Final report

Team Gamma

Ajda Frankovič, Martin Preradović, Niki Bizjak

1 Introduction

2 Methods

In this section, the algorithms and methods we have used in our system are presented.

2.1 Camera pixel to world position

Object detectors must be able to compute world positions of detected objects. To do that, `rgb` camera and depth images are used. Because the robot is moving, and the environment is changing very fast, we must make sure that the received colour and depth image are synchronized with one another. Even a small delay between images can cause computed world positions to be inaccurate.

Figure 1: Pinhole camera model

When object is detected on `rgb` colour image, the pixel position on image plane is known. Using camera calibration matrix, we can obtain information about focal length and image format. Focal length is the distance from the centre of camera coordinate system to image plane. The image format gives us information about camera sensor width and height.

Using this information, the ray from the centre of camera coordinate system to the detected object pixel on image plane can be created. Using depth image information, we can obtain the distance from camera centre to the detected object in space. Using the computed ray and distance to the object, we can compute its position in three-dimensional space in camera frame.

Using frame transformations, we can then compute the detected object position in world coordinate system using transformation matrix.

In our system, the face and ring positions are computed this way. The cylinder detection works with point clouds, where the points are already computed in the world frame.

2.2 Faces

In this section, we present algorithms that we have used to detect faces, compute their orientation in space and classify them.

2.2.1 Face detection

Face detection was done using Viola-Jones face detection algorithm. It was chosen because it can be run in real-time and it has a high detection rate.

2.2.2 Face orientation computation

After the face has been detected in image, the position in world coordinate system is computed. The approaching point is then calculated using static map information. Approaching point is a point directly in front of the face that the robot must visit.

To compute approaching point, we must first get the face orientation. To do that, the static map information is used. The calculated face position in world coordinates is first converted to map coordinates. Then, the corresponding pixel in map is calculated.

Because of depth sensor inaccuracy, the computed map pixel coordinate doesn't always lie on the wall. In figure 2, computed face pixel is coloured red.

Using a simple breadth first search in face pixel position proximity, the closest wall pixel is determined. In figure 2, this pixel is also coloured red.

After wall pixel has been found, the Hough line finding algorithm is applied to an area around wall pixel. The detected lines in figure 2 are rep-

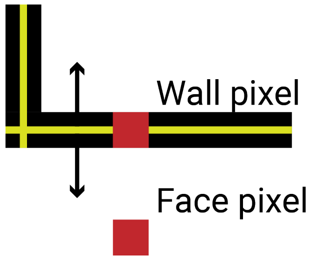

Figure 2: Face orientation computation

resented with green colour. If more than one line is found, the line which is closest to the wall pixel is selected. After the line is found, there are two possible orientations as represented in figure 2. The vector that is pointing in the direction of the robot is chosen to be the face orientation.

A similar algorithm is used to detect ring orientation.

2.2.3 Face classification

After faces are detected and localized in the world, the faces are identified.

2.3 Colour classification

The robot must be able to detect ring and cylinder colours. There are six possible classes: black, white, red, green, blue and yellow.

In homework 2, we tested different colour spaces and classification models and concluded that the k-nearest neighbours algorithm worked best. The colour space with the highest accuracy in homework was `HSV` colour model, but in the Gazebo simulator, `RGB` space worked better.

So the colour classifier in the final task uses the k-nearest neighbours algorithm and takes in an input vector in $(red, green, blue)$ format and returns a colour label.

Because of the non-even lighting in the Gazebo simulator (and probably real world too), the classifier sometimes returns incorrect label. The problem was solved using the robustification process as described in the 2.6 section. Colour classifier is run multiple times on different object detections and the most frequent colour is chosen as the object colour.

2.4 Rings

In this section, the algorithm for ring detection is presented.

2.4.1 Ring detection

2.4.2 Ring color detection

2.5 Cylinder detection

2.5.1 Removal of planes

2.5.2 Cylinder detection

2.5.3 Approaching point computation

2.6 Robustification

The sensor data that the robot receives is not always accurate. When object is detected, the depth sensor might have computed the distance inaccurately or the robot was not localized. The detectors might also return false positives and so on.

To combat this, a robustification is performed on detected objects. The object detectors are sending detections to robustifier, which collects data and determines if the detection is a true positive or not.

For detection to be considered true positive, the following must hold:

1. Detected object should be detected at least `threshold` times
2. There must be at least `minimum distance` meters between this detection and every other previous detection.

If two detections are close enough (the distance is less than `minimum distance`), they are considered the same object. The world position and approaching point position of those two detections are averaged. So the object position is closer to its actual position in space.

2.7 Smart exploration of space

2.8 Fine movement

The `move_base` package can be used with `ROS` to move the mobile base around the map. It uses local and global planner information to avoid obstacles and try to reach the given goal. If it fails to reach the goal, it tries to clear its local map. If that fails, the robot stops trying to reach the goal and notifies us that it has failed to move to the goal.

The most common reason for failure is that the goal is too close to any wall in our world. In our final task, the robot had to be able to pick up rings that are positioned very close to the wall. So close actually, that the package is unable to reach the goal. To do that, odometry data must be used combined with `Twist` messages to move the robot manually.

Because the robot must be able to visit the ring from specific direction (ring orientation), we used a modified version of `A*` algorithm, that not only finds the shortest path between two points, but also uses the starting and ending robot orientation.

Because the world is flat and the robot can only rotate along one axis, any robot pose can be represented using a tuple (x, y, φ), where φ is the orientation along the vertical axis. In theory, the φ can be any real number, but in our case, the robot will only have a finite number of possible orientations (the first one being $[0, \frac{2\pi}{n}]$, where n is the number of all orientations). In our final task, we found that $n = 8$ or $n = 16$ works best.
The algorithm will use map to access environment information, so the x and y represent pixel coordinate in map coordinate system.

If the robot is currently in state $\vec{s_i} = (x_i, y_i, \varphi_i)$, we can define the next states to be:

1. Move forward
$$\vec{s_{i+1}} = (x_i + \alpha \cos \varphi_i, y_i + \alpha \sin \varphi_i, \varphi_i)$$
2. Rotate left by $\frac{2\pi}{n}$ and move forward
$$\vec{s_{i+1}} = (x_i + \alpha \cos \varphi_i, y_i + \alpha \sin \varphi_i, \varphi_i - \frac{2\pi}{n})$$
3. Rotate right by $\frac{2\pi}{n}$ and move forward
$$\vec{s_{i+1}} = (x_i + \alpha \cos \varphi_i, y_i + \alpha \sin \varphi_i, \varphi_i + \frac{2\pi}{n})$$

Figure 3: Next possible robot states

Where α is the step size (to move 1 pixel forward, $\alpha = 1.5$). Each state is only possible if the map does not contain a wall at pixel (x_{i+1}, y_{i+1}). Because the orientations are cyclical, if the robot orientation is less than 0 or more than 2π, the orientation is wrapped around to be inside $[0, 2\pi]$ range.

The heuristic function can be defined as an Euclidean distance between two states. $h(s_i, s_{i+1}) = \sqrt{(x_{i+1} - x_i)^2 + (y_{i+1} - y_i)^2 + (\varphi_{i+1} - \varphi_i)^2}$. To avoid getting too close to the walls, we can add penalty to the pixels that are too close to the wall.

The `A*` algorithm can now be run using states as defined above. We start with initial robot state (x_0, y_0, φ_0) and set the goal to the ring position (x_m, y_m, φ_m). The closest path between those two states is the path that the robot should follow to get from its current position to the goal position including its orientations.

After the path is computed, the robot can subscribe to odometry data to get its position and follow the list of poses that the previous algorithm has computed. To move from one pose to another, we wait until next odometry message is received, then rotate until we are oriented in the direction to the next goal and then move until we are close enough to the next pose.

3 Implementation and integration

4 Results

5 Division of work

6 Conclusion