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 the positions of the detected objects in the world. 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: Inacurate detection of the object's position due to lack of time synchronization among depth and colour image

Figure 2: 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 the width and height of the camera sensor.

With this information the robot can "shoot" the ray from the centre of the camera coordinate system to the pixel on image plane, where the object was detected. Using depth image information, we can obtain the distance from the centre of the camera to the detected object in space. Using the computed ray and the distance to the object, we can compute object's position in a three-dimensional space in camera frame.

After that, we can use a transforamtion matrix to finally convert the position of the detected object from the camera frame to the world coordinate system.

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

2.2 Faces

In this section, we present algorithms that our robot uses 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. We chose this algorithm because it can be run in real-time and it has a high detection rate.

2.2.2 Computing the orientation of the faces

After the face has been detected in an image, it's position in the world coordinate system is computed. The approaching point is then calculated using static map information. `Approaching point` is a point close to the face and directly in front it that the robot must visit to approach the face.

To compute the `approaching point`, we must first find the orientation of the detected face (from now on `face orientation`). In order to do that, we need the static map information. The calculated position of the detected face (from now on `face position`) in the world coordinates is first converted to map coordinates. Then, the corresponding pixel in map is calculated.

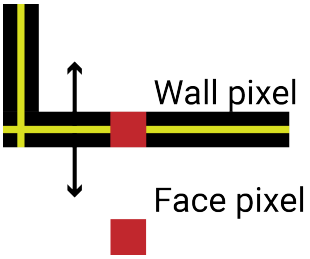

Figure 3: Face orientation computation

Because of depth sensor inaccuracy, the computed map pixel coordinate doesn't always lie on the wall. In figure 3, computed face pixel is coloured red.

Using a simple breadth first search in the proximity of the pixel which corresponds to the position of the detected face (from now on `face pixel`), the closest pixel from the set of pixels, which correspond to the positions occupied by the walls (from now on `wall pixel`), is determined. In figure 3, this pixel is also coloured red.

After the wall pixel has been found, the Hough line finding algorithm is executed on an area around it. The detected lines in figure 3 are represented with a yellow colour. If the algorithm finds more than one line, the line which is closest to the wall pixel is selected. After the line is selected, there are two possible orientations as represented in figure 3. 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 the orientation of the rings (from now on `ring orientation`).

2.2.3 Face classification

After faces are detected and localized in the world, they have to be identified.

2.3 Colour classification

The robot must be able to detect the colour of the rings and cylinders. 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 with one of the six possible classes listed above.

Because of the uneven lighting in the Gazebo simulator (and probably in the real world too), the classifier sometimes returns an incorrect label. We solved this problem by using the robustification process as described in the 2.6 section. Colour classifier is run multiple times on different detections of the same object and the most frequent colour is chosen as the colour of the object (from now on `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 sensors are not 100% accurate, so we have to take into account some deviation. For example, when an object is detected, the depth sensor might comput the distance inaccurately, or the robot may not have been localized. Detectors might return a false positives and so on.

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

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

1. Detected object was detected at least `threshold` times
2. There is at least a `minimum distance` meters distance between this detection and every other previous detection.

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

2.7 Smart exploration of space

2.8 Fine movement

Fine movement is a special way in which our robot can move. It is used when the normal movement wouldn't be able to get us to the goal even though the goal is reachable.

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. The robot builds a costmap of its environment, where the closer the position is to the wall, the highest cost it has. This way a robot can avoid obstacles and plan the path from point A to point B. The robot builds a costmap in a way that it makes sure not to bump into anything, which means it also takes into account all possible errors in data from its detectors. This means that some of the points that the robot can reach in reality unfortunately fall into the high-cost area and so the normal movement cannot get us there because the risk of bumping into something is too high.

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[1] 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.

[1] Does this make us flat-earthers? Nah. Does this make our robot a flat-earther? Maybe.

Figure 4: All possible orientations a robot can have if n=8 (blue/left) and if n=16 (red/right)

The algorithm will use map to access environment information, so the x and y represent pixel coordinate in map coordinate system.

Figure 5: Next possible robot states

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

$$\vec{s_{i+1}} = (x_i + \alpha \cos \varphi_i, y_i + \alpha \sin \varphi_i, \varphi_i + \frac{2\pi}{n})$$

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 current state and the end state.

$$h(s_i) = \sqrt{(x_{end} - x_i)^2 + (y_{end} - y_i)^2 + (\varphi_{end} - \varphi_i)^2}$$

. To avoid getting too close to the walls, a penalty is added 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

3.1 Object detectors

There are three different nodes responsible for object detections. Face and ring detectors are using `rgb` and depth images and the cylinder detector is using point cloud and `rgb` image to detect object in our world.

All three object detectors are publishing a custom `ObjectDetection` message type. The `Robustifier` component subscribes to this type of message and tries to minimize the number of false positives.