Robotics Tutorials Report: Following Robot using PID controllers and Subsumption Architecture

Clara Maine (s1032005): clara.maine@student.ru.nl
Arne Wittgen (s1034858): arne.wittgen@student.ru.nl

Task One: Initial Steps

Initial exploratory activities in CoppelliaSim mostly involved testing the available mobile robots such as . I didn't know how scripts worked, so I just observed the built-in demo functionalities the robots had. In order to learn the basics of the software, I followed the Tutorial CoppeliaSim (V-REP) English by Leopoldo Armesto, and watched all videos up to the Line Tracking with Obstacle Avoidance Demo. At that point I had a handle on how CoppeliaSim functioned, but was still not very sure how to use scripts to create a PID controller. My final reference was the blobDetectionWithPickAndPlace.ttt demo scene provided by CoppeliaSim. This provided a basis for how to extract information from a vision sensor and locate brightly colored vision blobs. After attempting to setup a remote python API, I determined it would be easier to just work with the sensor data directly in Lua, and the language barrier didn't prove to be too difficult to get over.

Task Two: Proposal

For this assignment, I have constructed a single scenario which contains the requirements for both tasks using two separate dr20 robots. Therefore, the environment design and robot choice will be the same for both tasks.

Subsumption Architecture in the Leader

The first requirement, implementing a subsumption architecture, will be satisfied with the "Leader" robot. This robot will use its proximity sensors to randomly explore its environment. My plan for this is to have it work in a grid-based system, where at each junction it can choose to continue straight, turn left or right, or reverse to the previous intersection the robot would need to choose a path unobstructed by obstacles.

This would fulfill the subsumption architecture requirement, since the robot would have no prior knowledge about the layout of the environment and has a set number of behaviors in a hierarchy of importance (highest priority is avoiding obstacles, second priority is choosing a direction). The benefit of the subsumption architecture here is that this task can be used in various environments. You shouldn't need a complex environment to implement a simple Follow the Leader task, so a subsumption architecture is perfect for this.

PID Controller in the Follower

The second requirement will be used in the "Follower" robot. As its name suggests, this robot will be tasked with using its various sensors to follow the Leader wherever it may go. The error signal which will be assessed and minimized using the PID controller will be the distance from the Leader robot. There will be a set optimal distance, so the Follower robot is not always bumping into the Leader, and so if the leader reverses, the follower will also reverse to keep that optimal distance.

In more specific PID terms, the reference angle (where the robot wants to be) will be the Leader's current position – the optimal distance. The Measured value is the actual distance from the Leader (calculated using the ultrasonic proximity sensor or possibly the vision sensor). By using the PID controller to minimize the error (the difference from the reference angle and the measured value), the follower robot will

The follower robot should attempt to mimic the leader robot's movements as well as possible. One notable difference might be seen when the leader makes a sharp turn, the follower robot's turn would be a bit more curved, since the follower robot is only concerned with the distance to the Leader robot, and not exactly replicating the path it took.

Environment Design

The environment design I anticipate for this task will be quite simple. The example scene will involve a flat floor and a series of obstacles (basic geometric shapes such as the cuboid object) to test the Leader's obstacle avoidance function. In theory, these two robots should be able to function in any environment where the robots can freely move around. In terms of the grid-based navigation of the Leader, if the robot is heading straight and runs into an obstacle before reaching the next "intersection" then the robot will consider it's current position to be an intersection and randomly choose a direction (taking care to not choose the direction in which an object is blocking its path).

Choice of Robot: dr20

The robot I will be choosing is the dr20 model from CopelliaSim's standard options. I chose the dr20 because it is a mobile robot, and can therefore fulfill the requirement of 2-D movement along the floor for both the follower and leader robot, and it also has many types of sensors: a wision Sensor, a force sensor, an ultrasonic proximity sensor, and 180 degrees of infrared sensors. I chose this robot over the simpler dr12 because although I believe this task will only require the force and ultrasonic proximity sensors, the vision and infrared sensors are a nice option in case I wish to expand the capabilities of this task.

I went with the dr20 over the other options mainly for simplicity. The other mobile robots I worked with had too many confusing features or were too large and clunky for the task I had in mind.

Task Three: Implementation

Final Task Design

The final design which was decided upon for this project involves a combination of the initially proposed ideas, but also a thorough rework of the subsumption architecture. Whereas before the subsumption architecture and PID controller were going to be implemented in two separate robots, the final task involves one robot using both. The robot of interest—which this report will refer to as the “follower”—uses a subsumption architecture with a hierarchy of control managing different levels of function and priority. The lowest layer of the architecture covers collision avoidance, the higher layers involve following a bright green object at a certain distance and wandering around the environment. This combination of the PID controller and subsumption architecture allowed us to focus on creating a more robust follower robot, since the previous implementation would have not added collision avoidance to both robots. Also, this allowed us to do the implementation purely in Lua, the reasons for which are discussed in a following paragraph.

As proposed in our initial ideas, we are using a modified dr20 for our implementation. The original plan was to make use of its already present sensors, but since especially our plan for the subsumption architecture changed a lot, we added two proximity sensors and a camera for the PID controller.

In our example scene, we use the default dr20 provided by Copelliasim as the target, modified by adding a renderable green cuboid which is carried around. The addition of this robot is purely to test the functionalities of the follower robot, and is not necessary for the function of either the subsumption architecture or PID controller.

If the robot is to be tested in a different environment, a green cuboid must be present to test the PID and the environment should also be confined by walls or something similar, as the follower can have trouble finding the target otherwise.

We ended up implementing the robot solely in Lua, as using the Python API proved too complicated when unpacking sensor data. We found the API documentation and general instructions for CoppeliaSim difficult to understand, which made working with the API more focused on figuring out how things work rather than actually implementing the robot’s behaviour. Additionally, the original subsumption architecture design became very difficult to implement in the context of the API because the return values from the API functions were difficult to work with. The originally planned subsumption architecture (based on Arne’s proposal) required computation that was much more complex than originally anticipated. So, our final design included the PID and subsumption architecture implemented in one robot using non-threaded child scripts written in Lua.

PID Controller

Implementation

There are two PID controllers implemented in the following task. This task uses a detection strategy called blob detection where it localizes a bright green "blob" in the robot's field of view and attempts to follow it. In order to build a functioning follower robot, the controller had to account for two central sources of error.

The first PID controller we built used a value given by the vision sensors which calculated the size of the blob, `currentblobsize`. After some trial and error, we determined the optimal value for `targetblobsize` which would allow the robot to follow at a manageable distance. Using basic rules of perspective, if `targetblobsize` is smaller than `currentblobsize` (as illustrated in Figure 1 below) with a negative `blobsize_error` value we know that the follower is too close, and needs to reduce its speed. The opposite goes for a positive `blobsize_error`, which indicates the follower should speed up.

Using only `blobsize_error`, the follower robot worked well when following the leader in a straight line, but it was unable to turn quickly enough to effectively follow non-linear paths. So, a second PID controller was added to correct the horizontal distance, `horizonal_error`. This error value was found by calculating the distance of the blob from the center of the visual field (illustrated in Figure 1). Using these two PID controllers, the robot corrects its movement to keep the target in the front center of its vision sensor while also following it at an acceptable distance.

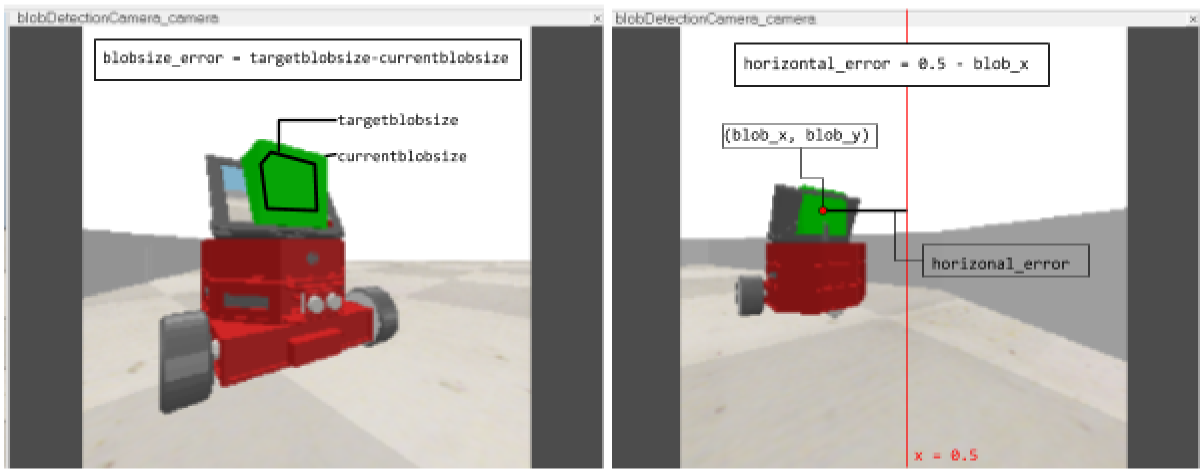

blobDetectionCamera_camera
blobsize_error = targetblobsize-currentblobsize
targetblobsize
currentblobsize
blobDetectionCamera_camera
horizontal_error = 0.5 - blob_x
(blob_x, blob_y)
horizonal_error
x = 0.5

Figure 1: Illustrating the two sources of error which are corrected by the PID controllers

The first PID controller (controlling the size of the blob in the visual field) uses all three coefficients. It had a relatively high proportional value (15) because the values for `targetblobsize` and `currentblobsize` are quite small, and in order to transform this to an error correction that will actually cause any difference in the motors, you need to give it that high proportional controller. The derivative coefficient's initial value is 0.1, which helped to smooth out the follower's changes in speed. Finally, the integral coefficient is equal to 0.05, which is then multiplied to a variable called `pidCumulativeErrorForIntegralParam` and added to the final `ctrl` value of the PID controller, which is applied to the velocities of the left and right wheels.

The second PID controller (controlling the horizontal distance) only uses the proportional (0.2) and derivative(0.01) coefficients, which combine to add to the final `xCtrl` value of the controller. This value is then added to the final corrected velocity of the left wheel and subtracted from the right. Therefore, if the blob is left of center the xCtrl will be a negative value and the right wheel will speed up to shift the visual field further in the left direction.

Performance

The PID controller works quite well in a controlled setting, but of course can become inaccurate due to noise. We noticed that the vision sensor data (the blob location and size) was much more sensitive to noise than the output motor data (with noise applied to the velocities of the wheels). This is probably because the smallest changes to values from the vision sensor may cause drastic corrections made by the PID controller; and those values operate on a smaller scale than the wheel velocities. Ultimately, we set the standard deviation value for the vision sensor noise to 0.01 and the std for the motor output noise to 0.03.

Additionally, the PID controller is essentially useless when there is no green blob detected. If the leader robot moves out of the range of the follower's vision sensor, the robot has no way of correcting the error it cannot see. We solved this problem by implementing a subsumption architecture that could relocate the leader robot and the green blob if they were lost.

Subsumption Architecture

Behaviours

The subsumption architecture consists of three layers which are individually concerned with object avoidance, following a target at a constant distance and wandering around (as shown in Figure 2).

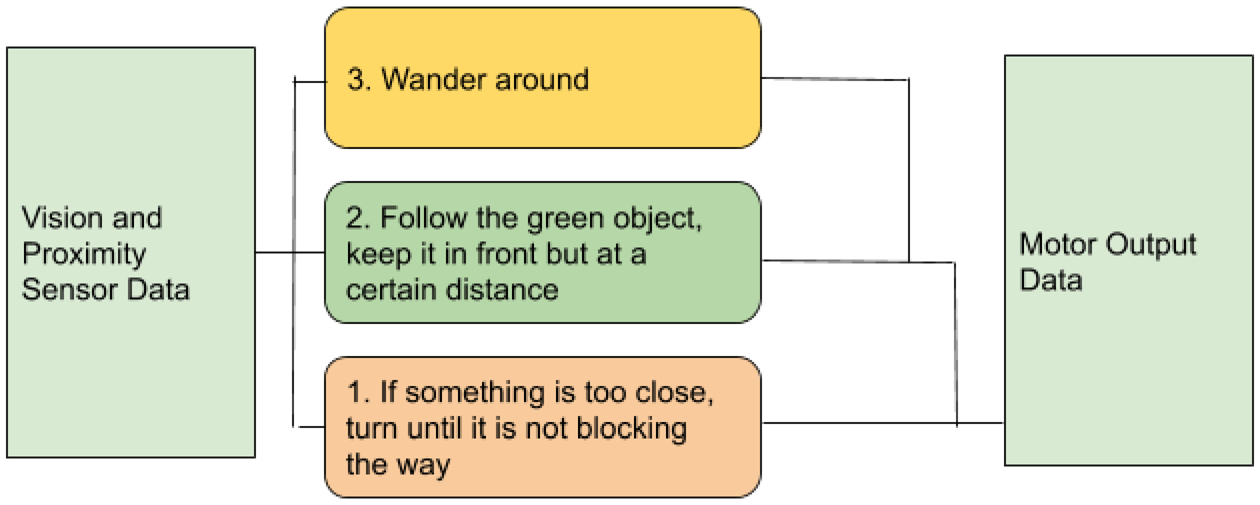

Vision and Proximity Sensor Data
3. Wander around
2. Follow the green object, keep it in front but at a certain distance
1. If something is too close, turn until it is not blocking the way
Motor Output Data

Figure 2: Visual Model of the follower’s Subsumption Architecture

The lowest layer, and therefore with highest priority, is object avoidance. This is a relatively rudimentary design: if the robot detects a target, it turns around until it is pointed away from the obstacle, and then continues with its other behaviors.
The middle layer concerns following a target, making use of our PID controller. If the follower robot detects a fitting target with its vision sensor, it will follow it at a constant distance.
Lastly, the highest layer is wandering around in the environment. This is done at random, and the robot switches between turning left, turning right, and going straight ahead (until one of the other layers becomes active).

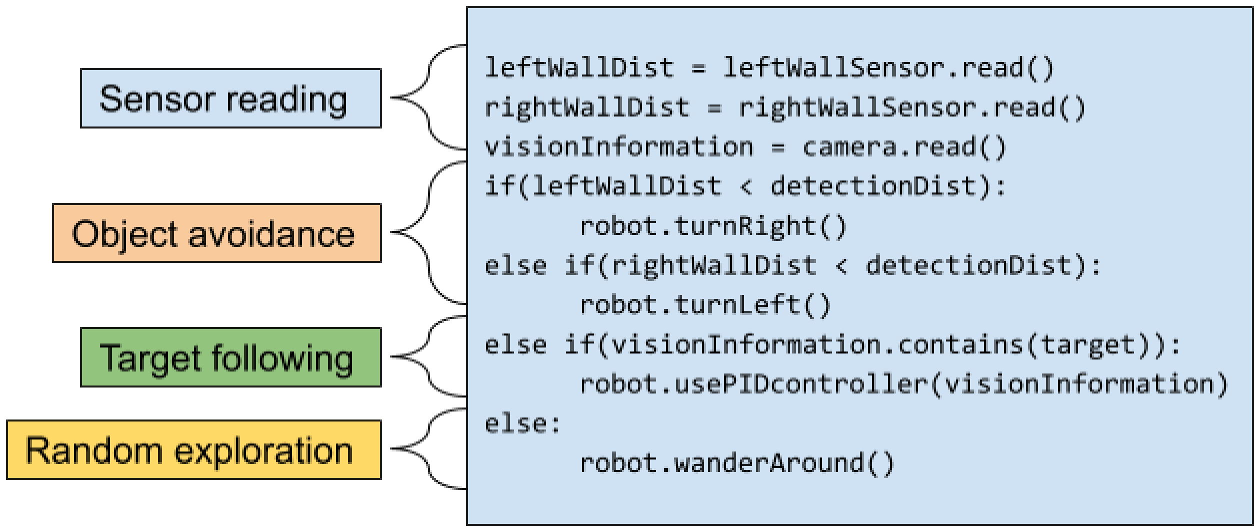
This behaviour is more efficient than just turning on the spot until the target is found again, as the target has most likely moved out of the follower’s vision range. Still, the top layer is quite rudimentary since the robot is now exploring the environment randomly which is not very efficient for finding the target again. A better implementation would require estimating where the target could be based on previous sensor data. Also, the current behaviour is not optimal in a large open environment, as there is a chance that the follower moves in a completely different direction from the target.

The first two layers (object avoidance and target following) are reactive, as they are triggered by sensor input and the following motor commands are entirely based on what is sensed.
The third layer is deliberative, as it doesn’t require sensory input to function. Still, it chooses its actions randomly rather than properly computing the best direction to travel in, as we found that implementing that solution was not feasible given the time constraints for this report.

Implementation

The following implementation will be explained with python-like pseudocode, to give a general understanding of the algorithms being used without being limited by actual language restrictions. Also, the pseudocode will leave out finer details of the computations as to only explain the underlying functionality of the subsumption architecture; Details will be given as the separate behaviors are explained.

After reading the sensor data, the currently active layer is chosen via if-else statements, with the layers being checked in order of their priority (as shown in Figure 3).



```
leftWallDist = leftWallSensor.read()
rightWallDist = rightWallSensor.read()
visionInformation = camera.read()
if(leftWallDist < detectionDist):
      robot.turnRight()
else if(rightWallDist < detectionDist):
      robot.turnLeft()
else if(visionInformation.contains(target)):
      robot.usePIDcontroller(visionInformation)
else:
      robot.wanderAround()
```

Figure 3: Abstract structure of the Subsumption architecture

To detect whether there is an obstacle in front of the robot, it uses the two proximity sensors. If the readings are below a certain threshold (which is equal to the maximum sensing distance), the robot will turn away from the obstacle by instructing the left wheel to move backwards and the right wheel to move forward to turn to the left, and vice versa to turn to the right. The wheel speeds have an absolute value of 2, as higher values cause the robot to turn further than it would have to. This virtually creates a while loop, since the robot will turn left or right every simulation step as long as the proximity sensors detect an obstacle.

If the robot recognizes the target in the vision sensor data, it will use the PID controllers to follow it (explained under “PID Controller” above). Lastly, if neither of these two layers are active, the robot will wander around. It does so by randomly choosing whether to turn left, turn right or go straight (with a probability of ca. 15%, 15% and 70%). To achieve this, it will only move the right (to turn left) or left (to turn right) wheel forward while keeping the other one still, or by moving both wheels forward at the same speed; In either case, the robot uses an absolute wheel speed of 7.

Performance

Overall, the subsumption architecture works as expected. It took some time to fine-tune the exact values for object avoidance and the probabilities for the random exploration, but current combination allows the layers to properly interact with each other. In an optimal environment (the robot sees the target from the beginning and there is little to no sensor noise), the target following layer is the only one active, since the target will already avoid walls and obstacles, and the robot rarely loses track of the target. When sensor noise increases, the wandering layer becomes more useful, as the robot is more likely to lose the target and the wandering is not influenced by sensor noise and only marginally affected by the motor noise.

The object avoidance layer is quite stable even with higher sensor noise, the reason for which lies in its design. Even with noise up to a standard deviation of 0.5, the robot still functions normally enough, which is impressive given that the proximity sensor operates in the range of 0-0.3. At this high noise, the robot starts to recognize obstacles quite late at times and has a lot of “hiccups” (as described later), but doesn’t fail completely. At a noise level of std = 0.25, the functionality is nearly indistinguishable from noise-free behaviour, based on visual inspection. Because the sensors only return a value when something is detected, noise can only influence its behavior once it detects something. With added sensor noise, the robot can either over- or underestimate how close an obstacle is. This can have two main effects on the behaviour:

1. Overestimation when the target is within detectable range and the robot is approaching it: The robot will falsely assume that the obstacle is not yet close enough, and will start turning away very late.
2. Overestimation when the target is within detectable range and the robot is in the turning process: This will result in a sort of “hiccup”, as the robot shortly goes straight ahead since it assumes that the obstacle isn’t within detectable range anymore. Once the noise disappears, it will resume turning.

Interestingly, underestimation does not have an effect on the robot’s behaviour: As the robot only returns a distance value once an object is measured and the threshold for turning is the size of the sensor, they will never assume that an obstacle is detected when it is still a bit further away. Similarly, underestimation while turning does not have an effect since the value still stays under the threshold.

The robot is also quite stable in the presence of motor noise: Starting with the layer for random movement, the movement becomes more unpredictable and includes more turning around instead of going straight as the noise increases. This makes sense, since now the command for going straight rather becomes a command for big or small turns, given that the wheel speeds are not going to be exactly the same anymore. Still, the robot manages to cover a reasonable amount of the environment.
As for object avoidance, the main difference here is that the robot takes longer to turn away from an obstacle. Here, the reason is that the robot does not consistently turn away from the wall, but slows down or even turns the opposite way at times.

This behaviour is tolerable up to a standard deviation of 5, which is a little under its maximum speed of 7. At this point, the functionality is severely impacted but not completely disabled.

Appendix

To give an overview of our code, and which values to change to add artificial noise, the following section contains a brief manual for our robot.

Code Structure

The robot's code can be found in the blobDetectionCamera script of the robot named "follower". The function `sysCall_actuation()` handles the subsumption architecture and basic computations for object avoidance and wandering around. The PID controller and noise generation are handled in the functions `PID_controller()` and `norm_noise()` respectively.
To add artificial noise, the four parameters on lines 32-35 can be changed. `noise_std_motors` and `noise_std_vision` are for the PID controller, `noise_std_sensors` and `noise_std_motors2` for the subsumption architecture (the two different motor parameters are needed as they operate on different scales).
Noise generation is done with a normal distribution centred around a mean equal to the sensor reading or motor speed, with a standard deviation equal to the value set in the parameters.
To severely impact, but not completely disable the robot's functionality, the noise parameters should be up to the following values, based on our testing:

- `noise_std_motors` <= 0.05
- `noise_std_vision` <= 0.015
- `noise_std_sensors` <= 0.25
- `noise_std_motors2` <= 3