EMARO+ M1

European Master on Advanced ROBotics

Project report

Turtlebot localization with visual markers

Author:
Kevin Serrano
Mahmoud Ali

Supervisor:
Olivier Kermorgant

June 27, 2017

Contents

1 Introduction **3**
- 1.1 Localization . . . 3
- 1.2 Turtlebot . . . 3
- 1.3 OmniCamera . . . 5
- 1.4 Beacons . . . 6
- 1.5 Environment . . . 6

2 Image processing **7**
- 2.1 Image rectifying . . . 7
- 2.2 Image tiling . . . 8
- 2.3 Color segmentation . . . 9
  - 2.3.1 The different color spaces . . . 9
  - 2.3.2 Best color space . . . 11
  - 2.3.3 Thresholding . . . 12
  - 2.3.4 Implementation results . . . 13
- 2.4 Blob detection . . . 15
- 2.5 Distance calculation . . . 17
- 2.6 Time performance . . . 18
- 2.7 ROS . . . 18

3 Absolute localization **20**
- 3.1 Trilateration . . . 20
- 3.2 Optimization techniques . . . 21

4 Hybrid localization with EKF **22**
- 4.1 EKF introduction . . . 22
- 4.2 Odometry . . . 25
  - 4.2.1 Odometry error and uncertainty . . . 27
  - 4.2.2 Odometry Interface . . . 28
- 4.3 EKF Algorithm . . . 29
  - 4.3.1 covariance tuning: . . . 32
  - 4.3.2 Results: . . . 33

5 Visualization with Rviz **35**
  - 5.0.1 Static transform publisher . . . 35
- 5.1 Results . . . 36

6 ROS package turtle_ekf **37**
- 6.1 image_rectifier Node . . . 37
- 6.2 image_tiles Node . . . 37
- 6.3 blob_detector Node . . . 37
- 6.4 blob_dist_interface Node . . . 38
- 6.5 odom_interface Node . . . 38
- 6.6 ekf_node Node . . . 38
- 6.7 turtle_ekf/Pose2DWithCovariance Message . . . 38

7 Future work **39**
- 7.1 Improving image processing . . . 39

7.2 Advanced EKF 39
7.3 Azimuth angles 39
7.4 Different localization Methods 39

Chapter 1

Introduction

1.1 Localization

The robot localization problem is key in making truly autonomous robots. If a robot isn't able to determine where it is, it can be difficult to decide on what to do next. In order to localize itself, a robot needs to have has access to relative and absolute measurements giving the robot feedback about its driving actions and the situation of the surrounding environment. Given this information, the robot has to determine its location as accurately as possible. The difficult part is the existence of uncertainty in both the driving and the sensing of the robot. These must be combined in an optimal way [1].

1.2 Turtlebot

TurtleBot is a low-cost, personal robot kit with open-source software. TurtleBot was created at Willow Garage by Melonee Wise and Tully Foote in November 2010. With TurtleBot, you'll be able to build a robot that can drive around your house, see in 3D, and have enough horsepower to create exciting applications [2].

Currently there are 3 versions of the TurtleBot in the market. The one we'll be using was released in the year 2012 and corresponds to the second generation. The main hardware includes:

- Kobuki Base
- Asus Xion Pro Live
- Netbook (ROS Compatible)
- Kinect Mounting Hardware
- TurtleBot Structure
- TurtleBot Module Plate with 1 inch Spacing Hole Pattern

Regarding the software, we'll be using the open-source ROS developed packages available online. We'll be mostly needing the odometry information for localization purposes, therefore we need to be able to read and extract information from its built-in sensors.

A brief installation guide is presented in order to use the TurtleBot with the ROS kit but further documentation can be found here. The first thing to do is download the debs by issuing the following command:

```
> sudo apt-get install ros-<ubuntu-distribution>-turtlebot
```

If one wants to use its own laptop to connect directly with the TurtleBot it may be necessary to setup `udev` rules to work with the Kobuki base.

```
> . /opt/ros/<ubuntu-distribution>/setup.bash
> rosrun kobuki_ftdi create_udev_rules
```

Network configuration plays an important role for being able to remotely control the robot and for having access to all the published topics from the TurtleBot. For this we must know the ip addresses of both the turtlebot and our pc. `ifconfig` provides us with this information.

Secondly, we'll need to establish an ssh connection to launch/run files remotely.

Figure 1.1: TurtleBot2.

```
> sudo apt-get install openssh-server
> ssh username@<TURTLEBOTP_IP>
```

Next up are the `ROS_MASTER_URI` and `ROS_HOSTNAME` environment variables which are used to identify the ROS master and should be exported to the workspace setup script. The laptop connected to the TurtleBot must be the ROS master.

```
> echo export ROS_MASTER_URI=http://IP_OF_TURTLEBOT:11311 >> ~/turtlebot/devel/setup.sh
> echo export ROS_HOSTNAME=IP_OF_TURTLEBOT >> ~/turtlebot/devel/setup.sh
```

On the remote PC we must also set up and export these variables but the `ROS_MASTER_URI` should contain the turtlebot IP address!

```
> echo export ROS_MASTER_URI=http://IP_OF_TURTLEBOT:11311 >> ~/.bashrc
> echo export ROS_HOSTNAME=IP_OF_PC >> ~/.bashrc
```

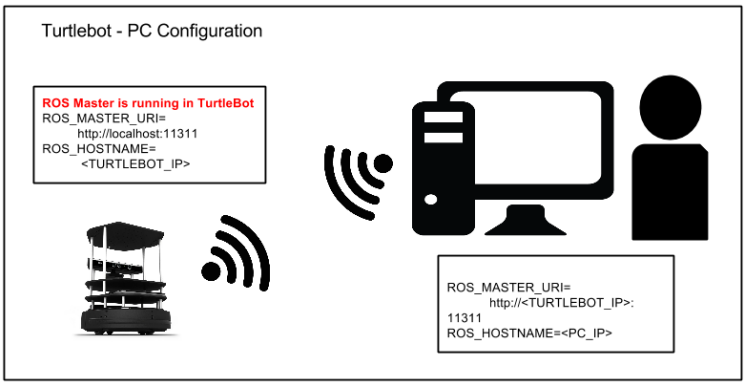


Figure 1.2: Natural habitat.

If everything went correct, one should see the list of topic when running the following command from a **new** terminal.

```
> rostopic list
```

After verifying network connection we can finally bring the TurtleBot to life. Connect the USB cables to the TurtleBot laptop and press the power button. Close the lid of the laptop and place it on the TurtleBot.

Firstly, ssh from your PC to the turtlebot and **source your setup.bash** and launch the bringup instructions.

```
> source /opt/ros/<ros\_distro>/setup.bash
> roslaunch turtlebot_bringup minimal.launch --screen
```

We can now run keyboard remote control and move freely the TurtleBot.

```
> roslaunch turtlebot_teleop keyboard_teleop.launch --screen
```

1.3 OmniCamera

Omni 60 captures 360° panoramic video at 60Hz, for applications including robotics localization and mapping, telepresence, videography, augmented reality, and surveillance. Available models: monochrome, color [3].

- 1.8 MP raw video
- 60 Hz synchronized capture
- Real-time panorama
- Compatible with OpenCV and **ROS**
- C, C++ open-source SDK
- Windows 7, 8, 10, and **Linux 3.x**

Figure 1.3: Occam omni 60 camera.

We're particularly interested in exploiting the ROS compatibility of the camera and we provide a brief installation guide. More information available here.

The software can be downloaded from the official website. Conveniently for us, our supervisor, Olivier Kermorgant, has already a ROS package available in his github repository with some extra features added such as available launch files and the calibration parameters for each one of the five cameras along with a script for publishing them through ROS topics. We simply clone this package (either the official or the ECN one) to our ROS workspace and execute the `catkin_make` command.

The tricky part is getting the camera to work with the computer since an `occam.rules` file must be generated since we must allow non.root access to this device. In order to create the file we can type in:

```
> sudo gedit /etc/udev/rules.d/occam.rules
```

This, of course, requires root access. Once having created the file we need to add the following line:

```
SUBSYSTEM=="usb", ATTRS{idVendor}=="285e", ATTRS{idProduct}=="3efd", MODE="0666"
```

And lastly restart the udev service

```
> sudo service udev restart
```

1.4 Beacons

(a) blue beacon **(b)** orange beacon **(c)** green beacon

Figure 1.4: Beacons

Absolute localization of the robot at 2D space requires at least three beacons to derive the robot 2D pose using triangulation or trilateration. the type of the beacons to be used should be determined according to the sensors that will be used to detect them. our objective is to use vision to detect the beacons, and in order to make the process of the beacon detection more reliable and more simple we chose the beacons to have distinct color and the shape. it is easier to to detect color objects using a camera. besides that each beacon has its own color (blue, green, red), they have spherical shape. the advantage of the sphere that its projection in any plan is always a circle. all the beacons are sphere with diameter equal to 45 cm.

1.5 Environment

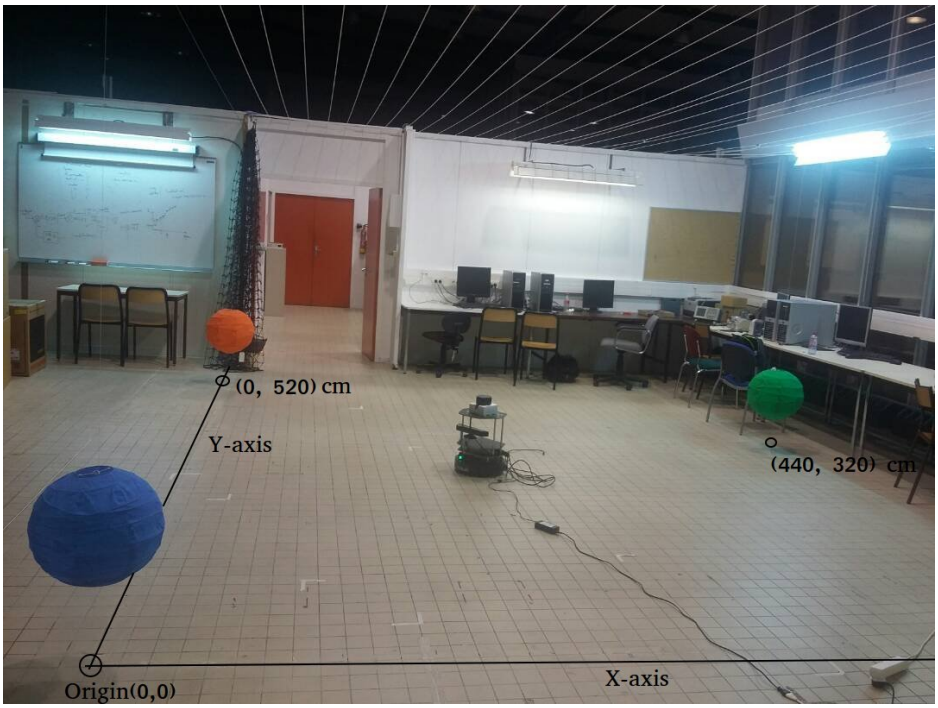


Figure 1.5: Robot Environment

The envireonment is the project room, the ground is a mosaic consist of 10x10 cm flagstones. the mosaic is used as grid to determine the coordinate of the robot. the beacons are located at fixed coordinates:

- Blue Beacon is located at the origin(0,0).
- Orange Beacon is located at point (440,320).
- Green Beacon is located at point (0, 520).

The the beacons are set at a hight 50 cm -which is along z axis- to be at the same level of the omnicamera.

Chapter 2

Image processing

Since we'll be using ROS for this project, we'll be making use of the OpenCV package that comes along with it. OpenCV (Open Source Computer Vision Library) is an open source computer vision and machine learning software library. It has C++, C, Python, Java and MATLAB interfaces and supports Windows, Linux, Android and Mac OS. It was designed for computational efficiency and with a strong focus on real-time applications [4].

2.1 Image rectifying

Images taken from cameras are captured through a lens which usually add distortion to the image. Of course there exist different cameras for specific purposes and the lenses vary from one another. Rectifying an image consists on removing this distortion. But, how to remove it?

Camera calibration is a widely used process in image processing and computer vision for estimating the parameters of a lens and image sensor of an image or video camera. Once having these parameters we can get the so called distortion coefficients [5].

The pin-hole camera model can be used to calibrate most cameras since, in this model, a scene view is formed by projecting 3D points into the image plane using a perspective transformation.

$$s\, m' = K[R|t]M'$$

$$s \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_2 \\ r_{31} & r_{32} & r_{33} & t_3 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} \tag{2.1}$$

where:

- s is just a scaling factor in order to extract u and v.
- (u, v) are the coordinates of the projection point in pixels.
- K is a camera matrix (intrinsic parameters).
- $[R|t]$ a matrix composed of a rotation and translation (extrinsic parameters).
- (f_x, f_y) are the focal lengths expressed in pixels
- (c_x, c_y) is the principal point, usually at the center of the image.
- (X, Y, Z) coordinates of a 3D point expressed in world coordinates.

The matrix of extrinsic parameters doesn't depend on the scene viewed. So, once estimated it can be re-used as long as the focal length is fixed (no zoom). However, if the camera's image is scaled or the resolution changes, these must be scaled accordingly. The extrinsic parameters matrix, also referred to as the projection matrix, is used to describe the camera motion around a static scene or vice versa. The transformation above is equivalent to the following (when $z \neq 0$):

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = R \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} + t \tag{2.2}$$

where:

- $x' = x/z$
- $y' = y/z$
- $u = x' f_x + c_x$
- $v = y' f_y + c_y$

Now for th eimportant part, most real lenses have radial distortion and slight tangential distortion. So the above model is extended to:

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = R \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} + t \tag{2.3}$$

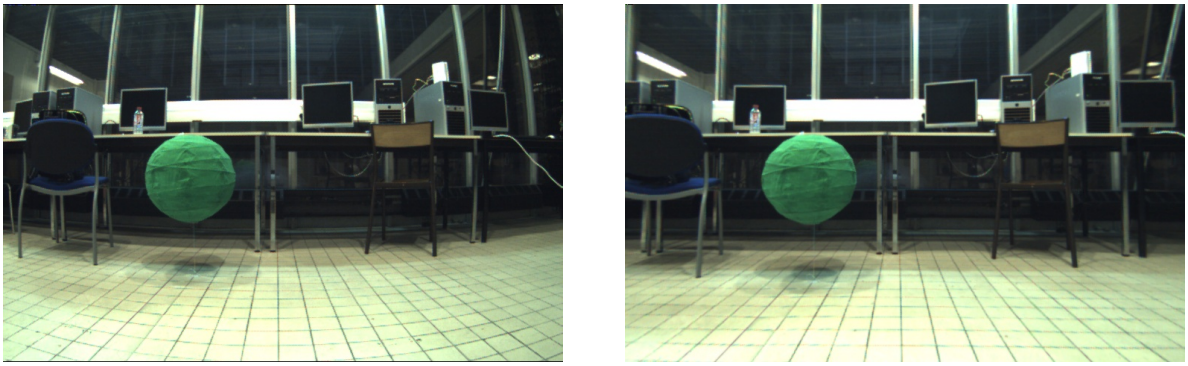
where:

- $x' = x/z$
- $y' = y/z$
- $x'' = x' \frac{1+k_1 r^2 + k_2 r^4 + k_3 r^6}{1+k_4 r^2 + k_5 r^4 + k_6 r^6} + 2p_1 x' y' + p_2(r^2 + 2x'^2) + s_1 r^2 + s_2 r^4$
- $y'' = y' \frac{1+k_1 r^2 + k_2 r^4 + k_3 r^6}{1+k_4 r^2 + k_5 r^4 + k_6 r^6} + 2p_2 x' y' + p_1(r^2 + 2y'^2) + s_1 r^2 + s_2 r^4$
- $r = x'^2 + y'^2$
- $u = x'' f_x + c_x$
- $v = y'' f_y + c_y$

The radial distortion coefficients are k_1, k_2, k_3, k_4, k_5 and k_6. p_1 and p_2 are tangential distortion coefficients and s_1, s_2, s_3 are the thin prism distortion coefficients.

Even though the extended model has some messy equations, the concept of rectifying an image is relatively easy to grasp. When having a distorted image we get (u, v) depending on (x'', y'') and we would like to map them such that they depend on (x', y'). Calibrating the camera provides us with the distortion coefficients that allows us to make this rectification.

Since the omnidirectional camera is already calibrated, we just need to extract the camera matrix and the distortion coefficients to carry out the geometric transformation and obtain the undistorted image. For this we use the `undistort` OpenCV function [6] inside a dedicated ROS node that subscribes to an image and its `camera_info` (message containing the calibration parameters) and publishes the undistorted image.



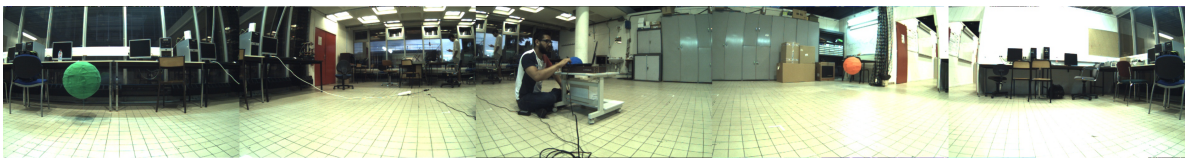
(a) Distorted (raw) image **(b)** Undistorted image

Figure 2.1: Image rectification with calibration parameters.

2.2 Image tiling

Since we are using a 360° panoramic camera we would like to obtain a single panoramic image from it. The ROS package developed for the Omni 60 camera provides numerous image topics, among these we can find a stitched image topic and an image tiles one. The latter image is the one that results more handy for us but has a main drawback, the tiled images are undistorted and one cannot simply perform a rectification along the tiled image as a whole.

(a) Stitched image (with some flaws)



(b) Distorted tiled images

Figure 2.2: Images published from Occam driver.

To solve this we must rectify each individual image and then tile them together. This is a fairly simple process, we must create an image big enough to hold the 5 images coming from the camera and insert each image at its corresponding place. A ROS node was created for this specific purpose, subscribing to the 5 rectified images and publishing the tiled one.

Figure 2.3: Undistorted tiled images

2.3 Color segmentation

The beacons used for this experiment are composed of three different colors such that we can identify each on of them by means color segmentation. For these we must make use of **color spaces** [7].

2.3.1 The different color spaces

This section covers some of the most important color spaces used in computer vision. No theory will be described but we'll present some basic intuition and main characteristics for each one of them which will be useful in making and justifying decisions later on.

Two images of the same object (Rubik's cube) will de processed, one of them taken under outdoor conditions (bright sunlight) and the other one with normal indoor lighting. We can convert between different color spaces using the OpenCV function `cvtColor()`.

Figure 2.4: Same object under different illumination.

RGB color space

The RGB, normally the default color space used to represent images, has the following properties.

- Additive color space where colors are obtained by a linear combination of red, green and blue values.
- The three channels are correlated by the amount of light hitting the surface.

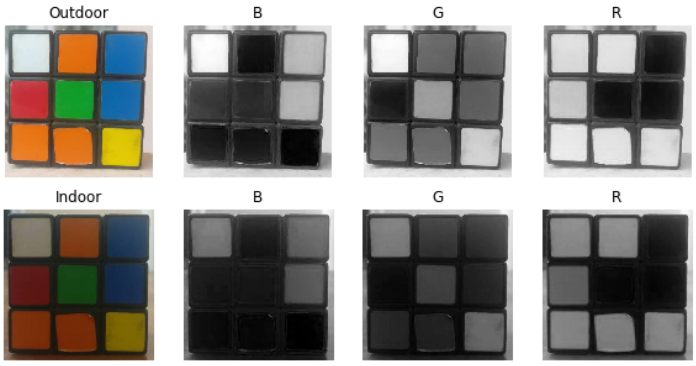


Figure 2.5: Blue (B), green (G) and red (R) channels of the RGB color space shown separately.

A great difference between indoor and outdoor lighting conditions can be observed from the results. This kind of non-uniformity makes color based segmentation very difficult in this color space. The overall problems associated with RGB are.

- Significant perceptual non-uniformity.
- Mixing of chrominance (color related information) and luminance (intensity related information) data.

Lab color space

The three components of this color space are lightness (L), color component ranging from green to magenta (a) and color component ranging from blue to yellow (b). The L channel is color independent, the other two are responsible for encoding color. It has the following properties.

- Perceptually uniform color space which approximates how we perceive color.
- Independent of device (capturing or displaying).
- Used extensively in Adobe Photoshop.
- Is related to the RGB color space by a complex transformation equation.

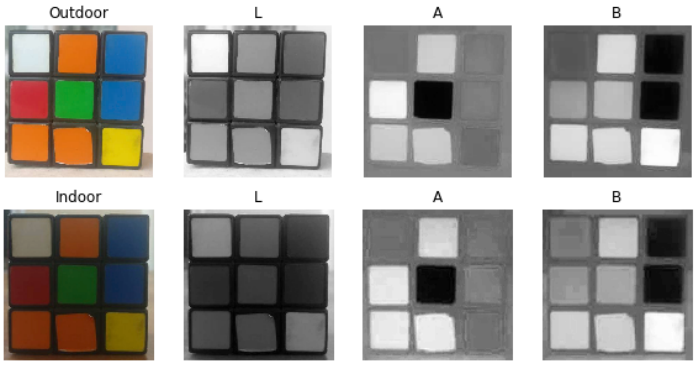


Figure 2.6: Lightness (B) and color components (a and b) of the Lab color space.

The fact that two channels encode color may be counterproductive for color segmentation. The main observations are as follows.

- Indeed the only channel that went under major changes was the L corresponding to illumination.

- The respective values of Green, Orange and Red (which are the extremes of the A Component) has not changed in the B component and similarly the respective values of Blue and Yellow (extremes of the B component) has not changed in the A component.

YCrCb color space

The three channels or components of this independent device color space are the following.

1. Y: Luminiscence or Luma
2. Cr: R - Y (how fat is the red component from Luma)
3. Cb: B - Y (how fat is the blue component from Luma)

It separates the luminance and chrominance components into different channels and is mostly used in compression for TV Transmission.

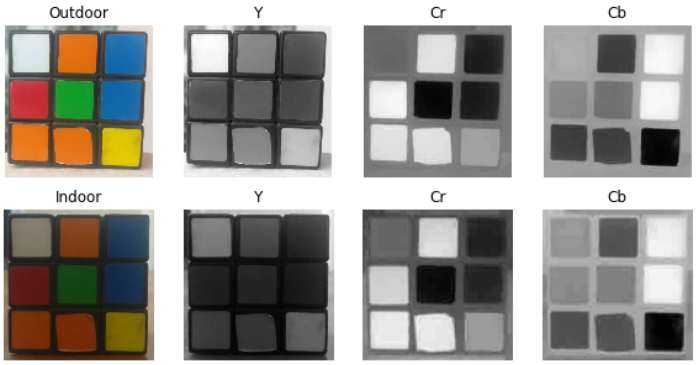


Figure 2.7: Luma (Y) and Chroma (Cr and Cb) components of the YCrCb color space.

Similar observations as Lab can be made for intensity and color components with regard to illumination changes.

- Perceptual difference between Red and Orange is less even in the outdoor image as compared to Lab.
- White has undergone change in all 3 components.

HSV color space

Hue (H), saturation (S) and value (V) are the three components of this color space.

- Highly convenient that it uses only one channel to describe color (H), making it very intuitive to specify color.
- Device dependent.

Observations

- The H component is very similar in both the images which indicates the color information is intact even under illumination changes.
- The S component is also very similar in both images.
- The V component changes due to the fact that it captures the amount of light.

There is drastic difference between the values of the red piece in the outdoor and indoor image. This is because Hue is represented as a circle and red is at the starting angle. So, it may take values between [300, 360] and again [0, 60].

2.3.2 Best color space

Illumination plays an important role when trying to achieve color segmentation. We've already seen that we don't have to worry about intensity when working with Lab, YCrCb or HSV color space. However, in HSV only the one component (H) contains information about the absolute color. Thus, it becomes our first

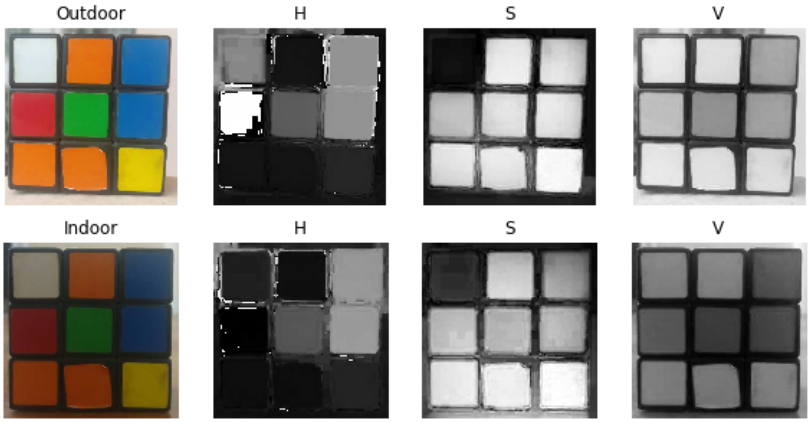


Figure 2.8: Blue (B), green (G) and red (R) channels of the RGB color space.

Figure 2.9: Red color is on the limits of the Hue scale.

choice of color space since there is only one knob to tune compared to 2 knobs in YCrCb (Cr and Cb) and Lab (a and b). Of course, the saturation (S) and (V) values can be tuned as well but one might want to maintain a wide range in order to cover different lighting conditions.

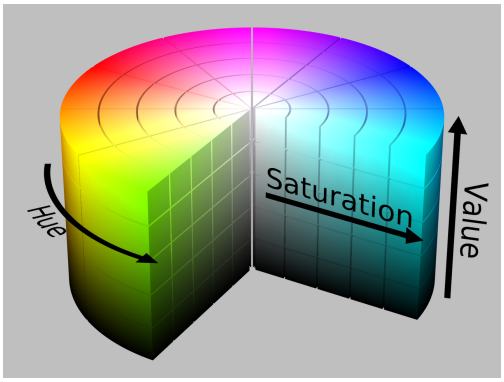


Figure 2.10: HSV cylinder.

2.3.3 Thresholding

Once selected our desired color space, we must segment the image using two thresholds and only the pixels whose value is within these thesholds should be visible. A methodical way for setting the minimum and maximum thresholds is using interactive scrolling bars while seeing the live result. OpenCV counts with `Trackbars` for implementing the commented methodology.

Further proof of the robustness against lighting situations can be depicted in the following images, where the same thresholds were used to filter the blue color in the two different lighting conditions presented before (indoor and outdoor). Figure 2.12 shows that HSV performs best, closely followed by Lab and YCrCb. RGB has remarkable bad performance in outdoor conditions.

A ROS node was implemented for this purpose, it subscribes to an image and outputs a corresponding window for tuning the HSV thresholds while displaying the resulting thresholded image.

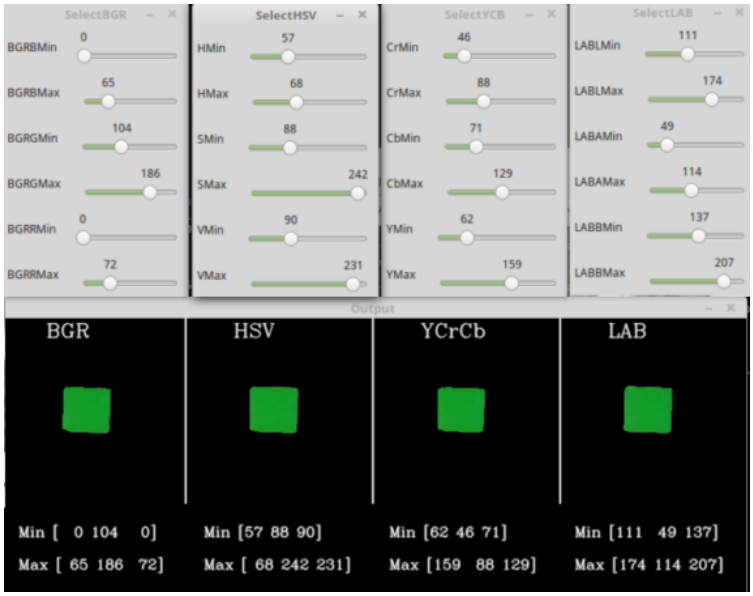
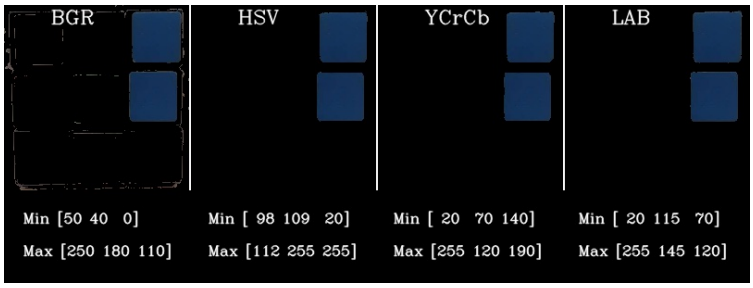
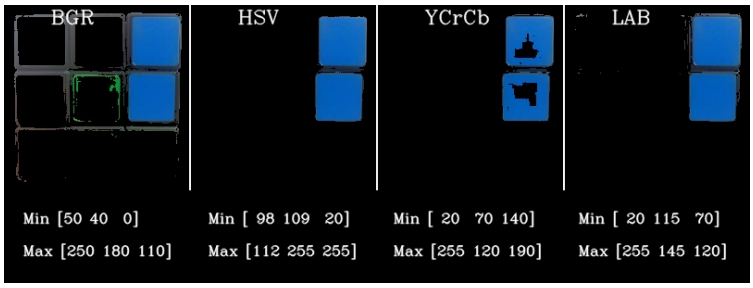


Figure 2.11: Interactive segmentation for different color spaces.



(a) Indoor lighting conditions.



(b) Outdoor lighting conditions.

Figure 2.12: Blue color segmentation results.

2.3.4 Implementation results

Using OpenCV, the color segmentation consists of two main steps.

1. Obtain binary mask of the pixels within the HSV thresholds' range.
2. Apply mask over the RGB image to display results.

This one segmentation turned out rather tricky. Recall from figure 2.9 that red is on the limits of the hue scale, so thresholding for orange would involve a more tedious process. A workaround solution for this problem is to invert the RGB image, convert the inverted image to HSV and then filter for cyan.like color (which would be the opposite of orange). Of course this would also affect the saturation and value components which will be inverted as well.

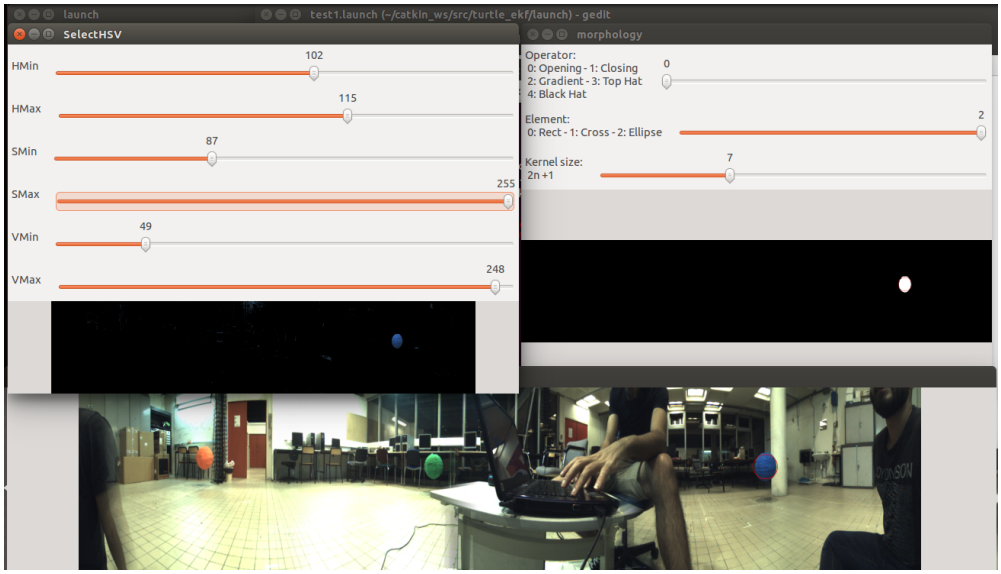


Figure 2.13: Blue color segmentation.

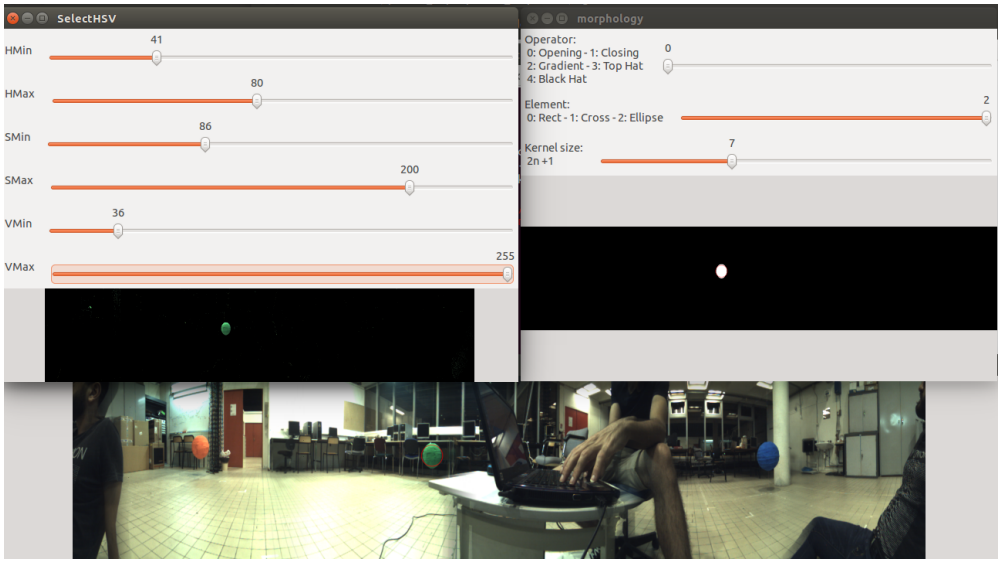


Figure 2.14: Green color segmentation.

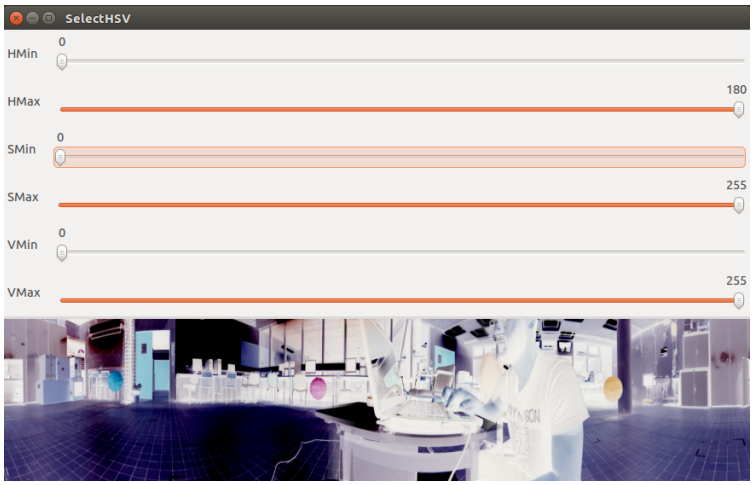


Figure 2.15: Inverted image for segmenting redish colors.

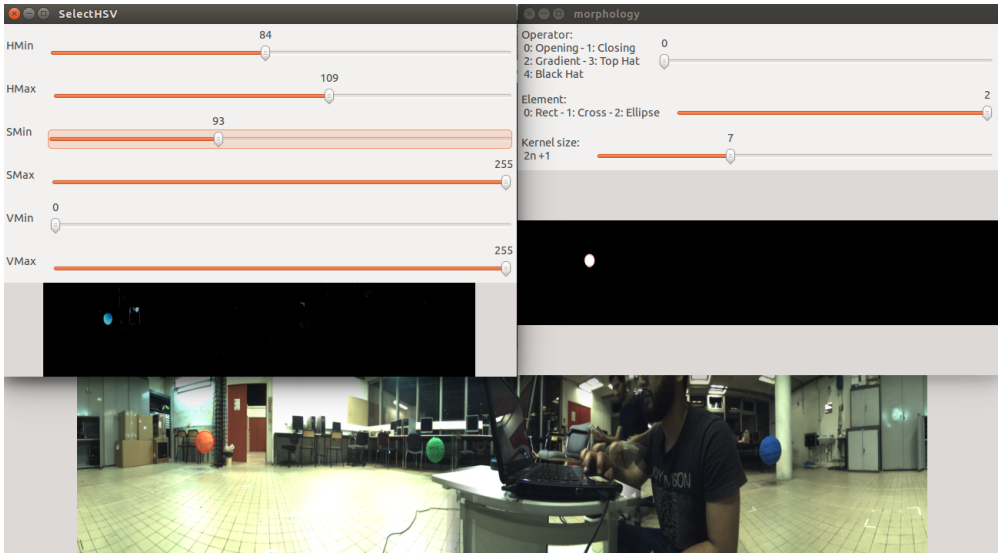


Figure 2.16: Orange (Cyan) segmentation.

2.4 Blob detection

Having segmented the image for filtering the three beacons we end up with three different binary masks. Next step is to search for blobs in these binary masks. What is a **blob**? In terms of computer vision, a blob is an image that share common value (e.g. grayscale value). Figure 2.17 shows different types of blobs corresponding to dark connected regions.

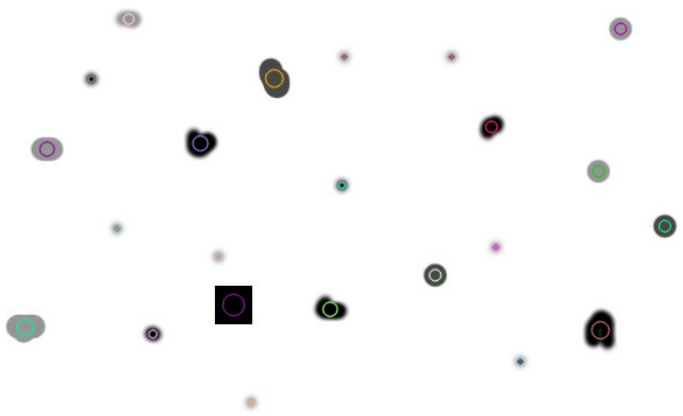


Figure 2.17: Dark-blob detection.

OpenCV provides a convenient way to detect blobs and filter them based on different characteristics. Its library includes a `SimpleBlobDetector` which is based on a rather simple algorithm described below [8].

1. **Thresholding**: Convert the source images to **several** binary images by thresholding the source image with thresholds starting at a minimum and increasing towards a maximum one.
2. **Grouping**: In each binary image, connected white pixels are grouped together. We'll call these binary blobs.
3. **Merging**: The centers of the binary blobs in the binary images are computed, and blobs located closer than a certain distance are merged.
4. **Center and radius calculation**: The centers and radii of the new merged blobs are computed and returned.

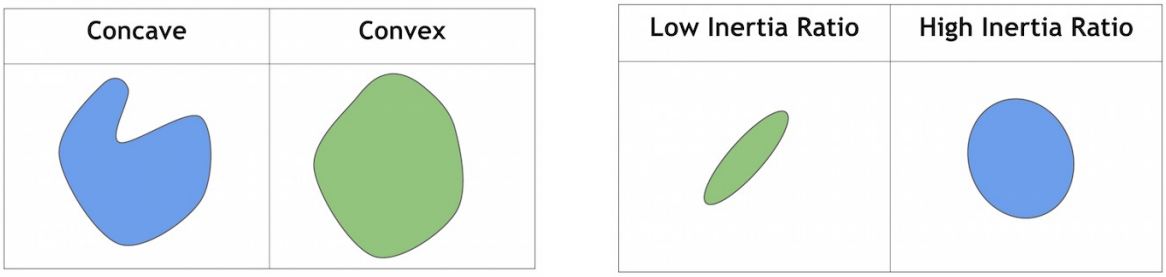
Note: since our input image to this algorithm is already binary one, there is no need for any more thresholding. Nonetheless, the algorithm groups binary blobs obtained at different thresholded images, so we must at least guarantee two binary images on the thresholding step.

Furthermore, the blob detector has several filters.

- **By color**: Set filterByArea = true and give an appropiate value to blobColor. Miss-leading name since it filters either dark (0) or white (255) blobs.
- **By size**: One can filter the blobs based on size by setting the parameters `filterByArea = true`, and appropriate values for `minArea` and `maxArea`. The area is given in pixels.
- **By shape**: shape has three different parameters.
  - **Circularity**: This just measures how close to a circle the blob is. E.g. a regular hexagon has higher circularity than say a square. Set `filterByCircularity = true`. Then set $0 \leq$ `minCircularity` $<$ `maxCircularity` ≤ 1. Circularity is defined as

$$\frac{4\pi Area}{perimeter^2}$$

    .
  - **Convexity**: it is defined as the ratio between the area of the blob and the area of it's convex hull. Convex hull of a shape is the tightest convex shape that completely encloses the shape. To filter by convexity, set `filterByConvexity = true`, followed by setting $0 \leq$ `minConvexity` $<$ and `maxConvexity` ≤ 1.
  - **Inertia ratio**: This measures how elongated a shape is. E.g. for a circle, this value is 1, for an ellipse it is between 0 and 1, and for a line it is 0. To filter by inertia ratio, set `filterByInertia = true`, and set $0 \leq$ minInertiaRatio $<$ `maxInertiaRatio` ≤ 1 appropriately.



(a) Blobs with different convexity.

(b) Inertia of blobs.

Figure 2.18: Filtering by shape.

For our implementation we must set the filtering by color to `blobColor = 255` since the white pixels on the mask (obtained from color segmentation) represent the blobs we want to detect. Secondly, we would like to detect in a wide range of distance, that is, they might have a great area because they are close to the camera or small area when far away. So, the min and maximum of these parameter has a wide range. Lastly, the shape of our blobs is pretty much circular, so we define the parameters accordingly. Final tuning can be depicted in the following strip of code.

Code 2.1: `simpleBlobDetector` parameters

```
// Setup SimpleBlobDetector parameters.
SimpleBlobDetector::Params params;

// Filter by Color
params.filterByColor = true;
params.blobColor = 255;

// Change thresholds
params.thresholdStep = 80;
params.minThreshold = 20;
params.maxThreshold = 120;

// Filter by Area.
params.filterByArea = true;
params.minArea = 300;
params.maxArea = 160000;

// Filter by Circularity
params.filterByCircularity = true;
params.minCircularity = 0.7;

// Filter by Convexity
```

```
params.filterByConvexity = true;
params.minConvexity = 0.8;

// Filter by Inertia
params.filterByInertia = true;
params.minInertiaRatio = 0.5;
```

These algorithm turns out to be fairly robust as long as the color segmentation is carried out successfully. The valuable information obtained from this is the blob's **radius** which will be further used.

2.5 Distance calculation

We're at the last step for measuring the distance between each beacon and the camera. The method used is the **triangle similarity** for object to camera distance. The triangle similarity is often used when having objects of the same shape but not necessarily the same size, which is exactly our case. Adapting it to our scenario, we want to measure the blob's diameter. The relation between a blob with diameter/width W at a distance D from the camera and with P apparent diameter (measured normally in pixels) is given by the following equation [9].

$$f = \frac{P \times D}{W} \tag{2.4}$$

where

- f is the focal length

Normally one would have to compute the focal length experimentally but we've already got that information from the calibration process. From the previous formula, the distance from the object to the camera is computed as follows.

$$D = \frac{f \times W}{P} \tag{2.5}$$

The apparent diameter P is obtained from the `SimpleBlobDetector` algorithm, therefore we have all the elements necessary to compute the distance from the camera to the beacon. One detail to take into consideration is that each camera has slightly different focal lengths, therefore we must be able to identify which camera is *seeing* the beacon. This is an easy task since we get the center coordinates (in pixels) of the detected blob.

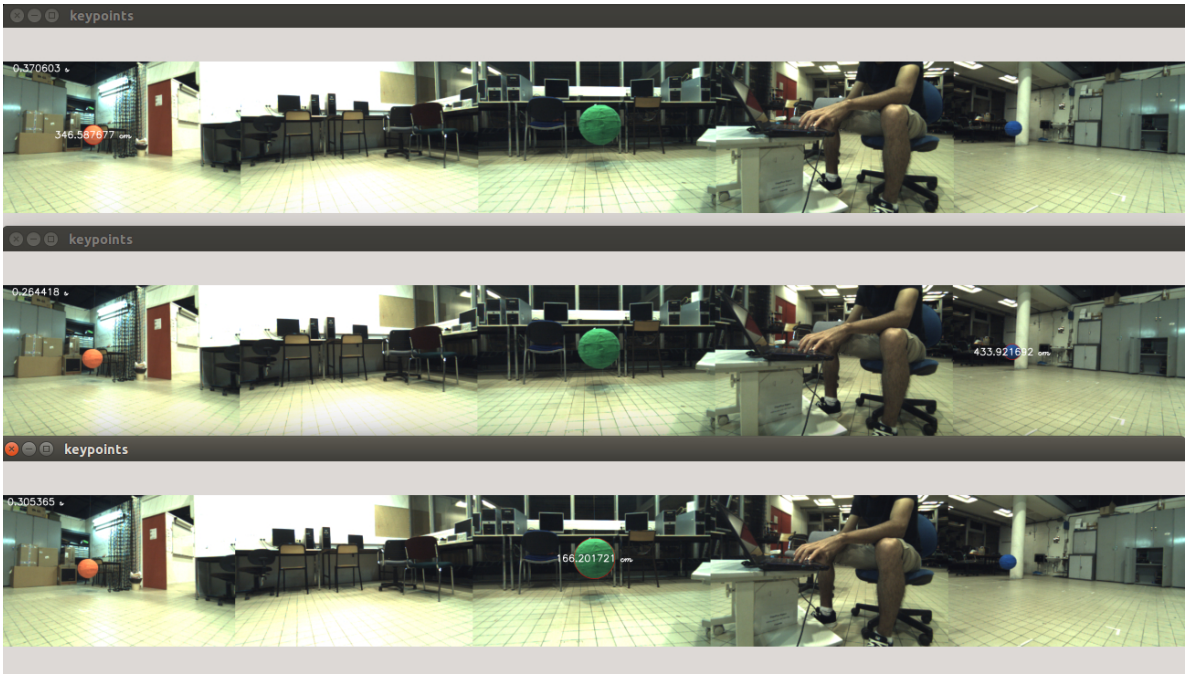


Figure 2.19: Measuring distances to the 3 beacons, orange (upper), blue (middle), green (lower).

2.6 Time performance

Computer vision is commonly computationally expensive and requires a lot of power. In most cases, we require to execute this algorithms in **realtime** which imples that a single frame must be processed within a certain amount of time.

Publishing images from one laptop and accessing them on another one is slow because of the great amount of data to be transported. Therefore, we opted for carrying out all the image processing on the computer connected to the camera.

Since we're processing a considerably large image, we must measure the time it takes for a frame to be processed. By using the `ros::Time::now()` clock we were able to measure the duration of the whole frame processing. Although it varies from frame to frame, the maximum elapsed time per frame is 0.5 seconds and the minimum is 0.2 seconds. Proof of this can be found on the images of figure 2.19, the number located on the top-left corner shows the amount of time (in seconds) spent on processing the visualized frame.

Which means that we can process at a **2 fps** frequency, rather slow! But we must also keep in mind that this is not the only processed carried out by the computer, rectification and image tiling are also running on the same computer along with other ROS processes, making it hard for the computer to process in real time.

2.7 ROS

Four main nodes were developed to fulfill the following functions (one node per function).

- Rectify image.
- Tile images together.
- Blob detection and measurement.
  - Convert from RGB to HSV.
  - Perform HSV thresholding.
  - Run SimpleBlobDetector with HSV binary mask as input.
  - Measure distance with corresponding camera focal length.
- Arrange beacon distances in a consistent format (interface).

The last node hasn't been commented on the previous sections but it is highly important since these distances will be later used for localization and they must have a consistent format regardless of the order in which the distance was published. This can be easily achieved with an interface node. It subscribes to all the three computed distances and stores them in an array where the first element corresponds to the blue beacon distance, the second one to the orange and the third to the green beacon.

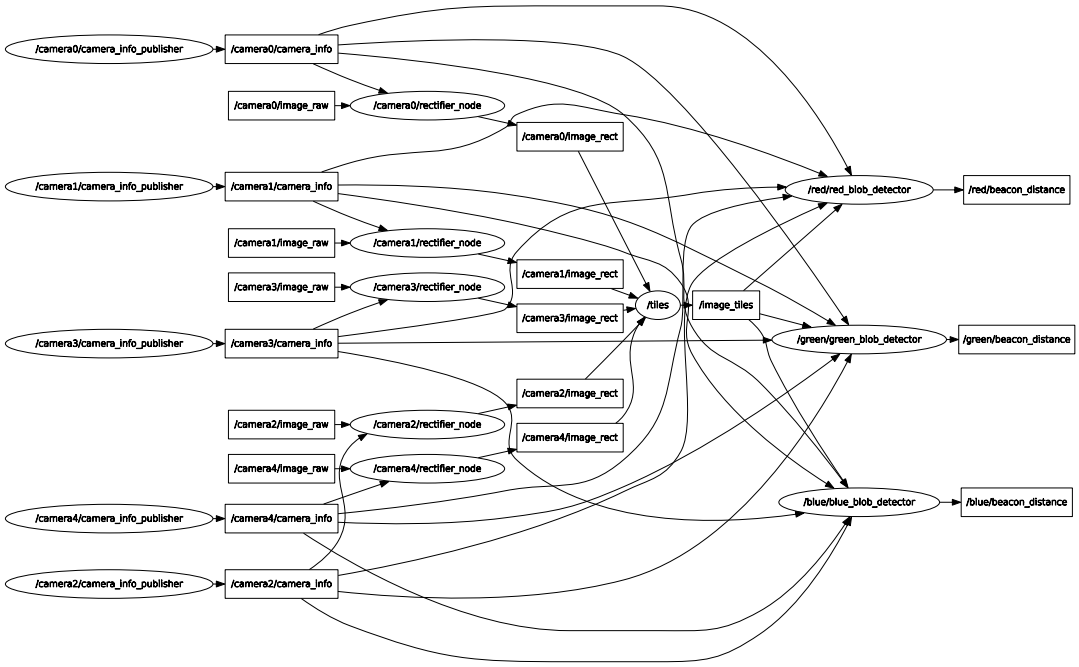


Figure 2.20: Rqt graph

Chapter 3

Absolute localization

As an early approach to absolute localization, one can use the three previously extracted distances to localize the robot because the beacons' positions are known.

3.1 Trilateration

In geometry, trilateration is the process of finding the absolute or relative locations of points by measurements of distances. In contrast to *traingulation*, it does not involve the measurement of angles [10].

Here, a simple derivation using three spheres is presented. However, one must design the scenario such that:

1. All spheres must intersect.
2. All three centers are in the plane $z = 0$.
3. The sphere center P_1 is at the origin.
4. The sphere center P_2 is on the x-axis.

Our beacons configuration satisfies the previous conditions. P_1 corresponds to the center of the blue beacon, P_2 to the red one, and P_3 to the green one.

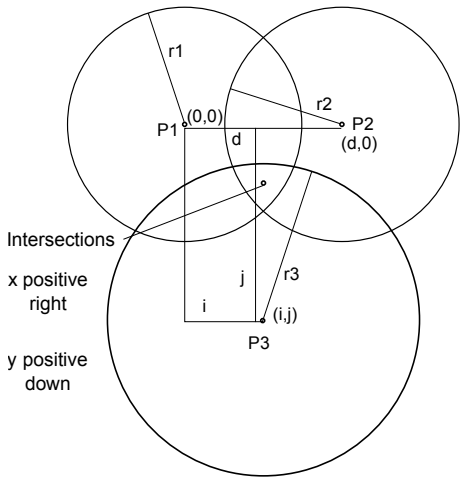


Figure 3.1: Three spheres intersecting at plane $z = 0$.

We start off with the equations for the three spheres.

$$r_1^2 = x^2 + y^2 + z^2$$
$$r_2^2 = (x - d)^2 + y^2 + z^2$$
$$r_3^2 = (x - 1)^2 + (y - j)^2 + z^2$$

We need to find a point (x, y, z) that satisfies all the three equations. We first use r_1 and r_2 to eliminate y and z from the equation and solve for x.

$$\begin{aligned} r_1^2 - r_2^2 &= x^2 - (x-d)^2 \\ r_1^2 - r_2^2 &= x^2 - x^2 - 2xd + d^2 \\ r_1^2 - r_2^2 + d^2 &= 2xd \\ x &= \frac{r_1^2 - r_2^2 + d^2}{2d} \end{aligned} \tag{3.1}$$

For y, we proceed to substitute $z^2 = r_1^2 - x^2 - y^2$ into the formula for the third sphere and solve accordingly.

$$y = \frac{r_1^2 - r_3^2 - x^2 + (x-i)^2 + j^2}{2j} = \frac{r_1^2 - r_3^2 + i^2 + j^2}{2j} - \frac{i}{j}x \tag{3.2}$$

Having x and y we can find the z-coordinate, however in our case it is not necessary since we are dealing with a 2D localization space.

3.2 Optimization techniques

A more elaborated method can be used for determining the absolute position of the robot based on the distances. Optimization techniques can be used for finding the minimum of a function while respecting several constraints.

Similar to the `fmincon` function in MATLAB, we can use the RobOptim library for C++, which contains numerical optimization applied to robotics [11]. Although it is quite hard to get it installed and running, one its done the optimization process can be defined in simple steps.

For our case, we would like to minimize the distances from the robot to the beacons while maintaining as minimum the distance computed using the camera. Contrary to the previous case, the algorithm would work best if the three circles don't intersect between each others. Mathematically speaking, the objective and constrain functions are as follow.

Objective

$$\min \sum_{i=1}^{3} (x_i - x)^2 + (y_i - y)^2 \tag{3.3}$$

where

- (x, y) are the 2D coordinates of the robot.
- (x_1, y_1) are the 2D coordinates of the i-th beacon.

Constraints

$$g_i = (x_i - x)^2 + (y_i - y)^2 \geq d_i^2 \tag{3.4}$$

where

- $i = 1, 2, 3$
- d_i is the distance from the camera to the i-th beacon.

Due to the noise and uncertainty of the distance measurements obtained, it was hard to reliably compute a solution with this method

Chapter 4

Hybrid localization with EKF

4.1 EKF introduction

The Kalman filter is widely used in localization systems. Generally speaking, the Kalman filter is an algorithm that uses a series of measurements observed over time, containing noise and inaccuracies, and produces estimates of unknown variables that tend to be more accurate than those base on single measurements.

Lets take a simple example to explain how Kalman Filter works, if we consider that our robot are forced to move in one direction -supposing along x-axis. Considering the available data are the velocity of the robot -velocity is measured by encoder-, and the distance between the robot and one marker fixed at the origin which represent the x-coordinate of the robot -distance is measured by camera-. also we know the initial position of the robot with certain accuracy. the velocity can be used to predict the robot position using the simple relation "the distance is the integration of the velocity". All the available data are represented by normal distribution density function. so we can describe the initial position by the expected value "mean" and the standard deviation or the variance to represent how the data are spread around the mean. the same rule for the predicted position and the measured one.

Figure 4.1a shows the initial position of the robot. After time equal T, figures 4.1b, and 4.1c show the predicted and the measured position of the robot at one time instant.

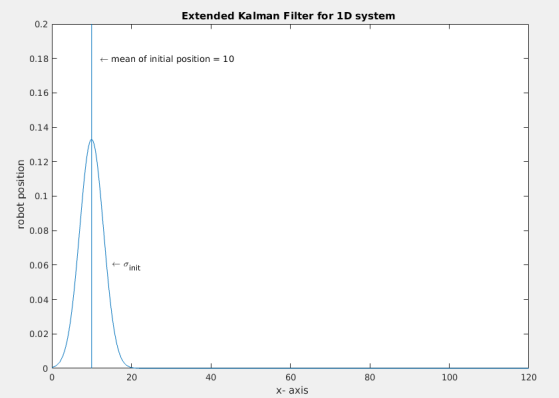
Now lets ask where should be the optimal estimated position?, the logical guess should depend on the accuarcy of the predicted and the measured position. considering the estimated the position to be located inbetween the predicted and the measured positions. This estimated position is represented by the green curve at figure 4.1d.

There are three distinct cases to describe the situation:

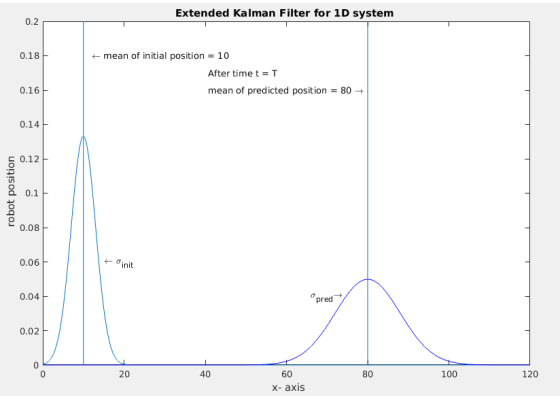
- The case when the predicition using the odometry is instabl and inaccurate, and our measurenent is 100 % accurate we can omit the odometry and take only into account the measurement position to be the estimated position, at this situation we can say that we trust our measurement rather than the odometry.
- The case when the predicition using the odometry is stabl and all the parameter used in odometry are 100% accurate, and our measurenent is inaccurate we can omit the measurement and take only into account the prediciton position to be the estimated position, at this situation we can say that we trust our measurement rather than the odometry.
- the case at which both of the measurement and the odometry has errors, we can use the measurement to modify the odometry in such a way that the estimated position will be located in-between the predicted and the measured one.

The third situation is the general one, a Kalman Gain K is defined to locate the estimated position of the robot between the predicted and the measured one. this gain depends on the accuracy of both of them. Kalman gain has a value in range between zero and one (for 1D system), if its zero so there will be no modification on the predicted state.

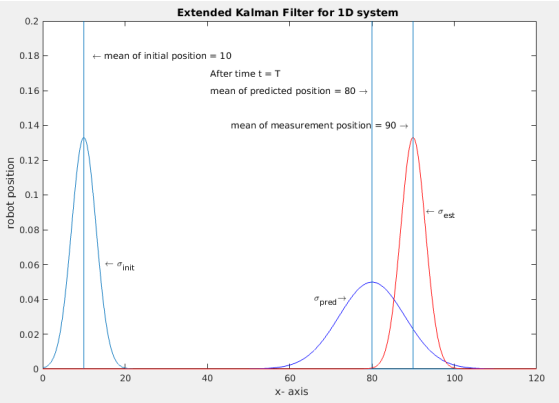
Lets take a similar example 2D system, now we consider that the robot moves in 2D plan, and we need to determine its pose x, and y coordinates. we still have the robot velocity for the odometry and we have also



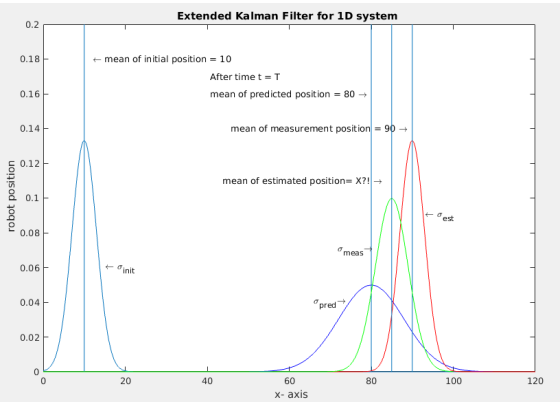
(a) initial position



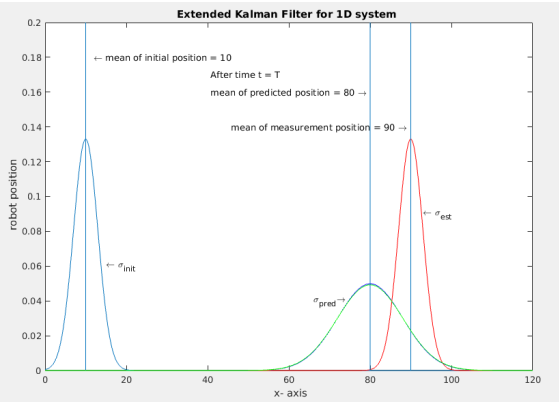
(b) predicted position 'blue'



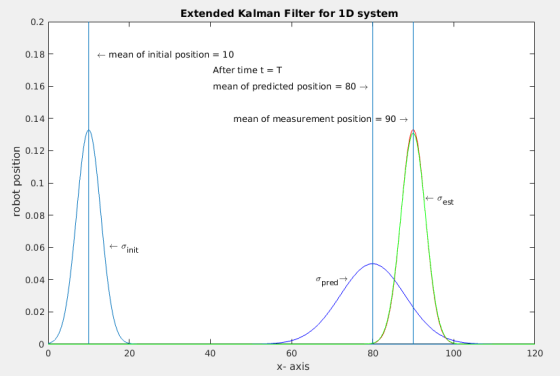
(c) measured position 'red'



(d) the estimated position of the robot 'green'



(e) stable prediction and inaccurate measurement case



(f) instable prediction and accurate measurement case

Figure 4.1: EKF 1D system

the measurement as the coordinates (x,y) of the robot.

Figure 4.2 shows that we can use the same strategy that we have used in the previous 1D problem. considering that all the equation will be used in this case will be in matrix form.

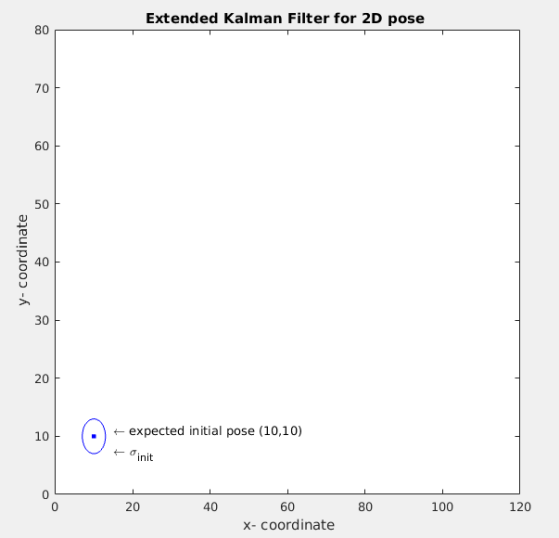
General EKF equations

State equations:

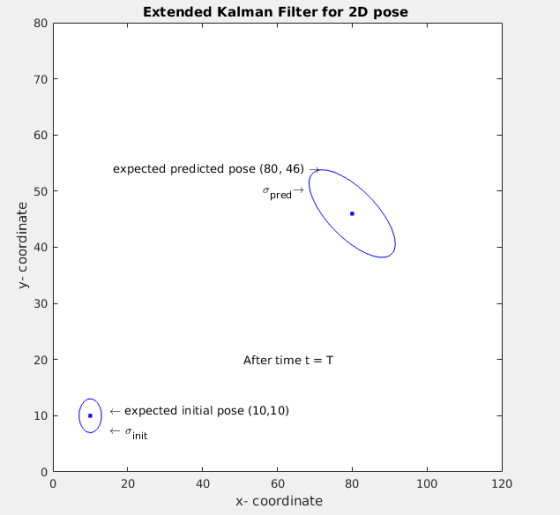
$$X_{k+1} = f\left(X_k, U_k^*\right) + \alpha_k \tag{4.1}$$

$$U_k^* = U_k + \beta_k \tag{4.2}$$

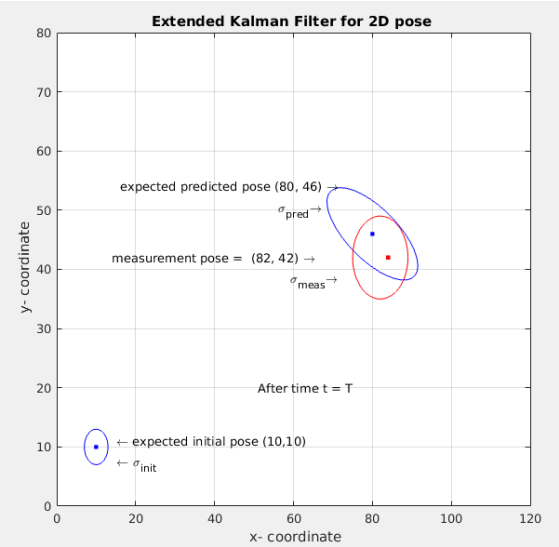
$$Y_k = g_{mag}\left(X_k\right) + \gamma_k \tag{4.3}$$



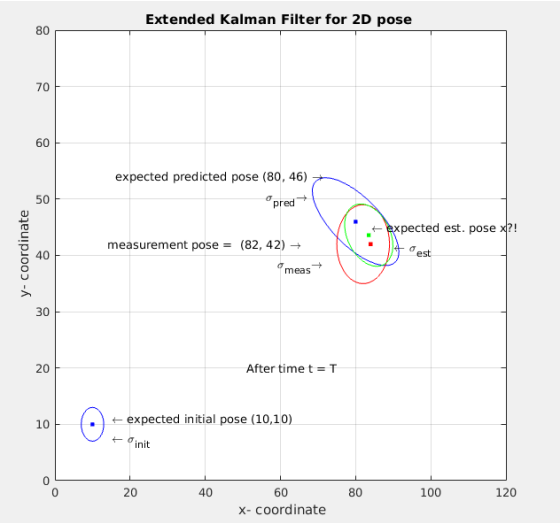
(a) initial pose



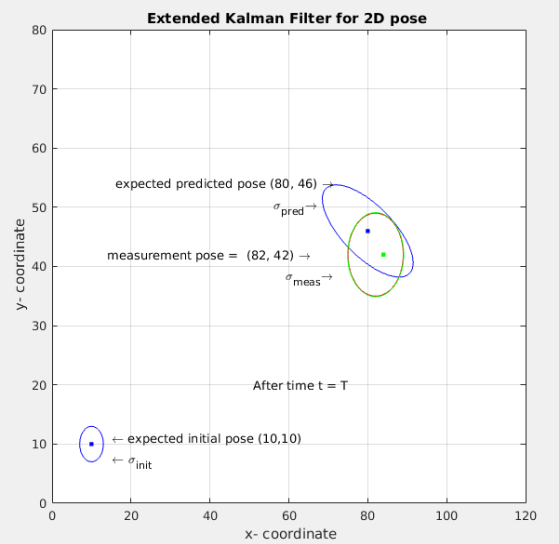
(b) predicted pose 'blue'



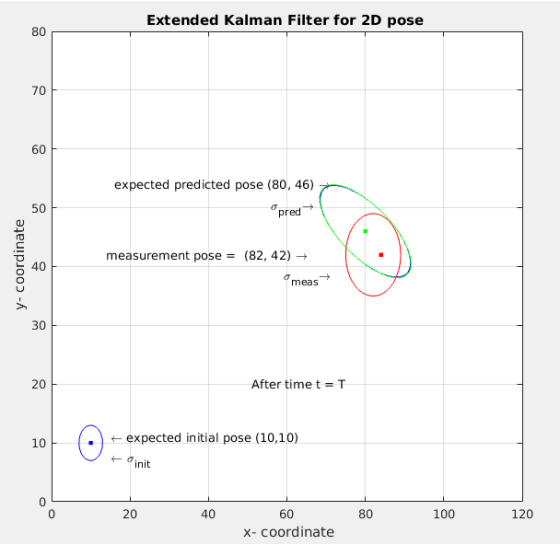
(c) measured pose 'red'



(d) the estimated pose of the robot 'green'



(e) stable prediction and inaccurate measurement case



(f) instable prediction and accurate measurement case

Figure 4.2: EKF 2D system

Prediction phase:

$$\hat{X}_{k+1|k} = f\left(\hat{X}_{k|k}, {U^*}_k\right) + \alpha_k \tag{4.4}$$

$$\hat{P}_{k+1|k} = A_k\, P_{k|k}\, {A_k}^T + B_k\, Q_\beta\, {B_k}^T + Q_\alpha \tag{4.5}$$

with A_k and B_k defined as:

$$\begin{cases} A_k = \frac{\partial f}{\partial X}\left(\hat{X}_{k|k}, {U^*}_k\right) \\ B_k = \frac{\partial f}{\partial U}\left(\hat{X}_{k|k}, {U^*}_k\right) \end{cases}$$

Estimation phase:

$$\hat{X}_{k+1|k+1} = f\left(\hat{X}_{k+1|k}, {U^*}_k\right) + K_k\left(Y_k - \hat{Y}_k\right) \tag{4.6}$$

$$\hat{P}_{k+1|k+1} = (1 - K_k\, C_k)\, P_{k+1|k} \tag{4.7}$$

with

$$\begin{cases} \hat{Y}_k = g_{mag}\left(\hat{X}_{k+1|k}, {U^*}_k\right) \\ C_k = \frac{\partial g}{\partial X}\left(\hat{X}_{k|k}, {U^*}_k\right) \\ K_k = P_{k+1|k}\, {C_k}^T \left(C_k\, P_{k+1|k}\, {C_k}^T + Q_\gamma\right)^{-1} \end{cases}$$

Definitions

- U^*_k is the noisy measurement of input U.
- β_k is the additive noise in measured input.
- Q_β is the covariance matrix of input U.
- X_k is the state which contains the posture of the robot (x, y, θ) at time k.
- $X_{k+1|k}$ is the predicted state, an estimation of X at time t_{k+1} using all information available until instant t_k.
- $X_{k+1|k+1}$ is the estimated state, an estimation of X at time t_{k+1} using all information available until instant t_{k+1}.
- α_k is the additive noise in measured input.
- Q_α is the covariance matrix of the state equation.
- Y_k is the measurement at t_k.
- $\hat{Y}_k$ is the expected measurement at t_k.
- Q_γ is the covariance matrix of Y.
- $P_{k+1|k}$ is the predicted covariance matrix of the predicted state.
- $P_{k+1|k+1}$ is the estimated covariance matrix in the estimated state.

4.2 Odometry

The general definition of odometry is the use of data from motion sensors to estimate change in position over time. Odometry is used by the TurtleBot to estimate its position and orientation. TurtleBot uses encoders to estimate its position relative to a starting location given in terms of coordinate (x, y) position and it uses Gyroscope to estimate its orientation.

The theory behind odometry is that by using the kinematic model of the robot -turtlebot is (2,0) robot- we can express the translational and rotational speed of the robot as functions of the rotation speed of some of the robot's wheels. These equations are then put into discrete form to yield relations between elementary translations and rotations of the robot and elementary rotations of the wheels.

kinematic model of a mobile robot of type (2,0)://

$$\begin{cases} v = \left(r_r \dot{q_r} + r_l \dot{q_l}\right)/2 \\ \omega = \left(r_r \dot{q_r} + r_l \dot{q_l}\right)/e \\ \dot{x} = v\cos(\theta) \\ \dot{y} = v\sin(\theta) \\ \dot{\theta} = \omega \end{cases}$$

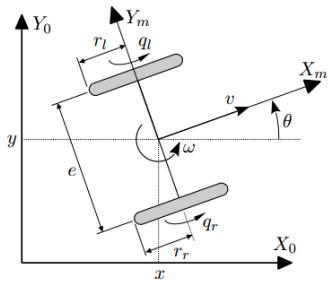


Figure 4.3: kinematic model for turtlebot "2,0 robot"

where

- v is the linear of the robot.
- ω is the angular velocity.
- q_r is the rotation speed of the right wheel.
- q_l is the rotation speed of the left wheel.
- r_r is the radius the right wheel.
- r_l is the radius of the left wheel.
- $\dot{x}$ is the linear velocity along x axis.
- $\dot{y}$ is the linear velocity along y axis.
- $\dot{\theta}$ is the angular velocity of the robot around z axis.

Discrete kinematic model:

$$X_{k+1} = \begin{bmatrix} x_{k+1} \\ y_{k+1} \\ \theta_{k+1} \end{bmatrix} = \begin{bmatrix} x_k + \Delta D_k cos(\theta_k) \\ y_k + \Delta D_k sin(\theta_k) \\ \theta_k + \theta_k \end{bmatrix} \tag{4.8}$$

where $U_k = [\Delta D_k, \theta_k]$

ROS topic "/odom":
TurtleBot publishes its pose and its twist through a topic called "/odom" using nav_msgs/Odometry message .

nav_msgs/Odometry Message

File: nav_msgs/Odometry.msg

Raw Message Definition

```
# This represents an estimate of a position and velocity in free space.
# The pose in this message should be specified in the coordinate frame given by header.frame_id.
# The twist in this message should be specified in the coordinate frame given by the child_frame_id
Header header
string child_frame_id
geometry_msgs/PoseWithCovariance pose
geometry_msgs/TwistWithCovariance twist
```

Figure 4.4: nav_msgs/Odometry message

figure 4.4 shows the content of message of type nav_msgs/Odometry, it contains four fields:

- header of type std_msgs/Header: contains sequence number, time, and frame_id which is called odom frame.
- child_frame_id of type string: represents the frame at which the position and orientation are expressed.
- pose of type geometry_msgs/PoseWithCovariance: contains the position of the robot using 3D coordinates (x, y, z) and the orientation in term of quaternion (x, y, z, w) besides their covariance matrix of size $6x6$.
- twist of type geometry_msgs/TwistWithCovariance: contains the linear velocities along x, y, and z axis and angular velocities around x, y, and z axis besides their covariance matrix of size $6x6$.

in our case -turtlebot-, we do not need to measure the geometrical parameters (wheels radius, gauge) nor using the kinematic model because the odometry is processed inside the robot. we just subscribe to the /odom topic at which the robot publish its 3D pose and twist.

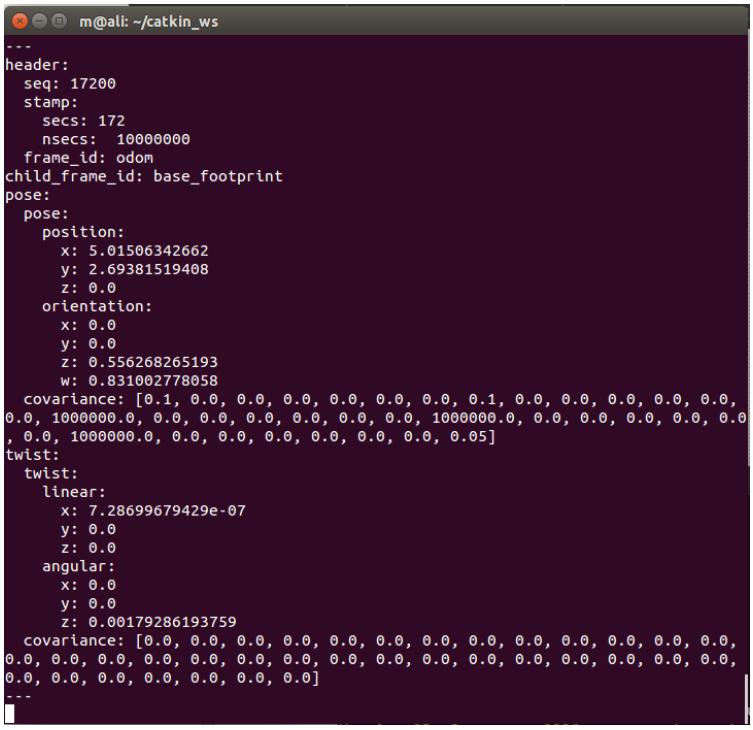


Figure 4.5: ROS message of type nav_msgs/Odometry

4.2.1 Odometry error and uncertainty

why we do not trust odometry:

Odometry is sensitive to errors due to the integration of velocity measurements over time to give position estimates.

- the different types of error that can occur during odometry are:

Systematic errors: Systematic errors are very serious because their effects accumulate over time, we can consider them the dominant errors on smooth indoor surfaces:

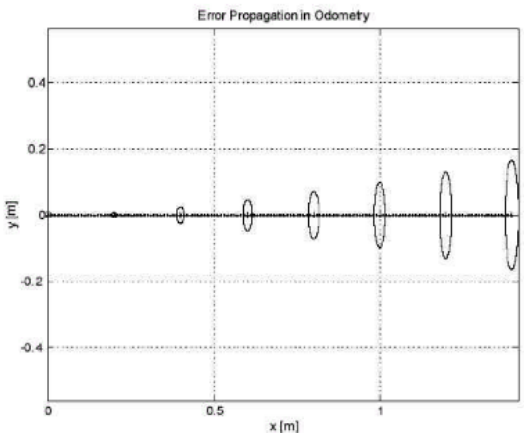
- Wheel radius and track gauge errors.
- Misalignment of wheels.
- Non-point wheel-to-floor contact.
- Finite resolution and sampling rate.

Systematic errors deterministic, so they can be eliminated through proper calibration.

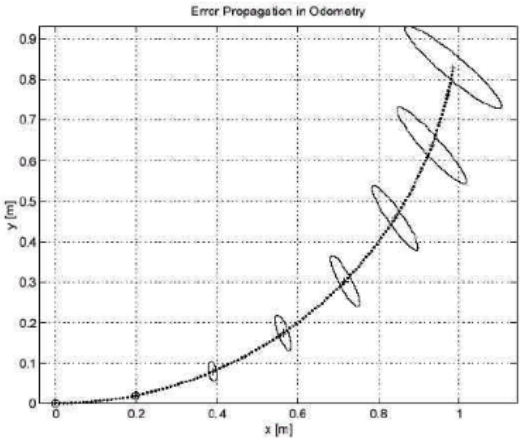
Non systematic errors:
Non systematic errors are difficult to handle because they are unexpected, so they can not be compensated. we can consider them the dominate errors on rough, and irregular terrains.

- Slipping due to Slippery floor.
- Skidding due to Over-acceleration and fast turning.

Odometry error propagation the odomtry data that the robot are publishing have a certain precision. the covariance matrix that associated with the odometry data represents its uncertainty.
Error propagation in case of odometry depends on the direction of motion:
figure 4.6 shows that error propagation perpendicular to the direction of motion grows much larger than the error propagation at the same direction of the motion.



(a) motion along x axis



(b) motion along arbitrary direction

Figure 4.6: odometry error propagation

Figure 4.7: Odometry interface node

4.2.2 Odometry Interface

As we mention above that the nav_msgs/Odometry message contains the 3D pose, orientation, twist and their covariances. our objective is to localize the robot on a flat plan, so we just need 2D Pose to express the x,y coordinates and the orientation theta of the robot plus their covariance matrix. although there is alresdy exist a message of type geometry_msgs/Pose2D that contain three fields for x, y, and theta, it does not contain any field for the covariance matrix.
Extended kalman filter will need also the covariance matrix of the 2D pose. So we have defined new message "turtle_ekf/Pose2DWithCovariance" to be used associated with Extended Kalman filter node, it contains the data we need from odometry "x, y, theta, plus the covariance matrix $3x3$"

- turtle_ekf/Pose2DWithCovariance message: contains
  pose2D (x, y, theta), covariance[9]
- nav_msgs/Odometry message: contains
  pose(x,y,z), qauternion(x,y,z,w), covariance [36]),
  twist (linear_velocity(x,y,z,), angular_velocit(x,y,z,), covariance [36]) .

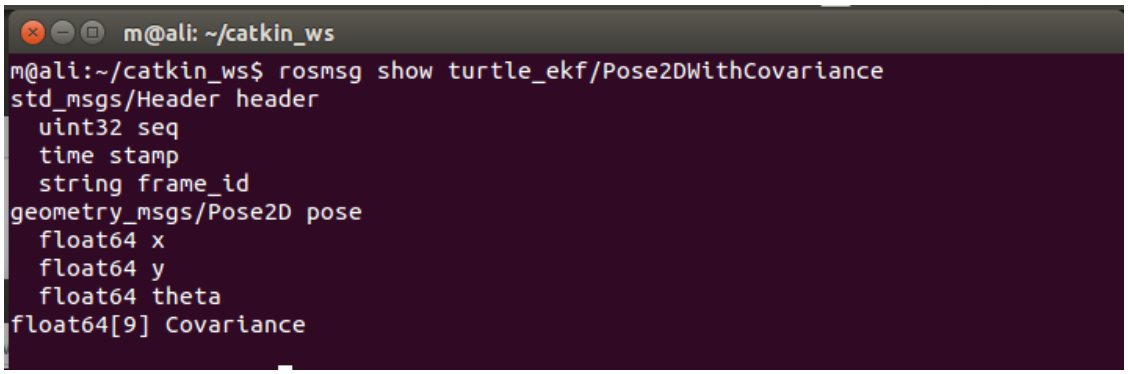


Figure 4.8: turtle_ekf/Pose2DWithCovariance message

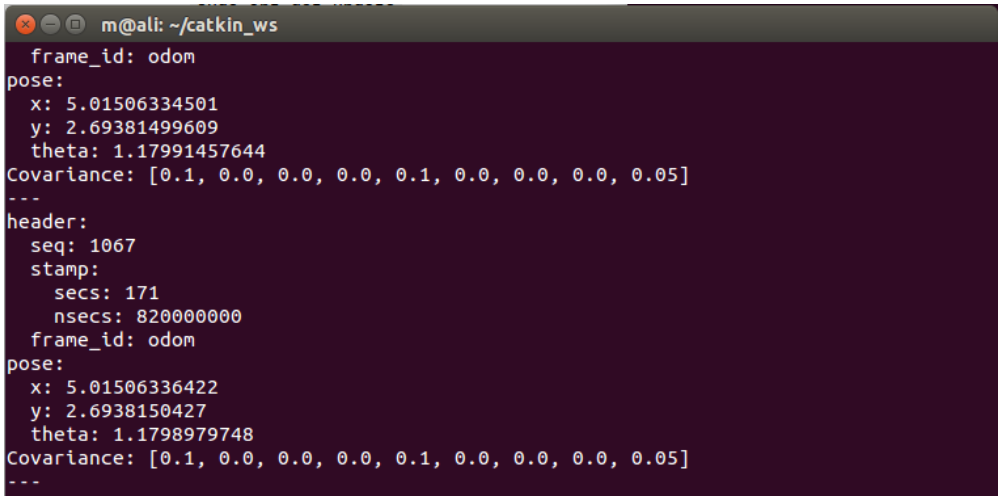


Figure 4.9: /odom2D topic data

These values in figure 4.9 are corsponding to the values shown in figure 4.5, both represent same pose $x = 5.0150, y = 2.6938$ and the same orienataion. but the nav_msgs/Odometry message uses quaternion to rpresent the orienatation. in the other hand, turtle_ekf/Pose2DWithCovariance message represents the orientation in term of rotation angle around z axis -yaw angle- theta= 1.1798 radian.
Odometry interface or can be called odometry adapter is a node to convert the information from one data type to another one, here we convert the nav_msgs/Odometry message to turtle_ekf/Pose2DWithCovariance message which contains the relevant data that we need from odometry.

Quaternion to Roll, Pitch , and Yaw angles "RPY" conversion:

Code 4.1: Quaternion to Roll, Pitch , and Yaw

```
/********************************* Steps: ********************************************
* subscribe to /odom topic and save the message in variable odom3D of type "nav_msgs/Odometry"
* define a message "odom2D" of type "turtle\_ekf/Pose2DWithCovariance" to be published
* define Quaternion of type tf transform using the received data
* form a rotation matrix m using the tf:quaternion
* define temporal variables roll, pitch, and yaw
* get the roll, pitch, and yaw angles from the rotation matrix m using m.getRPY method
* yaw is the rotation around z axis
* assign x,y,theta, and conariances values for the message that will be published
*************************************************************************************/

   tf::Quaternion q(odom_3D.pose.pose.orientation.x,
                    odom_3D.pose.pose.orientation.y,
                    odom_3D.pose.pose.orientation.z,
                    odom_3D.pose.pose.orientation.w);

       tf::Matrix3x3 m(q);
       double roll, pitch, yaw;
       m.getRPY(roll, pitch, yaw);
       odom_2D.pose.theta=yaw;

       odom_2D.pose.x=odom_3D.pose.pose.position.x;
       odom_2D.pose.y=odom_3D.pose.pose.position.y;
       odom_2D.pose.theta=odom_3D.pose.pose.position.theta;

       odom_2D.Covariance[0]= odom_3D.pose.covariance[0];
       odom_2D.Covariance[4]= odom_3D.pose.covariance[7];
       odom_2D.Covariance[8]= odom_3D.pose.covariance[35];
```

4.3 EKF Algorithm

- **Initialization step:**
  The first step is to initialize the filter based on our knowledge of the robot position. We assume that

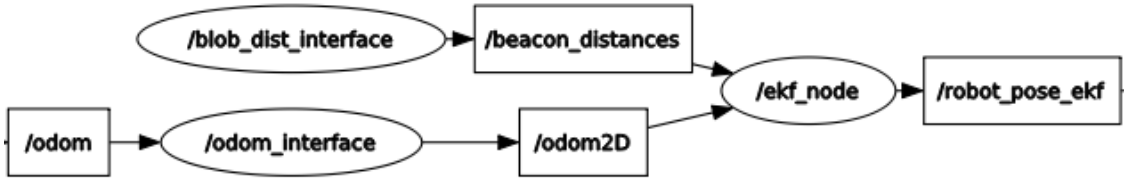


Figure 4.10: rqt_graph "EKF node"

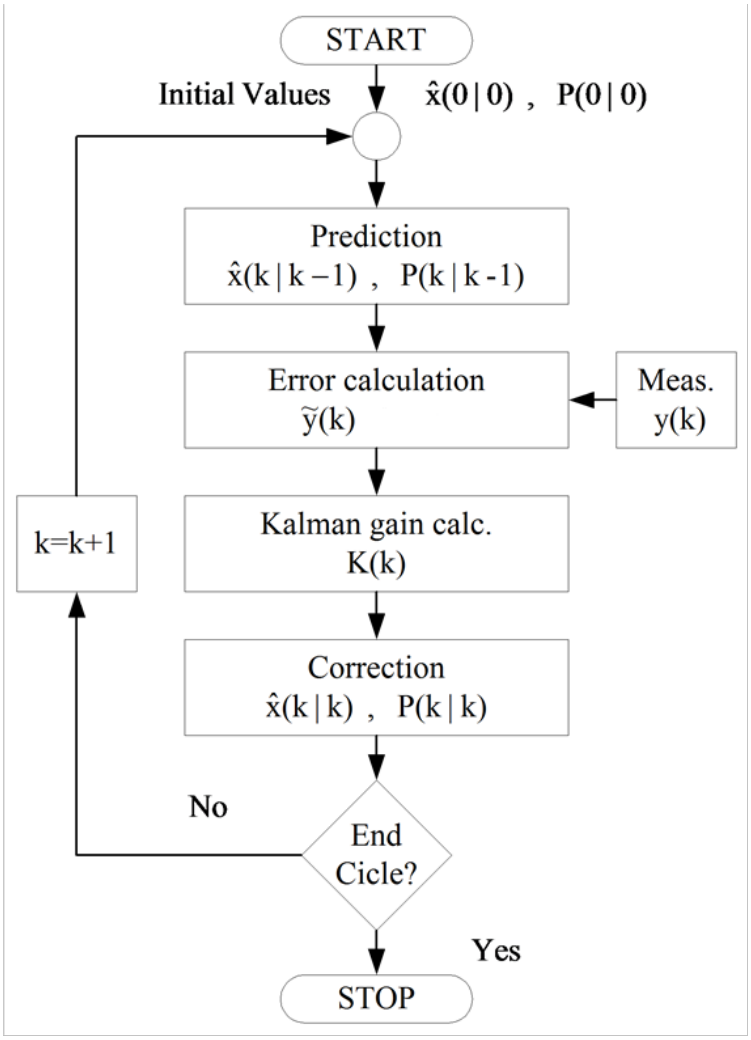


Figure 4.11: EKF flow chart

we know the initial pose of the Turtlebot in the world at the starting time X_{init}. we used Eigen3 libarary to able to make all the computations in a matrix form.

Code 4.2: Initilization Step

```
/****************************** Initilization Step ******************************
* include the Eigen3 libarary
* define the matrices and vectors will be used in the computation X, P, C ...
* assign initial values for the values will be used on the kalman filter equation
*******************************************************************************/
    // Initilization
    MatrixXd Q_alpha =  MatrixXd::Zero(3,3); // evolution model error
    MatrixXd Q_gamma =  MatrixXd::Zero(3,3); // measurement error
    MatrixXd P =  MatrixXd::Zero(3,3);       // state covariance matrix
    MatrixXd C =  MatrixXd::Zero(3,3);       // observation jacobian Matrix
    MatrixXd D =  MatrixXd::Zero(3,3);       // residual covariance
    MatrixXd K =  MatrixXd::Zero(3,3);       // kalman gain

    VectorXd X(3);                           // state vector (x, y, theta)
    VectorXd Y(3);                           // measurement vector ( d1, d2, d3)
    VectorXd Y_hat(3);                       // expected output

    geometry_msgs::Point B1, B2, B3;         // beacon coordinates
    B1.x=0;  B1.y=0;  B2.x=0;  B2.y=5.20; B3.x=4.40; B3.y=3.20;

    X(0)=1;   X(1)=1;   X(2)= 0;
```

- **Prediction Step:** The second step is to make a prediction about where the Turtlebot currently is based on odometry. the prediction is easy to obtain via the /odom topic published by the turtlebot:

Code 4.3: Prediction Step

```
/****************************** Prediction Step *******************************
* subscriber to the /odom2D topic to obtain odometry relevent data
* store the prediction obtained from /odom2D to the predicted state X
* update covariance matrix P
*******************************************************************************/

     // Predicted_state Xk+1/k "extracted from odom2D"
      X(0)= odom2D.pose.x;
      X(1)= odom2D.pose.y;
      X(2)= odom2D.pose.theta;

     // state_trans_uncertainty_noise: Q_alpha "extract covariance matrix from odom2D"
      Q_alpha(0,0)= odom2D.Covariance[0];
      Q_alpha(1,1)= odom2D.Covariance[4];
      Q_alpha(2,2)= odom2D.Covariance[8];

     // update covariance matrix
      P=P+Q_alpha;
```

- **Measurement Update Step:** The third step is to use the available measurement to correct the predicted state.

Code 4.4: correction step

```
/****************************** Correction Step *******************************
* IF WE HAVE RELEVANT MEASUREMENT
* Subscribe to the /beacon_distances topic published by distance_interface_node
* use measurement information to update the prediction state.
* calculate the expected measurement from the predicted state .
* calculate the residual by subtracting the expected measurement from the actual one.
* calculate the uncertainty inherent in the residual.
* calculte the kalman gain.
* update the predicted state using the kalman filter.
* update the covariance matrix by calculating the uncertainty of our final estimate
*******************************************************************************/
// measurement part:

 if (Y(0)> 0.3 && Y(0)< 6 && Y(1)> 0.3 && Y(1)< 6 && Y(2)> 0.3 && Y(2)< 6 ){
     // Actual measurement  Y
       Y(0)= dist.data[0]/100;    // divide over 100 to convert from cm to m
       Y(1)= dist.data[1]/100;
       Y(2)= dist.data[2]/100;

     // Expected measurement Y_hat
       Y_hat(0)= sqrt( (B1.x-X(0))*(B1.x-X(0)) + (B1.y-X(1))*(B1.y-X(1)) );
       Y_hat(1)= sqrt( (B2.x-X(0))*(B2.x-X(0)) + (B2.y-X(1))*(B2.y-X(1)) );
```

```
        Y_hat(2)= sqrt( (B3.x-X(0))*(B3.x-X(0)) + (B3.y-X(1))*(B3.y-X(1)) );


    // Measurement jacobian C_matrix
        C(0,0)= 2*(X(0) - B1.x) / Y_hat(0);
        C(0,1)= 2*(X(1) - B1.y) / Y_hat(0);
        C(1,0)= 2*(X(0) - B1.x) / Y_hat(1);
        C(1,1)= 2*(X(1) - B1.y) / Y_hat(1);
        C(2,0)= 2*(X(0) - B1.x) / Y_hat(2);
        C(2,1)= 2*(X(1) - B1.y) / Y_hat(2);
    // measurement covariance Q_gamma

        Q_gamma(0,0)= (.0000842*Y(0))+0.0158; // we tried to make the variance be variable depend on
        Q_gamma(1,1)= (.0000842*Y(1))+0.0158; // the current distance as the large distances have ...
        Q_gamma(2,2)= (.0000842*Y(2))+0.0158; // large variance .

     // Residual covariance D is Q_(Y-Y_hat)
        D= ( C*P*C.transpose() + Q_gamma );

     // Correction propotional gain  K
        K= P*C.transpose() * D.reverse();


      // Estimation Step: estimated_state Xk+1/k+1 = Xk+1/k + Kk (Yk - Yk_hat)
        X = X + K*(Y-Y_hat);

      // Estimated_covariance_est: Pk+1/k+1 = ( I - Kk*Ck ) * Pk+1/k
        P= ( I - K*C ) * P;
}
```

4.3.1 covariance tuning:

we recorded some data to calculate the measurement error and to measure how the measurement is accurate. using the data set we recorded, we did some analysis and we were able to compensate the error. first we draw the error versus the measurement distance then the actual distance versus the measurement one the relation is shown on figure **??** then we tried to compensate it. the compensation result is shown on figure 4.14f. also the variance and the standard variation is calculated and plotted in figure 4.15

Results

Total numbers (N):	41
Mean (average) value:	0.973853658537
Sample variance (s^2):	**4.40244287805**

Figure 4.12: variance of the measurement error (cm)

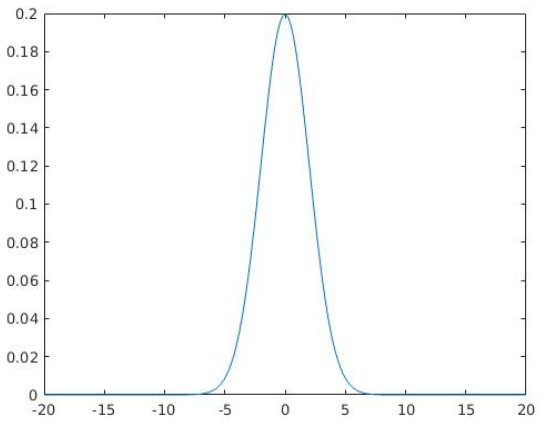
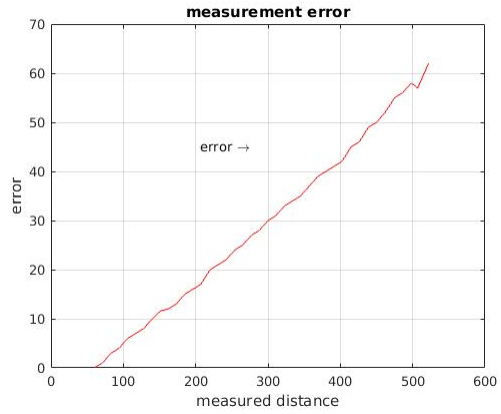
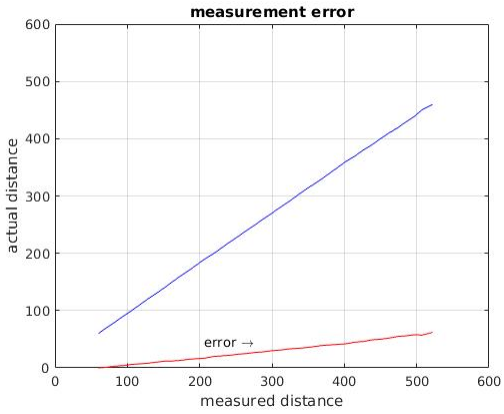


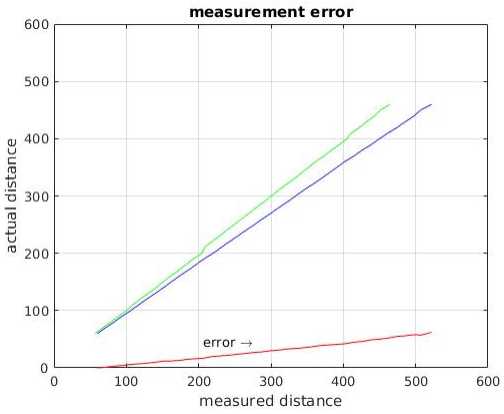
Figure 4.13: error distribution



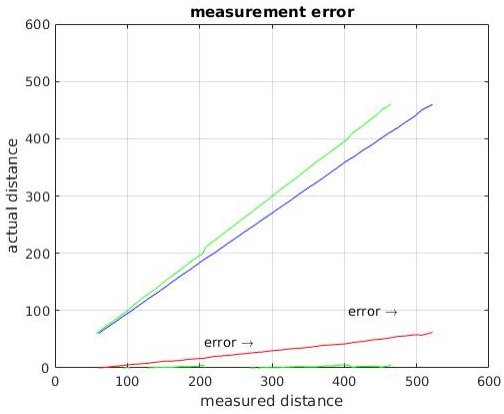
(a) error



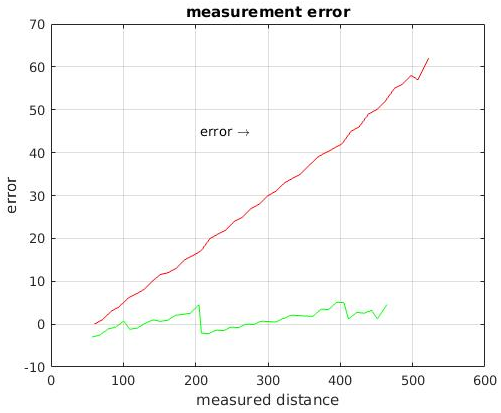
(b) actual vs measured distance 'cm'



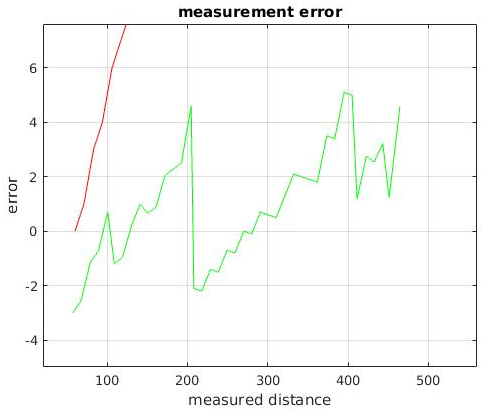
(c) error compensation 'cm'



(d) error after compensation 'cm'



(e) effect of compensation 'cm'



(f) error limits

Figure 4.14: error analysis and compensation 'cm'

4.3.2 Results:

Figure 4.16 shows one track has been recorded for the robot, it shows both the odometry path and the EKF path. figure **??** shows that at the begining the EKF path had a large overshoot -the part that selected by a red circle- and then it begins to converge to the actual pose of the robot.

The behavoir of the kalman filter is not so good due to that the measurement are not stable and also due to the random tuning for both the covariances of the odometry and measurement.

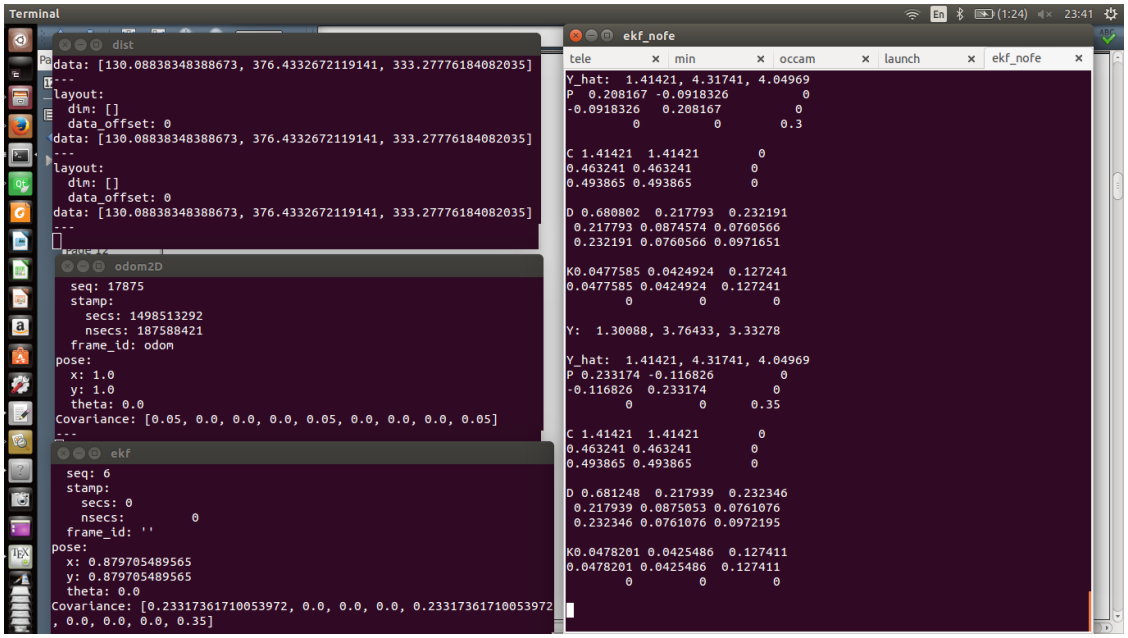
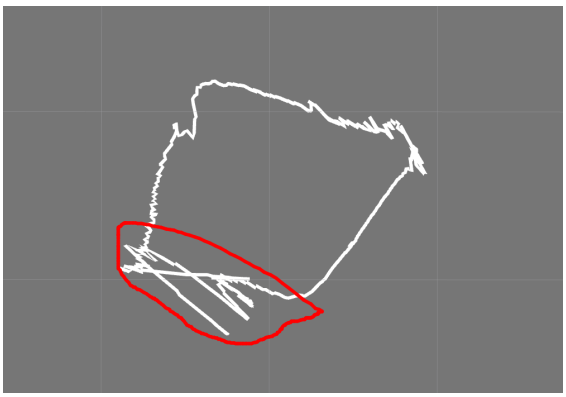
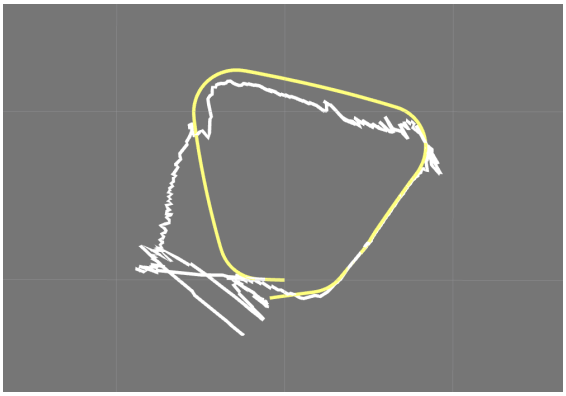


Figure 4.15: odometry and EKF pose

(a) Odometry track



(b) EKF track



(c) odometry "yellow" and EKF pose "white" of the recorded track of the robot

Figure 4.16: Odometry vs EKF

Chapter 5

Visualization with Rviz

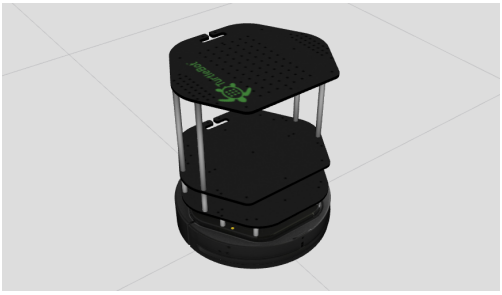


Figure 5.1: 3D TurtleBot2 model in Rviz.

Dedicated ROS nodes were develop in order to publish markers into Rviz to get a better visualization of our measurements. The 4 main things to be visualized are:

- **TurtleBot**: possible to have the 3D model of the robot by launching `roslaunch turtlebot_rviz_launchers view_model.launch`.
- **Beacons**: spheres representing the beacons were also visualized with the appropriate position.
- **Circle representing distance**: The distance obtained from each beacon is represented as a circle to get a better grasp of the trilateration scenario.
- **Paths**: the odometry and the path obtained from the extended Kalman filter are plotted for further comparison.

Markers (`visualization_msgs::Marker`) were mainly used for visual representation. Beacons were modeled with a `SPHERE` and the circles and path with a `LINE_STRIP`.

5.0.1 Static transform publisher

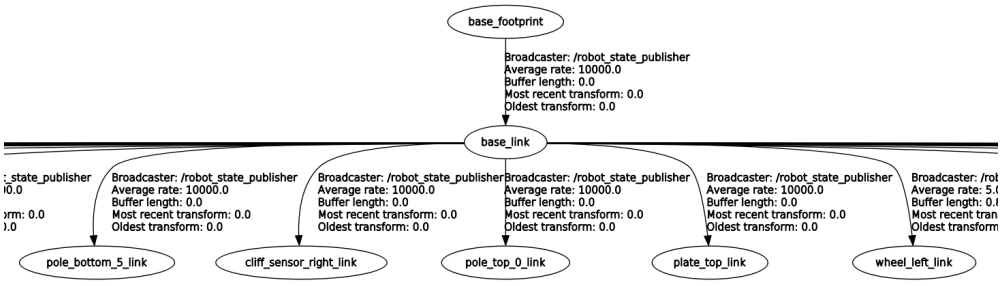


Figure 5.2: tf tree for TurtleBot Rviz model.

The TurtleBot model's position is modified based on actual readings from its sensors so a workaround is necessary to be able to freely assign any given position to the model at any given time. Further analyzing

the tf tree of the model, we notice that everything is connected to the `base_footprint` tf. Therefore, we just need to modify the pose of this transform and all the robot will then move along with it.

The way to achieve this is through a static transform publisher which periodically publishes a static coordinate transform to tf using an (x, y, z) offset in meters and (yaw, pitch, roll) in radians. (yaw is rotation about z, pitch is rotation about y, and roll is rotation about x). The period, in milliseconds, specifies how often to send a transform. Based on the source code `static_transform_publisher.cpp` we created our own publisher which subscribes to a given pose message and sets the transform between the `map` frame and the `base_footprint` one.

5.1 Results

In the following results, odometry is represented by a yellow path and the EKF pose by the white path. All objects are scaled properly.

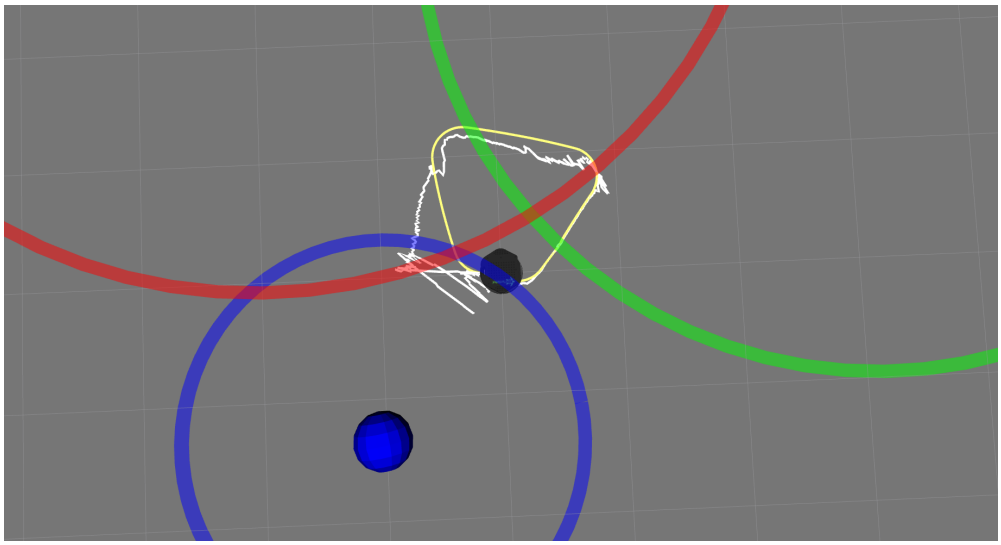


Figure 5.3: tf tree for TurtleBot Rviz model.

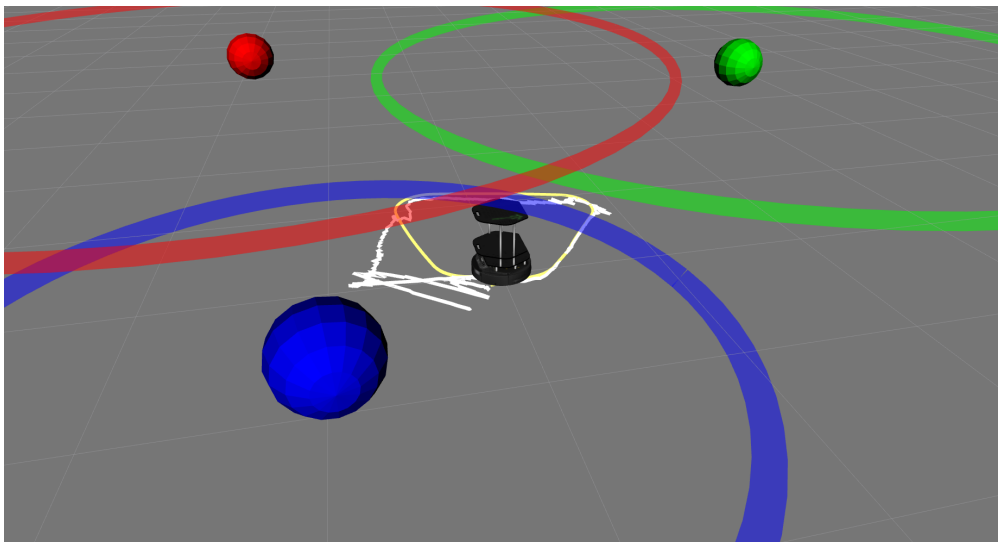


Figure 5.4: tf tree for TurtleBot Rviz model.

Chapter 6

ROS package turtle_ekf

6.1 image_rectifier Node

this node is used to rectify the image published by one camera.

- It subscribes to:
  - **camera_info:** this topic contains the information of the camera "interinsic paramters and distortion coefficients" required to the rectification process.
  - **image_raw;** this topic contains the image itself that will be rectified.
- It publishes to:
  - **image_rect**: it publishes the rectified image through this topic.

The three topics should be precedent by "/camerax/" as x represent the camera that publish the image that will be rectified, this is achieved easily using name spaces in the launch file.

6.2 image_tiles Node

this node is used to combine the 5 images from the 5 different cameras to obtain a single panoramic image.

- It subscribes to:
  - **/camera0/image_rect:** the rectified image obtained from camera0.
  - **/camera1/image_rect:** the rectified image obtained from camera1.
  - **/camera2/image_rect:** the rectified image obtained from camera2.
  - **/camera3/image_rect:** the rectified image obtained from camera3.
  - **/camera4/image_rect:** the rectified image obtained from camera4.
- It publishes to:
  - **image_tiles**: the tiles image obtained by combing the 5 images.

6.3 blob_detector Node

this node is used to combine the 5 images from the 5 different cameras to obtain a single panoramic image.

- It subscribes to:
  - **/image:** the image at which we want to check if there is a beacon or not. this topic will be remapped to another topic contains a real image to be checked, often it is remapped to image_tiles.
  - **/camera0/camera_info:** this topic contains the information of camera0.
  - **/camera1/camera_info:** this topic contains the information of camera1.
  - **/camera2/camera_info:** this topic contains the information of camera2.
  - **/camera3/camera_info:** this topic contains the information of camera3.
  - **/camera4/camera_info:** this topic contains the information of camera4.
- It publishes to:

  - **beacon_distance**: the distance to the detected beacon, it is often remapped to represent a specific beacon "e.g /blue/beacon_distance".
- Parameters:
  - invert: boolean variable to determine if the image should be inverted or not.
  - HMin: Minimum threshold for Hue channel to be used in the inRange function.
  - HMax: Maximum threshold for Hue channel to be used in the inRange function.
  - SMin: Minimum threshold for Saturation channel to be used in the inRange function.
  - SMax: Maximum threshold for Saturation channel to be used in the inRange function.
  - VMin: Minimum threshold for Value channel to be used in the inRange function.
  - VMax: Maximum threshold for Value channel to be used in the inRange function.

6.4 blob_dist_interface Node

this node is used to three distances associated with the three beacons in order (blue, orange, green).

- It subscribes to:
  - **/blue/beacon_distance:** this topic contains the distance to the blue beacon.
  - **/red/beacon_distance:** this topic contains the distance to the red/orange beacon.
  - **/green/beacon_distance:** this topic contains the distance to the green beacon.
- It publishes to:
  - **/beacon_distances**: it publishes the 3 distances associated with the three beacons in order (blue, orange, green)..

6.5 odom_interface Node

this node is used to extract the relevant data from the nav_msgs/Odometry message published through /odom topic and convert them to suitable data type and publishes them using the turtle_ekf/Pose2DWithCovariance message throught /odom2D topic.

- It subscribes to:
  - **/odom:** this topic contains the information of the 3D pose of turtlebot "3-translation, 3-rotation" and its twist "linear and angular velocity along/around the 3-axis x, y, and z".
- It publishes to:
  - **/odom2D**: it publishes the 2D pose $(x, y, theta)$ of the turtlrbot and their corresponding covariance through this topic.

6.6 ekf_node Node

this node is used to implement the Extended kalman Filter equations.

- It subscribes to:
  - **/odom2D:** this topic contains the 2D pose (x,y,theta) of the turtlrbot and their covariance.
  - **/beacon_distances :** this topic contains three distance from the robot to each beacon in order "blue, orange/red, green".
- It publishes to:
  - **/robot_pose_ekf**: it publishes the 2D pose (x,y,theta) of the turtlrbot and their corresponding covariance after modifying the predicted pose using the measurement.

6.7 turtle_ekf/Pose2DWithCovariance Message

This message is used to combine the covariance matrix with the 2D pose of the turtlebot. **Message fields**

- header std_msgs/Header
- Pose2D geometry_msgs/Pose2D
- Covariance float64[9]

Chapter 7

Future work

7.1 Improving image processing

- Searching for different more reliable algorithms that can decrease the image processing time, also try different types of beacons.

7.2 Advanced EKF

- Implementation of a complex algorithm that take into consideration the time required for the image processing.
- using mahalanobois distance to omit the irrelevant inaccurate measurement.
- Finding a robust method to tune the covariance matrices of the measurement and Odometry.

7.3 Azimuth angles

- Considering the angle to the beacons as the measurement rather than or beside the distances, in order to calculate the orientation of the robot theta.
- Comparing between the absolute localization obtained by the distances and the angles, also we can consider the final output of the absolute localization to be a combination of both.
- Using Azimuth angles as measurement with EKF.

7.4 Different localization Methods

- Making a comparision between different method of localization e.g Odometry"relative", Absolute, Sequentail relative and absolute, and Hybrid localization.

References

[1] Rudy Negenborn. Robot localization and kalman filters. Master's thesis, UTRECHT UNIVERSITY, 2003. Thesis number: INF/SCR-03-09.

[2] Open Source Robotics Foundation Inc. Turtlebot. http://www.turtlebot.com/. [Online; accessed 26-June-2017].

[3] Occam Vision Group. Omni 60: High fps omnidirectional camera. http://occamvisiongroup.com/product/omni-60-omnidirectional-camera/, 2017. [Online; accessed 15-June-2017].

[4] OpenCV team. About. http://opencv.org/about.html, 2017. [Online; accessed 15-June-2017].

[5] OpenCV dev team. Camera calibration and 3d reconstruction. http://docs.opencv.org/3.0-beta/modules/calib3d/doc/camera_calibration_and_3d_reconstruction.html#reprojectimageto3d, 2014. [Online; accessed 23-June-2017].

[6] OpenCV dev team. Geometric image transformations. http://docs.opencv.org/3.0-beta/modules/imgproc/doc/geometric_transformations.html#undistort, 2014. [Online; accessed 23-June-2017].

[7] Vikas Gupta. Color spaces in opencv (c++ / python). http://www.learnopencv.com/color-spaces-in-opencv-cpp-python/, May 2017. [Online; accessed 16-June-2017].

[8] Satya Mallick. Blob detection using opencv (python, c++). https://www.learnopencv.com/blob-detection-using-opencv-python-c/, February 2015. [Online; accessed 17-June-2017].

[9] Adrian Rosebrock. Find distance from camera to object/marker using python and opencv. http://www.pyimagesearch.com/2015/01/19/find-distance-camera-objectmarker-using-python-opencv/, January 2015. [Online; accessed 19-June-2017].

[10] Trilateration. https://en.wikipedia.org/wiki/Trilateration, June 2017. [Online; accessed 22-June-2017].

[11] RobOptim Team. Numerical optimization for robotics. http://roboptim.net/, 2016. [Online; accessed 20-June-2017].