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

Object detections are performed on images. Object detectors output pixels from `rgb` image (or in case of ring detector from depth image) where the object is detected. Using camera calibration matrix, focal length and image format can be obtained. 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.

Figure 2: Pinhole camera model

Assuming that the camera is using pinhole camera model (and the camera calibration matrix is known), a ray can be "cast" from the centre of the camera coordinate system to the pixel on image plane, where the object was detected. This can be seen in the figure 2. 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 Haar cascade face detection algorithm. We chose this algorithm because it can be run in real-time and it has a high detection rate.

Figure 3: A few examples of haar features

The face images are first cropped to the same size and aligned so that the eyes approximately match. Then, a set of Haar features is generated. Each feature is defined at certain position in the face rectangle and consists of black and white regions (see figure 3). The value of the feature is computed as as difference between the sum of pixel intensities in the dark regions and the sum of pixel intensities in the light regions.

Then, for each feature, a simple and fast binary classifier is trained on the training data that can recognize if it is looking at a face or not. Not all features prove to be very good at this task, so the features are ranked by their classification accuracy. With a boosting machine learning algorithm, many weak classifiers (as described above) are used to improve face detection accuracy. A cascade of weak detectors is created such that we start with the most accurate classifier and continue with less accurate ones. This gives the haar cascade algorithm its real-time ability. For some regions, the first few classifiers detect that there is definitely not a face and the detection can stop. This way, the most processing power is given to the areas that most likely contain a face.

The detection is then done with a sliding window method. This simply means that we are evaluating every possible rectangle area in the image. If we want to detect faces of different scales, the image must be resized and the entire algorithm is repeated. So the cascade is crucial for speed here.

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.

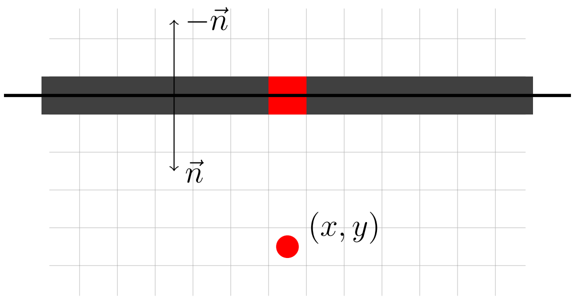


Figure 4: Face orientation computation

Because of depth sensor inaccuracy, the computed map pixel coordinate doesn't always lie on the wall. In figure 4, 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 4, 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 4 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 4. 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

When robot finds the woman that is willing to marry Gargamel, it must help him find a ring that she will like. Luckily, she is kind enough to tell us her favourite colour. The robot must then find the ring in that colour, give it to her and ask her to marry him.

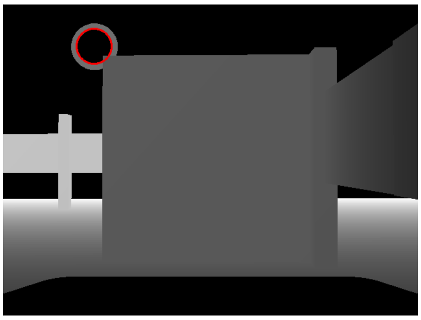
But before the ring can be picked up, it must first be detected by the robot. The ring detection is performed on the depth image and the ring colour detection is performed on the `rgb` image. Again, both images must be synchronized, so it can be accurately localized.

Depth image is computed from disparity image, which is calculated from Kinect stereo camera system. The further away an object is, the smaller its disparity will be. Using calibration matrix, depth of each pixel can be approximated from disparity image. Object that appear further away from the camera have a smaller disparity, which reduces the accuracy of depth computation. To combat this, our algorithm first removes all objects that are over a certain distance away from the camera.

After inaccurate depths are removed, a blob detection algorithm is applied to our depth image. The algorithm finds regions in our image that differ in colour. It actually searches for dark areas (areas that are further away from the camera) in the depth image. Because the inside of the ring is darker than the ring itself, the inner part is considered a blob. But not all blobs are considered rings. We can then apply some domain knowledge to the problem. We know that all the rings are positioned 12 cm above the cubes, so we can filter all blobs that lie below that height. If we look at the ring from any angle, they look elliptical, so the blobs are filtered by their roundness too.

In figure 5a, we can see an example of detected ring. Even though the corner of the cube is overlapping with the circle, the ring is still detected. This also presents another problem. To localize the ring, we must know the distance to the ring, but which distance should we use?

To compute the distance from camera to the ring, a histogram of distances is created. In figure 6, we can see an example histogram for the figure



(a) Ring detection

(b) Cropped ring detection

Figure 5: Ring detection using blob detector

5b with 24 bins. As we can see, there are two bars that stand out. One of them is the edge of the cube and the other is the ring. A mask is constructed so that only distances that lie in the highest bucket are retained, everything else is removed.

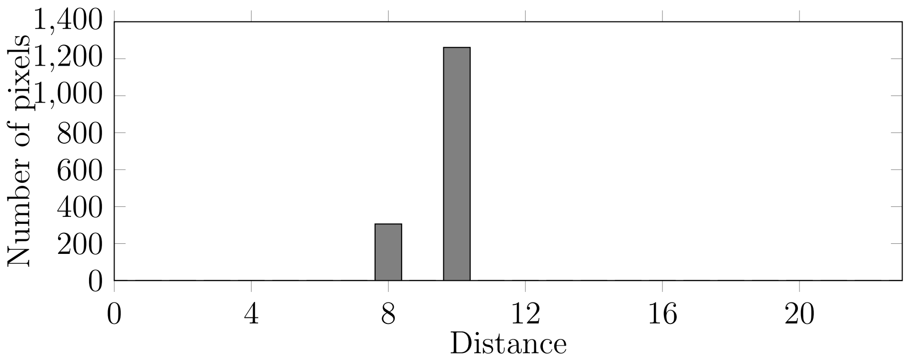


Figure 6: Distance histogram

The mask is then applied to both, the distance image and the colour image. In figure 7, we can see the result of filtering of the `rgb` colour image using the mask obtained in the previous step.

The distance to the ring is computed as the average of distances to each pixel in the masked distance image. The ring position in world frame is computed using as explained in section 2.1.

2.4.2 Ring colour detection

The ring colour is computed using the mask that we have computed in section 2.4.1. The `rgb` image is filtered so that only ring is left and then the colour is averaged. The average colour is then classified using the algorithm explained in section 2.3.

Figure 7: Masked colour ring

2.5 Cylinder detection

Cylinder detection runs on a point cloud data that is computed from depth information.

2.5.1 Removal of planes

2.5.2 Cylinder detection

2.5.3 Approaching point computation

The cylinder approaching point computation is not as difficult as the one for faces. If cylinder has been detected from current robot position, that means that the robot must have a clear view of the cylinder. Because of that, the approaching point can be computed as a point on the line connecting the robot and the cylinder.

A point that lies 0.5 metres from the centroid of the cylinder on this line is selected as the approaching point. If the point can't be reached (that is, if it is too close to the wall), approaching point is repositioned so that there is enough space for the robot to visit the point. This is done by finding the closest wall on the map and then moving the point in the opposite direction. This is repeated until there is no wall too close to the point.

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 default movement component is unable to reach the goal. To do that, odometry data must be used and 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.

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

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

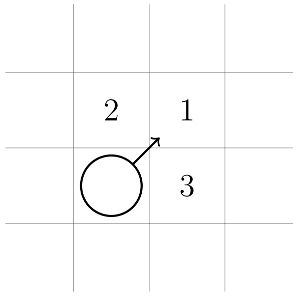


Figure 8: Next possible robot states

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 figure 8 represents a robot at some coordinates (x, y, φ). The robot is oriented in the direction of the arrow. In the example, the orientation is $\varphi = \frac{\pi}{4}$. The next possible states are numbered as defined above with numbers 1, 2 and 3 respectively. In this case, the number of possible orientations $n = 8$.

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

In this section, the implementation of the algorithms explained in section 2 will be discussed. To solve our task, the following architecture was designed. We tried to keep the entire project as modular as possible, not only because this is the `ROS` operating system's philosophy, but also because three different people needed to work on the same project at the same time.

This proved to be very difficult because nodes are so interconnected, that if any part of the system didn't work as intended, it seemed as if nothing worked. Some parts of the system were discussed beforehand, but as the tasks became more difficult, the arhitecture had to be changed in order to solve them.

Figure 9: Final architecture

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.

3.1.1 ObjectDetection message

The object detection message contains the following information

1. `Header header`, which contains the timestamp of the detection and other metadata
2. `string type`, which represents the type of the detected object (it can be either `"face"`, `"ring"` or `"cylinder"`)
3. `Pose object_pose`, which represents the object position in world coordinate frame
4. `Pose approaching_point_pose`, which is a point that the robot has to visit in order to complete the task
5. `Image image`, which is an image of the object
6. `ColorRGBA color` color and `string classified_color` which represent the object average colour in `RGBA` format and the colour label that the colour classifier computed

3.1.2 Face detector node

The face detector is responsible for detecting and localizing faces. To do that, it fist synchronizes received `rgb` and depth images using the `Time Synchronizer` class in `message_filters` package.

After both, the `rgb` and depth images are received, it uses the Haar cascade algorithm explained in section 2.2.1 to detect faces. If face has been found, it tries to compute its world position. This is done using raycasting from camera center, through the face center pixel in the image plane as explained in section 2.1.

After face is localized, `Approaching point` is computed. `Approaching point` is a point that is positioned directly in front of the face. Because the faces are positioned on the walls, the closest wall to the face is found. Then, the wall normal is computed as explained in section 2.2.2. The `Approaching point` is positioned 0.5 metres from the face position in the direction of the face orientation.

3.1.3 Ring detector node

Ring detector is a node that is responsible for ring detection and localization. It works similarly to the face detector node. First, it synchronizes depth and `rgb` images. Then it uses the ring detection algorithm as explained in section 2.4.1. Ring localization is done using the algorithm explained in section 2.1.

Ring orientation is determined using similar algorithm as explained in section 2.2.2, but instead of finding the normal of the wall, its direction is chosen as ring orientation.

3.1.4 Cylinder detector node

3.2 Robustifier

For each object detector, a new `Robustifier` node is created. So each `Robustifier` node is listening for different object detections. The detections are then grouped together using the algorithm as explained in section 2.6. When number of detections sent to the `Robustifier` node exceeds the `threshold` value, the detection is considered a true positive and is sent to the `Brain` node.

3.3 Brain

`Brain` is a node that is responsible for solving the main part of the task. It subscribes to robustified object detections and uses them to solve the task.

3.4 Movement controller

3.5 Greeter

4 Results

5 Division of work

6 Conclusion