Swarm Intelligence Team Project Simulation: Effects of Environment on the Swarm Behaviour

Tino Liebusch Florian Bannier Maximilian Kühne

August 9, 2016

Advisor:
Christoph Steup

Contents

1 Introduction

The paparazzi package provides the user with a lot of functions to control and work with the quadrocopters. The current simulation can run some copters and get them in to cluster behaviour.
Task of this project was to make the VREP simulation much more realistic to environmental and also copter-based influences. The following chapters depict, how the copter influences the wall of the arena, when it flies by, how the ultrasonic sensor work and affect each other in a multi-copter flight and the implementation of a omnidirectional virtual ranging sensor.

2 Simulation of the Arena Walls

2.1 Description

This includes a sufficient representation of the SwarmLabs arena, the environment in which FINken robots mainly operate in.
For now, they operate at a fixed height of 60cm. This allows us to focus on the arena walls and to severely simplify them in our simulation model.

2.2 Challenges

The walls consist of an inner net and an outer sheet of plastic and behave essentially like a curtain. The robots influence these indirectly through movement of air.
V-Rep provides us with rigid body physics, but it is neither capable of simulating cloth nor fluid dynamics. A severe simplification of our simulation model is therefore necessary.

In addition to that, collecting accurate real world data on air flow and movement of the walls is hard and evaluating the simplified simulation model is even harder.

2.3 Realization

The assumption of a fixed altitude allows us to only require accuracy at this altitude. Furthermore, we decided to focus on Bernoulli's principle as the main influence on the walls. Bernoulli's Principle causes the walls to be attracted to the robots whenever they get near them (the FINken accelerates air, the faster moving air causes a zone of lower air pressure).

The necessary model can be realized very cheaply as a row of long, narrow boxes (called segments) on hinges. For each of these segments, a script calculates the distance between the segment and each quadcopter in the scene. The script configures the joint motor of each segment's hinge depending on the smallest calculated distance.
We make use of the joint's PID position controller to approximate an amplitude calculated using the following formula:

$$f(x) = k(3.3x^3 - 7x^2 + 4.8x)$$

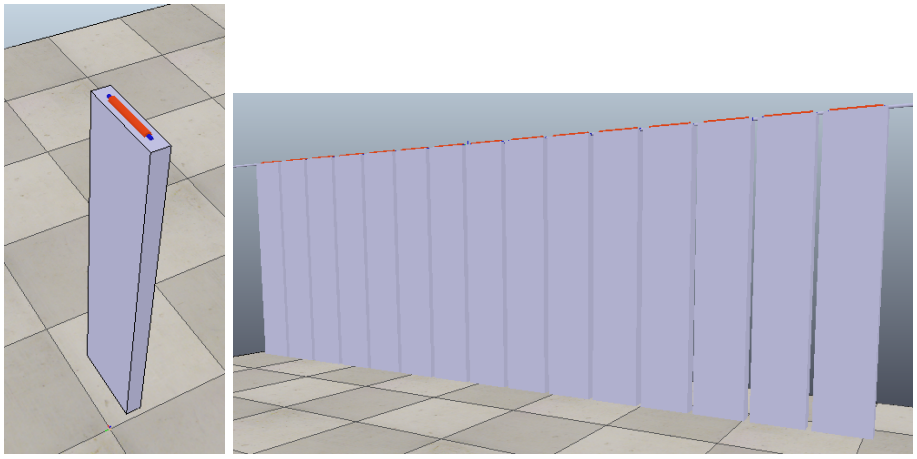
$$x = 1 - distanceinmeters$$

$$k = maximumamplitude$$

This formula is a result of observations using the arena's ceiling camera. It approximates the following relationship between the robot's distance and the segment amplitude:

- distance > 1m: no visible amplitude due to Bernoulli forces
- 1m > distance > 0.5m: approximately proportional relationship between distance and amplitude
- 0.5m > distance: amplitude is at maximum / barely increases with decreasing distance

Because simulating many segments strongly affects our performance, we deemed it necessary to make the number of segments variable. Therefore, the segments are generated dynamically at simulation start.



(a) Single Segment (b) Few Segments

Figure 1: Example of Wall Segments for the Arena in the VRep Simulation

2.4 Usage Guidelines

2.4.1 Implementation

- Insert a dummy object into the V-REP scene.
- Apply a non-threaded child script and copy and paste the Lua code from "finkenWalls.lua".
- At the beginning of the initialization section, add the object handles of our FINkens to the FINkenHandleList.
- Set script parameters according to the next step.

2.4.2 Configuration

The following script parameters have to be configured:

- numSegments: Number of wall segments. Increase for precision; decrease for performance (recommended value: 15-20)
- wallWidth: total width of the generated wall (arena dimensions: 3m x 4m)
- segmentMass: mass of the wall segments in kg (recommended value: 40)
- jointForce: force of the joint motor when it is PID controlled (recommended value: 15)
- threshold: max distance at which bernoulli force is noticable (should always be: 1)
- friction: joint friction, which is implemented as joint force when the PID is off (recommended value: 3)
- maxAmplitude: maximum amplitude that the wall achieves when the distance to the nearest finken is 50cm or less (recommended: 30)
- PID_P: Proportional gain of PID controller (recommended: 1)
- PID_I: Integral gain of PID controller (recommended: 2)
- PID_D: Derivative gain of PID controller (recommended: 1)

3 Error Implementation Ultrasonic Sensor

3.1 Description

The realistic ultrasonic sensor can be influenced by each other. This can be a problem in regard to the distance measurements. To analyse this behaviour a few tests have been made with the quadcopter. After the evaluation of the collected data it was clear, that the ultrasonic sensor mostly didn't depict the real distance.

We identified two different types of error:

- the distance is usually to short than real,
- there is "cross talk" between the sensors of multiple copters.

The log files from the collected data best fitted a Rayleigh distribution. The following picture is an example for this kind of distribution[1]:

[1] http://www.mathworks.com/help/examples/stats/ComputeAndPlotRayleighDistributionPdfExample_01.png

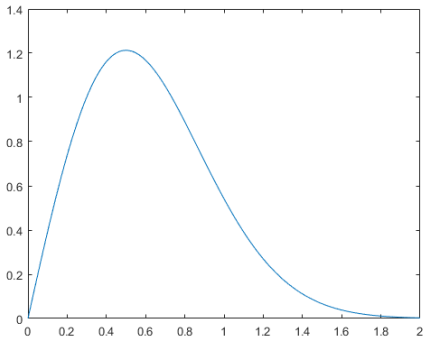


Figure 2: Rayleigh-Distribution

The ultrasonic sensor scans its surroundings at a fixed frequency. The measurement is based on the time the signal needs to be reflected back to the sensor. When there are two or more sensors sending at the same moment it comes to "cross talk" and a false distance will be measured. This "cross talk" depends on the angle between two sensors and how the copters move.

In Lua there is only the possibility to work with normal distributed values, so the function has to be modified. How this was made and how users can work with this is explained in the next part.

3.2 Realization

The implementation of this behaviour in the simulation was dependent on two requirements. The "cross talk" didn't take place all the time and it also influences only one sensor. The second requirement is the inaccuracy of measurements, which is based on the Rayleigh-distribution. This error occurs constantly, when ever there is no cross-talk error. To calculate, the function of the Rayleigh-Distribution was implemented with the aid of the inversion method. This enables the use of the normal distributed values given by Lua.

When the first variable is smaller or equal to 67, which is equal to a probability of 68%, "cross-talk" will be simulated in our simulation. The cross-talk is simulated by halving the real distance. If the first random number is above 67, the real distance will be accumulated with an error using the customized distribution function.

This filter is located in the *getErrors*-function. It takes the measured distance from the virtual sensor, which is the perfect value, and overwrites it with the new, other distributed and calculated distance. First of all one can change the deviation of the distribution like this[2]:

[2]https://de.wikipedia.org/wiki/Rayleigh-Verteilung/media/File:Rayleigh-verteilung.svg

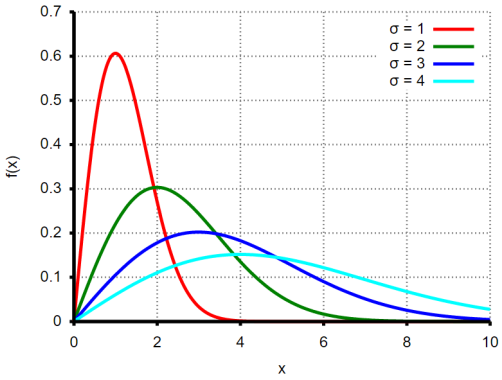


Figure 3: Rayleigh-Distribution with different densities

The function for the Rayleigh-Distribution looks like the following:

$$f(x) = 1 - e^{-\frac{x^2}{2\sigma^2}}$$

After the using of the inversion method the new function looks like this:

$$f(x) = \sqrt{(-1) * \log(u) * (2 * \sigma^2)}$$

The variable u stands in this context for the calculated normal-distributed random value. To work with the changed function it has to be reduced by the the deviation:

$$s(x) = \sqrt{(\frac{4-\pi}{2}) * \sigma^2}$$

At the end of this filter the calculated values will be compared to a minimal value of 0.2 an 3.0 as maximum. Every number in this interval will be accepted, else one of this borders will be handed over. If needed users can uncomment calculations for the bottom sensor. The probabilities can also be changed with setting an other percentage.

4 Realistic Virtual Ranging Implementation

4.1 Description

The goal of this task was to implement a realistic, omnidirectional, proximity sensor. The sensors requirements are:

- Measurements are not influenced by occlusion
- Only objects with a sensor equipped can be measured
- The direction of the unit is not measured
- Proximity is measured between 0-250 cm
- The accuracy must match the real sensor.

In the following this sensor will be called "virtual ranging sensor" and the method will be called "virtual ranging".
The sensor was implemented in the FINken simulation environment in V-Rep Ver. 3.2.2. FINken is the name for the quadcopter, which are used in the swarm research of the OvGU SwarmLab.
The setup of the FINken is not being further discussed in this document, other than them being quadcopter with the necessary equipment to perform virtual ranging via radio signals. For further reference, how the sensor and quadcopter work in reality, please contact the team in charge of the FINken.

4.2 Sensor Implementation

For an efficient and flexible usage of the sensor, not only on FINken, but also on other equipment (i.e. heterogeneous swarms), the virtual ranging sensor was designed modularly. As a base object in the V-Rep environment a dummy object has been used. This dummy object can be applied to any object in the scene and will be measured. V-Rep provides a function, which will solely retrieve dummy objects, from which the virtual ranging dummies will be filtered. This is an additional advantage, since it reduces the amount of objects the routine has to process.
Each virtual ranging dummy object has a non-threaded child script added to it. We selected non-threaded child scripts, since threaded child scripts can cause the following problems[3]:

- they are more resource-intensive
- they can waste some processing time
- they can be a little bit less responsive to a simulation stop command.

The virtual ranging script has two methods:

1. Initialization
2. Actuation

The first method is being called when the simulation starts. The second step is being called twice per simulation step: from the main script's actuation phase and from the main script's sensing phase.
The virtual ranging works in both methods almost the same. The first method additionally contains some data and methods required for the error implementation, which will be explained later on and can be ignored at this point.
The general procedure of our virtual ranging algorithm is as followed:

[3]http://www.coppeliarobotics.com/helpFiles/en/childScripts.htm

```
Define virtual ranging dummy object name
Retrieve handle of current sensor
Create VRangeList string that will store distances to other objects
Define the threshold (how far the sensor measures in meter)

while there are dummy objects do
        if dummy object name is equal to sensor name, then
                if handle is equal to own handle then
                        Distance is 0
                else
                        Calculate distance between dummy objects
                        if distance is not valid then
                                Print Error
                        else if distance > threshold then
                                Distance is threshold
                        end
                        Add distance to VRangeList
                end
        end
end
Replace SimulationParameter "dist" with VRangeList
```

First all dummy objects are iterated and all virtual ranging objects are filtered. Next the distance between the current sensor and all other sensors is calculated. The distance between the sensor and itself is defined as 0. The default threshold is 2.5m. Distances over the threshold as well as the distance to itself are fixed values and are not accumulated with the simulated error.

All distances are stored as string in VRangeList divided by a space. This format can be easily iterated in LUA using a for-loop like:

```
for dist in string.gmatch(vRangeList, "%S+") do
        <put your code here>
end
```

Once the distance to all virtual ranging dummy objects has been calculated and stored in VRangeList, the VRangeList will be stored in the simulation parameter "dist". To retrieve the distances to each object from a given FINken only the suffix of the FINken is necessary, as shown in the following code example:

```
local virtualRangingScriptHandle =
        simGetScriptAssociatedWithObject(
        getObjectHandle("Sensor_virtual_ranging", suffix))
local vRangeList = simGetScriptSimulationParameter(
        virtualRangingScriptHandle, "dist")
```

The method getObjectHandle is a custom made function that combines a name and a suffix and then uses the VRep function simGetObjectHandle to retrieve the handle. This is necessary since VRep's object naming convention are not following a trivial pattern. The virtual ranging dummy objects will be called:

```
Sensor_virtual_ranging
Sensor_virtual_ranging#0
Sensor_virtual_ranging#1
Sensor_virtual_ranging#2
...
```

The getObjectHandle method is then defined as:

```
function getObjectHandle(name, suffix)
        if suffix and suffix ~= "" then
                name = name .. "#" .. suffix
        end
        return simGetObjectHandle(name)
end
```

4.3 Error Model

In order for the virtual ranging sensor to be realistic, it was necessary to add an error to the distance measurement between two virtual ranging dummy objects. The real world error was modelled as a Bayesian mixture model by Simon Parlow, which kindly provided his model for this project.

A Bayesian mixture model is a collection of K distributions. In this case the distributions are all normal distributed. The Bayesian mixture model allows to draw realistic distance samples, depending on the real, not error-prone distance, between the two objects. Each distribution has a weighted influence on the final error. These weights are updated depending on the real distance.

The following formula shows the sampling.

$P(\theta|X) = \sum_{i=1}^{K} \phi_i N(\mu_i, \Sigma_i)$

The formula samples each distribution described by μ^i and Σ_i. And multiplies the sample with the according weight φ_i of the distribution. All weights are normalized and sum up to 1. Therefore, adding all samples multiplied by these weights leads to our desired realistic distance.

The error model provided by Parlow contains several mixture models which differed in the amount of components K. This means each model has a different number of normal distributions. Since the true distribution of the error is only approximated by the mixture model, a higher number means a better fitting. However, since each component has to be sampled a high number of components also means higher processing costs. In the choice of the best mixture model we accepted Parlow's suggestion to use the model with 100 components, as it is the best fitting model, which is neither over fitted, nor too general.

In order to use another mixture model, it is necessary to read the several values from Parlow's application. To run it the python libraries: sklearn, pypr, cPickle and numpy are required as well as any requirement of these libraries. Additionally, the plot functions in his code need to be disabled or the matplotlib is also required.

To change the model it is necessary to retrieve the following values from the given .gmm-file and replace them in the VRep virtual ranging script:

- conv_means (called "mein" in the VRep virtual ranging script),

- conv_covars (called "covar" in the VRep virtual ranging script),
- conv_weights (called "weights" in the VRep virtual ranging script),
- cond_means (called "cond_mean" in the VRep virtual ranging script),
- cond_covars (called "cond_covar" in the VRep virtual ranging script).

All these values are independent on the given distance. A print command for these values has to be written in the gaussian_mixture.py file in the get_conditional_samples method in order to retrieve them.

To run the method:

1. Run RangeGenerator.py in a python shell
2. Create a GaussianMixtureModelGenerator with the path to the new .gmm-file and the noise parameter 1 (i.e. model = GaussianMixtureModelGenerator("GMM_100_diag.gmm",1) if the .gmm is in the same folder as the python script)
3. Call calculate_range method with the mixture model and any distance 0-2.5 (i.e. GaussianMixtureModelGenerator.calculate_range(model,1))

The extracted parameters are hardcoded in the child script of the virtual ranging script and can easily be replaced by the new values. There is no change required other than updating all the values, which were named above.

Generally, it is not recommended to change the mixture model unless it is necessary due to performance optimization.

4.4 Error Implementation

In order to sample the Bayesian mixture model depending on the true distance X it is necessary to first update the weights. In order to do this, we multiply each weight with the result of the probability density function (pdf), using the mean and variance of the related distribution and the true distance X.

$$PDF_i(X) = \frac{1}{\sqrt{2\pi\sigma^2}}\, exp\left(-\frac{X-\mu^2}{2\sigma^2}\right)$$

$$newWeight_i = weight_i * PDF_i(X)$$

Subsequently it is necessary to normalize the weights again by dividing each weight by the sum of all weights.

$$normWeight_i = \frac{newWeight_i}{\sum_{i=1}^{K} newWeight_i}$$

We concluded the update process by following the numPy library. The resulting weights were exactly the same as in Parlow's application.

Before we can use the formula for the Bayesian mixture model, additionally a process is necessary to sample a normal distribution. For this the Box-Muller transform by George Edward Pelham Box and Mervin Edgar Muller was applied.

$Z = \sqrt{-2\ ln\ U_1} * cos(2\pi U_2)$

Using this method, it is possible to generate pairs of independent, standard, normally distributed (zero expectation, unit variance) random numbers, given a source of uniformly distributed random numbers U_1 and U_2. For a single sample only one number Z of the pair is needed.

LUA already provides a random number generator, thus it is only necessary to scale the random variable Z to fit to the required mean and variance. This can be achieved with the following transformation:

$X_i = Z * \ \sigma_i + \mu_i$
Note: σ is the standard deviation

X_i is now the sampled distance from the given distribution. Using this process, it is now possible to sample the mixture model:

$P(\theta|X) = \sum_{i=1}^{K} \phi_i N(\mu_i, \Sigma_i)$
Where X_i is equal to $N(\mu_i, \Sigma_i)$

The whole process of sampling the mixture model is implemented in the script functions calculate_range and updateWeights, which uses the support function div and sum to normalize the weights.

4.5 Evaluation

Using the model of Parlow we were able to simulate the error. Although we were not able to reach quite the same precision as his data had shown. While the mean values seem rather accurate, the variance is a little less realistic.

We were not able to find the reason for this. Using the formula mentioned above, the first result was very unrealistic. Figure 3 shows the data, which was collected using the real virtaul ranging sensors from the FINken. Figure 4 shows our initial sampled values. When comparing Figure 3 and 4 it is clear, that the variance is not correct.

Since we couldn't find an error in the calculations, our approach to fix the problem was to find parameters we can tweak. This means however, that we could only approximate Parlows error model. The challenge in tweaking the parameters were the weights. The updated weights were correct in comparison with Parlows weights. So adjusting the variance was limited, so that the weights wouldn't change.

Finally, we approximated the error by multiplying the variance by 10 (see Figure 5). The factor is eliminated in the process of normalizing the weights. The resulting samples were a little better spread and therefore the result was closer to Parlows model.

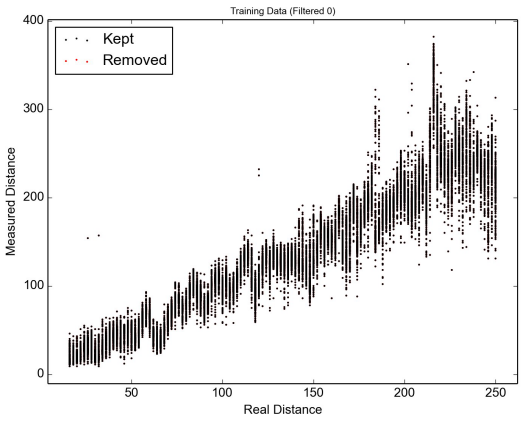


Figure 4: Real-World Sensor plotted by Simon Parlow

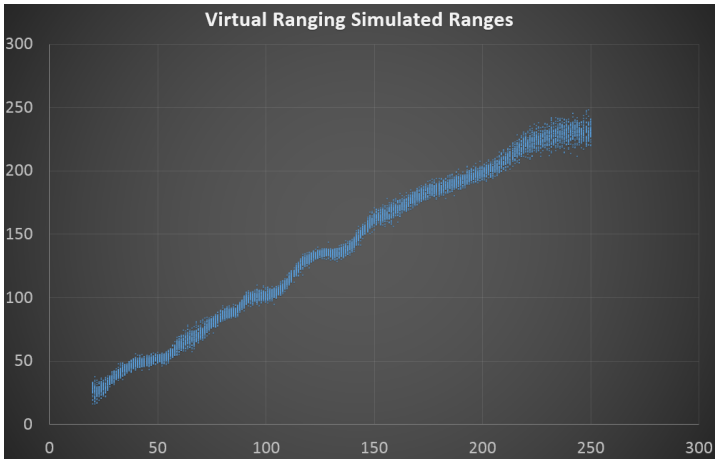


Figure 5: Simulated Senor without Fix

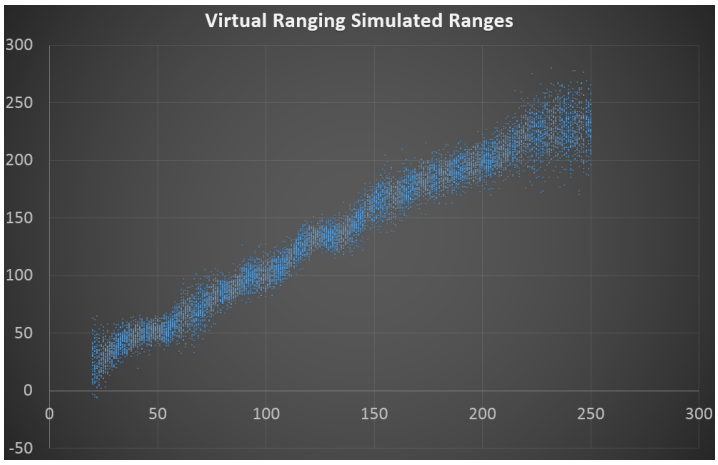


Figure 6: Simulated Sensor with Fix

Figure 6 shows a plot of virtual ranging measurement towards another FINken over time. Although the virtual ranging (the red line) is following the real distance, the error is quite impactful, which needs to be considered in any application using this sensor.

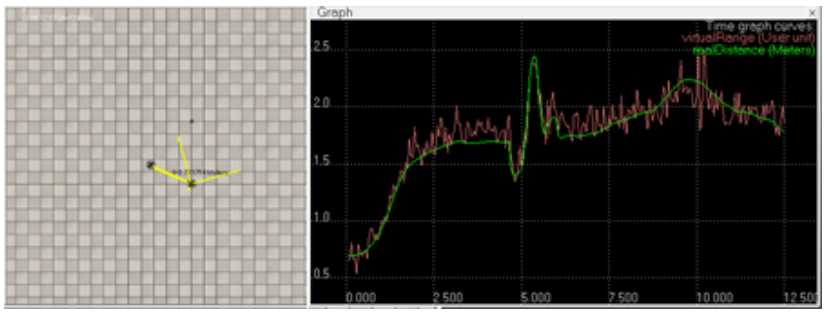


Figure 7: Virtual Ranging in Vrep

4.6 Conclusion

Using this new sensor, it might be possible to conclude new knowledge from the surroundings. One of the problems of the FINken is caused by the fact, that it is so far impossible to distinguish between wall and FINken using only ultrasonic sensors. The attraction-repulsion function, which tries to make the swarm agents stick together, makes the robots stay close to the walls, as they are treated as part of the swarm. Combining the virtual ranging, which only measures other FINken and the ultrasonic sensor measurements, it is could be possible to perform a plausibility check, whether or not the ultrasonic sensor measurement is a FINken or a wall, by comparing the ultrasonic sensor measurement with the virtual ranging measurements.

Since it is unlikely, that there is a FINken at the same distance as the wall, we can filter a lot of the wall measurements, allowing the swarm to move more freely through the arena. An example of this has been implemented in the alternative Finken util lua file "finken_util_with_Virtual_Ranging_Wall_Check.lua". This approach however also leads to some problems:

1. It is not impossible for a FINken and a wall to be at the same range
2. Due to the inaccuracy of ultrasonic sensor and virtual ranging many measurements between two FINkens will be filtered as well, leading to problems with the swarm cohesion

As a solution to these problems a system should not judge on individual measurements, but rather on a sequence of measurements. A wall will very frequently cause the plausibility check to fail, whereas FINken to FINken failures will only occur with a specific distribution.

Using this effect, it should be possible to either conclude where the wall is and specifically navigate the FINken away from it, or to simply ignore the ultrasonic sensor measurements. This should be possible, since the FINken's orientation doesn't change drastically, while being stationary at a wall, the same ultrasonic sensor will therefore detect the same wall over and over. When calculating attraction, the measurements of that ultrasonic sensor can be ignored until the FINken moved away, which can be measured by the inertia sensors. The inertia sensors can also measure the rough orientation change of the FINken to conclude, if another ultrasonic sensor starts measuring the wall. The repulsion must still be computed however, so that the FINken does not crash into the wall.

When used right the virtual range sensor should be able to assist in a broad range of tasks, which allows for a more stable and useful behaviour of the swarm.

5 Conclusion to the project

It was great to work with realistic robots and the simulation at the same time. Also it was pretty hard and there were many struggles to deal with, but at the end it was pretty great to see the result.
Due to a member leaving the team, we were not able to fulfill all goals of this project. This includes the error implementation for the inertia sensors, the resizing of the arena to the original size and changing the ground material to a more realistic material. However the task was to make the simulation more reliable and realistic, and this we were able to achieve.