# Odometric Navigation

AI and Robotics - Lab 01

Giacomo Gaiani - giacomo.gaiani@studio.unibo.it

Submission Date: 27/03/2024 Submission Number: 1

# Contents

## 1 Task 1: Implementation of the ROS navigation loop

### 1.1 Problem Description

The objective of this first task is to implement the navigation loop of a simulated Tiago robot. In order to do so, it is necessary to complete four different sections of the robot control program, namely the sensors reading, the state estimation, the action decision and the relative actuation. This is done by modifying three python files provided for the laboratory activity [1]: `robot_gtw.py, observer.py` and `controller.py`

### 1.2 Objects and Data Structures

Most of the subtasks encountered in the resolution of the problem are solved by modifying the existing functions present in the python files or designing new ones. Those functions are then imported when needed in order to be used. The parameters needed for the robot initialization (wheels radius, axis and starting position) are store in a vocabulary, while the variables that are used by multiple functions in the same file are store as global variables. An example of global variables used this way are the stored values of the wheels position in the precedent control cycle.

### 1.3 Control and Information Flow

The operational structure of the program can be divided in five section. The first one is the initialization, preformed in `toplevel.py` It initialize the parameters of the gateway, observer and controller components. Once the initialization is completed the program progresses to the cyclic control operations. The second session is the sensors reading. Performed in `robot_gtw.py`, the program reads the current values of the wheels encoders using a rospy subscriber listening to the `/joint_states` topic. From the list of joint states, only the values of the wheels encoders are stored. These information are passed down to the state estimation where are used, pared with the reading of the previous cycle, to perform a relative odometric localization. The output of this procedure is a triple (*x*, *y*, *th*), where *x* and *y* is the relative position of the robot and *th* the orientation. Subsequently, the robot position is passed as input to the controller that return the values of both linear and angular velocities. At this point the controller does not perform any computation over the current position of the robot, but simply sets the velocities a constant value. The last operation is to send the velocity information to the robot. To do so, the variables are converted into the `geometry_msgs.msg.Twist` format and sent to the robot using a rospy publisher on the `mobile_base_controller/cmd_vel` topic.

## 1.4 Parameters

The parameters used in this first section of the laboratory are the wheels radius and axis, used to compute the linear and angular displacement of the robot, and the linear and angular velocities that were set to *0.1* in order to perform the initial tests.

## 1.5 Results

The first set of tests consisting in running the navigation loop with constant velocity showed coherent results. The altering of the estimated position over time seems to be consistent with the expectations.

## 1.6 Conclusions

The discussion of the more quantitative tests and the relative results is treated in the following chapters.

# 2 Task 2: Extensive tests on position estimation

## 2.1 Problem Description

The objective of the second task is to extend the tests performed on the simulated Tiago robot to verify the quality of the position estimation by means of two assessments. The first is to compere the position computed via the odometric calculations with the real position of the robot by plotting the two trajectories. The second is to perform the so called *square trip* test to check the behaviour of the robot under different aspects. This second test consists in maneuver the robot in a square path, and can be performed under normal circumstances or by manipulating the robot parameters in order to appreciate the different results.

## 2.2 Objects and Data Structures

The objects and data structures utilized in this task are the same of the previous one, extended to contain parameters relevant to this section. A new file called `utilities.py` has been included to contain various functions to ease the implementation of the other parts of the program. In addition, a list has been added to store the computed and real robot positions at each time step.

## 2.3 Control and Information Flow

The overall structure of the program is similar to the previous task, so it is easier to list the changes following the execution proggres. A new rospy subscriber has been added to read the ground truth odometry values of the robot both during the initialization and in real time. This information is used during the initialization to give the robot its starting position using the world reference frame, and it is used at runtime to compare it with the computed absolute position of the robot. In order to use this data, the rospy subscriber reads the `/ground_truth_odom` topic and convert it from quaternion into eulerian coordinates. As mentioned, the state estimation is now performed using the absolute position of the robot and passed down to the motion controller. If specified in the execution of the program, both the computed positions and the real positions are stored each time step. The controller, instead of returning constant liner and angular velocities, perform the square test. To do so it computes and sets as the current goal, a position ($x_G$, $y_G$) given the odometric information of the robot (*x*, *y*, *th*) and the relative linear and angular distance from the goal *LEN_OFFSET* and *ANGLE_OFFSET*, in this setup equal to 1 unit and 90°. The controller then set the velocities to perform separately a static rotation and a linear movement to reach the goal position. Both procedures terminates when the distance to the goal position, in term of length or angle, is fewer then a specified tolerance. The operation is performed a number of time equal to the specified parameter *N_SIDES*, initialized at 4. When the maneuver is completed, a *done* flag is raised to stop the running of the program and the trajectories of the real and computed positions are plotted in a graph.

## 2.4 Parameters

The parameter declared in this second task are: *LEN_OFFSET, ANGLE_OFFSET* and *N_SIDES* used to perform the square test, and *LINEAR_TOL* and *ANGULAR_TOL*, which are respectively the linear and angular tolerance to consider complete the respective movimentations. In addition to that, the *TopLevelLoop* class as been extended with two new input parameters: *plot_positions* to print the trajectory after the completion of the movement, and *mode* to specify the type of motion the robot has to execute.

## 2.5 Results

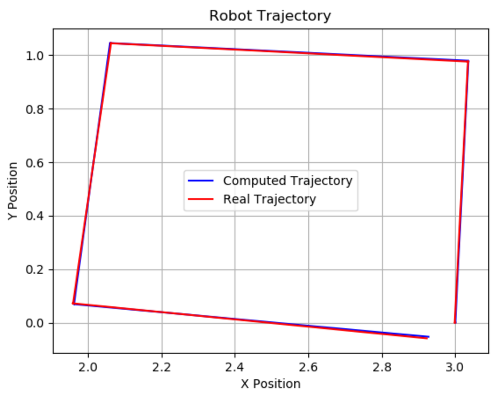
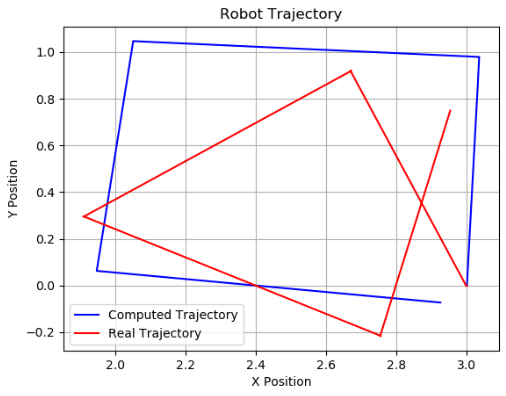


Figure 1: The square trip test under normal circumstances



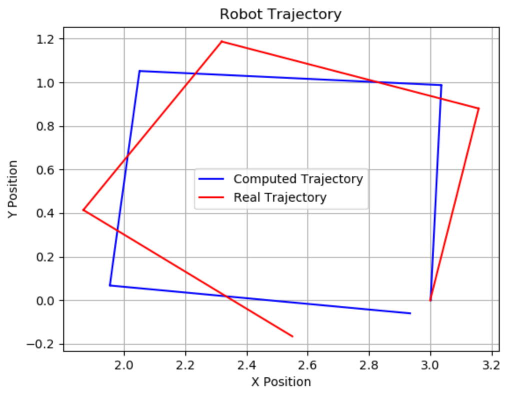


Figure 2: The square trip test with erroneus wheel axis length

Figure 3: The square trip test with erroneus wheel radius

## 2.6 Conclusions

The graph in figure1 shows the trajectory of the robot by plotting both the real and the computed positions at each time step. It is possible to see that the difference between the two trajectories is marginal, confirming a correct process in computing the odometry of the robot. However, the trajectory shows both a linear and an angular imprecision in reaching the goal positions due to the tolerances used as stopping mechanism. The value of those tolerances is lower bounded by the cycle frequency of the program. In addition, the plots in figure 2 and 3, show how the behaviour of the robot changes by artificially corrupting the robot parameters, respectively the wheel axis and radius.

# 3 Task 2: Linear control strategy implementation

## 3.1 Problem Description

The objective of this last task is to implement one or more navigation strategies to make the simulated Tiago robot reach a specified goal position.

## 3.2 Objects and Data Structures

The objects and data structures are the same of the previous task.

## 3.3 Control and Information Flow

The overall structure of the program is equal to the task 2. The only difference is the further development of the motion controller. First of all, the value of the two velocities is computed as a function of the relative distance to the goal position:

$$vlin = k_1 \cdot distance\_lin$$

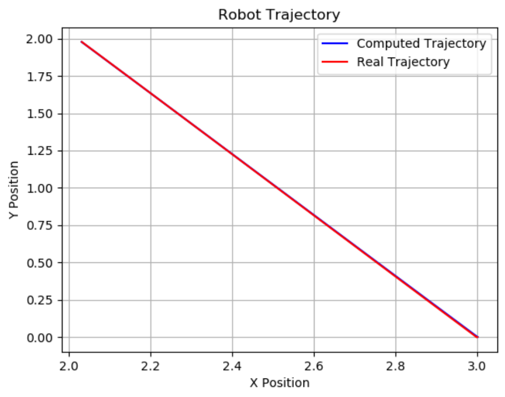
$$vlin = k_2 \cdot distance\_th$$

where $k_1$ and $k_2$ are both equal to 1.0. This method ensures smaller velocities near the goal position, giving the possibility of lowering the tolerances. The second difference is the implementation of two new controller functions. The first, named *rotate_then_move*, rotates the robot until it reaches the relative angular interval and then moves it forward to reach the goal position. The second, named *rotate_and_move*, rotate the robot until it reaches a wider angular interval, in this case 45° from the goal position, and then moves it forward to reach the goal position while still adjusting the angular velocity.

## 3.4 Parameters

The only parameter added for this task is the intermediate angular tolerance used in the *rotate_and_move* function, in this case set at 45°.

## 3.5 Results



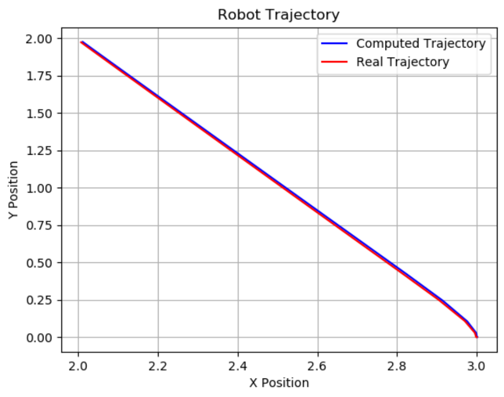


Figure 4: The robot's trajectory performing the *rotate_then_move* function

Figure 5: The robot's trajectory performing the *rotate_and_move* function

## 3.6 Conclusions

As shown in the figures 4 and 5, the behaviour of the robot is similar in both instances. The only differences are the angle to which the robot starts moving forward and the fact that, in the *rotate_and_move* case, the robot rotates while moving. In order to perform these tests, the goal position was set relatively close to the starting position of the robot. The logic behind the decision is to ease the movement of the robot: in the *rotate_then_move* case, the distance at which the goal can be set is limited by the angular tolerance. A solution for that is to either increase the linear tolerance or decrease both the angular velocity and tolerance. Another solution is to implement a way to adjust the angular position while moving forward, as seen in the *rotate_and_move* function.

## References

[1] Alessando Saffiotti. *BaseCode-2024*. 2024. URL: `https://github.com/asaffio/BaseCode-2024`.