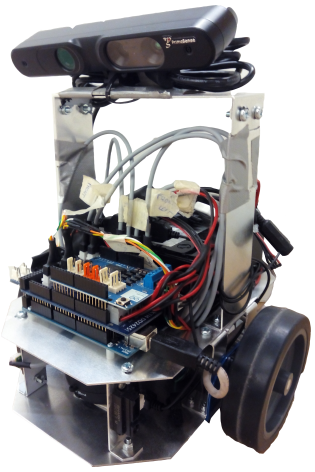
DD2425 - Robotics and Autonomous Systems

Group 6

December 16, 2014

Project Report



Authors

Carlos Gálvez del Postigo
Mathias Lindblom
Tingyu Liu
Gundars Kalns

Abstract

This report describes the work done for the Robotics and Autonomous Systems project. The task consisted on designing a robot that would be able to autonomously explore a maze, avoid obstacles, detected and identify objects and build a map as it navigates. In addition, on a second run, it had to be able to "fetch" the objects as fast as possible using the previously acquired knowledge. The robot was tested in the laboratory and in a contest with satisfactory results. Finally, a performance analysis and some conclusions summarize this report.

Contents

1 Introduction

This report describes the work done for the final project for the course Robotics and Autonomous systems. The purpose is to reflect about ideas and solutions we provided in order to try to reach the goal of this course – making autonomous robot that can explore maze, create map of it and detect and recognize different geometrical objects located in maze.

2 Mechanical design

The robot was built following a standard differential drive configuration (XXXX reference) because of an easier control (e.g.: rotate around it's center point without moving). A picture of the robot can be seen in Figure 1.

(a) Front view (b) Side view

Figure 1: Pictures of the robot

The dimensions are 23 cm x 23 cm wide and long (including wheels) and 29 cm high. Two aluminium plates are connected with four pillars in order to create two floors. On the first floor are located motors, the battery and speaker whereas on the second floor the NUC and the Arduino sandwich can be found. For camera there is pillar which raises it high enough so it reaches 29 cm height. This is required in order to get depth information, which requires a minimum distance of 35 cm from the camera. The IMU is located at the center of the robot, under the second plate, as well as the long range IR sensors looking forward and backwards. The short range IR sensors are located on the pillars which connect first and second floor, with the aim of performing wall following.

3 Electronics

The robot includes the following components:

- Intel NUC computer.
- Arduino Mega 2560 + Motor Shield + custom I/O board.
- PrimeSense RGB-D 1.09 sensor.
- 2 motors + wheels
- 4 Sharp GP2D120 (short range)
- 2 Sharp GP2D12 (long range)

- IMU Phidgets
- Speaker + sound card
- LiPo battery 3-cell 5000 mAh.

3.1 IR sensors

3.1.1 Calibration

Because of non-linear sensor output vs distance curve it is not practical to use raw values from IR sensors in higher level nodes. Therefore it was needed to create node which converts raw data from sensors to distance. As the raw output changes values more rapidly in lower distances, measurements in low range were made quite densely (by measuring sensor output every 0.5 cm) whereas in longer distances such accuracy were not needed because output values were approximately the same even within 5 cm range. Sensor converter node is the only one which subscribes to raw sensors node and other nodes in ROS subscribes directly to distance data.

3.1.2 Filtering

To filter out the noise from the IR sensors a set of independent Kalman Filters XXXXX REF is used due to its simplicity and ability to directly model the sensor and process noise. In particular, we consider the following process and measurement models:

$$x_{t+1} = x_t + \epsilon$$
$$z_t = x_t + \delta$$

, where $\epsilon \sim \mathcal{N}(0, Q)$ and $\delta \sim \mathcal{N}(0, R)$. A value of $R = 0.01$, $Q = 0.04$ was chosen in order to effectively filter but not reduce the bandwidth (responsiveness) too much. XXXX figure?

4 Motion control

To be able to reliably use motors instead of sending just PWM data to motors there needs to be a node which translates linear and angular speeds to necessary PWM values. However as PWM vs speed relation is highly non-linear and depends on many factors such as ground friction it is not possible to just find suitable function for translating speed to PWM. The use of motor controller is therefore crucial. Its task is to control PWM values at the same time getting feedback from encoders. This data then can be used to adjust PWM values so that motor reaches the necessary speed and keeps it. In this project PID controller is used for motor control. Equation 1 shows general PID controller algorithm. In case of motor controller error e(t) is difference between desired angular speed and current angular speed which can be calculated with equation 2. M describes ticks per revolution.

$$u(t) = K_p e(t) + K_i \int_0^t e(\tau)d\tau + K_d \frac{de(t)}{dt} \tag{1}$$

$$\omega(t) = \frac{2\pi\Delta\text{encoder}}{\Delta t \cdot M} \tag{2}$$

With PID controller it is very important to find right parameters otherwise motion can be with oscillations, too slowly rising or could have other problems.

For this project, we have the controller architecture shown in Figure 2.

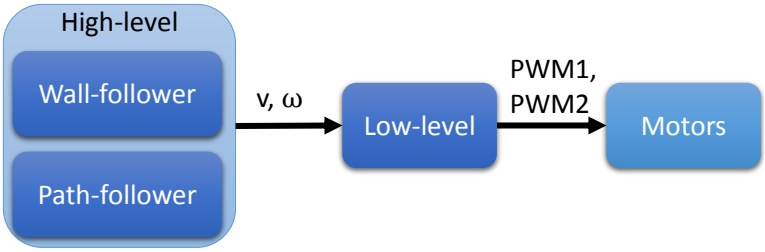


Figure 2: Controller architecture.

In order to translate from v, ω to angular velocity for the wheel, we apply the following:

$$\omega_l = \frac{(v - (B/2.0) \cdot \omega)}{R} \tag{3}$$

$$\omega_r = \frac{(v + (B/2.0) \cdot \omega)}{R} \tag{4}$$

, where B is the distance between wheels, and R is the wheel radio.

Remarks

The main problem we found out is that the motors are **non-linear** in the low-power region: they require a minimum amount of PWM signal to start actually moving. We solved this by simply adding a constant value of 45 and 47 to the left and right motor PWM signal, respectively. This provided a much faster response from the PID controller.

5 Software

In this project ROS software is used. It creates good and feature rich environment for simple nodes which can easily communicate to each other.

5.1 Global picture

The overall block diagram of the system is presented in Figure 3. Each of them will be explained in more detail in the following pages.

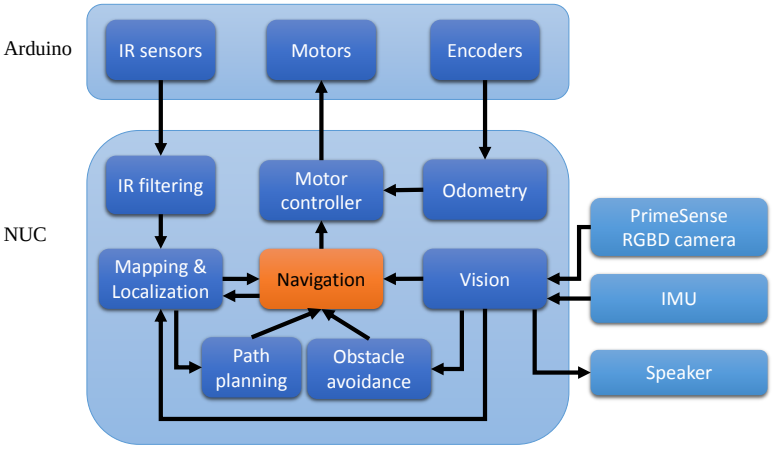


Figure 3: Block diagram of the system

5.2 Odometry

Odometry node takes raw data from motor encoders and translates it into global coordinates and angle according to equations 5.2. This data can be used for mapping and localization. Problem with odometry is that it tends to drift especially when speed increases.

$$\Delta_l = \frac{2\pi \cdot R \cdot n_l}{M} \tag{5}$$

$$\Delta_r = \frac{2\pi \cdot R \cdot n_r}{M} \tag{6}$$

$$\Delta x = \frac{\Delta_l + \Delta_r}{2} \cdot \cos(\theta) \tag{7}$$

$$\Delta y = \frac{\Delta_l + \Delta_r}{2} \cdot \sin(\theta) \tag{8}$$

$$\Delta\theta = \frac{\Delta_l - \Delta_r}{B} \tag{9}$$

, where Δ_l and Δ_r are the distance travelled by the left and right wheel, respectively, R is the wheel radius (assumed to be equal for both wheels), n_l and n_r is the number of ticks that the corresponding has rotated, $M = 360$ is the number of ticks per revolution, B is the distance between wheels and Δx, Δy and $\Delta\theta$ is the change in robot pose in x, y and θ (in global coordinates).

5.3 Mapping

Mapping is done in a separate node called mapping. We create 3 different maps of the same sizes. The so called raw map, the thick map, and the cost map. For the raw and thick map we chose to use the occupancy grid package from ROS. We use three distinct values that each represents the states unknown, free or occupied area. For the cost map we simply used an int array in the form of a ROS message of type Int64MultiArray.

For the raw and thick map the free cells were calculated in similar fashion. Every cell that the robot went over, gets set to a free cell, no matter the value before. All cells touched by the IR sensors on the sides, up to a certain maximum distance or up until we get an occupied reading, also gets set to free cell. All cells in between the end and start points of the 2 sensors on each side gets set to free. However these cells do not overwrite occupied cells since we are just assuming that the area is free. The reason for why this is done is to make sure we do not leave behind lonely unknown cells that can influence the path planning. Furthermore to make the path planning more stable and react faster on close walls in front of the robot, we set all cells 10 cm ahead of the robot to free cells. These also of course do not overwrite occupied cells since we have no assurance that the cells are in fact not occupied by something.

(a) Raw map

(b) Thick map

Figure 4: Generated maps

5.3.1 Raw map

The raw map does no special processing when detecting occupied area. Initially all area is unknown and by driving around and getting corresponding data from IR sensors, free area and occupied area (obstacles and walls) are detected and drawn in map. The four short IR sensors on the sides of the robot provide data about distances to the walls. This data together with position information from odometry node are the basic ingredients to create the map. We also use the laser scanner mentioned in section **??** to fill occupied cells in front of the robot. And finally we use the coordinates of detected objects mentioned in **??** to make sure we will fill cells occupied by objects. The raw map can be observed in Figure 5.

5.3.2 Thick map

The basic map, or raw map, we only use as an intermediate step in creating the processed map, also called the thick map because of how it visually looks. The reason for this maps existence is that the data from the sensors is quite noisy, therefore some processing is needed to mitigate the noisy results and also to build a more solid map to be used for path planning. Compare Figure XXXX with Figure XXXX.

Our first approach to minimize the noise and the gaps in the wall was the following:

1. For each IR sensor, if that sensor detects something, save the corresponding coordinate.
2. When a sensor once again detects something. Compute the distance to the previously saved coordinate.
3. If the computed distance is lower than a certain threshold, we fill every cell in between these points as occupied cells. If not, then we just throw away the saved point.
4. The second coordinate retrieved now becomes the saved coordinate in step 1.
5. Repeat steps 2-4.

The main problem with this method was that depending on the threshold we either got too many gaps in the map, especially at corners, or we got noisy lines that impacted the path planning too much. We tried several different works around for this, both for limiting the noise and gaps and for reducing the negative impact on path planning. But the solutions were simply not good enough and quickly became complex. We replaced this method with another method that works as follows:

1. When an IR sensor detects something, each cell within a certain threshold, 11 cm in our final setup, gets some value increased by one.
2. It is not until the value of a cell reaches a certain threshold, 7 in our final setup, that it becomes an occupied cell.
3. If the value should drop below 7, it would turn back into a free cell.
4. Repeat steps 1-3.

This worked really well for both noise cancellation and for creating solid occupied area. The reason is that in order to get noisy data, we would need at least 7 noisy coordinates within a 10 cm area instead of having just 2 bad readings in close proximity like with the first method. Also, since all points are stored in the map and none removed, we rarely create a gap in the walls. The main reason for the choice of 11 cm in step 1 is that this results in the walls bulking out roughly 12 cm, see Figure 6. This is about half the robots width so as long as the robots center position is at a free cell, it would theoretically never hit a wall. This method also made it easy to update the map, or repainting, since we only change some values around a cell in the thick map when repainting that cell in the raw map.

5.3.3 Cost map

The cost map was created for the path planner to be able to create a path that tries to stay at some distance away from the walls, see Figure XXXX. Without it the path planner would simply take the shortest possible path which would greatly increase the risk of crashing. One might initially think that you could just increase the threshold of 11 cm in the thick map. However, that leads to an increased risk of accidentally filling a entrance completely because of noise. So the cost map can be seen as a complementary map to the thick map. The cost map is created in similar fashion as a potential field where the only thing that changes the cost, or potential, of each cell are the occupied cells in the thick map. It works similarly as the chosen method for the thick map but instead of reacting to occupied cells in the raw map, it reacts on occupied cells in the thick map. So when an occupied cell is added, each cell within some threshold, 20 cm in our final setup, gets its cost increased. The only difference is that we increase the cost the most closests to the occupied cell in question, and the least at the distance of 20 cm. Initially the cost was calculated using the raw map but that led to a quite unfair distribution of cost. This was because the raw map could have large relative differences in compactness of occupied cells at different wall segments, compare Figure XXXX with Figure XXXX.

An important detail is that the cost distribution must be non-linear. If it was linear, all free cells in between occupied area would roughly get the same value. In our robot, we chose the exponential value 1.5 that simply worked well enough. There are several other parameters that could be mentioned, for example the default value that the cells have before any cost is applied. A too low default value results in that cells that are within range for some cost distribution might rarely be considered for the path planner, which in turns leads to poorly chosen routes. A too high value results in that the path planner does not care about the cost at all, since it is roughly the same. However, to get the cost map to work we just needed a somewhat even non-linear distribution.

5.4 Navigation

For navigation the navigation node was created. We wanted to be able explore the entire maze and first we tried using wall following together with a topological map but later changed to path planning.

5.4.1 Wall following

The setup was quite straight forward. We tuned 3 PID's. One was simply for following the wall. The other was for keeping a certain distance to a wall. The third was to turn as close as possible to 90-degrees. Everything actually worked pretty great and we had this setup for the first milestones. However when we started building the topological map we realized that it will be very hard to try to explore the entire maze. Questions started to arise such as how will we handle large open areas? How to deal with multiple entrances next to each other that are only separated by a slim wall segment? We realized that this might get quite complex quickly. So we chose to disband this form of navigation in favor of a more general solution.

5.4.2 Path planning

The path planner has a separate node since it can take up to 1 second to compute the path on a maze of size 16 m^2. The path planner is essentially a greedy best-first search trough the cost map and thick map. How the path planner finds a path to a unknown looks as following:

1. Start the greedy best-first search from the robots front position using the cost map.
2. Never accept a step into an occupied cell in the thick map.
3. Expand the search until you hit a cell with the status unknown.

4. Redo the search from the robots center position until we hit the cell found in step 2.

5. Return the path as a sequence of cells.

See Figure XXXX for an example path to unknown area. The reason for the first search is that want to give some priority for finding unknown cells in front of the robot. The second search is because we want the path to origin from the center of the robot. Step 3 actually uses the second functionality of the path planner. That is to go from point A to point B. When we try to navigate using a path given by the path planner we have several procedures to select a certain point in the path and then we calculate the angle to that point and try to align the robot to that angle.

5.4.3 Global path planning

The global path planning consists on determining the fastest sequence of objects to be visited. This is applied in Phase 2 during the contest.

This is a classical formulation of the Travelling Salesman Problem (TSP). XXXXX ref Here objects can be considered as nodes within a graph and edges that connect them with some cost.

In our particular case, we propose the following formulation:

- The nodes are **not** the object's position, but the position from which the robot first saw the object instead. This is done to make the path planning easier: since the objects are commonly really close to walls, we might not even find a path to them considering that it is computed is the "thickened" map.

- The cost between two objects is measured in terms of the **travel distance**, measured as the size of the path computed from running a Best-First Search from one object to another. This metric is much better than just using the Euclidean Distance between objects (for example, consider the case where the objects are separated by just a wall but the robot cannot go through that wall and has to turn around instead). Even a more accurate metric would include an additional cost for turning, but we did not include this in our formulation.

Therefore, after Phase 1 we can create a high-level graph connecting all the object's positions with the associated costs.

Solution: Genetic Algorithm

Once the TSP problem, we find the best solution for it. As XXXXX shows, this is an NP-hard problem, which means it cannot be solved in polynomial time with standard search algorithms. Instead, a very elegant solution is to use a Genetic Algorithm (GA) XXXX ref, which we apply to this particular problem. The main advantages for this is that it is simple to implement and understand, really scalable and offers a good trade-off between computational time and optimality.

The mechanism behind the GA is quite common and the reader is referred to XXXX to get more details about it. We will mention here the relevant details:

- **Codification**. Genetic algorithms work with strings of chromosomes, so we need to somehow encode the graph information into a string. For this, we gave an integer ID number to each of the nodes, and formed a string as a sequence of IDs (see Figure XXXX).

- **Fitness function**. It determines the performance of a single individual from a generation. We choose the fitness of individual i to be:

$$f(i) = \left[\sum_{i=1}^{N-1} C(i, i+1) \right]^{-1} \tag{10}$$

, where $C(i, i+1)$ is the travel cost of going from node i to node $i+1$, and N is the total number of nodes.

- Parameters.
  - Number of generations: 2000.
  - Number of individuals per generation: 100
  - Number of ellite individuals: 2
  - Cross-over probability: 0.7
  - Mutation probability: 0.05

This implementation gave excellent results, with nearly optimal solutions under a reduced time (around 1.0 s on the NUC). Figure XXXX shows an example test with a graph with 20 nodes.

5.4.4 Exploration, phase 1

The abstract view of the exploration phase is quite simple. It looks like following:

1. Set path planner to search for unknown area.
2. Follow the path.
3. Purge points along the path that we have already passed by checking what point in the path we are closest to at the moment.
4. If we receive information about an object detected at some coordinate. Change path planner to look for a path close to this object for easier object classification. Once the path has been traveled, set the path planner to continue to search for unknown area.
5. Repeat steps 1-4 until the path planner starts sending empty paths.
6. Change path planner to look for a path towards coordinate (0,0), aka. go home.
7. Stop when home.

Under the hood we obviously do a lot more but these are the core elements. For instance when following the current path that we have, we try to go towards the point in the path that is as far away from the robot as possible where the line between the robots center position and the point in question does not come within a certain distance from a occupied node. This makes sure that the robot does not rapidly change direction just because the path changes a lot locally. If you look at Figure XXXX where the path starts to zigzag quite a lot, you get a good example for when this helps stabalize the robots movement.

5.4.5 Retrieving objects, phase 2

This is the abstract view of phase 2 where we try to retrieve the objects:

1. Retrieve the list of object positions as a queue, that was stored in a certain order according to the global path planner.
2. Pop the next object position and set the path planner to find this position.
3. Follow the path.
4. When reached within a certain threshold, stop.
5. Turn the robot against the object for easier recognition and stop for a short amount of time.

6. Repat steps 1-5 until the queue is empty.

7. Change the path planner to look for a path towards coordinate (0,0), aka. go home.

8. Stop when home.

Because of problems with localization we had a hard time running phase 2 succesfully since it would eventually crash into a wall because of drift.

5.5 Localization

Given the lack of time for the project, it was not possible to achieve a fully working localization module, which was really an impediment for running Phase 2. The following solutions were tested.

5.5.1 EKF Localization

Following the theory from the Applied Estimation course (XXX REF), we found out that a good solution was to use an Extended Kalman Filter (XXXX REF) to solve this problem. It would use as input u the data from the encoders, and the measurements would come from a *range-and-bearing* sensor, which provides the distance and angle to a recognized object in the map relative to the robot, as shown in Figure XXXX.

The implementation followed the instructions in XXXXX REF, but the results were not so good. A really large process noise (Q) and really low measurement noise (R) was required to notice some updated in the position. However, we discarded this idea when we realized that the localization could be non-unique: the same θ and ρ could be obtained at any position on a circle of radius ρ around the object, given a proper orientation for the robot. In addition, the object's position would not be stable enough to heavily rely on this information.

5.5.2 Theta correction

A solution which did work was to compute the actual orientation of the robot from it's orientation with respect to the wall. This can be shown in Figure XXXXX

From this, the difference in angle, $\Delta\theta$, can be computed as:

$$\Delta\theta = \tan^{-1}\left(\frac{d_2 - d_1}{D}\right) \tag{11}$$

Finally, we assume the robot has an orientation $\theta 0 \in \{-\pi, -\pi/2, 0, \pi/2, \pi\}$, so the final pose estimate is given by: $\theta = \theta 0 + \Delta\theta$.

This was applied when starting the robot and it worked pretty well; however this could not be applied afterwards since wall following or 90-degree turning were not used in the end in favour of just path following.

5.5.3 On-line localization

Finally, we implemented a promising idea for localization, but we did not have the time to test and debug it properly. The algorithm is as follows:

- The current pose estimate $x_0, y_0, theta_0$ and a map (occupancy grid) are given.
- The orientation (θ) is updated according to the previous section.
- We define a search region around the current pose estimate in X and Y coordinates (e.g.: a square of 10×10 cm.

- Every position $\{x, y\}$ inside that region (within some discretization margin) is assigned a cost based the current sensor readings and the map, according to Equation 12.

$$J(x, y) = \sum_{i=1}^{6} (z_i - \hat{z}_i) \tag{12}$$

, where z_i is the current sensor reading for the ith sensor reading, and $\hat{z}_i$ is the *expected* sensor reading from the position $\{x, y\}$. This is done by raycasting from every sensor position to the saved map.

- The new position is updated as follows:

$$\{x, y\} = \arg\max_{x,y} J(x, y) \tag{13}$$

This approach would give the optimal robot position assuming that the saved map was perfect and we were given perfect sensor readings. Unfortunately this was not the case. In addition, another challenging question is when to run this procedure. Ideally, it would be better to run it after 90-degree turns, but again that was not used in the robot anymore. We are confident that a little bit more time to improve this and test would have probably resulted in satisfactory results.

6 Computer Vision

Another big part of the project involves Computer Vision, allowing the robot to detect and identify objects in the maze. The task is subdivided into smaller subtasks that are now discussed.

6.1 Camera extrinsic calibration

First, a camera extrinsic calibration is required in order to be able to express measurements in camera frame (`camera-link`) into the `world` frame. In this process, a 6 DoF transform is computed. Given that the transformation between `world` and `robot` is computed by the `odometry` node, it is natural to compute here the transformation between `robot` and `camera-link`. Then, it is easy to just use the `tf` package to transform between coordinate frames.

In this particular case, we only compute 4 out of the 6 DoF of the pose, since we assume the roll and yaw of the camera to be 0. Therefore, we only compute the pitch (ϕ) and the translation vector $\mathbf{t}$.

Pitch

To estimate the pitch, we first take the Point Cloud published by the PrimeSense camera and extract the main plane, corresponding to the floor. Then, the normal vector $\mathbf{n}$ is computed. Finally, we extract the pitch of the camera from Equation 14.

$$\phi = \cos^{-1} |n_y| \tag{14}$$

Translation To compute the translation vector $\mathbf{t} = \{t_x, t_y, t_z\}^T$, we take into account the transformation equation 15 in homogeneous coordinates:

$$\tilde{\mathbf{p}}^R = T_R^C \tilde{\mathbf{p}^C} \implies \begin{pmatrix} x^R \\ y^R \\ z^R \\ 1 \end{pmatrix} = \begin{pmatrix} R & \mathbf{t} \\ \mathbf{0} & 1 \end{pmatrix} \begin{pmatrix} x^C \\ y^C \\ z^C \\ 1 \end{pmatrix} \tag{15}$$

, where $\mathbf{p}_R$ and $\mathbf{p}_C$ are a 3D point in robot and camera coordinate frames, respectively, T_R^C is the transformation between the robot and the camera frame, R is the 3D rotation and $\mathbf{t}$ is the translation vector.

R is computed from the yaw(θ), pitch (ϕ) and roll (ψ) angles:

$$R = R(\theta)R(\phi)R(\psi) = \begin{pmatrix} \cos(\phi) & 0 & \sin(\phi) \\ 0 & 1 & 0 \\ -\sin(\phi) & 0 & \cos(\phi) \end{pmatrix} \tag{16}$$

, since $\theta = \psi = 0$.

To compute the translation vector, we require only one known 3D point in robot coordinates that we try to transform into the camera coordinate frame. To do that, we use a "calibration sheet" as shown in Figure XXXXX.

We detect the four corners of the rectangle using a Fast Feature Detector from OpenCV and take the mass center as the known point. It is easy to extract the 3D coordinate by just getting it from the depth image. Finally, we compute the translation vector applying Equation 17.

$$\mathbf{t} = \mathbf{p}^R - R\mathbf{p}^C \tag{17}$$

This concludes the calibration, which provides a transformation from `robot` frame to `camera-link` frame, and is published to the `tf` tree afterwards.

6.1.1 Tilt compensation

Sometimes the robot tilts forward when braking and this greatly affect the calibration. Therefore, we implemented a small solution to fix this using the IMU. We basically compute the tilting of the robot (ϕ^R) taking the measurements from the accelerometer, a_x and a_z:

$$\phi^R = \tan^{-1}\left(\frac{|a_x - a_{x0}|}{a_z}\right) \tag{18}$$

Since the construction is not perfect, we first calibrate the IMU by storing the initial tilt, measured by a_{x0}.

Finally, the transformation between `robot` and `camera-link`, T_0, computed before, is corrected according to Equation 19.

$$T' = T_{\text{IMU}} \cdot T_0 = \begin{pmatrix} R(\phi^R) & \mathbf{0} \\ \mathbf{0} & 1 \end{pmatrix} T_0 \tag{19}$$

6.2 Object detection

The core of the computer vision component is divided into two modules. Object detection is mean to be fast (close to real-time computing time) and robust. Therefore, our aim was to optimize it as much as possible. The following pipeline was implemented.

1. First, we subscribe to the RGB and Depth images from the camera (for which we use the `ApproximateTime` and `Synchronizer` packages in ROS) and build a point cloud out of them. The reason for doing this is that for some reason the PrimeSense did not publish its own point cloud at a fixed rate, and it was quite far from 30 Hz (e.g.: every 200 ms or so), which was unacceptable. To make it faster, we downsampled the images by a factor of 4 in X and Y. For transforming points from 2D to 3D, we reverted the projection equation, as shown in Equation 20.

$$X = z \cdot \frac{u - c_x}{f_x} \quad ; \quad Y = z \cdot \frac{v - c_y}{f_y} \quad ; \quad Z = z \tag{20}$$

where $\{X, Y, Z\}$ is the 3D point in `camera-link` frame, $\{u, v\}$ is the 2D point from the RGB image, z is the depth at the given point, and c_x, c_y, f_x, f_y are the camera intrinsics, which are obtained from the `CameraInfo` message.

2. We transform the point cloud into the `robot` coordinate frame and extract the floor as a separate point cloud. This is done using a simple `PassThrough` filter, preserving the points whose `z` component is smaller than 0.01 m. We could have used the `Plane Segmentation` library from PCL, but this was more computationally expensive and would not always work since it would remove only the dominant plane, which might not be always the floor.

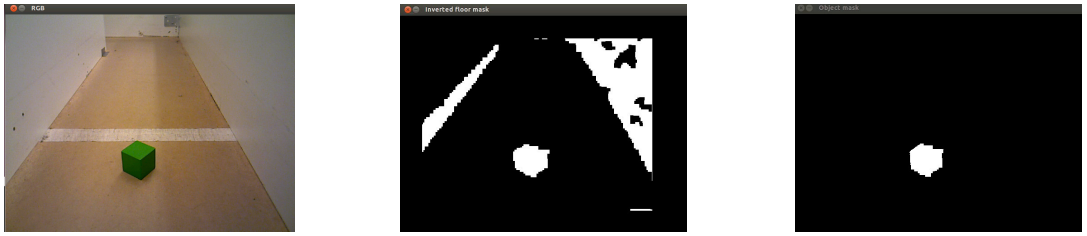
3. A binary mask is created representing the floor by just reprojecting the floor point cloud extracted before. The process is done simply by reverting Equation 20 and getting u and v. Since a subsampling was done at the beginning, it was necessary to apply dilation in order to have a solid mask.

4. Next, the floor is removed from the original RGB image by an AND operation with the negated floor mask. Now it that the yellow floor will not trigger false positives, it is possible to perform **color filtering**. We select to do this in the HSV color space given its better robustness against illumination. This is performed using a bank of 5 color filters (red, green, blue, yellow, purple) with ranges in H and S manually tuned for the application. The function `inRange` of OpenCV is used to quickly perform this filtering. The result is a set of binary masks, from which we select the one with a larger color response. After that, we find contours and filter them by size (a minimum is required) and the aspect ratio (a maximum value of 2 is used).

   If a contour still remains, it is likely that it belongs to an object. The biggest contour that fulfill these conditions is taken and a binary mask is created out of it using the `drawContours` function.

5. In case a contour belonging to a coloured object was found, we compute the 3D position of its mass center both in robot and world coordinates. If the object has not been recognized yet, and if it is close enough to the robot (less than 30 cm from it), the recognition module is called.

Figure 7 shows the process.



(a) Original RGB image (b) Inverse of the floor mask (c) Object mask

Figure 5: Detection process

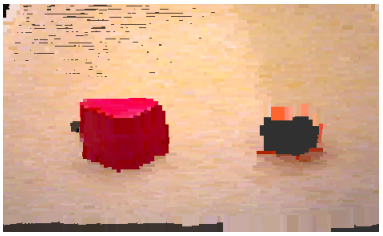
6.3 Object recognition

A different number of approaches for object recognition have been tried. We decided to have a parallel research in both 2D and 3D methods for object recognition to analyze their advantages and disadvantages.

6.3.1 3D object recognition

First, several attempts to perform 3D recognition were tried. We started using **feature matching** object recognition. The first question to ask was: what features? The task objects were too simple and had not texture, and therefore no keypoints could be extracted from them. Therefore, we tried to just subsample the point cloud to see if those points could act as keypoints.

Then, we tried several feature descriptors. We started with SHOT and PFHRGB, which were proven to provide a great accuracy (XXXXX REF). They did work well for cubes and balls, but we sadly discovered that it was not possible to do that with the hollow objects: they became "invisible" to the PrimeSense due to interferences with the IR projection, as can be seen on Figure 8a:



(a) Red cube and "invisible" Patric

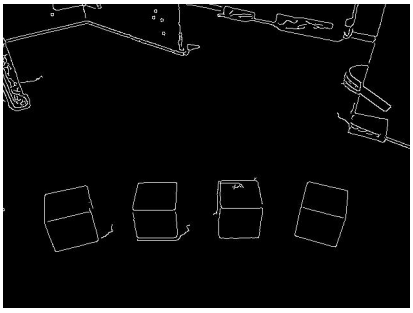
The computational time was also quite expensive (around 1-2 seconds), and it was not robust against illumination since it was using RGB data. We further tried shape-only descriptors (PFH and FPFH) without a significant increase in performance. Therefore we abandoned the idea of 3D-only recognition.

6.3.2 2D object recognition

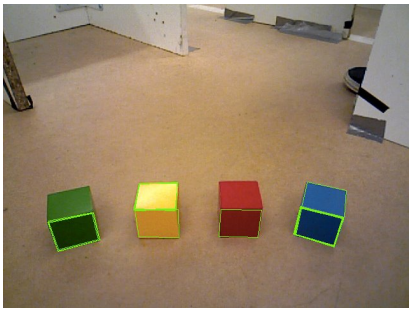
The "2D vision" refers to object recognition using only the RBG image from the camera as input and ignore the depth information, 2D vision returns a belief in terms of probability (0%-100%) of which object it thinks is in the input image (85% red cube, 13% yellow ball for example). The 2D vision basically combines color information with shape information of an object for identification. The 2D vision was coded, tested and implemented to the robot but was not used during the context due to worse performance compared with 3D object recognition.

The 2D vision contains 3 functions, one for circle recognition, one for triangle recognition and one for square recognition. The circles recognition function uses OpenCV built-in "Circle Hough Transform" method to find circular objects in an image. The triangle and square recognition functions perform contour analysis to identify a triangle or a square in an image by analysing the number of lines in the contours and their crossing angles (for example a square always has 4 lines and 90 degrees between the lines). 2D vision also needed to perform some image transformations to the raw input RGB image before running the recognition algorithms to reach better performance and robustness, these transformations include resize/rescale of the original image, transforming RGB image into gray-scale image, Gaussian smoothing filter, Canny method to find contours in an image and finally dilate and erode images to remove single pixel objects and fill empty holes. See Figures 8b and 8c.

2D vision is perhaps the most "natural way" of how a robot can recognize objects, just like we humans, when we see an object we check out the color and the shape of the object and then decide/analyse what the object is. Unfortunately many problems can occur even in such a "controlled" environment such as the maze that makes the 2D vision very hard to be 100% robust. Different light condition in the environment (due to weather, lamp, shadow etc), different rotation of how the object is placed in front of the camera, and noise from the camera and the environment (robot vibration, tape on the walls etc) can all make the 2D vision perform worse than expected. Resize/rescale the raw image was a method used to successfully reduce execution time of the algorithm; it could also



(b) Canny method to find lines and contours.



(c) Found squares

Figure 6: 2D recognition example: square detection.

be used to remove noise from the image. Another way to reduce image noise is Gaussian smoothing, however it also removes useful information such as object contours if the parameters are not chosen carefully, the same problems apply to dilate and erode method, essentially it is always a trade-off between removing unnecessary information and keeping useful information. Canny method was used for finding lines and contours in an image, it works best on gray-scale image so we must convert all image to gray-scale image. When converting an RGB image into a gray-scale image, if you already knew for example that the object being recognized is red, one smart way of doing it is to just split the RGB image into 3 channels and choose the R channel as your gray-scale image, this way most of the red contours are being saved compared with the normal OpenCV "RGB2GRAY" function. An attempt to fight against different light condition was to multiply the raw RGB image with different constants such as from 0.8 (to make the image darker) and 1.5 (to make the image brighter) with 0.1 intervals. This method greatly improved the recognition rate and made sure that at least the object will be detected in one or two light constants. However misdetection problem also followed and it also caused problem for calculating the probability of recognizing an object.

Text below the images: Fig1: Fig2: .

6.3.3 Final approach

The final approach was to determine a **probabilistic formulation** of the problem in order to gain in robustness and be able to reason about the certainty of the recognition. The most likely object o^* was determined using a MAP estimate given the measurements in 3D shape z^s and color z^c, according to Equation 21.

$$o^* = \arg\max_o p(o|z^s, z^c) = \arg\max_o \eta p(z^s, z^c|o)p(o) = \arg\max_o p(z^s|o)p(z^c|o) \tag{21}$$

Here we are assuming that all the objects have the same prior probabilities and that the shape and color are conditionally independent given the object. We now describe the way to compute the previous probabilities.

3D shape probability For the task of 3D shape recognition, we make use of the VFH descriptor included in the PCL library. Unlike the previously described descriptors (FPFH, PFHRGB,...), this is a **global** descriptor, which means it does not have to be applied to all possible keypoints of the object, but rather it is applied to the whole point cloud. It is rotation and scale invariant, which is important for the application.

We build a 3D model database for the objects. We realized early in the course that **hollow objects** are invisible to the PrimeSense due to interference generated by the IR projector. Therefore, we only built 3D models for the ball and cube. 50 models were taken for each of them at different distances from the camera and with different poses.

After that, the test point cloud (which contains a potential object) is matched against the database. This is done quickly by histogram matching, for which OpenCV has some metrics. We opted for using the **correlation** metric between histograms. W

A value of -1 means complete opposite correlation, while 1 means perfect match. Therefore, we compute the probability of 3D shape as follows:

$$p(z^s = \text{cube}|o) = \max_i 0.5 \cdot (d(H_{\text{test}}, H^i_{\text{cube}}) + 1.0) \tag{22}$$

$$p(z^s = \text{ball}|o) = \max_i 0.5 \cdot (d(H_{\text{test}}, H^i_{\text{ball}}) + 1.0) \tag{23}$$

$$\tag{24}$$

where H^i_{cube} and H^i_{ball} are the descriptors of the 3D models from the database.

Hollow probability

We also take into account the probability that an object is hollow, according to Equation 25.

$$p(z^s = \text{other}|o) = 1.0 - N_d/N_T \tag{25}$$

, where N_d is the number of pixels inside the object mask that have a valid (non-zero, not NaN) depth and N_c is the number of pixels within the object mask.

These three probabilities are normalized afterwards so that they add up to 1.

Color probability

In this task, it is not possible to recognize all the objects without color information. For this we implemented a Bayes Classifier XXXX ref on the HSV color model. We currently only use the H and S components, which influence most the outcome.

The first step is to create a **color model**. We decided to have independent Gaussian distributions for 7 colors (red, green, blue, yellow, purple, orange, light green) and for each of the channels (H and S), so the total number of parameters is $7 \cdot 2 \cdot 2 = 28$. For every color, the model $\mu_h, \sigma_h, mu_s, \sigma_s$ is computed from training data. In particular, 50 pictures from each of the 10 objects were taken at different positions and different light conditions. In sum, 500 pictures form the color model. An application was implemented in order to speed up the capture of data. In particular, the application would cut out an image from a predefined region of interested at a fixed rate (e.g.: every 5 seconds). Finally, the color model was computed by extracting the mean and variance values for each color and channel.

During the test phase, the Bayes Classifier was applied to every pixel included in the object mask to determine to which class it belonged to. Finally, the probability of any color is computed as follows:

$$p(z^c = c|o) = N_c/N_T \tag{26}$$

, where N_c is the number of pixels belonging to class c, and N_T is the total number of pixels in the object mask.

6.4 Obstacle detection

In addition to the so-called "IR bumper" (see section XXXXXXXXXXXXX), an obstacle avoidance solution using the PrimeSense camera was implemented. The aim was to detect also objects right in front of the robot, especially the edge of walls that would not be detected by the front IR sensor.

To do this, we simulated a laser scanner using the depth image. The implemented pipeline is as follows:

1. Build a point cloud from the depth image and transform it into the `robot` coordinate frame. Here color was not required.

2. Filter the point cloud to determine the working region. In particular, the 3D points following these constraints were selected:

   - $0.2 < x < 0.4$
   - $-0.115 < y < 0.115$
   - $0.05 < z < 0.06$

   All the values are expressed in meters. In other words, a thin slice (0.01 m thick) covering an area of a width equals to the robot width (23 cm) and a maximum depth of 0.4 m (from the center of the robot).

3. Simulate laser beams. For this, the previous point cloud was filtered multiple times (once per line) using the constraint: $i < y < i + 0.005$, for $i = -0.115, \ldots, 0.115$, increasing in steps of 0.005 meters. That is: lines with 0.005m resolution where taken.

4. Analyze the occupancy of the laser beam. This was done simply by checking if the previously filtered cloud lines contained any point. In that case, information about the minimum depth at which there was a point in the cloud was obtained.

The result of the algorithm is a set of lines in the forward direction of the robot that determine the occupancy of the region in front of it. An example is shown in Figure 9

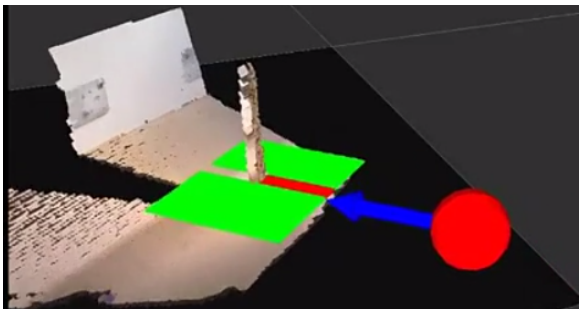


Figure 7: Simulated laser scanner detecting thin wall.

This information is afterwards sent to the `mapping` node, which sets to *blocked* the cells corresponding to the end point of the lines of the laser scan that do not show free space.

7 Discussion

7.1 Results

7.2 System Evaluation

7.3 Project management

8 Conclusions

Conclusions

References

[1] S. Thrun, W. Burgard, and D. Fox. *Probabilistic Robotics (Intelligent Robotics and Autonomous Agents)*. The MIT Press, 2005.