UNIVERSITY OF TRENTO
Department of Information Engineering and Computer Science
Fundamentals of Robotics

Professors: Luigi Palopoli, Niculae Sebe, Michele Focchi, Placido Falqueto

Project Report: A mobile robot and a robotic arm to find and store objects

Authors:
Samuele Pozzani, Daniel Marcon
Enea Strambini, Giacomo Tezza

21 January 2023

Abstract

Robotics is the engineering discipline which studies and develops methods that allow a robot to perform specific tasks by autonomously reproducing human work. An introductory computer engineering course aims to achieve a general understanding of the different challenges faced during the development of a robotic application. This project focused the practical software modelling and development of a real-world problem, tested in both a simulated and a real environment. For this purpose, the group learned and applied the basics of mathematical models, kinematics and dynamics, machine learning, computer vision and the state-of-the-art software technologies for robotics development.

1 Introduction

To complete the course Fundamentals of Robotics, the group is required to develop a project which completes an assigned mission. A mobile robot, Shelfino, is equipped with a 3D camera [1] and moves in a room with four circular areas with known center and radius, and it has to localise and classify four objects (megablocks) which are positioned inside every area. A robotic arm with six degrees of freedom, UR5 [2], picks up the object and places it in a basket according to its class.

The project is organised as a sequence of three assignments of increasing complexity. The code for the project is available on GitHub at the following link, along with Doxygen documentation, the instructions for building and running the simulation and some short videos presenting the results:

https://github.com/sampozz/RoboticsProject

1.1 Assignments Specification

In the **Assignment 1**, the mobile robot has to visit all four different areas. For each area, it has to exactly localise the object and identify its class. The object can be anywhere within the circular area, but it is positioned in a natural configuration (base on the ground).

In the **Assignment 2**, after Shelfino has localised and classified the object (which is positioned in an arbitrary position), an operator moves the block to the UR5 workbench. The robotic arm picks the block and stores it in the right basket.

In the **Assignment 3**, after Shelfino has localised and classified the object, an operator loads the block on top of the mobile robot which then parks near the UR5 workbench. A second 3D camera [3] repeats the classification and finds the exact position of the block. This information is used by the robotic arm to pick the object from the mobile robot and store it in the correct basket.

2 ROS Codebase Architecture

The project is implemented using the ROS middleware [4] and the Catkin build tools. The Robot Operating System (**ROS**) is the de-facto standard for robot software development, and it provides services designed for a heterogeneous computer cluster such as hardware abstraction, message-passing between processes and package management.

The codebase is designed like a web application with distributed microservices architecture [5] leveraging most of the functionalities provided by ROS, including messages over topics, remote procedure calls with services and the parameter server. An high-level view of the infrastructure with its most important building blocks is shown in figure 1, and the fundamental concepts are explained in the following subsections.

2.1 Main Controller

The Main Controller is the principal component for the high level planning and it can be seen as the front-end of a web application. It contains the implementation of the Finite State Machine described in section 3. Main Controller is a Catkin Package containing a single ROS Node, `fsm_controller.cpp`, which communicates with the other nodes using RPC via ROS Services. In this way, the only concern of the Main Controller consists of managing the high-level states, as the underlying implementation of the robotic functions is completely abstracted by the most specialized controllers. The `fsm_utils.cpp` library contains the functions used to communicate with the other controllers.

In order to deal with the three different assignments, the state functions had to be re-implemented three times. Nevertheless, a clean interface could be maintained by declaring the state functions with the same names in three custom C++ namespaces, one for each assignment, into the `fsm.h` header file.

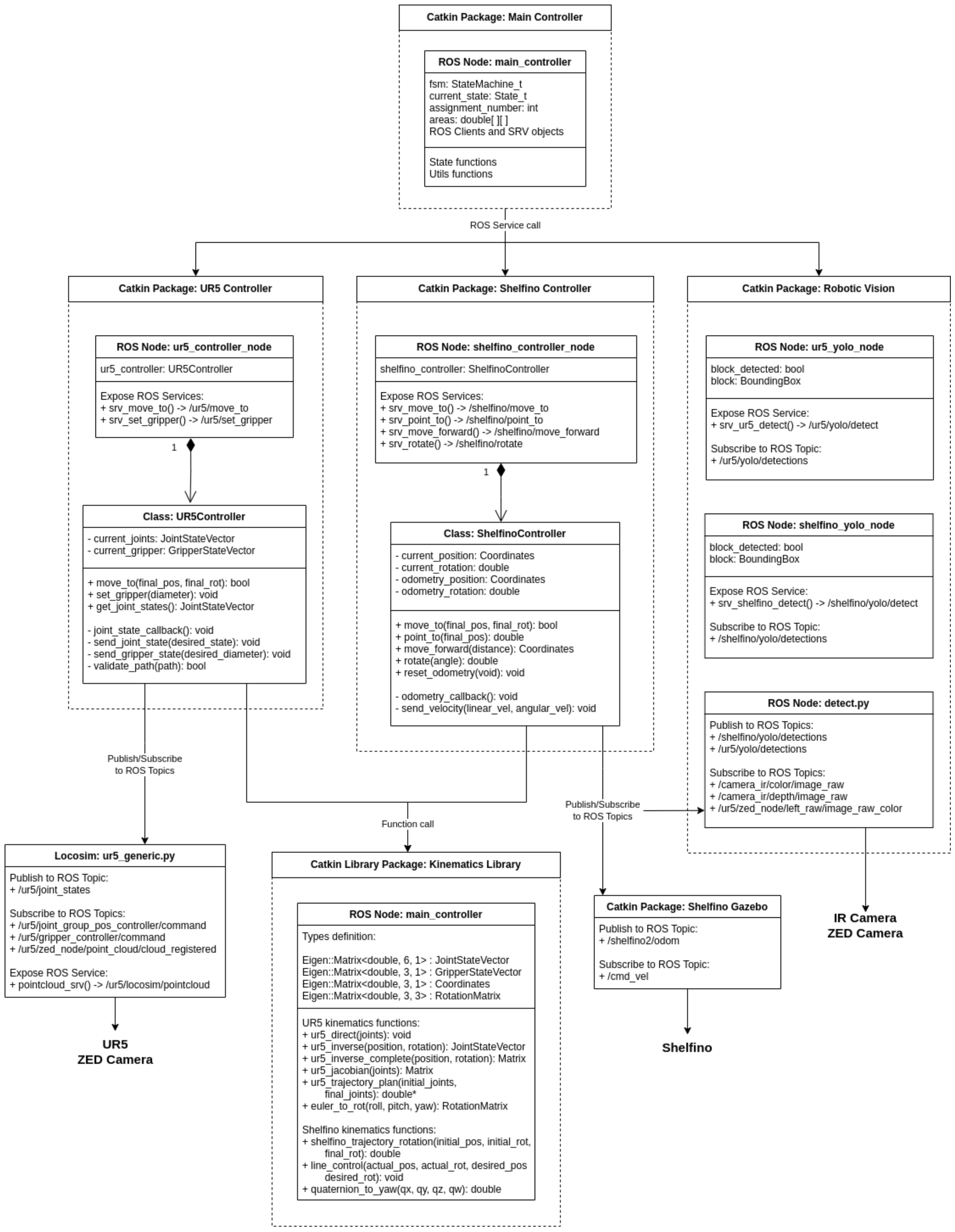
Catkin Package: Main Controller
ROS Node: main_controller
fsm: StateMachine_t
current_state: State_t
assignment_number: int
areas: double[ ][ ]
ROS Clients and SRV objects
State functions
Utils functions
ROS Service call
Catkin Package: UR5 Controller
ROS Node: ur5_controller_node
ur5_controller: UR5Controller
Expose ROS Services:
+ srv_move_to() -> /ur5/move_to
+ srv_set_gripper() -> /ur5/set_gripper
1
Class: UR5Controller
- current_joints: JointStateVector
- current_gripper: GripperStateVector
+ move_to(final_pos, final_rot): bool
+ set_gripper(diameter): void
+ get_joint_states(): JointStateVector
- joint_state_callback(): void
- send_joint_state(desired_state): void
- send_gripper_state(desired_diameter): void
- validate_path(path): bool
Catkin Package: Shelfino Controller
ROS Node: shelfino_controller_node
shelfino_controller: ShelfinoController
Expose ROS Services:
+ srv_move_to() -> /shelfino/move_to
+ srv_point_to() -> /shelfino/point_to
+ srv_move_forward() -> /shelfino/move_forward
+ srv_rotate() -> /shelfino/rotate
1
Class: ShelfinoController
- current_position: Coordinates
- current_rotation: double
- odometry_position: Coordinates
- odometry_rotation: double
+ move_to(final_pos, final_rot): bool
+ point_to(final_pos): double
+ move_forward(distance): Coordinates
+ rotate(angle): double
+ reset_odometry(void): void
- odometry_callback(): void
- send_velocity(linear_vel, angular_vel): void
Catkin Package: Robotic Vision
ROS Node: ur5_yolo_node
block_detected: bool
block: BoundingBox
Expose ROS Service:
+ srv_ur5_detect() -> /ur5/yolo/detect
Subscribe to ROS Topic:
+ /ur5/yolo/detections
ROS Node: shelfino_yolo_node
block_detected: bool
block: BoundingBox
Expose ROS Service:
+ srv_shelfino_detect() -> /shelfino/yolo/detect
Subscribe to ROS Topic:
+ /shelfino/yolo/detections
ROS Node: detect.py
Publish to ROS Topics:
+ /shelfino/yolo/detections
+ /ur5/yolo/detections
Subscribe to ROS Topics:
+ /camera_ir/color/image_raw
+ /camera_ir/depth/image_raw
+ /ur5/zed_node/left_raw/image_raw_color
Publish/Subscribe
to ROS Topics
Function call
Publish/Subscribe
to ROS Topics
Locosim: ur5_generic.py
Publish to ROS Topic:
+ /ur5/joint_states
Subscribe to ROS Topics:
+ /ur5/joint_group_pos_controller/command
+ /ur5/gripper_controller/command
+ /ur5/zed_node/point_cloud/cloud_registered
Expose ROS Service:
+ pointcloud_srv() -> /ur5/locosim/pointcloud
UR5
ZED Camera
Catkin Library Package: Kinematics Library
ROS Node: main_controller
Types definition:
Eigen::Matrix<double, 6, 1> : JointStateVector
Eigen::Matrix<double, 3, 1> : GripperStateVector
Eigen::Matrix<double, 3, 1> : Coordinates
Eigen::Matrix<double, 3, 3> : RotationMatrix
UR5 kinematics functions:
+ ur5_direct(joints): void
+ ur5_inverse(position, rotation): JointStateVector
+ ur5_inverse_complete(position, rotation): Matrix
+ ur5_jacobian(joints): Matrix
+ ur5_trajectory_plan(initial_joints,
final_joints): double*
+ euler_to_rot(roll, pitch, yaw): RotationMatrix
Shelfino kinematics functions:
+ shelfino_trajectory_rotation(initial_pos, initial_rot,
final_rot): double
+ line_control(actual_pos, actual_rot, desired_pos
desired_rot): void
+ quaternion_to_yaw(qx, qy, qz, qw): double
Catkin Package: Shelfino Gazebo
Publish to ROS Topic:
+ /shelfino2/odom
Subscribe to ROS Topic:
+ /cmd_vel
Shelfino
IR Camera
ZED Camera

Figure 1: High level view of the codebase in the ROS Architecture

2.2 UR5 and Shelfino Controllers

Two different packages are used to implement the motion algorithms for the UR5 and Shelfino robots, see sections 4.3 and 5.1 for the details. In the web application analogy, this part represents the back-end, where the business logic is processed. In both the packages, a C++ class is defined as robot controller which contains ROS publishers and subscribers to communicate with real or simulated robot. A `NodeHandle` object is a private attribute of the class, so that it can be used as a standalone node, and potentially multiple robots can be controlled with this code.

A `*_controller_node.cpp` node is created as an interface for this project to instantiate a robot controller object, expose the ROS Services and manage the corresponding methods on the robot controller. This node is the end point for the communication with the Main Controller.

In the **Kinematics Library** package, a collection of C++ functions is implemented to support the motion operations and algorithms of the robot controllers.

2.3 Robotic Vision

The Robotic Vision package is the software component intended for managing the input from the 3D cameras. The `detect.py` source is a Python node which applies the YOLOv5 [6] object detection algorithm (described in section 6) on the image data retrieved from the camera topics. This project requires two cameras, therefore two instances of this node are executed simultaneously. The result of the detection algorithm is published on a ROS Topic and is handled by the other C++ nodes as follows.

The ROS Nodes `ur5_yolo_node` and `shelfino_yolo_node` are used to filter the results of the YOLO algorithm and provide a clean interface to the Main Controller via an exposed ROS Service. Both the nodes contain a topic subscription to communicate with the `detect.py` node described above: if a block is identified, its bounding box is stored in a buffer with a related timestamp, so that the Main Controller may request this information at any time.

The `ur5_yolo_node` also communicates with the Locosim `ur5_generic.py` node to gather the Point Cloud information from the ZED Camera via ROS Service call.

3 High Level Planning

3.1 Finite State Machine

A straightforward way to plan the operations of the robots and correctly manage the execution of the numerous functions for achieving the assigned goals consists of modelling a Finite State Machine (FSM). Mathematically, a FSM can be defined as an abstract machine which is in exactly one of a finite number of states at any given time [7]. A state is executed as a C++ procedure and is associated with a state number. The `StateMachine_t` type is therefore a map defined into `fsm.h` as follows:

```
// Type definition for a pointer to state procedure of the FSM
typedef void (*state_function)(void);
```

```
// Type definition for the FSM
typedef std::map<int, state_function> StateMachine_t;
```

The state numbers are given a unique mnemonic identifier using a C++ `enum` which is defined as `State_t`.

3.2 Assignment 3 FSM

A simplified view of the FSM designed to complete the third assignment of this project is shown in figure 2. The FSMs designed for the other two assignments are not reported since they are just a subset of the one presented here.

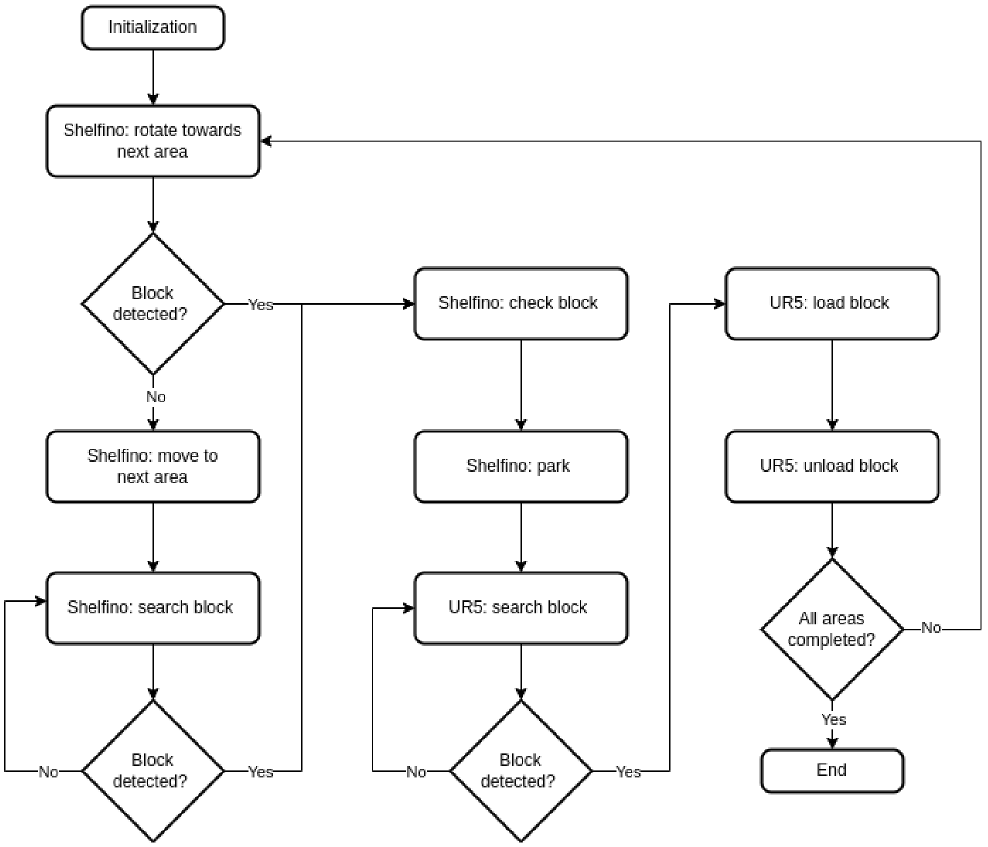

Figure 2: Finite State Machine designed to complete Assignment 3

Follows a brief description of the state functions executed by the Main Controller. Whenever Shelfino or UR5 are moved or the vision nodes are checked, it is a consequence of a ROS Service call invoked by the Main Controller.

- **Initialization**: Initialize global variables and constants. Move UR5 to the selected home position.
- **Shelfino: rotate towards next area**: The coordinates of the center of the next area to be checked are loaded. Shelfino rotates towards the requested position.

- **Block detected**: This building block represents a decision. Shelfino or UR5 vision nodes are asked to check whether a block was detected and buffered in the few previous seconds. In case a block was detected, its distance from Shelfino is computed using the depth camera and a correction angle is decided based on the bounding box position. The class of the detected block is finally stored along with its probability.

- **Shelfino: move to next area**: Shelfino moves forwards by the computed distance, and the movement is controlled with the Lyapunov Function as described in section 5.3.

- **Shelfino: search block**: Shelfino is now in the center of the current area and it rotates on its position to find a block. The area's radius is assumed to be relatively small (max 2 meters), since Shelfino has to move inside the laboratory.

- **Shelfino: check block**: A block and its position was found. Shelfino moves closer, repeats the classification and keeps the result with the greater probability. The currently identified block is now blacklisted, so that it will not be checked again. Finally, Shelfino's position is checked to decide in which area it is actually located.

- **Shelfino: park**: The detected block is moved on top of Shelfino and the robot moves to the known parking position.

- **UR5: search block**: A request to the ZED Camera is sent to identify the exact position of the block on top of Shelfino combining the bounding box with the point cloud. The classification is also repeated by the ZED Camera.

- **UR5: load block**: Check the last classification result and compare the probability with the classification made by Shelfino: keep the most accurate one. Select the correct basket based on the block class. The UR5 is moved to the block position and the gripper is closed.

- **UR5: unload block**: UR5 is moved to the selected basket and the gripper is opened.

4 UR5 Robotic Arm

4.1 Kinematic Model

A manipulator consists of a series of rigid bodies called links. Links are connected by kinematic pairs called joints. The whole structure forms a kinematic chain. One end of the chain is connected to a base, the other end is connected to an end effector (gripper) that is used for the manipulation activities.

Given a manipulator with joint variables $\boldsymbol{q} = [q_1, q_2, ..., q_N]^T$, let $\boldsymbol{p}$ be the pose (position and orientation) of the end effector. The **direct kinematic** problem is about finding a function $f(\cdot)$ such that $\boldsymbol{p} = f(\boldsymbol{q})$ [8]. Likewise, the **inverse kinematic** problem is about finding the function $f^{-1}(\cdot)$ such that $\boldsymbol{q} = f^{-1}(\boldsymbol{p})$ [9] [10].

The UR5 robotic arm is composed of six joints, whose values can be controlled by sending the state vector on the proper ROS Topic. Following the Denavit-Hartenberg convention, the reference frames are chosen as showed in figure 3.

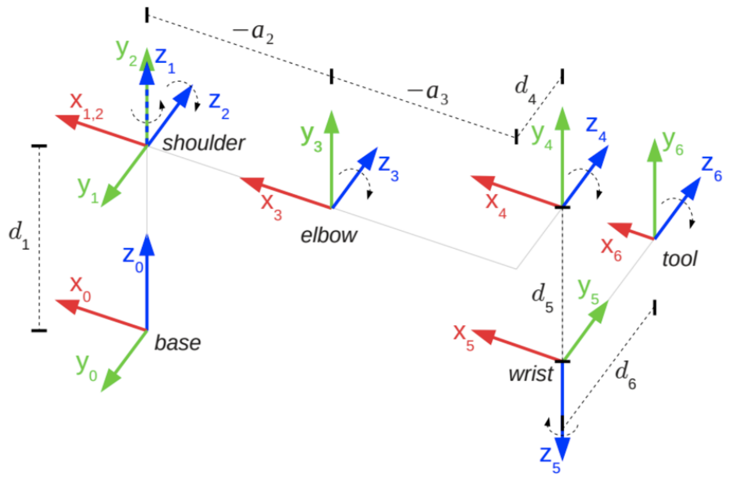

Figure 3: UR5 robotic arm kinematic model

With this model in mind and the proper geometric background, the functions for direct and inverse kinematic functions are implemented in the `kinematic_lib` package using C++ and leveraging the Eigen library. In particular, `ur5_inverse_complete()` function returns a 8x6 matrix whose lines contain the eight possible solutions of the inverse kinematic: every solution is processed in the motion planning phase to determine the best movement trajectory.

4.2 Trajectory Planning

The **path** is defined as a locus of points that the robot has to follow in order to move from an initial to a final configuration. The **trajectory** is a path associated with a function of time that specifies (for each time) the desired configuration of the manipulator.

The `ur5_trajectory_plan()` function in the kinematics library computes a trajectory in the joints space by creating a function which connects an initial configuration of the joints $\boldsymbol{q}_s$ with a final (desired) configuration $\boldsymbol{q}_f$. As no via-points are considered, it is only necessary to ensure that:

$$q(t) = \begin{cases} \boldsymbol{q}_s & t = t_s \\ \boldsymbol{q}_f & t = t_f \end{cases} \tag{1}$$

It is possible to see that the energy is minimised by choosing a higher degree polynomial [11]. In particular, a cubic polynomial is used: $q(t) = a_3t^3 + a_2t^2 + a_1t + a_0$ where the coefficients $\boldsymbol{a}$ are found by imposing the terminal constraints on velocity and position and solving a system of linear equations:

$$\begin{aligned} q(t_s) &= a_3t_s^3 + a_2t_s^2 + a_1t_s + a_0 = \boldsymbol{q}_s \\ q(t_f) &= a_3t_f^3 + a_2t_f^2 + a_1t_f + a_0 = \boldsymbol{q}_f \\ q(t_s) &= 3a_3t_s^2 + 2a_2t_s + a_1 = \dot{\boldsymbol{q}}_s \\ q(t_f) &= 3a_3t_f^2 + 2a_2t_f + a_1 = \dot{\boldsymbol{q}}_f \end{aligned} \tag{2}$$

4.3 Motion Algorithm

The `ur5_move_to()` function defined in the `UR5Controller` class is the main procedure used to move the end effector to a desired final position. The function only requires the final position `JointStateVector pos` and rotation `RotationMatrix rot` as parameters to execute the algorithm 1. The algorithm contains the following function calls:

- **sortIKResult**: computes the norm of the difference between the initial and final joint configurations and sorts the eight inverse kinematics results in ascending order. Heuristically, this function enables the motion controller to test the "easiest" or "shortest" trajectories first, and the results are quite effective.
- **validatePath**: this functions decides whether the trajectory is feasible or not. Given a list of intermediate joint configurations, for every configuration follow the steps:
  1. Compute the direct kinematics to find the cartesian position of the end effector: return false if it collides with the table.
  2. Compute the determinant of the Jacobian matrix: return false if it is near zero (moving into a singularity).
  3. Compute the minimum singular value of the Jacobian matrix: return false if it is near zero (moving into a singularity).
- **sendJointStatesFiltered**: sends the desired joint states to ROS topic to move the UR5 robotic arm. In order to avoid steps, it first filters the desired final configuration.

5 Shelfino Mobile Robot

The mobile robot used in this project is Shelfino, an advanced robot with two motorized wheels and two caster wheels made by the University Department. It also includes a Lidar sensor (not used in this project) and an odometry topic which use data from motion sensors to estimate change in position over time.

5.1 Mobile Robot Model

As shown in figure 4, the generalised coordinate chosen to localise the vehicle on the plan of motion are $\boldsymbol{q} = [x, y, \theta]^T$, where (x, y) represents the midpoint of the traction wheels axle, while θ is the orientation of the vehicle with respect to the horizontal X axis. The vehicle cannot translate in the direction of the wheel axle [12].

5.2 Motion Planning

When the development started, the only available way to move Shelfino in the simulation environment was by sending the velocity values to the specified ROS Topic. Shelfino can move forward if it receives a linear velocity from the topic and it can rotate if it receives an

Algorithm 1 UR5 end-effector movement function

```
  Input:
   JointStateVector pos      // The desired final position of the end-effector
   RotationMatrix rot        // The desired final rotation of the end-effector
   int n                     // The number of intermediate configurations

  procedure UR5MoveTo
      $initialJoints \leftarrow currentJoints$          ▷ get current configuration from ROS topic
      $IKResult \leftarrow$ UR5InverseComplete($pos, rot$)
      sortIKResult($IkResult, initialJoints$)
      $isValid \leftarrow false$
      for $i \leftarrow 0$ to 7 do                        ▷ Check all configurations computed by IK
          $finalJoints \leftarrow IKResult[i]$
          $path \leftarrow$ UR5TrajectoryPlan($initialJoints, finalJoints, n$)
          if validatePath($path, n$) then
              $isValid \leftarrow true$
              break                               ▷ Found a valid path, skip other configurations
          end if
      end for
      if not $isValid$ then
          return                                            ▷ No valid path could be found
      end if
      for $i \leftarrow 0$ to $n$ do                   ▷ Move UR5 throughout the computed trajectory
          $intermediateJoints \leftarrow path[i]$
          sendJointStatesFiltered($intermediateJoints$)
      end for
  end procedure
```

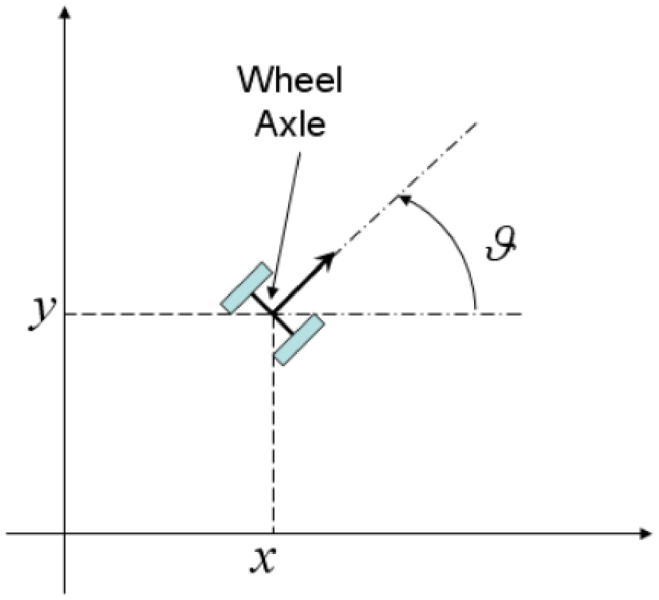

Figure 4: Shelfino Model

angular velocity. Therefore, the motion planning is based on the velocity topic, even though a new topic was added later which accepts a path message.

The simplest way to move this mobile robot from an initial point $\boldsymbol{p}_s$ to a final point $\boldsymbol{p}_f$ consists of rotating the robot to look towards the final point and then moving it forward following a linear trajectory.

For the first rotation it is necessary to compute a trajectory angle α as in equation 3 (shown in figure 5), therefore the mobile robot will rotate clockwise of $\beta = \theta - \alpha$. Shelfino is rotated at constant angular velocity ω for a time $t_{rot} = \beta/\omega$. It is important to choose the right rotation direction to optimize the movement.

Finally, Shelfino moves forward at constant linear velocity v for a time $t_{lin} = dist/v$, where $dist$ is the distance between $\boldsymbol{p}_s$ and $\boldsymbol{p}_f$ as shown in equation 4.

$$\alpha = arctan_2(\Delta y, \Delta x) \tag{3}$$

$$dist = \sqrt{(x_f - x_s)^2 + (y_f - y_s)^2} \tag{4}$$

5.3 Motion Control

In the real world it is not possible to simply move the robot and expect the movement to be as precise as it was imagined. This issue leads to localization error, and it happens because the robot has a dynamics and does not respond instantaneously to commands; sensors are actuated by noise and actuators are not perfectly accurate. Therefore, the robot deviates from the ”ideal” trajectory and the error accumulates over time: at the end it may be far away from where it is expected to be. For this reason a control algorithm is needed [13].

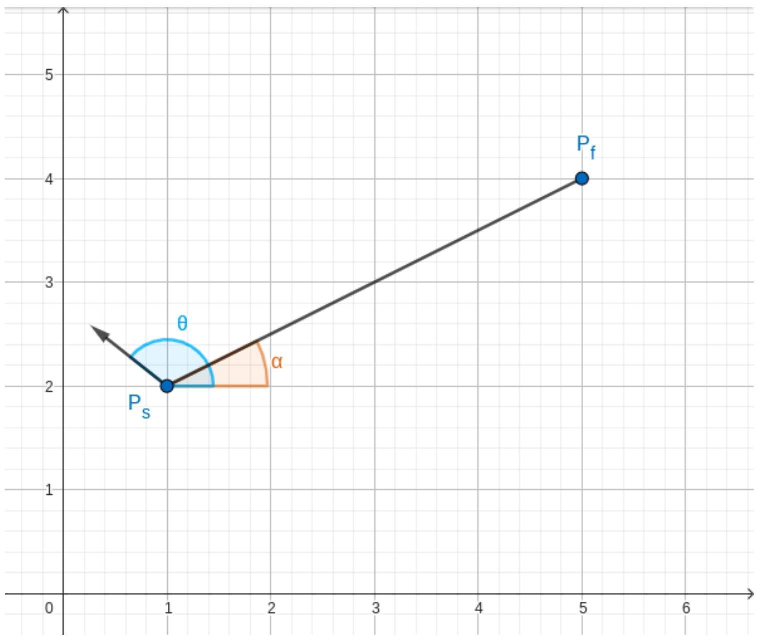

Figure 5: Shelfino first rotation

It is possible to use **Lyapunov function** to construct a feedback control that guarantees "by construction" the convergence to a desired trajectory [14]. A reference robot moves according to the following model:

$$\dot{\boldsymbol{p}}_d = \begin{bmatrix} \dot{x}_d \\ \dot{y}_d \\ \dot{\theta}_d \end{bmatrix} = \begin{bmatrix} v_d \cos(\theta_d t) \\ v_d \sin(\theta_d t) \\ \omega_d \end{bmatrix} \tag{5}$$

Where $\boldsymbol{p}_d$ is the desired position of Shelfino along the trajectory, v_d and ω_d are the desired linear and angular velocities respectively. Also, an actual robot with state evolution is given:

$$\dot{\boldsymbol{p}}_a = \begin{bmatrix} \dot{x}_a \\ \dot{y}_a \\ \dot{\theta}_a \end{bmatrix} = \begin{bmatrix} v_a \cos(\theta_a t) \\ v_a \sin(\theta_a t) \\ \omega_a \end{bmatrix} \tag{6}$$

Where $\boldsymbol{p}_a$ is the actual position of Shelfino, v_a and ω_a are the linear and angular velocities respectively which come from the odometry data. Objective of the Lyapunov control is to find a control law such that the state $\boldsymbol{p}_a(t)$ converges to the state $\boldsymbol{p}_d(t)$.

Let $v = v_d + \delta_v$ and $\omega = \omega_d + \delta_\omega$, by minimizing the error function the following equations are obtained, which are used to correct the trajectory of Shelfino during the linear movement towards the final point:

$$\delta_v = -k_p e_{xy} \cos(\theta - \psi) \tag{7}$$

$$\delta_w = -v_d sinc(\frac{e_\theta}{2}) \sin(\psi - \frac{\alpha}{2}) - k_\theta e_\theta \tag{8}$$

Where k_p and k_θ are the gain values, e_{xy} and e_θ are the errors between desired and actual positions/rotations, ψ is the angle between the x error and the y error and α is the sum of θ_d and θ_a.

The values v and ω are sent to the velocity topic as linear and angular velocities respectively, in order to move the robot in a controlled way.

6 Robotic Vision

Two of the challenges to overcome are the localization and the classification of the MegaBlocks placed around the field. Given the complexity of the problem, computer vision and machine learning are probably the best technologies to implement. Therefore, the choice is to use the already well established YOLOv5 model, written in Python and PyTorch.

6.1 You Only Look Once

YOLOv5 is a model suited for object detection and classification, it uses a series of convolutional layers for feature extraction and a few fully connected layers for classification. The main advantage of using YOLO is the possibility to start with a pre-trained model, in order to use a smaller dataset in the training phase and train mainly the last layers of the model.

The output of the model is a table containing all the detected bounding boxes and the probability of belonging to a certain class. There are also several parameters that may be configured during the inference phase to reduce the amount of false positives.

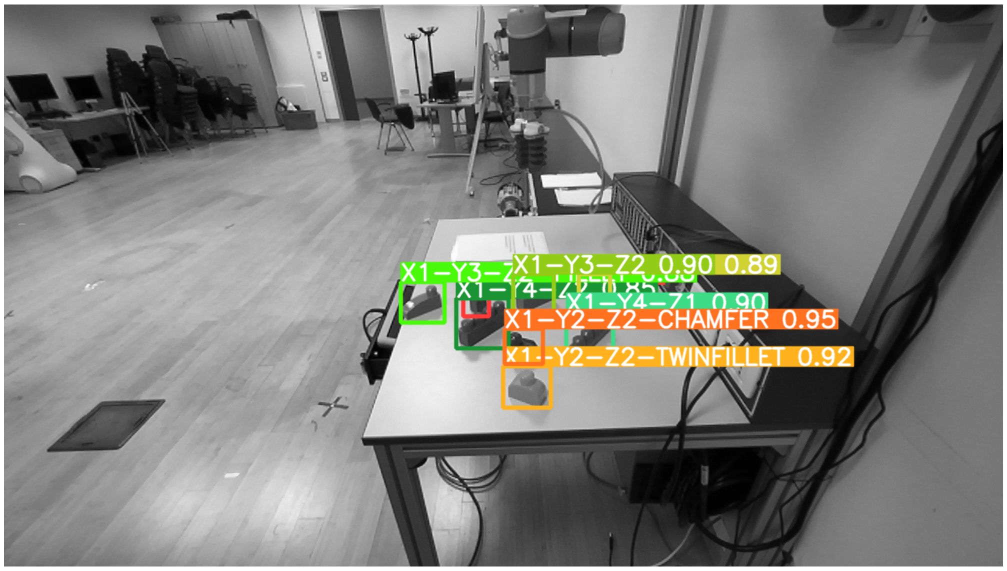

Figure 6: YOLOv5 output result. A bounding box with the class name is created for every detected object.

6.2 Training

The YOLOv5 algorithm is trained using a mix of over 500 images rendered with Blender and a set of over 100 images taken in a variety of different environments and manually labeled. In the dataset there are also 100 *background* images without any labels, for drastically decreasing false positives. Everything is trained using **YOLOv5s** pre-trained weights as basis.

The images from Blender are generated automatically using the Python Blender library, each one with different color, location and orientation. The images are saved and automatically labeled using open-cv and Python.

The results with the rendered images only are good in the simulation but the model does not generalize well in a real world scenario. The solution is to also photograph 100+ images and label them using label-studio [15]. The photos are taken in a variety of lighting environments and on different surfaces.

6.3 Metrics

The model performs well, especially in the real world, which is the focus of the training. False positives are rare and the box position accuracy is very good. Metrics on validation data are present, but the set is composed of just 80 images, hence sacrifices are made to keep the training dataset as large as possible, given the relatively few images available.

The **box_loss**, i.e. the bounding box regression loss (mean squared error), are 0.014 and 0.046 for training and validation sets respectfully. The **cls_loss**, i.e. the classification loss (cross entropy), are 0.0016 and 0.014 for training and validation sets respectfully.

Currently, the only problem is the classification error with cubes of different heights, which are sometimes swapped, and with images of blocks turned around, thus hiding their main characteristic (like the **chamfer** blocks).

6.4 Object Localization

The bounding box information retrieved from YOLO detection algorithm is very useful together with the 3D data retrieved from the depth map and point cloud of the ZED and RealSense cameras.

The detection node related to Shelfino computes YOLO on every image it receives on the topic. Whenever an object is detected, a bounding box is created. The same detection node also retrieves the depth data published by the RealSense camera to the proper topic and, based on the position of the bounding box, it gets the distance from the camera to the detected object. Some simple trigonometric calculations are used to express the object position with respect to Shelfino position (instead of the camera).

A similar approach is used with the ZED camera positioned near the UR5 table. In this case, the point cloud data gives more useful 3D information used by the UR5 to compute a trajectory and grasp the detected object. A transform function is applied to the point cloud to get the euclidean position with respect to the world frame (instead of the camera frame).

Assignment	Total time	Localization accuracy	Classification errors	UR5 time
1	4m 1s	97.7 %	0 / 4	-
2	6m 22s	97.6 %	2 / 4	1m 40s
3	13m 58s	97.4 %	2 / 4	2m 28s

Table 1: KPIs measured on Gazebo for the recorded simulations

7 Results and Conclusions

After several weeks of work, study and development, the project has been tested both in Gazebo simulation environment and in the University laboratory with the real robots. One of the main learning outcomes from this experience is the actual difficulty to deploy a functioning application on real hardware. Dealing with real robots exposes issues which are not considered when testing on the simulation environment, such as communication lag and delay on the ROS channels, compatibility issues with drivers, errors in measurements, networking configuration between ROS nodes, and electronics anomalies with batteries which lead to inaccurate data retrieved from sensors.

Nevertheless, in the provided lab hours, the group managed to command the UR5 robot to grasp a block and move it to a defined position, command Shelfino to perform controlled trajectories, and connect to the ZED and RealSense cameras to test YOLO detection algorithm.

7.1 Simulation Performance

Various different tests were performed on Gazebo simulation environment, during the development to check the progress and make improvements, and in the days before the exam to get the final results and measure the requested KPIs. Those measures are reported in table 1 and refer to specific tests which are recorded and available online (accessible from the GitHub repository).

Completion time and localization accuracy: The total execution time is dominated by the movement of Shelfino, which was intentionally very slow to privilege accuracy (which is indeed very high) over completion time. Shelfino may be easily sped up by increasing linear and angular velocities, but this operation would cause slightly lower localization accuracy and some difficulties to perfectly reach the parking position. The excellent results in localization accuracy in the third assignment (which requires a lot of movements) confirm the effectiveness of the implemented motion control using Lyapunov function as described in section 5.3.

Classification errors: The objects on the map are always detected by YOLO, but the classification accuracy is a bit disappointing when the object is not placed on its natural pose. These kind of errors are actually limited to the simulation environment, because the majority of the training focused on the detection of real objects, and the excellent results are described in section 6.

7.2 Possible Improvements

Many different features, improvements, adaptations, tests and experiments have been left for the future due to lack of time and difficulties that may raise with the beginning of new developments, i.e. tests in the laboratory resulted to be quite time consuming because many issues had to be fixed on the real robots before effectively trying to complete the assignments.

There are many ideas which could be implemented to improve the overall functioning of the project. This experience has been mainly focused on the basic understanding of the learned technologies and methodologies, therefore the results are not optimal. The following ideas may be discussed:

1. UR5 trajectories are very simple. The end effector goes through the home-load-home-unload positions in a discontinuous movement. The motion planning would improve [11] by considering the home position as a via point and imposing the passage with constant velocity in the equation 1.
2. Shelfino movements may be improved implementing Dubins paths [16] and publishing them on the path topic, instead of controlling velocities. Also other sensors may be used, like the Lidar to construct a map of the surrounding environment and thus improving localization. In this case, the point cloud from the real sense camera may be used instead of the depth map, to localize objects more precisely.
3. The overall code may be cleaned and refactored, because the fast continuous integration and development probably created some inefficient and hard-to-maintain functions. Different structures may also be tested to find the best solution, for instance by combining the controller nodes in a single Catkin Package which operates as single end point for the communication with the Main Controller. A lot of hardcoded constants may be parametrized to be easily modified in launch files.

References

[1] Intel Corporation. *Intel Real Sense*. URL: https://www.intelrealsense.com/. (accessed: 08.01.2023).

[2] Universal Robots. *UR5 Robot*. URL: https://www.universal-robots.com/products/ur5-robot/. (accessed: 08.01.2023).

[3] StereoLabs. *ZED 2 Camera*. URL: https://www.stereolabs.com/zed-2/. (accessed: 08.01.2023).

[4] ROS.org. *ROS Middleware*. URL: https://www.ros.org/. (accessed: 08.01.2023).

[5] Chris Richardson. *Microservice Architecture*. URL: https://microservices.io/patterns/microservices.html. (accessed: 08.01.2023).

[6] Ultralytics. *YOLOv5*. URL: https://docs.ultralytics.com/. (accessed: 08.01.2023).

[7] Wikipedia. *Finite State Machine*. URL: https://en.wikipedia.org/wiki/Finite-state_machine. (accessed: 08.01.2023).

[8] Wikipedia. *Direct (or Forward) Kinematics*. URL: https://en.wikipedia.org/wiki/Forward_kinematics. (accessed: 08.01.2023).

[9] Wikipedia. *Inverse Kinematics*. URL: https://en.wikipedia.org/wiki/Inverse_kinematics. (accessed: 08.01.2023).

[10] L. Palopoli. *(Slides) Direct and Inverse Kinematics*. Course: Fundamentals of Robotics, 2022.

[11] L. Palopoli. *(Slides) Advanced Topics on Manipulation*. Course: Fundamentals of Robotics, 2022.

[12] L. Palopoli. *(Slides) Mobile Robots*. Course: Fundamentals of Robotics, 2022.

[13] L. Palopoli. *(Slides) Trajectory Control*. Course: Fundamentals of Robotics, 2021.

[14] Wikipedia. *Lyapunov Function*. URL: https://en.wikipedia.org/wiki/Lyapunov_function. (accessed: 08.01.2023).

[15] Heartexlabs. *label-studio*. URL: https://labelstud.io/. (accessed: 08.01.2023).

[16] L. Palopoli. *(Slides) Dubins Manoeuvres*. Course: Fundamentals of Robotics, 2022.