Master CORO M1
Master Commande et Robotique
Master in Control and Robotics

EMARO+ M1
European Master on Advanced Robotics

Project Report
24/06/2020

Localization using the detection of floor tiles

Andrea Gotelli
Bianca Lento

Supervisor(s)

Gaëtan Garcia

Introduction

The aim of this project is to:

- *Develop a hybrid localization system based on the detection of floor tiles using infrared sensors*
- *Study the observability of the state in particular situations (single sensor, straight line motions...)*
- *Tune the filter and evaluate its performances*
- *Use the algorithm to perform closed loop control of a robot along long paths, thus illustrating the absence of drift*

Summay

Summay 2
The aim of the project 4
Equations of odometry 5
The limitations of odometry 7
The sensor 8
The measurements 9
A point belonging to a “horizontal” line 9
A point belonging to a “vertical” line 10
The matrices of the Kalman filter 11
The C matrix for a point belonging to a “vertical” line 11
The C matrix for a point belonging to an “horizontal” line 11
The C matrix for a point belonging to generic line 12
The intended approach 13
The Incremental Process Model 15
Feasibility Study 15
Requirements Analysis and Specifications 15
Design 15
Coding 15
Integration and System Testing 15
Maintenance 16
Ground Truth Generation and Sensor Status 17
Feasibility study 17
Requirements and specifications 17
Design 17
Rviz 18
joy_node 18
Robot node 18
Alternative solution: key_node 18
Version 1 19
Some comment 20
Version 2 21
File_handler 22
Version 3 22
Version 4 23
The estimator 25
Feasibility study 25
Observability study 25

Commenting the results 27
Requirements and specifications 27
Design 27
Version 1 28
Version 2 29
Plot analysis 30
Plots layouts 30
Inclined line 31
"Horizontal" line 33
How to avoid observability singularities 35
Squared path 37
Loop path 39
Estimator performance 41
Conclusions 43
Appendix 47
Setting up a joystick 47
Setting up PyQtGraph 48

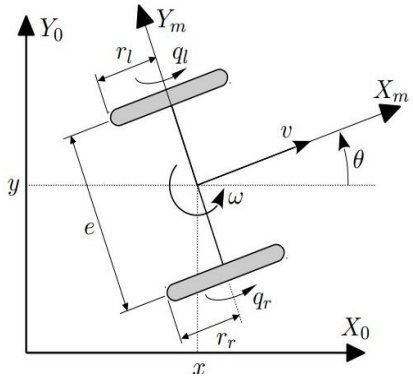
The aim of the project

The aim of this project is to develop an estimator for a mobile robot. The estimator makes use of both the intrinsic and extrinsic robot sensors. The intrinsic sensors are the wheels encoders, which provide the rotation angles for the actuated fixed wheels. The extrinsic sensors are two IR sensors, which change their state when they are over a line separating two tiles.

The robot model is presented and discussed before the explanation of the sensor's behaviour and characteristics. After the presentation of the mathematical model, the approach to develop the application will be presented and detailed. All the passages and the considerations for this software architecture will be discussed. In the end, some cases of interests are discussed. This project relies on libraries that are not build-in Ubuntu, however, the installation is explained in the appendix.

Equations of odometry

The robot is a (2, 0) robot, also known as differential drive robot, that has the two fixed wheels actuated. For this kind of robot, the general kinematic model is the one following.
The physical meaning of the variables is shown in the figure below. Where r_r andr_l are the right and left wheels radius, respectively; e stands for the robot track gauge, *i.e.* the distance between the two wheels. The robot uses encoders to measure wheels rotations. The knowledge of these rotations is used to reconstruct the robot motion.



$$\begin{cases} v_{(t)} = \dfrac{\left(r_r \dot{q}_{r(t)} + r_l \dot{q}_{l(t)}\right)}{2} \\ \omega_{(t)} = \dfrac{\left(r_r \dot{q}_{r(t)} - r_l \dot{q}_{l(t)}\right)}{e} \\ \dot{x}_{(t)} = v_{(t)} cos\left(\theta_{(t)}\right) \\ \dot{y}_{(t)} = v_{(t)} sin\left(\theta_{(t)}\right) \\ \dot{\theta}_{(t)} = \omega_{(t)} \end{cases}$$

The previous model describes the continuous time evolution of the robot. The robot posture is expressed by the state vector: $X_{(t)} = [x_{(t)} \quad y_{(t)} \quad \theta_{(t)}]^T$
The first two equation describes the input of the system: $U_{(t)} = [v_{(t)} \quad \omega_{(t)}]^T$
The last three equations express the derivative of the state: $\dot{X}_{(t)} = \left[\dot{x}_{(t)} \quad \dot{y}_{(t)} \quad \dot{\theta}_{(t)}\right]^T$.

However, the robot has to be simulated and implemented using a discrete time controller, so the previous system of equations must be discretized. When considering the discrete time equations for odometry, the continuous time equations in the kinematic model are sampled with a period $T = t_{k+1} - t_k$. The terms $\dot{q}_{r(t)}$ and $\dot{q}_{l(t)}$, here representing the right and left wheel rotation speed, change when dealing with the discrete time model. In fact, they become a difference in the wheel's rotations angle, between two-time instants. These elementary rotations of the wheels between two-time instants are:

$$\Delta q_{r,k} = q_{r,k+1} - q_{r,k} \text{ and } \Delta q_{l,k} = q_{l,k+1} - q_{l,k}$$

Considering the right wheel $q_{r,k+1}$ is the rotation measured at time $t = k + 1$ and $q_{r,k}$ is the angle measured at time $t = k$.[1] With the use of these equations, the input is defined as the elementary translation ΔD_k and the elementary rotation $\Delta\theta_k$ which expression is the following:

$$U_k = \begin{bmatrix} \Delta D_k \\ \Delta\theta_k \end{bmatrix} = \begin{bmatrix} \dfrac{\left(r_r \Delta q_{r,k} + r_l \Delta q_{r,k}\right)}{2} \\ \dfrac{\left(r_r \Delta q_{r,k} - r_l \Delta q_{r,k}\right)}{e} \end{bmatrix}$$

1 These measurements are obtained using incremental encoders, that performs a discretization of the wheel rotation in function of the encoder resolution, resulting in a systematic error. They are also read with a finite frequency.

The input could be chosen as $U_k = [\Delta q_{r,k} \quad \Delta q_{l,k}]^T$ as the two elementary rotations are the lowest level of motion for the robot, but this last decision makes the computations more complex. On the other hand, with the assumption of $U_k = [\Delta D_k \quad \Delta\theta_k]^T$, some unnecessary geometrical properties are left out of the evolution equation, resulting on the next equation:

$$X_{k+1} = \begin{bmatrix} x_{k+1} \\ y_{k+1} \\ \theta_{k+1} \end{bmatrix} = \begin{bmatrix} x_k + \Delta D_k cos(\theta_k) \\ y_k + \Delta D_k sin(\theta_k) \\ \theta_k + \Delta\theta_k \end{bmatrix} = f(X_k, U_k)$$

As the system considered is a nonlinear discrete one, it can be written in state-space form as:

$$\begin{cases} X_{k+1} = f(X_k, U_k) \\ Y_k = g(X_k) \end{cases}$$

This system defines the prediction phase: where the state and the measurement are predicted based on the current input and robot posture. The first equation is called evolution equation, the second one the observation equation, X_k is the state vector at $t = k$, $f_{(X_k, U_k)}$ is the state function, U_k is the system input at $t = k$ and Y_k is the measurement.

The limitations of odometry

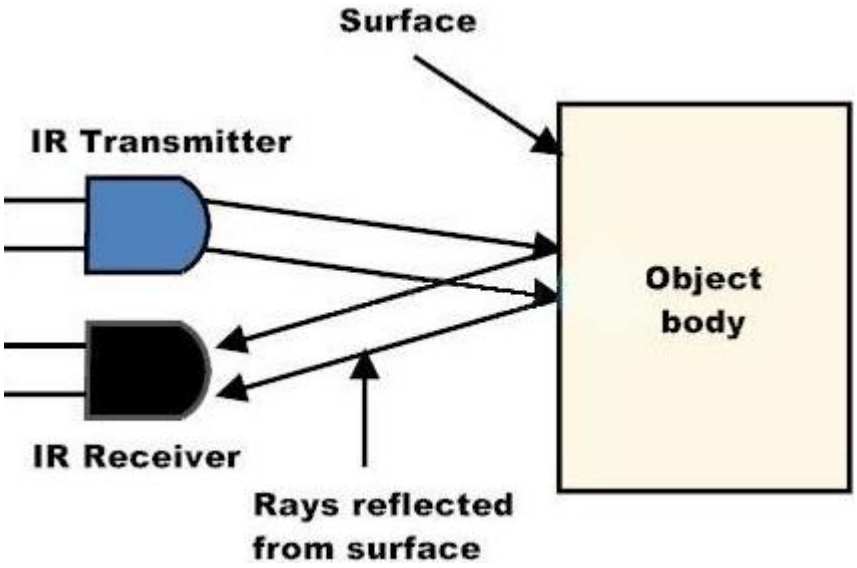
Odometry is a very efficient relative localization method[2]. One of the advantages is the possibility of implementing simple sensors, like incremental encoders, to obtain the elementary displacement of the robot. On the other hand, a robot cannot only rely on odometry. In fact, odometry suffers from high levels of systematic errors, making the localization impossible as it diverges after few meters.

So, it comes the need of an absolute localization method. This localization method is performed with respect to a fixed frame in the environment and it allows to obtain information about the robot's absolute position and orientation.
In order to extract information about its posture, the robot must assess some environment parameters. The goal here is to measure the lines separating the floor tiles. To perform such a measurement, an infrared sensor is installed in a known position of the robot. The following paragraph describes the sensor, its characteristics and how it performs the measurements.

2 In fact, odometry allows to know the actual robot position with respect to the previous one.

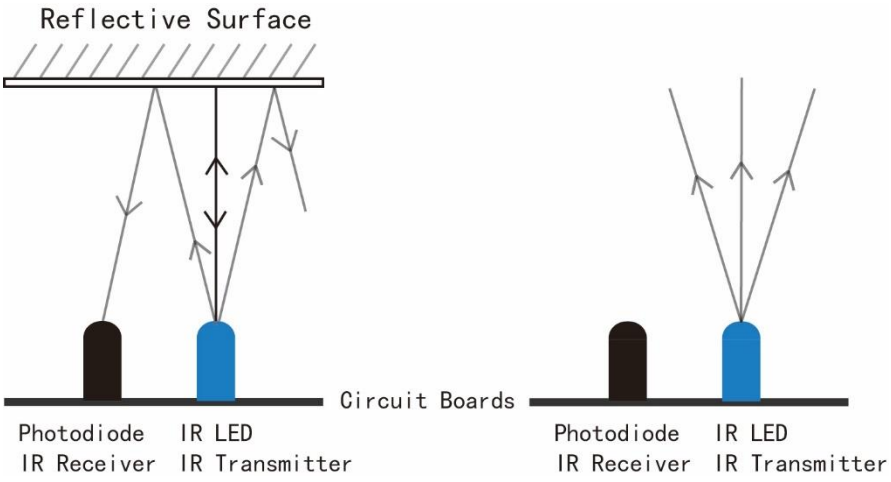
The sensor

The sensor is an IR sensor. Specifically, it works with the use of a transmitter and a receiver. The transmitter is a light emitting diode which emits infrared radiations. The receiver is an infrared photodiode which is excited only by the infrared radiations. The sensor setup consists in the implementation of both the components. The figure below briefly shows the implementation.



The object reflects the infrared radiation, in all directions, and the ones pointing to the receiver excite it. Once the receiver is excited its resistance drops and a current is measured in the circuit. In our case the sensor is facing the floor and the goal is to detect the line in between two tiles. Usually the tiles are brighter than the line separating them. As result, a distinction between the two is possible.

The sensor reacts differently if the surface hit by the infrared radiations is clear or dark.
The figure below shows the behaviour for a sensor that is facing a reflective surface (left image) and when is pointing to nothing, a far surface or a dark surface (right image).



The two cases can be referred to our implementation. In fact, the tiles, assumed to be bright or even white, reflect all the infrared radiations, resulting as reflective surfaces.
On the other hand, the line separating the tiles, assumed dark or even black, behaves as a non-reflecting surface as dark colours adsorb all the radiations. This last property leads to a behaviour similar to the case when the sensor is pointing to nothing or a very far surface; because the infrared radiation emitted by the transmitter does not excite the receiver. In summary, the sensor can be used to detect the floor tiles as it detects the lines separating the tiles.

The measurements

The robot is provided with an IR sensor like the one previously presented. The aim of this part is to show how the localization makes use of the sensor measurement.
The setup is the following: the robot is provided with an IR sensor in a known position with respect to the moving platform frame. The robot is placed in an environment where the lines separating the floor tiles are considered the beacons. While the robot is moving, the sensor may detect a beacon. To establish the beacon that has been detected, the point is mapped in the world frame. The beacon that has been detected applies a constraint on the point position.
In other words, the measurement consists in detecting a point, which coordinates are known in the robot frame, **that belongs to a known line**.
For being mapped into the world frame, the point must be expressed in homogeneous coordinates.
When a measurement happens, the robot is in the position: $X = [\mathrm{x} \quad \mathrm{y} \quad \theta]^T$ and the sensor is in the fixed position: ${}^m s = [x_s \quad y_s \quad 1]^T$, with respect to the moving platform frame. The point is mapped into the world frame with the use of the homogeneous transformation:
${}^o T_m = \begin{bmatrix} cos(\theta) & -sin(\theta) & x \\ sin(\theta) & cos(\theta) & \mathrm{y} \\ 0 & 0 & 1 \end{bmatrix}$ as usual when dealing with homogeneous coordinates. The result expression of the point in the world frame, ${}^o s = {}^o T_m \, {}^m s$ is the following:

$$ {}^o s = \begin{bmatrix} {}^o s_x \\ {}^o s_y \\ 1 \end{bmatrix} \begin{bmatrix} x_s cos(\theta) - y_s sin(\theta) + \mathrm{x} \\ x_s sin(\theta) + y_s cos(\theta) + \mathrm{y} \\ 1 \end{bmatrix} $$

The localization makes use of the knowledge of the line that contains the point to retrieve information about the robot posture. Two different cases are considered:

1. A point belonging to a “horizontal” line
2. A point belonging to a “vertical” line.

In the following, the lines separating the tiles are assumed to be known. Once the tile sides are known, or measured, the lines separating them lie with a constant offset between two consecutives. This last assumption is another key-point in this localization method.

A point belonging to a “horizontal” line

In this scenario, the point belongs to a “horizontal” line that has the equation $y = b$. As said, the line applies a constraint on the point, resulting in the following expression:

$$ {}^o s_y = x_s sin(\theta) + y_s cos(\theta) + \mathrm{y} = b $$

In this case, the measurement equation is:

$$ Y = g_y(X) = x_s sin(\theta) + y_s cos(\theta) + \mathrm{y} $$

A point belonging to a “vertical” line

In this scenario, the point belongs to a “vertical” line that has the equation $x = a$. As said, the line applies a constraint on the point, resulting in the following expression:

$$^{o}s_x = x_s cos(\theta) - y_s sin(\theta) + \mathrm{x} = \mathrm{a}$$

In this case, the measurement equation is:

$$Y = g_x(X) = x_s cos(\theta) - y_s sin(\theta) + \mathrm{x}$$

The matrices of the Kalman filter

Once the discrete time system is defined, there is the need to linearize the system. The linearization is obtained by calculating the partial derivatives of the two functions: $f(X_k, U_k)$ and $g(X_k)$, with respect to their variables. The following equations explain the linearization of $f(X_k, U_k)$.

$$A = \frac{\partial f_{(X_k,U_k)}}{\partial X}\bigg|_{X_k} \text{ and } B = \frac{\partial f_{(X_k,U_k)}}{\partial U}\bigg|_{U_k}$$

In other words, the linearization shows how the robot posture and the input affect the evolution model of the robot. The matrix A defines how the prediction phase is affected by the state vector. On the other hand, the matrix B expresses the influence of the input on this phase.
Computing the partial derivatives, the following matrices are obtained:

$$A = \frac{\partial f_{(X_k,U_k)}}{\partial X} = \begin{bmatrix} 1 & 0 & -\Delta D_k sin(\theta_k) \\ 0 & 1 & \Delta D_k cos(\theta_k) \\ 0 & 0 & 1 \end{bmatrix}$$

$$B = \frac{\partial f_{(X_k,U_k)}}{\partial U} = \begin{bmatrix} cos(\theta_k) & 0 \\ sin(\theta_k) & 0 \\ 0 & 1 \end{bmatrix}$$

For what concerns the influence of the state in the measurement, it directly comes from the consideration made in the dedicated chapter. The measurement is in function of the robot posture, but it depends also on which type of line has been detected. The general form for the linearization of the measurement function is:

$$C_v = \frac{\partial g_x(X)}{\partial X}\bigg|_X = \begin{bmatrix} \frac{\partial g_x(X)}{\partial x} & \frac{\partial g_x(X)}{\partial y} & \frac{\partial g_x(X)}{\partial \theta} \end{bmatrix}$$

$$C_h = \frac{\partial g_y(X)}{\partial X}\bigg|_X = \begin{bmatrix} \frac{\partial g_y(X)}{\partial x} & \frac{\partial g_y(X)}{\partial y} & \frac{\partial g_y(X)}{\partial \theta} \end{bmatrix}$$

Considering now the two previous cases analysed, the C matrix will indeed change in function of the detected line as it is reported below.

The C matrix for a point belonging to a “vertical” line

The following form of the matrix is the result of the partial derivative of $g(X)$ when a point in a “vertical” line has been measured.

$$C_v = [1 \quad 0 \quad -x_s sin(\theta) - y_s cos(\theta)]$$

The C matrix for a point belonging to an “horizontal” line

The following form of the matrix is the result of the partial derivative of $g(X)$ when a point in a “horizontal” line has been measured.

$$C_h = [0 \quad 1 \quad x_s cos(\theta) - y_s sin(\theta)]$$

The C matrix for a point belonging to generic line

In this case the line has the generic expression $ax + by + c = 0$. In the previous line equation, $x = a$ or $y = b$, the other coordinate, respectively y or x, does not play any role. In fact, in a line with the equation $x = a$, the coordinate y has no role in the position estimation, but, if the line equation is $ax + by + c = 0$, then the two coordinates x and y are bonded by the line equation.

The sensor when detects the line is at the position: ${}^0s = \begin{bmatrix} {}^os_x & {}^os_y & 1 \end{bmatrix}^T$. Using the line expression, this particular case is a combination of the two previously discussed. In fact, again for the property of the point to belong to the line: ${}^os_x = x$ and ${}^os_y = y$, where x and y refer to the line abscissa and ordinate. As a result, the following equation expresses the link between the two coordinates of the point:

$$a\ {}^os_x + b\ {}^os_y + c = 0$$

However, the measurement must be in the form $Y = g(X)$, that is simply obtained by rearranging the previous equation:$a\ {}^os_x + b\ {}^os_y = -c$. As a result, the measurement equation has the expression $Y = g(X) = -c = a\ {}^os_x + b\ {}^os_y$. Substituting the values of ${}^os_x$ and ${}^os_y$, and rearranging the terms, the measurement equation has the form:

$$Y = g(X) = a\mathrm{x} + b\mathrm{y} + cos(\theta)(ax_s + by_s) + sin(\theta)(bx_s - ay_s)$$

This last equation confirms the hypothesis that the measurement depends on both the coordinates of the point. The C matrix for this case is:

$$C = \left.\frac{\partial g(X)}{\partial X}\right|_X = \begin{bmatrix} \frac{\partial g(X)}{\partial x} & \frac{\partial g(X)}{\partial y} & \frac{\partial g(X)}{\partial \theta} \end{bmatrix}$$

By performing the partial derivatives, the explicit expression of the matrix is:

$$C = [a \quad b \quad -sin(\theta)(ax_s + by_s) + cos(\theta)(bx_s - ay_s)]$$

The previous can be also synthetize using the homogeneous coordinates. In fact, the line can be expressed as: $l = [a \quad b \quad c]^T$ and the sensor as: ${}^0s = \begin{bmatrix} {}^os_x & {}^os_y & 1 \end{bmatrix}^T$.

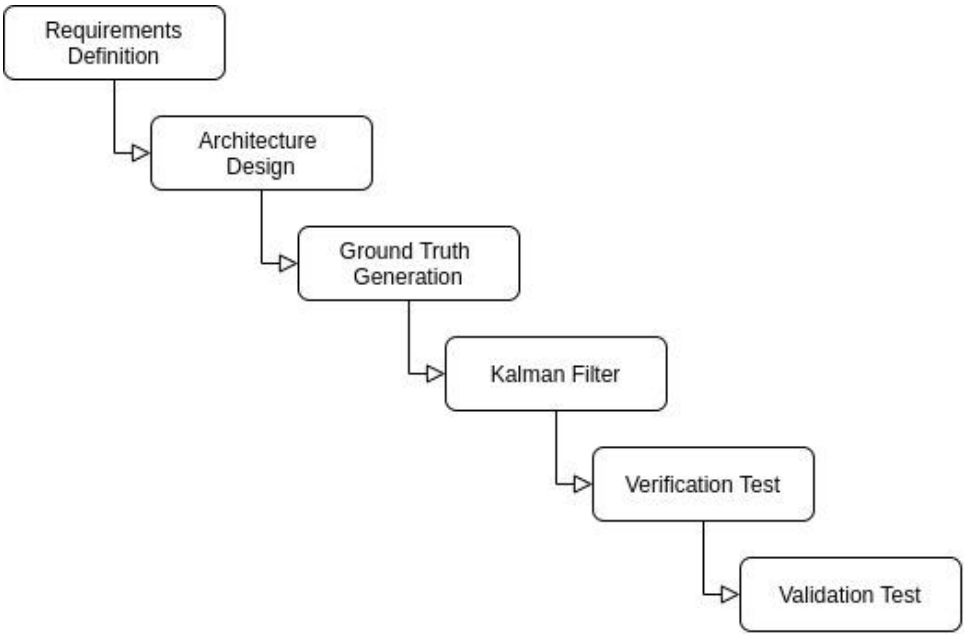
If the point belongs to the line, the dot product between the two is zero: ${}^0s \cdot l = 0$. As a result, the same expression is obtained: $a\ {}^os_x + b\ {}^os_y + c = 0$.

The intended approach

The aim of the project is to develop an estimation algorithm for a mobile robot, detecting the limit of some floor tiles. The implemented algorithm is based on a Kalman Filter. To achieve this goal, the project has to implement the discussed algorithm alongside a simulation. The simulation is for generating a ground truth and simulate the sensor status along the path.

To develop this project, the form of an Incremental process model has been chosen. The incremental model allows to interact with the supervisor and/or some customers during the all procedure. Moreover, it allows to develop several versions of the project, adding features and improving the performance, once a stable and correct version is obtained. Additionally, it allows developing different parts sequentially or in parallel.

The project can be summarized in two distinct parts: the simulation for a ground truth and sensors status followed by Kalman filter-based estimator. On virtually, these two parts could be developed in parallel, using the Parallel Development Model, allowing to save time. However, as the team is composed of only two developers, it may not be the best strategy. In fact, when there are few developers another model is recommended: The Staged Delivery Model, where the two part will be realized sequentially. The project development and phases are shown in the figure below.



This approach allows to reduce the errors that could happen during the development of the project. Additionally, it allows the users, the customers or the supervisors to evaluate the procedure as every version is accessible and controllable. In the end, the core of the application will be developed first and it will be the most tested, resulting a more stable final result. On virtually, this approach finds difficulties when the task is not well defined or when the modules which compose the project are not well established. However, in this case, the objective is well defined as well as all the modules.

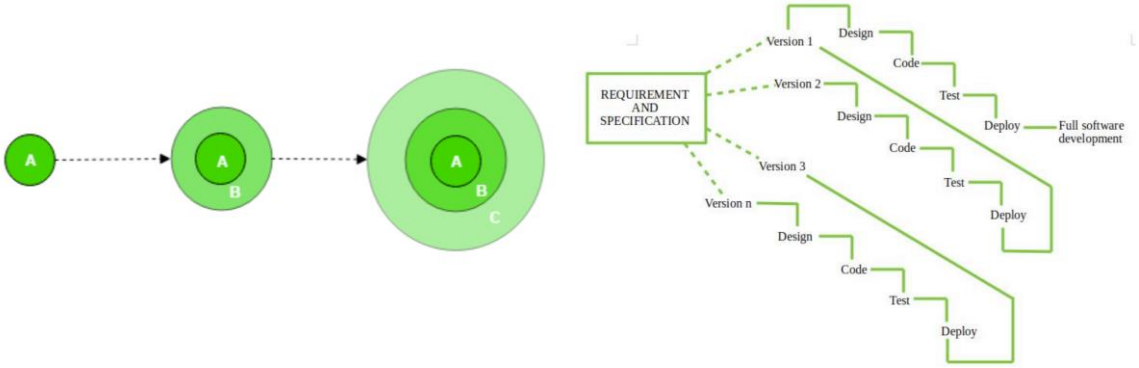
In the iterative model every loop should be develop as an iterative waterfall process. However, as the objective is very well defined, and there are no much components that could cause problem, the Incremental Process Model is proposed for each of the two loops: Ground Truth Generation and Kalman Filter.

As already discussed, the incremental model allows to integrate and add features during the development of the application. In other words, the project can start simple, performance and features will be considered later in the development. Indeed, the initial conditions and objectives must be carefully chosen, if not the result will be far from the expectations. The following are the expectations:

- The robot must be able to localize itself.
- The localization must ensure uncertainty lower or at least comparable to the tiles dimension.
- The localization must cope with the robot max speed.
- When simulating the estimator, the time required should be of a bunch of seconds.
- When generating a trajectory, the simulations should be almost instantaneous.
- When performing the simulation, the interface should be simple and user friendly.
- The estimator should be suitable to an implementation in a simulation (time speed up) and in a real time in a moving robot (so should be able to run indefinitely).

The Incremental Process Model

The Incremental Process Model is shown in the image below.



From the previous figures, it can be seen that the model is done in steps: each iteration passes through the requirements, design, coding, testing and deploying phases and each version of the system adds functions to the previous one until all designed functionality has been implemented.

UML diagrams are provided to describe in detail and in a clear way each version: in particular, a structural diagram to capture the overall system's structure and a behavioral diagram, an activity one, to specify the order in which the operations take place.

The various phases of this model will be discussed now and not further in this document, where there will be directly applied on the considered problem.

Feasibility Study

This is a preliminary study of the system; it will be discussed if the system is financially sustainable and technically feasible. Additionally, all the possible strategies, to achieve all the objectives, are discussed and the best one is chosen.

Requirements Analysis and Specifications

In this second phase, the developers may have to cope to some incompleteness or inconsistencies of the project requirements. In fact, all the specifications of the system must be listed and ranked, ordering them from the most critical and important to the less necessary. The order must be chosen carefully because the more important will be realized first, and they will represent the core of the project. This order of implementation will be saved in the Software Requirements Specifications, called the SRS document. In fact, this document is the output of this phase, and it will be a guideline for the development of the software.

Design

Here the requirements are transformed in a structure suitable to implementation. In this section every component, that in this case will be ROS nodes, must be defined in detail.

Coding

In this part the approved design has to be implement for real. This section just follows the design specifications. However, some problems may come upon, and to solve them the design could be slightly changed. On the other hand, the final result should not be too far from the design specifications.

Integration and System Testing

In this session tests will be done: firstly, each executables or node by unit testing; secondly, the system by integration testing and finally, the whole system in different environments to evaluate how the performance

differs from the ideal conditions. At the end, all the requirements must be verified and the whole system should be tested in a non-optimal situation to ensure good properties also in adversary situations.

Some efforts must be put in defining how each node could be tested. For example, testing the numerical range of values as input data the node can handle, how many messages per seconds it can cope with and in which cases it won't be able to perform all the computation in the allocated time. Additionally, the internal computations of the node should be ensured to be stable and to not cause any failure.

Maintenance

This last phase is the one more demanding in terms of time. In fact, once the application is finished, it will be tested in other environments and applications. In fact, users and costumers will use it and some hidden problem may come upon. In other words, the aim of this last phase is to ensure the project to work in different situations.

Ground Truth Generation and Sensor Status

The aim of this part is to generate the ground truth and the sensors status, along a custom trajectory, for feeding the estimator. The system specifications for this task are, in order:

- It should generate as close as possible real values.
- It should not take too much time to output a solution.
- It should be very versatile.
- The whole simulation should be as clear as possible and user friendly.
- The result should be stored somewhere and be easy to use.

Concerning the last one, a ROS package comes to help. In fact, it's just necessary to use an existing package called *rosbag.* The usage of this one will be discussed in the design section.

Feasibility study

The model can be simulated implementing the equation discussed. Once the mathematical model is implanted in an executable, the program needs a controller to move the robot and generate a path.

For the generation of the path, it has been chosen as principle solution the implementation of a joystick. The use of this remote control allows to obtain a more precise trajectory. This task can be accomplished by using several ROS tools. In fact, ROS already provides a package to handle such a hardware. A very simple and fast configuration is required before being able to use the controller, as explained in the appendix, but then the implementation is very simple.

Requirements and specifications

The requirements of this package are listed below. The order goes from the most important to the less one.

- Generate a coherent path, robot posture, encoder values and sensor states.
- The user must be free in generating a path without constraints.
- No specific hardware required.
- If possible, the simulation should run at high frequency to increase the accuracy.

The reason of this rank of system specifications comes from the following considerations.

The first specification finds its reasons in a practical point of view. The simulation itself provides data to the estimator and the provided data must be reliable. Additionally, the user should feel not constrained in the generation of the path. There should not be any problem to generate any random path. Moreover, the computations in the simulation should be as much efficient as possible, in order to run the simulation itself at high frequency. The reason to desire a high frequency is because it is directly proportional to the resolution. The less iterations are done the greater will be the displacements in between two iterations.

These requirements can be achieved by a correct definition of the robot node. Moreover, the second one is satisfied by the definition of this package as the controller allows the user to feel free in moving the robot without constraints. For what concerns the need of a specific hardware, *i.e.* the joystick, it may be a problem for a user who does not have a controller or does not know how or want to define a new adapter node.

Design

This loop consists of another package which contains all the files needed to use a joystick to move the robot in the simulation, resulting an immediate and precise path generation. The robot posture, encoder values and sensor state are saved into a file.

The design of this package is now discussed and presented. First the package makes use of different nodes, each one with a specific and well-defined function. The nodes will be listed and presented in the following.

Rviz

Used as the high-level user interface, this node is a basic and well know graphic interface embedded in ROS. The aim is to use its features to obtain an easier, more understandable, fault free and more stable interface.

joy_node

This node belongs to the Joy package, which installation and usage is discussed in its appendix. This node provides the low-level features to read the joystick state and publish it in the ROS network.

Robot node

The kinematic model, presented in the chapter of the Feasibility study, will be the core of this node. Additionally, some function will be added in order to make it possible to see the results and the robot moving in the world

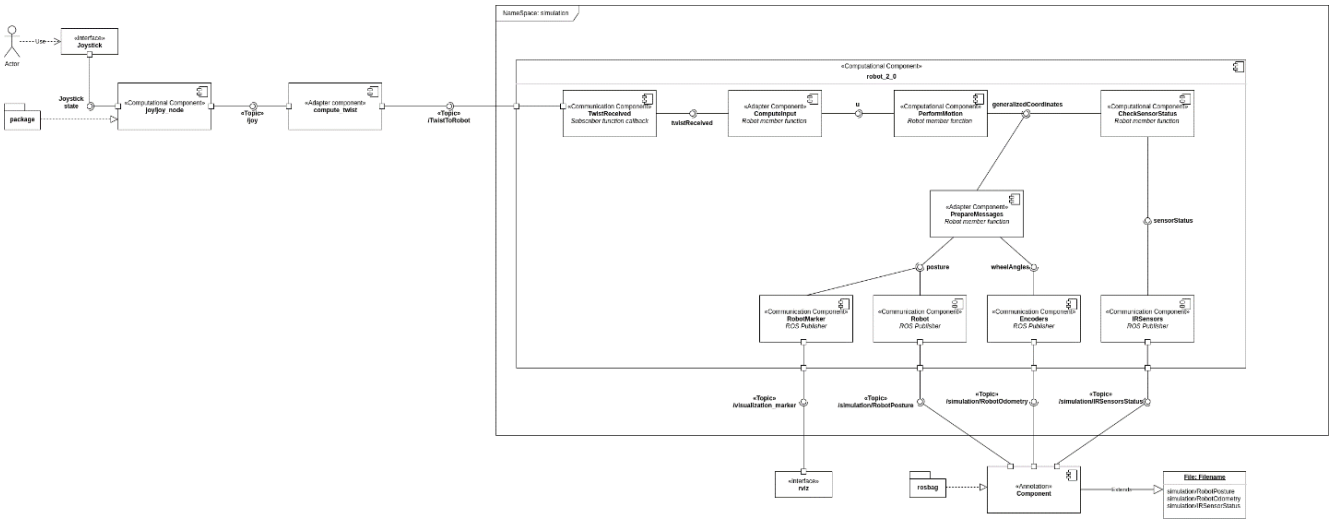
Alternative solution: key_node

An alternative solution is proposed using the keyboard, instead of the joystick to simulate the robot motion. The only part that changes in the design is that instead of the joy_node, it will be the key_node in which the simulation of the motion of the robot is controlled by some keys of the keyboard. The aim of proposing an alternative solution is to make the project accessible to everyone and give it completeness. However, this feature will not be present in the first stages of the simulation development, as it is not necessarily part of the core.

Version 1

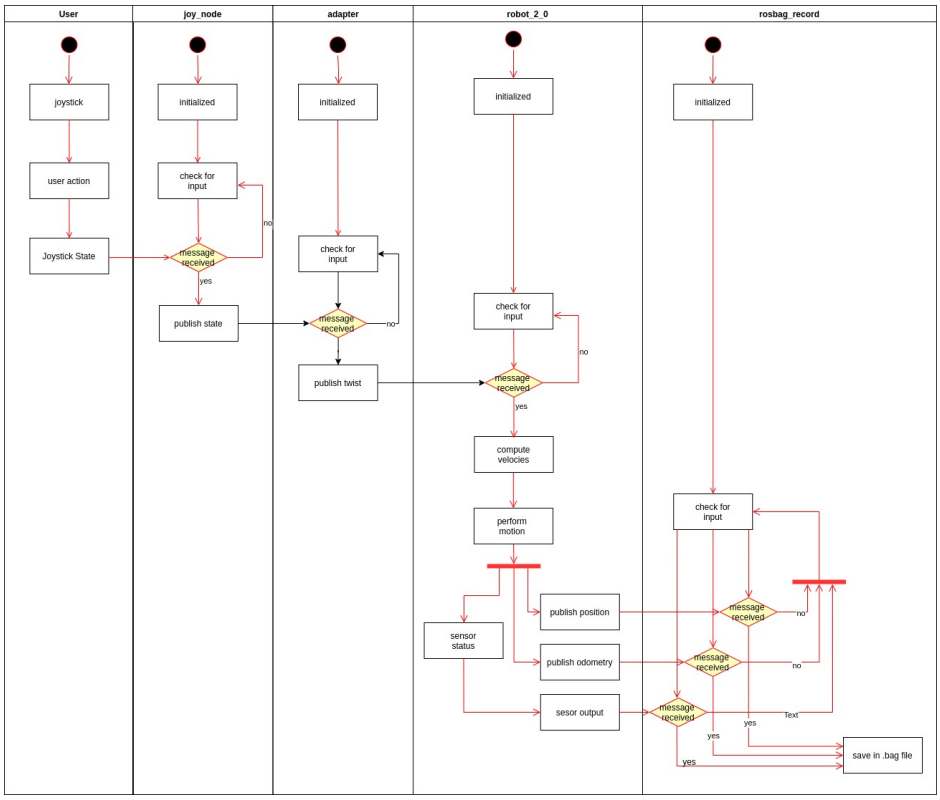
The aim of the first version is to develop the core of this task. To keep the initial work simple, the simulation will show a visualization marker moving in *rviz*.; it will be controlled using the joystick; the recording and saving will be done with a simple call of the *rosbag* record node in the launch file. Moreover, the implementation of the recording will store the file in a fixed folder, without any kind of error handling.

The structural diagram of this version is shown below:



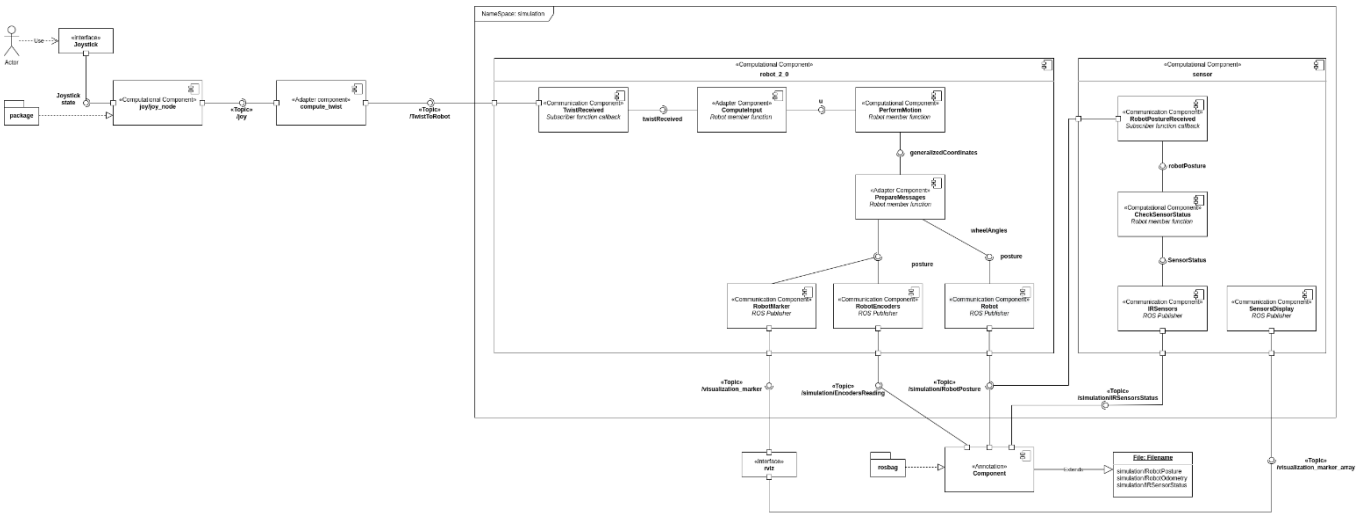
Whenever the user simulates the robot motion with the joystick, this one sends information about its state to a component called joy/joy_node. Since these data are not compatible with the one that the robot system should receive, a component adapter has been implemented. In this way, all the details of the simulation arrive to the system in the topic /TwistToRobot. The system, a ROS node called robot_2_0, is constituted by different component to achieve all the tasks required. Firstly, the input data are computed and adapted by the TwistReceived and ComputeInput component for the next computations, made by the PerformMotion function that allows to obtain the generalizedCoordinates. These data are used to retrieve information about the sensors state by the CheckSensorStatus and about the robot pose and the odometry by the PrepareMessage component. All the data computed, the odometry, the robot pose and the sensors state, are then recorded through a specific component and saved using the ROS package rosbag, in different files.

Once defined the structure of the system, to give more details about the process the activity diagram is shown below:



Some comment

Our approach is to develop the first version of the project focusing on robustness and simplicity as the aim is to create the core of the simulation. Based on the design provided, the ROS nodes are coded and tested step by step. Some changes are made in the design part to satisfy in a better way the requirements and specifications, as is shown in the figure below:



The developing of this part starts with two components: the adapter one, to allow the simulation per se, and the robot. As this one is composed by different parts, firstly the core is implemented and then the other functions and features are added.

In order to make the code more readable, two header files are created: the robot_base to embed all the ROS related functions, *i.e.* all the publishers, subscribers and operations; and the robot_2_0, to get all the

information regarding the robot position and velocity. The idea is to separate what is related to the kinematics of the robot and what is needed to make the simulation to work.

For all the computations related to the kinematic model, the Eigen library is used as it is useful for matrix and vector computations and a structure called generalized coordinates, perfectly suitable with this type of computations.

Once the program to simulate the robot was finished, it was the time to develop a simulation for the sensor. However, with the design that was proposed the program would be too much complicated. In fact, the sensor must have an instance of the world and the robot position. Even if this is feasible in C++, with the use of pointers and some class inheritances, it is simply more convenient to slightly change the design and use a specific node to simulate the sensor behaviour. At the beginning, the implementation of a ROS Service was the first choice as possible solution, but during the development it turned out that was better to use a common publisher/subscriber interface. In fact, in practice, the robot goes on its path and the sensor works by its own, this decoupling is possible with the selected interface. Moreover, a marker is added to highlight when a sensor passes on a line, so when it is active. Since the robot implemented has two sensors, the colour of the marker is different according to the sensor (green and red respectively left and right sensor).

Creating the sensor as a component apart has different advantages: in fact, it allows to change easily the position of sensors, as it is passed as parameters, and the type itself. This makes the project much more versatile and suitable for different intents.

At the end, the annotation component is implemented. The aim of this part is to record and save all the data, using the ROS package *rosbag*, as already mentioned before. It saves in three different file the robot posture, the odometry and the sensor status.

All the functions regarding the markers and the User Defined DataTypes needed are implemented in a header file, called utility.

Regarding the odometry, in this first version it is just computed without plotting it on rviz.

Once finished the development of the first version some drawbacks were notice. Firstly, the sensors are not so accurate as some measurements are not taking into account. Then, the simulation is too slow after some markers are positioned and stops to work after running for a certain interval of time. However, the time is sufficient to get all the data needed so, even if it is an issue, it can be solved later. Another drawback is that in the encoders message the information are not updated properly.

Version 2

In this second version, keeping the same design with some a posteriori insight, it is possible to make some adjustments and modifications to the project. Specifically, the requirements that want to be achieved are:

- Use the *tf* instead of markers
- Implement an URDF for the robot
- Improve the performance
- Avoid overwriting of the data, improving the saving process
- Make code more readable

In order to make the code more readable the header files of the robot are organized in a different way. Instead of two files, three are implemented in this version: the robot_base as before, the robot_2_0_generalizedcoord, that contains the whole structure for the computations of the kinematic model, and the robot_2_0, that contains all the functions related to the simulation of the robot motion. This organization is way clearer and makes the code more understandable.

In the utility header file, some functions are modified in order to have a more readable code, using tf and templates that are useful when you have to generalize efficiently computation over a set of type with similar properties.

In this version of the project, the URDF is also implemented, but without the using of *xacro*. The visual model is built as the standard configuration of a robot (2,0) with two fixed wheels and a castor one.

To improve the performance of the simulation, a better way of ensuring the max speed is implemented. Correcting this error, the robot moves slower and some drawbacks of this first version are solved. The sensor is more accurate, all the markers are positioned, and the information are updated in a more effective and appropriate way in the proper message.

File_handler

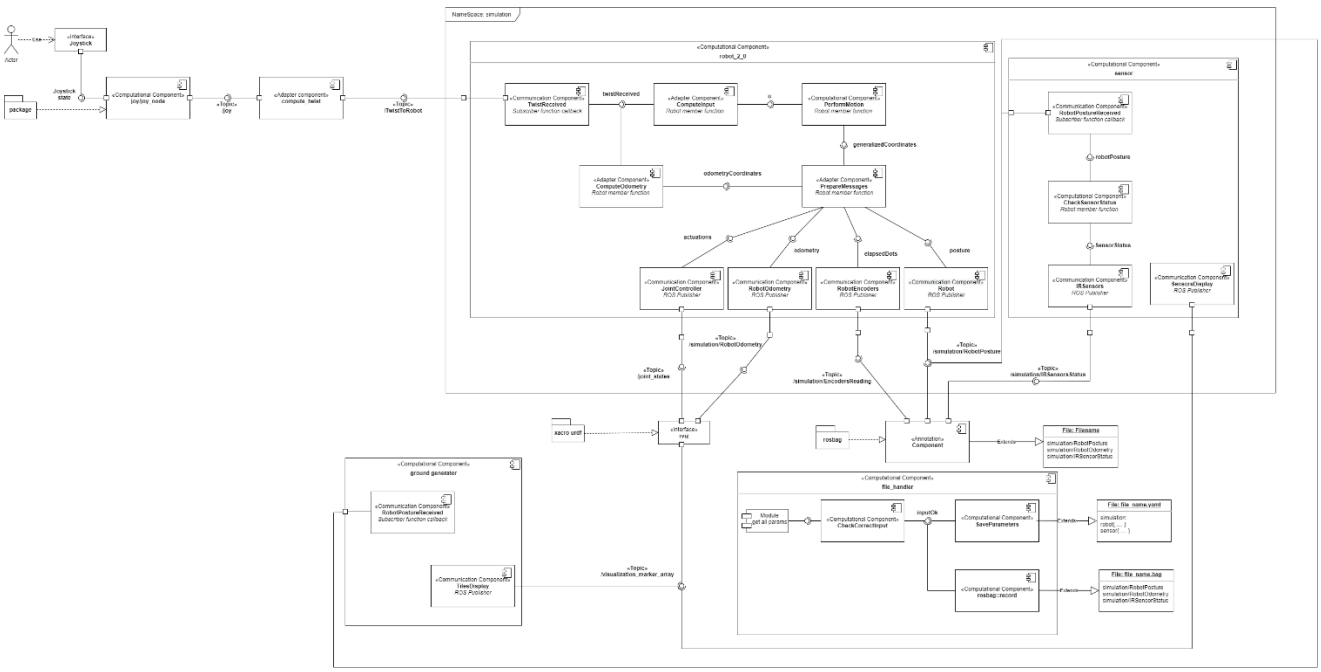
A new executable called file_handler was created. The aim of this file is to save all the recorded data in the proper folder. Moreover, it has a versatile architecture. Some parameters can be changed, based on the specification of the project. In this way, the details of the robot, of the world and of the sensor are always visible and available. In fact, the main goal of this node is to ensure a safe saving for the rosbag, containing the simulation data, and the YAML file, containing the simulation parameters.

Version 3

In this version, some adjustments and modifications are made, in particular:

- Plot the odometry
- Add the keyboard control
- Implement the URDF using *xacro*
- Add the initial position of the robot
- Handle the sensor error
- Save more file of the same type (file handler)
- Add the ground generator

As this version is quite complete, an exhaustive UML diagram is shown below:



Even if it isn't shown in the diagram, in the project a node is added to control the simulation using the keyboard. The user can move the robot in all the directions. This node is implemented to handle both keyboards, qwerty and azerty, as they have different disposition of the keys.

Moreover, the robot model is built in this version as URDF using *xacro* as it allows to parametrize the all design. In this way, it is more versatile. In fact, the model changes accordingly with the settings in the launch file. However, the relation between some of them are hidden and shouldn't be changed, they do not play any role in the simulation, but are for ensure a suitable graphic for the model.

Besides, using *xacro* gives the opportunity of modelling the wheels in a more accurate way as they can be designed in *Autocad* and then included in the model.

An error handling feature is included in the sensor part: if the user introduces two sensors that do not belong to the same world the simulation will be shut down.

Version 4

In this version, that is the last one for this part of the project, the following adjustments and modifications are made:

- Introduce errors in wheel radius and track gauge
- Adjust plot of odometry
- Publish the velocities
- Write the README.md file
- Plot of chunk just for 5 sec
- Add a timer for the marker of the generated path
- Adjust the file handler for the new functionalities

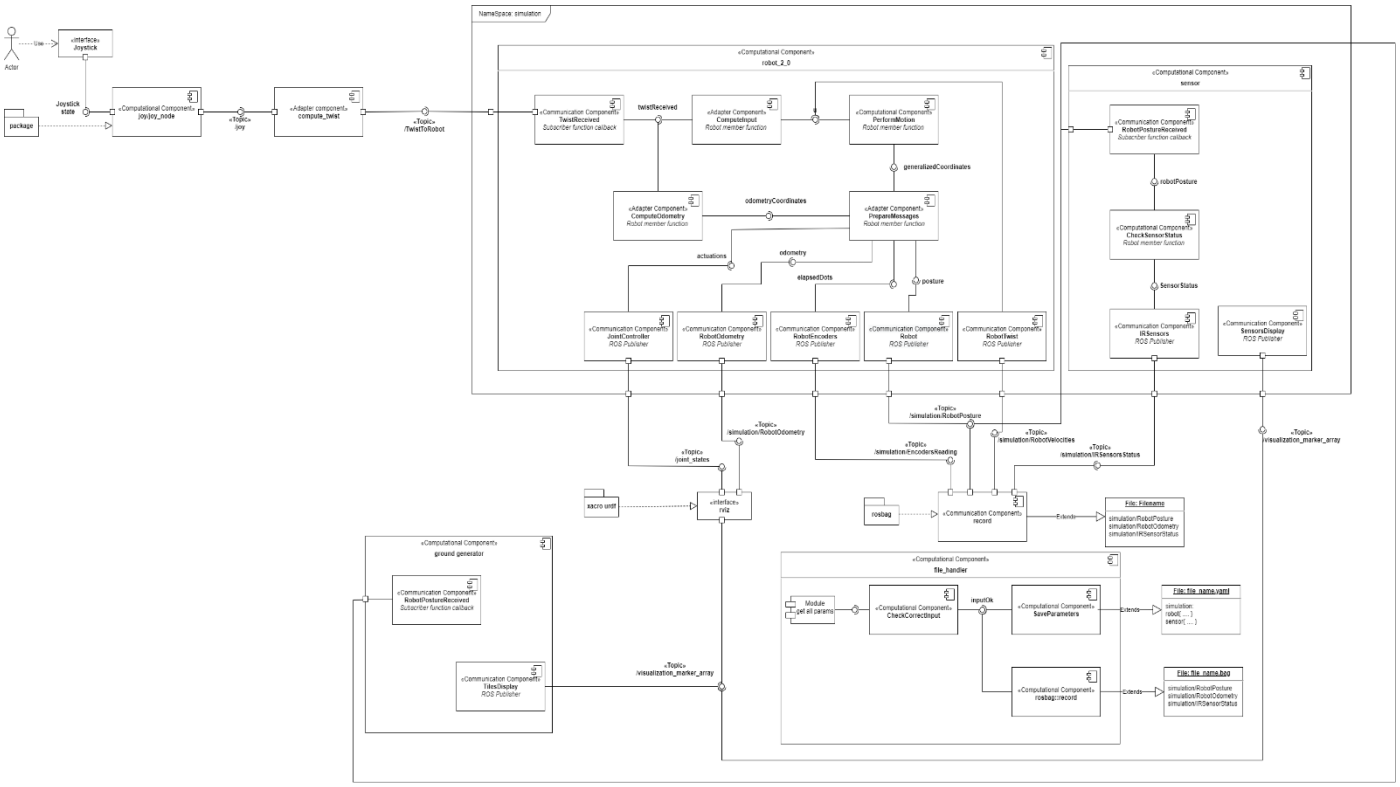
Until now, the simulation is ideal and as the aim of this project is to apply it on a real robot, so in the real world, it is necessary to introduce errors in the model. In this case they are related to the geometry of the robot. In particular, an error on the radius of one of the fixed wheels and on the track-gauge is added. They are passed as parameters so in this way the user can decrease or increase depending on the conditions of the situation that it's being simulated. However, an error of 2% can already influence in a significant way the estimated trajectory. As a result, the odometry plot differs from the real one according to the introduced errors.

Moreover, in this version a README.md file is uploaded in the GitHub repository of the project in order to make the whole project more understandable for the user, giving a general idea of what is the aim of each part and how it should be used.

As *rviz* was having some issues related to the fact that too many markers were continuously published, a timer is implemented in the robot_base.h file to publish the marker and update the path every 0.5 seconds and not at the simulation frequency. Moreover, the life duration of the marker published by the sensors when they became actives, is set as 60 seconds. In this way, the simulation performances are improved and *rviz* works smoother and without memory leaks. Moreover, the ground generator, that is a component only publishing markers to visualize the tiled floor, was modified to stop publishing after a few seconds. In fact, as the floor is static, the markers are only published in the first five seconds of simulation, in this way, some time is ensured to allow *rviz* displaying the markers, but then they no longer need an update.

Finally, the file handler was modified in order to make it possible to implement in various situations. In fact, it was provided of the possibility to take the topic to save with the *rosbag* as parameter and the possibility to choose if to save the parameters too.

The UML of the final version of the simulation is shown below:


NameSpace: simulation
Actor
Use
«Interface»
Joystick
Joystick
state
package
«Computational Component»
joy/joy_node
«Topic»
joy
«Adapter component»
compute_twist
«Topic»
/TwistToRobot
«Computational Component»
robot_2_0
«Communication Component»
TwistReceived
Subscriber function callback
twistReceived
«Adapter Component»
ComputeInput
Robot member function
«Computational Component»
PerformMotion
Robot member function
generalizedCoordinates
«Adapter Component»
ComputeOdometry
Robot member function
odometryCoordinates
«Adapter Component»
PrepareMessages
Robot member function
actuations
odometry
elapsedDots
posture
«Communication Component»
JointController
ROS Publisher
«Communication Component»
RobotOdometry
ROS Publisher
«Communication Component»
RobotEncoders
ROS Publisher
«Communication Component»
Robot
ROS Publisher
«Communication Component»
RobotTwist
ROS Publisher
«Computational Component»
sensor
«Communication Component»
RobotPostureReceived
Subscriber function callback
robotPosture
«Computational Component»
CheckSensorStatus
Robot member function
sensorStatus
«Communication Component»
IRSensors
ROS Publisher
«Communication Component»
SensorsDisplay
ROS Publisher
«Topic»
/simulation/RobotPosture
«Topic»
/simulation/RobotOdometry
«Topic»
/simulation/EncodersReading
«Topic»
/simulation/RobotVelocities
«Topic»
/simulation/IRSensorsStatus
«Topic»
/visualization_marker_array
«Topic»
/joint_states
xacro until
«Interface»
rviz
rosbag
«Communication Component»
record
Extends
File: Filename
simulation/RobotPosture
simulation/RobotOdometry
simulation/IRSensorStatus
«Computational Component»
ground generator
«Communication Component»
RobotPostureReceived
Subscriber function callback
«Communication Component»
TilesDisplay
ROS Publisher
«Topic»
/visualization_marker_array
«Computational Component»
file_handler
Module
get all params
«Computational Component»
CheckCorrectInput
inputOk
«Computational Component»
SaveParameters
Extends
File: file_name.yaml
simulation:
robot[...]
sensor[...]
«Computational Component»
rosbag::record
Extends
File: file_name.bag
simulation/RobotPosture
simulation/RobotOdometry
simulation/IRSensorStatus

The estimator

Feasibility study

The estimator makes use of the robot intrinsic and extrinsic sensors. The intrinsic sensors are the encoders and, as discussed in the related chapter, they are not sufficient to ensure a robust localization system. As a result, the estimator uses also the extrinsic sensors, that are the infrared sensors previously described and discussed. In other words, it uses both the wheels angles and the measurements of the lines to locate the robot with respect to the reference frame.

The simulation has proven to correctly simulate the sensor behavior, in a way that each time a line is crossed by the related sensor, it changes its state. However, in order to be sure that it is possible to use the intended measurement to obtain a robust localization algorithm, an observability test is presented.

Using the knowledge about the measurements, the state vector and the concept of Lie derivatives, the convergence of the observer can be guaranteed. Additionally, the situations where the estimator may not converge can be determined.

Observability study

With the use of the measurements, it is possible to build the observability matrix. All the calculations are to be based on the continuous system, in order to use the continuous equations. The results of the chapter dedicated to the sensor measurements are listed below.

A measurement of a "vertical" line: $g_x(X) = x_s cos(\theta) - y_s sin(\theta) + \mathrm{x}$.

A measurement of a "horizontal" line: $g_y(X) = x_s sin(\theta) + y_s cos(\theta) + \mathrm{y}$.

The observability matrix is computed with the use of the Lie derivatives, which are defined as follows:

$$L_f g = \frac{\partial g}{\partial X} \cdot f_{(X,U)}$$

The reason in using the Lie derivatives is in the term $f_{(X,U)}$ which allows to consider the evolution model in the observability matrix. As a result, the input and posture of the robot play a role in defining the observability. The observability for the implemented model matrix is defined as follows:

$$O = \begin{bmatrix} dg_x & dg_y & dL_f g_x & dL_f g_y & dL_f^2 g_x & dL_f^2 g_x \end{bmatrix}$$

However, to prove the observability is sufficient that the rank of the observability matrix is equal to the dimension of the state vector. For a $(2,0)$ robot, the state vector is: $X^T = [x \quad y \quad \theta]$. As it has dimension equal to three, it is sufficient to prove that the matrix has a rank of this dimension. So, it is not necessary to develop the high order Lie derivatives.

The first step is to compute the gradient and the partial derivative of the measurement equations:

$$\left[\frac{dg_x}{dX}\right]^T = \frac{\partial g_x}{\partial X} = \begin{bmatrix} 1 & 0 & -\big(x_s sin(\theta) + y_s cos(\theta)\big) \end{bmatrix}$$

$$\left[\frac{dg_y}{dX}\right]^T = \frac{\partial g_y}{\partial X} = \begin{bmatrix} 0 & 1 & \big(x_s cos(\theta) - y_s sin(\theta)\big) \end{bmatrix}$$

And then the corresponding first order Lie derivatives:

$$L_f g_x = \frac{\partial g_x}{\partial X} f_{(X,U)} = \begin{bmatrix} 1 & 0 & -\big(x_s sin(\theta) + y_s cos(\theta)\big) \end{bmatrix} \cdot \begin{bmatrix} v cos(\theta) \\ v sin(\theta) \\ \omega \end{bmatrix}$$

$$L_f g_y = \frac{\partial g_y}{\partial X} f_{(X,U)} = \begin{bmatrix} 0 & 1 & \left(x_s cos(\theta) - y_s sin(\theta)\right) \end{bmatrix} \cdot \begin{bmatrix} vcos(\theta) \\ vsin(\theta) \\ \omega \end{bmatrix}$$

The previous equations have the following results:

$$L_f g_x = vcos(\theta) - \omega\left(x_s sin(\theta) + y_s cos(\theta)\right)$$
$$L_f g_y = vsin(\theta) + \omega\left(x_s cos(\theta) - y_s sin(\theta)\right)$$

The gradient of the Lie derivatives results in the following expressions:

$$\left[dL_f g_x\right]^T = \begin{bmatrix} 0 & 0 & -vsin(\theta) - \omega\left(x_s cos(\theta) - y_s sin(\theta)\right) \end{bmatrix}$$
$$\left[dL_f g_y\right]^T = \begin{bmatrix} 0 & 0 & vcos(\theta) - \omega\left(x_s sin(\theta) + y_s cos(\theta)\right) \end{bmatrix}$$

The observability matrix becomes:

$$O = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ -\left(x_s sin(\theta) + y_s cos(\theta)\right) & \left(x_s cos(\theta) - y_s sin(\theta)\right) & -vsin(\theta) - \omega\left(x_s cos(\theta) - y_s sin(\theta)\right) & vcos(\theta) - \omega\left(x_s sin(\theta) + y_s cos(\theta)\right) \end{bmatrix}$$

To check the rank, a 3x3 matrix is extracted, taking the first three columns.

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -\left(x_s sin(\theta) + y_s cos(\theta)\right) & \left(x_s cos(\theta) - y_s sin(\theta)\right) & -vsin(\theta) - \omega\left(x_s cos(\theta) - y_s sin(\theta)\right) \end{bmatrix}$$

For this matrix, the rank is ensured to be three if the term: $-vsin(\theta) - \omega\left(x_s cos(\theta) - y_s sin(\theta)\right)$ is not equal to zero. However, for this condition to apply, the only solution is to have a static robot, *i.e.* having $v = 0$ and $\omega = 0$. Any other condition does not represent a problem. In fact, one may point out that, if $v = 0$ and $x_s cos(\theta) - y_s sin(\theta) = 0$ then the configuration is an observability singularity. However, this problem is compensated by the other column. In fact, for the two columns being equal to zero the following system should be solved:

$$\begin{cases} x_s cos(\theta) - y_s sin(\theta) = 0 \\ x_s sin(\theta) + y_s cos(\theta) = 0 \end{cases}$$

The only solution for having both equations equal to zero is to place the sensor in the centre of the moving platform frame *i.e.* having $x_s = y_s = 0$[3].

[3] Practically speaking, this result does not come completely out of the blue. If the sensor is in the centre of the moving platform frame and the robot only rotates along the z axis of this frame, the fact that the sensor is measuring a line does not give much significant information. The actual robot heading is undeterminable.

Commenting the results

The system has been proved of being observable in the vast majority of situations. Meaning that, knowing the output (the measurements) and the input of the system, it is possible to reconstruct the state vector. This is sufficient to guarantee that the developing of an estimator is a feasible achievement. However, the observability study relies on two hypotheses:

- Both the “horizontal” and “vertical” lines are measured.
- The measurements are always available

These hypotheses are not satisfied in the simulation. However, the result of the observability study is still of interest in the intended application. For example, the fact that the system is not observable if the robot stays put, means that the estimator must be turned off when the robot is not moving. Unless, the error will diverge, and the localization may fail.

In the end, the observability condition does not ensure that every observer will converge, even if it points out what are the situations where an observer will not converge.

Requirements and specifications

The estimator has a few requirements, mostly because it doesn’t have a complex structure and it does a defined and precise task: estimate the state vector with a Kalman filter. The requirements are:

- It must embed a R.O.S. interface to be versatile in multiple situations.
- It must be accurately tuned to ensure satisfactory performance even in demanding situations.
- It should be tested offline, but the intention is to tune it to be used in a closed loop control.
- It should save the results to be always available.
- When tuning the filter, the estimation should run fast, in order to reduce the computational time.

The estimator will be built to receive the output of a *rosbag* playing the recorded messages of a simulation. However, it can be implemented in a context where the sensors outputs come from a real robot, if the correct topic names are chosen (or remapped). To ensure performances, the estimator is tuned offline. The estimator publishes all the important results of each iteration, that are recorded in a *bagfile* and saved in an appropriate folder.

In the first version of this package, the goal will be to build the structure and ensure that the estimator works in a very simple application, which is easier to debug. Further versions of the package will implement the possibility to reduce the estimator frequency, in order to test it in more adverse conditions, and to decrease the overall computational time.

Design

To implement the functions to simulate the sensor behaviour, the simplest choice was to implement some other member functions in the already existing class for the sensor, which was used in the scope of the simulation. In fact, what is to be implemented in the estimator is slightly different from what is already available in the simulation. As a matter of fact, it is better to reuse existing code, with some ad-hoc modifications, instead of implementing from scratch something close to what was done before. Moreover, the already existing class for the sensor was significantly tested during the whole development of the simulation package so, it is guaranteed of being performing, correct, and stable.

In order to visualize the results of the estimator, an executable is developed specifically for the task. The aim is to separate what is the estimator itself and what is needed as graphic interface to visualize the performances and tune the filter. In fact, there are no reason to put everything in one file. When implementing on a real robot, the estimator should be tuned and run the only the code needed to perform the estimation, and nothing else.

This executable is implemented as a python script. The reason behind this change of language is because in the scope of plotting some recorded data, performance at run time are not of interest, as the executable

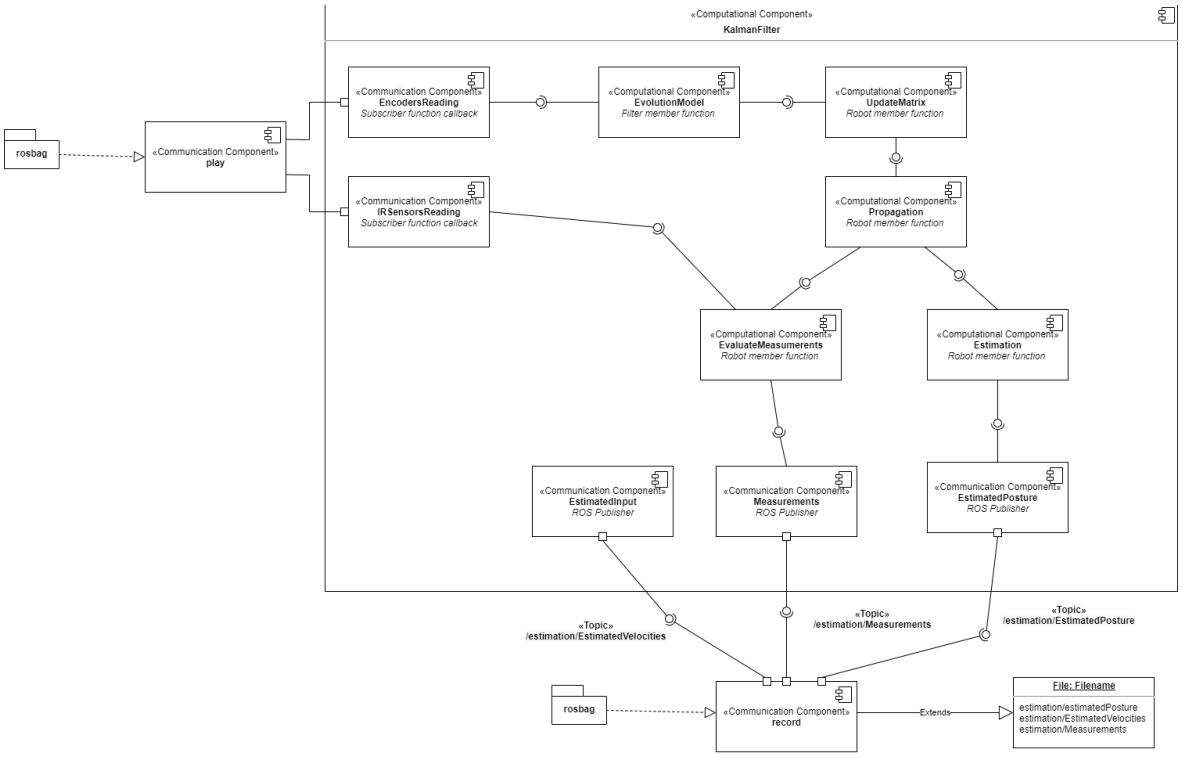
runs only once. What is of interest here is to implement the executable as simple as possible and the best choice is python. It allows to load the *rosbag* with the desired files and loop inside the messages, which is the fastest and safest way to collect all the data. Moreover, it is possible to use *PyQtGraph*, a toolbox which provides a simple but versatile graphic interface, allowing the creation of accurate plots. As this package consist of only one basic executable, the is no need of a dedicated chapter; it will be included in this one.

Another reason, less important but still quite relevant, for the choice of this toolbox is that it is available also in the newest version of Ubuntu with python3. In this way, the user does not have to adapt our project to his computer.

Version 1

The aim of the first version is to develop the core of this task. To keep the initial work simple, just the estimation and some plots will be developed. Taking the data from the simulation, it will perform the filter and process the data. As said in the design part, a python node is implemented to plot and compare the results. Three different windows are plotted. In the first one there is the Mahalanobis distances and the comparison of the three trajectories, the real one, the one estimated using the odometry and the one estimated using the filter. It also shows where the measurements are done. In the second window, instead, the error in x, y and theta with the respective range of +/-3 sigma is plotted. In the last window, there are the three plots of the standard deviations with the respect of time.

The structural diagram of this version is shown below:



Version 2

In this version, some adjustments and modifications are made, in particular:

- Tune of sigma tuning
- Add possibility to change frequency of estimation and simulation
- Speed up in the estimator
- Add the plot of the velocities
- Add table of parameters

The first thing that is done in this version is to tune the sigma tuning. In order to do that, different paths with different conditions were estimated and plotted until it was found the correct value of this parameter.

Once sigma tuning is set, other features are added to satisfy all the requirements and specifications and to give more completeness to the project. Firstly, in order to test the simulation in worst conditions, a parameter that changes the frequency of the estimation is added, that has the function to run the estimator at a frequency n-th times lower than the simulation frequency.

Secondly, it is implemented the possibility to change the rate at which the *rosbag* publishes the data and the rate of the estimator accordingly. This allows to speed out the first runs when start tuning the filter. However, for a finest tuning, it is recommended not to use this parameter with a high multiplication ratio.

Moreover, a new plot is added with the velocities, both the ones of the simulation and the ones estimated.

In order to know and check which values are set for the parameters, a table is added as first window when the results are plotted. This clearly helps the user to be more aware of what are the conditions of the simulation and of the estimation.

Plot analysis

Different paths are generated with the simulation and then filtered with the estimator in order to obtain the plots of different situations and analyse them. In particular, the following trajectories are simulated and shown in this chapter:

- "Horizontal" path
- Diagonal path
- Squared path
- Loop

The estimator was configured to run at half the simulation frequency. In other words, the simulation was running at $150\ Hz$ while the estimator at $75\ Hz$. The model in the simulation was corrupted with some errors. As it can be seen from the following tables, an error of 3% in the radius and 1% in the track gauge were introduced for the fixed wheel one. In other words, they correspond to assumption that the fixed wheel one is 3% bigger and 1% farther, from the origin of the moving platform frame, than the fixed wheel two. Considering the model parameters, these errors are quite severe. In fact, the robot has a $10\ cm$ diameter wheels, a track gauge of $40\ cm$. With the introduced errors, it means having an increment in the wheel radius of $3\ mm$ and an additional distance of $2\ mm$ from the moving platform frame. In practice, these errors are greater of the ones that a good robot assembly may have. Moreover, the encoders that are simulate have not a high resolution. They have a resolution of 120 dots per wheel revolution.

Plots layouts

In the following some plots are shown for each situation in which the filter was tested. The first image will always show two plots, one on the left and one on the right. The plot on the left illustrates:

- The trajectory the robot has followed, that is represented in green. It comes from the user control of the robot during the simulation.
- The trajectory that is obtained by considering only the odometry, in red. In this case, for the severe errors that were added in the model, this path will be far from being correct.
- The trajectory that was estimated by the filter in blue. This path shows that it corresponds to an accurate estimation.
- The measurements that are accepted as blues points. These measurements have passed the coherence test and they are used to perform the estimation.
- The lines that were estimated of being detected, represented by dashed yellow lines. They correspond to the lines that were estimated to be detected by the accepted measurements.
- The measurements that were rejected as little red points. For completeness, these measurements are also shown.

The second plot shows the Mahalanobis distances for the measurements, that are represented as before: blue points for the accepted measurements and little red points for the rejected ones.

The third image is for representing the error between the state vector and the robot posture in the simulation. The plots also show two red lines, which delimit the $\pm 3\sigma$ interval.

The fourth image shows the standard deviations during the execution. They do not correspond to the standard deviations assigned as initial. In fact, they are higher, which is since in the plot the motionless parts are deleted, allowing have more readable plots. However, when the estimation starts, the first part in the rosbag that is played contains motionless data. As a result, for few seconds, the uncertainties increase.

The fifth image contains two plots that shows the linear and angular velocities of the simulated robot and the ones computed by the estimator. This plot is not presented for every test.

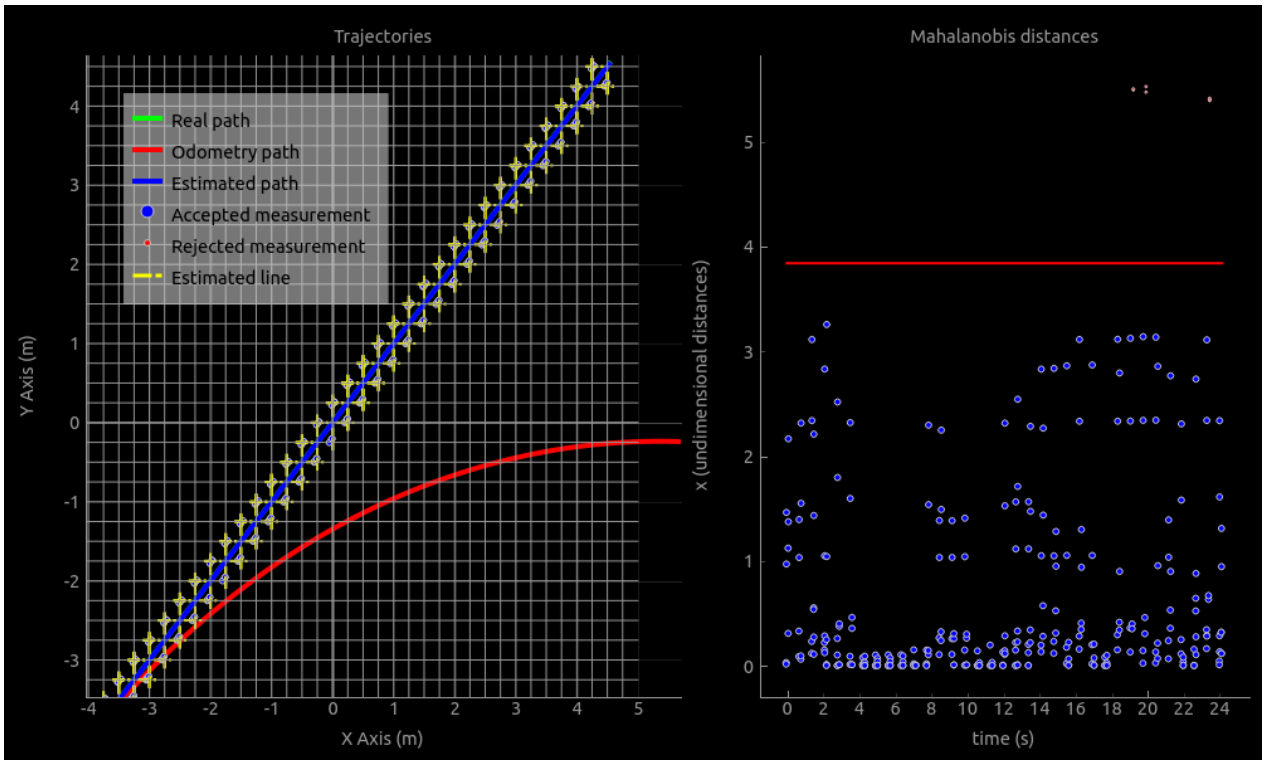
Inclined line

In this situation, the robot moves on a $45°$ line. This is an optimal situation for the estimator. In fact, both "horizontal" and "vertical" lines are detected frequently. As a result, the estimator is always available to correct the state vector and correctly localize the robot.

Parameter	Value	Units
WORLD PARAMETERS		
x_spacing	0.25	[m]
y_spacing	0.25	[m]
line_thickness	0.005	[m]
ROBOT PARAMETERS		
Wheel radius	0.05	[m]
Track gauge	0.4	[m]
Initial position x	-4.2	[m]
Initial position y	-4.2	[m]
Initial position theta	45.0	[grad]
ODOMETRY PARAMETERS		
wheel_1_error	3.0	[%]
track_gauge_error	1.0	[%]
encoders_resolution	120.0	[dots/revolution]
ESTIMATOR PARAMETERS		
sigma_tuning	0.075	
Threshold	3.84	
Initial position x	-4.19	[m]
Initial position y	-4.21	[m]
Initial position theta	48.0	[grad]
ESTIMATION RESULTS		
Rejected measurements	8.0	
Percentage of rejected	2.63	[%]
Path lenght	12.32	[m]

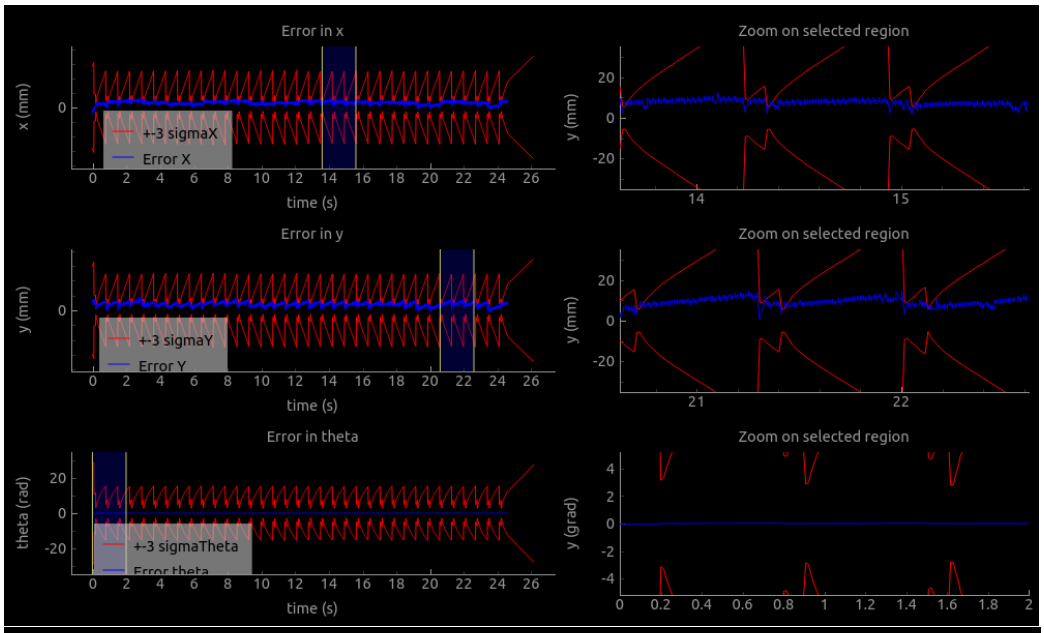
The table on the left shows the parameters that have been used in this simulation. The initial position for the estimator is slightly different from the real one, but the estimator does correct the robot state vector when the first measurement occurs.

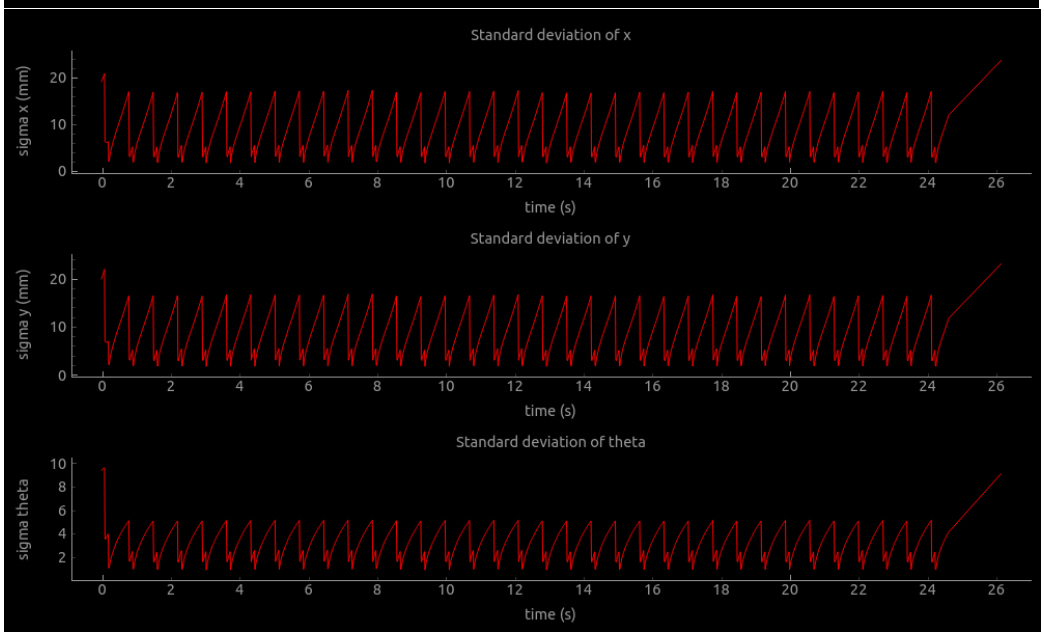
The image below shows the path and the Mahalanobis distances. From the plot of the left, it is possible to notice that the estimator corrects the state vector in a way that the two paths correspond almost perfectly. In the plot on the right, the measurements that are rejected are few, and the Mahalanobis distances are homogeneously distributed below the threshold.

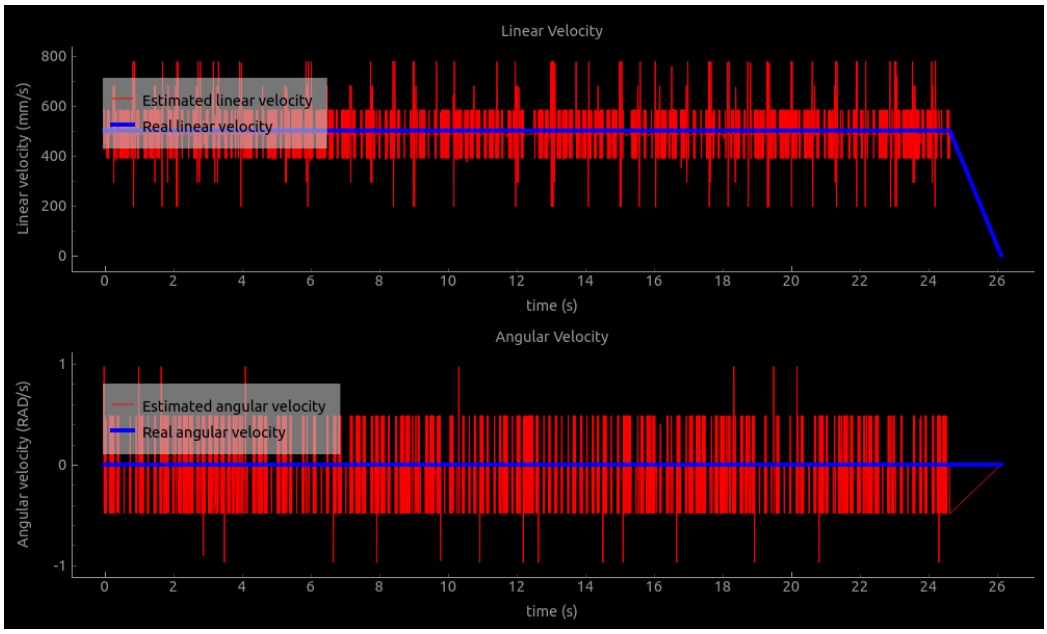


The following figure shows the error between the estimated posture and the real posture. From the plots it is possible to notice that the error remains in the order of few millimeters. Moreover, it is always in between them $\pm 3\sigma$ range.

For what concerns the standard deviations, they behave as expected, they drop every time the two types of lines have been detected.


Error in x
x (mm)
+-3 sigmaX
Error X
time (s)
Zoom on selected region
y (mm)
Error in y
y (mm)
+-3 sigmaY
Error Y
time (s)
Zoom on selected region
y (mm)
Error in theta
theta (rad)
+-3 sigmaTheta
Error theta
time (s)
Zoom on selected region
y (grad)

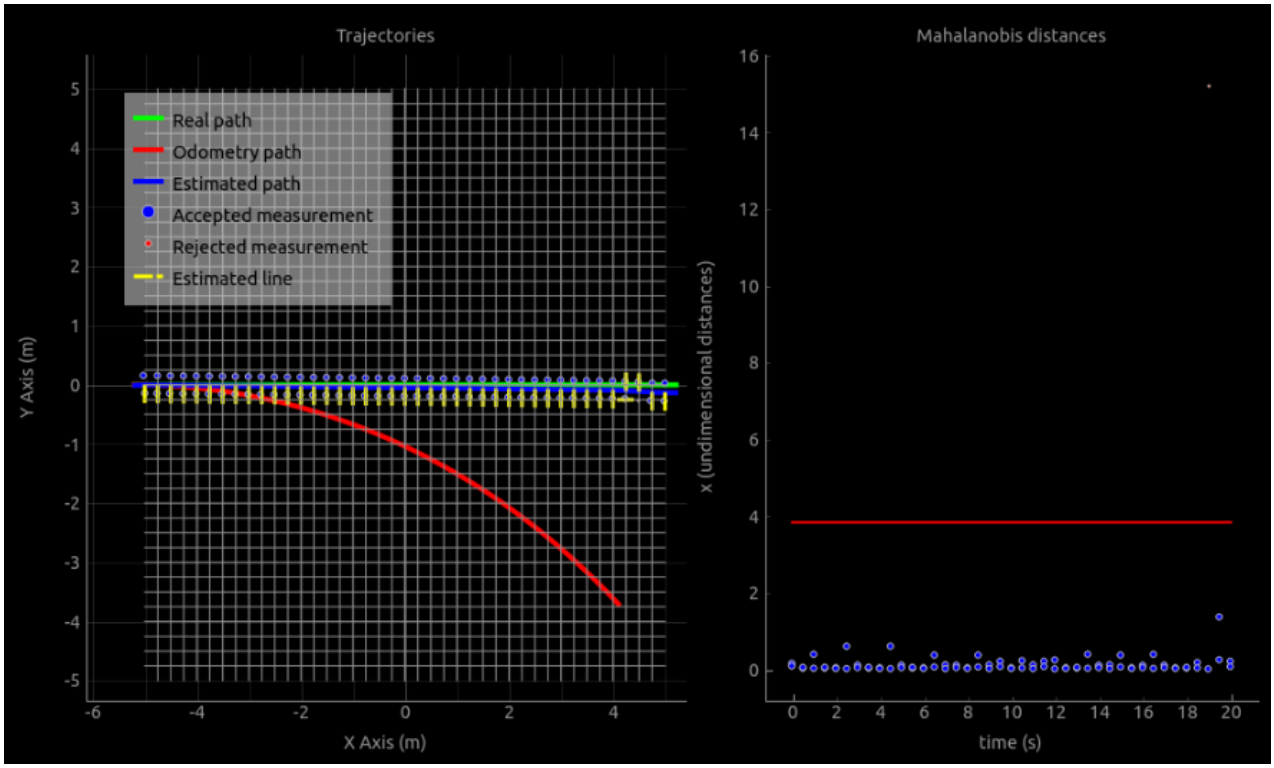

Standard deviation of x
sigma x (mm)
time (s)
Standard deviation of y
sigma y (mm)
time (s)
Standard deviation of theta
sigma theta
time (s)


Linear Velocity
Linear velocity (mm/s)
Estimated linear velocity
Real linear velocity
time (s)
Angular Velocity
Angular velocity (RAD/s)
Estimated angular velocity
Real angular velocity
time (s)

“Horizontal” line

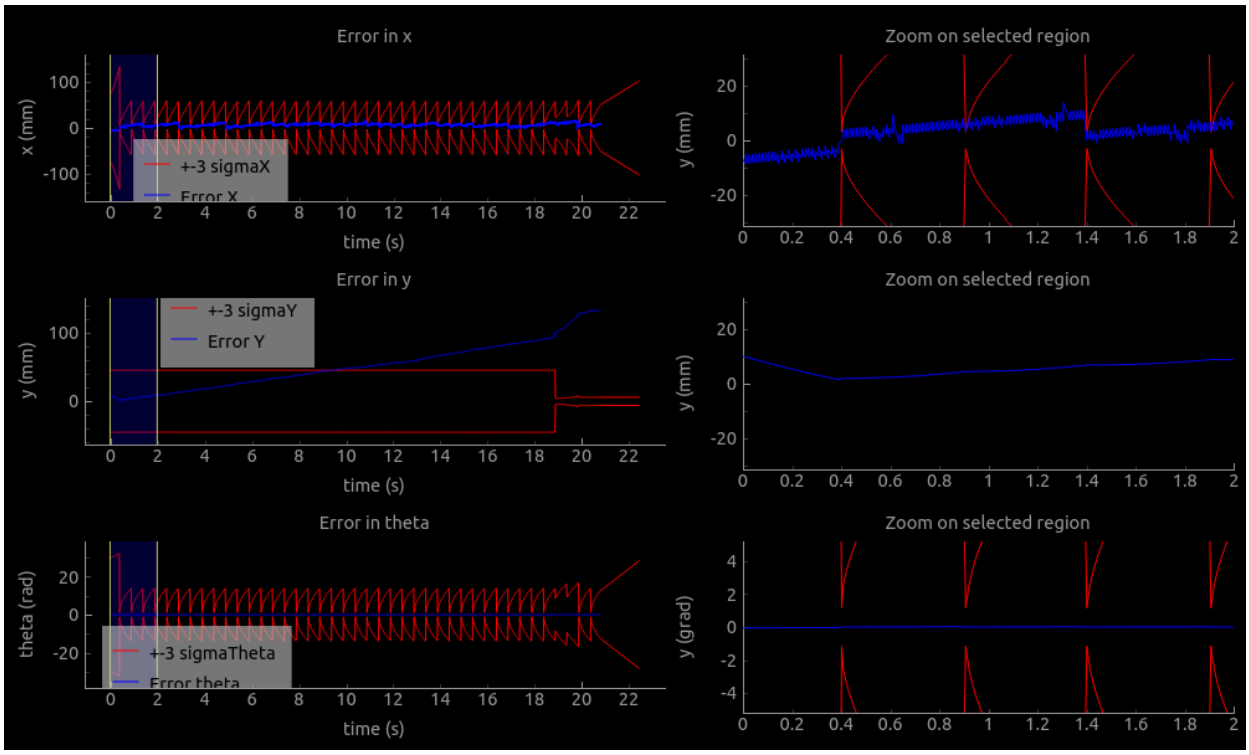
In this run, the estimator is tested in a situation where it is not expected to work properly. In fact, in the simulation the robot was controlled to move in straight line, in a trajectory perfectly along the x axis. In this situation, no “horizontal” lines are detected. The estimator is not able to fully correct the errors and the result is an estimation far from the reality. The two paths clearly differ in the image below. Moreover, the distances in the plot on the right result very small. The reason is that the standard deviation does not decrease enough to allow higher Mahalanobis distances.

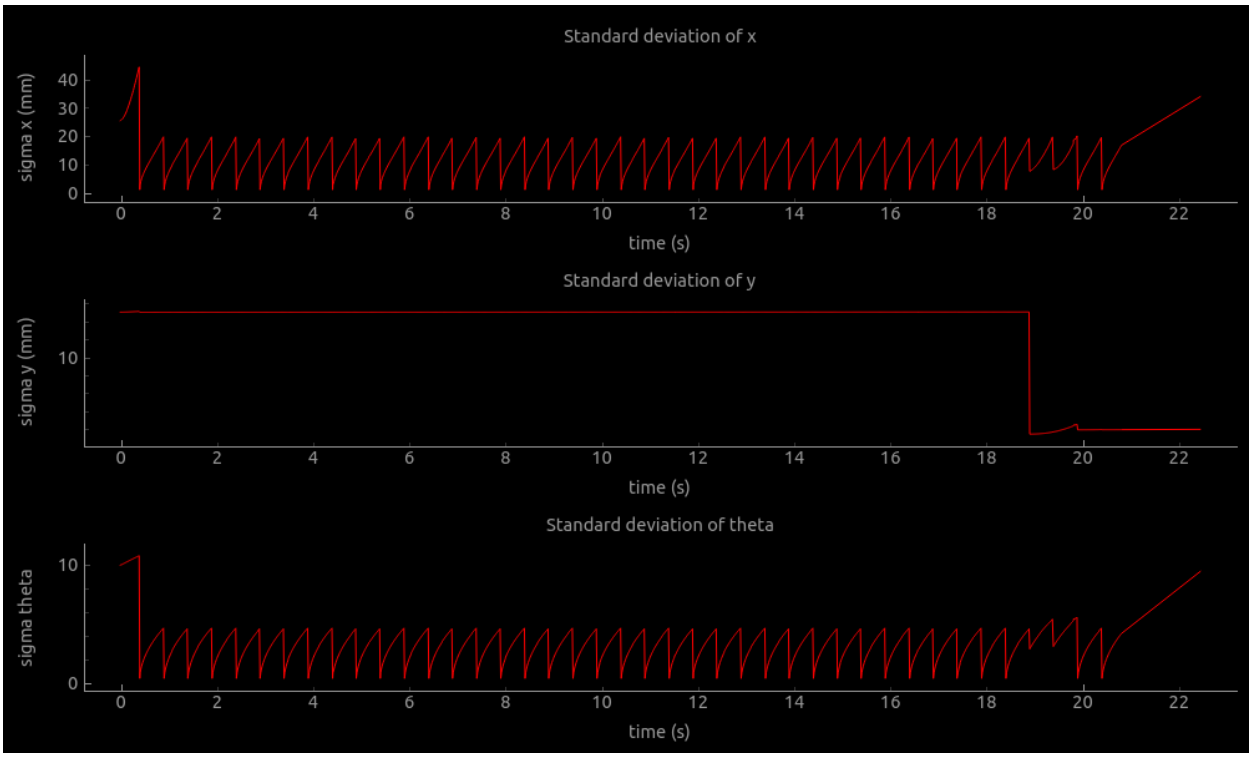
Parameter	Value	Units
WORLD PARAMETERS		
x_spacing	0.25	[m]
y_spacing	0.25	[m]
line_thickness	0.005	[m]
ROBOT PARAMETERS		
Wheel radius	0.05	[m]
Track gauge	0.4	[m]
Initial position x	-5.2	[m]
Initial position y	0.0	[m]
Initial position theta	0.0	[grad]
ODOMETRY PARAMETERS	0.0	
wheel_1_error	3.0	[%]
track_gauge_error	1.0	[%]
encoders_resolution	120.0	[dots/revolution]
ESTIMATOR PARAMETERS		
sigma_tuning	0.075	
Threshold	3.84	
Initial position x	-5.19	[m]
Initial position y	-0.01	[m]
Initial position theta	3.0	[grad]
ESTIMATION RESULTS		
Rejected measurements	1.0	
Percentage of rejected	1.22	[%]
Path lenght	10.41	[m]



The following images confirm what previously stated, the estimator is only able to correct the x coordinates while the error in the y continues to increase as well as the uncertainty on this coordinate.

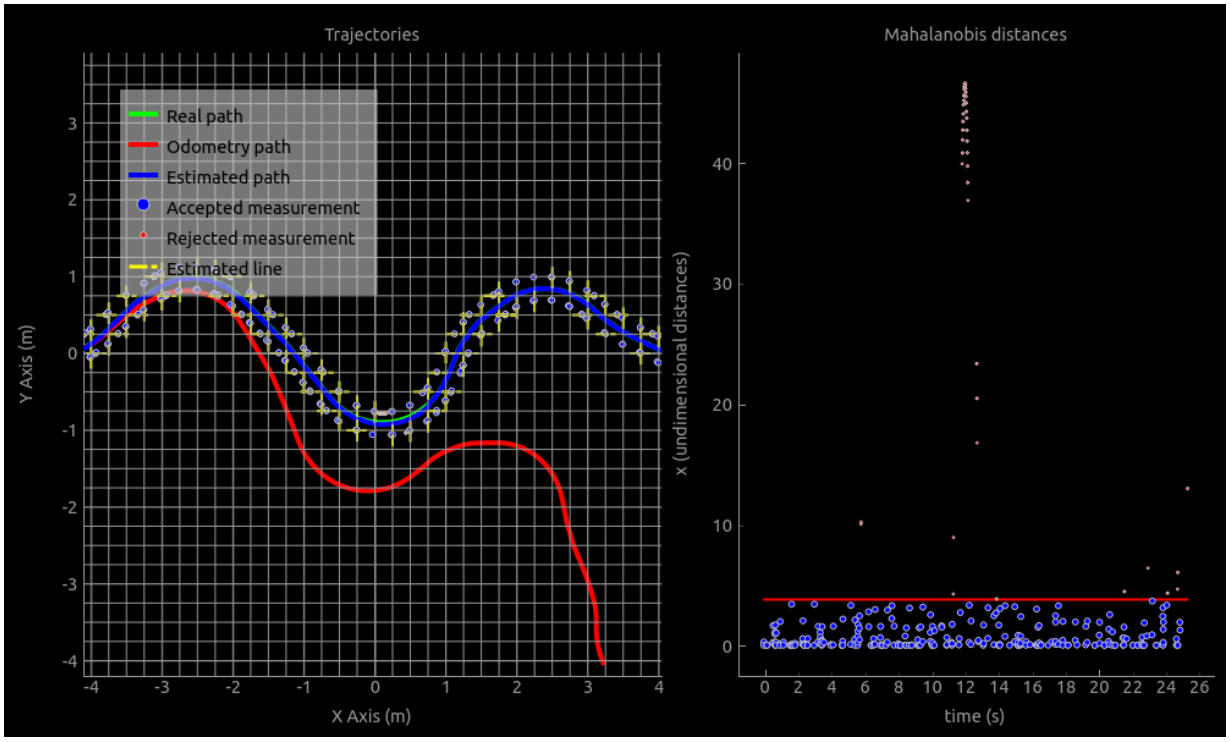
The fact that, at around 19 seconds, the standard deviation on y drops is due to a misinterpretation of a measurement. In fact, from the simulation, it is possible to see that no horizontal lines were detected. However, as the estimator was not able to correct the y position, it has assumed that a certain measurement corresponded to a horizontal line. Additionally, for the high uncertainty in the y coordinate, it has updated the state vector with this wrong measurement. This test was to emphasize the limitation of this algorithm. If a line is erroneously assumed to be measured, the result is to have an estimator completely lost.

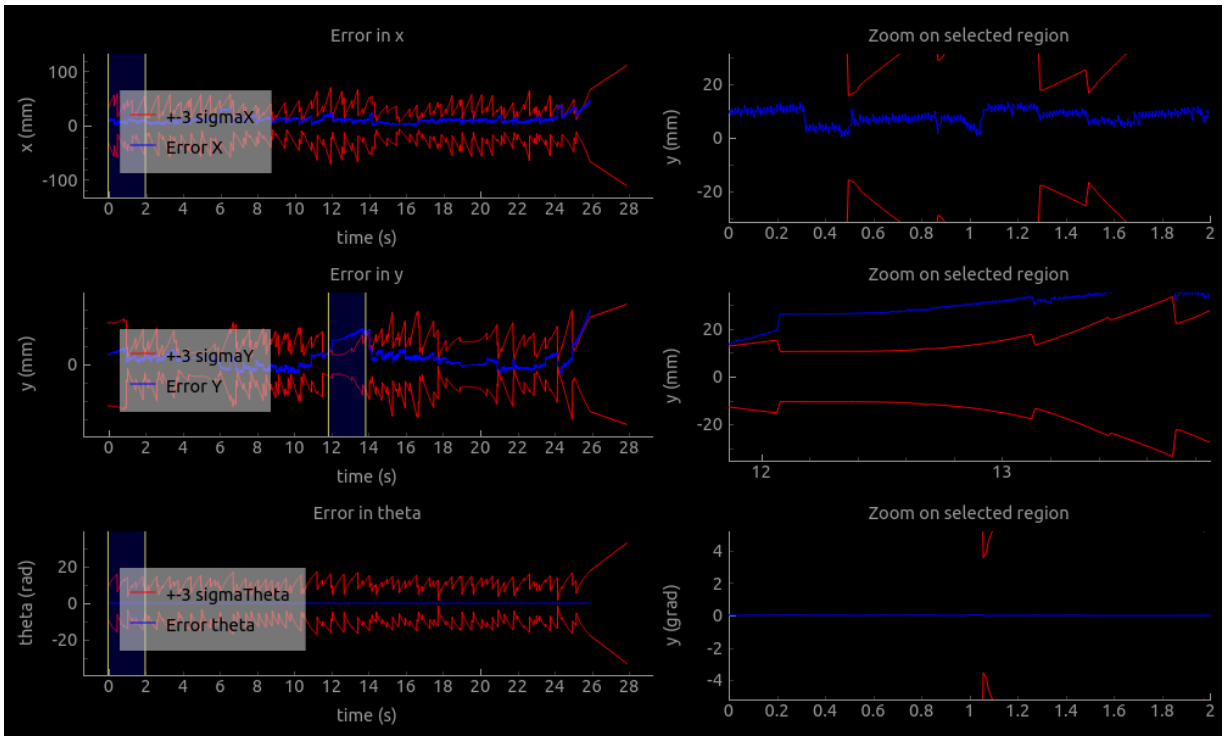

Error in x
x (mm)
time (s)
+-3 sigmaX
Error X
Zoom on selected region
y (mm)
Error in y
y (mm)
time (s)
+-3 sigmaY
Error Y
Zoom on selected region
y (mm)
Error in theta
theta (rad)
time (s)
+-3 sigmaTheta
Error theta
Zoom on selected region
y (grad)


Standard deviation of x
sigma x (mm)
time (s)
Standard deviation of y
sigma y (mm)
time (s)
Standard deviation of theta
sigma theta
time (s)

How to avoid observability singularities

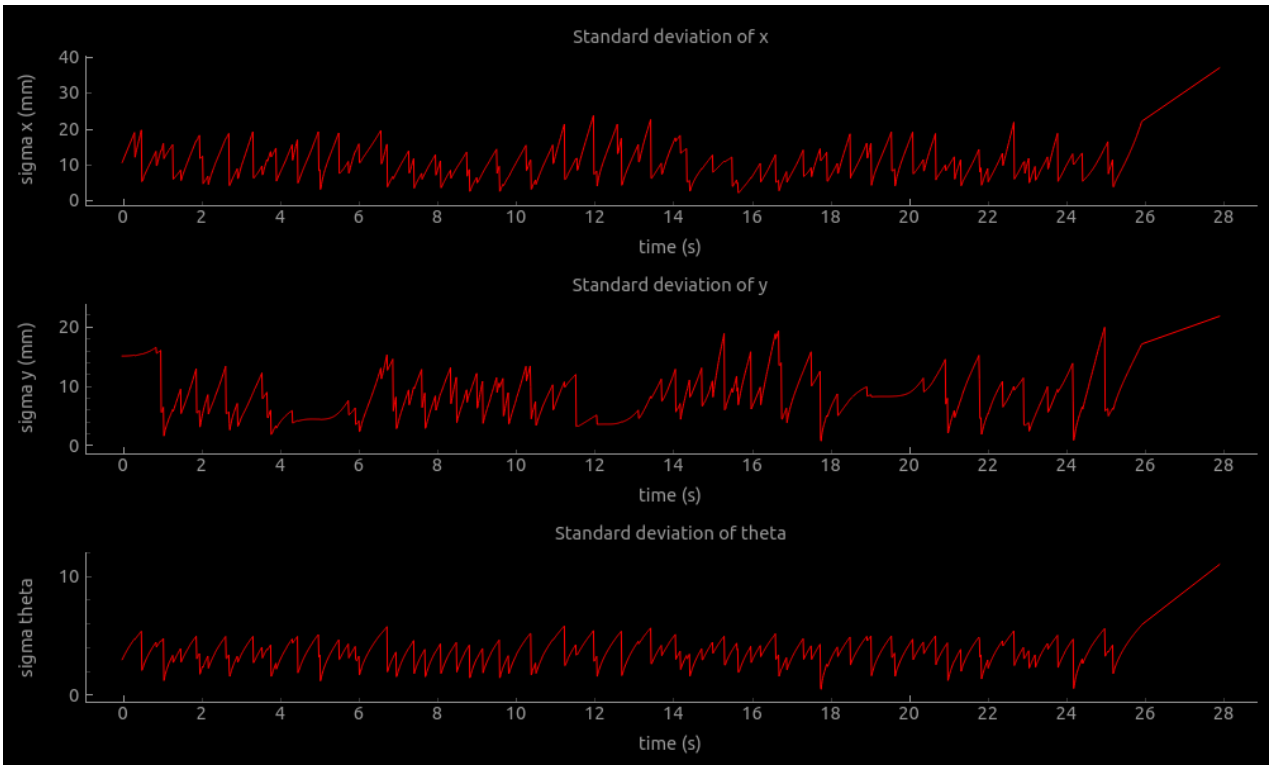
The following plots show the only way in which the robot should be controlled, using this localization algorithm. For having the estimator able to locate the robot, it must move in zig zag. The most efficient way is to turn the robot with a pure rotation motion. In fact, with a rotation on the place, the robot does not move in a dangerous way. As the robot model is a $(2, 0)$, this motion can be easily achieved. However, to test the estimator, the robot was rotating while moving forward, resulting in missing one type of measurement when steering. Nonetheless the estimator was able to correct the posture and the results are shown in the images below:





In the last figure, the second plot shows an increment in the error along y. This is since, for a few seconds, the sensors did not detect any “horizontal” line. As a matter of fact, it was not possible to correct the errors along y until the next measurement of the missing line type.

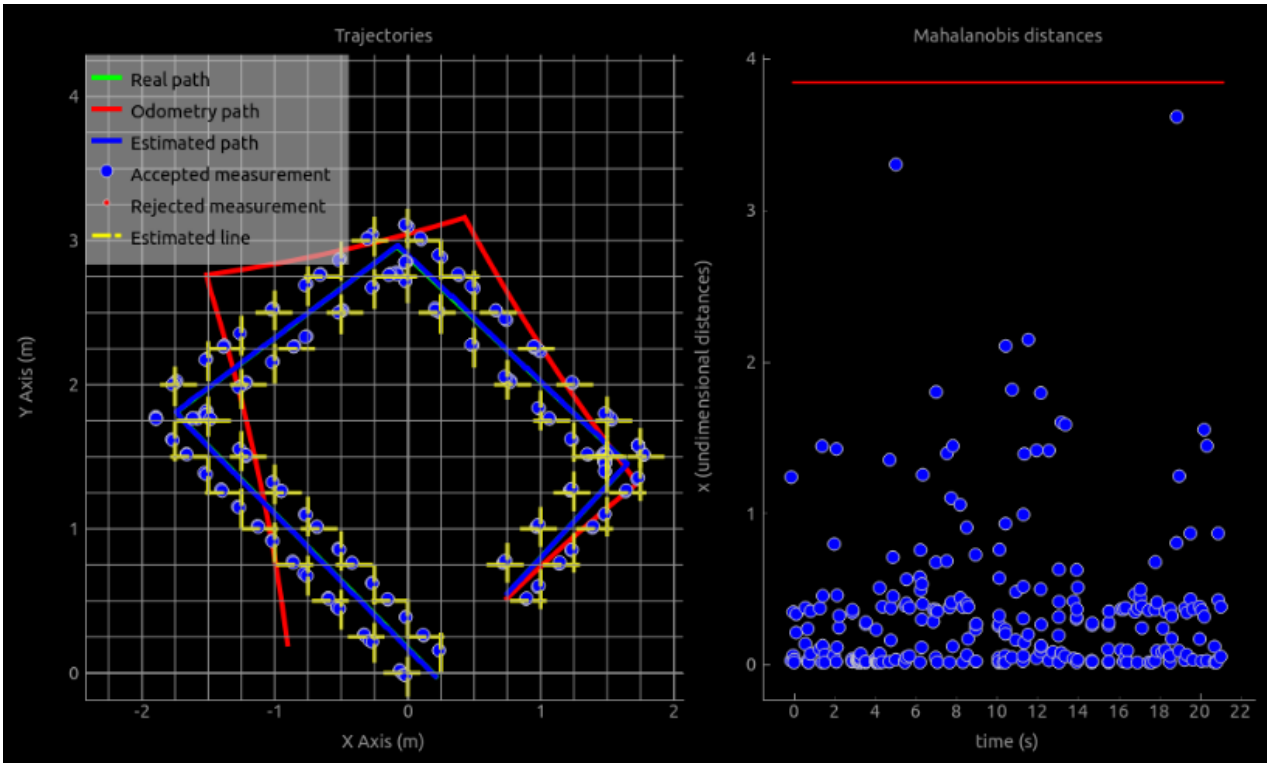
The following figures shows the three plots of the standard deviation of the state vector elements.



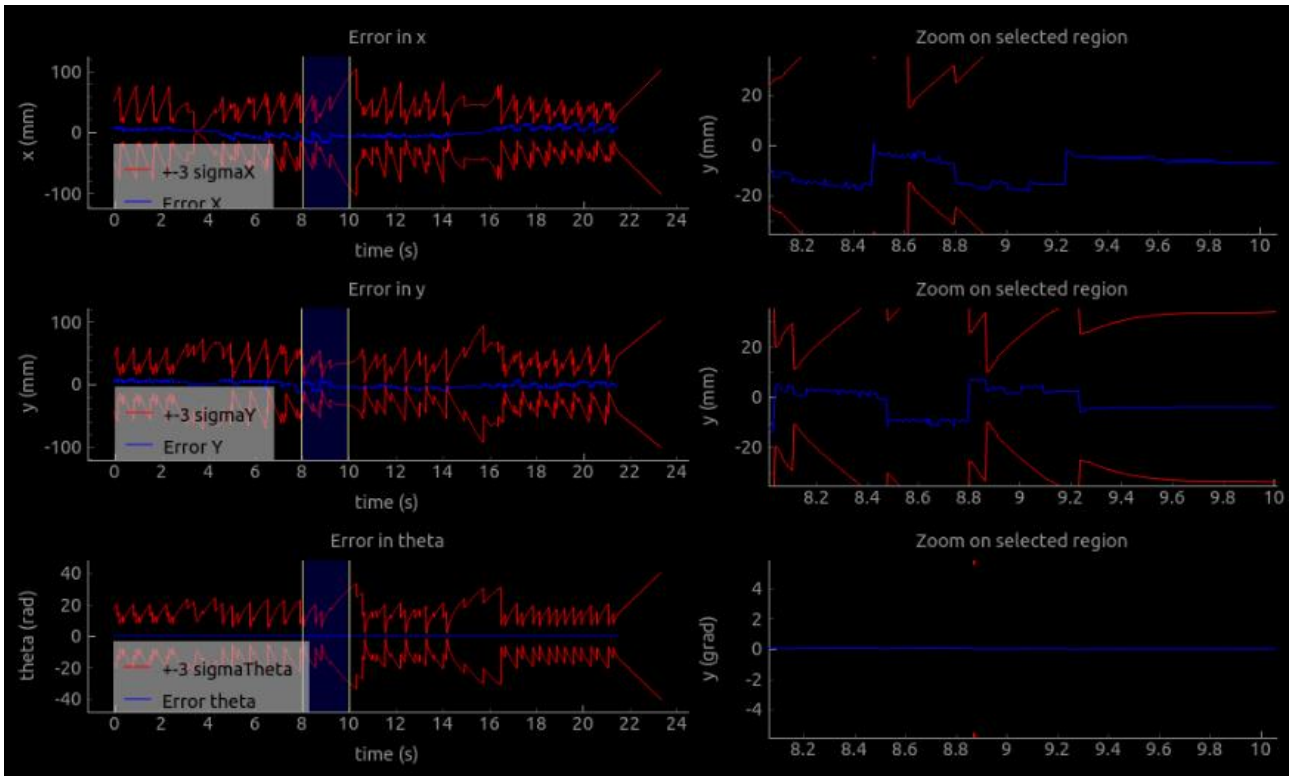
Squared path

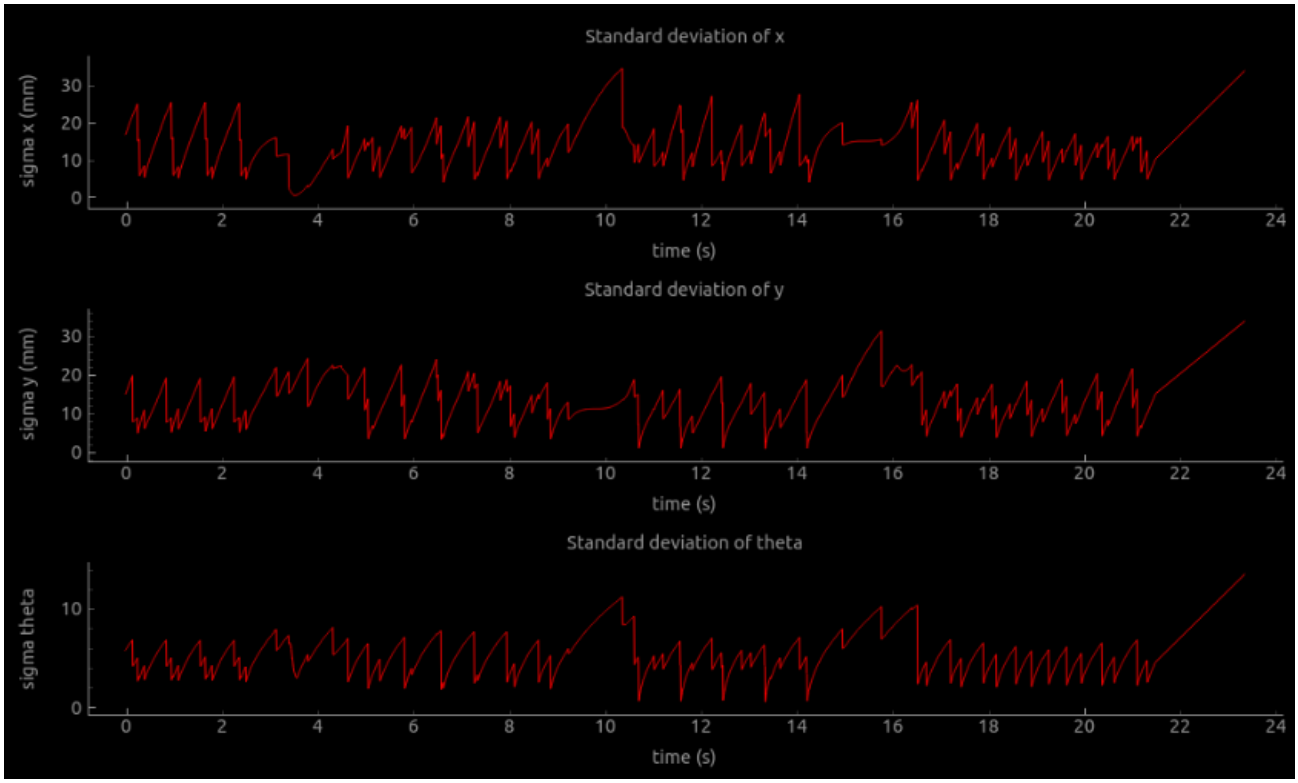
The estimator in this case, as expected, behaves in the proper way. As it is shown in the plots below, the filtered path follows perfectly the real one and there are not rejected measurements. This situation in fact represents the optimal one for this type of localization system as the path is basically composed by diagonal line. In this way, the sensors detect horizonal and vertical lines and the system takes all the measurements, as it shown in the following plots.

Parameter	Value	Units
WORLD PARAMETERS		
x_spacing	0.25	[m]
y_spacing	0.25	[m]
line_thickness	0.005	[m]
ROBOT PARAMETERS		
Wheel radius	0.05	[m]
Track gauge	0.4	[m]
Initial position x	0.2	[m]
Initial position y	0.0	[m]
Initial position theta	45.0	[grad]
ODOMETRY PARAMETERS		
wheel_1_error	3.0	[%]
track_gauge_error	1.0	[%]
encoders_resolution	120.0	[dots/revolution]
ESTIMATOR PARAMETERS		
sigma_tuning	0.075	
Threshold	3.84	
Initial position x	0.2	[m]
Initial position y	0.0	[m]
Initial position theta	45.0	[grad]
ESTIMATION RESULTS		
Rejected measurements	0.0	
Percentage of rejected	0.0	[%]
Path lenght	8.27	[m]



Since in this case the filter works in an optimal way, the error in the three coordinates, x, y and theta, is inside the range for all the estimation, as it shown in the following plots:



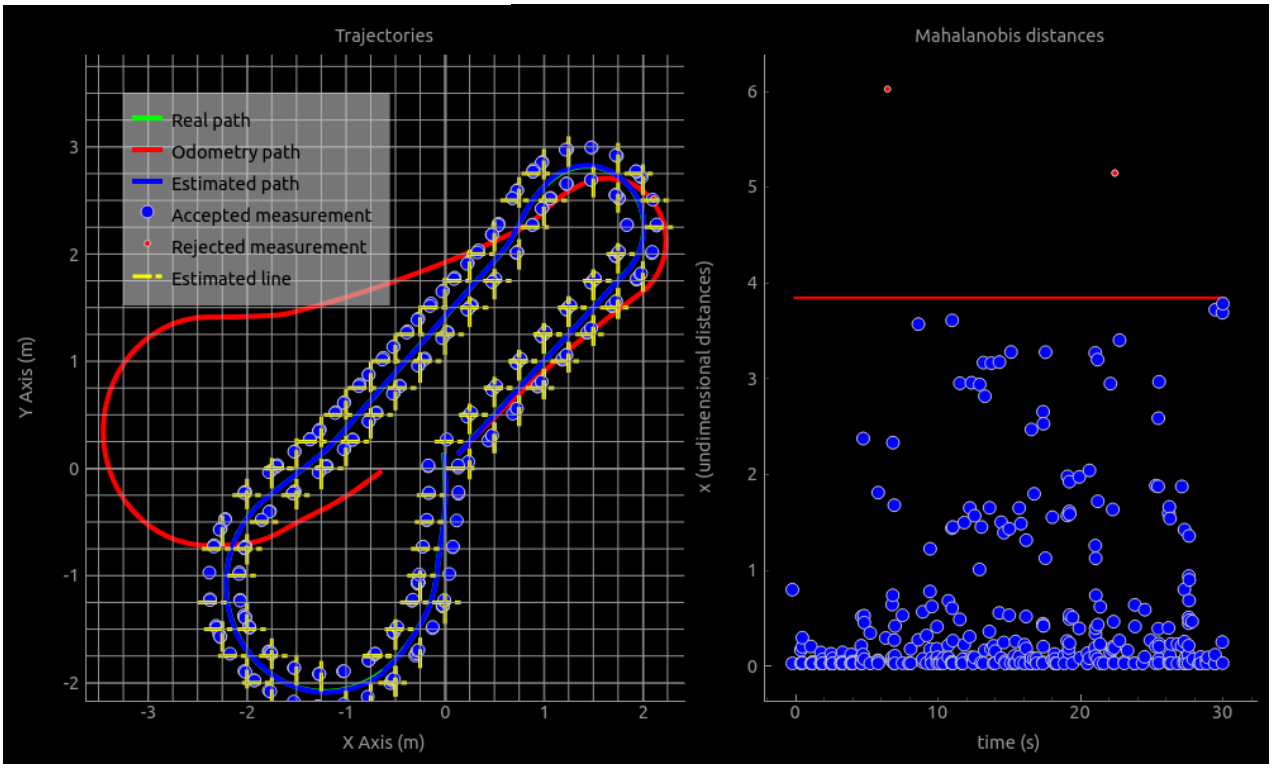


Loop path

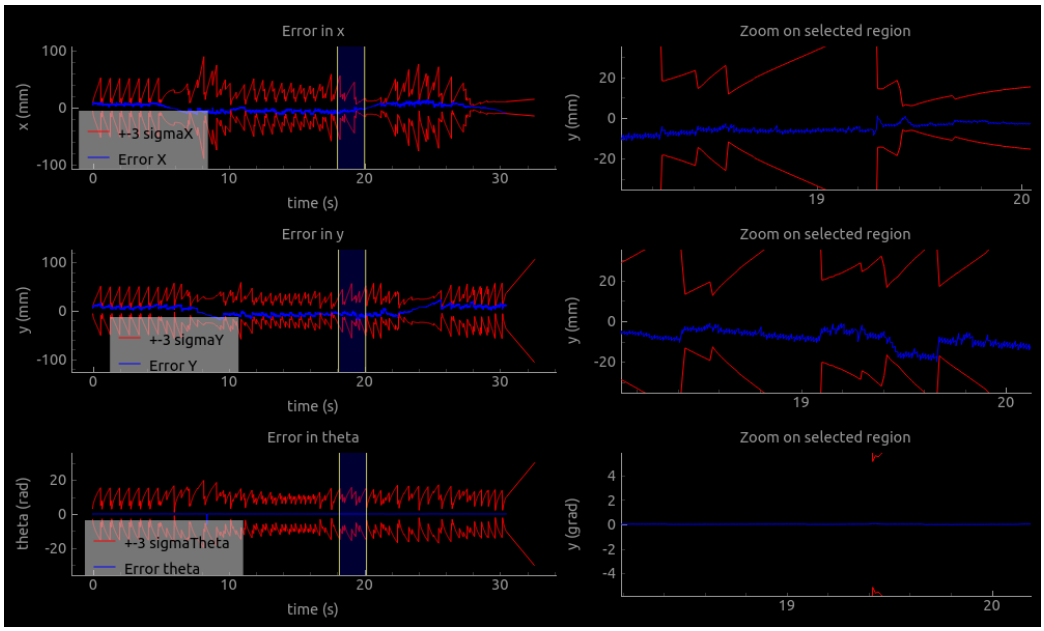
The estimator was tested in a situation where the robot was moving a path that does not satisfies the optimal configuration. In fact, for this estimator, when the robot steers, there is a potentially problematic situation as one kind of line may be not detected. During this time, the estimator is not able to correct one coordinate of the state vector. This is potentially fatal, as it might lead the estimator to be lost.

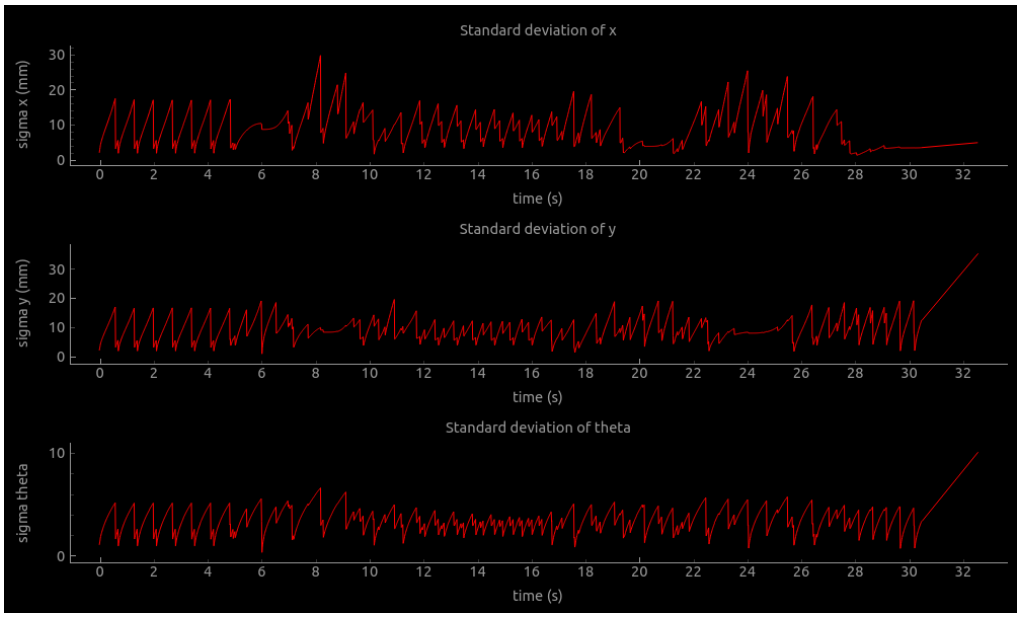
Parameter	Value	Units
WORLD PARAMETERS		
x_spacing	0.25	[m]
y_spacing	0.25	[m]
line_thickness	0.005	[m]
ROBOT PARAMETERS		
Wheel radius	0.05	[m]
Track gauge	0.4	[m]
Initial position x	0.0	[m]
Initial position y	0.0	[m]
Initial position theta	45.0	[grad]
ODOMETRY PARAMETERS		
wheel_1_error	3.0	[%]
track_gauge_error	1.0	[%]
encoders_resolution	120.0	[dots/revolution]
ESTIMATOR PARAMETERS		
sigma_tuning	0.075	
Threshold	3.84	
Initial position x	0.01	[m]
Initial position y	-0.01	[m]
Initial position theta	48.0	[grad]
ESTIMATION RESULTS		
Rejected measurements	3.0	
Percentage of rejected	0.93	[%]
Path lenght	13.62	[m]

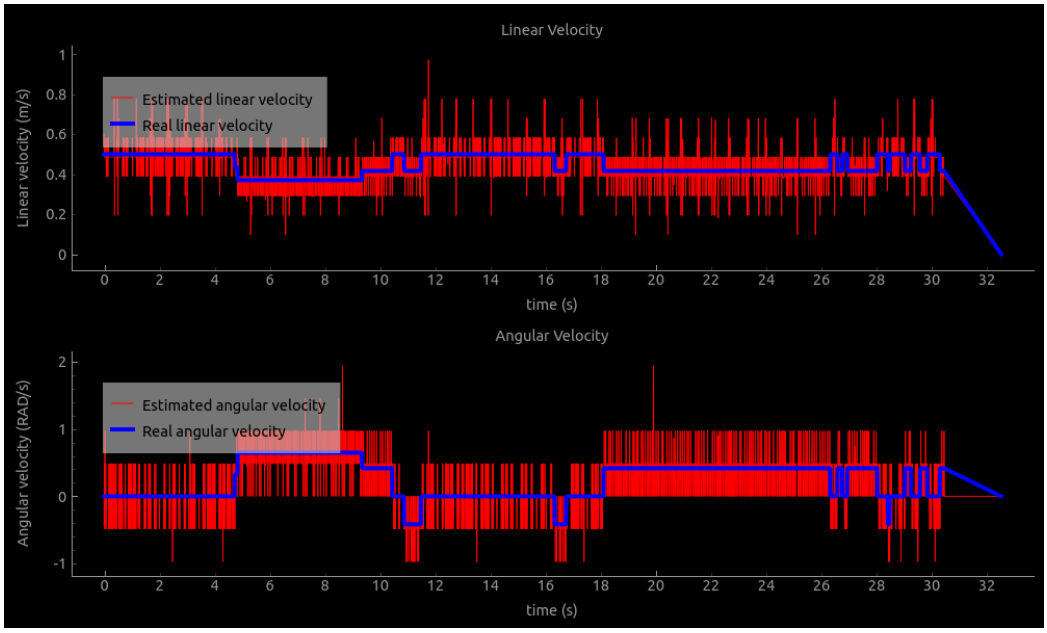
However, in the case presented, the estimator was able to cope with this problematic situation two times. The first time with a little radius curve and the second time with a curve of a wider radius.



Even if it does correct the posture, the effect of not detecting one of the type of lines can be seen in the errors and standard deviations plots. In fact, the following images show the errors and uncertainties to increase when one type of measurement is missing.


Error in x
x (mm)
+-3 sigmaX
Error X
time (s)
Zoom on selected region
y (mm)
Error in y
y (mm)
+-3 sigmaY
Error Y
time (s)
Zoom on selected region
y (mm)
Error in theta
theta (rad)
+-3 sigmaTheta
Error theta
time (s)
Zoom on selected region
y (grad)


Standard deviation of x
sigma x (mm)
time (s)
Standard deviation of y
sigma y (mm)
time (s)
Standard deviation of theta
sigma theta
time (s)

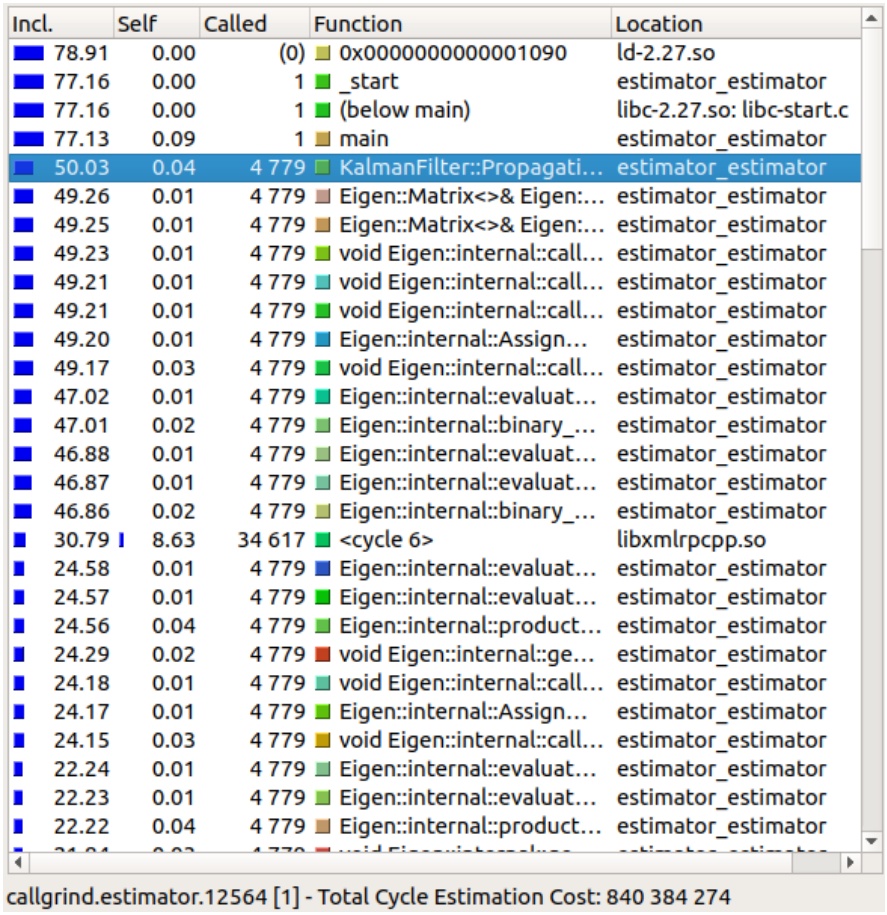

Linear Velocity
Linear velocity (m/s)
Estimated linear velocity
Real linear velocity
time (s)
Angular Velocity
Angular velocity (RAD/s)
Estimated angular velocity
Real angular velocity
time (s)

Estimator performance

A few words should be spent concerning the estimator performances. In fact, this software component is designed for being installed in a real robot, which moves on a tiled floor.

The estimator was tested in unusual or adverse condition, to proof its capabilities and performances. However, when comes the time to implement it, the goal is to ensure as much as possible ideal working condition. It is not possible to make assumption about the performances of the sensors that may be implemented[4], but it is possible to ensure the code performances.

After having tuned the filter, the estimator was executed under a profiler to retrieve information about its performances and detect eventual bottlenecks. The following images shows the more time-consuming functions during an estimator execution.



Incl.	Self	Called	Function	Location
78.91	0.00	(0)	0x0000000000001090	ld-2.27.so
77.16	0.00	1	_start	estimator_estimator
77.16	0.00	1	(below main)	libc-2.27.so: libc-start.c
77.13	0.09	1	main	estimator_estimator
50.03	0.04	4 779	KalmanFilter::Propagati…	estimator_estimator
49.26	0.01	4 779	Eigen::Matrix<>& Eigen:…	estimator_estimator
49.25	0.01	4 779	Eigen::Matrix<>& Eigen:…	estimator_estimator
49.23	0.01	4 779	void Eigen::internal::call…	estimator_estimator
49.21	0.01	4 779	void Eigen::internal::call…	estimator_estimator
49.21	0.01	4 779	void Eigen::internal::call…	estimator_estimator
49.20	0.01	4 779	Eigen::internal::Assign…	estimator_estimator
49.17	0.03	4 779	void Eigen::internal::call…	estimator_estimator
47.02	0.01	4 779	Eigen::internal::evaluat…	estimator_estimator
47.01	0.02	4 779	Eigen::internal::binary_…	estimator_estimator
46.88	0.01	4 779	Eigen::internal::evaluat…	estimator_estimator
46.87	0.01	4 779	Eigen::internal::evaluat…	estimator_estimator
46.86	0.02	4 779	Eigen::internal::binary_…	estimator_estimator
30.79	8.63	34 617	<cycle 6>	libxmlrpcpp.so
24.58	0.01	4 779	Eigen::internal::evaluat…	estimator_estimator
24.57	0.01	4 779	Eigen::internal::evaluat…	estimator_estimator
24.56	0.04	4 779	Eigen::internal::product…	estimator_estimator
24.29	0.02	4 779	void Eigen::internal::ge…	estimator_estimator
24.18	0.01	4 779	void Eigen::internal::call…	estimator_estimator
24.17	0.01	4 779	Eigen::internal::Assign…	estimator_estimator
24.15	0.03	4 779	void Eigen::internal::call…	estimator_estimator
22.24	0.01	4 779	Eigen::internal::evaluat…	estimator_estimator
22.23	0.01	4 779	Eigen::internal::evaluat…	estimator_estimator
22.22	0.04	4 779	Eigen::internal::product…	estimator_estimator

callgrind.estimator.12564 [1] - Total Cycle Estimation Cost: 840 384 274

The previous figure shows the functions called during the execution. The order is given by the length of the little blue bar on the left. This bar represents the percentage of time spent in each function.

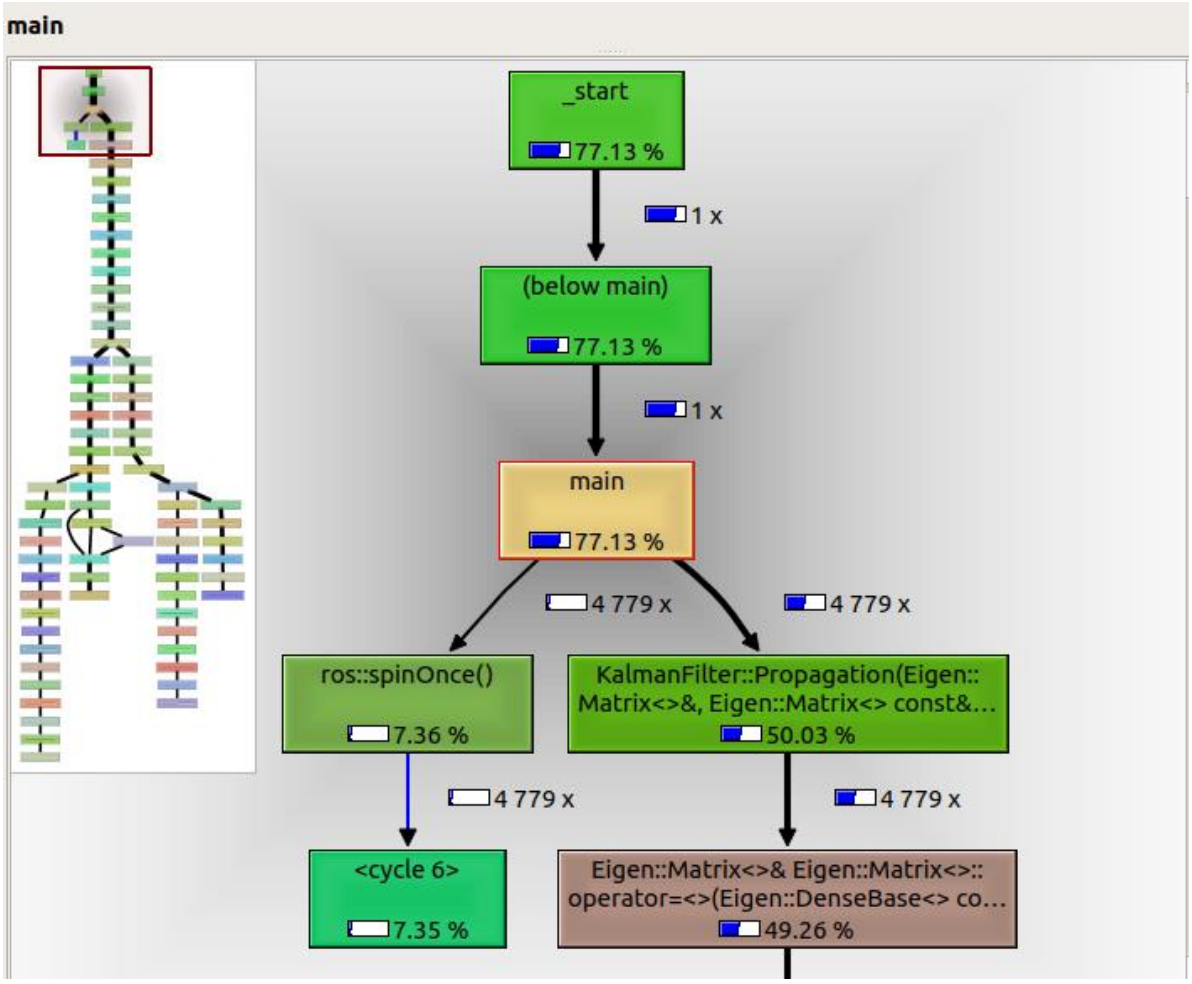
Apart of some initialization functions, it is possible to see that the $77.13\ \%$ of the execution time is spent on the main function, that is called only once and contains all the function calling. More relevant is the highlighted function; it is one from the Kalman Filter class that is implemented in the code. It is called $4\,779$ times and the 50.03% of the execution time is spent inside this member function. However, it is wrong to assume that it consists of a bottleneck. In fact, by looking just below, it is possible to see that the other functions are from the Eigen library, that is implemented for the matrix computations. As a matter of fact, the difference from the Kalman Filter member function and the copy operator for and Eigen matrix is 0.77%, meaning that this function does not represent a bottleneck.

[4] Even if both the extrinsic and intrinsic sensor are very standard and they should be able to work at high frequencies.

From another point of view, by taking a step backward from the code, the function performs the error propagation that is computed at each iteration. It is so normal to see this function in the top five. The fact that directly followed by the copy operator for the Eigen matrix, means that this function works as fast as the operations it performs. Moreover, this function performs few matrix operations, that are from a widely used mathematical library that is already optimized. In other words, there is not much to do to improve the code performances, at it already runs almost as fast as it can do.

After having profiled the estimator, it is possible to establish that it can run at higher frequency than $75\ Hz$ that was the frequency at which it was tested.

The next image is for providing another point of view for the functions in the code.

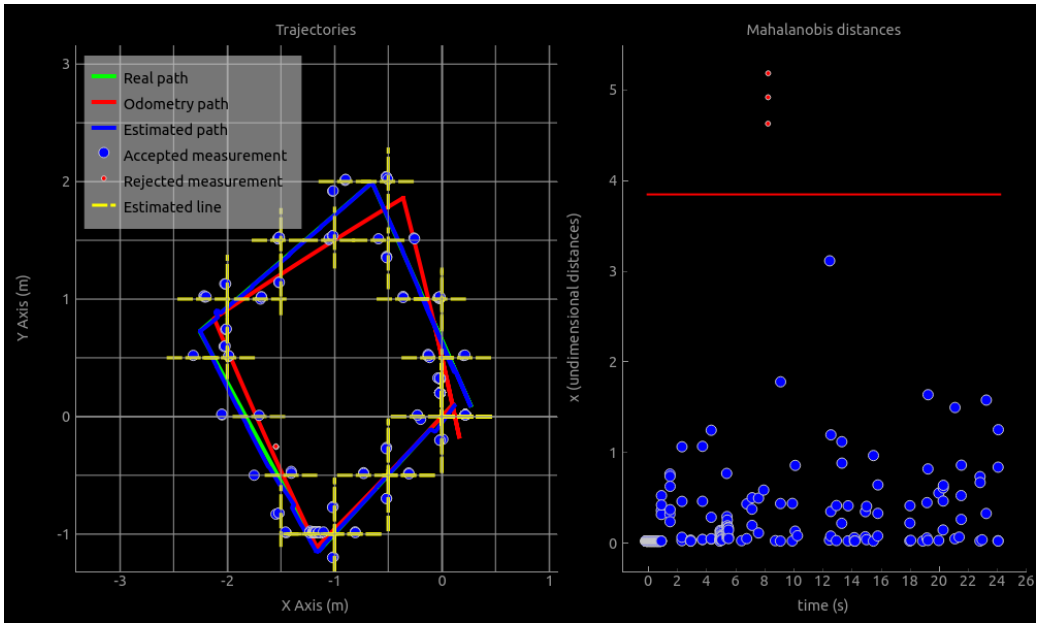


Conclusions

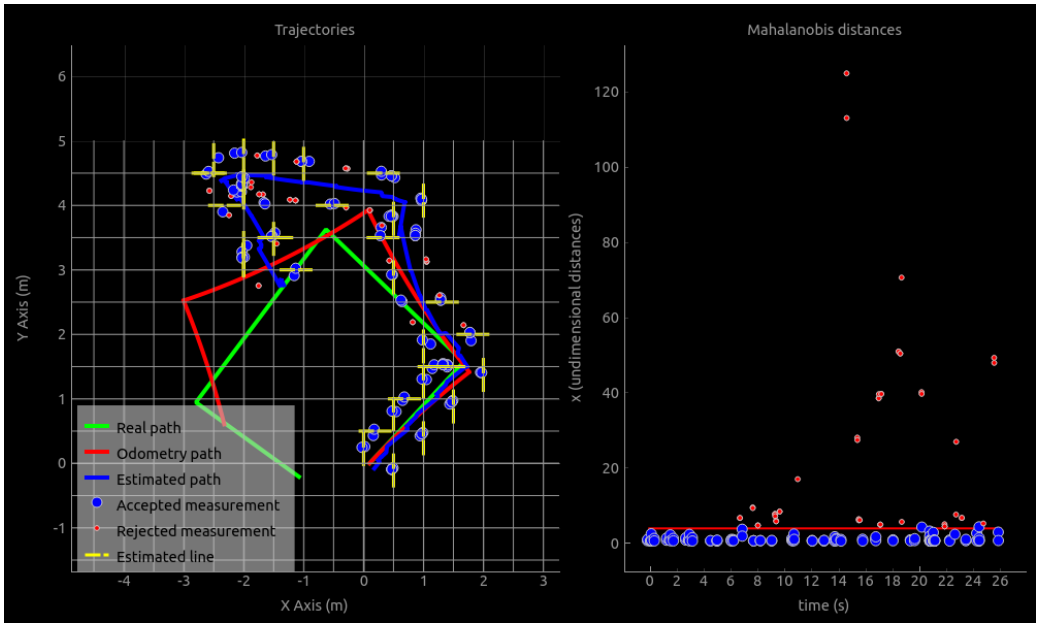
In conclusion, the system can simulate the motion of a robot (2,0), to detect the floor tiles using the infrared sensors and to localize itself using a hybrid localization method.

Considering different situations, a test was performed to understand how the system behaves in critical situations and which are its limits. Regarding the dimension of the tiles, the system works as it should even with tiles of 0.5 m. For example, introducing small errors in the error and track gauge, both equal to 1%, the estimator works correctly. However, to increase the performances in this situation, the two sensors where placed further from the moving platform frame.

Increasing the distances from the sensors to the moving platform frame increases the precision in the correction of the robot heading. Moreover, with this setup, the robot performs a better detection of the lines and the estimator can perform the filter properly, as shown below:

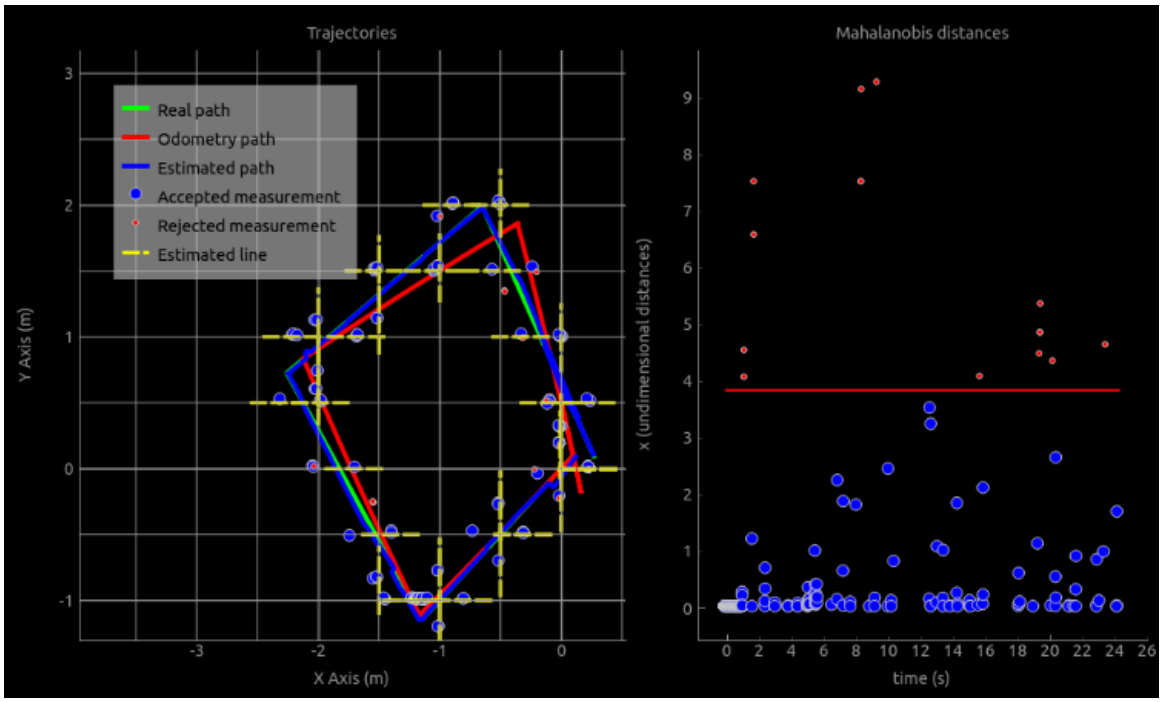


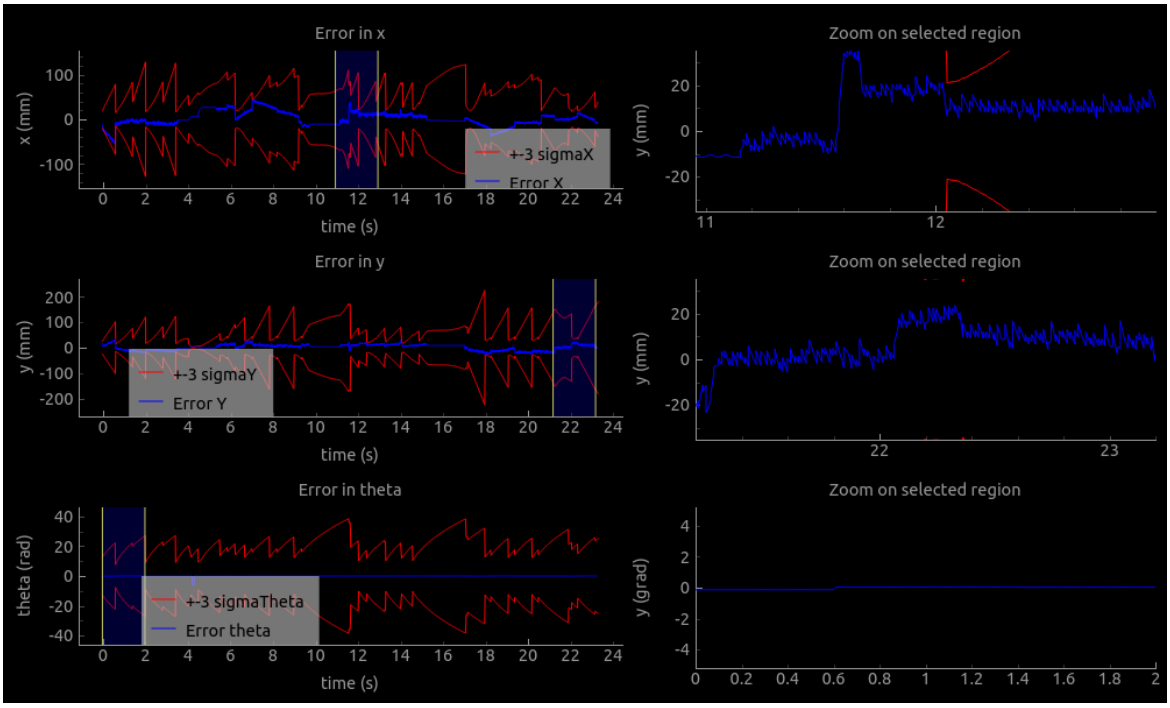
However, the system has some limitations: for example, even with a higher value of sigma tuning, it diverges if there is an error of 3% in wheel radius and track gauge, as it shown in the following plot:



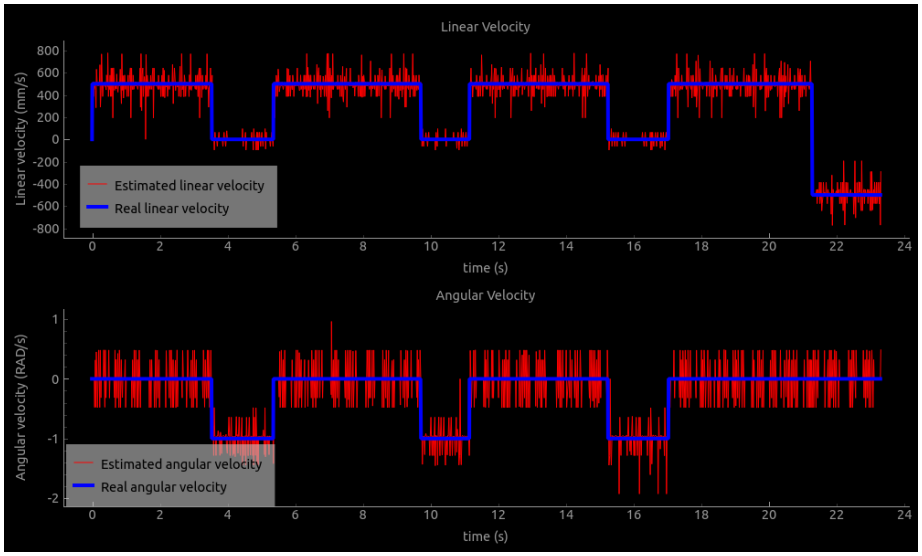
Besides, in the last test there was an error of the initial position of 8 *cm* in x and y and 5 degrees in theta, that is considerately an exaggeration, but it gives an idea of how much the system can handle. In fact, in the beginning of the trajectory, the estimator does correct the initial error.

Regarding the frequency, the estimation works even if the frequency of the estimator is 5 times smaller than the one of the simulation *i.e.* 30Hz, as it shown in the following plot:

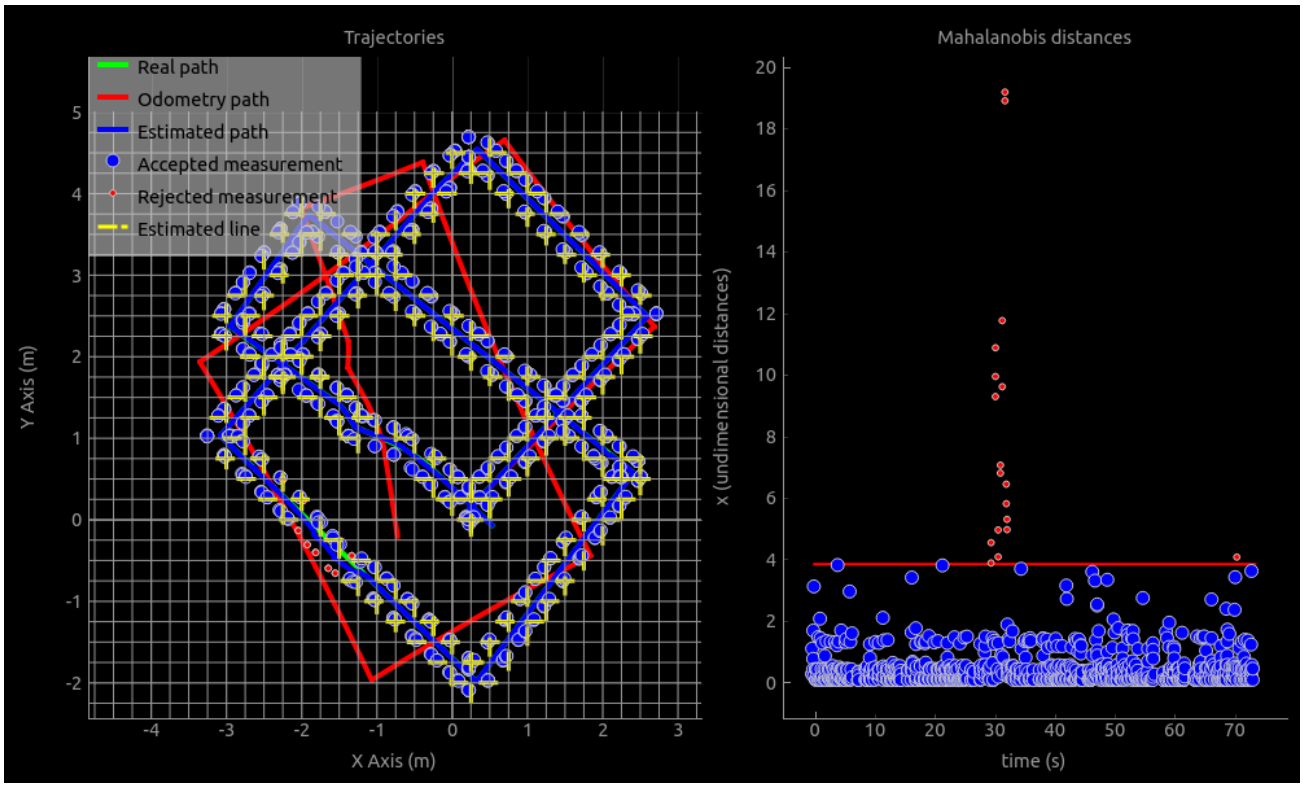


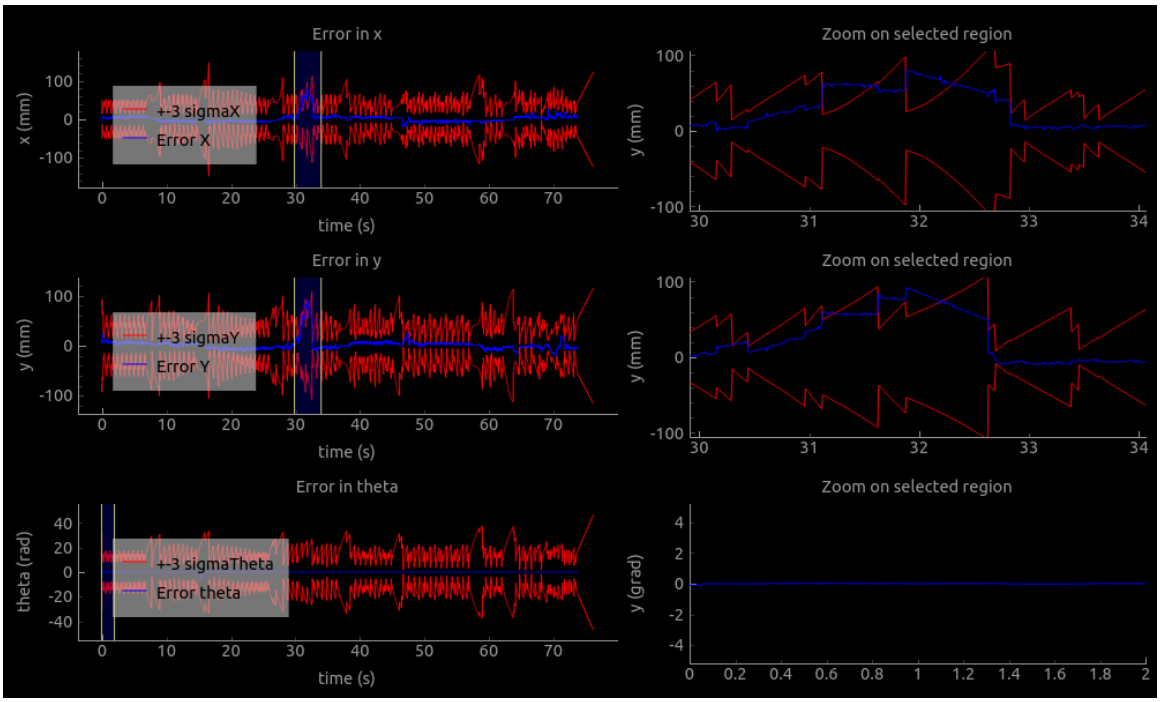


From a different point of view, using a lower frequency improves the estimation of the robot velocities. In fact, taking into account a greater time interval between two iterations, the effects of the noise are reduced, as it can be seen in the figure below.

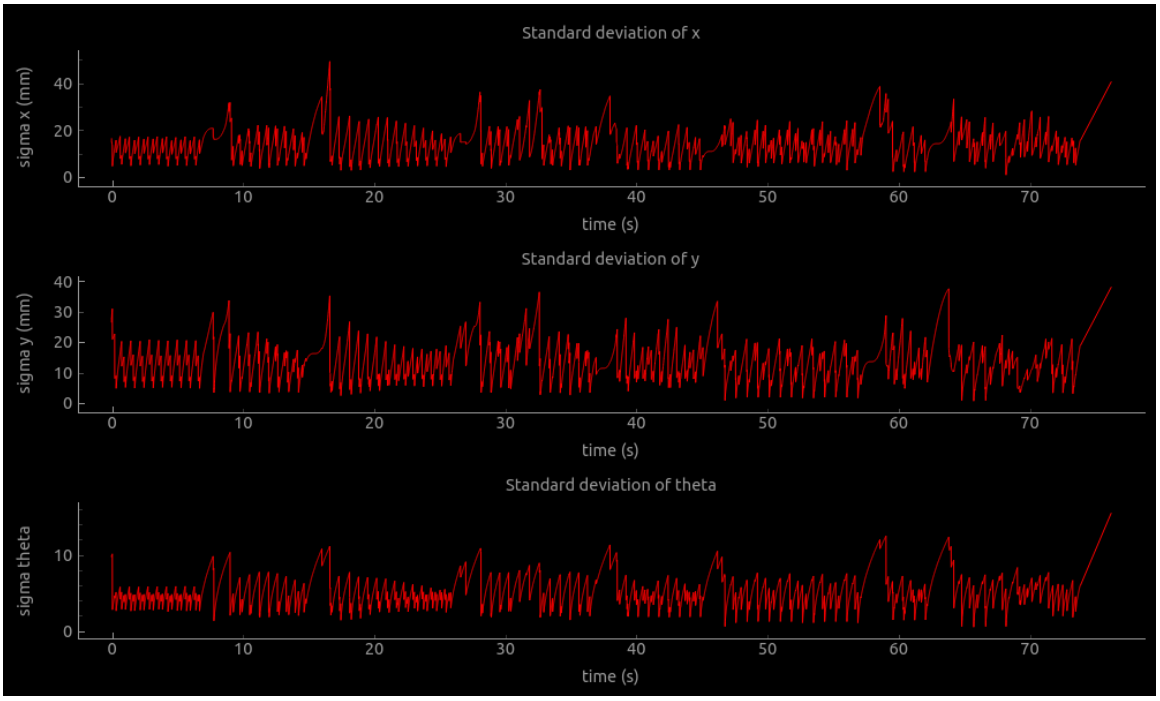


Last but not least, the estimator was tested on a long path. To be tested in this situation, the system must be repeatable, and it should ensure some condition to be satisfied. For example, the robot should move only diagonally, to always ensure both lines to be detected. Moreover, the errors in the model are bot reduced to 1%. With a frequency of half the simulation, the estimator has good performances along a 30 meters path, as shown in the following images.





In the previous image, the highlighted areas show an interval of approximately one second, where the errors are outside the range of $\pm 3\sigma$. This part corresponds to the path at around $x = -1.5$ and $y = -0.5$ where the estimated path is slightly different form the real one. However, the estimator does correct this error successfully. This also shows that this kind of estimator can recover from some errors that may occur when implemented on a real robot.



The previous test had also the intent of demonstrate that, if the optimal motion, *i.e.* diagonal motion, is ensured, the estimator is shown to be stable. On the other hand, the errors in the model and the frequency at which the estimator is executed should not be too severe. On the other hand, a good assembly should ensure small errors meanwhile the software component was proven to be able of running at high frequencies.

Appendix

Setting up a joystick

In order to set up a controller, to be used in ROS, is necessary to follow the steps reported in in this tutorial. However, the following reports the steps in detail.

First, connect the joystick to the computer and then check if it recognised by the computer. The command to run is:

ls /dev/input/

The output should be something like:

event0 event11 event14 event17 event2 event5 event8 js1 mouse1

In this case, the joystick is recognised as js1

Now, check if the *jstest* package is installed, just typing it in the command line. In the case it is not installed, it is just necessary to run these two lines:

sudo apt-get update -y

sudo apt-get install -y jstest-gtk

With the correct package it is possible to test the joystick status:

sudo jstest /dev/input/js1

The last command is necessary to make sure to have the controller working on Ubuntu. The buttons and axes status should change if the controller is used.

The ROS package which handle the joystick has to be installed:

sudo apt-get install ros-<distro>-joy

Writing the implemented ROS distribution instead of <distro>. Then, at this point, the permission to the joystick. Running the command:

ls -l /dev/input/js1

The output should be something like:

crw-rw-rw-+ 1 root input 13, 1 mai 16 19:55 /dev/input/js1

Here is important to notice the characters after *crw-rw-* it should be written *rw*; if not (it may be *crw-rw-r--+*) then run the command:

sudo chmod a+rw /dev/input/js1

Then check again with the previus command. At this point is possible to run the ROS node and check the output. First set the parameter:

rosparam set joy_node/dev "/dev/input/js1"

And then run the node:

rosrun joy joy_node

To check the output, a simple rostopic is sufficient:

rostopic echo joy

Now each time a button of a lever is pressed, moved or released; in the terminal a message should appear

Setting up PyQtGraph

PyQtGraph is an easy and intuitive interface. It is also said to be more interactive and fluid compared to MatPlot, it also works with python3. The package has to be installed because it is not build-in in Ubuntu.

To help the user in the installation, a tested procedure is reported below:

The first step is the suggestion to create a folder in the home named "program_sources" (or another desired name). Once inside the created folder the package has to be downloaded:

git clone https://github.com/pyqtgraph/pyqtgraph.git

The package is now downloaded in the computer and needs a setup. First, run the following:

python3 -m pyqtgraph.examples

A window should open, in contains a list of examples on the left and the related code on the right. The examples provide all the needed insight to create custom plots and multiple windows. However, if it doesn't work, that's because the package is in your computer but not installed. The problem should result in the following error:

user@user:~$ python3 -m pyqtgraph.examples

/usr/bin/python3: Error while finding module specification for 'pyqtgraph.examples'
(ModuleNotFoundError: No module named 'pyqtgraph')

The solution is to simply run a pip installation

sudo pip install pyqtgraph

If pip is not included in your computer, install it running:

sudo apt install python-pip

To test the package now run the command:

python3 -m pyqtgraph.examples

If it doesn't work, that's because PyQt5 is missing. It is thus needed to install it. Run the following:

sudo pip install pyqt5

At this point there are not any missing dependencies, and the package will open.