Univerza *v Ljubljani*
Fakulteta *za računalništvo*
in informatiko

Development of Intelligent Systems

Final Report

Team TAU
Julija Klavžar, Rok Mokotar, Jan Zorko

Ljubljana, 2022

Contents

1 Introduction

In this report we describe the methods, implementation and integration and the results achieved solving Task 3 in the course Development of Intelligent Systems in the academic year 2021/2022. We also state how the work was divided and give some final remarks on our work at the end of the report.

Task 3 includes designing and implementing a system in ROS that completes the following task successfully in a simulator: the robot should autonomously explore the given fenced area. While exploring, the robot should:

- find all persons in this area and recognize them (the faces are attached to the inner walls),
- find all restaurants (different coloured cylinders),
- recognize the food each of these restaurants serve (a circular image on top of the cylinder),
- find a green ring that marks a parking spot.

After finding all of these and saving their positions, the robot should go to the parking spot and accept orders from the link given in the QR code on the wall next to the parking spot. The robot should then plan the delivery of each of the orders, i.e. go to the restaurant that serves the type of food in the order and then go to the person in the order. When delivering the order to the persons, the robot should have a short dialogue with them.

2 Methods

2.1 Autonomous exploration

The robot performs autonomous exploration using K-means clustering method. For goal generation, the implementation uses a .pgm image of the map and transforms in to a binary map, keeping only the bits that are part of the fenced area. It then fits a certain amount of clusters to this area. After obtaining these cluster centers, it moves them into the navigation frame. These are then goals for autonomous exploration. The details of the implementation are in 3.1.

Clustering is a set of techniques used to divide data into groups called clusters. K-means clustering in an unsupervised machine learning algorithm, i.e. it works with unlabelled data like our input map. It aggregates data points into clusters around a certain number of centroids. The number of centroids it the algorithms' input parameter. The algorithm works in the following way: it starts off with a group of randomly selected centroids and iteratively optimizes the positions of the centroids. In each iteration a data point is assigned to its nearest centroid. We then calculate the mean of all points for each cluster. This is a new centroid for the next iteration. We continue until we have reached a certain number of iterations or there was no further change in the centroid locations.

2.2 Cylinder detection

Cylinder detection in our implementation uses data from the robot scanner. Using this data, we constructed a binary image of an edge map showing the birds-eye view of the space that the laser scan scanned. This image is then analysed to detect circles using the Hough Transform. If a circle is found, we add a marker to the map to show the cylinders location and remember its position. The details of the implementation are in 3.2.

Hough Transform is a feature extraction technique used in image analysis, computer vision, and digital image processing. The Hough Transform was introduced to address the problem of imperfect image data. It was introduced to be able to recognize even incomplete shapes (most commonly lines, circles or ellipses). It identifies different structures using their given parametric equation. The input for the Hough Transform must be a edge image.

The Hough transform for line detection works in the following way: we are given some edge points that we are trying to fit to two equations $y = mx + c$ (eq. 1) and $c = -mx + y$ (eq. 2). For a select point (x_i, y_i) the right-hand side of the second equation is known, which allows us to look at the problem in two spaces: the image space (eq. 1) with parameters x_i and y_i and the parameter space (eq. 2) with parameters c and m. The parameter space represents all lines that pass through our selected point. When we look at multiple points, we see that in the parameter space the lines of the points intersect at some point, since these values of m and c represent the equation of the straight line all of these points are on. To find a line, we construct an accumulator array. In the accumulator array we look at the parameter space of each separate point and increment the cells that the lines in the parameter space pass through. We then get the maximum, which tells us the equation of the line. Or in the case of multiple lines, we use some peak extraction algorithm to find multiple lines.

For circle detection we use the equation $(x_i - a)^2 + (y_i - b)^2 = r^2$, where x_i and y_i are parameters in the image space, and a and b are parameters in the parameter space, assuming that r is known. If r is not known, we get a 3D parameter space. If we do more work in image space, the work is easier in the parameter space and vice versa. For the Hough Gradient method we are given not only the edge point location (x_i, y_i), but also the edge direction ϕ_i. Because of this, we require some additional information about our circle. We know that the edge direction should point towards (or from) the center of the circle and that it should lie on a distance r from the point. This means that in parameter space we do not need to look at the whole circle of possible parameters, but only at two point in the direction ϕ_i at distance r from the given point.

For the implementation we used the OpenCV library which actually uses the 2-1 Hough Transform [1] to find the circles in an image.

2.3 Ring detection

Ring detection in our implementation uses the image data collected from the depth camera, where differences in the distance of the ring and the background around it are shown as a large difference in intensity on the depth image.

An additional required step before performing the circle detection is edge detection. That was not needed when detecting cylinders, as the laser data came in the form of a thin line, while here we must look for circles on an image with various shapes and sizes. We used the Canny edge detector, as that came integrated with the Hough transform in OpenCV.

The Canny edge detector uses intensity gradients to determine areas where a large difference in intensity occurs.

To reduce the influence of noise a Gaussian kernel is first used to first slightly blur the image, upon which the intensity gradients are computed using the derivative of a Gaussian kernel. The gradients are computed in both the vertical and horizontal direction, allowing the detection of edges in any orientation. The gradient values are then thresholded, which removes areas without high enough differences in intensity. Finally, hysteresis is used to further remove weaker edges that are not connected to strong edges.

We then perform the Hough transform on the detected edges, again looking for circles, just as described in 2.2.

Finally we filter out any detections of circles that are not hollow. This is simply done by checking the depth value at the center of any detected circle, and discarding the circle if the value is not above a certain distance threshold.

2.4 Color detection

For cylinder color detection we look at the hue of an image after transforming it from RGV to HSV. The image for this is an RGB image from the arm scanner, from which we cut a part of the detected cylinder. HSV is an alternative representation of the RGB color wheel and has attributes hue, saturation and lightness. We use this alternative colorspace because it is more robust in color detection when it comes to lightning changes, which is very important in our implementation. To determine the color of the object/image, we use the hue attribute which has a range from 0 to 360. When we have acquired the average hue value of the cut-out image of the object, we check whether the hue value in within certain values to detect the color.

For ring color detection we use KMeans to find some number of most common colours in the image. We do this on an image in which the found circle has been cropped out. The KMeans method has been described in the section about autonomous navigation. Since we only have some number of color names, we have to approximate the color we extract. We do this by building a KDTree to find an approximation to the color we got, i.e. we use it to look up nearest neighbours to a point (in our case in the RGB colorspace). A KD-tree is a binary tree and each node represent an axis-aligned hyperrectangle. The node splits a set of points based on whether the coordinate in this particular axis is greater or less that some particular value.

2.5 Face detection and recognition

For face detection, we used the Python dlib library function `get_frontal_face_detector` which returns a pre-trained HOG + Linear SVM face detector. To detect and localize faces we use RGB images and depth images from the robot camera.

Histogram of Oriented Gradient (HOG) is a feature descriptor that is used in computer vision and image processing for object detection. A feature descriptor is a representation of an image that simplifies the image by extracting only useful information. In the HOG feature descriptor, the distribution (histograms) of directions of gradients are used as features. The magnitude of gradient is larger around edges and corners, which holds more information about objects than flat regions. The general procedure of the HOG descriptor is the following: (optional) preprocessing, gradient computation by applying a derivative mask in the horisontal and vertical direction, orientation binding (creating the cell histograms), generate

descriptor blocks (group cells into larger blocks) and normalize blocks. Features generated by HOG are fed to a Linear SVM model.

Support Vector Machine (SVM) is a linear model for classification and regression problems. The basic idea of SVM is that it separates the given data into classes with lines or hyperplanes. Linear SVM is used for linearly separable data, i. e. for datasets that can be classified into two classes by using a single straight line. Given a training set, the goal of linear SVM is to find a maximum-margin hyperplane that divides the points into two groups - positive and negative instances.

For facial recognition we compare the RBG image from the robot camera to a set of images of faces that were given in advance for this task. These were encoded with a (128,1) dimension encoding. The received image of a detected face is then compared to the encoded faces using the Euclidean distance measure, which tells us how similar the faces are. The smaller the distance, the more similar they are. After comparing the new face image with all of the given faces, we recognize it as the one that has the smallest Euclidean distance to the face from the robot camera.

2.6 Food recognition

For food recognition we used a pre-trained model, on which we only tweak the last layer to adjust it to our problem, i.e. to recognize the five possible different foods for our task - baklava, cake, pizza, fries and salad. That is, we only adjust the classification part of the neural network, but keep the feature extraction. This is done because we do not have enough data to train a model from scratch. While training, we keep track of the model accuracy to know how well our model is improving with each epoch. The pre-trained model is a SqueezeNet model. This is a convolutional neural network. We optimize the last classification layers using stochastic gradient descent.

Convolutional neural networks are neural networks that are appropriate for usage with images, speech and audio signal inputs. They are comprised of three main types of layers: the convolutional layer, pooling layer and fully-connected layer. The convolutional layers apply filters (kernels) to parts of the input and slide over all parts the input thereby convolving the image and creating an activation map. In the case of images with multiple channels, the kernel has the same depth as the input image. The each added convolutional layer, we detect more and more complex high-level features in the input. The pooling layers perform downsampling. This is done to decrease the computational power required to process the data. The fully-connected layer then performs the classification of the input on the high-level features generated.

The training and testing programs were taken from the 9th exercises. The testing program was incorporated into into node `food_detection` which positions the robot arm above the cylinder and waits for an RGB image from the robot arm camera. To determine the position of the food image on the camera, we fit ellipses to the contours of the image, looking for the top of the cylinder. A bounding box around fitted ellipses is then used as the input to the previously trained food classification model.

2.7 Parking

If a green circle has been found, we save its coordinates for later parking. When we have discovered all rings, cylinders, and faces, we proceed to park. We send a goal of the location of the green ring to the robot, when this goal has been reached, the robot starts parking. We wait for RBG images from the arm camera, from where we try to find a red circle in it with the CV2 function `HoughCircles`. The Hough transform was previously described in 2.2. When the robot has positioned itself appropriately close to the position of the red circle, we scan a QR code.

2.8 Speech recognition

Speech recognition is done using the Python library Speech Recognition with the Google Web Speech API.

3 Implementation and integration

3.1 Autonomous exploration

As described in section 2.1, we used the K-means clustering method to find a number of cluster centres in the map of the fenced area. For this we used the `KMeans` method from the Python module `sklearn.cluster`, where we used the input parameter `n_clusters`, which we after some experimentation set to 8. That is, we instructed the method to find 8 cluster centers in our binary map. Using the `KMeans` method we construct a `KMeans` object with the given parameters.

On this object we then call function `fit` using a list of coordinates of the traversible area of our map as the parameter. The `fit` function returns a fitter estimator object from which we extract the coordinates of the cluster centers using the attribute `cluster_centers_`. This procedure is shown in the code below. The coordinates of the traversible area are acquired by first loading the .pgm image of our map with the `cv2` function `imread`.

```
# Get map from image
map = cv2.imread(os.path.relpath("..."), cv2.IMREAD_GRAYSCALE)

# Treshold drivable area
drivable_area = np.where(map >= 254, 1, 0)

# Get valid coordinates
drivable_coordinates =  list(zip(*np.nonzero(drivable_area)))

# Execute clustering on valid coordinates
clusters = KMeans(n_clusters=8).fit(drivable_coordinates)
goals = clusters.cluster_centers_
```

After acquiring the cluster centers of the map, we still have to move this into the navigation frame. To do this, we have to acquire some map metadata about the map. To do this we use the `rospy` function `wait_for_message` which subscribes to a topic, wait to receive exactly one message and the unsubscribes. We subscribe to the topic `/map` on which the `map_server` node publishes the message type `OccupancyGrid` from `nav_msgs`.

To transform the coordinates of our goals to the navigation frame, we use the `info` attribute of the message, and the attributes `origin.position` and `resolution` of `info`. The first holds the coordinates of the origin, while the other is a measure of $m/cell$ of the map. We then adjust the x and y coordinates of the goals using these attributes as shown below.

```
goals[:, [1, 0]] = goals[:, [0, 1]]
goals[:, 0] = goals[:, 0] * resolution + position.x
goals[:, 1] = (map.shape[1] - goals[:, 1]) * resolution + position.y
```

After this, we just order the generated goals by distance. The first goal is the closest goal the robots starting position. We then go through all the remaining goals and add them one by one according to distance from the previously chosen goal. We do this by initializing a service proxy to the service `make_plan` which allows an external user to ask for a plan to a given pose from `move_base` without causing `move_base` to execute that plan. The service uses the message `nav_msgs/GetPlan`. The return type is `nav_msgs/Path` which returns an array of stamped poses. We then look at the length of this array to determine the length of the path between two goals.

```
# Initialize service
get_plan = rospy.ServiceProxy("/move_base/make_plan", GetPlan)

# Initialize both goals as type PoseStamped
...

# Initialize GetPlan request
req = GetPlan()
req.start = start
req.goal = end
req.tolerance = 0.5

path = get_plan(req.start, req.goal, req.tolerance)

# Get plan length
path_len = len(path.plan.poses)

# Compare to current smallest
if path_len < min_len:
    ...
```

3.2 Cylinder detection

Cylinder detection in our implementation uses data from the manipulator arm laser scanner. We created a node named `cylinder_detection` that is subscribed to the `/arm_scan` topic which publishes the `LaserScan` message from `sensor_msgs`. When a message is received, we use the `ranges`, `angle_min` and `angle_increment` attributes to convert the data in polar coordinates to more useful Cartesian coordinates, which represent a top-down view of the laser scan, as you'd see it in Rviz.

The coordinate transformation is demonstrated below:

```
# Prepare an empty image
image = np.zeros((self.N, self.N), np.uint8)

# Determine the starting angle
angle = scan.angle_min
# Iterate over range values
for r in scan.ranges:
    # Each value is for one angle increment
    angle += scan.angle_increment

    # Ignore values outside the defined range
    if np.isnan(r):
        continue

    # Compute Cartesian cooridnates from the angle and distance
    x = r * np.cos(angle)
    y = r * np.sin(angle)

    # Round the results to pixels
    x_p, y_p = self.to_pixel_space(x, y)

    # Check if pixels within image
    if x_p < 0 or x_p >= self.N or y_p < 0 or y_p >= self.N:
        continue

    # Write the value to output image
    image[y_p, x_p] = 255
```

We can use these new points as an edge on which we perform the Hough transform and detect circles. Due to to Hough transforms robustness to partial occlusion, it is enough to see the cylinder from only

one side to detect it. Due to the arena being designed in such a way where there are no circular edges that are not cylinders, we were able to tweak the parameters to ensure that only cylinders were detected.

```
circles = cv2.HoughCircles(
    image,
    cv2.HOUGH_GRADIENT,
    2,
    100,
    param1=10,
    param2=25,
    minRadius=30,
    maxRadius=35,
)
```

We use `tf` to convert these coordinates into world coordinates so we know where to place the marker.

To detect the colour, we must reverse our computations to get the location of the cylinder in image coordinates. We do that by computing the polar angle of the coordinates where we detected the circle, and interpolating it into the image. We are only interested in the horizontal component, as the laser scan is taken from the middle of the image and we then know that the cylinder will be vertically centered as well.

```
# Compute the polar angle of detection
angle = np.arctan2(y, x)

# Get the colour image
image = self.bridge.imgmsg_to_cv2(arm_image, "rgb8")

# Get the cylinder x coordinate by interpolation
x_cyl = np.interp(
    -angle,
    [scan.angle_min, scan.angle_max],
    [0, image.shape[1]],
)
```

Once we have the image coordinates of the cylinder we perform colour detection as we do elsewhere.

3.3 Ring detection

Ring detection in our implementation is based on the depth camera image to detect the position of rings and then the colour camera to detect their colour.

We created a node named `ring_detection` that is subscribed to topics `/camera/depth/image_raw` and `/camera/rgb/image_raw`, which both publish the `Image` message from `sensor_msgs`.

We convert these images to the `cv` image format so we can use it's functions.

The main part of the detection is again the Hough transform. The `cv::HoughCircles` function implements it's own Canny edge detector, for which we only provide the parameters. We determined the parameters empirically, by testing what worked best. The output of this function are center coordinates and diameters of detected circles.

```
// prepare arguments for circle detection
int cannyTreshold = 100;
int accTreshold = 75;
int centerTreshold = 50;
int minRadius = 10;
int maxRadius = 100;
int accResolution = 2;
int minDist = 1;

// perform detection
cv::HoughCircles(input, all_circles, cv::HOUGH_GRADIENT, accResolution,
    minDist, cannyTreshold, accTreshold, minRadius, maxRadius);
```

We then iterate all the detections and filter out misses. We are only interested in hollow circles, so we filter out circles whose center points on the depth image are not far enough away.

```
cv::Point center(cvRound(all_circles[i][0]), cvRound(all_circles[i][1]));

// ignore circle if not hollow
int center_color = input.at<uchar>(center);
if (center_color > centerTreshold) {
    ROS_INFO("Ignoring circle with center value: %d", center_color);
    continue;
}
```

These detected circles are then sent to the colour classification node that determines which colour they are.

3.4 Color detection

3.4.1 Cylinder color detection

Cylinder color detection is performed in the node `cylinder_detection`. Once we have cut-out the cylinder in the RGB image, as described in 3.2, we transform the image to HSV colorspace using the CV2 function `cvtColor` and the transformation `COLOR_BGR2HSV`. As this returns hue in the range 0..180, we multiply the hue values by 2 to get the standard range 0..360. We then calculate the mean hue of the area. Then we check to see if the hue is within certain ranges, thereby recognizing different colors of the cylinder. This is done as shown below:

```
hsvImage = cv2.cvtColor(image, cv2.COLOR_BGR2HSV)
hue = np.mean(hsvImage[0] * 2)

color = ""

if hue < 30 or hue > 330:
    color = "red"
elif 30 < hue < 80:
    color = "yellow"
elif 80 < hue < 150:
    color = "green"
elif 160 < hue < 240:
    color = "blue"
```

3.4.2 Ring color detection

For ring color detection we initialize a service node that works with the following message:

```
float32[] circle
sensor_msgs/Image image
---
string color
```

We sent the RGB image to the service, along with the coordinates and radius of the circle. We return the detected color. First we process the image by putting a white mask over it around the detected circle (with some padding). Then we find the most common colours with the KMeans function. We find three centers, that should be white, gray and some other color. We then return the color that is not white or gray. The library Webcolors has only a certain number of defined colors, but more easily recognizes the color black that if we were to try to find it with a transformation to the HSV color space like in cylinder color detection. To recognize the approximate colors, we use a dictionary of hex values of colors and their names. Then we create a KDTree from scipy.spatial library. Once we have the tree and the RGB color values we use the `query` function to find the closest color. Excerpts of this are shown in the code below:

```
class ColorRecognizer:
# Initialize hex to name dictionary
css3_db = webcolors.CSS3_HEX_TO_NAMES
self.names = []
self.rgb_values = []
for color_hex, color_name in css3_db.items():
  self.names.append(color_name)
  self.rgb_values.append(webcolors.hex_to_rgb(color_hex))

# Build KD tree
self.tree = KDTree(self.rgb_values)

...

# Crop out circle in image
mask = np.zeros(image.shape, dtype=np.uint8)
mask = cv2.circle(mask, (round(outerCircle[0]), round(outerCircle[1])), round(
    outerCircle[2]) + 10, (255,255,255), -1)

# Mask input image with binary mask
result = cv2.bitwise_and(image, mask)
# Color background white
result[mask==0] = 255

# Turn image into vector
result = result.reshape((image.shape[0] * image.shape[1], 3))

# Perform KMeans
KM_cluster = KMeans(n_clusters=3).fit(result)

# Get colors of the clusters
for cluster in KM_cluster.cluster_centers_:
  distance, index = self.tree.query((int(round(cluster[0])), int(round(cluster
    [1])), int(round(cluster[2]))))
  color = self.names[index]

  if color != "white" and "gray" not in color:
    return ring_colorResponse(color)
```

3.5 Face detection and recognition

3.5.1 Face detection

For face detection we used the script `face_localizer_dlib.py` from the 4th exercises as a base for further work. The program initializes the node `face_localizer`. Then the classifier is initialized using the dlib library function `get_frontal_face_detector` which returns a pre-trained HOG - Linear SVM face detector.

```
class FaceLocalizer:
    def __init__(self):

        rospy.init_node('face_localizer', anonymous=True)

        # An object for converting images between ROS and OpenCV format
        self.bridge = CvBridge()

        self.face_detector = dlib.get_frontal_face_detector()
```

The function `find_faces` that runs in the main loop of the node is is executed with a frequency of 10Hz. It expects messages from two topics: `/camera/rbg/image_raw` and `/camera/depth/image_raw` using rospy function `wait_for_message`. It then transforms both images to an OpenCV format using `imgmsg_to_cv2`

with encodings *bgr8* and *32FC1* respectively. The faces are then detected in the rgb image using the previously initialized face detectors as shown below.

```
def find_faces(self):
    # Get the next rgb and depth images that are posted from the camera
    try:
        rgb_image_message = rospy.wait_for_message(
            "/camera/rgb/image_raw", Image)
    except Exception as e:
        print(e)
        return 0

    # Convert the images into a OpenCV (numpy) format
    try:
        rgb_image = self.bridge.imgmsg_to_cv2(rgb_image_message, "bgr8")
    except CvBridgeError as e:
        print(e)

    # Similar procedure for depth image

    # Detect the faces in the image
    faces = self.face_detector(rgb_image, 0)

def main():
    face_finder = face_localizer()

    rate = rospy.Rate(1)
    while not rospy.is_shutdown():
        face_finder.find_faces()
        rate.sleep()

    cv2.destroyAllWindows()
```

For each detected face we then get its position using the previously acquired depth image using the function `get_pose` that was provided in the dlib implementation of face detection in exercises. The face is then added to the list of detected face locations, if it is at a distance of at least 0.5 a meter from all previously detected faces.

3.5.2 Face recognition

Face recognition was implemented with the Python library Face Recognition. We feed the extracted image region where the detected face is to the function `face_locations` which returns an array of bounding boxes of human faces in an image to confirm that we have in fact found a face. We know this when `face_locations` returns at least one hit.

After this we encode this extracted face region with the function `face_encodings`. The parameters of this function are the image that contains the face, the bounding boxes of the faces previously recognized with `face_locations`, the number of times to re-sample the face when calculating encoding and which model to use. The function returns a list of 128-dimensional face encodings (one for each face in the image) from which we take the first encoding.

```
# Returns the number of face encoding the model finds.
# If length = 0, then we found 0 faces, else we found atlest 1
# In our case that just means we definetly found a face.
boxes = face_recognition.face_locations(face_region)

encoding = face_recognition.face_encodings(
    face_region, boxes, num_jitters=3, model="large")

if len(encoding) == 0:
    print(
        f'{Colors.WARNING}> dLib found a face, but face_recognizer didn\'t
    recognize it')
    continue
else:
    encoding = encoding[0]

face = self.face_database.add_face(encoding, pose)

# Euclidan distances of comparisons -> Smallest, the closest to our face
lookup_comparison = face_recognition.face_distance(
    self.lookup_encodings, encoding)

index_min = np.argmin(lookup_comparison)

name = self.lookup_names[index_min]
```

When we have acquired the encoding, we compare it to the previously generated lookup table of encodings with the function `face_distance` from the Face Recognition library for provided faces for Task 3. We recognize the person as being the one out of the provided images, that has the smallest Euclidean distance between the encodings.

```
# Euclidan distances of comparisons -> Smallest, the closest to our face
lookup_comparison = face_recognition.face_distance(
    self.lookup_encodings, encoding)

index_min = np.argmin(lookup_comparison)

name = self.lookup_names[index_min]
```

3.6 Food recognition

We training and test programs for food recognition were implementation with the help of the work provided in 9th exercises. After the robot arm is set to the given joint positions, we wait for an image from the arm camera using the rospy function `wait_for_message`. We then transform it to an 8-bit BGR CV2 format. We then perform some additional transformations on the acquired image of food as shown below. After these transformations, we use the OpenCV function `findContours` to get the image contours. We then try to fit ellipses to these contours and take the smaller one to be our food place. We then call the function `recognize` which classifies the image with the code from program `test.py` from 9th exercises.

```
def find_food(self):
    self.set_arm([0, 0.3, 1, -0.5], 1)
    # Get image and transform to CV2
    ...
    img_hsv = cv2.cvtColor(image, cv2.COLOR_BGR2HSV)

    mask = cv2.inRange(img_hsv, (0, 0, 40), (0, 100, 150))
    output = cv2.bitwise_and(img_hsv, img_hsv, mask=mask)
    output[np.where((output == [0, 0, 0]).all(axis=-1))] = [255, 255, 255]

    # Transform image to grayscale
    gray = cv2.cvtColor(output, cv2.COLOR_BGR2GRAY)

    # Do histogram equlization
    img = cv2.equalizeHist(gray)
    # Binarize the image
    thresh = cv2.adaptiveThreshold(
        img, 255, cv2.ADAPTIVE_THRESH_MEAN_C, cv2.THRESH_BINARY, 15, 25
    )

    # Extract contours
    contours, _ = cv2.findContours(thresh, cv2.RETR_LIST, cv2.CHAIN_APPROX_NONE
    )
    # Fit ellipses to contours and cut out smaller ring
    ...

    food = self.recognize(path)
    return food

def recognize(self, image):
    input_size = 224
    data_transforms = { ... } # omitted for brevity
    class_dict = {
        0: "baklava", 1: "pizza", 2: "pomfri",
        3: "solata", 4: "torta"
    }

    model_path = "..."
    model = torch.load(model_path)
    model.eval()

    img_p = PIL.Image.open(image)
    img = data_transforms["train"](img_p).unsqueeze(0)

    pred = model(img)
    pred_np = pred.cpu().detach().numpy().squeeze()
    class_ind = np.argmax(pred_np)
    if pred_np[class_ind] > 6:
        return class_dict[class_ind]
```

3.7 QR code detection and getting orders from link

The implementation was largely adapted from the program `extract_qr.py` provided in the 8th exercises.

We initialize a node called `image_converter`. We subscribe to the robot camera RGB image topic and initialize a publisher for the topic `/qr/data` that publishes the QR code data. After receiving an image we transform the image into an 8-bit RBG OpenCV format. Using the pyzbar library function `decode` we get an array of objects of class `Decoded`. Each element of the array represents a detected barcode. We then initialize a QR decoder with constructor `QRCodeDetector` and use it to detect and decode QR codes with `detectAndDecode`. If the array returned by `decode` is of length or `detectAndDecode` has found a QR code, we publish this data with our publisher.

In our main node we then split this data and extract the name-food pairs and construct the goals for the destination pairs and then alternatively execute these goals. In the case of executing a goal which brings food to a person, we also have a short dialogue with the person, which is described in the next section.

Below are some excerpts of the code.

```
# Initialize publisher
self.qr_pub = rospy.Publisher("/qr/data", String, queue_size=1000)

...

# Detecting QR codes
decodedObjects = pyzbar.decode(cv_image)

decoder = cv2.QRCodeDeqtector()
data, points, _ = decoder.detectAndDecode(cv_image)

if len(decodedObjects) == 1:
    dObject = decodedObjects[0]
    self.qr_pub.publish(dObject.data.decode())
elif points is not None:
    self.qr_pub.publish(data)
elif len(decodedObjects) == 0 and points is None:
    print("No QR code in the image")
else:
    print("Found more than 1 QR code")
```

3.8 ASR

For ASR we used parts of the code in the program `extract_speech.py` provided in 8th exercises. The program was adjusted to that it acts as a service that accepts and returns string messages. The incoming message is the text for speech generation and the outgoing message is the recognized speech. The text is spoken with `sound_play`. Then we listen for the microphone input response using the Python Speech Recognition library function `listen`. We then recognize the spoken text with the function `recognize_google` which utilizes the Google Web Speech API. Excerpts of the code are below.

```
# Initialize needed objects
self.sr = sr.Recognizer()
self.mic = sr.Microphone()
self.soundhandle = SoundClient()
self.speech_pub = rospy.Service(
    "automated_speech_recognition", speech, self.speech_callback
)

...

# Say received text
request = req.data
self.soundhandle.say(
    request,
    self.voice,
    self.volume,
)
rospy.sleep(3)

# Listen for response
audio = self.sr.listen(self.mic)
recognized_text = ""
try:
    recognized_text = self.sr.recognize_google(audio)
except ...

# Return recognized text
...
```

3.9 Parking

We implemented the parking by using the arm camera and the CV2 function `HoughCircles` to detect a red circle in the resulting image, based on which we then orient ourselves and move towards it. We determined the robot's action based on where the red circle was found in the image. Possible actions were moving forward, rotating left or right, and moving backward.

Given that the circle was extremely close to the wall, we used `Twist` messages to navigate the robot in order to get closer to the circle. We performed this procedure continuously while the robot was in the parking state. Due to the excessive precision of the robot's parking, we limited the procedure to the `PARKING_NUM` number of iterations, as we came to the conclusion that the robot always approaches the wall well enough in a certain number of iterations to be appropriately parked and can clearly read the QR code on the wall in front.

```
# Get image and transform to CV2
...

# Detect the red circle
circle = cv2.HoughCircles(
    image=image,
    method=cv2.HOUGH_GRADIENT,
    dp=1,
    minDist=20,
    param1=30,
    param2=10,
    minRadius=0,
    maxRadius=25,
)[0][0]

# Create a Twist message
twist = Twist()
...

# Choose the appropriate action
if ...: # Turn rigth
    twist.angular.z = -1
elif ...: # Turn left
    twist.angular.z = 1
elif ...: # Move forward
    twist.linear.x = 0.15
else: # Move backward and scan QR code
    twist.linear.x = -0.50

# Publish Twist message
...
```

4 Results

In this section the experimental results and possible discarded methods are described, alongside a showcase of the robots performance in pictures.

4.1 Autonomous exploration

With autonomous exploration the largest point of experimentation was the parameter k in k-means, determining the number of points that are generated for the robot to follow around the map. We achieved the best results when $k = 8$, as less points caused the robot to follow non-optimal lines as far as detecting items is concerned, and more points could cause a jagged path that was not circular enough.

We quickly determined that moving in only one direction around the map would not be sufficient, as particularly some cylinders are hidden behind obstacles if only viewed from one direction. We thus opted for a two-way round trip, making the robot retrace its steps in the opposite direction if it did not detect all the items. For safety, we loop this infinitely until all the items are detected, that is the robot will drive in circles, each time in opposite direction, until it detects everything.

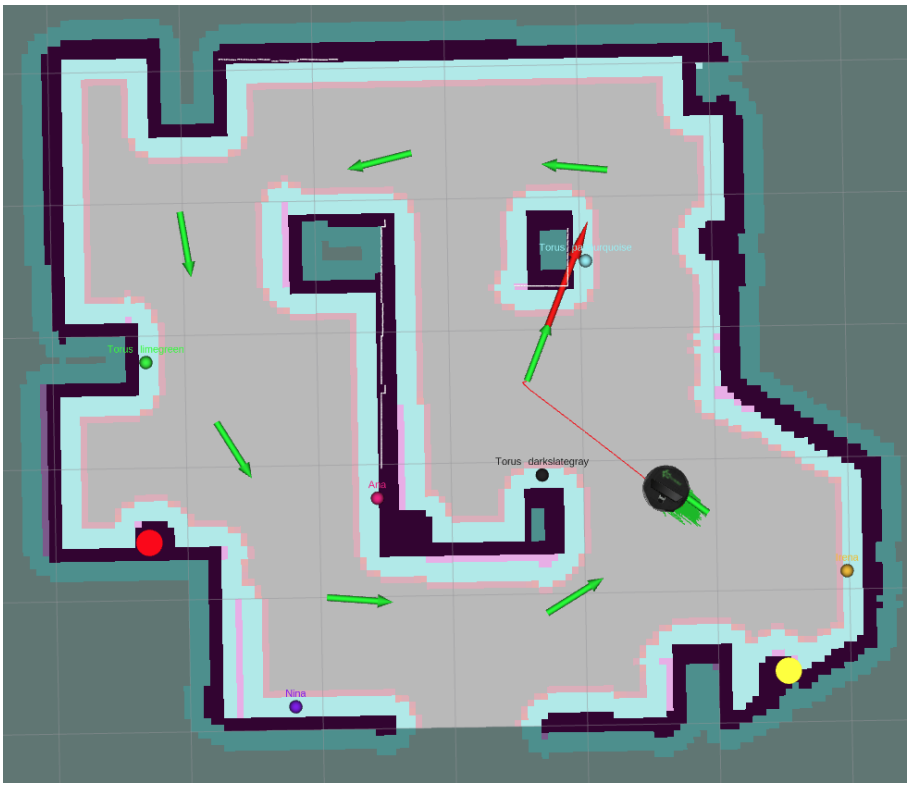

Figure 1: Map with exploration points.

4.2 Ring detection

With ring detection the only problems we encountered were determining what part of the colour image to take the colour from. Ring detection itself was very robust thanks to the large differences in depth between the ring and it's background, which caused a very nice inherent thresholding on the image. But due to slight errors in alignment between the depth and colour camera, alongside the time discrepancy between the two images, the ring did not occur in exactly the same position on both images.

We tried to take the middle between the detected inner and outer circle, but for far away circles the error was too large and the ring on the colour image was missed completely.

What worked best in the end was cropping out a slightly enlarged bounding rectangle, and essentially performing a histogram of colour values in that area to determine the most prevalent non-black colour, and then performing the final HSV transformation on that.

In the end the system was robust enough to consistently detect all the rings, with only very rare occurrences of mislabelling a colour of a ring.

An example of a visualised detection is shown on the following image:

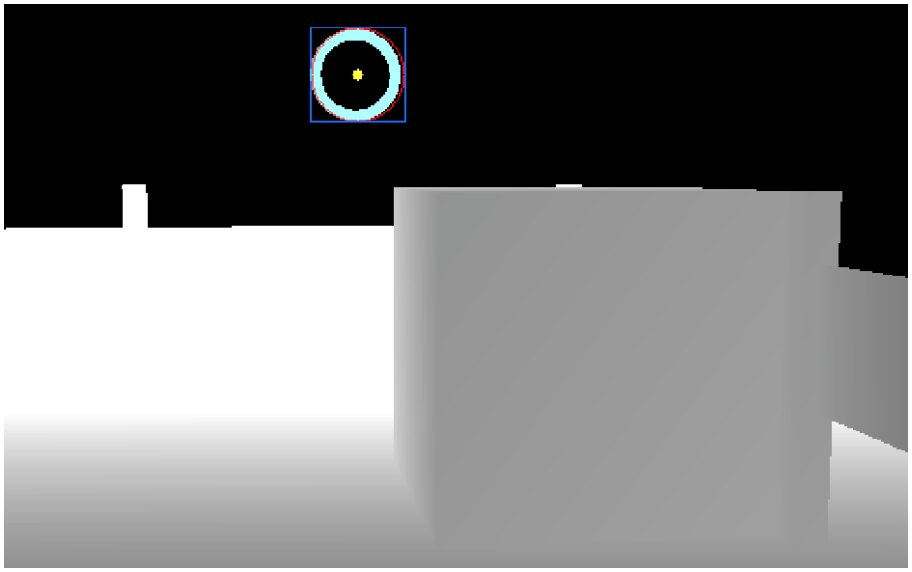

Figure 2: Circle detection from medium distance.

The detection also works well in cases where a part of the ring is not visible, as one of the main features of the Hough transform is robustness to partial occlusion:

Figure 3: Circle detection with only one half visible.

4.3 Cylinder detection

We first set out to detect cylinders as described in the exercise, on the point cloud with PCL. But this proved cumbersome and very resource intensive, even if we limited the density of the cloud. We therefore switched our approach and implemented detection in 2D with the laser scan.

We first converted the scan data from the polar coordinates it comes in to a top down view in Cartesian coordinates, and then used the Hough transform, just as with ring detection. This approach proved to be much more efficient, but also more robust, as we were able to run the detection at a higher rate.

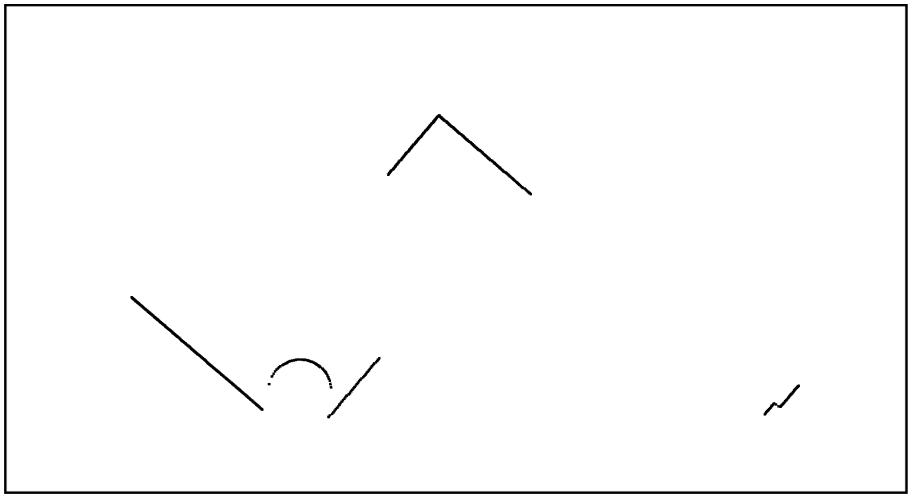

Figure 4: An example of the laser scan data used to detect cylinders.

Cylinders are much larger than rings, and we had little to no issues with color detection, as the margin for error in camera alignment is much higher.

4.4 Color detection

With cylinders, where we used the simple HSV approach, we had no issues and the detection came out very robust, mainly due to the small ammount of possible colours and the larger area of cylinders to work with.

With rings on the other hand we used a different method, a Python library that promised to label many different colors, and it does, but we sometimes got mismatches due to the color recognized not being exactly the colour we were looking for. Specifically, the library differentiates the normal and dark version of the colour, and the detection was not always consistent, so we decided to, for instance, treat green and dark green as the same colour.

The color detections are best demonstrated in figure 1, where the cylinders get matching colour dots and the rings are also labeled with the colour name.

4.5 Face detection and recognition

Since we had implemented face detection in previous exercises we had little issues with it for this task. We then only had to recognize the faces from our available roster. Due to knowing that we have a small set of possible faces we immediately opted for a distance, or rather similarity, approach. We encode the detection same as our sample pictures and find the one most similar.

The resulting classifier is very robust and we have not seen it mislabel a face.

Figure 5: A marked recognized face.

4.6 Food recognition

When a cylinder is detected, the robot is to move towards it and determine which food sits on top of it.

We had slight issues with occurrences where the cylinder got detected right as the robot reached its intermediate exploration goal. Since our exploration is designed in such a way where the robot receives it's next goal when it reaches the previous one, what sometimes happened was that the next exploration goal would be received just after the goal of investigating the cylinder, thus overriding it.

But once the robot correctly moved to investigate the cylinder, the food recognition was very robust.

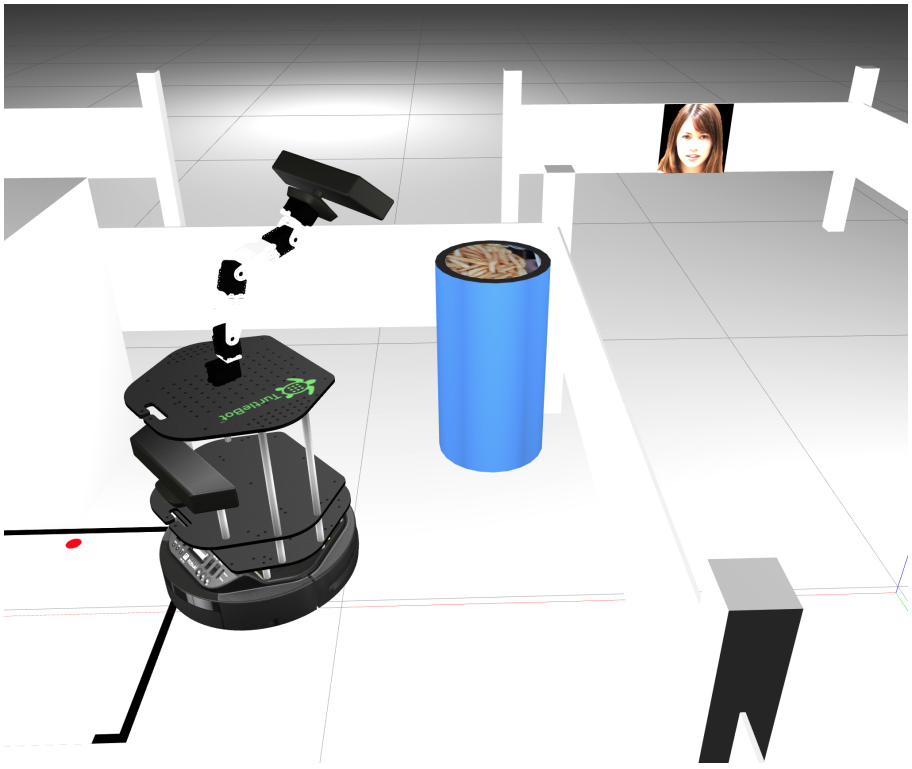

Figure 6: Robot investigating the cylinder to detect food.

Retraining the model on the provided images took us 54 minutes and yielded a model with a 96% accuracy.

We determined the exact location of the food, or rather the location of the top of the cylinder, by fitting ellipses to the image of the manipulator arm looking at the top of the cylinder. We could then extract a bounding box around the eclipse and get a nice clean image with only the food and minor perspective variation, which we passed to the classifier.

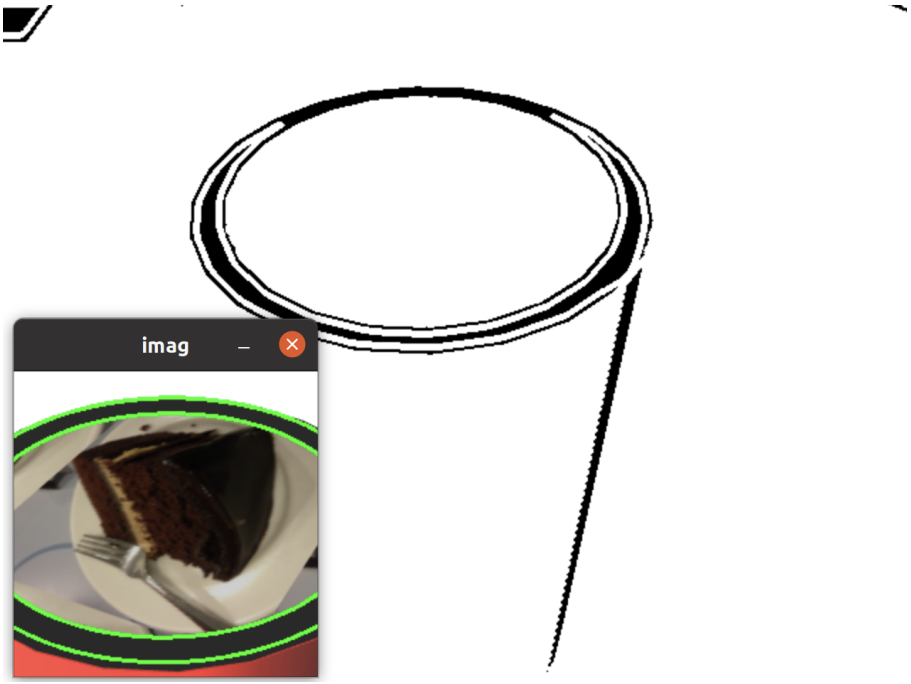

Figure 7: Food recognition on the blue cylinder.

In our testing we have not encountered a case where the robot would incorrectly classify a piece of food once the cylinder was correctly detected and investigated.

4.7 Parking

After the robot detects all the rings, cylinders, and people in the environment, it moves to the place below the green ring and prepares to park in the parking space in front of it. Based on the mentioned procedure, the robot then moves towards the red dot located inside the parking space. When the robot is properly parked in the parking space under the green ring, we detect and read the QR code in front of it with the help of a camera.

To avoid the precision parking of the robot, we limited the parking procedure to the `PARKING_NUM` number of iterations. The value of the mentioned constant was determined based on testing and it turned out that the value of 250 iterations works best for our simulated environment.

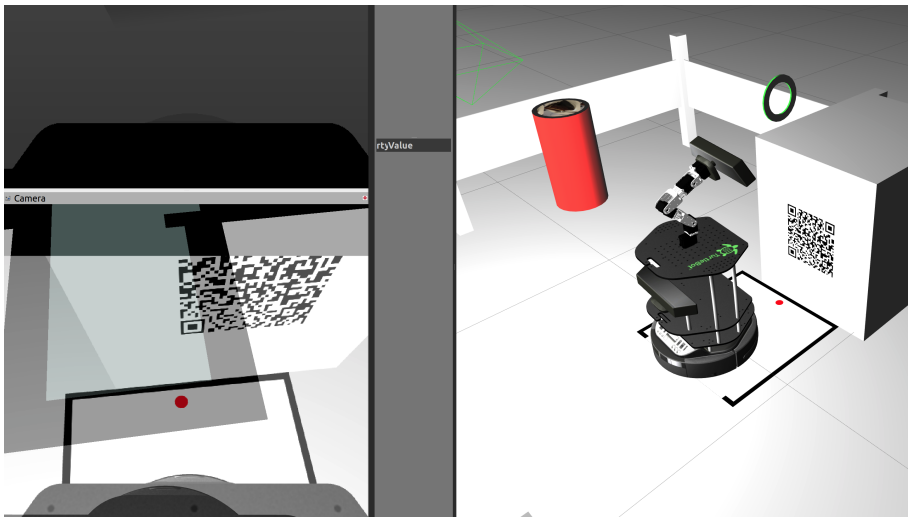

Figure 8: Approaching the parking spot under the green ring.

4.8 Food delivery

Once all the items are detected and the robot is parked, the food delivery process starts. Reading the QR code in the parking space tells the robot what food needs to be delivered to which person. We perform a simple route optimisation, and we simply first serve people who are closer to the robot. That is start with the first item, fetch the food and deliver it to the person, then repeat for all the other items in the list.

To avoid setting the goal inside the cylinder, we move the goal slightly away from the center of the cylinder, in the direction of the angle of approach.

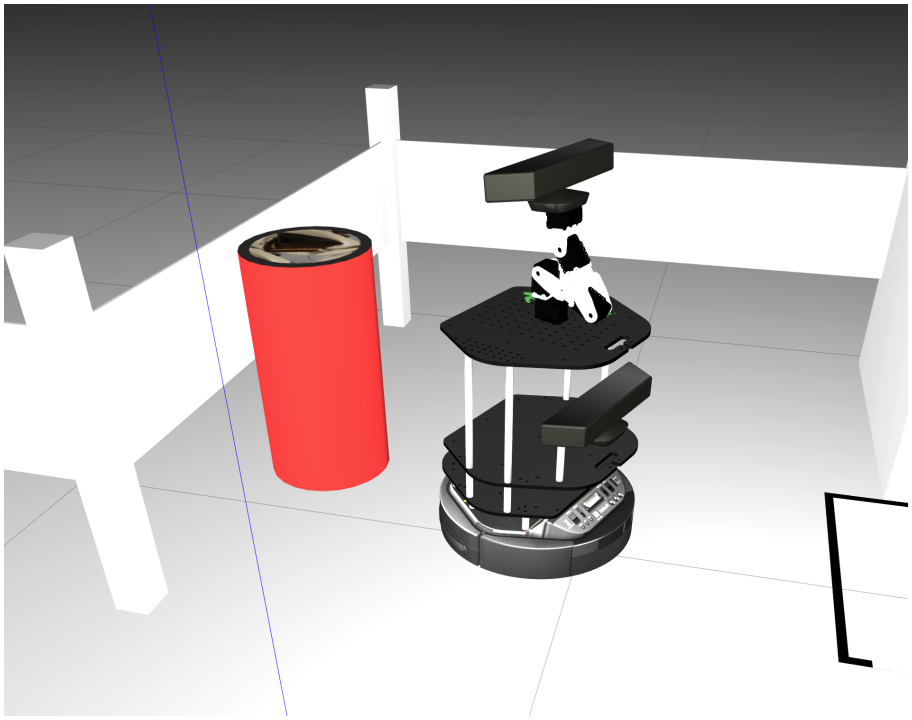

Figure 9: Fetching the food to be delivered.

It is then a trivial matter of setting the goal at the appropriate face marker for where the food needs to be delivered, according to the names gotten in the face recognition step.

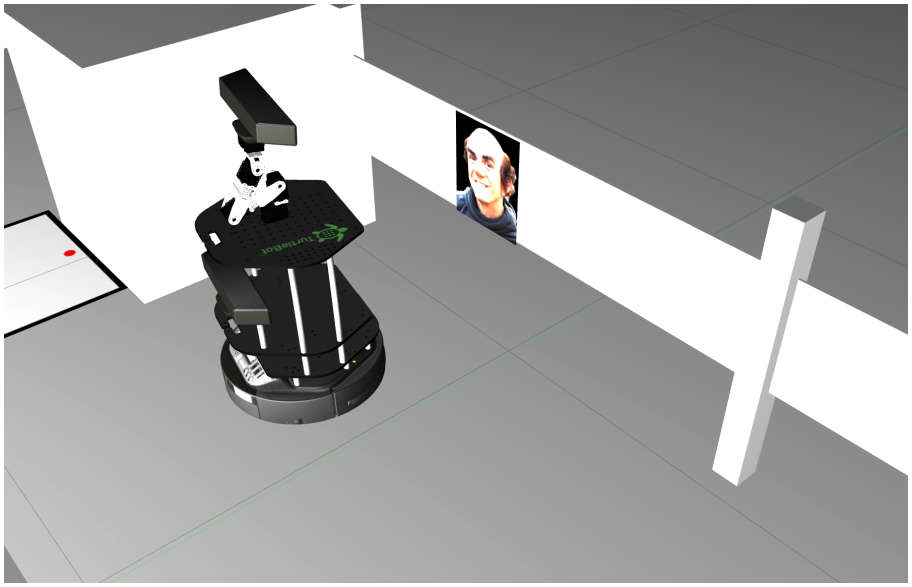

Figure 10: Delivering the food to the appropriate person.

5 Division of work

A rough division of work across the requirements is provided in the following list, followed by descriptions of their work by each member:

- Face detection (*Rok Mokotar*)
- Ring detection (*Jan Zorko*)
- Cylinder detection (*Jan Zorko*)
- Colour recognition (*Julija Klavžar*)
- Face recognition (*Rok Mokotar*)
- Approaching cylinders (*Jan Zorko*)
- Approaching faces (*Rok Mokotar*)
- Taking images of food (*Jan Zorko*)
- Food recognition (*Jan Zorko*)
- Parking (*Rok Mokotar*)
- QR-code detection (*Julija Klavžar*)
- Taking orders from the link (*Julija Klavžar*)
- Optimal path planning (*Rok Mokotar*)
- Auton. space exploration (*Julija Klavžar*)
- Dialogue with ASR (*Julija Klavžar*)
- Weaving with manipulator (*Rok Mokotar*)

Julija Klavžar: "I implemented color detection for cylinders and rings, where I built upon the work that Jan had done in cylinder and ring detection. I also implemented the QR-code detection that contains the delivery data. I worked on generating the goals for autonomous exploration and implemented the dialogue with ASR when delivering food to a person. I would say that I contributed a bit less than my team members, as I found the work very difficult and also had very slow equipment. The details of the implementation and theory behind it are provided in the Methods and Implementation sections."

Rok Mokotar: "I did face detection, where the robot detects faces in a given environment. The robot then had to properly recognize the faces, approach them, and address them in later parts of the project. I implemented parking the robot in the appropriate place after it detected the green ring and all other rings, cylinders, and faces in the environment. After the robot obtained the commands from the QR code, I worked on optimal route planning so that the robot first serves the person closest to it. I also added functionality to the robot that responds appropriately with his robot hand based on the customer's response. Finally, I also had the main role of integrating all implemented functionalities into one coherent system."

Jan Zorko: "I mainly focused my work on detection and recognition. I first started with ring detection in the first task and then implemented cylinder and food detection in later ones. Once I had food detection, or rather cylinder top detection, working, I built the recognition model and implemented labeling. I also worked with Rok to design the robot approach that is used for both faces and cylinders, and adapted it to work with food recognition. Due to work I may not have done my exact fair share of work, but I think we distributed it effectively."

To sum up, as a team we worked as a whole, and despite doing different things we always helped each other and advised how to make the implementation even better. In a rough estimate, each member brought about the same to the final value, and the values below better illustrate the percentage contribution of the member to the final product.

Julija Klavžar $\approx 30\%$ Rok Mokotar $\approx 40\%$ Jan Zorko $\approx 30\%$

6 Conclusion

In this report we described the theoretical and practical aspects of the implementation of the final task for the course Development of Intelligent Systems.

Along the way we faced many issues with both hardware and software. Most prominently our laptop computers were under-powered, causing frequent freezes and generally slow performance, which slowed down our development speed drastically. We also grappled with the intricacies of ROS and its many areas of potential errors, taking up days of debugging time to resolve some errors that prevented us from running the program on specific computers. Some issues also stemmed from the fact that not all of us had full ownership of our computers, so dual-booting Ubuntu was not an option and we had to make due with the Windows Subsystem for Linux, which even further potentiated the slower hardware. But in the end we managed to either resolve or work around the issues and complete the tasks at hand.

Compared to our poor performance on the previous task we feel quite satisfied with the outcome of this one. There were no major shortcomings and the demonstration went very well.

We learned a lot and are grateful for this experience, as we are very likely to run into ROS in our further careers and so a solid foundation will help us immensely.

To better present the implemented project, we made a recording of how the robot works in a given simulated environment. The video is available at the following link: https://youtu.be/kIe1qp8iarY.