DD2425 - Robotics and Autonomous Systems

Group 6

December 16, 2014

Project Report

Authors

Carlos Gálvez del Postigo
Mathias Lindblom
Tingyu Liu
Gundars Kalns

Abstract

Abstract Abstract

Contents

1 Introduction

This is the project report in course Robotics and Autonomous systems. The purpose of this report is to reflect ideas and solutions we provided for trying 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

Robot size is 23 cm x 23 cm width and length (including wheels) and 29 cm height. Same width and length were chosen due to ease of maneuverability. For the same reason wheels are located in the middle of its sides and corners of the robot are cut off in 45Ř angle. Two aluminium plates are connected with pillars in such way making two floors. On the first floor are located motors, acumulator battery and speaker whereas on the second floor is located NUC and Arduino sandwitch. For camera there is pillar which raises it high enough so it reaches 29 cm height. IMU is located under the second plate as well as long range IR sensors which points to front and back direction. Short range IR sensors are located on the pillars which connect first and second floor.

$$y = f(x) \tag{1}$$

Equation 1 shows that...
Thrun *et al.*[1] show that...

3 Electronics

3.1 NUC

3.2 Arduino

3.3 IMU

3.4 IR sensors

3.4.1 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.

3.4.2 Filtering

To filter out noise from IR sensors data Kalman filter is used.

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. 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 2 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 3. N describes ticks per revolution.

$$u(t) = K_p e(t) + K_i \int_0^t e(\tau) d\tau + K_d \frac{de(t)}{dt} \tag{2}$$

$$\omega(t) = \frac{2\pi \Delta encoder}{\Delta t * N} \tag{3}$$

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.

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

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.

$$distance_{left} = \frac{2\pi * Wheelradius * encoder_{left}}{ticksperrevolution} \tag{4}$$

$$distance_{right} = \frac{2\pi * Wheelradius * encoder_{right}}{ticksperrevolution} \tag{5}$$

$$\Delta x = \frac{distance_{left} + distance_{right}}{2} * cos(\Phi) \tag{6}$$

$$\Delta y = \frac{distance_{left} + distance_{right}}{2} * sin(\Phi) \tag{7}$$

$$\Delta\Phi = \frac{distance_{left} - distance_{right}}{Wheelbase} \tag{8}$$

5.3 Mapping

For map we chose to use occupancy grid package from ROS. IR sensors provide data about distances to the walls. This data together with position information from odometry node are used to make a map. Initially all area is unknown and by driving around and getting corresponding data from IR sensors free area and obstacles are detected and drawn in map. Data from sensors are quite noisy therefore some processing is needed to mitigate noisy results.

5.4 Exploration

For exploration there was created navigation node.

5.4.1 Wall following

5.4.2 BFS Search

5.5 Path planning

5.5.1 Global

5.5.2 Local

5.6 Localization

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 9.

$$\phi = \cos^{-1} |n_y| \tag{9}$$

Translation To compute the translation vector $\mathbf{t} = \{t_x, t_y, t_z\}^T$, we take into account the transformation equation 10 in homogeneous coordinates:

$$\tilde{\mathbf{p}}^R = T_R^C \tilde{\mathbf{p}^C} \implies \begin{pmatrix} x^R \\ y^R \\ z^R \\ 1 \end{pmatrix} = \begin{pmatrix} R & \mathbf{t} \\ \mathbf{0} & 1 \end{pmatrix} \begin{pmatrix} x^C \\ y^C \\ z^C \\ 1 \end{pmatrix} \tag{10}$$

, 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:

$$R = R(\theta)R(\phi)R(\psi) = \begin{pmatrix} \cos(\phi) & 0 & \sin(\phi) \\ 0 & 1 & 0 \\ -\sin(\phi) & 0 & \cos(\phi) \end{pmatrix} \tag{11}$$

, 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 12.

$$\mathbf{t} = \mathbf{p}^R - R\mathbf{p}^C \tag{12}$$

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:

$$\phi^R = \tan^{-1}\left(\frac{|a_x - a_{x0}|}{a_z}\right) \tag{13}$$

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 14.

$$T' = T_{\text{IMU}} \cdot T_0 = \begin{pmatrix} R(\phi^R) & \mathbf{0} \\ \mathbf{0} & 1 \end{pmatrix} T_0 \tag{14}$$

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 object detection pipeline is presented in Figure XXXXXX.

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 15.

$$X = z \cdot \frac{u - c_x}{f_x} \quad ; \quad Y = z \cdot \frac{v - c_y}{f_y} \quad ; \quad ; Z = z \tag{15}$$

   where $\{X, Y, Z\}$ is the 3D point, $\{u, v\}$ 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 we get 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 15 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.

6.3 Object recognition

6.4 Obstacle detection

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.