Vision for Autonomous Robots
Lab Project 2: Detection & Localization

Wouter Bant (13176676)
Angelo Broere (13168215)
Gidi Ozery (14643618)
Valeria Sepicacchi (15187853)

Due: November 24th, 2024 CEST

1 Introduction

In this assignment, we implement marker detection, localization, pose estimation, and adversarial robot detection for the RAE robot. We combine these components with wheel odometry information to play curling against other RAE robots. We implement strategies to avoid hitting other robots while trying to get as close as possible to the target location.

2 Session 1: Marker detection by an RAE robot

2.1 Code for Marker Detection

We used the OpenCV Aruco module to allow the RAE robot to detect the markers:

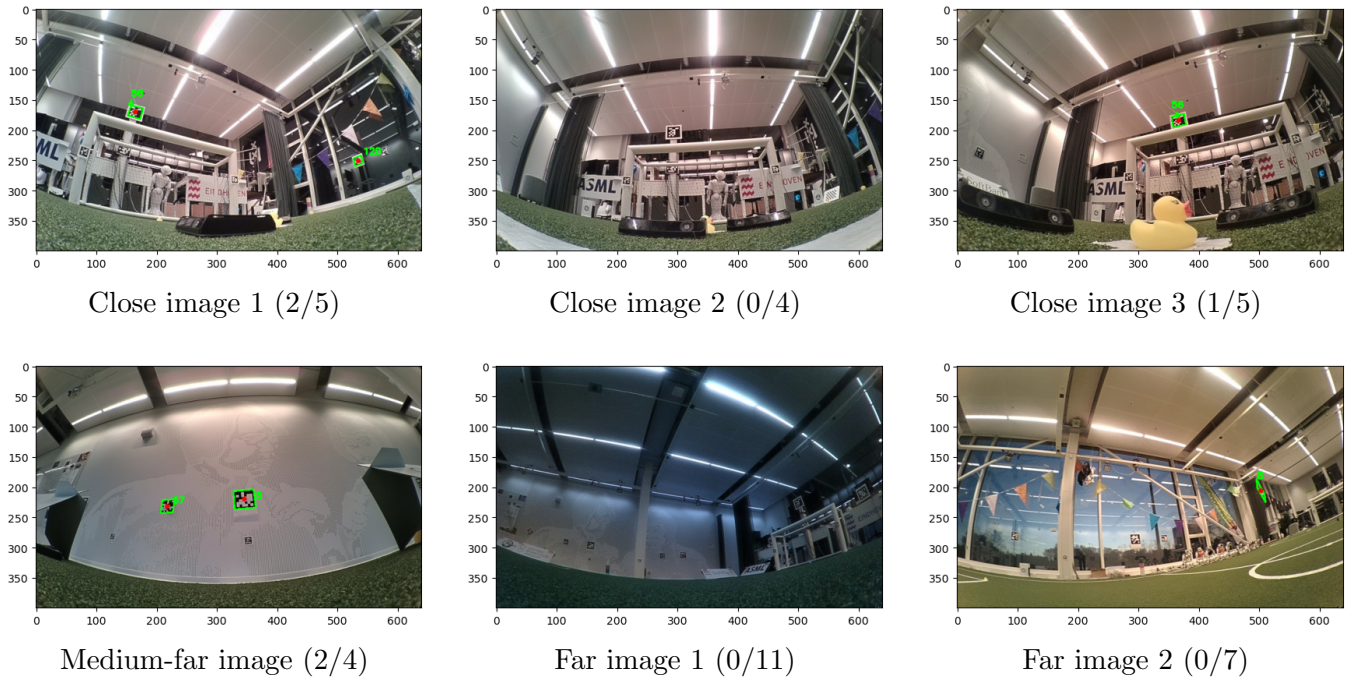
```
ARUCO_DICT = {
    "DICT_4X4_1000": cv2.aruco.DICT_4X4_1000,
    "DICT_5X5_1000": cv2.aruco.DICT_5X5_1000,
    "DICT_6X6_1000": cv2.aruco.DICT_6X6_1000,
    "DICT_7X7_1000": cv2.aruco.DICT_7X7_1000,
    "DICT_ARUCO_ORIGINAL": cv2.aruco.DICT_ARUCO_ORIGINAL,
    "DICT_ARUCO_MIP_36h12": cv2.aruco.DICT_ARUCO_MIP_36h12,
    "DICT_APRILTAG_36h11": cv2.aruco.DICT_APRILTAG_36h11,
}

for desired_aruco_dictionary in ARUCO_DICT.keys():
    this_aruco_dictionary = cv2.aruco.getPredefinedDictionary(
        ARUCO_DICT[desired_aruco_dictionary]
    )
    (corners, ids, rejected) = cv2.aruco.detectMarkers(
        frame, this_aruco_dictionary, parameters=self.aruco_params
    )
```

2.2 Initial Robustness and Analysis of Failure Cases

We display the initial results with default parameters and no preprocessing in Figure 1. In the next section, we display the image size along the axis to compare it with our improved method. Note that we did not rectify the image as we saw that detection performance was better without it. For example, some markers are detected in close images 1 and 3 and in the medium-far image. However, many markers were not detected. In particular, none of the small markers were detected. Further, we can see a false positive in the far image 2. We can conclude that either the marker is too small or too far away to be detected. The small markers on the goalposts are a prime example, these are much closer than the small markers on the window, but still not detected.

2.3 Solutions for Robustness and Comparison

From our discussion above, we conclude that small or distant markers are hard to detect. We tried several preprocessing steps like contrast enhancement, binarization, and image sharpening, but those



Close image 1 (2/5) | Close image 2 (0/4) | Close image 3 (1/5)

Medium-far image (2/4) | Far image 1 (0/11) | Far image 2 (0/7)

Figure 1: Detected markers with default parameters and no preprocessing. The fractions indicate the accuracy of the detection.

all did not fix the issues. However, increasing the image size by three times (this exact value worked best for us) using Bicubic interpolation significantly improved the results for small markers. Figure 2 shows the same marker with the same number of pixels. What can be observed is that the colors of the markers appear more uniform and of course, the marker appears larger. We postulate that combining these factors makes it easier for the default function in the Aruco module to detect the marker.

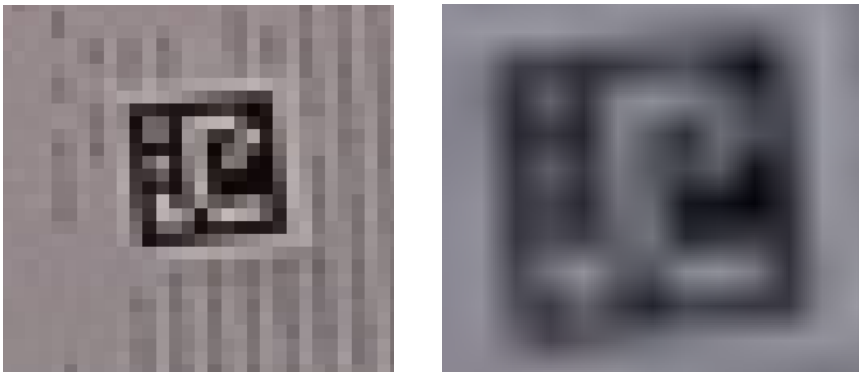
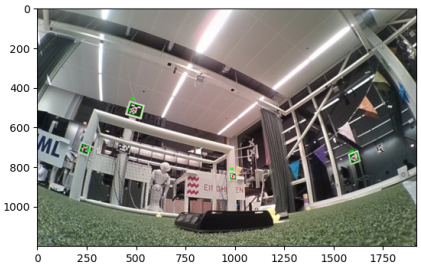


Figure 2: Impact of Bicubic upsampling in the medium far image (left original, right with upsampling).

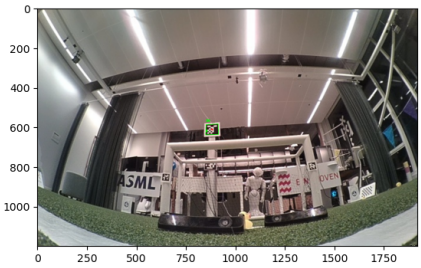
We did an extensive manual hyperparameter search and found that the function's parameters rarely influence detection results. Even lowering the 'minMarkerPerimeterRate', which controls when to reject a marker when it is too small, did not impact results. Besides that, we found that most rejected detections were false positives and therefore decreasing this value too much increased the rate of false positives. The only parameter that showed slight improvements in the results was 'adaptiveThreshConstant'. Slightly lowering this helped detect the far marker in far image 1 (See Figure 2). This value controls the constant that is subtracted from the mean in the adaptive thresholding process, and can possibly increase results in dim light.

The results with the new 'adaptiveThreshConstant' and bicubic resizing are displayed in Figure 3. Again the image size is shown along the axis for comparison with the previous results. It can be seen that a lot more markers are detected including the smallest ones. Also, the false positive disappeared in far image 2. Also, even from this distance a marker on the window is detected. In far image 2, two markers are detected, even though there is poor lighting and the markers are far away. The marker on the wall is detected because of the lower adaptiveThreshConstant, showing its importance in dim light. In close images 1 and 3, almost all markers in the field of view are detected (4 out of 5). We found

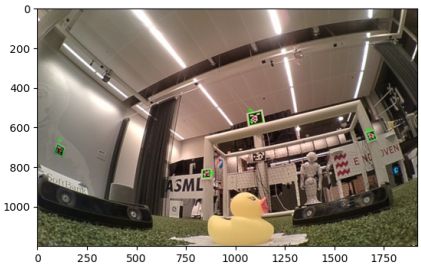
that the detection results are robust enough to have at least two detected markers, enabling accurate triangulation results.



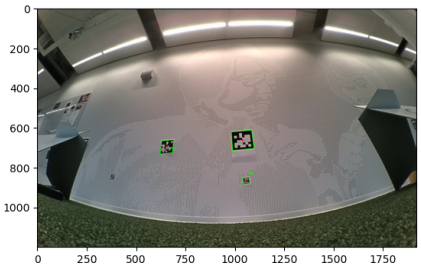
Close image 1 (4/5)



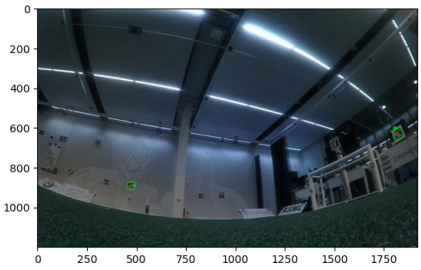
Close image 2 (1/4)



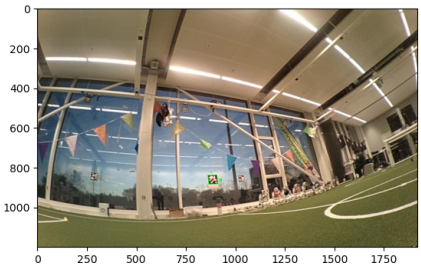
Close image 3 (4/5)



Medium far image (3/4)



Far image 1 (2/11)



Far image 2 (1/7)

Figure 3: Detected markers with bicubic resizing and lower adaptiveThresholdConstant. The fractions indicate the accuracy of the detection.

3 Session 2: Localization by an RAE robot

3.1 Distance Measurement and Error Reduction

We use Aruco for the distance estimation and this requires the size of the marker as input. Hence, the error reduction part is already done immediately. To further reduce the error, we specify our intrinsic matrix (appropriately rescaled to adjust for the image resizing) and distortion parameters obtained in the first lab assignment. This avoids reestimating the intrinsic matrix every time. We found that our obtained parameters are already accurate. These can be used to reliably estimate the remaining unknowns.

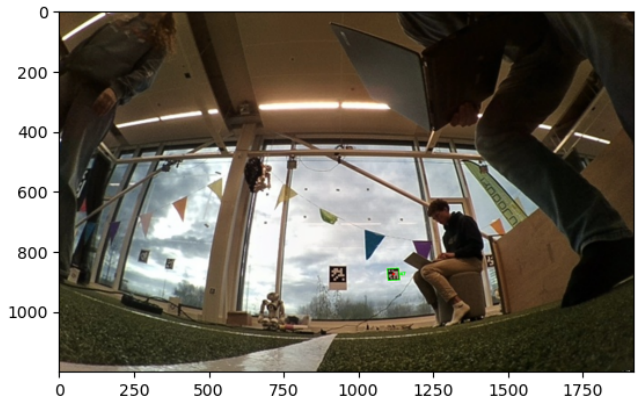
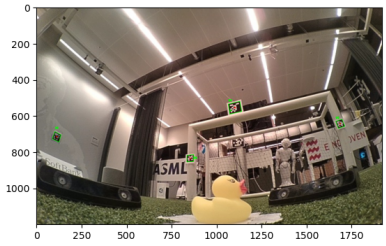


Figure 4: True distance: 430cm, Estimated with proposed method: 427.35cm, Estimated default: 0.56cm.

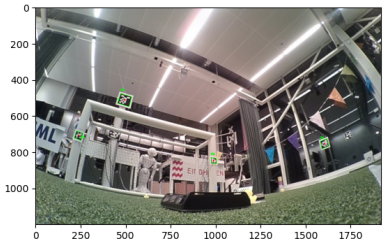
```
K *= resize_factor # adjust intrinsic matrix because of enlarging image
K[2,2] = 1.0
rvecs, tvecs, _ = cv2.aruco.estimatePoseSingleMarkers(
    marker_corner, marker_size, K, distortion_coefs
)
estimated_distance = tvecs[0][0][2]
```

Figure 5 shows typical results for different distances. We can consistently and accurately determine the distance to the markers for close and medium far tags. However, for targets far away the distance estimation can be quite off. An example of this is shown in the third image (See Figure 5). In this case, the distance estimation is off by almost one meter. We also measure the distance from the center to the right target on the window and its true distance to 430cm (this can be seen in Figure 4). When we provide the calibration and distortion coefficients, we are off by 3cm with an estimated distance of 427cm. Without the intrinsic matric and distortion coefficients the estimated distance is 0.56cm. This is because of the hefty barrel distortion.

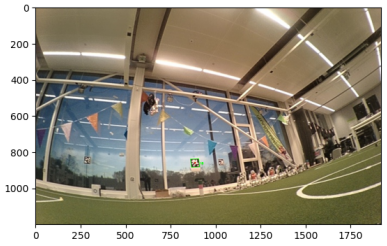
In our experiments, we did not observe exclusive over or underprediction. Since both were present, multiplying by some constant to get more accurate results is not feasible. To reduce the localization error, we will abstain from using markers that are far away when markers close by are detected.



(a) For marker on right goal post: Actual: 133.72, Predicted: 121.08, Distance error: 12.64



(b) For the highest marker: Actual: 278.10, Predicted: 272.08, Distance error: 6.02



(c) Actual: 584.20, Predicted: 488.90, Distance error: 95.3

Figure 5: Actual and estimated distances.

3.2 Triangulation

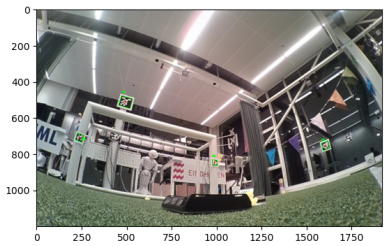
Let $L_1 = (X_1, Y_1, Z_1)$ and $L_2 = (X_2, Y_2, Z_2)$ be the locations of the detected markers and d_1 and d_2 the estimated distances to these markers. With Pythagorean theorem we calculate the distance to the markers as if they were located on the ground: $d_i^g = \sqrt{d_i^2 - Z_i^2}$. Now we have to find the intersection points of two circles with radius d_1^g and d_2^g centered around (X_1, Y_1) and (X_2, Y_2) respectively. The easiest equations for finding the intersection points of two circles were found here. Applying these to our radii:

$$R = \sqrt{(X_2 - X_1)^2 + (Y_2 - Y_1)^2}$$

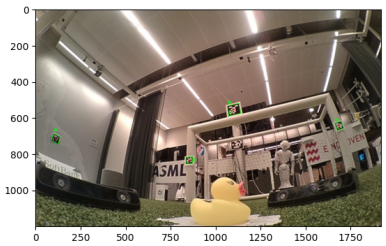
$$(x, y) = 0.5(X_1 + X_2, Y_1 + Y_2) + \frac{(d_1^g)^2 - (d_2^g)^2}{2R^2}(X_2 - X_1, Y_2 - Y_1) \pm$$

$$0.5\sqrt{2\frac{(d_1^g)^2 + (d_2^g)^2}{R^2} - \frac{((d_1^g)^2 - (d_2^g)^2)^2}{R^4} - 1}(Y_2 - Y_1, X_1 - X_2)$$

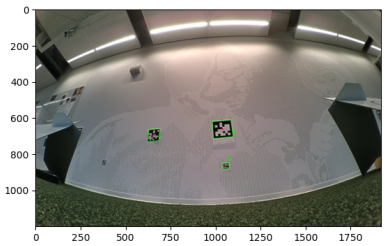
We now have two intersection points (x, y). We pick the one closest to the previous location of the robot. For initialization, we specify the start position of the robot. This prevents using the extraneous solution in almost all cases as we only move the robot slowly.



(a) Actual: (-85, 310), Predicted: (-87.63,338.55), Distance error: 28.67



(b) Actual: (15,338), Predicted: (38.29,339.13), Distance error: 23.32



(c) Actual: (-301,0), Predicted: (-300.89,1.87), Distance error: 1.87

Figure 6: Results with Exact method using closest two markers.

Figure 6 shows the results for the exact triangulation when using only the two closest markers. The

errors are 28.67, 23.32, and 1.87 respectively. For the frame on the left, the error is mostly in the y location while the x location for the second image is worse estimated.

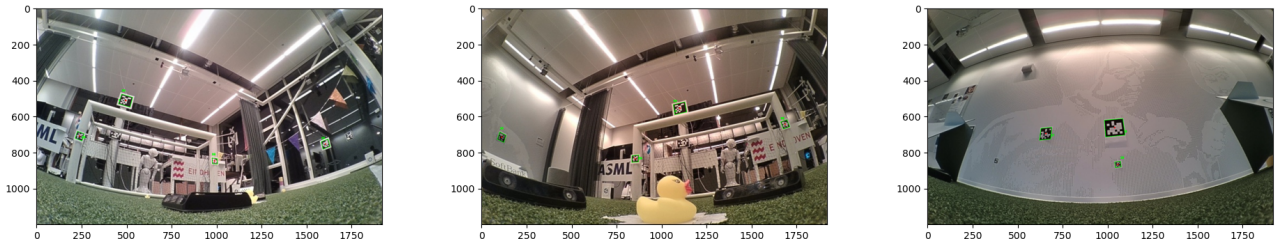
3.3 Triangulation with more than 2 points

For triangulation with more than 2 points we consider two solutions. The first still uses the exact triangulation from the previous section and selects the two closest markers, motivated by our finding that close marker distances are more accurately estimated.

However, in practice, we found that we could obtain more accurate results when we used the estimated distances from all markers. We now do this via a least squares solution:

```python
def triangulate_2d_ls(self, marker_detection_results):
    if len(marker_detection_results["marker_ids"]) < 2:
        return self.previous_location

    def residuals(camera_position_2d, landmarks, distances):
        camera_position = np.array([camera_position_2d[0], camera_position_2d[1]])
        estimated_distances = np.linalg.norm(landmarks - camera_position, axis=1)
        return estimated_distances - distances

    initial_guess = self.previous_location
    landmarks = np.array(
        [
            [location.x, location.y]
            for marker_id in marker_detection_results["marker_ids"]
            for location in [MARKER_ID_2_LOCATION[marker_id]]
        ]
    )

    distances = np.array(
        [   # get the distance over the ground to the marker
            np.sqrt(distance**2 - MARKER_ID_2_LOCATION[marker_id].z ** 2)
            for marker_id, distance in zip(
                marker_detection_results["marker_ids"],
                marker_detection_results["marker_distances"],
            )
        ]
    )

    result = least_squares(
        residuals,
        initial_guess,
        args=(landmarks, distances),
    )

    if abs(result.x[0]) > 300 or abs(result.x[1]) > 450:
        return self.previous_location  # robot cannot be outside the field

    self.previous_location = result.x
    return result.x
```

Importantly, we should initialize the least squares procedure with the previous location. We found that this allows us to almost always filter out the extraneous solution. In cases where this did not happen or when we have less than two detected markers (rarely), we return the previous approximated location. This is typically close to the actual location as we limit the linear speed to 0.5 and only move every 0.2 seconds. Within that time, we process approximately 5 frames. Note that we have a similar procedure for the exact triangulation in the sense that we also project the marker to the level of the robot camera. We found that the least squares solution provides the same solution as the exact method when provided with two markers and provides better locations in approximately 90% of the frames when more than two markers are detected. For these reasons, we use the least squares solution by default. These results can be seen in Figure 7 and compared to the results from the exact method, only using the 2 closest markers, the least squares solution performs better. This is likely because all measurements are a bit noisy and when multiple estimates are combined the variance of the final estimate will be lower (even though the estimated distances of far markers are typically less accurate).



(a) Actual: (-85, 310), Predicted: (-84.73,332.14), Distance error: 22.14

(b) Actual: (15,338), Predicted: (7.26,344.22), Distance error: 9.93

(c) Actual: (-301,0), Predicted: (-301.14,-2.13), Distance error: 2.13

Figure 7: Results with Least Squares method using all markers.

3.4 Realtime Localization of the RAE robot

With the least squares solution from above we can accurately determine the position of the robot when at least two markers are detected. Our solution for when we detect only one or no markers consists of two parts.

First, we do **pose estimation** (visualized in Figure 8) when we detect at least one marker. This is done by using each marker's estimated x translation. Based on each marker's location and its estimated x translation, we determine the robot's angle on the field and take the average, in the case of having more than 1 detected marker, to obtain a more robust estimate. Also, this pose estimate is almost always accurate. When the robot detects fewer than two markers, we slow down the robot and do less steering so that the pose does not change much.

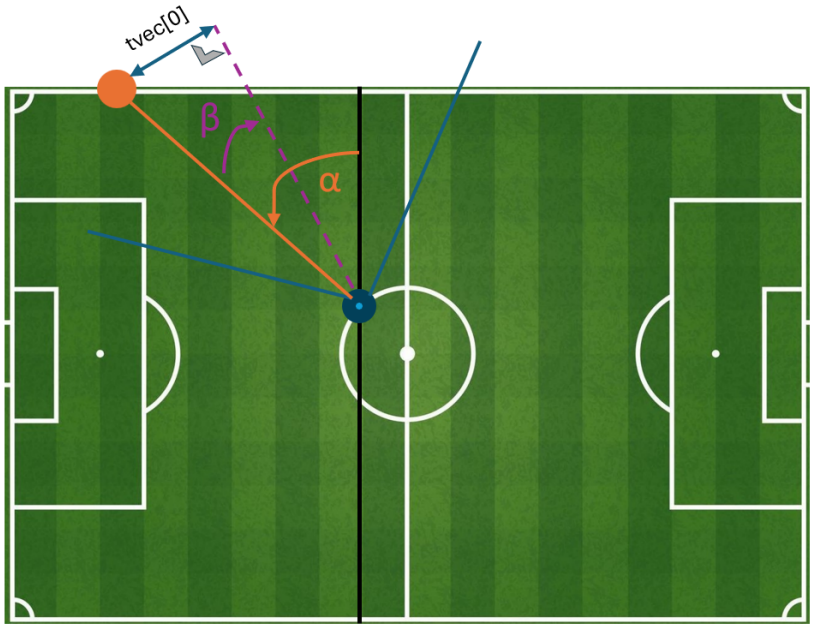


Figure 8: Two-step process for pose estimation. First, α is computed using the known marker location (orange dot) and estimated robot position (blue dot). Second, β is derived from the estimated x-translation (tvec[0]). The final pose is calculated as $\alpha - \beta$ when the marker is on the left side (negative x-translation), or $\alpha + \beta$ when the marker is on the right side (positive x-translation). Blue lines indicate the field of view, with the dotted purple line representing its center.

Second, we tracked the wheel rotations of the robot, however, we found that this information is unreliable, especially on the white lines on the field the estimated location based on this information can be off by many centimeters.

Theoretically combining odometry information with the visual information in the Kalman filter should reduce the variance of the estimated position when we assume that both estimates are unbiased. However, we found many biases in the odometry information. When driving the robot in a straight line, the left wheel consistently showed more rotations. We compensated for this by multiplying the right wheel rotations by a constant which makes the number of wheel rotations equal when driving a straight line. However, this introduced some biases for rotations which we did not further address. Based on

the number of rotations we also calculated the actual distance traveled so we know the correspondence between distance traveled and wheel rotations.

In practice, we found that using odometry information is marginally useful when no markers are detected. In these situations we do less steering and slow down the robot. As a result, the errors from steering and slipping are less prominent.

4 Session 3: Using RAE robot for curling match

4.1 Driving the RAE Robot to the Target

The general algorithm:

1. Marker detection.
2. Triangulation to determine the position.
   - Do exact triangulation with two detected markers.
   - With more detected markers use least squares.
   - With one or zero detected markers use odometry info and the last location.
3. Pose estimation to determine the angle of the robot on the field.
4. Robot detection.
5. Action based on location, pose, location of detected robots, and target location.

The first three steps of the algorithm have already been discussed. We will now describe how we determine the action when no other robots are on the field. We do not drive directly to the target location, but, like in real curling, we can control the position where the robot will go first. After the robot has achieved an acceptable distance to that goal it will switch to the real target and stop when this distance is within the same threshold (we found that 30cm worked well). We determine the angle to move by differences in the target location and the current location and consider the pose. Further, we slow down the robot when it gets closer to the target location. We do this as the latency can be some seconds and slowing down ensures that the distance between the frame used and the actual location of the robot are not too different. Finally, we limit the linear and angular speed such that the robot will never turn or move too fast. We collect the actions and send a move command every second to the robot which is the average of the last 5 actions, this makes the solution more robust against false positive detections.

```
def move_to_target(
        self, current_pos: tuple[float, float], obstacle_detection: Tuple[bool, Set[
    str]], pose: float
    ):

    if self.location_1:  # Move towards a first location
        dx = self.config.get("target_x_location1") - current_pos[0]
        dy = self.config.get("target_y_location1") - current_pos[1]
    else:  # Move towards the actual target
        dx = self.config.get("target_x_location2") - current_pos[0]
        dy = self.config.get("target_y_location2") - current_pos[1]

    distance = math.sqrt(dx * dx + dy * dy)
    if distance < self.config.get("position_tolerance"):
        if self.location_1:
            self.location_1 = False
        else:
            self.stop_robot()

    target_angle = np.degrees(math.atan2(dy, -dx))
    use_angle = pose - target_angle  # Take pose into account

    cmd = Twist()
    cmd.linear.x = self._get_linear_velocity(distance)  # Slow down if close to target
    cmd.angular.z = self._get_angular_velocity(use_angle)
```

```
    # Limit speeds
    cmd.linear.x = min(cmd.linear.x, self.max_linear_speed)
    cmd.angular.z = max(
        min(cmd.angular.z, self.max_angular_speed), -self.max_angular_speed
    )
    return cmd
```

4.2 Danger handling

4.2.1 RAE robot detection

The robot detection pipeline is visualized in Figure 9. We rectify (b) the input image (a). Subsequently, as we know robots can only be on the ground and do not care about robots far on the side we apply cropping (c). Now we mask out the upper corners (d), we do this as distant robots that are not in the direct line of the robot are less important, and because these areas often contain dark objects like chair wheels at the end of the field. Before we apply contour detection we convert the frame from RGB format to HSV format as this makes the detection of dark object contours more robust to lighting conditions. This is because black and dark shades are primarily defined by a low value, irrespective of their hue or saturation. For contour detection (e), we determined the HSV lower bound values to be (0, 0, 0) and the upper bound values to be (180, 255, 85). To remove noise we apply erosion (f).

Now to determine if there is actually a robot, we look at the bottom lines of each contour. If this is wider than 35 pixels we conclude that it is a robot. We look at the center of these lines and determine where they are in the image (left, center, or right). Based on the contours and their position in the image (left, center, or right), we determine if the robot is 'in danger'. The robot is in danger when one of the robots is sufficiently close, which is the case when the bottom line of the contour is in the bottom half of the cropped image.

(a) Input image.

(b) Image rectification.

(c) Image cropping to RoI.

(d) Mask out upper corners.

(e) Contour detection based on dark color.

(f) Noise removal with erosion.

Figure 9: Robot detection pipeline.

4.2.2 Trajectory adjustment

When robots are detected, we adjust the trajectory to the target in different ways:

- If there is a robot exactly on the target location, we stop just before it (when the contour line is a couple of pixels from the bottom).
- When the robot is in danger we slow down and move around the robot if this side is not blocked by other robots. If no escape route is available we stop.
- If the robot is not in danger, but the calculated trajectory contains a distant adversarial robot, we slow down our robot.

This can be summarized in the following pseudo-code (the actual code works similarly but is distributed over many functions and types are a bit different):

```
inputs = {inDanger: boolean, danger_positions: set[adversarial robot positions], trajectoryDirection}
if inDanger:
    Slow down the robot
    # Choose an escape direction
    if trajectoryDirection not in danger_positions:
        turn in trajectoryDirection
    else if "middle" not in danger_positions:
        Go straight
    else if "left" not in danger_positions:
        Turn left
    else if "right" not in danger_positions:
        Turn right
    else:  # Robot is fully blocked
        Stop the movement of the robot
else if trajectoryDirection in danger_positions:
    Slow down the robot # Do not adjust angular direction (yet)
```

4.2.3 Curling match results

We managed to have an average distance to the target across 5 rounds of 40.9 centimeters, coming in at second place. Overall, our robot was able to evade adversarial

4.2.4 Discussion and analysis

Our algorithm works well when:

1. Many near markers are detected.
2. There is a clear path to the target that does not involve going back.

In situation (1) the triangulation typically provides an estimated location within 10cm of the actual location. Also, the pose estimation is accurate within a few degrees making it easy to drive to an exact location.

Since our algorithm first goes to a specified location before going to the target destination, we can easily bypass robots near the target as long as the target is not fully blocked from the front (situation 2). When we see the target is blocked from the left, we first send the robot to a location in front of the target and more to the right. From there, the robot can simply drive in a straight line to the target.

Our algorithm does not work well when:

1. All detected markers are far away.
2. A target next to the window is fully blocked from the front.
3. High latency is present.

In situation (1) our distance estimates are poor and the estimated distances of the targets are off because of this the location and pose of the robot are inaccurate. This causes the robot to drive in a suboptimal direction. Incorporating odometry information did not help much to prevent this as this was often unreliable itself. Particularly after crossing white lines on the field did we experience unreliability.

In situation (2) the only way to get close to the target is to drive back. If we drive by on the left side and turn right the light that comes through the window causes shadows on the grass nearby sometimes causing a false positive detection of a robot. This then leads to the robot stopping too soon. If we bypass the target on the right and turn left only the distant markers on the left wall get detected and then triangulation and pose estimation will be bad. This was less of a problem on the left side of the field as there the nearby markers could still be detected.

In situation (3) the robot is basing its action on a frame from a couple of seconds ago. We tried to prevent this by only taking an action every second based on the last 5 images and using compressed images. Also, we decrease the speed when the robot gets nearer to its target or adversarial robots. However, when the robot gets hot the messages are sent late and this can cause the robot to drive too far or detect robots too late.

5 Conclusion

In this lab assignment, we designed a pipeline for an RAE robot that can robustly detect Aruco markers, localize its position inside a global coordinate system according to these markers, and drive to a target position, all while evading and preventing collision with adversarial robots. We found that using triangulation with all detected markers was the most accurate method for localization. Using odometry for localization proved to be unreliable and only slightly helped when less than two markers were detected. Our strategy of going to a different location before the target location in the curling match allowed us to drive in a straight line to the target when it was only blocked from one side.