Robotics Project

A mobile robot to pick up LEGO bricks

Pietro Fronza · Stefano Genetti · Giovanni Valer

Chapter 1

Introduction

We are required to explore a known environment with a mobile robot equipped with a 6-DoF-manipulator. In this environment there are 4 circular areas with a known radius and a known centre, with LEGO bricks inside each of them.
LEGO bricks belong to different classes, each with known geometry and a corresponding basket located at a specified point of the map.

There are three different tasks of increasing complexity:

Assignment 1

The mobile robot has to visit all the 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).

Assignment 2

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

Assignment 3

The mobile robot has to visit all the four different areas. In each area, there can be more than one object (possibly of different classes). For each area, the robot has to exactly localise the objects, identify their class, pick them up one by one and take them to the basket. The objects can be anywhere within the circular area and their position on the ground can be arbitrary. Every time the robot picks up an object, it is not forced to go directly to the basket but on its way to the baskets, it can also explore other areas.

Project Repository

The code of our project can be found on **GitHub** at the following link:
https://github.com/jo-valer/Robotics

Furthermore, it might be useful to have a look at some simulations we have done. You can find them on YouTube at the following links:
- **First assignment** (https://youtu.be/23n-PxkJd8o)
- **Second assignment** (https://youtu.be/45ijPx6vJCo)
- **Third assignment**

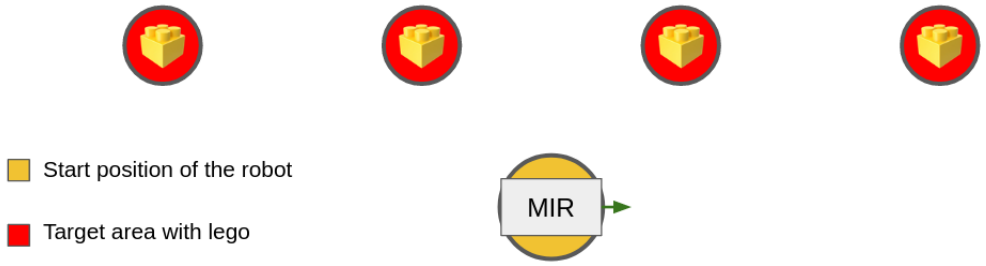
Chapter 2

Workspace environment

The purpose of this chapter is to illustrate the different workflows which have been implemented in order to achieve the requirements of assignment 1, assignment 2 and assignment 3.

2.0.1 Assignment 1

In the first assignment the workspace is populated with four areas. In each of them there is exactly one brick which can be anywhere within the circular area and is positioned in a natural configuration.



Workspace assignment 1

To complete the required tasks, we visit the four areas one by one and perform object classification and localization. The result of the computation is written in a file called `OUTPUT.txt`.
The structure of the file is explained with an example:

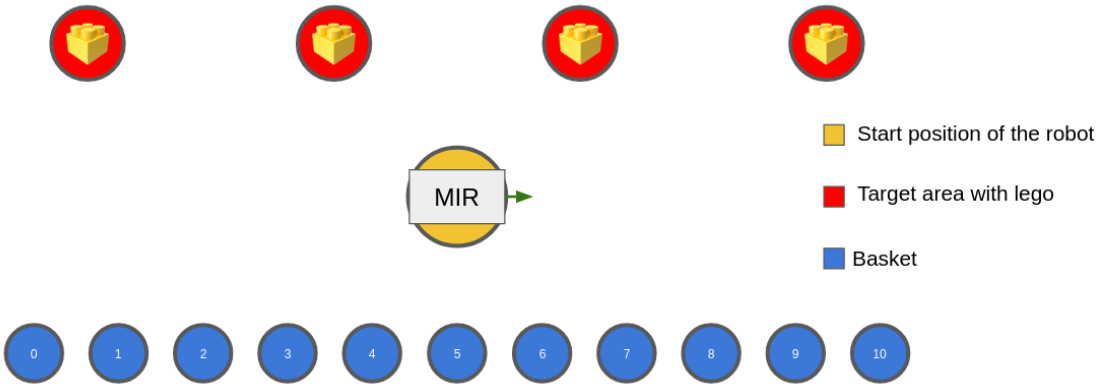
OUTPUT.txt

LEGO class: 9, name: X2_Y2_Z2, x:-3.00616, y:3.00785
LEGO class: 0, name: X1_Y1_Z2, x:-1.00700, y:3.00888
LEGO class: 1, name: X1_Y2_Z1, x:1.07805, y:3.08001
LEGO class: 3, name: X1_Y2_Z2_CHAMFER, x:3.06543, y:3.00799

2.0.2 Assignment 2

In the same way as assignment 1 there are four areas populated with one brick positioned in its natural configuration.
In this assignment we do not only perform object classification and localization in each area. Indeed, after the robot has visited the area, we have also to pick up the brick and bring it to the appropriate basket according to the class of the brick.

As a result also the workspace is different.



Workspace assignment 2

2.0.3 Assignment 3

In this section we present the solutions which have been adopted in order to solve assignment 3 requirements.

In this case the following parameters are decided randomly:

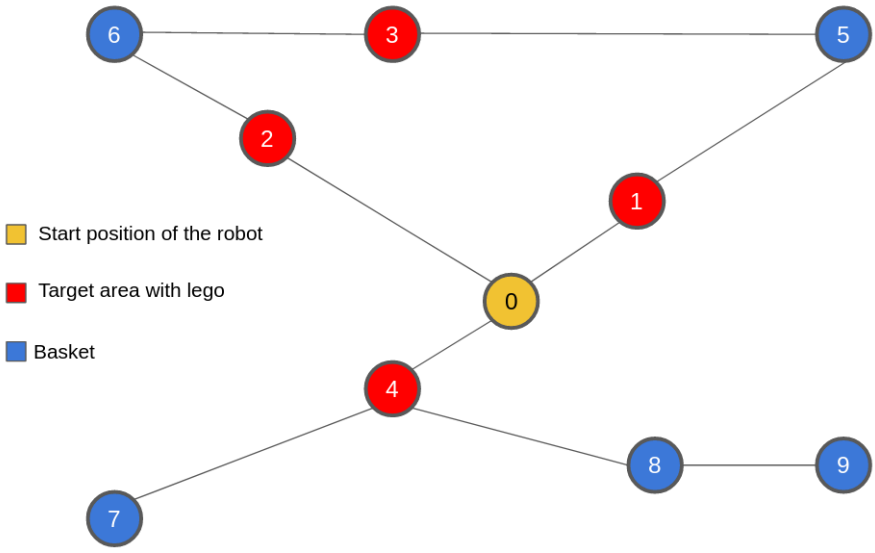
- Number of bricks (between a minimum of zero and a maximum of three) in each area
- Position of the brick in the area
- Pose of the brick on the ground, which can be arbitrary

What is more, target areas and baskets can be everywhere in the map. The latter is abstracted with an undirected graph whose nodes are the target areas and the baskets. All these nodes are connected with each other with edges which represents the routes that the robot can follow. The structure of the graph is written in a file (`map.txt`) with the following structure:

For the sake of clarity we illustrate an example with only five baskets rather than eleven.

map.txt

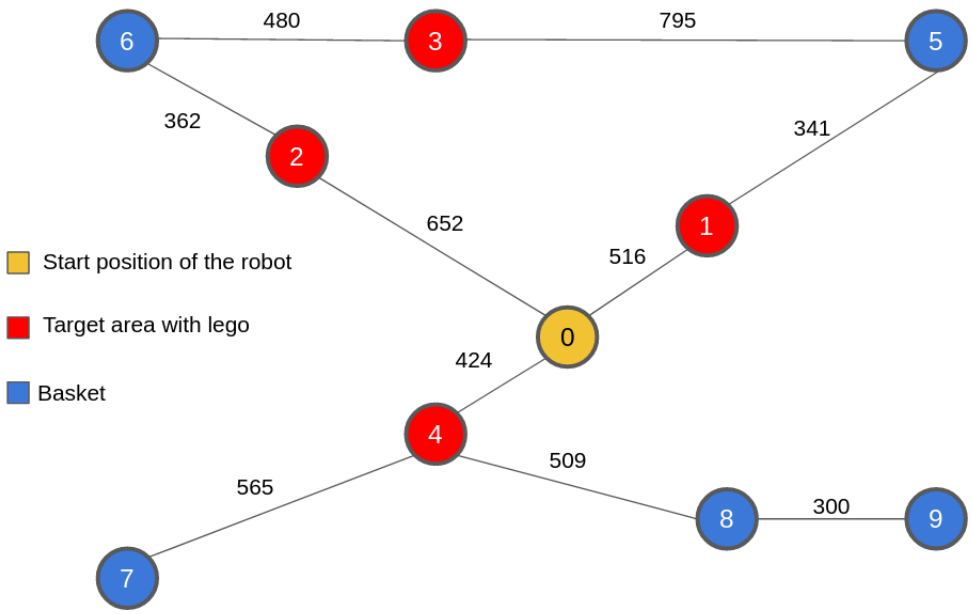
```
0.0 0.0
3.7 3.6
-5.3 3.8
-2.4 5.6
-3.0 -3.0
5.5 6.5
-7.0 7.0
-7.0 -7.0
2.0 -4.0
5.0 -4.0
10
0 1
0 2
0 4
1 5
2 6
3 5
3 6
4 7
4 8
8 9
```



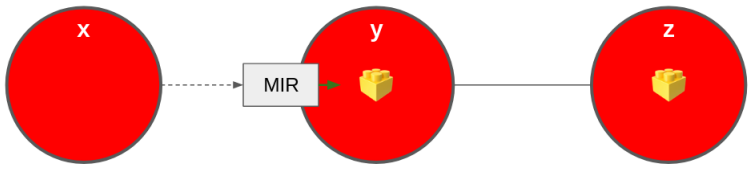
Workspace assignment 3

- in the first line there are two floating point numbers separated with a space which represent start position coordinates,
- the following 4 lines list the coordinates of the target areas,
- in the successive 5 rows there are the coordinates of the baskets,
- in the successive line there is an integer M which denotes the number of edges in the graph,
- in the last M lines there are couples of values $from, to$ which indicates that there is an undirected edge from node "$from$" to node "to".

At run-time we compute also the weight of each edge, which is given by the distance of the two interconnected nodes. Usage of integer values as weights is more convenient. As a consequence of this, we multiply each distance by 100 and then approximate to the integer. At the end of the day the graph looks as illustrated in the following picture:

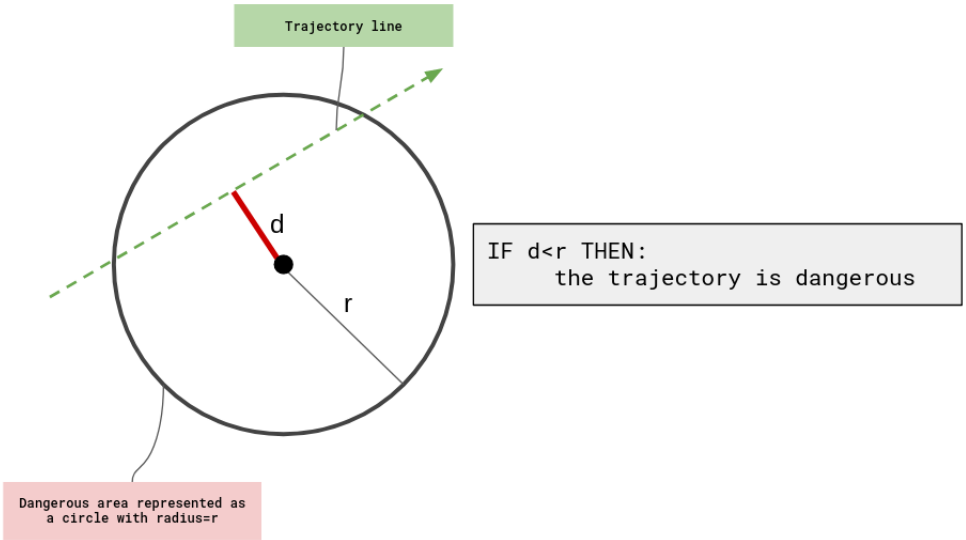


As a consequence of this new configuration of the areas, the robot can approach the area in any direction. Although the routes are designed in order to ensure that the robot doesn't collide with bricks, there exist some dangerous situations. For instance:

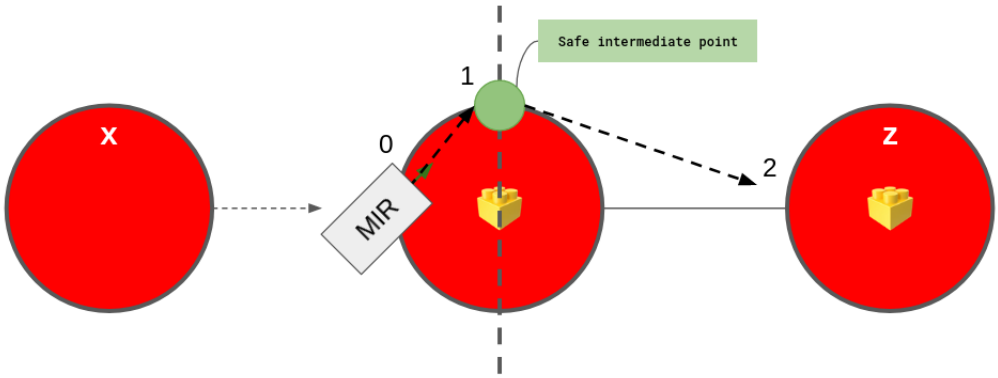


In this case if we follow a straight trajectory to the destination we bounce into some bricks inevitably. In particular we are in a dangerous situation when:

- We are in the first area of the path to the destination. Indeed we have observed that these kind of dangerous configurations are not experienced in the subsequent nodes of the path if we move the robot as it is explained in the paragraph *Following the trajectory path with the MIR robot* below,
- Current area has at least one brick or an unknown number of bricks,
- The straight trajectory to the destination intersects the current area. In order to understand if this is the case we act as illustrated:

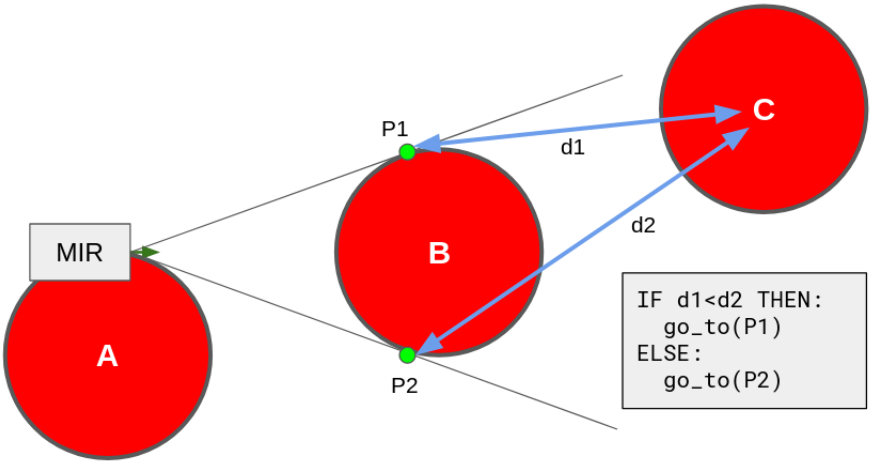


In this case the algorithm moves the robot in a safe intermediate position before travelling to the subsequent destination. The safe intermediate position is computed as illustrated below:



Following the trajectory with the MIR robot

The aim of this paragraph is to explain how we follow the path from an area A to an area B. First of all, suppose we have already computed the path from area A to area B. If A is the first area of the path, we have to check if at the beginning we are in a dangerous configuration. If this is the case we proceed as explained earlier. Otherwise we consider two tangent points and then we choose to follow the one which is more near to the subsequent area along the path. The situation is better explained in this picture:



Obviously, in the last point-to-point movement of the path there is not the area identified with letter C in the picture above. So, in the final maneuver we compute P1 and P2 as explained before and then we simply consider the closest.

Assignment 3 computational process

The purpose of the computational process is to explore all the different target areas, localize and classify all the bricks, pick up them one by one and bring them to the appropriate basket according to the particular brick's class. Particular attention has been paid to the sentence: *Every time the robot picks up an object, it is not forced to go directly to the basket but on its way to the baskets, it can also explore other areas.*

The workflow is illustrated with the following pseudocode.

Algorithm 1 Assignment 3 workflow

```
1: bool visited[N]          ▷ visited[i]=true iff target area number i has been explored, ie. the
   classification and localization process has been completed in area number i
2:
3: bool has_brick           ▷ has_brick=true iff the robotic arm has a brick in the gripper
4:
5: Initialization:
6: visited ← FALSE
7: has_brick ← FALSE
8:
9: Execution:
10: while There are bricks in the map which have to be picked up do
11:     if has_brick == FALSE then
12:         Go to the closest target area with at least one brick or an unknown number of bricks
13:     else if has_brick == TRUE then
14:         Go to the proper basket according to the class of the brick we have in the gripper
15:         foreach intermediate target area targetArea along the path do
16:             if visited[targetArea]==FALSE do
17:                 Explore area targetArea
18:                 visited[targetArea] ← true
19: end
```

For the purpose of finding the closest destination we have implemented Dijkstra's algorithm for the minimum spanning tree computation. The time complexity of the algorithm is $O(n+m)$ with $n =$ number of nodes of the graph; $m =$ number of edges of the graph. In this direction it is important to observe that we have always few nodes and the number of them is constant.

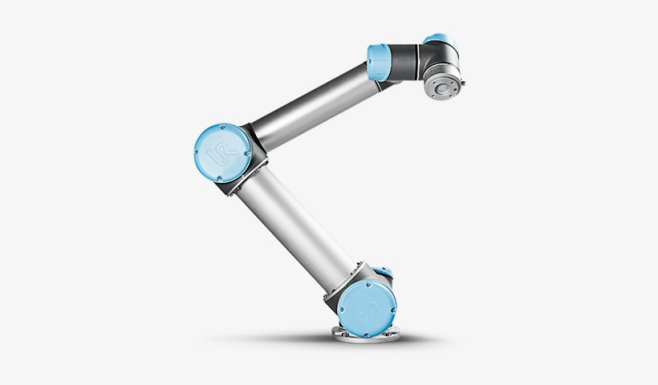
Another crucial difference in this assignment concerns the object localization process. In fact we do not approach the target area containing the bricks always in the same direction, but we can arrive to the area from every direction. The solution is explained in *Robot vision* chapter.

Chapter 3

Robotic arm

An important aspect of the project is about collecting the LEGO bricks. The purpose of this section is to present hardware and software solutions which have been adopted in order to solve this requirement.

We use the **ur5** robotic arm by *Universal Robots*, a 6 DoF manipulator with a gripper mounted as end-effector.

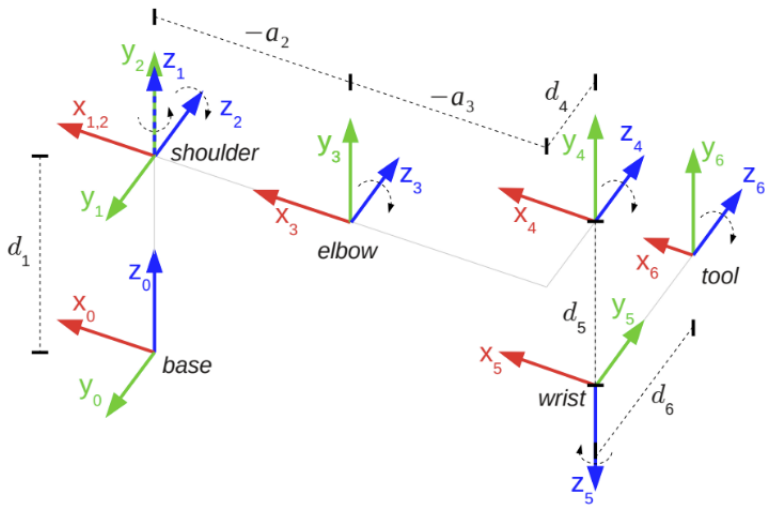


Universal Robot 6 degrees of freedom robotic arm

Robotiq gripper

3.1 Kinematics

From a mathematical point of view, we have described this structure as a sequence of reference frames as described by the following picture:



That said, the task is about moving the robot from a point to another. The motion is assigned in the operational space by means of the initial and final end-effector pose x_e. We have abstracted the latter with a class named *EndEffector*. We use kinematic inversion to find the corresponding joint space configuration q, abstracted with a class named *Ur5JointConfiguration*.
As a consequence, the goal is to move the robot between a joint configuration q_s and a joint configuration q_f.

In order to achieve this purpose we use a function $q(t)$ such that:

$$q(t_s) = q_s$$
$$q(t_f) = q_f$$

with t_s and t_f initial and final time instance respectively.

In particular we can use the following cubic polynomial:

$$q(t) = a_3t^3 + a_2t^2 + a_1t + a_0$$

The coefficients are found as a solution of a system of linear equations obtained by imposing terminal constraints on velocity and position:

$$q(t_s) = a_3t_s^3 + a_2t_s^2 + a_1t_s + a_0 = q_s$$

$$q(t_f) = a_3t_f^3 + a_2t_f^2 + a_1t_f + a_0 = q_f$$

$$q(t_s)' = 3a_3t_s^2 + 2a_2t_s + a_1 = q'_s$$

$$q(t_f)' = 3a_3t_f^2 + 2a_2t_f + a_1 = q'_f$$

For a better explanation, the whole procedure is summarized here with the following pseudocode:

Algorithm 2 End effector motion plan

Input:
    initial end effector pose xe0
    final end effector pose xef

Execution:
    qEs = inverseKinematic(xe0) ▷ Use inverse kinematic to compute initial joint configuration
    qEf = inverseKinematic(xef) ▷ Use inverse kinematic to compute final joint configuration

    foreach joint **do**
        Compute coefficient a_0, a_1, a_2, a_3

    Compute the sequence of positions (time dependent), described as joint configurations, to reach the final configuration:

    for time=minT ; time<maxT; time=time+Δt **do**
        $q(t) = a_3t^3 + a_2t^2 + a_1t + a_0$

end

Procedure for collecting a brick

In this paragraph we describe the sequence of motions that we have implemented to pick up a brick from the ground.

1. First of all we move the end-effector (with the gripper opened) over the target brick
2. Then we low the gripper to reach the brick. As we are going to explain in the following chapters, the gripper's orientation, aperture and final altitude depends on the particular brick's class and on the particular brick's position on the ground
3. Once we have reached the brick we close the gripper to grip the brick
4. Finally, we raise the arm to proceed with the subsequent tasks. When solving assignment 3, before raising the robotic arm as it has been just described in this step, it is important to first raise the arm perpendicularly in order to avoid collisions with other bricks which could be in the area.

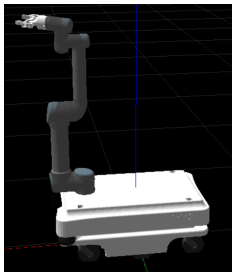
3.2 Grabbing objects

Working in a Gazebo simulation is quite different if compared to the real world, especially when speaking about inertia and gravity. To avoid unpredictable collisions, we decided to implement **dynamic linking**, meaning that once the LEGO brick has been grabbed, it can't slide away from the gripper.

Chapter 4

Mobile robot

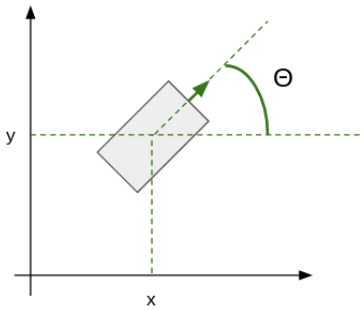
We use the **Mir100** by *Mobile Industrial Robots*.



For the `urdf` file of our robot we have taken **ur5_robotiq85_gripper.urdf** and **mir.urdf** and merged these two file together into our **mir_ur5.urdf**. We have modified the *shoulder_pan_joint* to link the *base_link* of the mir robot and the ur5 together. In addition we have mounted two cameras on our robot to perform **object classification and localization**.

4.1 Trajectory planning

The purpose of this section is to explain how we plan the trajectory that the MIR follows in order to travel from an initial configuration to a final configuration. To represent the latter we use three generalized coordinates $[x, y, \theta]$ which are convenient to make the solution of the motion easier.



Generic coordinates

With the aim of moving the vehicle between two different positions in an optimal way, we had implemented the solution which is known in the literature as "Dubins Curve Manouvers", but then we have experimented that this approach is computational demanding and we have decided that in our case there is a more convenient solution for our purposes. First of all it is important to observe that our robot can turn on the spot. That said we have implemented trajectory planning as follows. The procedure is concretized by the procedure *travel*.

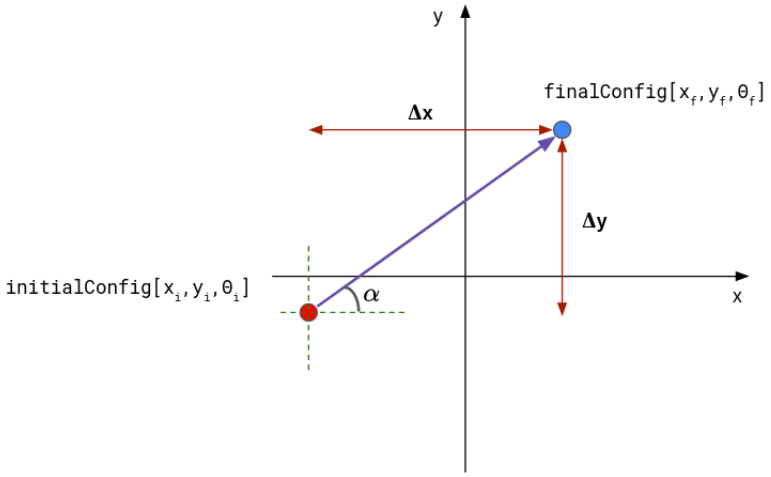
Input

- Initial configuration of the mobile robot: $[x_i, y_i, \theta_i]$
- Final configuration of the mobile robot: $[x_f, y_f, \theta_f]$

In order to achieve a better explanation, the process is divided into four steps.

STEP 1

We compute a straight trajectory which points from (x_i, y_i) to (x_f, y_f). So, the aim of this step is to calculate the trajectory angle α. The calculation is slightly different depending on coordinates (x_f, y_f) with respect to coordinates (x_i, y_i). Here we propose only one case, the others can be analogously reconstructed by the reader.

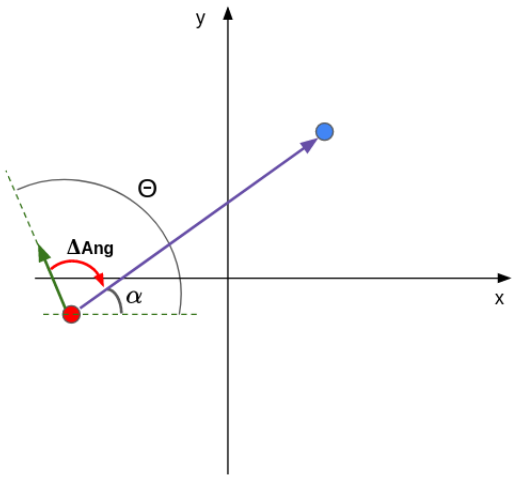


Final position is in the first quadrant with respect to a frame centered on the initial position.

$$\alpha = \arccos\left(\frac{\Delta x}{\sqrt{\Delta x^2 + \Delta y^2}}\right)$$

STEP 2

Given the angle α we rotate on the spot to align the robot with the straight trajectory. To do this we have first to calculate the angle (ΔAng) between the actual direction of the robot (θ) and the desired trajectory angle (α).



$$\Delta Ang = (\alpha - \theta + 540) mod 360 - 180$$

The formula gives the shortes rotation distance in degrees. Positive values indicate clockwise rotation, negative values indicate counterclockwise rotation.
At this point we rotate with constant angular speed ω. In this way we can use the uniform circular motion laws to compute the duration of the desired rotation:

$$t = \frac{\Delta Ang}{\omega}$$

In practice we publish angular velocity ω on the appropriate topic for t seconds.

STEP 3

After these, the robot has to move following the straight trajectory. For convenience, the robot has constant linear velocity v during the motion. To understand how much time we have to publish linear velocity v on the appropriate topic in order to reach the final position, we use rectilinear uniform motion laws.

$$t = \frac{\Delta s}{v}$$

STEP 4

Finally we have to align the robot to the final angle θ_f. To do this we proceed in the same way as explained in *Step 1*.

4.2 Trajectory control

As it happens in the real world, the robot has a dynamics and does not respond instantaneously to our commands and, as a consequence, the robot can deviate from the ideal trajectory and the error accumulates over time. As a result if we do not implement a proper trajectory control solution, at the end the robot could be far away from where we expected it to be.

In order to achieve our expectations we have decided to use Lyapunov functions to construct feedback control functions that guarantee by construction the convergence to a desired trajectory.

In Lyapunov-based control there are two relations to take into consideration:

1. A reference robot that moves according to the model:

$$p_d{}' = \begin{bmatrix} x_d{}' \\ y_d{}' \\ \theta_d{}' \end{bmatrix} = \begin{bmatrix} v_d \cos(\theta_d t) \\ v_d \sin(\theta_d t) \\ \omega_d \end{bmatrix}$$

with:
p_d desired configuration of the mobile robot;
x_d, y_d, θ_d desired generic coordinates of the mobile robot;
v_d and ω_d desired velocities of the mobile robot.

2. An actual robot with state evolution:

$$p_a{}' = \begin{bmatrix} x_a{}' \\ y_a{}' \\ \theta_a{}' \end{bmatrix} = \begin{bmatrix} v_a \cos(\theta_a t) \\ v_a \sin(\theta_a t) \\ \omega_a \end{bmatrix}$$

with:
p_a actual configuration of the mobile robot;
x_a, y_a, θ_a actual generic coordinates of the mobile robot;
v_a and ω_a actual velocities of the mobile robot.

Due to our convenient trajectory planning decisions, we can perform Lyapunov-based control only when we follow a straight trajectory to the destination.

In this way we can simply use rectilinear uniform motion laws to describe the desired motion of the mobile robot:

$$x_d = x_{0d} + (v * \cos(\theta_d)t)$$

$$y_d = y_{0d} + (v * \sin(\theta_d)t)$$

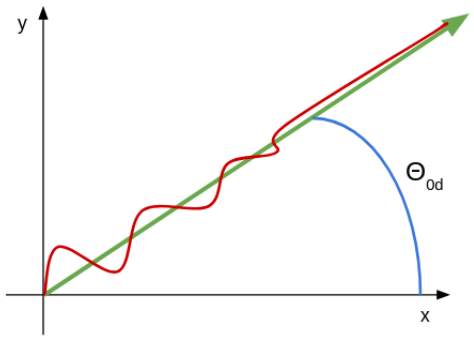
$$\theta_d = \theta_{0d}$$

$$\omega_d = 0\frac{rad}{s}$$

with:

x_{0d}, y_{0d}, θ_{0d} initial desired generic coordinates configuration of the mobile robot.

Finally, we use odometry sensors on the MIR as a hardware solution to estimate change in position over time and so the actual configuration of the robot. We read this data from the topic *base_pose_ground_truth*.



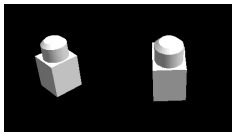
With Lyapunov based control the actual robot trajectory which is represented in red, converges to the desired straight trajectory which is in green.

Chapter 5

Robot vision

What we are required to do, right from the first assignment, is to **recognise the class** of the LEGO brick and to precisely **retrieve its position** inside the environment.

But here comes a problem: imagine having two different bricks as in the image, do they belong to the same class? Maybe, or maybe not! If only we had a second point of vantage...
So we decided to follow the saying "*two is better than one*", therefore we use two different cameras to obtain images from two different angles.



"Just checking."

In particular, a camera (from now on called the ***lower camera***) is attached on the mobile robot watching ahead, to have a near view of the objects and provide images for classification; the other camera (***upper camera***) has an *aerial view* of what stands in front of the robot, that is the fastest and **most precise** way to retrieve objects' positions.

5.1 Object localization

We recognize the object position using **OpenCV** (*Open Source Computer Vision Library*) in **Python**.
Starting from an image coming from the *upper camera*, we grayscale it and apply a lightness threshold (precisely L=50) to distinguish between objects (brought to white) and everything else (brought to black). Now it's possible to find the contours of the objects and to approximate them in order to always have a quadrilateral.

We derive the **centroid** of the object using the raw moments of the approximated contour:

$$(x, y) = \left(\frac{M_{10}}{M_{00}}, \frac{M_{01}}{M_{00}} \right)$$

Furthermore, we have to adjust the (x, y) coordinates from the camera frame to the robot frame. We need a dilation and a translation for both directions, and for y-axes we also need a preventive π rotation around the centre of the image:

$$X = \alpha_x \cdot x_0 + \delta_x$$

$$Y = \alpha_y \cdot (dim - y_0) + \delta_y$$

Where x_0 and y_0 are the coordinates in the camera frame, α_x and α_y are the dilation coefficients, δ_x and δ_y are the translation coefficients, and dim is the image dimension in pixels.

For assignment 2, we also calculate the **orientation** of the object, this time directly computing the slope of the line passing through the two consecutive vectors which are most apart one from the other (this because we want the orientation angle in respect of the longer side), and then applying $arctan$ to obtain an angle:

$$q = arctan\left(\frac{y_1 - y_2}{x_1 - x_2} \right)$$

Moreover, to adapt q to the robotic arm frame, we also need some adjustments, so that the robotic controller can take it in input and directly put it as gripper's *roll* parameter, which is in radians:

$$q' = -\left(q + \frac{\pi}{2}\right)$$

For assignment 3 we have to handle three new situations:

i. More than 1 brick per area;

ii. Bricks can be *lying*, instead of *standing* in their natural pose;

iii. The mobile robot can be rotated in any direction.

i. More than one brick per area

The solution is pretty simple and immediate: we just have to look at a brick at a time and store its features, then publish all brick's features together with the number of objects detected.

ii. Lying and standing bricks

To recognise the brick as *standing* (i.e. in its natural position) we can check if the approximated contour is a quadrilateral. In fact when LEGO bricks are lying down, they have caps, fillets, etc. that make their contour not rectangular.

iii. Arbitrary rotated robot

In order to precisely estimate the brick position in the environment, we have to apply a **homogeneous transformation** to the coordinates retrieved by the localization, from frame 1 (robot) to frame 0 (gazebo).
More precisely:

$$\tilde{\mathbf{p}}^0 = A_1^0 \tilde{\mathbf{p}}^1 \qquad \text{where } A_1^0 = \begin{bmatrix} R_1^0 & \mathbf{o_1^0} \\ \mathbf{0^T} & 1 \end{bmatrix} \qquad \text{with } R_z(\theta) = \begin{bmatrix} c_\theta & -s_\theta & 0 \\ s_\theta & c_\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

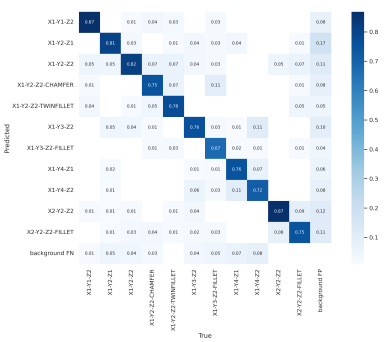
$\theta = -\theta'$, θ' is the robot orientation, and $\mathbf{o_1^0}$ is given by robot position.

5.2 Object classification

We notice that the existing 11 LEGO classes have some *quantized* dimensions (i.e. dimensions whose value can only be a multiple of a minimum *quantum*), which are X, Y, and Z; and possibly also some feature as *fillet*.
So we can take advantage of the *point-cloud* from our depth camera, in order to recognize those dimensions and features (using OpenCV).

We also use **YOLOv5**. We have trained it using the provided dataset (from prof. Sebe) an also on our own dataset (created over 1 thousand images that stick better to our scenario); after training we obtained a promising results.

In the chart is represented the **confusion matrix** between classes. It's easy to see how almost all the bricks are correctly recognized.



When we give an image to YOLO we obtain a .txt file which contains a row for every single object recognised.
A row consists of 6 space-separated numbers:

$$class \;\; x \;\; y \;\; width \;\; height \;\; confidence$$

Where x and y refer to the center of the object, *width* and *height* refer to the sizes of object's bounding box.

But we actually modified YOLO scripts in order to integrate them in our workspace. Next section is clarifying this.

5.3 Implementation solutions

As mentioned before, two **Microsoft Kinect** cameras are installed on the mobile robot.
The code implementing robot vision in our catkin workspace is stored inside the package ***robotic_vision***.

Localization

Everything is done by a single ROS node, **localize_listener.py**, which loads the image from the upper camera and publishes the objects positions and orientations on a topic: `localize`.
The structure of the message (`Localize.msg`) is as follows:

`int64 numLego` *how many bricks have been detected*

`int64 lego1_imgx` *x-coordinate of the brick's centre in the image from the upper camera*
`int64 lego1_imgy` *y-coordinate " " " " "*
`float64 lego1_x` *x-coordinate of the brick's centre in respect to robotic arm's frame*
`float64 lego1_y` *y-coordinate " " " "*
`float64 lego1_q` *orientation of the brick*
`float64 lego1_w` *width of oriented bounding box*
`float64 lego1_h` *height of oriented bounding box*
`int64 lego1_p` *pose of the brick*

... then again for `lego2` *and* `lego3` *(since in a target area there can be up to 3 bricks).*

Classification

The same **localize_listener.py** provides useful information about the brick dimensions; **mirController.cpp** subscribes to `localize` topic and handles this information together with the one coming from the *point-cloud*, so that it can discover brick's class.
We also modified the original **detect.py** of YOLO, in order to define it as a ROS node that subscribes to the lower camera `image_raw` topic and publishes results to `detect`. So the new script is called **my_detect.py**.