Drone Localization using the Extended Kalman Filter Method

Group10: João Ferreira, 78101 Pedro Mendes, 81046 Miguel Malaca, 81702 João Rosa, 84089

***Abstract*—Implementation of the Extended Kalman Filter for self localization of a drone that can only move in a 2D plane normal to the ground using ROS as the communication interface, python as the program main language and the Structure Sensor Camera and Inertial Measurement Unit to obtain data from the real world and to create the map.**

***Index Terms*—Extended Kalman Filter (EKF), Drone, ROS, Structure Sensor Camera, Inertial Measurement Unit (IMU), Yaw, Pitch, Pinhole, Projective Geometry.**

I. Introduction

Achieving full autonomy in a robotic system has been one of the main objectives in the area. The self-localization problem is, therefore, an important component in this matter. For a robot to be able to perform self-localization, i.e., find its own pose in the environment, it is required to compare the environment scan with a real map. If only with the odometry or some motion model it would never know how correct it is and would not be robust to the unmodeled changes (Kidnapping). One algorithm that can be used is the Extended Kalman Filter (EKF) with observations from a Depth Camera and IMU.

To solve this task using the Extended Kalman Filter, it was divided into subsections that together implement the method. Two alternatives for some models were studied, with or without the measure of the IMU.

II. Required elements

A. ROS

For the realization of this project, it was necessary to use *Robot Operating System* (ROS). The packages used are enumerated below:

- *openni2_camera and openni2_launch* - used to publish the frame of the depth camera;
- *imu_complementary_filter* - used to publish the camera's orientation (quaternions) using the information provided by the IMU (linear velocity, angular velocity and magnetic field)
- *hector_mapping* - used to create the map (without odometry);
- *depthimage_to_laserscan* - used to remove only the middle horizontal line from the camera image;
- *map_server* - used to create a file with the map information.

B. Map's Creation

In order to create the map, it was used three ROS packages, the *depthimage_to_laserscan*, *hector_mapping* and the *map_server*. Because of the absence of odometry, some restrictions were faced and to fight against these, it was used the *hector_mapping* package, which provides a SLAM approach without odometry. The depth camera was moved around the room, with a constant height, to scan the environment. The *depth_image_to_laserscan* package subscribes and converts a depth image published by the camera into a 2D laser scan (it obtains and publishes only the middle horizontal line of the image). The scan topic published by this package is subscribed by the *hector_mapping* package. This package created a 2D map. The *hector_mapping* requires a transformation matrix, *tf*, from the robot coordinates frame (base frame) to the camera coordinate frame. In this case, there was no rotation between the frames. With the aid of *rviz*, it was possible to visualize the mapping in real time. In the end, it was used the *map_server* to save the map and its information. The map obtained was an occupancy grid with a resolution of 0.05 meters/pixel. The map was obtained without odometry or IMU measures and therefore it had some errors due to the absence of orientation. To correct and improve the map, a program to edit the map was used. The resulting map is presented in figure 1.

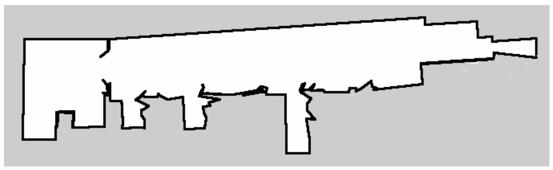


Fig. 1: Map created

The occupancy grid is a matrix that has the probability of each element. The map obtained has the probability in RGB code. When the map is opened in the beginning of the execution of the program, the occupancy grid is converted and it's obtained a matrix where the value 0 corresponds to free space, 1 to occupied space and finally -1 to unknown space.

III. Algorithm

A. Kalman Filter

The Kalman Filter is one stochastic filtering algorithm [1] that aims to achieve an estimate of the state of a linear time variant system considering noisy observations of this state. It represents the state by a Gaussian and also its observations and, afterward, intersects both in order to achieve the best estimation possible. Due to the fact that, in this case, the observations are not linear and if a Gaussian is applied the result will not be a Gaussian, a variant of the Kalman Filter was used: the Extended Kalman Filter.

B. Extended Kalman Filter

The Extended Kalman Filter (EKF) is used when the system is not linear. In order to use it, one must linearize the system and, afterward, the algorithm takes care of the nonlinear effects created. It is important to note that it was implemented the discrete version of the filter presented below. This algorithm is divided into two main steps: Prediction Step and Update Step. For each of them, there are different models which needed to be deducted.

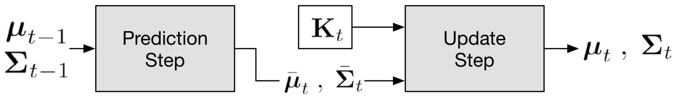

$\boldsymbol{\mu}_{t-1}$
$\boldsymbol{\Sigma}_{t-1}$
Prediction Step
$\mathbf{K}_t$
$\bar{\boldsymbol{\mu}}_t$, $\bar{\boldsymbol{\Sigma}}_t$
Update Step
$\boldsymbol{\mu}_t$, $\boldsymbol{\Sigma}_t$

Fig. 2: Extended Kalman Filter Chart Flow

C. Prediction Step

In this step, the EKF needs to predict the current state of the system. It creates an estimate that is modeled by a Gaussian following the equations 1 and 2.

$$\bar{\boldsymbol{\mu}}_t = \mathbf{A}_t\boldsymbol{\mu}_{t-1} + \mathbf{B}_t\mathbf{u}_t \tag{1}$$
$$\bar{\boldsymbol{\Sigma}}_t = \mathbf{A}_t\boldsymbol{\Sigma}_{t-1}\mathbf{A}_t^T + \mathbf{R}_t \tag{2}$$

$\mathbf{A}_t$ is the Covariance Matrix, $\boldsymbol{\mu}_{t-1}$ is the previous mean state. $\mathbf{B}_t$ is the Control Matrix and $\mathbf{u}_t$ is the Control Vector.

The equation 1 represents the expected value of the Gaussian that models the current state and the equation 2 represents the covariance matrix associated. In order to have the matrices needed, the motion model has to be derived from the movement laws of the system. The equations presented are the same as in the original Kalman Filter because the Motion Model is linear and, therefore, there is no need to linearize it.

D. Update Step

In this part, the predicted state in the previous Step is updated with respect to the observation. It compares what it is observing - Measurement Model - with what it should be observing if the predicted state was correct - Observation Model. This comparison is made through the Kalman Gain, calculated in equation 3. After computing it the update is made through the equations

$$\mathbf{K}_t = \bar{\boldsymbol{\Sigma}}_t\mathbf{H}_t^T(\mathbf{H}_t\bar{\boldsymbol{\Sigma}}_t\mathbf{H}_t^T + \mathbf{Q}_t)^{-1}, \tag{3}$$
$$\boldsymbol{\mu}_t = \bar{\boldsymbol{\mu}}_t + \mathbf{K}_t(\mathbf{z}_t - \mathbf{h}(\bar{\boldsymbol{\mu}}_t)), \tag{4}$$
$$\boldsymbol{\Sigma}_t = (\mathbf{I} - \mathbf{K}_t\mathbf{H}_t)\bar{\boldsymbol{\Sigma}}_t, \tag{5}$$

where $\mathbf{H}_t$ is the Jacobian of $\mathbf{h}$ and $\mathbf{Q}_t$ is the Confidance Matrix.

This last component of the algorithm compares the observations $\mathbf{z}_t$ with the predicted observations $\mathbf{h}(\bar{\boldsymbol{\mu}}_t)$. Finally, the Observation Model updates the estimated mean and covariance that will be fed back to the Motion Model for as long as needed.

IV. Implementation

A. Motion Model

Firstly, since the problem was approximated to a 2D problem, only 3 coordinates are needed to describe the state of the robot: the position (x, y) and the orientation (θ). To solve this problem, it was considered a 1^{st} order system and one without an input (u_t). This means that the velocity in each component is approximately constant, with some introduced Gaussian noise. However, this assumption doesn't model the reality perfectly, since the drone can have acceleration and it has not, in most cases, this uniform movement. This way, the state variables needed are, besides the position and orientation, the velocities in each of these components.

$$\mu_t = \begin{bmatrix} x_t \\ \frac{dx_t}{dt} \\ y_t \\ \frac{dy_t}{dt} \\ \theta_t \\ \frac{d\theta_t}{dt} \end{bmatrix} \tag{6}$$

$$\bar{\boldsymbol{\mu}}_t = \mathbf{A}_t\boldsymbol{\mu}_{t-1} + \varepsilon \tag{7}$$

$$\begin{cases} x_t = x_{t-1} + \frac{dx_{t-1}}{dt}\Delta t \\ \frac{dx_t}{dt} = \frac{dx_{t-1}}{dt} + \varepsilon_x \\ y_t = y_{t-1} + \frac{dy_{t-1}}{dt}\Delta t \\ \frac{dy_t}{dt} = \frac{dy_{t-1}}{dt} + \varepsilon_y \\ \theta_t = \theta_{t-1} + \frac{d\theta_{t-1}}{dt}\Delta t \\ \frac{d\theta_t}{dt} = \frac{d\theta_{t-1}}{dt} + \varepsilon_\theta \end{cases} \tag{8}$$

Using the previous equations, it's possible to determine the matrix in 9.

$$\boldsymbol{A} = \begin{bmatrix} 1 & \Delta t & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & \Delta t & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & \Delta t \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \tag{9}$$

The ε represented in 7 represents a Gaussian distribution with 0 average. This means that does not have influence in the average of the prediction of μ_t. However, it has some covariance that should be noticed. Since it is only applied to the velocities, its covariance matrix is the one in 10 and it is the matrix R that the algorithm needs.

$$\text{R} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & \sigma_x^2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & \sigma_y^2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & \sigma_\theta^2 \end{bmatrix} \tag{10}$$

The values for the covariances should be well chosen in order to make the movement less uniform and closer to the reality.

B. Measurement Model

1) Distance: The main sensor available to measure the current state of the system was an RGB-D camera whose purpose was to measure distances from an imaginary plane orthogonal to the orientation of the drone and the environment it is in. Once the camera captures an image with 640x480 pixels, the proper part of it has to be taken, due to the 2D approach to the problem. In a fully 2D approach, the only angle that changes in the system is the one around the Z-axis - *yaw*. In this case, only the horizontal line in the middle of the image is taken.

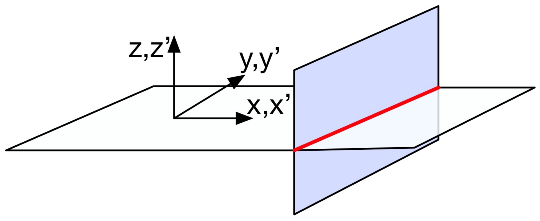


Fig. 3: In red: the array that contains the world's horizontal component seen by the drone

Although, this project is not a fully 2D because the drone, when moving, tilts. The new configuration of its frame is given by an IMU. Since there is no Gimbal available, the camera will have the same behavior. Anyway, the line that represents the horizontal plane has to be obtained. In order to do that, the image plane was intersected with the horizontal plane, because the projective geometry is used and the model is the pinhole. This is done taking out the yaw because it doesn't affect the line that should be obtained.

$$x.C = x.P \Leftrightarrow (1,0,0).(f,0,0) = (1,0,0).(x,y,z) \\ \Leftrightarrow x = f \tag{11}$$

$$^C_W e_z.(0,0,0) = ^C_W e_z.(x,y,z) \Leftrightarrow ^C_W Re_z.(x,y,z) = 0 \\ where,\ ^C_W R = ^W_C R^T \tag{12}$$

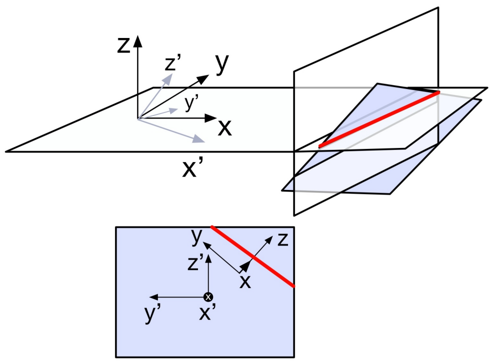


Fig. 4: In red: the array of pixes that contains the world horizontal component of the plane when the drone gets a YX rotation

In expression in 11 there is the equation of the image plane where f represents the focal length. In the equation 12 there is the horizontal plane in the world with respect to the camera frame. $^W_C R$ is given by the IMU. In the figure 4 this intersection is represented by the red line. The image plane is the blue one and the frame XYZ is associated with the world and the X'Y'Z' is representing the frame attached to the camera/drone.

Whit this procedure the new coordinates of the horizontal line were determined. As the camera measures the depth, the orthogonal distance of the imaginary camera plane to these points can be directly obtained by the values of the pixel. However, this is not the real distance that needs to be computed. Instead, one should transform the camera coordinates to world coordinates, and get the coordinate X. It can be done, once more, taking into account that the camera uses the projective geometry.

$$w \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \mathbf{K} \begin{bmatrix} \mathbf{R} & \mathbf{0} \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} \tag{13}$$

The equation 13 presents the coordinate tranformation between the camera and the world. The (u, v) are the image coordinates and w is the depth measured. The (X, Y, Z) are the coordinates in the world frame. The matrix **K** is the one with the intrinsic parameters of the camera and the **R** represents the rotation with respect to the world frame. The **0** represents the translation between the two frames. It was assumed that this world frame has the same configuration as the true world frame, but it is always centered in the same point as the camera, for the sake of simplicity. This procedure works under the assumption that the horizontal plane never leaves the camera field of view.

2) Orientation: In a different variant of this project, the other sensor used was the IMU which measures the linear velocity, the angular, and the magnetic field. These measures are published by the "IMU node". These topics are subscribed by the ROS package *imu_complementary_filter*, and this publishes the orientation in quaternions. Using the quaternions obtained, the rotation matrix is calculated using the function "*quat2mat*" of the "*Transform3d*" python's package. This one gives the configuration of the axes attached in relation to the ones in the world. This way one can get the orientation measure by computing the angle in the Z axis of the world frame between the heading of the drone and the X axis of the world. To do that it's determined the projection of the heading axis in the plane XY and the X axis of the world. With both projections, the angle can be easily computed through trigonometric identities.

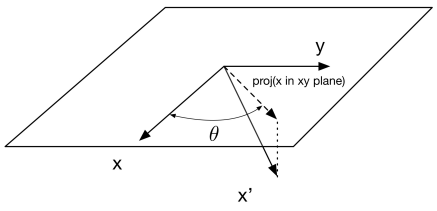


Fig. 5: $Angle\theta$

$$\theta = \arccos \frac{||proj(x', x)||}{||proj(x', xy)||} \tag{14}$$

C. Observation Model

The observation model tries to predict what the robot should see when its position and orientation are the ones predicted by the prediction step of EKF. The goal is to determine the distances between the imaginary plane orthogonal to the camera's principal point and the obstacles seen in the camera's field of view, as seen on figure 6.

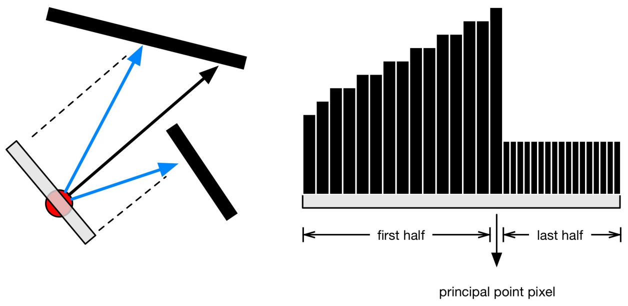


Fig. 6: On the left: drone's camera field of view marked with the blue vectors, and the black rectangles correspond to obstacles. On the right: visual representation of the values stored in each element of the camera's sensor array.

On figure 6, the drone is represented by the red circle, the imaginary camera plane is the grey rectangle, and the black bars correspond to an obstacle. The blue vectors correspond to the limit of the field of view. The first half of the vector stores information relative to the drone's left field of view and the last half stores the remaining distances in the field of view. When the drone is close to an obstacle (for example, on the right half of its field of view), the vector seems misaligned with the camera's optic center, however, this is not true. There is never a shift between this two. The closer an object is to the drone, the smaller the portion of the imaginary plane gets, despite the number of elements in the plane vector does not change.

The prediction step of the EKF determines the predicted robot position in the map (x_r and y_r), the velocity ($\dot{x}_r$ and $\dot{y}_r$), the orientation (θ_r) of the robot and the angular velocity ($\dot{\theta}_r$). With this information, the robot is placed on the map on position $p_r = (x_r, y_r)$. All the angles are given in radian and are defined between $]-\pi, \pi]$. The vector containing the projective distances that are obtained from the map needs to have the same number of elements of the vector obtained by the camera's frame.

The camera has a field of view of $58°$, that means it captures $29°$ for each side of the center of the frame. Using this information and knowing the number of pixels obtained on a frame in the measurement model, it's possible to determine the increment of each angle (α).

$$\alpha = \frac{\frac{29\pi}{180}}{\frac{no.pixels}{2}} \tag{15}$$

In each iteration, the new angle (β) between the robot position p_r and the points of the map, that the camera is measuring, is calculated. The orientation for each point is given by $\beta = \theta - \alpha$ for one side of the view and $\beta = \theta + \alpha$ for the other side.

For each β and using small increments (x_{incr}) of the coordinate x, the position (x, y) is calculated.

$$x = x + x_{incr} \tag{16}$$

$$\begin{aligned} \tan\beta &= \frac{y - y_r}{x + x_{incr} - x_r} \\ \Leftrightarrow y &= y_r + (x + x_{incr} - x_r)\tan\beta \end{aligned} \tag{17}$$

The increment x_{incr} depends of the value of β. If β is approximately $\frac{\pi}{2}$ or $-\frac{\pi}{2}$, the increment is small, because the slope of the tangent function is very high. If β is equal to $\frac{\pi}{2}$ or $-\frac{\pi}{2}$, the y is incremented or decremented by 1, respectively.

In each iteration, the new position (x, y) is verified in the map. If this position is free space the iteration continues and there is an increment of x and the new value of y is calculated. If the position (x, y) is occupied the iteration finish. After determining the coordinate of this point that the robot is seeing, $p_p = (x_p, y_p)$, using this iterative method, the radial distance, dr, between the initial and final points, is calculated using the Euclidean norm. To calculate the real distance, the Euclidean norm needs to be multiplied by the map's resolution.

$$dr = \|p_p - p_r\| \cdot resolution \tag{18}$$

Knowing this radial distance, the point p_r can be given in function of the position and orientation of the drone. However, as the radial distance dr is calculated in the real coordinates and not in the map coordinates, a transformation has to be done between the map coordinates and the real coordinates. Considering the point $p_p^* = (x_p^*, y_p^*)$ and the point $p_r^* = (x_r^*, y_r^*)$ that are the point p_p and p_r expressed in the real coordinates, respectivelly.

$$x_r^* = x_r \cdot resolution \tag{19}$$

$$y_r^* = y_r \cdot resolution \tag{20}$$

$$x_p^* = x_r^* + dr \cdot \cos\beta \tag{21}$$

$$y_p^* = y_r^* + dr \cdot \sin\beta \tag{22}$$

The angle between orientation and the image plane is always $\frac{\pi}{2}$. Let's call φ to the angle between the image plane and β. The angle φ is given by

$$\varphi = \frac{\pi}{2} - \alpha \tag{23}$$

The point $p_i = (x_i, y_i)$ is the projection of point p_p on the image plane in the map coordinates. This point expressed in the real coordinates is $p_i^* = (x_i^*, y_i^*)$. The real distance between the point p_r^* and p_i^* is

$$c = dr \cdot \cos\varphi \tag{24}$$

Since the image plane is orthogonal to the orientation, the angle of the image plane (γ) is

$$\gamma = \theta - \frac{\pi}{2} \tag{25}$$

The coordinates of point p_i^* can be determine using equation 26 and equation 27.

$$x_i^* = x_r^* + c \cdot \cos\gamma \tag{26}$$

$$y_i^* = y_r^* + c \cdot \sin\gamma \tag{27}$$

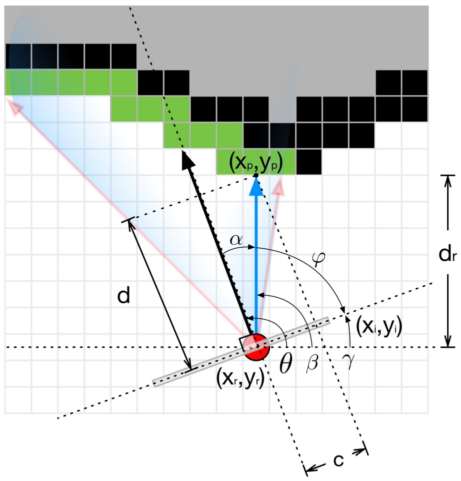


Fig. 7: Observation model

The distance, determined by this observation model, between the point p_i^* and the p_p^* is given by

$$h(\overline{\mu}) = \left\| p_p^* - p_i^* \right\| = \tag{28}$$
$$= \sqrt{(x_r^* + dr\cos(\beta) - x_i^*)^2 + (y_r^* + dr\sin(\beta) - y_i^*)^2}$$

The equation 28 needs to be calculated for each measure of the camera. The observation model is a non-linear function to calculate the projective distances between the image plane and the obstacles. To compute the EKF algorithm, a linearization on the system has to be done. This linearization is done using the first order approximation of Taylor series. For that, a Jacobian of h needs to be determined.

$$H = \begin{bmatrix} \frac{\partial h_{d1}(\overline{\mu})}{\partial x_r} & \frac{\partial h_{d1}(\overline{\mu})}{\partial \dot{x}_r} & \frac{\partial h_{d1}(\overline{\mu})}{\partial y_r} & \frac{\partial h_{d1}(\overline{\mu})}{\partial \dot{y}_r} & \frac{\partial h_{d1}(\overline{\mu})}{\partial \theta_r} & \frac{\partial h_{d1}(\overline{\mu})}{\partial \dot{\theta}_r} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ \frac{\partial h_{dn}(\overline{\mu})}{\partial x_r} & \frac{\partial h_{dn}(\overline{\mu})}{\partial \dot{x}_r} & \frac{\partial h_{dn}(\overline{\mu})}{\partial y_r} & \frac{\partial h_{dn}(\overline{\mu})}{\partial \dot{y}_r} & \frac{\partial h_{dn}(\overline{\mu})}{\partial \theta_r} & \frac{\partial h_{dn}(\overline{\mu})}{\partial \dot{\theta}_r} \end{bmatrix} \tag{29}$$

where,

$$\frac{\partial h_{d1}(\overline{\mu})}{\partial x_r^*} = \frac{x_r^* + d_r\cos(\beta) - x_i^*}{\sqrt{(x_r^* - d_r\cos(\beta) - x_i^*)^2 + (y_r^* + d_r\sin(\beta) - y_i^*)^2}} \tag{30}$$

$$\frac{\partial h_{d1}(\overline{\mu})}{\partial \dot{x}_r^*} = 0 \tag{31}$$

$$\frac{\partial h_{d1}(\overline{\mu})}{\partial y_r^*} = \frac{y_r^* + d_r\sin(\beta) - y_i^*}{\sqrt{(x_r^* - d_r\cos(\beta) - x_i^*)^2 + (y_r^* + d_r\sin(\beta) - y_i^*)^2}} \tag{32}$$

$$\frac{\partial h_{d1}(\overline{\mu})}{\partial \dot{y}_r^*} = 0 \tag{33}$$

$$\frac{\partial h_{d1}(\overline{\mu})}{\partial \theta_r^*} = \frac{(x_r^* + d_r\cos(\beta) - x_i^*)(-d_r\sin(\beta) + (y_r^* + d_r\sin(\beta) - y_i^*)(d_r\cos(\beta))}{\sqrt{(x_r^* + d_r\cos(\beta) - x_i^*)^2 + (y_r^* + d_r\sin(\beta) - y_i^*)^2}} \tag{34}$$

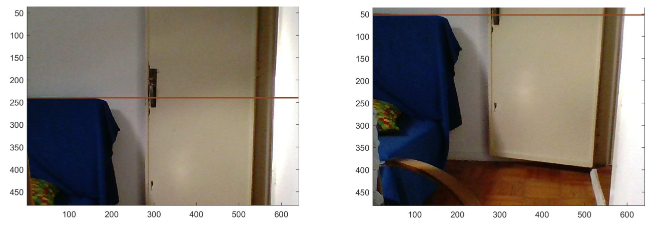
$$\frac{\partial h_{d1}(\overline{\mu})}{\partial \dot{\theta}_r^*} = 0 \tag{35}$$

In order to improve the results, a second alternativity implementation is proposed, where the orientation calculated by the IMU is compared with the orientation predicted by the EKF. The last measure of function $h(\mu)$ is the predicted orientarion. The last row of the jacobian matrix is the parcial derivatives of orientation in relation to the predicted state.

V. Experimental Results

A. Camera Rotation

In order to validate the measurement model related to the distance, a camera rotation was simulated and the pixels corresponding to the horizontal plane were obtained.



(a) No rotation (b) Pitch rotation

Fig. 8: Difference between the horizontal plane in the absence (A) and presence (B) of a rotation

The figure 8 represents which line of the image represents the horizontal plane. In the left one no rotation was applied and, therefore, the line obtained was in the middle. As for the right, a pitch rotation of 18º was applied and, as expected, the line obtained was closer to the top.

B. Complexity

Firstly, the complexity of the algorithm was studied. In this case, the most significant part in terms of time and complexity is the observation model, due to the high number of iterations performed. It depends on the number of points obtained from the image in the camera and the distance from the camera to the obstacle. In the figure 9 it is represented the time elapsed in the observation model depending on the number of points obtained from the camera's image.

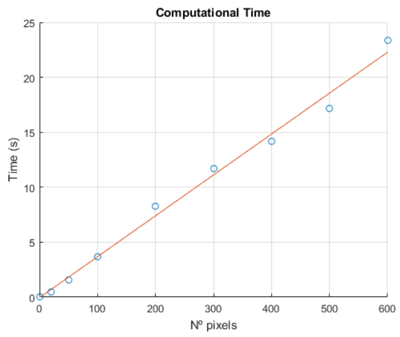


Fig. 9: Time elapse in the observation model depending of the number of pixels

C. Simulation

After the implementation of the algorithm previously explained (using the first alternative where the localization algorithm does not use the data from IMU), some simulations were performed, and it was not possible to find a combination of initial conditions and covariance values that would make the algorithm converge to the real pose of the robot, as it can be seen in the figure 10. For all the simulations the fully 2D problem was considered.

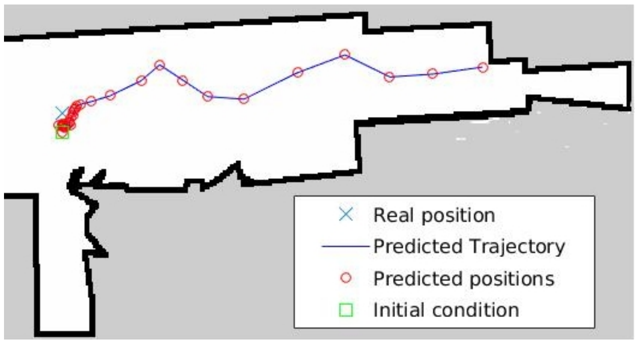


Fig. 10: Simulation Performed

In order to get some better results the second alternative was used. The *yaw* rotation was used in the observation. With this new model some tests in different maps were made.

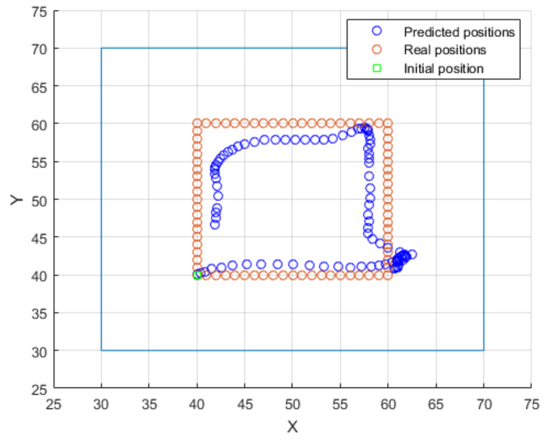


Fig. 11: Simulation Performed in a square map

In this test, it was used a square map. It is not the best one since there are a lot of points where, with the same orientation, the distance are the same. Anyway, the result shows that the prediction is following the reality.

Next, some simulations were made using the real map created. To obtain accurate results, an adjust of the covariance had to be made. The values of the covariance used in the simulation are

$$R = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} Q = \begin{bmatrix} 10 & 0 & \dots & 0 \\ 0 & 10 & \ddots & 0 \\ \vdots & \ddots & \ddots & 0 \\ 0 & \dots & 0 & 1 \end{bmatrix} \tag{36}$$

$$\Sigma_0 = \begin{bmatrix} 10 & 0 & 0 & 0 & 0 & 0 \\ 0 & 10 & 0 & 0 & 0 & 0 \\ 0 & 0 & 10 & 0 & 0 & 0 \\ 0 & 0 & 0 & 10 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \tag{37}$$

One of the results is presented in the figure 12. The result is good and the predicted trajectory is close to the real one.

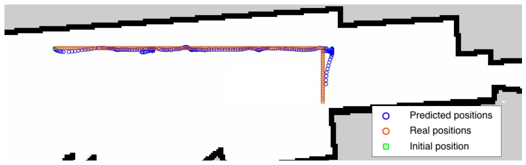


Fig. 12: Simulation Performed in a square map

D. Real Time

To take real time results, it was used an IMU connected to a raspberry pie, simulating the drone. The IMU was placed above the depth camera, on a structure created for the occasion, and both had the same orientation. During the tests, we could never make the Extended Kalman Filter converge, using both variants. This can be explained by two main reasons. The first one related to the map because there is a big difference when compared to the reality. As during the mapping, it was not used any kind of odometry, the map obtained had some errors and inaccuracies, so it was modified to create a better map. However, this map does not correspond exactly to reality. This way, the EKF will have lots of difficulties when trying to converge. The other main problem comes from the IMU. It was found that the angle in *yaw* was constantly drifting, due to errors of the magnetometer, making it impossible to have a proper measure. Another reason that could lead to this situation was the fact that the camera could only measure distances bigger than 40/50 cm which makes it hard since the drone could not go too close to the obstacles.

VI. Conclusion

As said before, a lot of problems were faced while implementing and testing the algorithm. The fact that, using the variant without the IMU, the model couldn't converge in simulations leaves the impression that some part of the algorithm could be implemented in a different way that could better describe the problem in question. In this case, some assumptions, different from the reality, were made under the observation model and they can make the results become worst. As for the variant with the IMU, there are some interesting results. Firstly, the effects of choosing the movement model of first order can be detected. The predicted state shows, sometimes, a behavior compared to a uniform movement. This was handled by changing the covariances associated with velocities, simulating a possible acceleration. In this case, a model that takes the pitch and roll in the account during the motion model would probably be better. The algorithm wasn't also very robust since there were many difficult trajectories that made the algorithm diverge. Apart from this, it is able to successfully convert the 3D characteristics to 2D even with the tilting that is present in the movement. The results in the simulation were also good, it can be well localized in the cases tested.

References

[1] Rodrigo Ventura, *Derivation of the discrete-time Kalman filter* Lisbon, Portugal: Instituto Superior Técnico, 2017.

[2] Maria Isabel Ribeiro, *Kalman and Extended Kalman Filters: Concept, Derivation and Properties* Lisbon, Portugal: Instituto Superior Técnico, 2004.