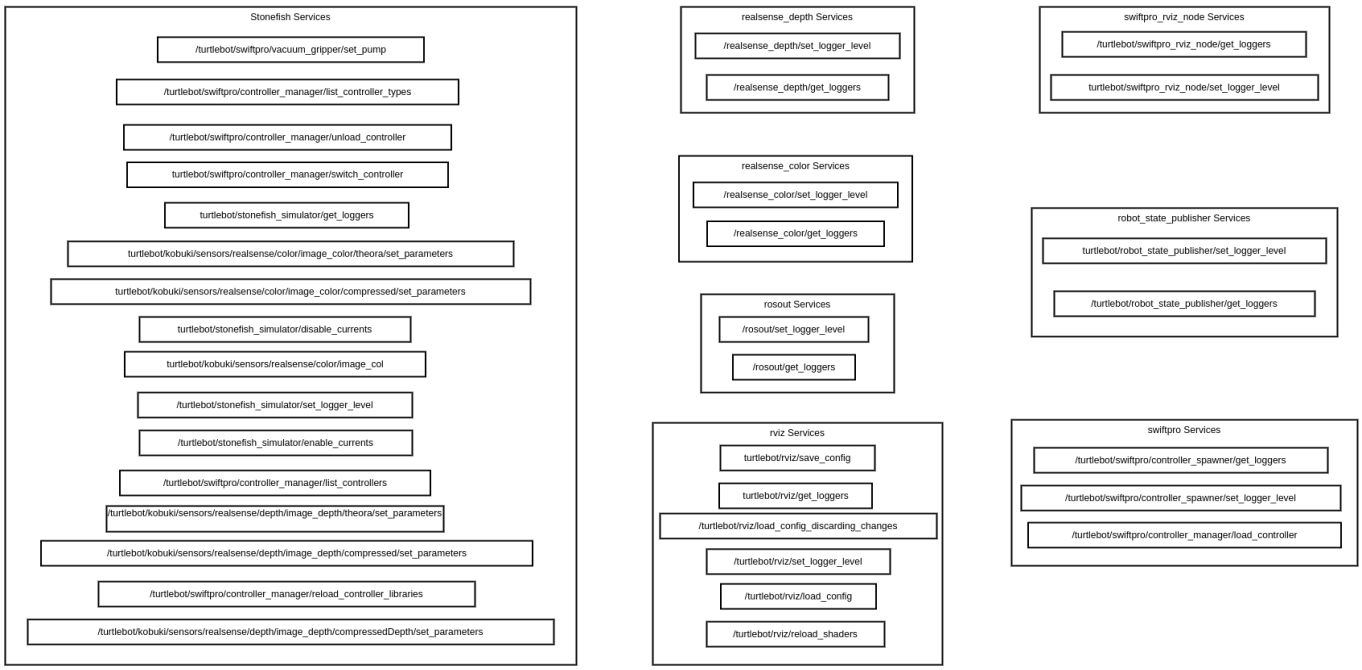


Fig. 24: Simulation Software Architecture - Services

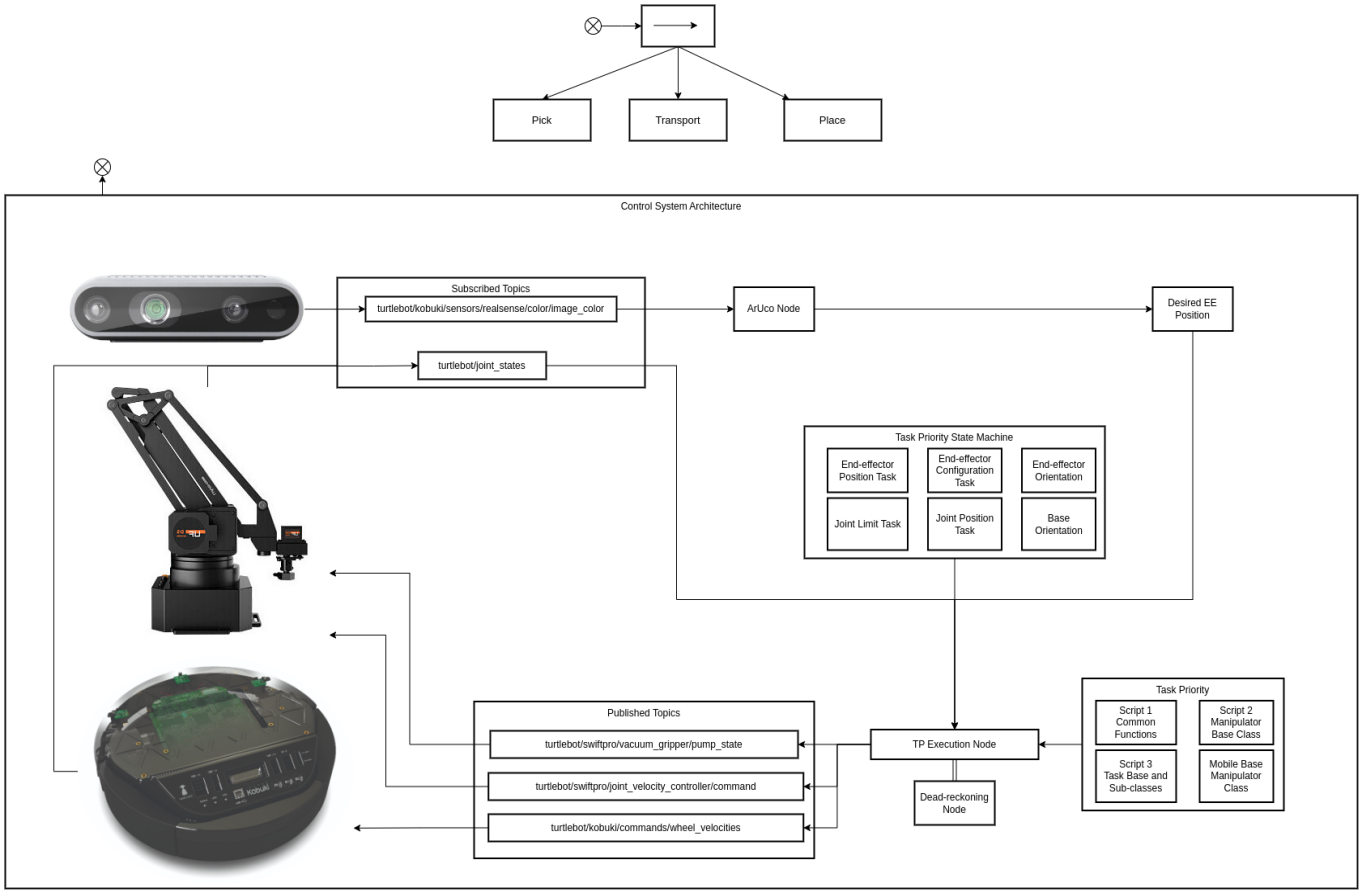


Fig. 25: Control System

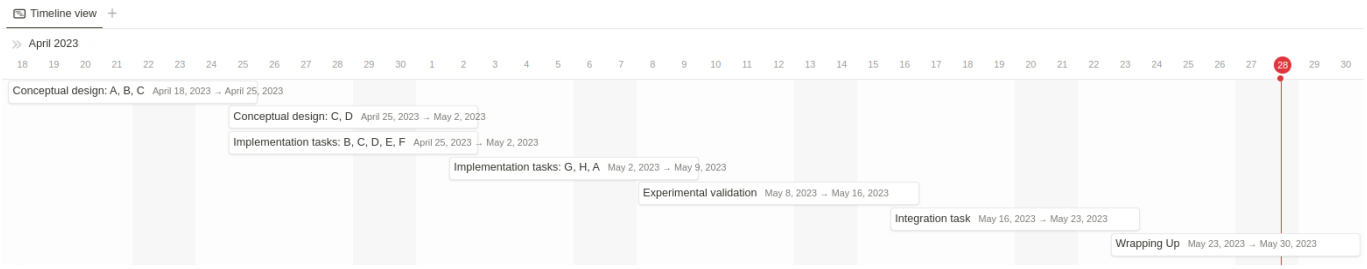


Fig. 26: Gantt Chart

B.3 Implementation of the Integration Task

- Link for **the Integration Task video**: Integration Task

REFERENCES

[1] Cieślak, P. *Hands-On Intervention Lecture Slides*, Spring, 2023.
[2] Baerlocher, P. and Boulic, R. (no date) 'Task-priority formulations for the kinematic control of highly redundant articulated structures', Proceedings. 1998 *IEEE/RSJ International Conference on Intelligent Robots and Systems.* Innovations in Theory, Practice and Applications (Cat. No.98CH36190) [Preprint]. doi:10.1109/iros.1998.724639.
[3] Splintered-Reality (no date) Splintered-reality/py_trees: Python implementation of behaviour trees., GitHub. Available at: https://github.com/splintered-reality/py_trees (Accessed: 10 June 2023).
[4] Fox, D., Burgard, W. and Thrun, S. (1997) 'The dynamic window approach to collision avoidance', *IEEE Robotics*; Automation Magazine, 4(1), pp. 23–33. doi:10.1109/100.580977.
[5] Cieślak, P. et al. (2020) 'Practical formulation of obstacle avoidance in the task-priority framework for use in robotic inspection and intervention scenarios', *Robotics and Autonomous Systems*, 124, p. 103396. doi:10.1016/j.robot.2019.103396.

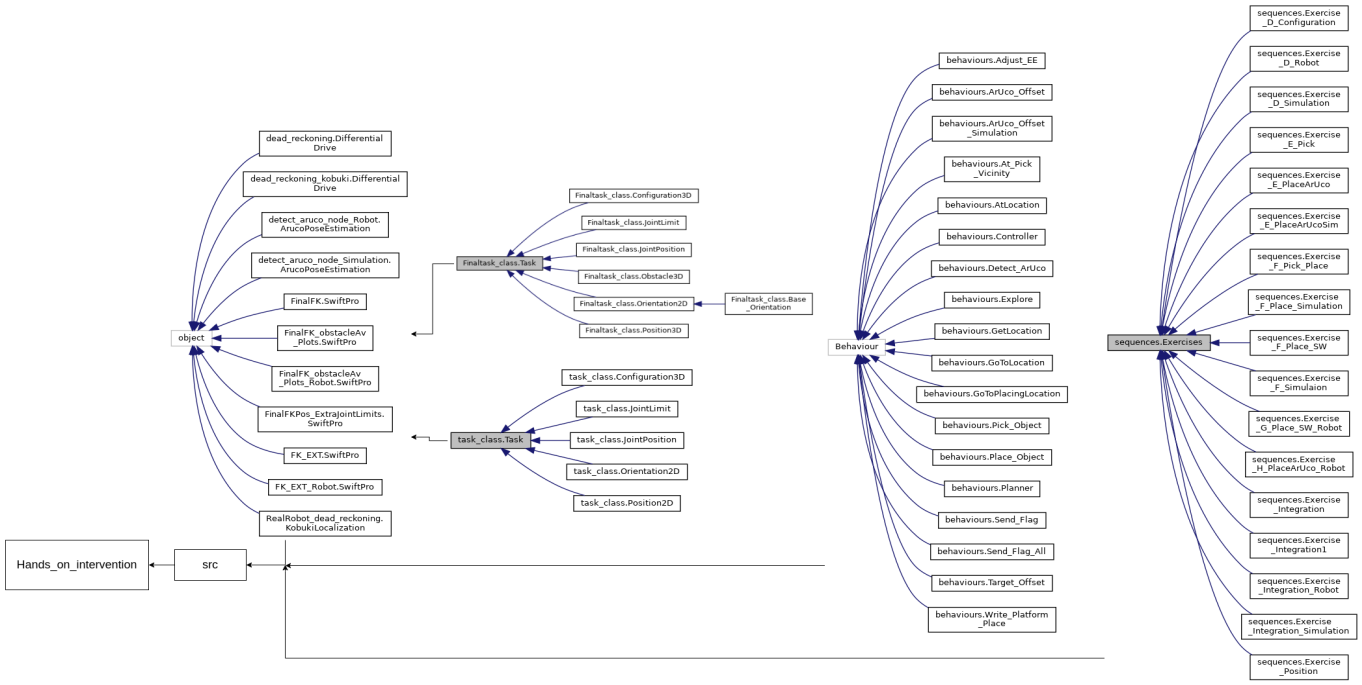


Fig. 27: Implementation Block using doxygen

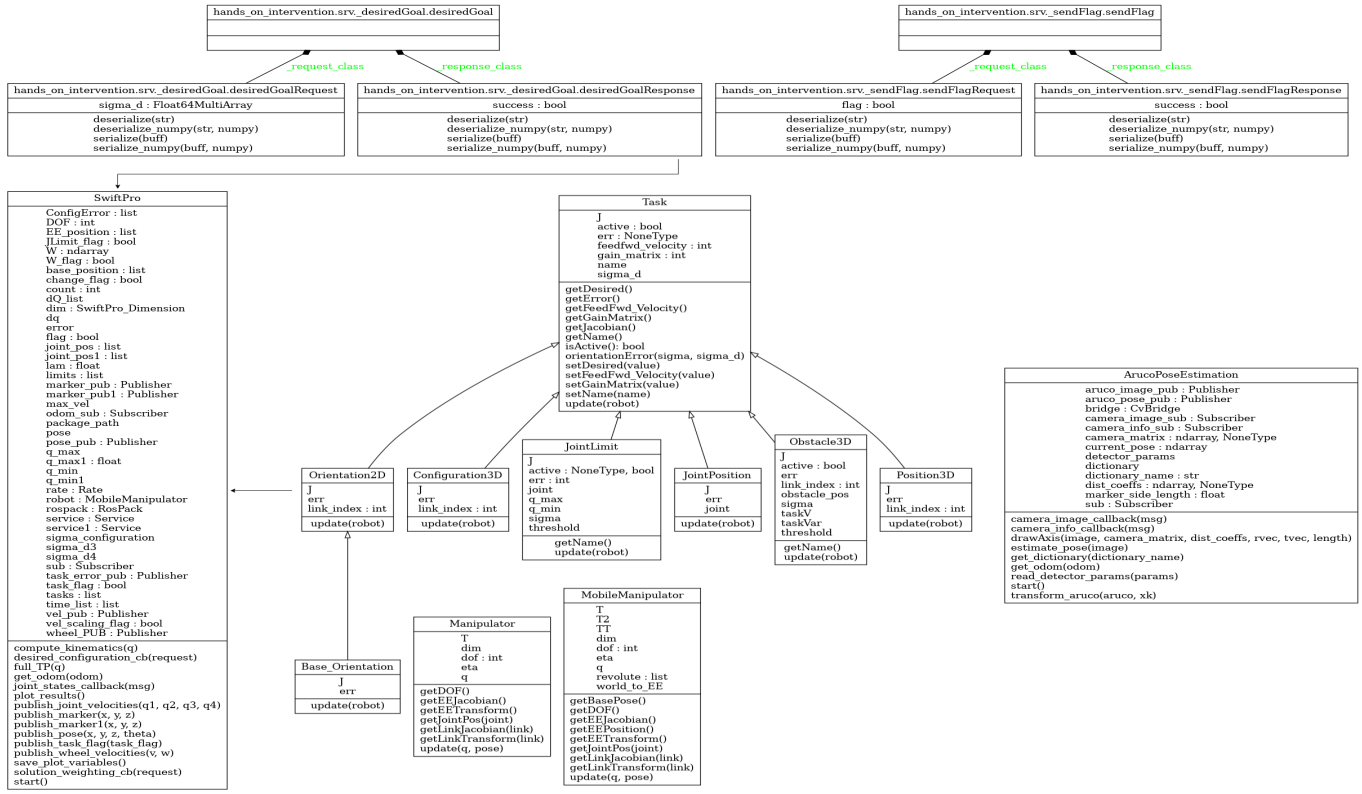


Fig. 28: Implementation Block using UML