

Technical University of Crete



Autonomous Agents

Project Report

Name: Nektarios Sfyris

AM: 2013030058

e-mail: neksfiris@gmail.com

Introduction

One of the main issues existing in robotics is the ability of a robot to move freely in an unknown environment, construct or update a map of it, and be able to locate its position in that map. This problem is referenced as simultaneous localization and mapping (SLAM) and it can be divided into the subproblems of localization and mapping. This project was going to solve the subproblem of active localization with a given map by using the Monte-Carlo localization with particle filters, but because of the inability to create the map limits in the real field, it was not possible given the time that was available.

Given that, this project will focus on finding the limits of a given field, making the robot stay behind those limits and giving it the ability to move freely in that field. This will be solved using machine vision and by controlling mainly the base joints. The robot used for this project is the humanoid Nao of the Softbank Robotics which is provided by the Intelligent Systems Laboratory of Electrical and Computer Engineers. The collection of data from the sensors and the communication with the robot is done using ROS (Robot Operating System). The programming language used is C++.

The field that will be used for experimenting is the right half (4x3 meters – general space it occupies) of the soccer field (which is used for robot soccer) that is presented in Figure 1.

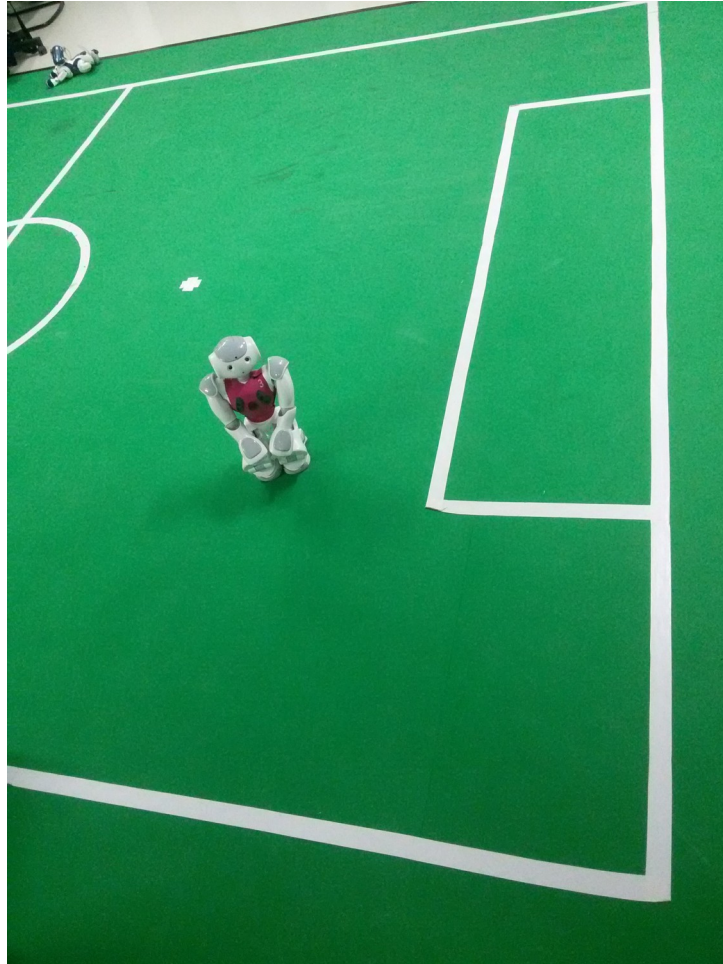


Figure 1.

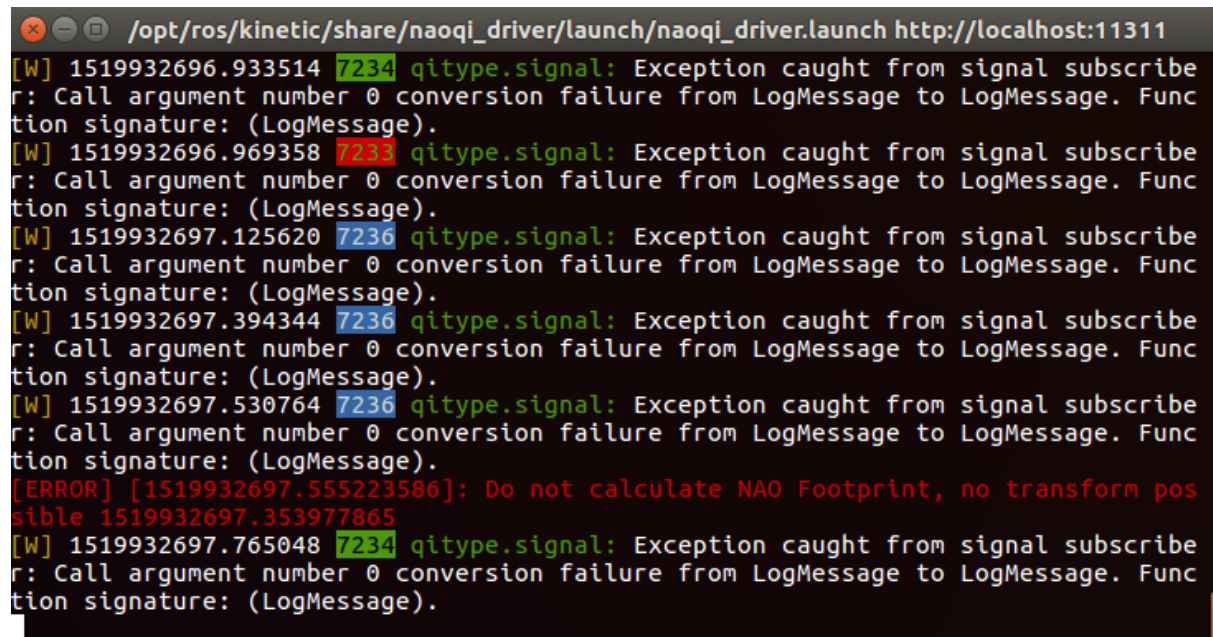
Design & Implementation

ROS is an open-source, meta-operating system for robots. It provides services, hardware abstraction, low-level device control, message passing between processes and package management. In our project we use it to remotely control our robot – Nao from our computer, while processing the data it sends us and giving back feedback for joint movement. It will work as a “bridge” between our Operating System and the Operating System that runs on Nao (Naoqi OS – Linux Based). Using ROS we will be able to see all sensor data and motor states as topics and can access them with ros commands.

The main package that implements the driver for ROS and NAOqi communication is the `naoqi_driver` which must be installed in order for the package delivery to start. After the installation and the `roslaunch` of the executable, we can see in Figure 2, that the package delivery has started.

The command:

```
$ roslaunch naoqi_driver naoqi_driver.launch nao_ip:=**** roscore:=***  
network_interface:=****
```



```
/opt/ros/kinetic/share/naoqi_driver/launch/naoqi_driver.launch http://localhost:11311  
[W] 1519932696.933514 7234 qtype.signal: Exception caught from signal subscribe  
r: Call argument number 0 conversion failure from LogMessage to LogMessage. Func  
tion signature: (LogMessage).  
[W] 1519932696.969358 7235 qtype.signal: Exception caught from signal subscribe  
r: Call argument number 0 conversion failure from LogMessage to LogMessage. Func  
tion signature: (LogMessage).  
[W] 1519932697.125620 7236 qtype.signal: Exception caught from signal subscribe  
r: Call argument number 0 conversion failure from LogMessage to LogMessage. Func  
tion signature: (LogMessage).  
[W] 1519932697.394344 7236 qtype.signal: Exception caught from signal subscribe  
r: Call argument number 0 conversion failure from LogMessage to LogMessage. Func  
tion signature: (LogMessage).  
[W] 1519932697.530764 7236 qtype.signal: Exception caught from signal subscribe  
r: Call argument number 0 conversion failure from LogMessage to LogMessage. Func  
tion signature: (LogMessage).  
[ERROR] [1519932697.555223586]: Do not calculate NAO Footprint, no transform pos  
sible 1519932697.353977865  
[W] 1519932697.765048 7234 qtype.signal: Exception caught from signal subscribe  
r: Call argument number 0 conversion failure from LogMessage to LogMessage. Func  
tion signature: (LogMessage).
```

Figure 2.

The stars at the above command are set network names that we can find for our computer by typing `$ifconfig` in terminal.

Now that the communication is set we can continue. The project is implemented in two nodes, the `line_finder` and the `walker`. `Line_finder`'s main task is, as its name implies, to find the boundary lines for the Nao to move. Mover node is responsible for deciding, when finding or not a boundary line, what action Nao should make. Important library for computer vision, also, is `OpenCV` which was used in this project so the Nao could analyse the input video and find the lines that it needs to recognise.

While we work on catkin workspace, we report every new addition in our project in the `CMakeLists.txt` and `package.xml` files. In `CMakeLists.txt` we declare package, nodes, messages and services. On the other hand, in `package.xml` file we write about package dependencies.

After every change and the right modification of the above two files, we build our project by executing `$catkin_make` while in `catkin_ws` folder. If an

error pops up, a report is written in the command window, about the place the error occurred and its type.

About the Line_finder Node. In this node we do the image processing. First, we create the NodeHandlers to publish and subscribe to topics. The topic we subscribe is the bottom video camera and the topic we advertise is a message made by us called PPVector. PPVector message is a vector of another msg made by us called Points. Points now is a message referring to the beginning and the end of a line with a float about the lines factor. Analytically this is the Points.msg content:

```
-int32 x1  
-int32 y1  
-int32 x2  
-int32 y2  
-float32 factor
```

By advertising PPVector message, we send to mover node information about the lines it has to analyse to decide about Nao's next action.

To find those lines we follow the model specified in Figure 3. While we subscribed to bottom raw image view we have the information the camera provides us. It is a 30 fps image of 320x240 pixel in dimensions. At first, we copy the input image. Then we apply an edge detector on it and we blur it with a 3x3 kernel. Blurring the image on this stage is pretty important because we want to make the detected lines thicker, so the points that construct each line are more in number and closer in range.

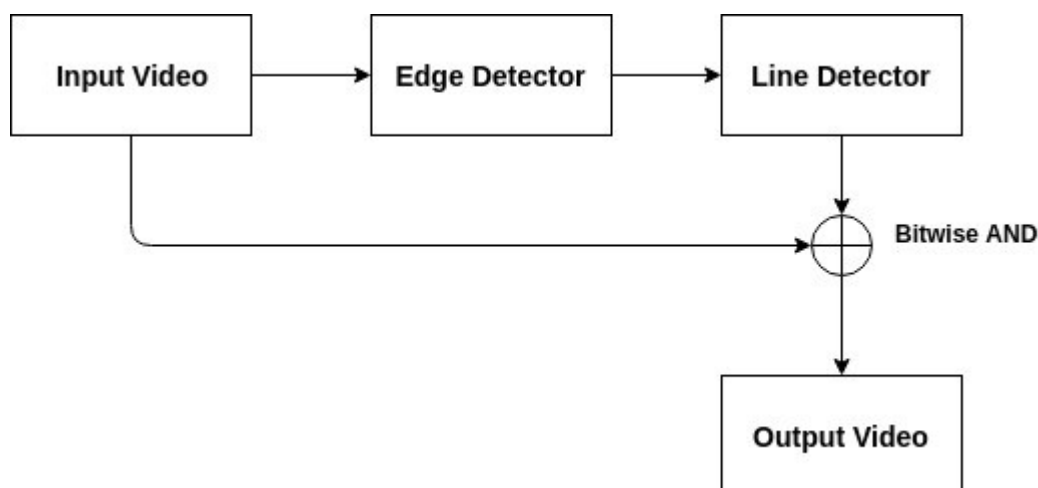


Figure 3.

Subsequently, we apply the probabilistic Hough transform in the blurred

image to detect the lines inside it. After the line detection, we add one more filter that compares all the lines with each other, to find the ones we don't need. To do that it checks the start and end points in each of the two dimensions (x, y) of the lines separately. That creates four boolean expressions that try to find if the distance between each point in each dimension is lower than 10 pixels in two lines. In the case three out of four boolean expressions are true, we believe that the line we are checking exists in a similar manner and therefore we don't take it into account. That saves us from a big number of computations.

Applying the Edge Detector

Canny(src, dst, 200, 600, 3)



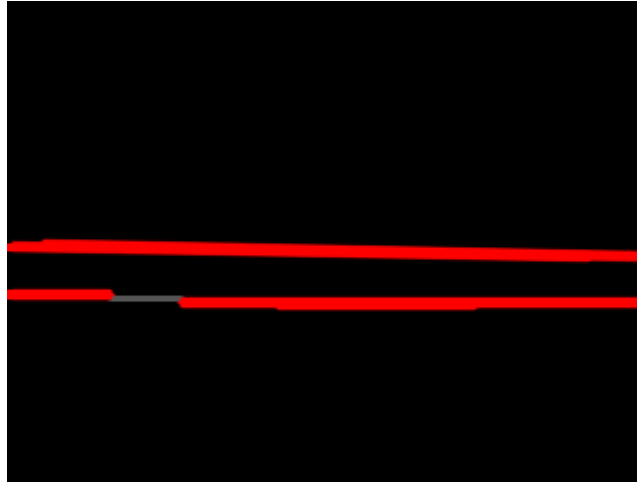
Applying the kernel 3x3 - Blurring

blur(dst, dst, Size(3,3))



Applying the Probabilistic Hough Transform
with the extra filter for line clarification

HoughLinesP(dst, lines, 1, CV_PI/180, 50, 50, 10)



In every camera frame, the extra filter we added cleans about the 30% of the lines the probabilistic Hough transform produces. Since the image processing is over we advertise the PPVector message we analyzed above.

About the **Mover Node**. In this node we take the information provided by the line_finder node, we subscribe to the topic being advertised, we check the position Nao currently is and accordingly we act. The topic we advertise is the HeadPitch of Nao and the base joints. The first thing we publish is the position of the HeadPitch joint and that is because we want the Nao to look down so we can a better understanding of what is close to its feet.

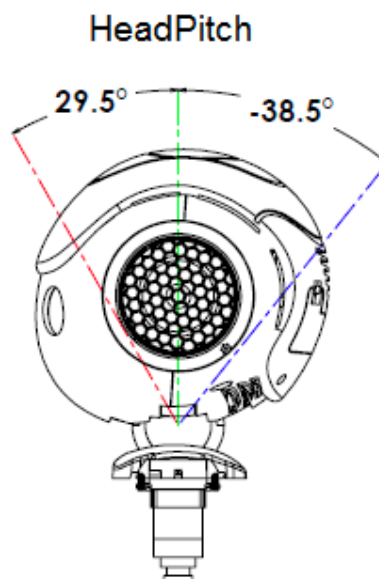


Figure 4.

The range in radians for the HeadPitch joint to move is -0.6720 to 0.5149. Figure 4 shows the range of values in degrees, HeadPitch can take. The value we chose is 0.3 which makes it to be around 20 degrees. The message we publish is as follows :

```
$ rostopic pub -1 /joint_angles naoqi_bridge_msgs/JointAnglesWithSpeed
'{header: {frame_id: "test", seq: 200}, joint_names: ["HeadPitch"],
joint_angles: [0.3], speed: 0.2}'
```

The general structure of our system is presented in Figure 5.

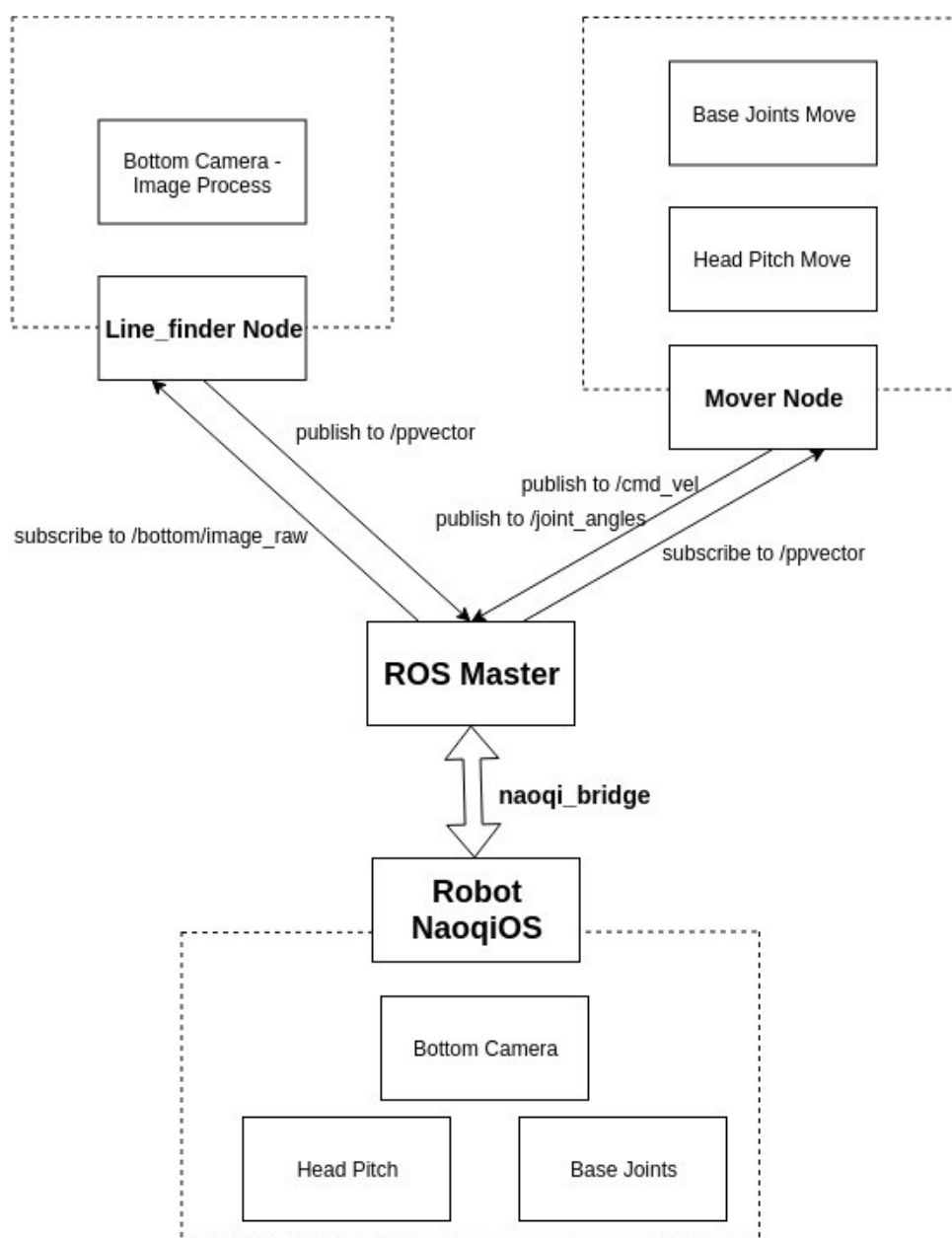


Figure 5.

Generally, we recognise the way the lines in front of Nao are viewed by their factor. If the factor of the line is negative, then the line it sees is closer to his left arm and with a greater distance from his right arm, while if the line has a positive factor, then it is closer to Nao's right arm and away his left arm.

When he notices the line, we check if that line passes a specific area of points in the image. Considering the factor of the line and its position in the image we make a further decision about Nao's action. If the line has a negative factor over a specific value and is in front of him then Nao turns right-handedly to avoid going out of boundaries. In case of a positive factor and in front of us we turn left-handedly.

Results & Demonstration

After we completed building our system in catkin workspace, it was time for testing. The results after running the launch file we created, which combines the function of the two nodes, are presented in Figure 6.



Figure 6-a.



Figure 6-b.



Figure 6-c.



Figure 6-d.



Figure 6-e.



Figure 6-f.

As we described in the above chapter of implementation, the robot will start moving freely in the area inside the half of the field, and when it finds a line in front of it, by processing the image given by its camera it will try to find a way to stay behind that line - inside the limits of the field. That will cause it to turn before crossing the line and start moving again in another direction.

Completing this behavior will be useful for working on the problem of localization later, while the problem of building a map in the real world still exists.

Links that helped

- http://wiki.ros.org/vision_opencv
- http://wiki.ros.org/cv_bridge/Tutorials/UsingCvBridgeToConvertBetweenROSImagesAndOpenCVImages
- https://docs.opencv.org/3.4.0/d9/db0/tutorial_hough_lines.html
- http://docs.ros.org/api/sensor_msgs/html/msg/JointState.html
- http://doc.aldebaran.com/2-1/family/robots/joints_robot.html