university of groningen / faculty of science and engineering

Robotics Practical 2 – WBAI030
Academic Year 2021-2022

Jakub Łucki (s3986209)

group07

I. DISCLAIMER

Without figures, references, and disclaimer section, the report is around 4.5 pages. Moreover, my lab partner Alexandru Dimofte and I exchanged some of the figures. For example, the photographs of the real-life robots were taken by him. Whenever I used images made by him I indicated it appropriately. Just as in the case of any material used from other sources. Except for images we did not share any other details regarding the report.

II. SIMULATION AND ROBOT - [0.5 POINTS]

The goal of this project was to design an overall control program that controls a swarm of three robots that are supposed to (1) avoid obstacles, (2) follow human legs, and (3) maintain a triangular formation. The members of the swarm are unicycle mobile robots, which means that they are propelled by two fixed independently actuated wheels, and a passive ball wheel that acts as a support. This configuration gives the robots two degrees of freedom. Moreover, each robot is equipped with a 360° 2D lidar (for details, see Section III). The real-life robot can be seen in Figure 1

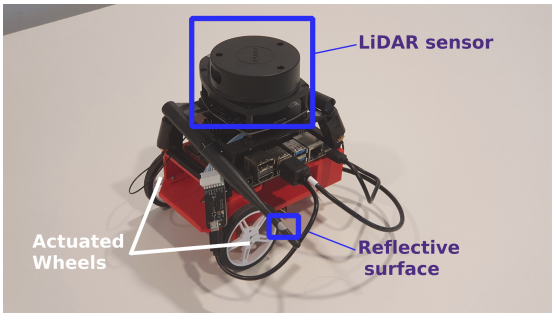


Fig. 1: Image of the unicycle mobile robot used in lab environment. *Source: courtesy of Alexandru Dimofte*.

The overall control was developed and tested in two environments: robotics lab and simulation. In the first setting, the robots were physical, which entails that they were equipped with parts necessary for their basic functioning such as batteries or processors. The lab environment consisted of a flat surface of size roughly $4m \times 4m$ to which arbitrary obstacles could be added. Furthermore, it included an OptiTrack motion capture system, which allowed to track the movements of robots in the world frame via reflective stickers placed on the robots. The latter setting is a virtual 2.5D environment generated by a simulator called Stage. It is a computationally-efficient simulator optimized for simulating multiple agents, their sensors, and arbitrary objects. Robots were represented as cubes.

III. LASER DATA CLUSTERING, FEATURE EXTRACTION AND CLASSIFIER - [3.5 POINTS]

The only sensor used by the robots is a 360° 2D lidar, which provides accurate range data in a short cycle time and is robust to illumination changes in the environment (according to the lecture slides). This effectively allows us to map the robot's surroundings. It gathers the sensory data by emitting a pulse of photons and measuring the time needed for the reflected photons to return. The time is measured via phase-shift between emitted and reflected signals. Then the range is calculated using the following formula: $\textit{distance} = 0.5 \times \textit{speed of sound} \times \textit{time}$. It is important to mention that, since the signal is emitted in all directions and the frequency of emitting pulses is finite, the further the obstacle from the sensor the sparser its mapping will be. In this project lidar was a primary source of information about the environment for the robots.

Readings from that sensor include the array of ranges, angle increment between each measurement, maximum range, and minimum range along with a few other, less relevant variables. Based on the first four pieces of information we can convert lidar's readings into proper polar coordinates and filter out erroneous measurements. Points in that coordinate system are of the form (r, φ), where r is the range indicating the distance from the center of the coordinate system to a point and φ is the angular coordinate that quantifies the angle from the x-axis to a point, measured counterclockwise. It is important to note that the origin $(0, 0)$ of this coordinate system will be the center of mass of the robot (where the lidar is placed). This coordinate system is very intuitive in the lidar setting, but for other applications, Cartesian system is more useful. Therefore, to convert points from the first to the second system we can use the equation (1).

$$x = r\cos\varphi, \;\; y = r\sin\varphi \tag{1}$$

Once all the points, forming the mapping of the environment, are converted to the Cartesian coordinate system, we can extract useful information from them. By clustering these points we can detect all the objects in the observable environment. Algorithm 1 responsible for that task is shown below. It iterates through all measured

points and checks whether the Euclidean distance (later: distance) between every two consecutive points is less than an arbitrary threshold. If this is the case, they form one cluster together, otherwise, a new cluster originates. There is one important edge case that was incorporated into the algorithm. The points are stored in a sequence, however, the measurements are taken in $360°$. As a consequence, the first and last points can belong to the same cluster, although they do not appear consecutively in the sequence. To overcome this the algorithm checks whether the distance between these two points is less than the threshold, in which case it merges the last-found cluster with the first cluster. The algorithm uses a threshold of $0.09m$. Minimum leg separation (in this project) is $0.1m$, thus the threshold had to be smaller, but the exact value was obtained through trial and error. Multiple objects at various distances from each other were placed around the robot. Then the threshold value was selected for which the number of detected objects was correct in the most of tested object arrangements.

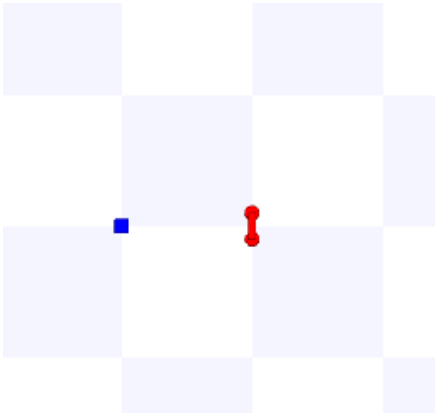
Algorithm 1 Clustering algorithm

```
points ← array of mapped Cartesian points
t ← 0.09                                   ▷ Empirically determined threshold (in m)
clusters ← empty array                     ▷ Array of clusters
prev ← points[0]
cluster ← array containing prev
for p in points[1 : end] do
    dist ← distance between prev and p
    if dist > t then
        clusters ← array containing clusters elements and cluster
        cluster ← empty array
    end if
    cluster ← array containing cluster elements and p
    prev ← p
end for
if clusters has any elements then
    dist ← distance between cluster[end] and clusters[0][0]
    if dist < t then
        clusters[0] ← array containing clusters[0] elements and cluster elements
        return clusters
    end if
end if
clusters ← array containing clusters elements and cluster
return clusters
```

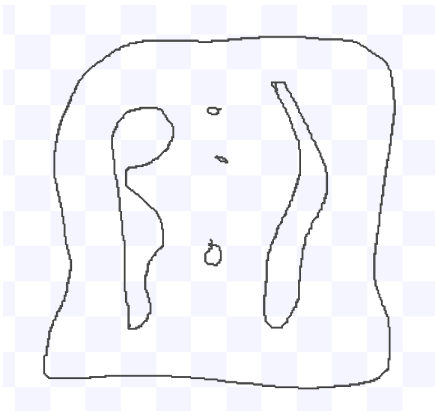
So far, detected objects are represented by a list of points. This can be troublesome, because the coordinates are in robot's frame of reference and objects can contain varying numbers of points. To enable object classification we need to extract a fixed amount of features, which are independent of robot's position. Chung, Kim, Yoo, Moon, and Park (2012) has shown that legs can be detected based on three features, depicted in Figure 3: width (blue), girth (red) and depth (green). Given a cluster $\{P_1, ..., P_i, ..., P_m\}$ of m points, we can get width by computing a distance between the first and last point of the cluster using Equation 2. Girth can be computed by summing the distances between all consecutive points in a cluster using Equation 3. Depth is the longest distance between $\overline{\boldsymbol{P_1 P_m}}$ and $\boldsymbol{P_i}$ and can be computed using Equation 4, where $\boldsymbol{L_1} = \boldsymbol{P_m} - \boldsymbol{P_1}$ and $\boldsymbol{L_2} = \boldsymbol{P_i} - \boldsymbol{P_1}$. To calculate the three features, there need to be at least three points in a cluster. Therefore, clusters with fewer elements are discarded beforehand.

$$\|\overline{\boldsymbol{P_1 P_m}}\| \quad (2) \qquad \sum_{i=1}^{m-1} \|\overline{\boldsymbol{P_i P_{i+1}}}\| \quad (3) \qquad max(depth(j)) = \sqrt{\frac{(\boldsymbol{L_2^T L_2})(\boldsymbol{L_1^T L_1}) - (\boldsymbol{L_1^T L_2})^2}{(\boldsymbol{L_1^T L_1})}} \quad (4)$$

With the abovementioned formulas, the features could be automatically computed from detected clusters and thus we could start to assemble the datasets. Since our tasks required us to distinguish legs from other objects, we had to collect two sets of data, one corresponding to legs and one corresponding to all the other objects. Moreover,



(a) Empty world with one robot and a pair of legs. No obstacles are present



(b) Cluttered world with no robots, nor legs, but with walls (black elements) working as obstacles.

Fig. 2: Examples of some environments used during developing the overall control. Both environments are simulations in Stage. *Source: Own private archive.*

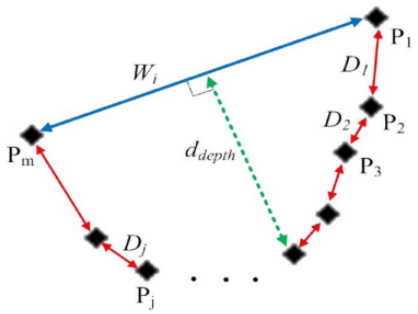


Fig. 3: Graphical representation of features: width (blue), girth (red) and depth (green), that can be calculated from a cluster of points. *Source: Chung et al. (2012).*

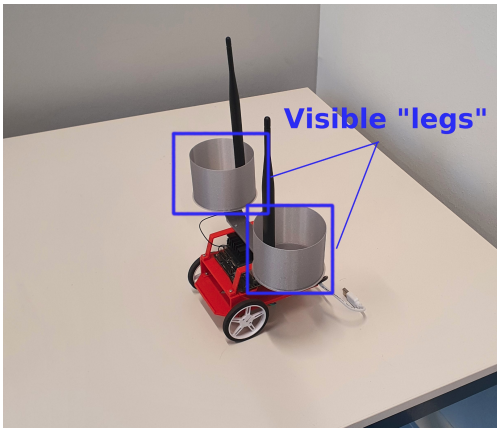


Fig. 4: The robot imitating human legs. *Source: courtesy of Alexandru Dimofte.*

the project concerned both the Stage simulation and real-life robots, hence we had to gather data once for each environment. In the first setting leg data was collected by placing the ”legs” and a single robot in an empty world, shown in the Figure 2a. Then the legs were moved around the robot, changing both the distance and alignment between them. Afterwards, we placed an individual robot in a cluttered environment, shown in the Figure 2b, and moved it around. Particular focus was put on jagged surfaces and small obstacles, which could easily be mistaken for legs. In the end, our simulation dataset contained 20443 data points corresponding to legs and 20443 data points corresponding to other not legs, where each data point consisted of 3 variables mentioned before.

In the lab environment collecting ”pure” leg data was more problematic, because of walls and other immovable obstacles. To overcome it, the points detected farther than $1.5m$ away from the robot were filtered out, before clustering. With a robot in the center of the lab and no obstacles within a $1.5m$ radius of the robot, we created an ”empty” world, where the legs were the only observable thing. The robot imitating human legs can be seen on Figure 4. Then we remotely steered them to collect data from all possible angles and distances. Once we gathered enough real-world leg data we removed the $1.5m$ filter, as well as, added extra obstacles to the environment. In the end the real-world dataset consisted of 20443 ”leg” data points and 20443 ”not leg” data points, each consisting of three variables. With the two datasets we could train the Support Vector Machine (SVM) classifier to distinguish leg clusters from the others. SVM is a machine learning algorithm, which aims to construct a hyperplane splitting all the points in a hyperspace so that on one side of the hyperplane there are only points belonging to one class. The algorithm projects the data points into higher dimensions where the classes, can become easily separable. One way to increase dimensions is to use a kernel which provides a way to quantify the similarity between two data

points. In our model we used Gaussian Radial Basis Function, shown in Equation 5, where $\boldsymbol{x}$, $\boldsymbol{x'}$ are samples, and σ is a free variable.

$$K(\boldsymbol{x}, \boldsymbol{x'}) = exp(\frac{\|\boldsymbol{x} - \boldsymbol{x'}\|^2}{2\sigma^2}) \tag{5}$$

The final algorithm used $C = 1000$, which quantifies the number of allowed misclassifications during training. The larger C, the more misclassifications are allowed. Moreover, our model had γ $(= 1/(2\sigma^2))$ set to $(3 \times var)^{-1}$, where var is the variance of the flattened dataset. We used `scikit-learn` to build and evaluate our model. As our primary metric, we decided to use accuracy, because the classes are evenly presented in our datasets. For both real-life and simulation datasets, we used 10-fold cross-validation to obtain the final performance. SVM trained on real-life data had 91.8% accuracy and SVM trained on simulation data had 92.6% accuracy.

IV. Null-space Control - [3 points]

A unicycle mobile robot's behaviour can be controlled on a low level by changing its linear velocity $v = \dot{x}$, where x is the robot's position on an axis, and rotational velocity $\omega = \dot{\theta}$, where θ is the counterclockwise rotation around the origin. All of the above variables are based on the robot's frame of reference, which means that the origin of the coordinate system is at the robot's center of mass and the x-axis is in the direction of the robot. The visualization of this frame of reference can be found in Figure 5.

A robot can be controlled by adjusting current velocities, but to select appropriate values for it to accomplish its goals we need a control framework called null-space-based behavioural (NSB) control. NSB control was introduced by Antonelli, Arrichiello, and Chiaverini (2008) and it aims to coordinate all assigned tasks by computing the most optimal velocities for each task individually and then combining the results. In NSB control each task is assigned a priority and the highest priority task is always fulfilled. Lower priority tasks are fulfilled only to the extent that they do not interfere with the immediately higher-priority task. The interference is avoided by projecting a velocity vector of a lower-priority task into a null space of an immediately higher-priority task, hence the name. This effectively removes the velocity components of lower-level tasks, which interfere with the higher-priority ones.

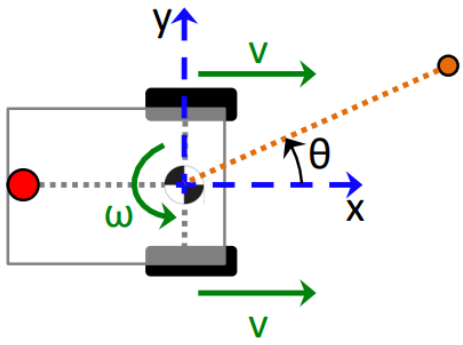


Fig. 5: Diagram of a robot with its frame of reference. Cartesian coordinate system (sparse blue) with robot's center of mass at its origin is shown. Moreover, the possible directions of velocities are shown in green. *Source: Carloni (2022)*

In this approach, we use $\boldsymbol{\sigma_i} \in \mathbb{R}^m$ for an arbitrary m, to describe i^{th} task to be accomplished and it is defined as $\boldsymbol{\sigma_i} = f(\boldsymbol{q})$, where $\boldsymbol{q} \in \mathbb{R}^n$ is the system configuration comprising of position x and rotation θ (hence, $n = 2$). It has the following differential relationship: $\dot{\boldsymbol{\sigma}}_{\boldsymbol{i}} = \frac{\partial f(\boldsymbol{q})}{\partial \boldsymbol{q}} = \boldsymbol{J}(\boldsymbol{q})\boldsymbol{v_i}$, where $\boldsymbol{J} \in \mathbb{R}^{m \times 2}$ is configuration-dependent task Jacobian matrix, and $\boldsymbol{v_i} = \dot{\boldsymbol{q}}$ is a velocity command computed for a i^{th} task. Then we can define desired outcomes for each task as $\boldsymbol{\sigma_{i,d}}$ and system configuration needed to achieve that desired state as $\boldsymbol{q_d}$. This results in a following equation that we want to solve for each task: $\dot{\boldsymbol{\sigma}}_{\boldsymbol{i,d}} = \boldsymbol{J}(\boldsymbol{q_d})\boldsymbol{v_{i,d}}$. We can do it by finding a least-squares solution: $\boldsymbol{v_{i,d}} = \boldsymbol{J}^{\dagger}\dot{\boldsymbol{\sigma}}_{\boldsymbol{i,d}} = \boldsymbol{J^T}(\boldsymbol{J}\boldsymbol{J^T})^{-1}\dot{\boldsymbol{\sigma}}_{\boldsymbol{i,d}}$, where $\boldsymbol{J}^{\dagger}$ indicates pseudo-inverse of $\boldsymbol{J}$. However, bluntly applying this solution would result in numerical drift, when integrating the velocities. In order to avoid it we used closed loop inverse kinematics version of the abovementioned solution: $\boldsymbol{v_{i,d}} = \boldsymbol{J}^{\dagger}(\dot{\boldsymbol{\sigma}}_{\boldsymbol{i,d}} + K_i(\boldsymbol{\sigma_{i,d}} - \boldsymbol{\sigma_i}))$, where K_i is a positive constant and $(\boldsymbol{\sigma_{i,d}} - \boldsymbol{\sigma_i})$ is task-specific error. Ultimately we can project the desired velocity of a task i onto the null space of immediately higher-priority task by multiplying $(\boldsymbol{I} - \boldsymbol{J_k^{\dagger}}\boldsymbol{J_k})$ with $\boldsymbol{v_{i,d}}$, where $k = i - 1$.

In this project robots had to accomplish three concurrent tasks, which were, in the order of decreasing priority, as follows:

1) *Obstacle avoidance* - Robots are supposed to stay away from any obstacle at a distance of $0.4m$. This task has the highest priority because to fulfill other tasks robot has to preserve its structural integrity. First, the distance from the robot to all the points in the environment mapping is calculated. If the closest point is less than $0.4m$, then it is treated as an obstacle, that needs to be avoided immediately. This tasks is specified as

$\sigma_1 = \|\boldsymbol{p} - \boldsymbol{p_o}\|$, where $\boldsymbol{p}$ is robot's position and $\boldsymbol{p_o}$ is obstacle's position. The desired minimum distance is described as $\sigma_{1,d} = 0.4m$. The resulting Jacobian matrix, which is a "unit vector aligned with the robot-to-obstacle direction" (Carloni, 2022, p. 15) is shown in Equation 6. However, the obstacles are in the robot's frame of reference so $x, y = 0$. Provided that $\dot{\sigma}_{1,d}$ is 0, because $\sigma_{1,d}$ is a constant, the optimal velocity for this task becomes Equation 7.

$$\boldsymbol{J_1} = \begin{bmatrix} \frac{x - x_o}{\|\boldsymbol{p} - \boldsymbol{p_o}\|} & \frac{y - y_o}{\|\boldsymbol{p} - \boldsymbol{p_o}\|} \end{bmatrix} \quad (6) \qquad \boldsymbol{v_1} = \begin{bmatrix} \frac{-x_o}{\|\boldsymbol{p_o}\|} & \frac{-y_o}{\|\boldsymbol{p_o}\|} \end{bmatrix}^T K_1 \left(0.4 - \|\boldsymbol{p_o}\|\right) \quad (7)$$

In the simulation, we found $K_1 = 3$ to be the most optimal constant, while in the lab setting the $K_1 = 1.3$ seemed to give the best results, meaning that the robot was reacting fast enough to avoid collision with obstacle approaching it at a constant speed. With larger Ks the robot overshoot significantly, resulting in jerky spins around its center of mass. This was particularly noticeable with real robots.

2) *Follow me* - Once leg clusters are detected in the environment, the centers of each cluster are computer, and the program checks if the distance between any two legs is in $[0.15\ m, 0.25\ m]$ interval which corresponds to the optimal leg separation. If this is the case then such two legs are selected and we compute a centroid between them, which is to be followed by the closest robot of the swarm at a distance of around $0.5m$. Additionally, the leg pair position is stored at every iteration so that in the following iterations if two or more optimal pairs are detected, the one closest to the last saved pair will be selected. This task is defined as $\sigma_2 = \|\boldsymbol{p} - \boldsymbol{p_f}\|$, where $\boldsymbol{p}$ is the position of the closest robot in a swarm and $\boldsymbol{p_f}$ is distance from the legs. Then the Jacobian matrix, which is a "unit vector aligned with the robot-to-feet direction" (Carloni, 2022, p. 18), is shown in Equation 8. Again, points are in robot's frame of reference so $x, y = 0$ and $\dot{\sigma}_{2,d} = 0$, thus the optimal velocity for this task is given by Equation 9.

$$\boldsymbol{J_2} = \begin{bmatrix} \frac{x - x_f}{\|\boldsymbol{p} - \boldsymbol{p_f}\|} & \frac{y - y_f}{\|\boldsymbol{p} - \boldsymbol{p_f}\|} \end{bmatrix} \quad (8) \qquad \boldsymbol{v_1} = \begin{bmatrix} \frac{-x_f}{\|\boldsymbol{p_f}\|} & \frac{-y_f}{\|\boldsymbol{p_f}\|} \end{bmatrix}^T K_2 \left(0.5 - \|\boldsymbol{p_f}\|\right) \quad (9)$$

In simulation our program used $K_2 = 5$, but in lab setting we used $K_2 = 2.5$. For these values, the robots reacted to leg movements fast enough to maintain a distance of around $0.5m$ and not fall behind. When Ks were too large the robots were not able to follow legs in a straight line because of overshooting the error. This usually resulted in increasing the distance between legs and the robot. Since, only the closest robot is following the legs directly only one robot will compute $\boldsymbol{v_2}$, while the other two will use $[0\ 0]^T$ as optimal velocity for that task. The other robots follow the legs indirectly, by maintaining the formation.

3) *Formation control* - In this task, robots try to maintain a triangular formation, where they stay $0.5m$ away from its center and they are at $120°$ angle relative to each other. The center of the formation is computed using robots' positions in the world's frame of reference. The task in this frame is specified by Equation 10, where $\boldsymbol{p_i}$ is the position of the i^{th} robot and $\boldsymbol{p_c}$ is the centroid of the formation. However, to make this task feasible it has to be converted to each robot's frame of reference so that the task becomes Equation 11, where $i \in [1, 2, 3]$ indicates i^th robot's frame of reference and ${}^i x_c, {}^i y_c$ indicate centroid's position in that frame. Then we can specify the desired position at $0.5m$ away from the centroid as given by Equation 12

$${}^0\boldsymbol{\sigma_3} = \begin{bmatrix} {}^0\boldsymbol{p_1} - {}^0\boldsymbol{p_c} \\ {}^0\boldsymbol{p_2} - {}^0\boldsymbol{p_c} \\ {}^0\boldsymbol{p_3} - {}^0\boldsymbol{p_c} \end{bmatrix} \quad (10) \quad {}^i\boldsymbol{\sigma_3} = \begin{bmatrix} {}^1x_c & {}^1y_c \\ {}^2x_c & {}^2y_c \\ {}^2x_c & {}^2y_c \end{bmatrix} \quad (11) \quad {}^0\boldsymbol{\sigma_{3,d}} = \begin{bmatrix} 0.5 & 0 \\ 0.5\cos 120° & 0.5\sin 120° \\ 0.5\cos 120° & -0.5\sin 120° \end{bmatrix} \quad (12)$$

Then the final formula for the optimal velocity of the third task, where each row specifies velocities for one robot in its frame of reference, is: ${}^i\boldsymbol{v_3} = K_3 \boldsymbol{I_{3\times 3}}({}^i\boldsymbol{\sigma_{3,d}} + {}^i\boldsymbol{\sigma_3})$. In the simulation, we used $K_3 = 4$ and in the lab, we used $K_3 = 1.75$, in both environments we found the respective values to work the best. For other values, robots sometimes got stuck behind obstacles, while with these they tend to slide around them "pulled" by formation control. Moreover, at these values robots reacted fast enough to leg movement so that the formation could not be broken. Additionally, they are small enough to avoid overshooting the error and resulting jerky spins.

Using NSB control we can combine the velocities optimal for each task into one final velocity for robot i, using

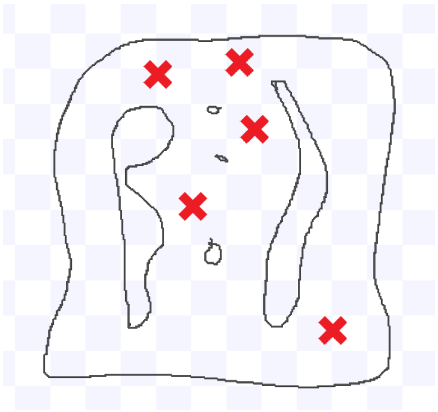


Fig. 7: Cluttered environment in Stage simulation. Red crosses indicate positions at which the SVM was checked and anylysed through `Rviz`. *Source: Own private archive.*

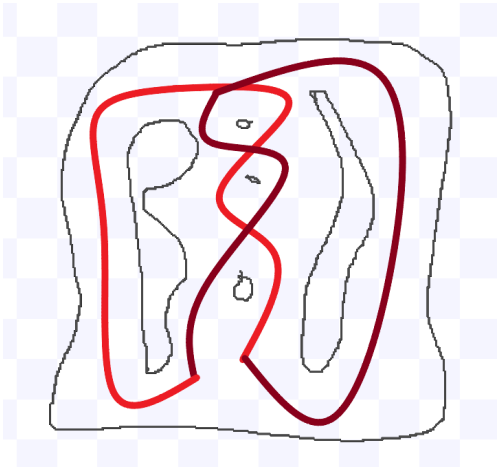


Fig. 8: Example path taken in a standard cluttered environment (in Stage). The two colors are used to avoid confusion in the center of the map. *Source: Own private archive.*

Equation 13.

$$ {}^{i}\boldsymbol{v_d} = \boldsymbol{v_1} + (\boldsymbol{I_{3\times 3}} - \boldsymbol{J_1^{\dagger}}\boldsymbol{J_1}) \left[\boldsymbol{v_2} + (\boldsymbol{I_{3\times 3}} - \boldsymbol{J_2^{\dagger}}\boldsymbol{J_2})\boldsymbol{v_3} \right]. \tag{13} $$

However, the velocities computed using this equation cannot be used directly. First, in order to make the robot rotate with a similar scaling as the linear velocity we use a scaling factor of 10, so that $\theta := 10 \times \theta$ (as given in the Assignment 2). Secondly, the "optimal" velocity can be impractical for a robot sometimes, therefore we restrict rotational velocity to $\left[-2\ \frac{rad}{s},\ 2\ \frac{rad}{s}\right]$ range, and linear velocity to $\left[-0.2\ \frac{m}{s},\ 0.2\ \frac{m}{s}\right]$ range.

V. Experiments and Results - [3 points]

Once the SVM model has been trained and reached sufficient performance on cross-validation, it was tested in multiple environments. The SVM trained on simulation data was first tested on the empty world shown in Figure 2a with one robot and legs. Then legs were moved around the robot and we verified on `Rviz` that whenever the legs are visible the markers indicating their centers are published, as in Figure 6. This is a baseline experiment as recognizing legs correctly is a necessary condition for the program to work. Thus, if an SVM model has not passed it, data was collected again, more diligently. A similar procedure was performed in case of SVM trained on real-life data, where the test environments contained no added obstacles.

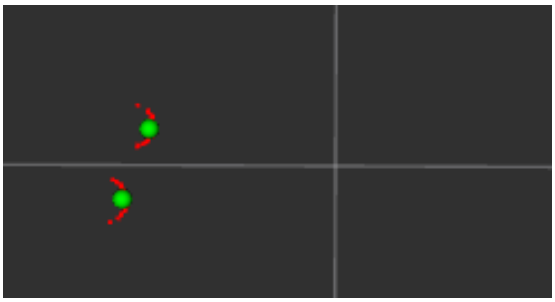


Fig. 6: Lidar mapping (red) of the empty environment with legs, clusters which are identified as legs are indicated with green markers. *Source: Own private archive.*

In the second stage of SVM testing, there were both legs and other objects in the environment. In the lab environment, apart from the legs, in the robot's sight, there were multiple other obstacles such as two rectangular towers. At first, they were arranged in a circle with a robot inside, but then their arrangement was changed multiple times. The robot was moving slowly ($0.05m/s$) in a circle around the center. When the SVM successfully recognized legs and dismissed other objects, it was remotely driven across the room maneuvering between various objects. If the SVM always detected legs and did not consistently recognize a single object as a leg it was deemed successful. We allowed rare errors, where some objects at certain angles were recognized as legs because they would be filtered out by our leg pair detector. In the case of simulation, a similar procedure was followed with a small change. The robot was placed in several positions on the map at first instead of actively changing obstacles in its range. The positions used for this stage of testing are shown in Figure 7. Similarly, the robot was then driven across the map taking the path marked on the Figure 8.

The NSB control was tested iteratively. Each task was tested one by one in both environments. The object avoidance task was tested in a setting with one robot and an obstacle. In a lab, the obstacle was a box that was

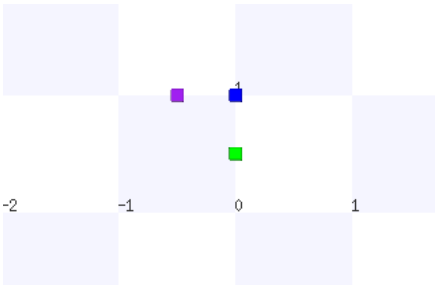


Fig. 10: Empty worlds with three robots and no legs. *Source: Own private archive*.

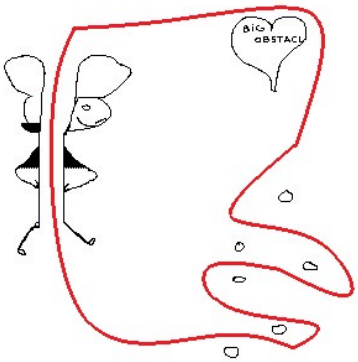


Fig. 11: Example path taken in a custom cluttered environment (in Stage). *Source: Own private archive.*

slowly moved towards the robot and in simulation, a wall was pulled towards the robot. It was repeated multiple times each time changing the alignment between the obstacle and robot. During these tests, multiple K_1 values were tested. An example test can be seen in the Figure 9. The robot has always avoided the obstacle, thus this task was deemed fulfilled.

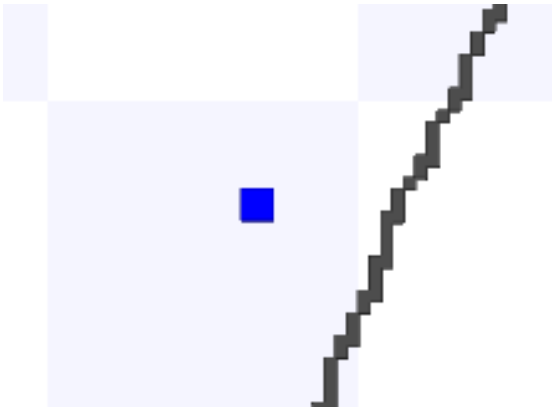


Fig. 9: A setting where a single what is pulled towards a robot. *Source: Own private archive.*

The follow-me task was tested by placing a single robot and robot legs in a cluttered environment. In the simulation, the standard map was used. While in the lab, rectangular towers, which could easily be misclassified as legs, were placed. The point of these settings was to make sure that the robot is always following the correct leg pair and not some non-existent legs. Then the robotic legs were remotely driven around the environment. In the case of simulation, an example path can be seen in Figure 8. When the robot consistently followed the robotic legs at a distance of around $0.5m$ the task was deemed fulfilled. Again, during this phase of testing, multiple K_2 values were tried. The formation control task was tested by placing three robots in an empty world in simulation (shown in Figure 10) and in the case of the lab, no obstacles were added. Then robots were put farther away from each other so that when the program was started they would have to form a triangular formation. When this behaviour was shown, one of the robots was remotely controlled so that the other would have to follow it to maintain the formation. The controlled robot was steered both away from the centroid and towards it to try to break the formation. If at all times the approximate formation was maintained the program passed the test. During these tests, multiple K_3 values were tried.

Once the program passed all the previous tests the overall control could be tested. This was done in two stages. In the first one, legs and three robots were placed in an empty environment (both in simulation and lab). Such a trivial setting allowed us to check whether the follow me and formation control tasks work correctly. In this stage, the legs were remotely controlled and the formation of 3 robots was supposed to follow it. Then the legs were steered closer to one of the other robots to check that the closest robot is selected correctly and the legs are followed directly by a different robot while the other two follow them indirectly. The last part of this stage involved steering the legs into one of the robots so that it had to evade and the whole formation would have to follow. Once this stage was completed multiple obstacles were added to the lab environment. They were supposed to be avoided by the robots following the legs. The obstacles were sparsely arranged at first, but various arrangements were tested, for instance, a singular object, a tight cluster of objects, or a ”gateway”. Then the legs were steered around obstacles to check if they were followed properly, avoiding collisions, while maintaining the formation. In the simulation environment,

the legs followed the path on one map shown in Figure 8. However, to check more situations a second map was created and a pathway across it was traversed in both directions to ensure correct behaviour. An example pathway can be found in Figure 11. When the robots successfully followed the legs, avoiding obstacles at all times, and maintained formation when possible along both pathways the NSB control was deemed successful.

REFERENCES

Antonelli, G., Arrichiello, F., & Chiaverini, S. (2008, 01). The null-space-based behavioral control for autonomous robotic systems. *Intelligent Service Robotics*, *1*, 27-39. doi: 10.1007/s11370-007-0002-3

Carloni, R. (2022). *Robotics practical II.* Retrieved 2022-06-019, from `https://nestor.rug.nl/bbcswebdav/pid-11309573-dt-content-rid-41194188_2/courses/WBAI030-05.2021-2022.2B/week3-introduction.pdf`

Chung, W., Kim, H., Yoo, Y., Moon, C.-B., & Park, J. (2012). The detection and following of human legs through inductive approaches for a mobile robot with a single laser range finder. *IEEE Transactions on Industrial Electronics*, *59*(8), 3156-3166. doi: 10.1109/TIE.2011.2170389