POLITECNICO DI TORINO

Master Degree course in Mechatronics Engineering

Master Degree Thesis

Control of an inflatable robot for space application

Supervisors

Prof. Stefano Mauro

Matteo Gaidano

Pierpaolo Palmeri

Candidate

Giulia Calvo

Academic Year 2022-2023

Acknowledgements

I would like to thank Prof. Stefano Mauro for making me a part of this ambitious project and for having assisted me, showing great helpfulness and professionality.
I would also like to thank my supervisors, Pierpaolo Palmieri and Matteo Gaidano, for their suggestions, precious advice and support over these months.
Lastly, my deepest gratitude go to my family, whose unconditional encouragement makes me feel lucky every day.

Sincerly,
Giulia

Abstract

In recent years, the collaboration between humans and robots has become essential, revealing to be crucial in space environment exploration. The presence of robots allows to perform dangerous tasks and to operate in extreme conditions. The interest in inflatable and deployable systems for space missions represents a growing trend in recent years. Soft robots offer higher dexterity, larger variability of movements, better usability in otherwise inaccessible places and higher safety. In several space applications, the use of inflatable structures could minimize bulk and mass, reducing space mission costs.

The object of study of this thesis work is the POPUP robot, a manipulator with inflatable links, developed for space application, whose first functional prototype has been developed in the laboratories of Polytechnic of Turin . It consists of two inflatable links, made of fibers with high elastic module, electric motors and a gripper. By introducing soft parts, the robot can adapt to various contexts by overcoming the limitations related to the rigid structure of traditional ones.

The aim of this thesis is first to simulate the prototype of the POPUP robot on a virtual environment. Gazebo is used as a 3D simulator and ROS 1.0 as a robotic interface. Different plugins that the gazebo environment offers are also used, including in particular one to simulate a camera to be placed on the end-effector of the robotic arm, and another to control the robot in the virtual environment. These two tools will enable the implementation of a possible visual servoing control algorithm in the virtual environment for future developments.

One of the main challenges in soft robotics is the control of the system. It requires more adaptive and flexible control algorithms that can adjust to the unpredictable nature of the robot's deformable structure. For this reason, the second part of the thesis focuses to propose a visual servoing control algorithm. It is based on the the information received from the frames captured by a camera placed on the robot's end-effector. The integration of the vision system will enable precise

and accurate control even in the case of inflatable links. This control technique can help to accurately position the soft manipulator's end-effector by using visual feedback to track the position of a target object or feature. The soft manipulator's deformations can be taken into account by the vision system to ensure that the end-effector is correctly positioned

Contents

List of Figures

Chapter 1

Introduction

Humans have always desired to explore and challenge the boundaries of what is already known.
Space robots deeply marked the space exploring age's beginning and evolution and their use is now essential.
They replace humans in conducting scientific experiments, external vehicular activities, space exploration, and other space activities [1].
One example is the Space Station Remote Manipulator System (SSRMS), better known as Canadarm2.

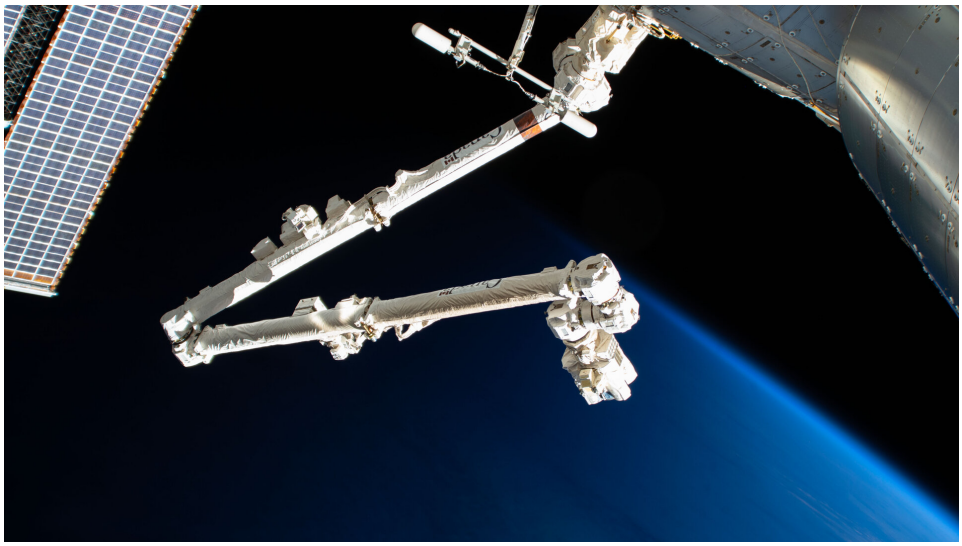


Figure 1.1. Canadarm2 robotic arm mounted on International Space Station

It is a robotic arm that lends a helping hand to: perform Station

maintenance, move supplies, equipment and even astronauts, perform "cosmic catches" by grappling visiting vehicles and berthing them to the International Space Station (ISS) [2].

The development of space robots presents many challenges starting from their design, as well as their fabrication and control, since these devices will operate in space where the environment differs greatly from Earth.
For example, the rigid structure of traditional robots is heavy and requires a high payload capacity when embarked on spacecraft.
In order to overcome the problem of weight, and thus to have lighter robots, soft manipulators (i.e. robots made of soft materials) have already been developed in the terrestrial field.
Their adaptability and low mass would allow being easier transported from the Earth, safety, and storage in relatively small packages .
For these observations, soft robotics can meet and resolve aerospace issues, developing inflatable deployable robotic manipulators, based on soft materials bodies [3].
However, they may face important challenges in their performance due to low structural stiffness. Accurate positioning of the end effector may prove a difficult task due to limitations in gravity compensation, or due to the occurrence of oscillations during motion, especially in the absence of precise information about the payload that is being manipulated [4].
Due to all of these factors, the control of a robot with inflatable links is a task not easy to perform.
Usually, the control algorithms for rigid body robot manipulators are based on precise mathematical models [5].
These models describe the kinematics (i.e., the relationship between the robot's joint angles and its end effector position and orientation) and dynamics (i.e., how forces and torques affect the motion of the robot) of the robot.
In contrast, a soft robot manipulator, requires more adaptive and flexible control algorithms that can adjust to the unpredictable nature of the robot's deformable structure.

These control algorithms may not be as reliant on precise mathematical models, but instead use sensory feedback to adjust the robot's motion in real-time [6].
Therefore, one approach is to use an on-board vision system to improve the control strategy and increase accuracy.
Visual servoing is a well-known tecnique to guide robots using visual information. Image processing, robotics and control theory are combined in order to control the motion of a robot depending on the visual information extracted from the images captures by one or several cameras [7].
This control technique can help to accurately position the soft manipulator's end effector by using visual feedback to track the position of a target object or feature. The soft manipulator's deformations can be taken into account by the vision system to ensure that the end effector is correctly positioned, even as the soft manipulator undergoes large deformations.

The object of study of this thesis work is the POPUP robot, a manipulator with inflatable links, developed for space applications, whose first functional prototype has been developed in the laboratories of Polytechnic of Turin .

It is a deployable robot with hybrid structure consisting of two inflatable links, three electric motors and rigid joints.
The first prototype presents inflatable links with cylindrical shape and made out PVC fixed to a 3D printed support which connect the link to the actuator. The links to be fixed through screws to the other joints, allowing the possibility to add elements, e.g. sensors, inside the links during development stage [8]. A pneumatic line is responsible to control the inflation and deflation stage. It allows the links to be inflated and deflated, providing the necessary pressure which is in the range of 10-60 kPa.
For the implementation of the project, a camera will be added to the robot's end-effector.
For future applications, a wrist will also be integrated, extending the

degrees of freedom from 3 to 6.

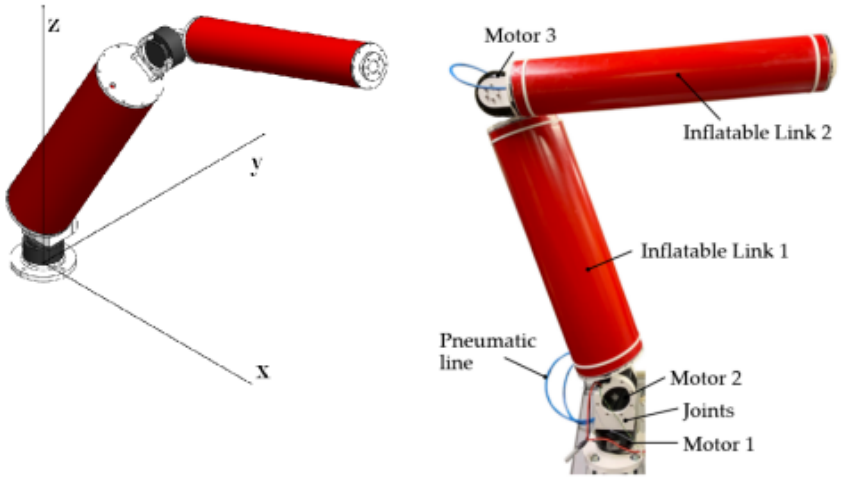


Figure 1.2. POPUP robot

The aim of the project is first to simulate the prototype on a virtual environment. To obtain realistic simulations of the robotic scenario, Gazebo is used as a 3D simulator and ROS 1.0 as a robotic interface.

Then, a control algorithm is developed using the information received from the frames captured by a camera placed on the robot's end-effector. Also in this case, the use of ROS 1.0 allows communication between several software and hardware tools employed in this robotics application.

Chapter 2

Software Tools

2.1 ROS 1.0

ROS (Robot Operating System) is a dynamic and versatile framework that offers a diverse set of tools and libraries for building robot software. Its main goal is to create a unified approach to programming robots, while also providing ready-to-use software components that can be seamlessly integrated into custom robotic applications [9].

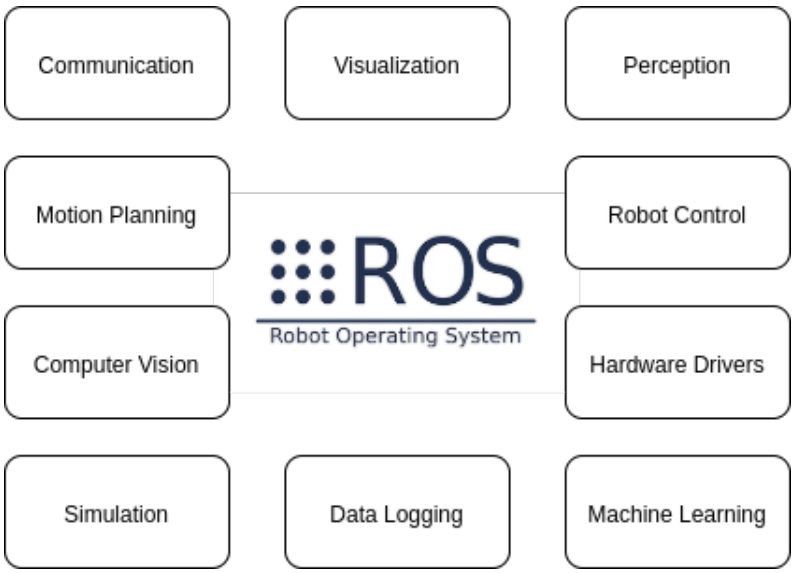


Figure 2.1. ROS 1.0

The advantages of using ROS as a programming framework are numerous, and some of them include:

- variety of advanced capabilities that can be used to build complex robot applications
- several tools for debugging, visualizing, and having a simulation
- support for a variety of high-end sensors and actuators commonly used in robotics. This makes it easy to incorporate these critical components into robot software without encountering any obstacles
- inter-platform operability: The ROS message-passing middleware allows communication between different programs.

The ROS architecture has been designed and divided into three sections or levels of concepts:

- The Filesystem level
- The Computation Graph level
- The Community level

a more detailed analysis of all levels is presented below

2.1.1 ROS Filesystem level

The main goal of the ROS Filesystem is to centralize the build process of a project, while at the same time provide enough flexibility and tooling to decentralize its dependencies.

Similar to an operating system, an ROS program is divided into folders, and these folders have files that describe their functionalities.In fact, ROS files are structured on the hard disk in a specific way, similar to how an operating system organizes files.
The following diagram illustrates this structure

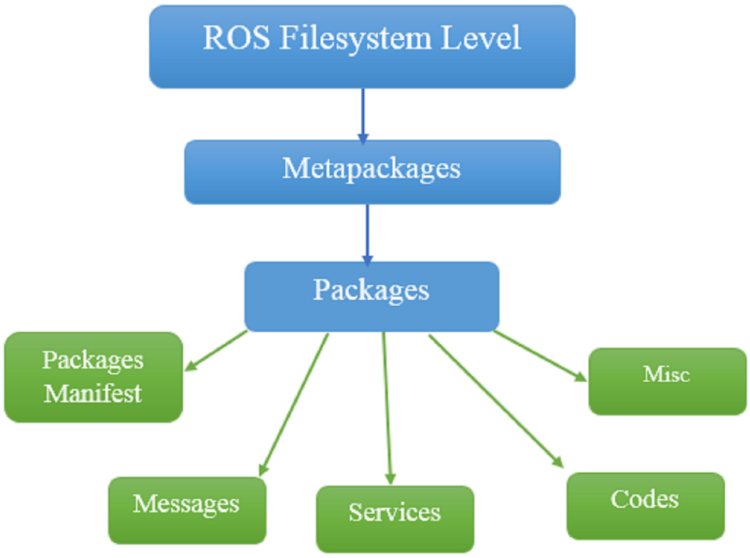


Figure 2.2. ROS filesystem level

- **Metapackages**: The term metapackage refers to one or more related packages which can be loosely grouped together. In principle, metapackages are virtual packages that don't contain any source code or typical files usually found in packages.
- **Packages**: The ROS packages are the most basic unit of the ROS software. They contain one or more ROS programs (nodes), libraries, configuration files, and so on, which are organized together as a single unit. Packages are the atomic build item and release item in the ROS software.

- **Package manifest**: The package manifest file is inside a package that contains information about the package, author, license, dependencies, compilation flags, and so on. The package.xml file inside the ROS package is the manifest file of that package.
- **Metapackages manifest**: The metapackage manifest is similar to the package manifest, the difference being that it might include packages inside it as runtime dependencies and declare an export tag.
- **Messages**: The ROS messages are a type of information that is sent from one ROS process to the other. The extension of the message file is .msg.
- **Services (.srv)**: The ROS service is a kind of request/reply interaction between processes. The reply and request data types can be defined inside the srv folder.
- **Repositories**: Most of the ROS packages are maintained using a Version Control System (VCS), such as Git, Subversion (svn), Mercurial (hg), and so on. The collection of packages that share a common VCS can be called repositories. The package in the repositories can be released using a catkin release automation tool called bloom.

The configuration of files and folders can be summarised by the following scheme :

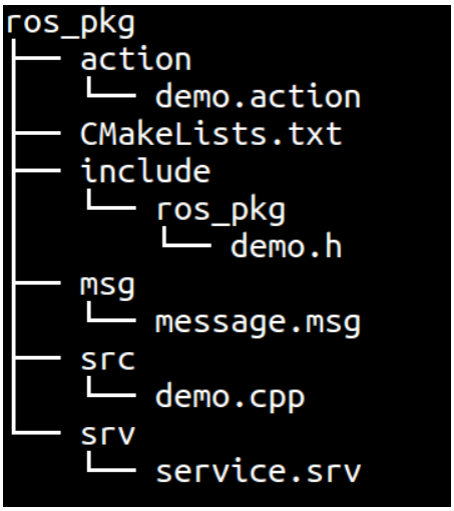


Figure 2.3. Organisation of files and folder

ROS Packages

ROS packages are the basic units of ROS programs.
The workspace is the folder containing all packages. In this project it will be called "*catkin_ws*".
This environment allows several packages to be compiled simultaneously and is a good way to centralise all developments . ROS packages tend to follow a common structure. The directories and files found in this work are as follows:

- **config**: all configuration files that are used in this ROS package are kept in this folder. This folder is created by the user and it is a common practice to name the folder config as this is where we keep the configuration files
- **include/package_name**: This directory includes the headers of the libraries that you would need.
- **launch**: This folder keeps the launch files that are used to launch one or more ROS nodes.

- **scripts**: These are executable scripts that can be in Bash, Python, or any other scripting language.
- **src**: This is where the source files of your programs are present. You can create a folder for nodes and nodelets or organize it as you want.

2.1.2 ROS Computation Graph level

Computation in ROS is done using a network of ROS nodes. This computation network is called the computation graph. The main concepts in the computation graph are ROS nodes, master, parameter server, messages, topics, services, and bags. Each concept in the graph is contributed to this graph in different ways.

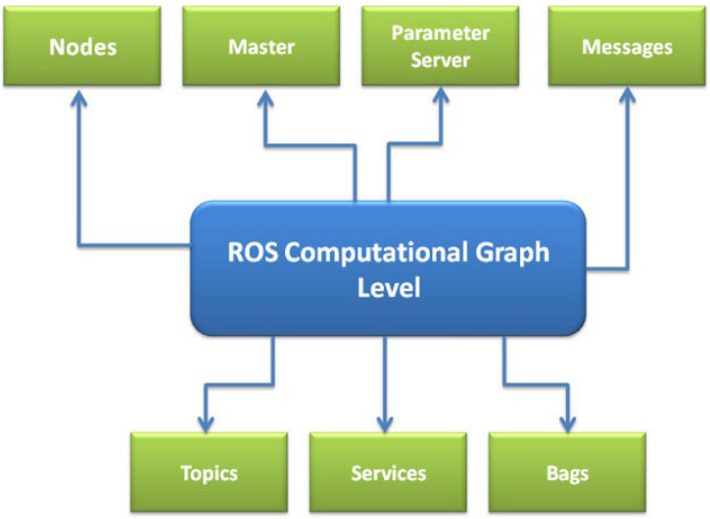


Figure 2.4. ROS Computation Graph level

- **Nodes**
  In ROS, computation is carried out through small units called nodes. Instead of writing a large and complex program, ROS divides it into smaller units or nodes. This approach of breaking down the program into smaller modules allows for better modularity and flexibility, making it easier to develop and maintain complex robotic systems. By dividing the program into smaller nodes, each node can focus on a specific task or function, such as sensing, perception, motion planning, or control. These nodes can then communicate and exchange data with each other, allowing for a distributed system that can work collaboratively to accomplish complex tasks.

- **Master**
  The ROS master plays a crucial role in the coordination of the various nodes that make up the system. It provides name registration and lookup processes for all other nodes, which means that without a ROS master, nodes would not be able to find each other, exchange messages, or invoke services. The ROS master is responsible for keeping track of all the available nodes in the system, their names, and the topics they publish and subscribe to.

- **Parameter server**
  It allows to store data. These values are accessible and modifiable by all nodes. The ROS master includes the parameter server

- **Topic**
  A topic acts as a channel through which nodes can exchange messages with each other. When a node sends a message to another node, it is said to be publishing to a topic. On the other hand, when a node receives messages from a topic, it is said to be subscribing to that topic. Each topic has a unique name that identifies it and enables nodes to locate and communicate with it. Topics are an essential component of the ROS system and are used extensively to share information between nodes.

- **Logging**
  ROS provides a logging system for storing data, such as sensor data, which can be difficult to collect but is necessary for developing and testing robot algorithms.

Publisher/Subscriber

The Publisher Subscriber Interface is a fundamental feature of the ROS library, and it provides a way for nodes to communicate with each other by passing messages. In ROS, a node can act as both a subscriber and a publisher, enabling it to both receive and send messages. When a node publishes a message, it sends it to a particular topic that has a unique name. On the other hand, a subscriber subscribes to a specific topic, and whenever a message is posted to that topic, the subscriber gets notified. A publisher can publish to multiple topics, and a subscriber can receive messages from multiple topics. The ROS master is responsible for keeping track of all the nodes' IP addresses, which enables them to communicate with each other effectively. The key objective of the Publisher Subscriber Interface is to separate the creation and consumption of information, which enhances modularity and flexibility in the development of robotic systems. Neither the publisher nor the subscriber knows about each other's existence, and the ROS master facilitates the communication between them.

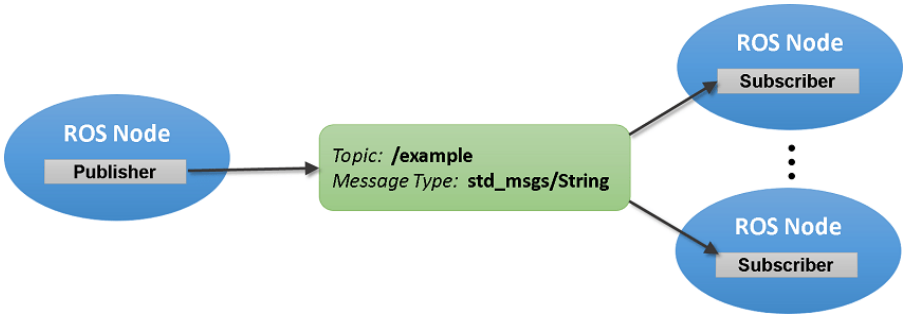


Figure 2.5. Publisher Subscriber ROS

2.1.3 Community level

The third level is the Community level, which comprises a set of tools and concepts to share knowledge, algorithms, and code between developers. This level is of great importance; as with most open source software projects, having a strong community not only improves the

ability of newcomers to understand the intricacies of the software, as well as solve the most common issues, it is also the main force driving its growth.

2.2 Gazebo

Gazebo is a 3D simulation software that is widely used in the field of robotics for modeling and testing robots in both indoor and outdoor environments. It is characterized by its advanced physics simulation, rendering engine, and sensor interfaces, which provide high accuracy and a high level of detail.

The physics simulation in Gazebo provides a more detailed and realistic representation of the physical world compared to other simulators. It is based on a physics engine that simulates a range of physical properties, such as gravity, friction, and collisions. This enables Gazebo to accurately model the behavior of robots and their interactions with their environments.

In addition to its simulation capabilities, Gazebo includes a collection of sensors and interfaces that enable users to interact with their robots and control their behavior. These sensors include cameras, lidars, and sonars, and can be used to collect data from the environment and feed it into the robot's control system.

Gazebo's flexibility and versatility are further enhanced by its integration with ROS thanks to a proper ROS interface, which exposes the complete control of Gazebo in ROS. It is precisely because of all these characteristics that it was chosen to simulate POP-UP control.

Chapter 3

Robot model in virtual environment

3.1 Robot description using URDF

ROS offers several useful packages for constructing 3D robot models, which can be created using the URDF (Unified Robot Description Format) file.
URDF is capable of representing both the kinematic and dynamic properties of a robot, as well as its visual and collision models.
To construct a URDF robot model, special XML tags are employed. The following tags are some of the most commonly used components of a URDF file [10]:

- **Link**:
  The link tag represents a single link of a robot in a robot description file. This tag is used to model a robot link and its properties including the size, shape, and color of the link. Furthermore, the tag can also import a 3D mesh to accurately represent the robot link in a virtual environment [10].
  A mesh is a collection of vertices, edges, and faces that define the shape and surface of an object in a three-dimensional space [11]. By importing a 3D mesh, it is possible to represent complex and irregular shapes of robot links with a high degree of accuracy.
  In addition to visual properties, the link tag can also define the dynamic properties of the link, such as the inertial matrix and the collision properties.

The inertial matrix defines the link's mass, center of mass, and moment of inertia. These properties are essential for calculating the link's motion and forces during a simulation.
Collision properties define the behavior of the link when it comes into contact with other objects in the simulation environment. These properties can include the shape of the collision volume and the material properties of the link's surface, such as friction and elasticity. By defining collision properties for each link, a more accurate simulation of the robot's behaviour in the real world can be created.
The syntax of the link tag is as follows:

```
<link name="<name of the link>">
    <inertial >..... </inertial>
    <visual> ..... </visual>
    <collision >..... </collision>
</link>
```

Displayed below is an illustration of an individual link, consisting of two sections: the Visual section, which depicts the physical appearance of the robot's link, and the Collision section, which encompasses the Visual section to detect potential collisions before they occur.

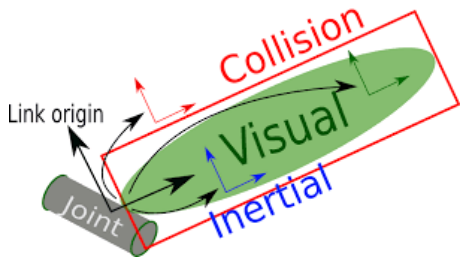


Figure 3.1. A visualization of the URDF link

- **Joint**
  The joint tag represents a robot joint that connects two links of a robot model. This tag provides information about the type of joint, its axis of rotation or translation, and its position relative to the links it connects [10].

The <joint> tag has several elements that can be used to describe different aspects of the joint, as follows:

- *name*: The name of the joint.
- *type*: The type of joint. The possible values for this attribute are revolute, continuous, prismatic, fixed, floating, or planar, depending on the type of motion that the joint allows.
- *parent*: The name of the link that the joint connects to the parent link of the joint.
- *child*: The name of the link that the joint connects to the child link of the joint.
- *origin*: The position and orientation of the joint relative to the parent link. This element includes the xyz attributes that specify the position of the joint in meters, and the rpy attributes that specify the rotation of the joint in radians.
- *axis*: The axis of rotation or translation of the joint. This element includes the xyz attributes that specify the direction of the axis in meters, and can only be used for revolute and prismatic joints.
- *limit*: The limits of motion for the joint. This element includes the lower and upper attributes that specify the lower and upper limits of the joint motion in radians or meters, respectively. It also includes the effort and velocity attributes that specify the maximum effort and velocity that the joint can exert

The syntax is as follows:

```
<joint name="<name of the joint>">
     <parent link="link1"/>
     <child link="link2"/>
     <calibration .... />
     <dynamics damping ..../>
     <limit effort .... />
</joint>
```

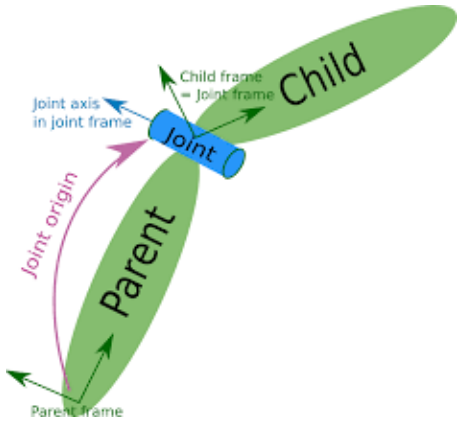


Figure 3.2. A visualization of the URDF joint

- **Gazebo**
  This tag can be used to include Gazebo-specific parameters that are not included in the standard URDF format [10]. For example, it can be used to include Gazebo plugins that provide additional functionality, such as physics engines, sensors, or controllers.
  In this work, the tag is used to add a camera sensor on the end-effector of the manipulator robot. These plugins can be used to simulate the behavior of the robot in a more accurate and realistic way.
  In addition, the <gazebo> tag can also be used to define Gazebo-specific material properties for the robot model. This can include the visual and collision properties of the links, such as the color, texture, and friction of the materials.
  An example of the syntax is as follows:

```
<gazebo reference="link_1">
    <material>Gazebo/Black</material>
</gazebo>
```

- **Robot**:
  This tag encapsulates the entire robot model that can be represented using URDF [10]. Inside the robot tag, we can define all the other tags that identify name of the robot, links, joints etc..

The sintax is as follow:

```
<robot name="<name of the robot>"
    <link> ..... </link>
    <link> ..... </link>
    <joint> ..... </joint>
    <joint> ..... </joint>
</robot>
```

3.2 URDF Pop-up model

Pop-up robot virtual model is a six-degrees of freedom (DOF) serial robot with a prismatic joint called "dummy joint" that fixs the base to the world and three revolute joints.
The graphical structure of links and joints is depicted in the figure below:

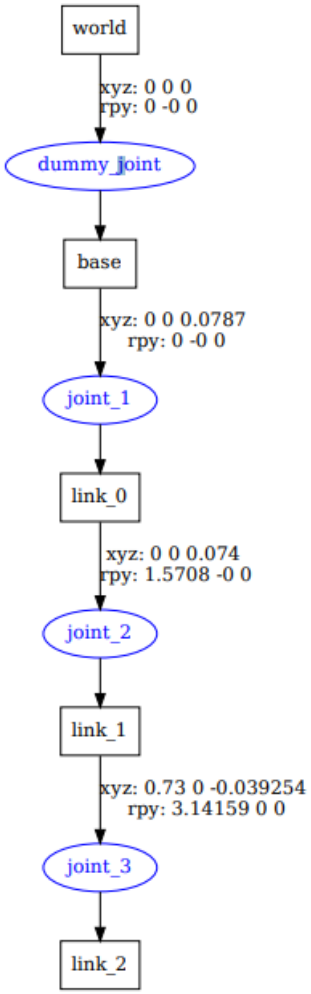


Figure 3.3. Graph of joint and links in POPUP

In order to realise the URDF model of the Popup robot, the popup_3dof.xacro file is create. First the XML version and encoding format used in the file are specified. In this case is used the 1.0 xml version.
Then, the robot name is defined.
To ensure that the robot remains rigidly attached to the ground, a link, that has no information about the geometry, mass, or inertial properties is created.

```
<?xml version="1.0" encoding="utf-8"?>
<robot name="popup_3dof">
  <link name="world">
  </link>
```

Next, the links and joints forming the structure of the manipulator robot are set up.
The links were previously modelled with *Solidworks software* and imported into the URDF file using the *mesh* tags. The mesh tag is used to define the visual and collision properties of a 3D object using a mesh file format. The mesh file format is in STL format.

The first link described is the base link. In this regard, it has been connected to the world via a fixed-type joint.

```
<link name="base">

    <visual>
    <origin
    xyz="0 0 0"
    rpy="0 0 0" />
    <geometry>
       <mesh filename="package://popup_3dof/meshes/Base.STL" />
    </geometry>
    <material name="">
        <color
        rgba="0.752941176470588 0.752941176470588 0.752941176470588 1" />
    </material>
    </visual>

    <collision>
    <origin
    xyz="0 0 0"
    rpy="0 0 0" />
    <geometry>
    <mesh filename="package://popup_3dof/meshes/Base.STL" />
    </geometry>
    </collision>

    <inertial>
    <origin
        xyz="-0.000211223053684081 -2.4904738873446E-05 0.0418999516779425"
        rpy="0 0 0" />
    <mass value="10" />
    <inertia
        ixx="0.000932883813436538"
        ixy="1.48232249872527E-06"
        ixz="9.90380531170409E-07"
        iyy="0.000928773050905896"
        iyz="2.22987138205791E-06"
        izz="0.00144668291013681" />
    </inertial>
</link>

<joint name="dummy_joint" type="fixed">
    <parent link="world"/>
    <child link="base"/>
</joint>
```

In the same way, the other three links and joints that constitute the robot arm are defined.
The three links are labelled: *link_0*, *link_1* and *link_2*.
As it has been done previously, geometry, mass and inertias are also specified for these links.
The three joints are labelled: *joint_1*, *joint_2* and *joint_3*.
Joint_1 is a revolute joint with an axis of rotation along the z-axis (perpendicular to the x-y plane). It connects the base of the robot to the first link (link_0) and is located at a distance of 0.0787 meters from the base.
Joint_ 2 is a revolute joint that connects link_0 to link_1. It has an axis of rotation along the z-axis, located at a distance of 0.074 meters from the joint_1 along the z-axis.
Joint_ 3 is a revolute joint that connects link_1 to link_2. It has an axis of rotation along the z-axis, located at a distance of 0.73 meters from joint_2 along the x-axis and -0.039254 meters along the z-axis.
physical limits within which each joint can rotate are also defined. An analysis of the mechanical characteristics then also set limits on the speed and effort they can support.
The table below shows all the characteristics of each individual joint:

name	**type**	**joint limit**	**velocity limit**	**effort limit**
dummy_joint	fixed			
joint_1	revolute	0 to 2π	5 m/s	10 Nm
joint_2	revolute	-π to π	5 m/s	10 Nm
joint_3	revolute	0 to 2π	5 m/s	10 Nm

3.2.1 Visualising Popup model

After creating the URDF file defining the robot popup, a launch file to visualise the model within the Gazebo environment is generated.
As can be seen from the code below, the file called *popup_3dof.launch* opens a launch tag and includes another launch file: *empty_world.launch*, from the gazebo_ros package, which is used to launch the Gazebo simulation environment with an empty world.

```
<?xml version="1.0" ?>

  <include file="$(find gazebo_ros)/launch/empty_world.launch">
    <arg name="debug" value="$(arg debug)" />
    <arg name="gui" value="$(arg gui)" />
    <arg name="paused" value="$(arg paused)"/>
    <arg name="use_sim_time" value="$(arg use_sim_time)"/>
    <arg name="headless" value="$(arg headless)"/>
  </include>
```

Then, it sets a parameter on the ROS parameter server called *robot_description* using an xacro file located at popup_3dof/urdf/popup_3dof.xacro.
Xacro is an XML macro language that is often used to generate URDF files for robots in ROS. The xacro command is used to convert the xacro file to URDF format and put it on the parameter server. The resulting output of this command is a complete URDF model of the robot described in the input file, which can be used by other ROS nodes and tools to visualize, control, or interact with the robot.

```
 <param name="robot_description" command="$(find xacro)/xacro
 --inorder '$(find popup_3dof)/urdf/popup_3dof.xacro'" />
```

Subsequently, it launches a gazebo_ros node called urdf_spawner, which spawns a URDF robot model in Gazebo using the spawn_model service. It specifies the name of the model as popup_3dof, and the robot_description parameter is used as the URDF description.
After all it launches a gazebo_ros node called urdf_spawner, which spawns a URDF robot model in Gazebo using the spawn_model service. It specifies the name of the model as popup_3dof, and the

robot_description parameter is used as the URDF description.

```
 <node name="urdf_spawner" pkg="gazebo_ros"
 type="spawn_model" respawn="false" output="screen"
   args="-urdf -model popup_3dof
  -param robot_description"/>
```

The model is launched using the following command in the terminal:

```
roslaunch popup_3dof popup_3dof.launch
```

this will result in the model of the robot in the virtual gazebo environment, as shown in the figure 3.4:

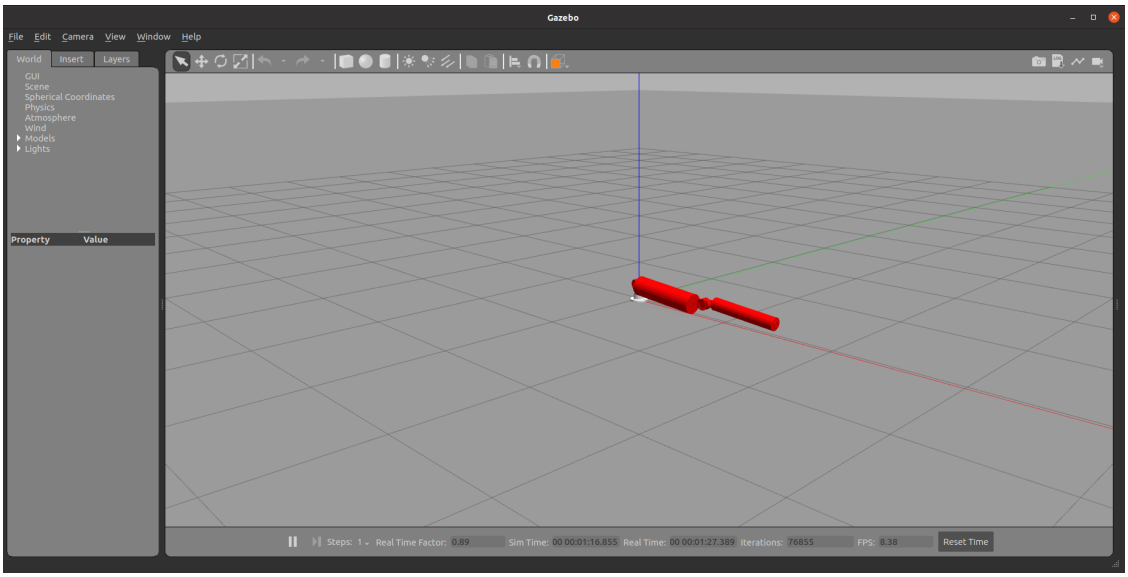


Figure 3.4. Popup model on Gazebo environment

3.3 Camera sensor

Once the robot model has been created, it was decided to place a camera on the end-effector. The camera will indeed be useful for future tests of the control algorithms. In gazebo, in addition to being able to simulate the robot's movements, it is possible to simulate a number of sensors including different types of cameras [10].
In this case, the camera was represented as a white block, adding another link and joind to the URDF representing the robot.
A new .xacro file is then created with the description of the camera.

In this file, the necessary camera links are defined: the 'camera_link' link representing the camera body, and the 'camera_link_optical' link representing the optical part of the camera. Two joints are defined. The first joint, "camera_joint", defines the position of the camera with respect to the robot, while the second joint, "camera_optical_joint", defines the position of the camera's optical element with respect to its structure. the geometry of the link is defined as a parallelepiped with dimensions of 0.008 m x 0.02 m x 0.07 m. The material used to display the link is also specified, namely 'darkgray', which is defined using the <material> tag.

To control and simulate the behaviour of the camera within the virtual environment, the plugin: "libgazebo_ros_camera.so".
The plugin receives data from the camera (such as the captured image) and publishes it on two ROS topics ("image_raw" and "camera_info") for other ROS nodes to process.
The plugin also defines some camera-specific settings, such as:

- The camera name ("rrbot/camera1") and the camera reference frame ("camera_link")
- The <updateRate> tag which specifies the plugin's update frequency. In this case, it is set to 0.0, which means that the plugin is updated at the same frequency as the camera (30Hz).

- The tags <distortionK1>, <distortionK2>, <distortionK3>, <distortionT1> and <distortionT2> specify the distortion parameters of the chamber, which are set to zero in this case, meaning that the chamber is distortion-free.

The model of the POPUP with the camera placed on the end effector is shown in the figure 3.5:

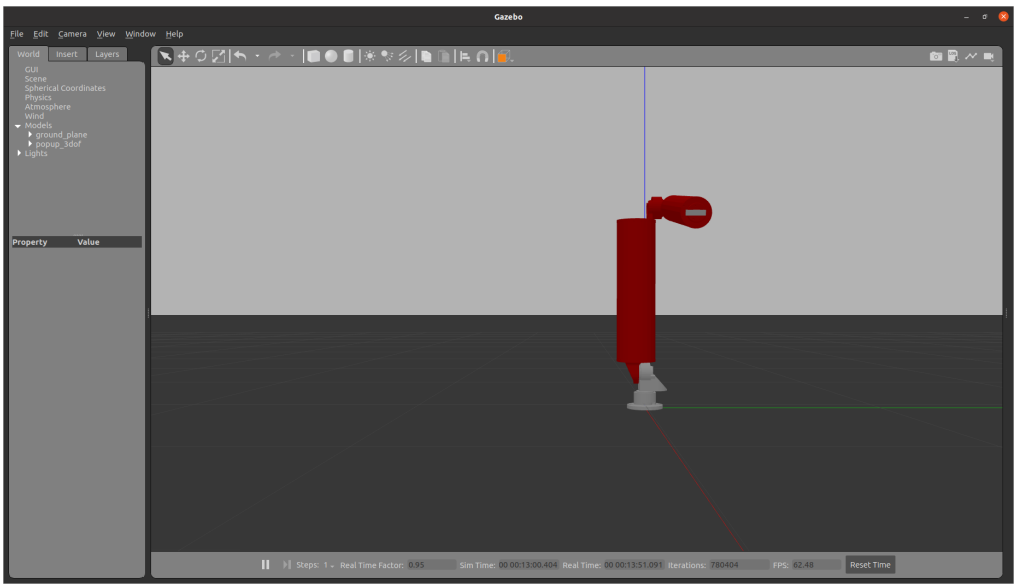


Figure 3.5. POPUP robot with camera sensor

To display the camera frame, a black sphere has been placed in front of the robot.
In the figure 3.6 it can be observed the disposition of the robot and the sphere in the virtual environment, while figure 3.7 shows the camera frame.

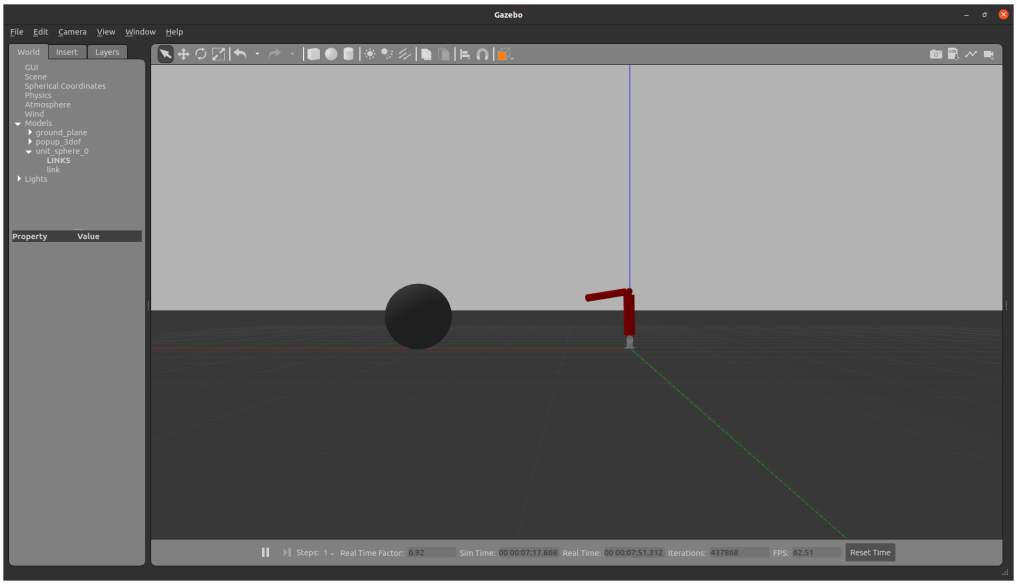


Figure 3.6. Placement of an object in front of the robot

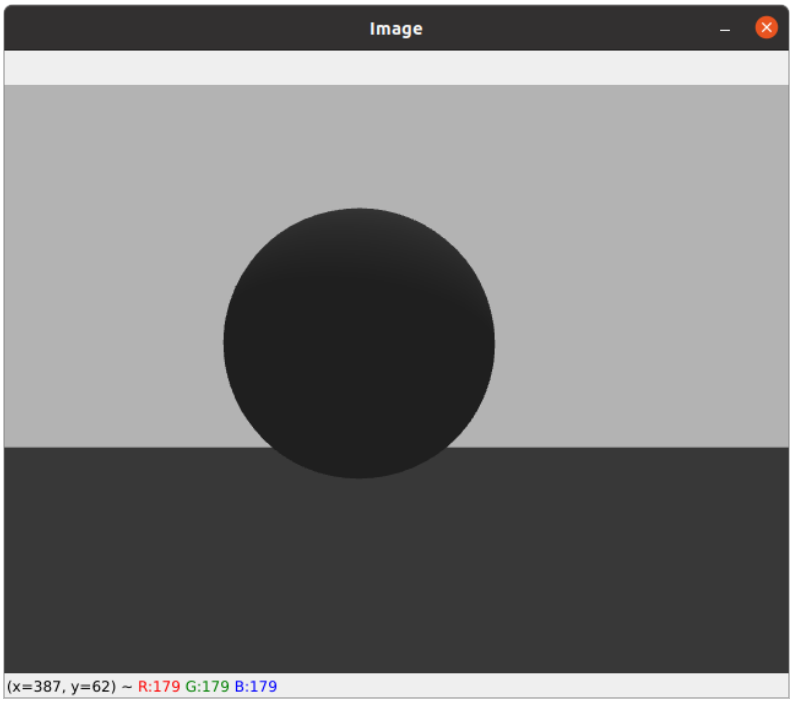


Figure 3.7. Image view

3.4 ROS control

Simulating a robot's controllers in Gazebo involves the use of the *ros_control* package and a Gazebo plugin adapter, which is a software component that enables communication between *ros_control* and the Gazebo simulator. The *ros_control* package provides a hardware abstraction layer for robot control, allowing the robot to be controlled through a set of hardware interfaces such as position controllers, velocity controllers, or effort controllers [12].
To use *ros_control* for robot control, additional elements must be added to the robot's URDF file.
The <transmission> element is used to link actuators to joints:

```
<transmission name ="trans_1">
  <type>transmission_interface/SimpleTransmission</type>
  <joint name="joint_1">
      <hardwareInterface>hardware_interface/
      EffortJointInterface</hardwareInterface>
  </joint>
  <actuator name="motor_1">
      <hardwareInterface>hardware_interface/
      EffortJointInterface</hardwareInterface>
      <mechanicalReduction>1</mechanicalReduction>
  </actuator>
</transmission>
```

In this case, the <joint name=""> tag is used to link the actuators to a joint, while the <type> tag specifies the type of transmission. Currently, the only transmission type supported is transmission_interface/SimpleTransmission. Finally, the <hardwareInterface> tag is utilized to define the controller interface to load the position, velocity, or effort interfaces. Currently only effort interfaces are implemented [10].
In addition to the transmission tags, a Gazebo plugin needs to be added to thr URDF file.
Adding the block of code below to active the gazebo ros control plugin, allow us to start a list o controller manager services, which can be used to list, start,stop or switch controllers.

```
<!-- ros_control plugin -->
<gazebo>
  <plugin name="gazebo_ros_control" filename="libgazebo_ros_control.so">
    <robotNamespace>/popup_3dof</robotNamespace>
  </plugin>
</gazebo>
```

The joint states are reported from sensors through the following controller [13]:

- **joint_state_controller**: This is a controller to publish joint states. reads joint states and publishes it to the /joint_state topic of type sensor_msgs/JointState

To control the individual joint instead, there are three packages that allow acting in different control spaces (position, speed and strain). The main ROS controllers are grouped according to the commands that are passed to the hardware/simulator:

- **effort_controller**: used to send commands to an effort interface. This means that the joints controlled accept an effort command.
  - joint_effort_controller: accepts effort set values as input
  - joint_position_controller: accepts position set values as input
  - joint_velocity_controller: accepts velocity set values as input
- **position_controller**: used to send commands to a position interface. This means that the joints controlled accept an effort command.
  - joint_position_controller: this subclass accepts only position set values as input
- **velocity_controller**: used to send commands to a velocity interface. This means that the joints controlled accept an effort command.
  - joint_velocity_controller: this subclass accepts only position set values as input

Configuration file

The controller settings of Popup robot are saved in a YAML file that is loaded via the roslaunch file.
The YAML file defines the controller type, which joint the controller is controlling, and the specific values for the PID gains (proportional, integral, and derivative).
Firstly it is defined a joint state controller for the *popup_3dof* robot arm, using the *JointStateController* type from the *joint_state_controller* package. The *publish_rate parameter* sets the frequency at which joint states will be published, in this case 50 Hz.

```
# Publish all joint states ----
  joint_state_controller:
    type: joint_state_controller/JointStateController
    publish_rate: 50
```

Subsequently, a position control is set up for each joint. It uses the *JointPositionController* type from the *effort_controllers* package, this means that it receives a desired position for each joint as input and computes the appropriate torques to achieve that position.
To choose the PID values, parameter tuning has been performed using the ROS software framework : *rqt_gui*.

```
# Position Controllers ----
  joint1_controller:
    type: effort_controllers/JointPositionController
    joint: joint_1
    pid: {p: 100, i: 0.01, d: 1}
```

This type of control (Effort/Position) has been selected because it performed more accurately and precisely than the others provided by the package.
The two figures below show the robot model before in its initial configuration and after the position command has been launched.

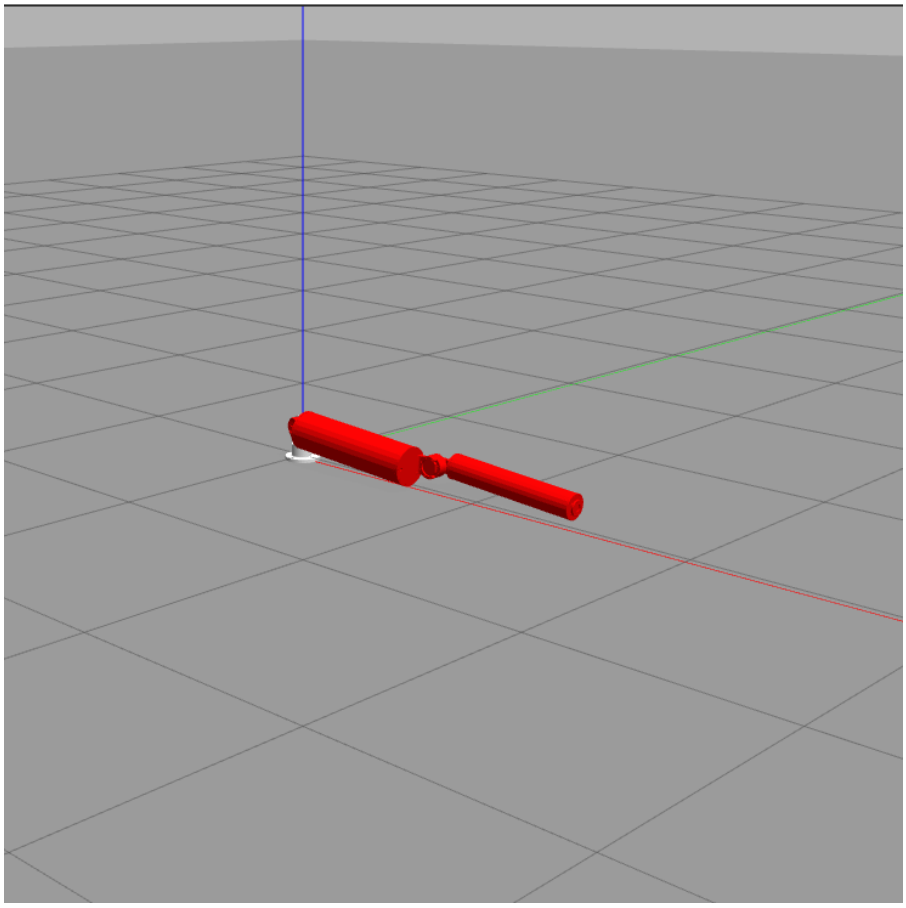


Figure 3.8. Popup initial position

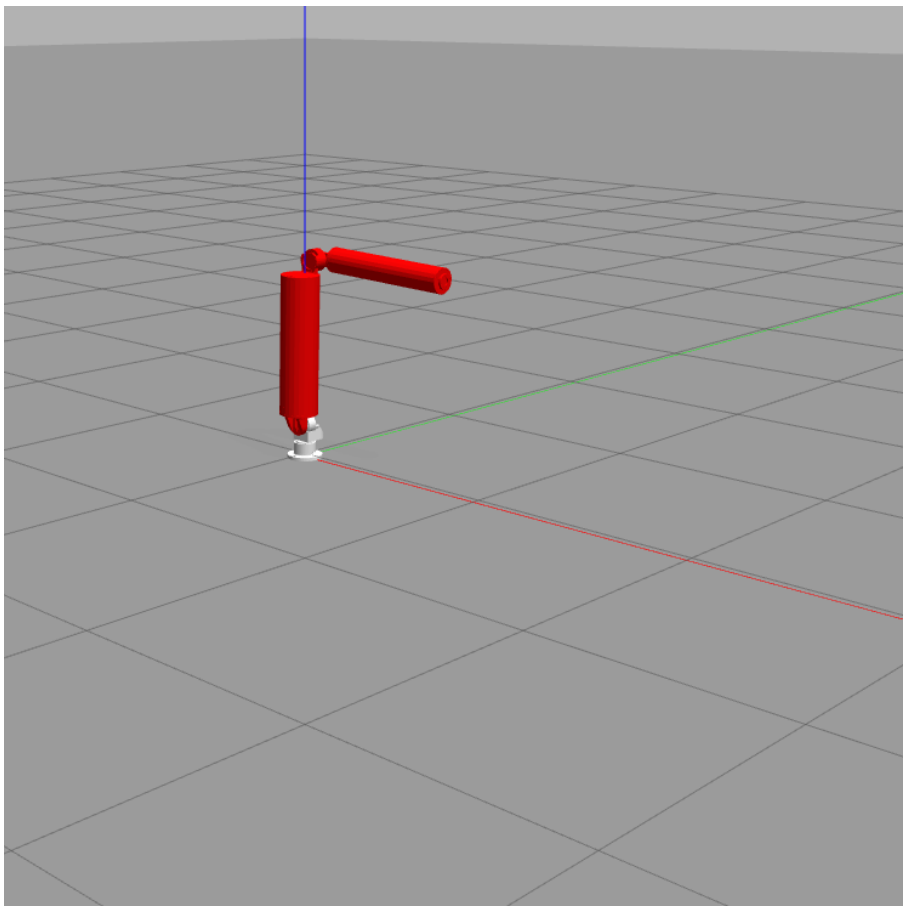


Figure 3.9. Popup final position

Chapter 4

Visual Servoing

Visual servoing is a control technique that uses visual information to control the movement of a robot or a dynamic system.
Specifically, it involves using a camera to provide feedback to a control system.

Visual servoing is divided into two main categories: position-based visual servoing (PBVS) and image-based visual servoing (IBVS).

In PBVS, the motion control is based on position information, such as the e current 3D pose (pose/orientation) of an object relative to the robot system. In this case, the motion control is obtained by adjusting the position of the system to reach the desired object position.

In IBVS, instead, the motion control is based on image information, such the features extracted on the 2D image plane, without going through a 3D reconstruction [7].

For the purpose of this work, it is chosen to implement a control based on position information (PBVS) .
The system will use information obtained from an algorithm in the OpenCV library that tracks the position and orientation of an ArUco marker in an image captured by a camera.
For future applications the marker will be replaced by an object.
OpenCv was selected since it uses high-optimised algorithms to ensure high image processing rate. The motion control is obtained by adjusting the position of the system to reduce the distance between the tool of the robot and the object to be reached.

Different types of cameras can be used for visual servoing, including 3D cameras, stereo cameras, and infrared cameras. The choice of camera depends on the specific requirements of the application, such as the required accuracy, lighting conditions, and operating environment. The camera chosen is an Intel RealSense d435.
It presents high precision, small size and weight and a wide field of view, features that make it suitable for this purpose

Intel RealSense d435

The D435 is a powerful and compact stereo camera that belongs to the D400 family. It is capable of streaming both RGB color data and depth information, thanks to its depth sensor. This low-cost and lightweight camera is highly effective in enabling the development of applications that can deal with their surroundings [14].
For this work, depth information is not required.
It is used as a normal 2D camera provided of a RGB module. It is connected through the PC using a USB cable and posed on the end-effector of the robot manipulator.
In order to estimate the pose of a marker, calibration parameters of the camera must be known.

Figure 4.1. Intel RealSense d435

4.1 Camera Calibration

The process of camera calibration is the estimation of the characteristics of a camera.
This involves obtaining all the necessary information about a camera, such as parameters or coefficients, to establish a precise relationship between a 3D point in the real world and its corresponding 2D projection in the image captured by that calibrated camera.
Two types of parameters are typically recovered during camera calibration.
The first type is *intrinsic or internal parameters*, which enable mapping between pixel coordinates and camera coordinates in the image frame. This includes: the optical center (c_x, c_y), focal length (f_x, f_y) and radial distortion coefficients of the lens that are combined in the 'camera matrix':

$$\text{Camera matrix} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}$$

and a vector of 5 or more elements that models the distortion produced by your camera.
The second type is *extrinsic or external parameters*, which describe the camera's orientation and location. This pertains to the rotation and translation of the camera relative to some world coordinate system [15].

To obtain these parameters a Python script denoted *calibration.py* is created. This code performs camera calibration using 100 images of a chessboard pattern. The images are stored in a directory and loaded one by one. An example of the images used for the calibration process is presented on the following page.

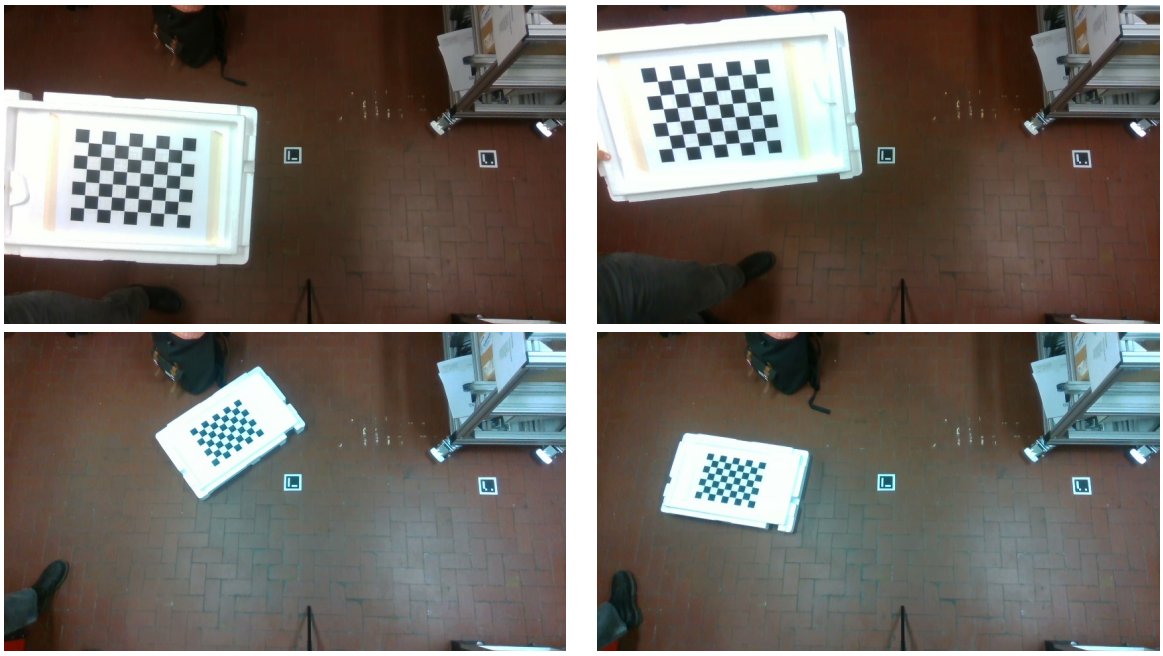


Table 4.1. Images for calibration process

The chessboard size is defined and object points are created for the corners of the chessboard.
The code loops over the calibration images and finds the chessboard corners in each image. If the corners are found, they are added to the image and object points lists, and drawn on the image.
After all the images have been processed, the camera is calibrated using the *cv2.calibrateCamera* function, which takes the image points and object points lists, as well as the shape of the images, as input.

```
# Loop over the calibration images
for i in range(1, 100):
    # Load the calibration image
    img = cv2.imread(f'/home/leonardo/catkin_ws/src/UR5/scripts/aruco_data/{i}.jpg')

    # Find the chessboard corners in the image
    ret, corners = cv2.findChessboardCorners(cv2.cvtColor(img,
    cv2.COLOR_BGR2GRAY), chessboard_size, None)

    # If we found the corners, add them to our lists
    if ret:
        image_points.append(corners)
```

```
        object_points_list.append(object_points)

        # Draw the corners on the image
        punti = cv2.drawChessboardCorners(img,
        chessboard_size, corners, ret)

        cv2.imshow('punti', punti)


# Calibrate the camera
ret, camera_matrix, distortion_coefficients, rvecs, tvecs =
cv2.calibrateCamera(object_points_list, image_points, img.shape[:2],
None, None)
```

The function returns the camera matrix and distortion coefficients, which are printed to the console.
The values obtained are the following:

$$\text{Camera matrix} = \begin{bmatrix} 620.11 & 0 & 430.98 \\ 0 & 619.66 & 252.6263 \\ 0 & 0 & 1 \end{bmatrix}$$

$$\text{Distortion coefficients} = \begin{bmatrix} 0.1661 & -0.5307 & 0.0051 & -0.0007 & 0.506 \end{bmatrix}$$

4.2 OpenCv

To accurately detect the position of an ArUco marker, the implementation of advanced algorithms available in the OpenCV library is required.
OpenCV (Open Source Computer Vision Library) is an open source computer vision and machine learning software library.
The library has more than 2500 optimized algorithms, which includes a comprehensive set of both classic and state-of-the-art computer vision and machine learning algorithms. These algorithms can be used to detect and recognize faces, identify objects, track moving objects, extract 3D models of objects, produce 3D point clouds from stereo cameras etc [16].
One of the algorithms provided by OpenCV is the "ArUco marker detection algorithm", which allows the detection and tracking of ArUco markers in real-time.
An ArUco marker is a marker used for augmented reality applications. It is made up of a black border and an inner binary matrix that determines its unique identifier or id. The black border is designed to make the marker easily detectable in an image, while the binary matrix allows for error detection and correction techniques to be applied.

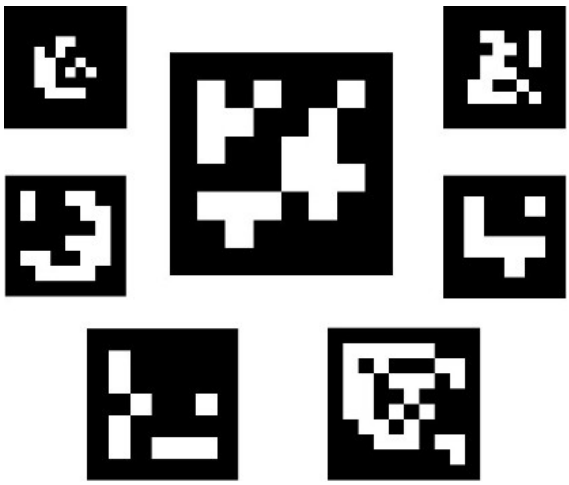


Figure 4.2. Example of markers images

The size of the marker determines the size of the internal matrix, with a 4x4 marker being made up of 16 bits.
A dictionary of markers is a set of markers that are considered in a specific application, represented by their binary codifications.
The main properties of a dictionary are the dictionary size and the marker size. The dictionary size refers to the number of markers that are included in the dictionary, while the marker size refers to the size of each marker, which is determined by the number of bits used to represent it.
The ArUco module includes some predefined dictionaries that cover a range of different dictionary sizes and marker sizes, making it easier for developers to choose the appropriate dictionary for their specific application [17]. The dictionary chosen for this application is the 6x6_250. This dictionary consists of 250 unique markers, each of which is a 6x6 square.

4.3 Detection and Pose estimation Algorithm

Detection

The marker detection process is a complex process that involves two main steps.
The first step is the detection of marker candidates, where the image is analyzed to identify square shapes that may be markers. This is done through an adaptive thresholding process that segments the markers, followed by the extraction of contours from the thresholded image. Any contours that are not convex or do not approximate to a square shape are discarded. Additional filtering is also applied to remove contours that are too small or too big, or too close to each other.
Once the candidate markers have been identified, the second step involves analyzing their inner codification to determine if they are indeed markers. This process starts by extracting the marker bits of each marker. To achieve this, a perspective transformation is first applied to obtain the marker in its canonical form. Then, the canonical image is thresholded, separating white and black bits. The image is then divided into different cells based on the marker and border size. The number of black or white pixels in each cell is counted to determine if it is a white or black bit. Finally, the bits are analyzed to determine if the marker belongs to the specific dictionary, with error correction techniques employed when necessary.
In the ArUco module, the detection is performed in the *detectMarkers()* function. This function is the most important in the module, since all the rest of the functionality is based on the detected markers returned by *detectMarkers()* [17].

Pose Estimation

After the ArUco marker is detected using the *detectMarkers()* function in the OpenCV ArUco marker detection algorithm, the next step is to estimate the pose of the marker.

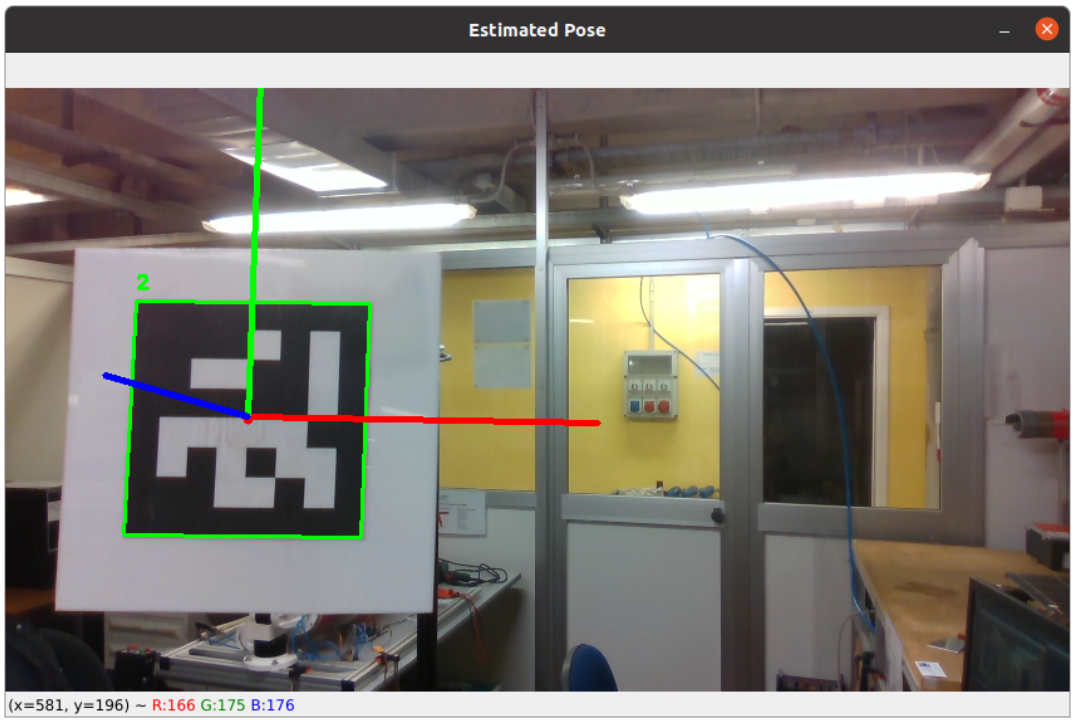


Figure 4.3. Aruco frame

The *estimatePoseSingleMarkers()* function in OpenCV can be used to estimate the pose of a single marker using the camera parameters and the marker's dimensions. The marker size is the physical size of the marker, which is usually known in advance.
Once these inputs are provided, the *estimatePoseSingleMarkers()* function estimates the rotation and translation vectors of the marker relative to the camera.
These vectors represent the pose of the marker in 3D space. The rotation vector contains information about the marker's orientation relative to the camera, while the translation vector specifies the marker's position in the camera's coordinate system [17].

Accurate position information is crucial for controlling the robot. To facilitate this task Robot Operating System (ROS) is exploited. Indeed, within the code, a ROS node known as 'aruco_pose_publisher' is created, which acts as a central hub for publishing the ArUco marker

pose information. By publishing this information in the '/aruco_pose' topic, other nodes in the ROS network can easily access and use this data.

Chapter 5

Control implementation

Once the ArUco marker position information has been obtained, the next step is to develop motion control.
It is achieved by calculating the desired velocity value and send it to the robot. The speed will be calculated using information on the position of the object to be reached.

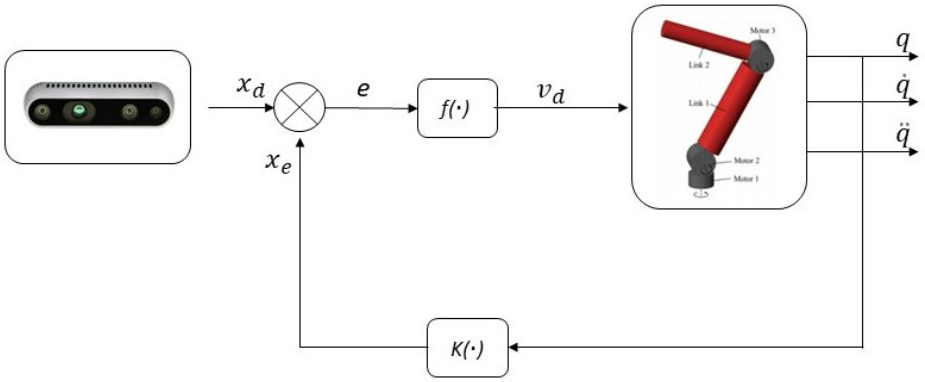


Figure 5.1. Control scheme

Since the prototype of the Popup robot is not yet available to use, the control codes are tested on the collaborative robot UR5 from Industrial Robot, which is located in the DIMEAS (Department of Mechanical

and Aerospace Engineering) laboratory.

UR5

The UR5 is a versatile and flexible industrial robot manufactured by Danish company Universal Robots. It has a reach of 850 mm (33.5 in) and a payload capacity of 5 kg (11 lbs), making it suitable for a range of applications such as pick-and-place operations, machine tending, assembly, and testing. The UR5 has six joints. Each joint can rotate of $\pm 360°$ and can reach a maximum speed of $\pm 180°$s.
The robot is almost completely made of steel besides the junctions between links which are covered with PP plastic [18].
It can be programmed using a simple, intuitive graphical interface, by manually moving the robot arm or, by sending commands using a Python script, as it is done in this work.

Figure 5.2. UR5

5.1 Communication protocols

A main python script is developed to implement the control.
This script receives the information of the position of the ArUco markers, the robot's joint and tool and uses it to calculate the speed commands to be sent to the UR5.
The data of the position of the Aruco is obtained from the topic subscribe "/aruco_pose".

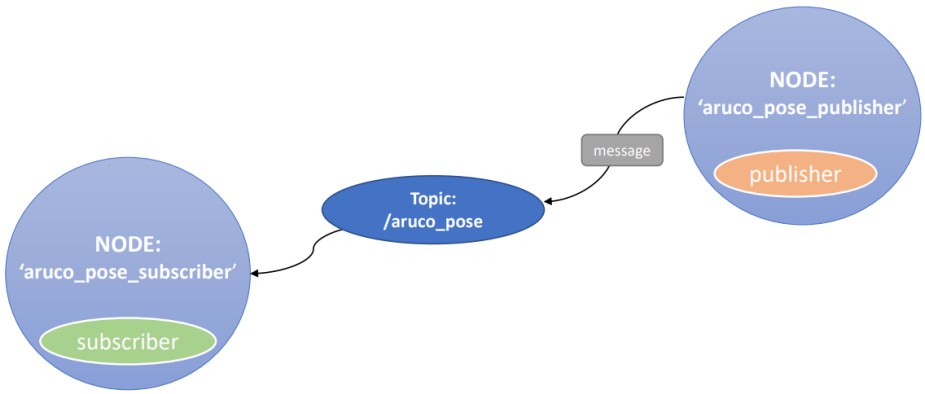


Figure 5.3. Publisher / Subscriber Communication

The communication with the robot is performed through the use of sockets.
Sockets are a mechanism for communication between processes running on different computers over a network [19]. They allow programs to send and receive data across a network.
The two main components for socket communication are: socket client and socket server.
A socket server is a programme that listens for incoming connections from other devices, known as clients. The server is usually run on a specific IP address and port number and waits for clients to connect to it.
A socket client, on the other hand, is a programme that connects to a socket server to initiate communication. The client usually knows the IP address and port number of the server it wants to connect to.

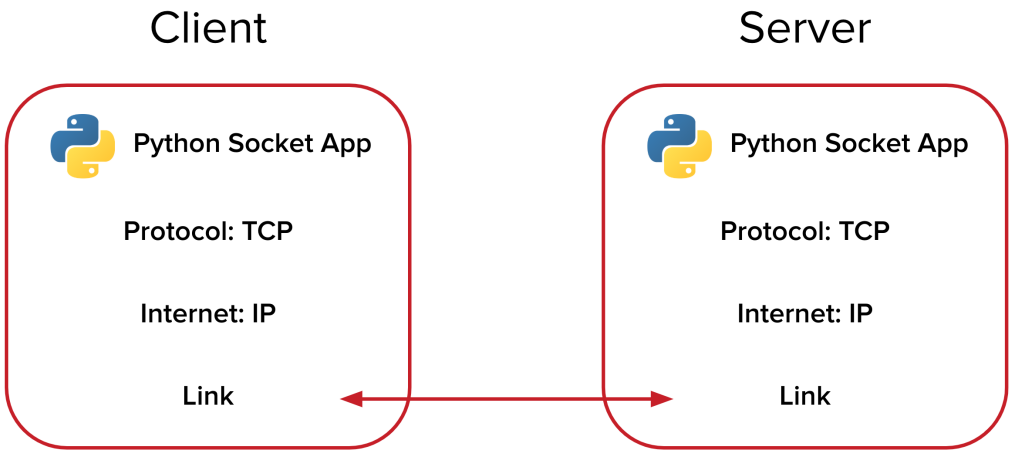


Figure 5.4. Client Server socket communication

Once the connection is established, communication between server and client is performed using send and receive operations.

The initialisation of the TCP/IP socket connection between a client device (UR5 in this case) and a server device (PC used) is realised by creating a Python function called 'initialisation()' which returns the

values required to continue communication.
The function initialises the IP addresses of the client and server devices, HOST and server respectively.
It also initialises the port number that the client will use to connect to the server and the port number of the fncPort function used to receive feedback on the execution of commands on the robot.
The function creates a new socket object using the socket module, a standard Python library.

```
def initialization():

    HOST= '169.254.123.5'
    server='169.254.123.1'

    PORT=30003
    fncPort = 2000
    # Memory preallocation

    # Flag initialization
    exit=False


    # TCP/IP connection
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.connect((HOST, PORT))

    return( exit, HOST, server, fncPort, s)
```

Finally, the function returns a tuple containing the initialised values: the exit flag (initialised to False), the IP addresses of the HOST and server, the port number of the fncPort function, and the socket object "s". The communication is executed at 125 Hz.

5.2 Control Algorithm

CONTROL_ONLINE.py is the main python script created to implement robot control. Several libraries and functions defined in advance are used within it.
A block diagram showing the control's operating scheme is shown below.

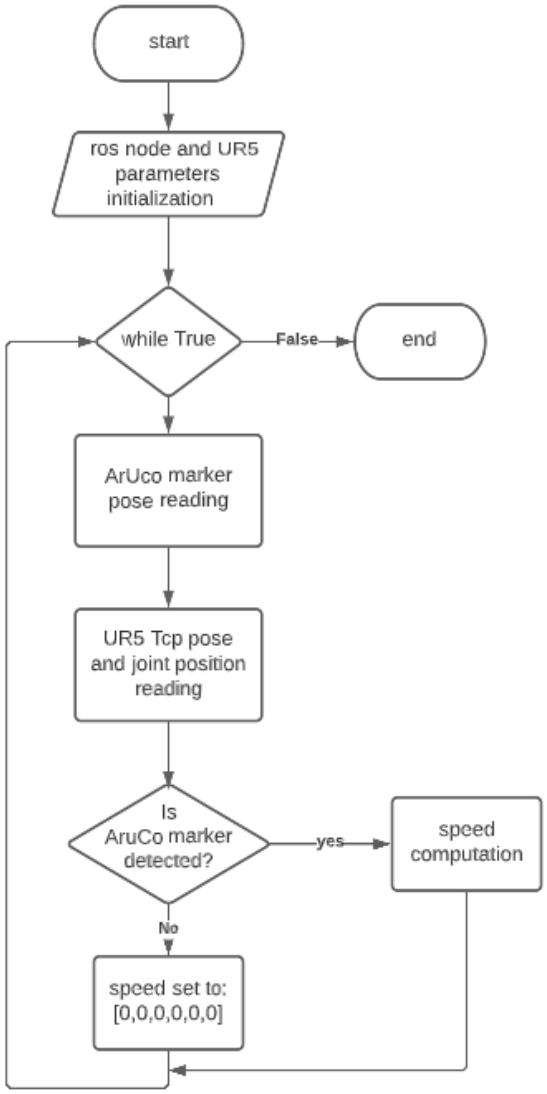


Figure 5.5. Control code block diagram

As shown in the diagram above, the first step is to initialise the parameters.
A TCP/IP connection with the UR robot is initialised using the "UR.initialisation()" function.
This function returns a number of variables including a socket ("soc") which is used for communication with the robot.
A callback function called "pose_callback" is then defined, which is executed whenever a ROS message arrives on the topic "/aruco_pose". This topic is used to receive the position and orientation of the AruCo object with respect to the camera.

```
[ exit , HOST, server , fncPort , soc] = UR. initialization ()

def pose_callback(pose_msg):
    global position_aruco

    position = pose_msg.pose.position
    orientation = pose_msg.pose.orientation
```

Once the AruCo object has been detected, the programme enters a while loop that controls the robot's movement based on the object's position.
The while loop is executed continuously until the programme is interrupted.
Within the while loop, the programme checks whether the position of the AruCo object is equal to [1000, 1000, 1000, 1000, 1000, 1000].
This value is used as a stop signal in the event that the AruCo object is not detected by the camera.
If the position has not been detected (i.e. position_aruco is equal to [1000, 1000, 1000, 1000, 1000, 1000]) the end-effector robot is stopped via the "speedl" function with a speed value of [0.0, 0.0, 0.0, 0.0, 0.0, 0.0]. In this way, the robot remains inactive until the position of the AruCo object is detected.
This command requires two main arguments: the first is an array of six elements representing the linear speed of the robot's tool in terms of metres per second for each axis (x, y, z, rx, ry, rz), while the second

is a maximum acceleration value allowed by the robot.

```
if(position_aruco == [1000,1000,1000,1000,1000,1000]):

    print("ARUCO_NO_DETECTED")
    messaggio = f'speedl([0.0,0.0,0.0,0.0,0.0,0.0], 0.00)\n'
    soc.send(messaggio.encode())
```

If the position of the AruCo object has been detected, the code proceeds to calculate the velocity vector.
In order to obtain the velocity vector, first, the position of the tool relative to the camera is calculated, with a safety margin of 20 centimeters in the z-axis, to ensure that the tool does not come into contact with the AruCo object during movement during the test phase, and and a displacement of 17 mm on the x-axis, due to the misalignment of the camera with the tool.

Next, the code calculates the maximum permissible speed for the movement of the robot of the end-effector robot. This value is directly proportional to the Euclidean distance between the robot tool and the target to be reached, multiplied by a factor of 0.5. In other words, the closer the tool is to the target, the lower the maximum speed.
This maximum speed value is then used to calculate the linear speed of the end-effector robot in each axis (x, y, z), based on the direction of the positional vector between the robot's current position and the desired position of the tool.
The maximum acceleration value is set at 0.05 m/s^2, a rather low value, which means that the robot's movement will be very slow and controlled, so as to ensure safety when handling objects.

```
else:
    print("Aruco_detection")
    posiz_a = position_aruco[:3]

    #position with safety margin
    posiz_tool = [posiz_a[0] - 0.0017 , posiz_a[1] , posiz_a[2] - 0.20]
```

```
norma = math.sqrt(sum([x**2 for x in posiz_tool]))
velocita_max = 0.5*norma
versore = [x/norma for x in posiz_tool]

#velocity vector
velocita_tool = [velocita_max*x for x in versore]
```

Finally, the code converts the linear speed from the camera reference system to the robot base reference system, using the 'rotation' function.

The "rotation" function takes a vector of six joint values as input and returns a rotation matrix R. The rotation matrix allows the velocity vector to be transposed from the tool reference system to the base reference system.

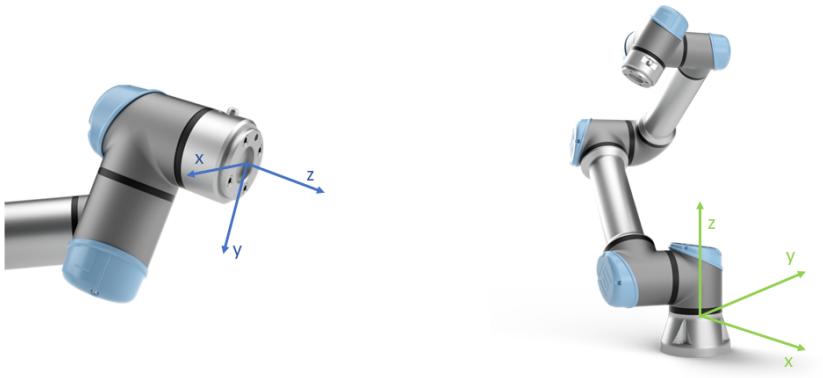


Figure 5.6. Tool reference frame and base reference frame

The calculation of these matrices is recursive and is obtained by simply multiplying the homogeneous matrices $A_i^{i-1}(q_i)$ where each of which is a function of a single joint variable. The Denavit-Hartenberg convention has been adopted to select the reference systems of each link. It is a method which defines four parameters associated to each link in

order describe the position of the reference frames of each joint of the robotic arm.
The four parameters defined for each link i are:

- **Offset Distance** a_i : distance between z_i and z_{i1} measured along x_i
- **Translation distance** d_i: : distance between axes x_i and x_{i1} measured along the positive direction of z_{i1}
- **Twist angle** α_i: between axes z_{i1} and z_i . It is the angle required to rotate the axis z_{i1} into alignment with the axis z_i in the right-hand sense about axis x_i
- **Joint angle** θ_i: between axes x_{i-1} and x_i. It is the angle required to rotate the axis x_{i-1} into alignment with the axis x_i in the right-hand sense about axis z_{i1}.

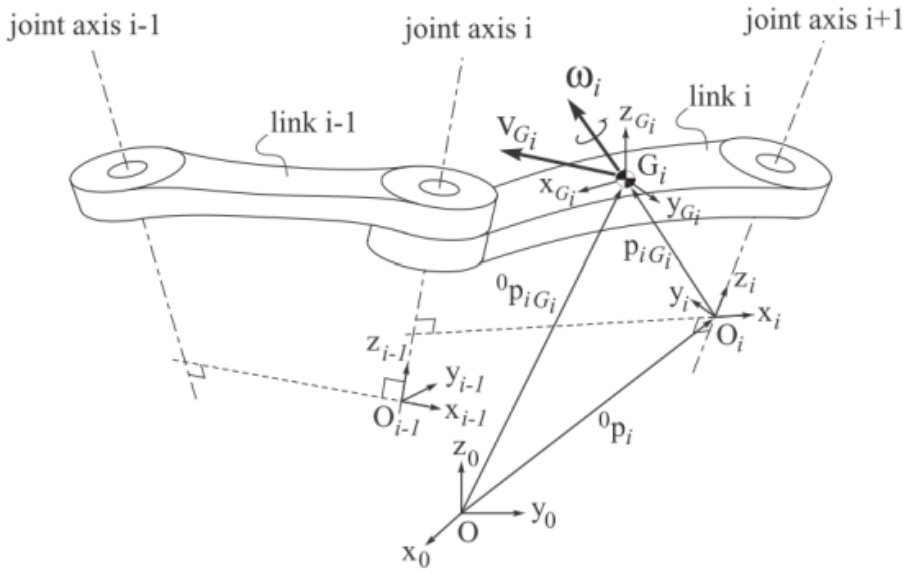


Figure 5.7. Standard Denavit Hartenberg convention representation

The table below, obtained from the Universal Robots website, outlines the DH parameters for each link of the UR5 robot.

The robot arm's degrees of freedom are represented by the letter q, and they differ based on the arm's configuration during a particular moment in its trajectory.

Joint	$q_i[^\circ]$	$d_i[m]$	$a_i[m]$	$\alpha_i[^\circ]$
Base	q1	0.089159	0	90
Shoulder	q2	0	-0.425	0
Elbow	q3	0	-0.39225	0
Wrist 1	q4	0.10915	0	90
Wrist 2	q5	0.09465	0	-90
Wrist 3	q6	0.0823	0	0

Using the Denavit Hartenberg parameters and taking the positions of the joints as input, the function created, i.e. 'rotation.py' calculates the translation matrix from the tool to the base, by postmultiplication of the single transformations as:

$$A_i^{i-1} = \begin{bmatrix} c\theta_i & -s\theta_i c\alpha_i & s\theta_i s\alpha_i & a_i c\theta_i \\ s\theta_i & c\theta_i c\alpha_i & -c\theta_i s\alpha_i & a_i s\theta_i \\ 0 & s\alpha_i & c\alpha_i & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{5.1}$$

A three-element vector composed by three zeros is added to the velocity vector.
This vector represents the rotation speeds around the three axes, which for this study has been intentionally set to zero. The final velocity vector, composed by 6 elements, is then sent to the robot via the 'speedl' command.

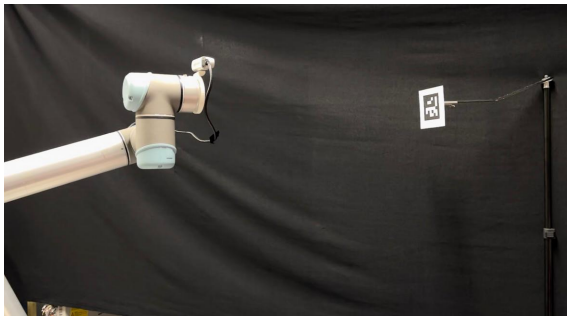
```
R = rpy.rotazione(theta)

#direct kinematics
velocita_base = R @ velocita_tool

#velocity vector
velocita = list(np.append(velocita_base,[0,0,0]))

messaggio = f'speedl({velocita}, 0.05, 2)\n'
soc.send(messaggio.encode())
```

Overall, the while loop is responsible for constantly checking the position of the AruCo object relative to the camera and moving the end-effector robot in a controlled and safe manner to the desired position. The speed of the while loop and thus of commands transmitted by the robot is 125 Hz.

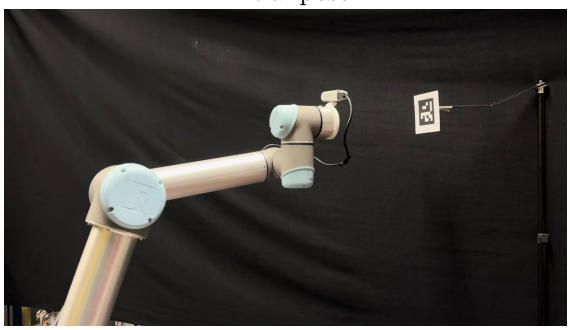
5.3 Results

5.3.1 Test with stationary target

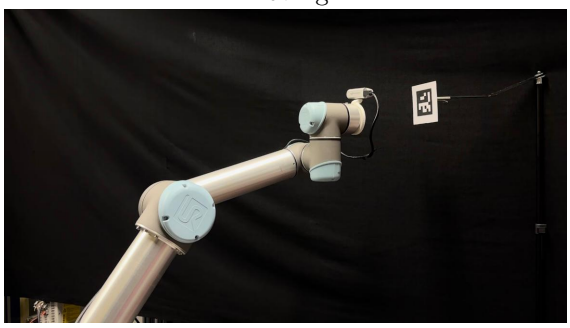
The first test was carried out by holding the target stationary and verifying that the robot detected it and approached it correctly.
Below is a sequence of images showing the movements of the manipulator robot, from the initial position, to the final position where it reached the target



Initial pose



Moving



Final pose

Table 5.1. Trajectory to reach the target

The graph below shows the evolution of distance and speed in relation to time.
The graph proves to be consistent with what is set in the algorithm for the maximum speed value. According to the relation:

$$speed = 0.5 * distance \tag{5.2}$$

Indeed, the maximum speed value for the duration of the test is always half of the distance value.
The robot reaches the target in approximately 10 s.

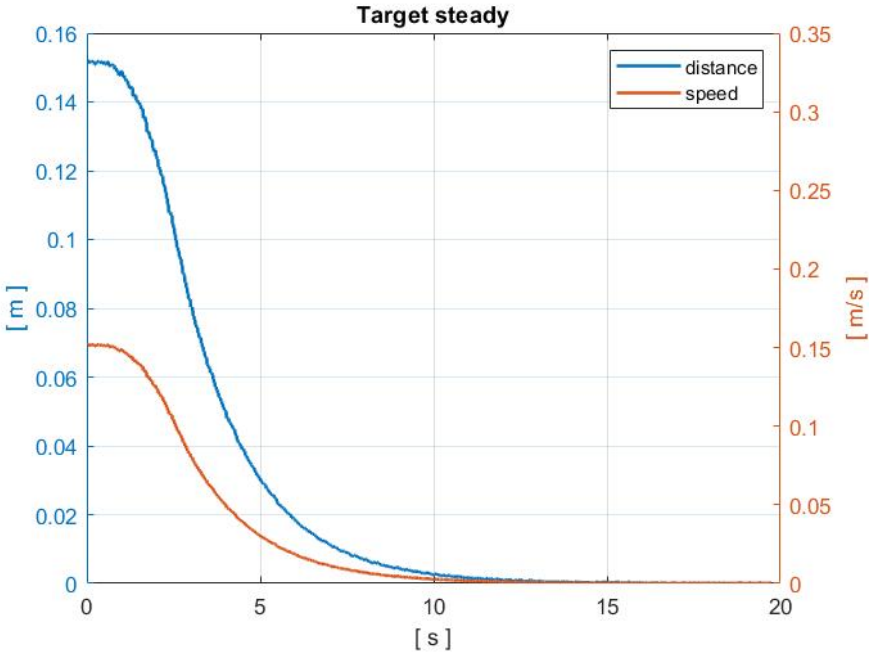


Figure 5.8. Distance and Speed in relation to time

The position of the ArUco target is then analysed in two different reference systems: base reference system and tool reference system. The position of the ArUco should remain unchanged with respect to the base, but calculating the standard deviation for the position with respect to each axis, we obtain:

$$\sigma_x = 0.0026m \tag{5.3}$$

$$\sigma_y = 0.0099m \tag{5.4}$$

$$\sigma_z = 0.0021m \tag{5.5}$$

According to the figure 5.9, the major displacement occurs along the y-axis. This was caused by the inaccuracy of the vision algorithm.

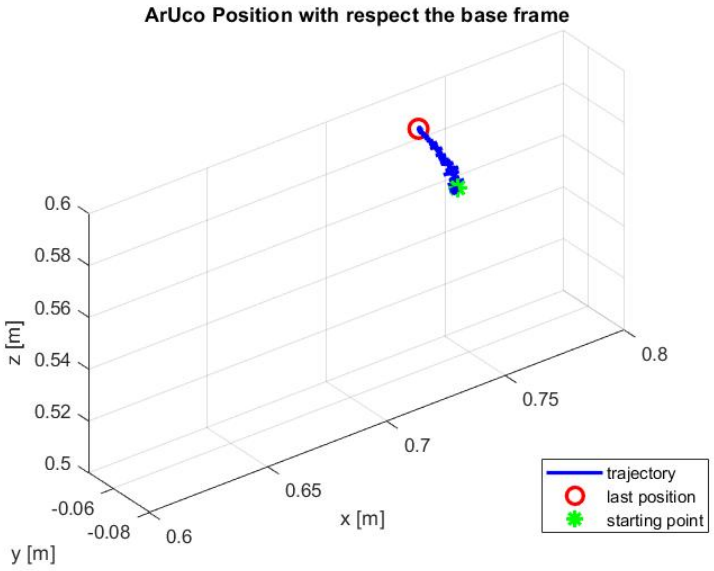


Figure 5.9. Aruco Position with respect the base frame

The plot depicts the trajectory of the ArUco marker in the tool's reference system.
The final position, should coincide with the point: [0.0017, 0, 0. 20]. It is the position of the target.
These values represent the offset that has been set to maintain a distance from the marker such that it always remains in the visible field.
The target point is represented by the black cross, whereas the actual end point reached in the simulation is represented by the red circle.
As the graph shows, the final position is very close. In particular, it coincides with [0.0171, 6.3770e-05, 0.1996] m.

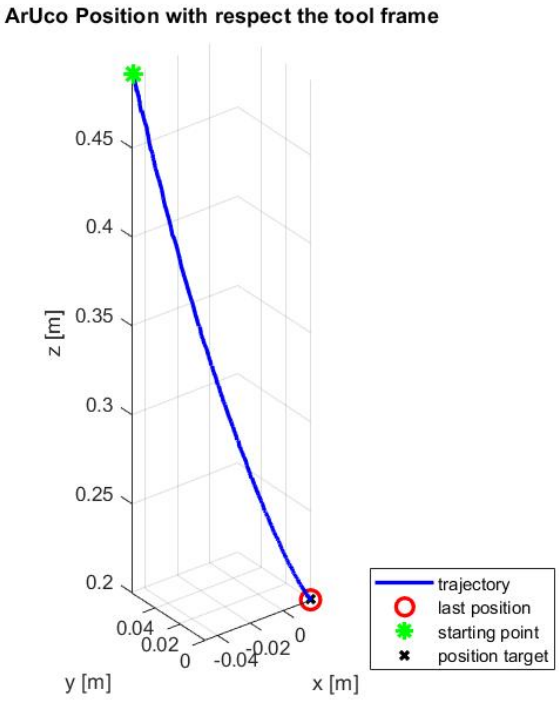


Figure 5.10. Aruco Position with respect the tool frame

Finally, the last graph illustrates the trajectory of the tool with respect to the robot's base reference system.

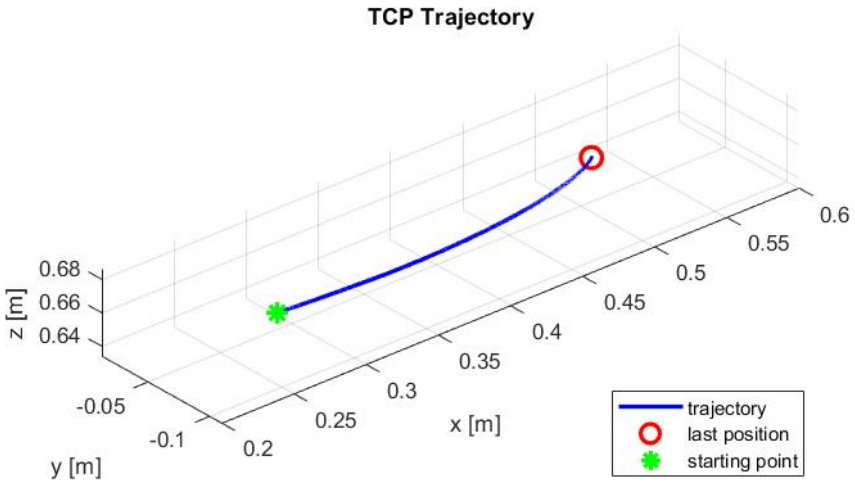
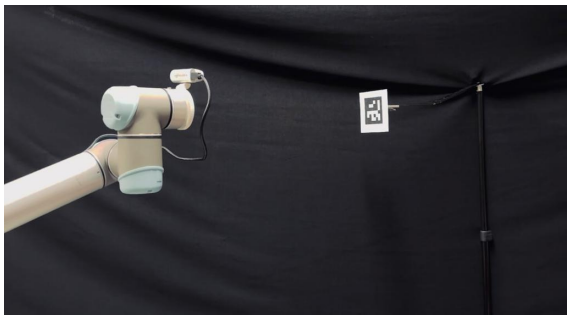


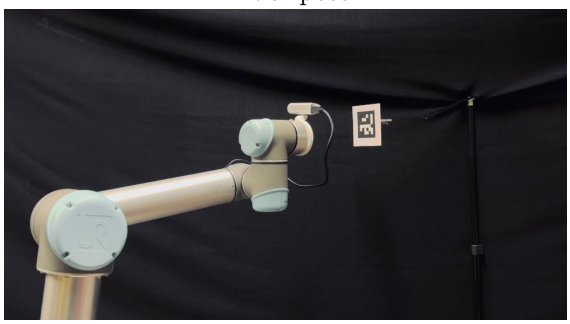
Figure 5.11. TCP trajectory in base frame

5.3.2 Test with moving target

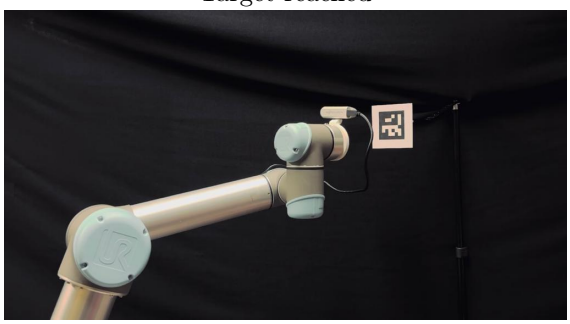
The second test was carried out by holding the target stationary and waiting for the tool to approach. Then the target was moved.



Initial pose



Target reached



Target following

Table 5.2. Trajectory to follow the target

As can be seen from the graph below, the distance and speed decrease when the robot is approaching the target, and then increase again when the marker is moved.

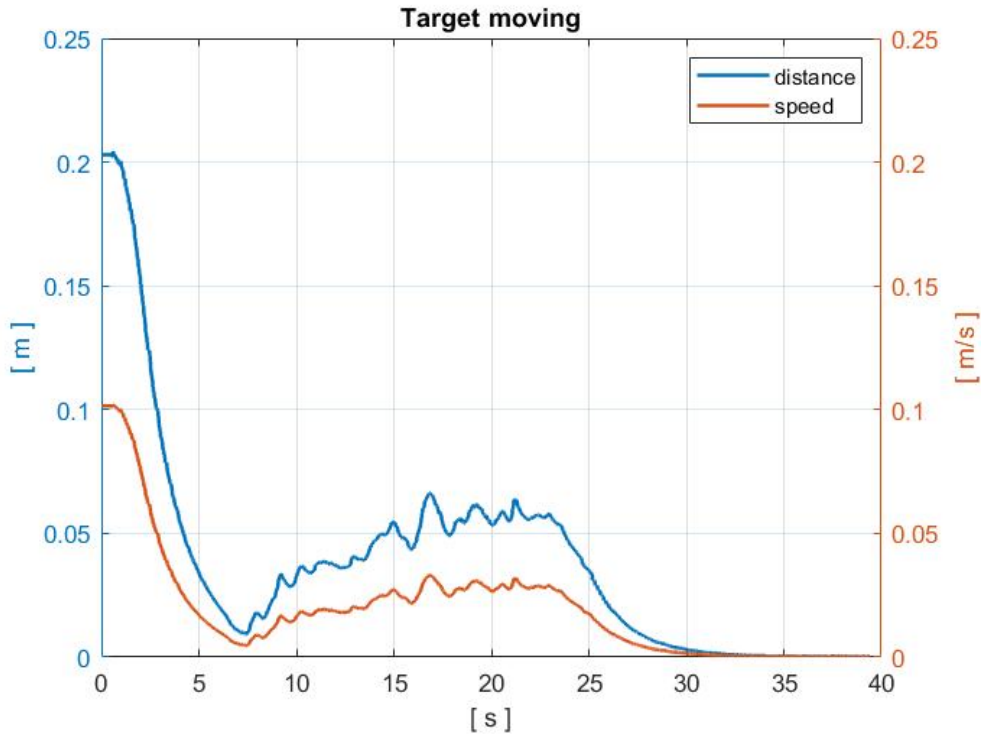


Figure 5.12. Distance and Speed in relation to time

The displacement of the target is more evident when analysed with respect to the base reference system.
The green dot represents the start position, the red circle the end position. In this case, the displacement occurs along the y-axis of the base reference frame.

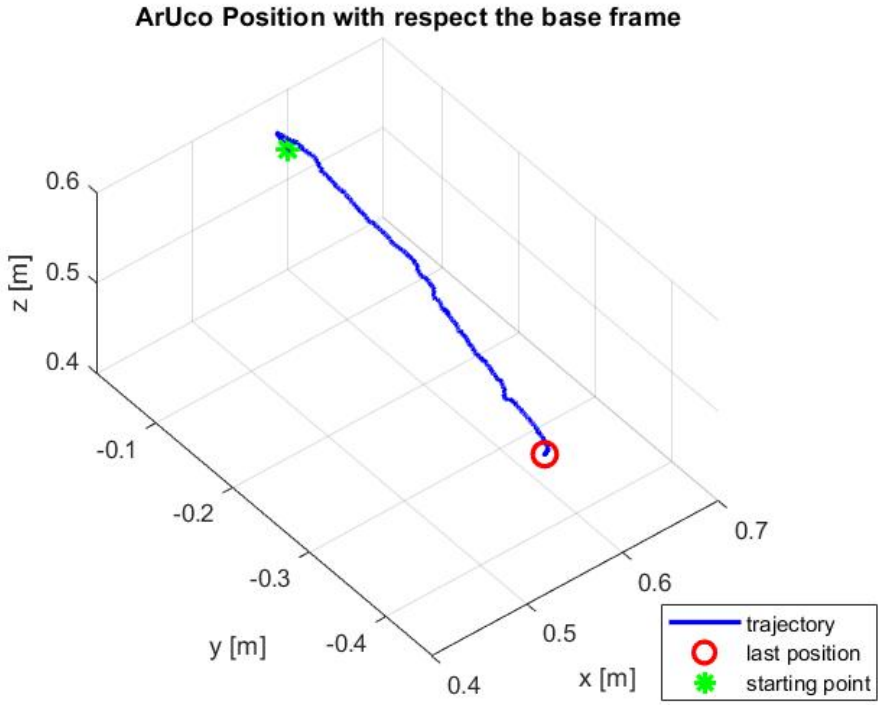


Figure 5.13. Caption

The graph shows the trajectory of the ArUco marker in the tool reference system.
It first approaches the target and then follow it.

As in the simulation discussed above, the robot in its final position should reach the point [0.017, 0, 0.2] m.
The actual final position reached is: [0.017, 6.25e-05, 0.199] m.

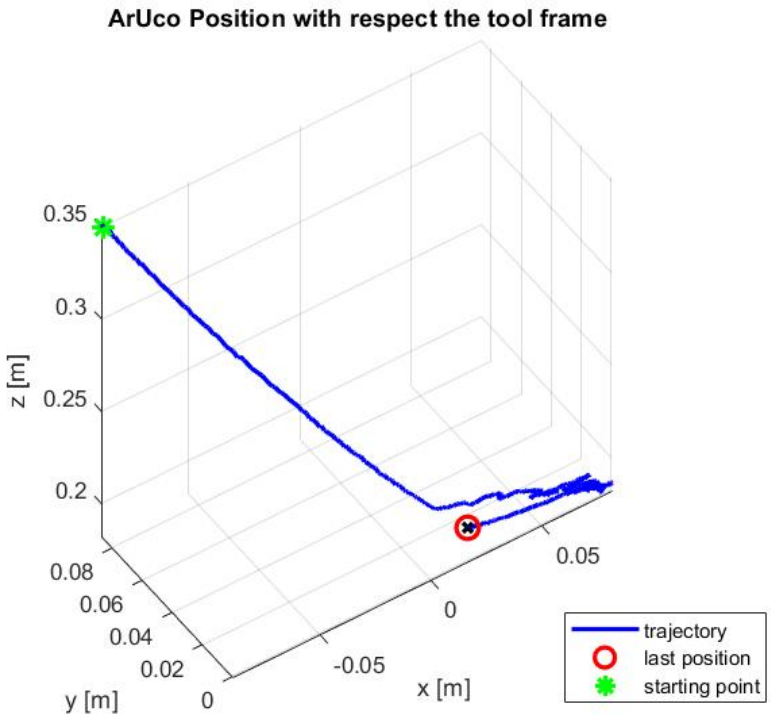


Figure 5.14. ArUco Pose in the tool frame

Finally, the last graph illustrates the trajectory of the tool with respect to the robot's base reference system.

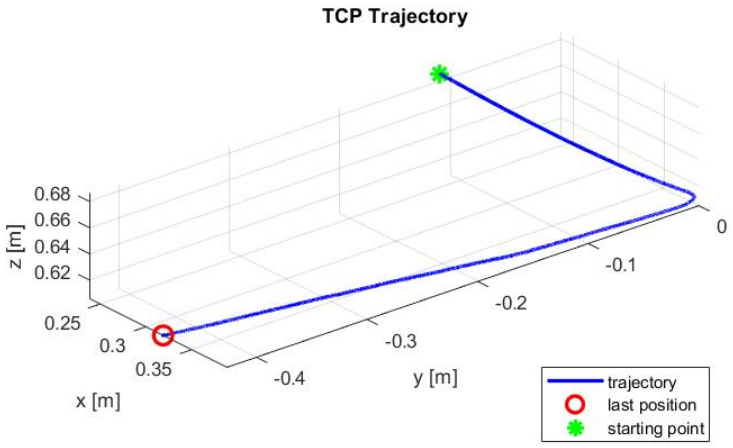


Figure 5.15. TCP Trajectory in base frame

Chapter 6

Conclusion

The principal goal of the thesis work was to propose a visual servoing control strategy for the POPUP robot prototype.

The first phase of the work involved the study of the ROS 1.0 middleware, the Gazebo simulation environment and all its packages. ROS 1.0 proved to be a powerful tool for controlling the robot, either on the virtual environment.

This phase was followed by the creation of the model of the POPUP robot on the virtual environment. The dissertation describes the procedure for obtaining a model of the robot that could describe its dynamic behaviour.
The model of a virtual camera on the robot's end-effector and a package to control the robot were included in the Gazebo world. These two tools will allow the test of possible control algorithms on the virtual model for future work

The next step was to develop a visual servoing control strategy and test it on Industrial Robot's UR5 robot in the DIMEAS laboratories. Before moving on to the development of the control algorithm, the different visual servoing techniques were analysed and the most suitable was chosen according to the robot's tasks. A careful analysis was also performed regarding the different cameras to be used, searching for the best compromise between precision, size and weight. Subsequently, the

camera was calibrated and the Python code for control realised.

The main difficulty in this step was the synchronisation between the data received by the ros node, by the UR5 and the commands sent to the UR5. To avoid communication delays, the communication channel between the control code and the robot (the socket) is left open. Data is received by reading the buffer and the selected the most recent data. Thanks to the power of the Python language, it was possible to achieve real-time communication.

Finally, the control was tested in two different situations: firstly, by holding the marker steady and waiting for the robot to approach it, and secondly by moving the marker and checking that the robot followed it. In both cases, accurate results were obtained and the market was followed in real time.
The visual servoing strategy thus proved to be feasible and robust. Although the test was carried out on a rigid robot, the independence of the robot's inverse kinematics, the optimal results obtained and the good performance demonstrated its transversality in being adopted in different applications and for controlling different types of robots.

Future developments will include the implementation of visual servoing in the POPUP robot prototype.
It could also be considered to include also a control algorithm for the orientation of the tool.

Bibliography

[1] Yongchang Zhang, Pengchun Li, Jiale Quan, Longqiu Li, Guangyu Zhang, and Dekai Zhou. Progress, challenges, and prospects of soft robotics for space applications. *Advanced Intelligent Systems*, 2022.

[2] Garcia Mark. Remote manipulator system (canadarm2). www.canadarm2.int [Online; Page Last Updated: Oct 24, 2018].

[3] Pierpaolo Palmieri, Matteo Gaidano, Andrea Ruggeri, Laura Salamina, Mario Troise, and Stefano Mauro. An inflatable robotic assistant for onboard applications, 2021.

[4] JoÃo Oliveira, Afonso Ferreira, and JoÃo C.P. Reis. Design and experiments on an inflatable link robot with a built-in vision sensor. *Mechatronics*, 2020.

[5] Luigi Villani Giuseppe Oriolo Bruno Siciliano, Lorenzo Sciavicco. *Robotic Modelling, Planning and Control*. 2009.

[6] Josie Hughes, Utku Culha, Fabio Giardina, Fabian Guenther, Andre Rosendo, and Fumiya Iida. Soft manipulators and grippers: A review. *Frontiers in Robotics and AI*, 2016.

[7] Jorge Pomares. Visual servoing in robotics. *Electronics*, 2019.

[8] Laura Salamina Mario Troise Pierpaolo Palmieri, Matteo Gaidano and Stefano Mauro. Design of a lightweight and deployable soft robotic arm for aerospace applications. *International Conference on Intelligent Robots and Systems (IROS*, February 2022.

[9]

[10] Jonathan Cacace Lentin Joseph. *Mastering ROS for Robotics Programming - Third Edition*. 2021.

[11] TechTarget Contributor. 3d mesh. *TechTarget*, Oct 2016.

[12] Gazebo. Tutorial: Ros control. www.roscontrol.int [Online; Last Accessed: March, 2023].

[13] Franz Pucher. Ros control, an overview. www.fjp$_r os.int$ [Online; Page Last Updated: 2023].

[14] Intel Corporation. Depth camera d435 â intel realsense depth and tracking cameras. www.intelrealsense.int [Online; Last Accessed: April, 2023].

[15] Dulari Bhatt. A comprehensive guide for camera calibration in computer vision. www.cameracalibration.int [Online, Last Modified On November 12th, 2021].

[16] OpenCv. About-opencv. www.opencv.int [Online, Last Accessed: March, 2023].

[17] OpenCv. Detection of aruco markers. www.aruco.int[Online; Page Last Accessed: March, 2023].

[18] Universal Robot. Ur5 collaborative robot arm | flexible and lightweight cobot. www.ur5.int [Online; Last Accessed: April, 2023].

[19] Nathan Jennings. Socket programming in python (guide). *Real Python*. www.sockets.int [Online; Last Accessed: March, 2023].