Delft University of Technology

CESE5020 - Research Internship

Internship: Performance Optimization of the Localization Code for Swarm Robots

Shashank Dilip Kumar (5694876, s.dilipkumar@student.tudelft.nl)

Friday 5th April, 2024

Contents

1
Introduction

1.1. Aim of this internship

The primary aim of this internship is to optimize the performance of a localization algorithm for swarm robots. This involves learning to interface the robots with Elisa3 software, and working with ROS, C++ and Python. We will also tackle hardware-related issues, gaining experience with hardware-software integration. To verify the performance of our new code-base, we had planned to implement a foraging algorithm from [1]. Due to time constraints, we implemented a consensus algorithm for the robots instead to accomplish our goal.

1.2. What is localization?

Robot localization involves estimating the pose (position and orientation) of a robot. This is essential to any task we wish our robot(s) to perform. The robot localization task can be categorized based on existence or non-existence of the map of its environment (or a reference coordinate system) [2]. In this report, we consider that a reference coordinate system is available to the user.

Given an initial condition of a robot, the subsequent pose of the robot is obtained using information from the robot's onboard sensors. A mobile robot is equipped with sensors such as wheel encoders, inertial sensors (such as IMUs), and proximity sensors that can help estimate the pose of the robot relative to the initial pose if a mathematical model of the robot's motion is defined. This method of position estimation is called odometry or dead-reckoning. Errors present in these sensor measurements can lead to an inaccurate estimation. Additional measurements that are not part of the robot's onboard sensors can help in reducing erroneous estimation. This can be in the form of known landmarks in the environment, camera measurements of the environment, etc. In order to obtain better estimates of the location of the robot, it is necessary to estimate not only the pose but also the uncertainty associated with it. This is possible by using localization algorithms such as the Kalman filter, or the particle filter.

In this report, we make use of the extended Kalman filter (EKF), which is an extension of the classic Kalman filter but for nonlinear systems. In particular, we make use of the multi-rate extended Kalman filter (mr-EKF), which is the work of a former master student at TU Delft [3]. In this work, three different sensor measurements are fused to obtain an estimate: odometry and accelerometer (onboard), and camera measurements from the OptiTrack motion capture system present in the DCSC Lab of TU Delft. These camera measurements act as the environment map. The reason for choosing mr-EKF instead of the standard EKF is to account for potentially different sampling times of the different sensors. This enables us to deal with asynchronous data from the sensors. The general workflow of an mr-EKF can be described in two steps. In the first step, the state is observed by each sensor individually. In the second step, the estimation obtained from all the sensors are combined concurrently.

1.3. Improving the software performance

Our work will focus on improving the performance of the existing code for localization. The python part of the code-base is responsible for the poor performance of the algorithm. Therefore, we will work on improving this part of the code-base. It is possible to achieve this in several ways, which include

- Using proper data structures and writing memory-efficient code

- Using a different compiler such as PyPy
- Identify the bottlenecks and convert them to Cython
- Convert the entire python code to C++ code
- Using multiple threads or parallel processing

We chose the approach of converting the entire python code to C++. Although an arduous task, the performance of the C++ code far exceeds that of its python counterpart, at least for this case. We will show this in subsequent sections.

2
Methodology

2.1. Physical design of the robot

Figure 2.1 shows the physical design of an Elisa3 robot. Some of the important features include [4]

- 8 MHz microprocessor
- A 3-axis accelerometer
- RF radio for communication
- 2 DC motors
- 8 proximity sensors for obstacle detection and avoidance
- 4 ground sensors for line following and cliff detection
- Max speed of 60 cm/s
- Wireless communication throughput of around 250Hz for 4 robots, and 100Hz for 10 robots

Additionally, due to limited memory capabilities of the robot, most of the processing is done on the PC side. The robot is responsible in computing odometry, transmitting the status of its sensors , and receiving actuator commands from the central PC.

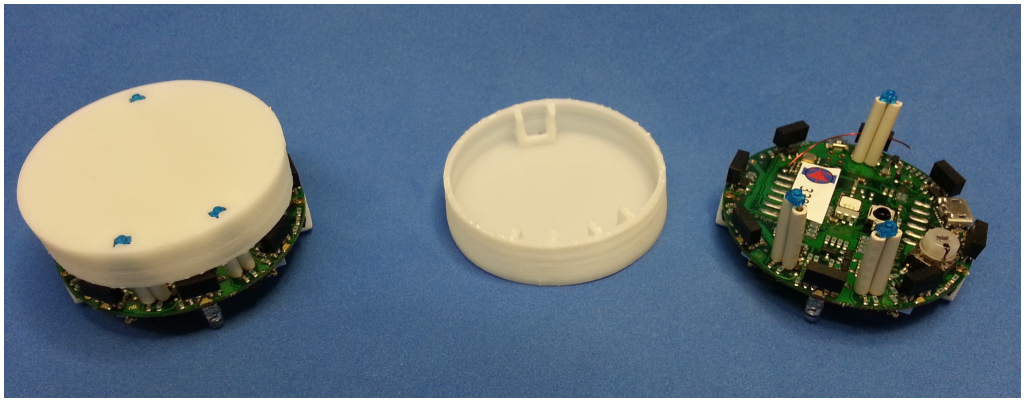

Figure 2.1: Physical design of an Elisa3 robot [4]

2.2. Dynamics of the robot

We consider an individual robot (or agent) of the swarm to be a unicycle-type robot [3][5] with state equations given by

$$\begin{aligned} x(k+1) &= x(k) + v(k)cos(\theta(k)) + w_1(k) \\ y(k+1) &= y(k) + v(k)sin(\theta(k)) + w_2(k) \\ \theta(k+1) &= \theta(k) + \Omega(k) + w_3(k) \\ \text{where } & x(k), y(k), \theta(k) \in \mathbb{R} \\ & v_x(k), v_y(k), \Omega(k) \in \mathbb{R} \\ & w_1(k), w_2(k), w_3(k) \in \mathbb{R} \\ & k \in \mathbb{N} \end{aligned} \tag{2.1}$$

The states $x(k)$ and $y(k)$ represent the position (in X and Y coordinates) with respect to the OptiTrack's coordinate system at instance k. The state $\theta(k)$ represents the orientation of the agent

in rad with respect to the positive X-axis. $v(k)$ and $\Omega(k)$ represent the input linear velocity and input angular velocity respectively. $w_1(k)$, $w_2(k)$ and $w_3(k)$ are assumed to be zero-mean white Gaussian noise acting on the states.

2.3. Filtering algorithms used

To obtain estimates from the odometry and accelerometer, we use the single-rate EKF (sr-EKF), as we obtain both the measurements synchronously. We then fuse these measurements with the camera measurements using an mr-EKF. While fusing measurements from different sensors, we may have to deal with missing measurements from some of these sensors. Missing measurements can be a result of poor communication, sensor drift, and sensor information arriving at different frequencies. Therefore, to combine estimates from different sensors, we use frameworks such as cascading, ordered weighted average (OWA), and fuzzy logic. In the current code-base, we use the OWA framework along with mr-EKF to combine different sensor estimates. It is simple, easy to implement and computationally efficient. The OWA framework assigns weights for different estimates that minimizes the mean square error [3].

2.4. Software architecture

Figure 2.2 shows the high-level architecture of the code-base. In the original code-base, the estimator node is written in python. One attempt to speed-up the this part of the code was to use a PyPy compiler. However, this did not yield significant differences in execution time. Another attempt was made to convert only a part of the code to C++. However, this made interfacing the C++ and Python parts of the code more complicated. The attempt to convert the entire Python part of the code to C++ was successful as we were able to obtain significant improvements in the execution time.

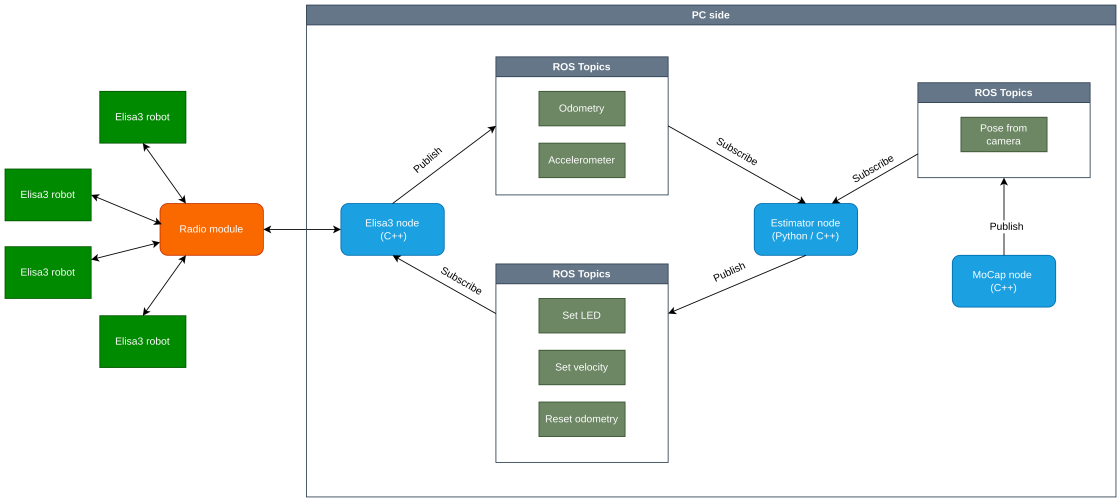

Figure 2.2: Architecture of the code-base

In figure 2.2, the individual Elisa3 robots communicate with the PC (central hub) using a radio module. The ROS node named “Elisa3” is responsible in sending and receiving information to and from the robots. Information received from the robots are published using ROS, which are subscribed to by the ROS node “Estimator”. This node estimates the current position of each robot and computes the necessary speed for the robots to move towards its goal in the next time step. The estimator node publishes this information using ROS, which are subscribed to by the Elisa3 node.

For the results obtained, we used a loop frequency of $50Hz$ ($20ms$) for the Elisa3 node, and $\frac{50}{3}Hz$ ($60ms$) for the Estimator node. The OptiTrack system operates at a frequency of $120Hz$.

2.5. Conversion from Python to C++

We now begin convert the python code in the Estimator node to C++ code. This node is responsible in getting information from all the sensors, performs sensor fusion to obtain the next estimate, and compute a control input, which is communicated back to the Elisa3 node. We avoid using dynamic variables as much as possible to achieve efficient memory usage. We also make use of optimized libraries like the Eigen library to speed-up vector calculations [6].

2.6. Measuring the performance of the implementation

To measure the performance of the code, we task the agents to implement a simple consensus algorithm. We implement the phase algorithm to achieve formation control of the agents. The phase algorithm uses a proportional controller to converge to the final position, which is the average of the initial positions of all the agents with an added bias. Further information related to the consensus of these agents can be found in [7].

The proportional controller for an agent is designed as per equation 2.2. The input velocity $v_i(k)$ given to an agent with index i is computed as the norm of the error $e_i(k)$ between the destination $(x_{d,i})$ and current position $(x_i(k))$ of the agent multiplied by a proportional gain. The input angular velocity $\Omega(k)$ is computed as the difference between the slope of the line connecting the current and destination coordinates $(\phi(k))$ and the current orientation $(\theta(k))$, multiplied by a proportional gain (see figure 2.3). The variable N in equation 2.2 refers to the total number of agents present in the environment. The variable b_i is the bias added to the destination position of agent i. We run the experiment for the case of two and four agents.

$$
\begin{aligned}
&e_i(k) = [x_{d,i} - x_i(k), y_{d,i} - y_i(k)] \\
&v_i(k) = ||K_{P1} e_i(k)|| \\
&\phi_i(k) = tan^{-1}\left(\frac{y_{d,i} - y_i(k)}{x_{d,i} - x_i(k)}\right) \\
&\Omega(k) = K_{P2} tan^{-1}\left(\frac{sin(\phi_i(k) - \theta_i(k))}{cos(\phi_i(k) - \theta_i(k))}\right) \\
&\text{where } e_i(k) \in \mathbb{R}^2, \phi_i(k), K_{P1}, K_{P2} \in \mathbb{R} \\
&x_{d,i} = \frac{1}{N}\sum_{i=0}^{N-1} x_i(0),\ y_{d,i} = \frac{1}{N}\sum_{i=0}^{N-1} y_i(0) + b_i,\ i \in \mathbb{N}
\end{aligned}
\tag{2.2}
$$

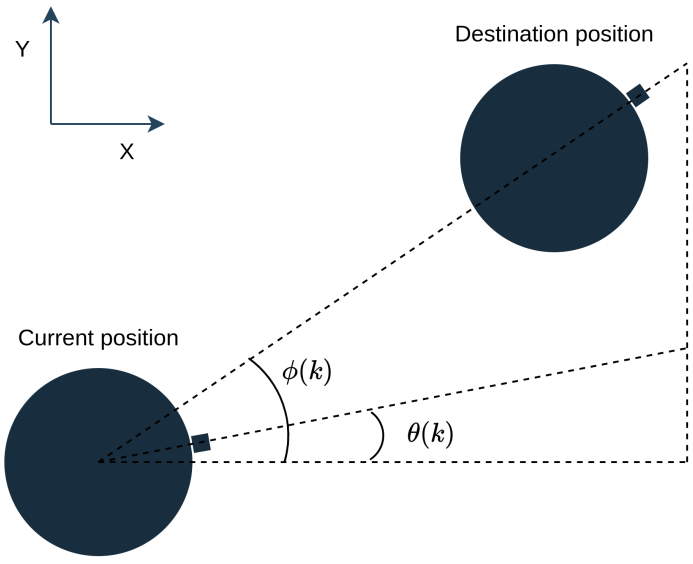

Figure 2.3: Depiction of angles $\theta(k)$ and $\phi(k)$ from equation 2.2

2.7. Conversion of control input to robot input

We compute the control input (linear and angular speed) for each agent using equation 2.2. This control input is published via ROS, which is then subscribed to by the Elisa3 node (see figure 2.2). Each robot accepts an integer value between -128 and 127 as the speed of each of its wheels. We call the commands to set the robot speed as “setLeftSpeed” and “setRightSpeed”. For the sake of understanding, let us refer to the linear velocity component of the command as “setLinSpeed”, and the angular velocity component as “setAngSpeed”. setLeftSpeed and setRightSpeed is a linear combination of setLinSpeed and setAngSpeed.

2.7.1. Converting linear velocity control input to Elisa3 command

To make our problem simpler, we consider the robot to only move forward or turn left or right. Therefore, setLinSpeed will contain values that primarily lie between 0 and 127. One unit corresponds to $5mm/s$ and the maximum speed of the robot is around $60cm/s$ (for a value of 127). Since our estimator node runs at a rate of $60msec$, we restrict our robot to a maximum speed of around **25cm/s** in order to allow smoother movement and avoid sudden jerks due to high speed. This motivates us to set an upper limit of 50 as input for setLinSpeed. Additionally, a lower limit of 5 is set for the input of setLinSpeed, as values lower than this do not provide any meaningful movement.

2.7.2. Converting angular velocity control input to Elisa3 command

The angular velocity control input is specified by $\Omega(k)$. To avoid erratic movements, we restrict the value of setAngSpeed to the range [-15, 15]. A positive value implies the robot must turn in the clockwise direction, and a negative value implies the robot must turn in the anti-clockwise direction. The speed in commands setLeftSpeed and setRightSpeed are computed as follows:

$$\begin{aligned} &\text{setLeftSpeed = setLinSpeed + setAngSpeed} \\ &\text{setRightSpeed = setLinSpeed - setAngSpeed} \end{aligned} \tag{2.3}$$

An additional feature is added, where the robot focuses on rotation instead of translation if setAngSpeed is greater than a value of 5. This allows the robot to orient itself towards the goal before moving forward, helping it to reach the goal in a shorter path.

3
Results

3.1. Performance comparison between the old and new code-base

We conducted experiments on Elisa3 robots to test the implementation of our code. For the first comparison between our code and the old version, we look at the execution time of the Estimator node for **50** loops (see figure 3.1). We can see that the average execution time per loop has gone down from **434msec** (Python version) to **2.06msec** (C++ version). It is important to note that we do not have to run our estimator node at a rate of **2.06msec** (as this only increases the processing cost), but it enables us to run the node at a fast enough frequency. Upon considering the communication delays and the sensor sampling rates, we run the Elisa3 node at a rate of **20msec**, and the estimator node at a rate three times slower, at **60msec**.

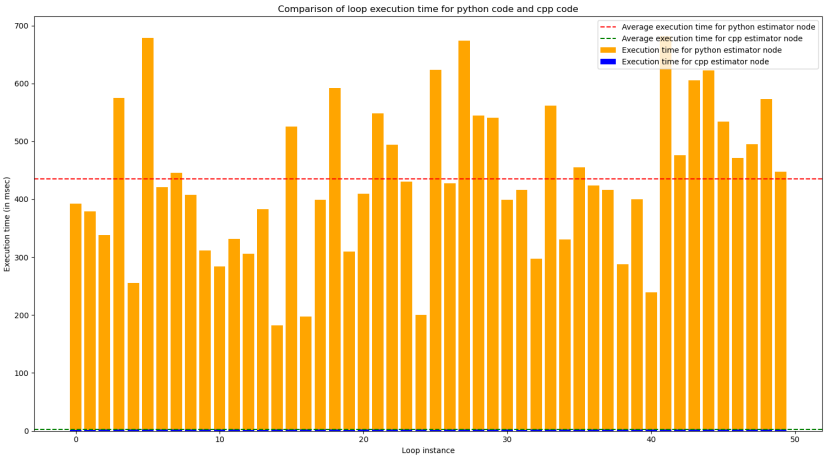

Figure 3.1: Comparison of execution time for a loop in the estimator node

3.2. Formation control for four agents

The agents are tasked with assembling into a straight line formation. The goal (destination) coordinates for an agent is an average of the initial condition of all agents plus a bias term unique to each agent. Figure 3.2 shows the trajectory of four robots reaching consensus. The experiment was run for **400** loops in the estimator node. We can see that the agents reach consensus, as indicated by the small error between the robot's estimate and the prescribed goal coordinates in figure 3.2. It is notable that not all robots needed 300-400 iterations to reach its goal. The code-base used for these experiments can be found in [8]

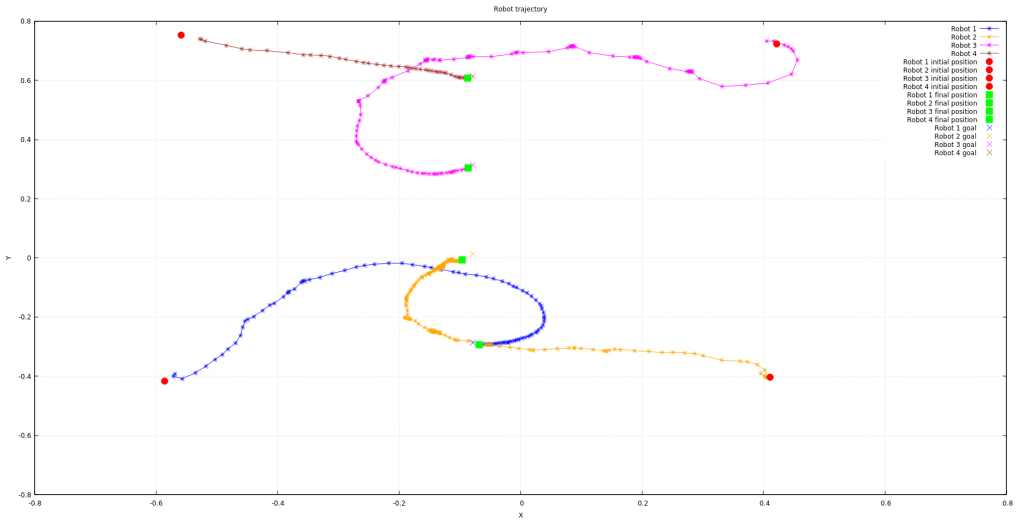

Figure 3.2: Formation control for 4 agents

Figures 3.3 and 3.4 show the evolution of the odometry, camera, accelerometer, and mr-EKF estimates for agents 1 and 4 in the above case. It is clear that the odometry is highly unreliable in the case of agents 1 and 4 despite calibration. The accelerometer readings are more reliable as we calculate its estimate using a combination of the accelerometer readings from the onboard sensor and the orientation reading from the mr-EKF estimate. The mr-EKF estimation fixes its weights accordingly, which assigns a very small weight to the odometry estimates and a higher weight to the camera and accelerometer estimates, which gives a more accurate estimation. The video of the agents reaching consensus can be viewed via this link

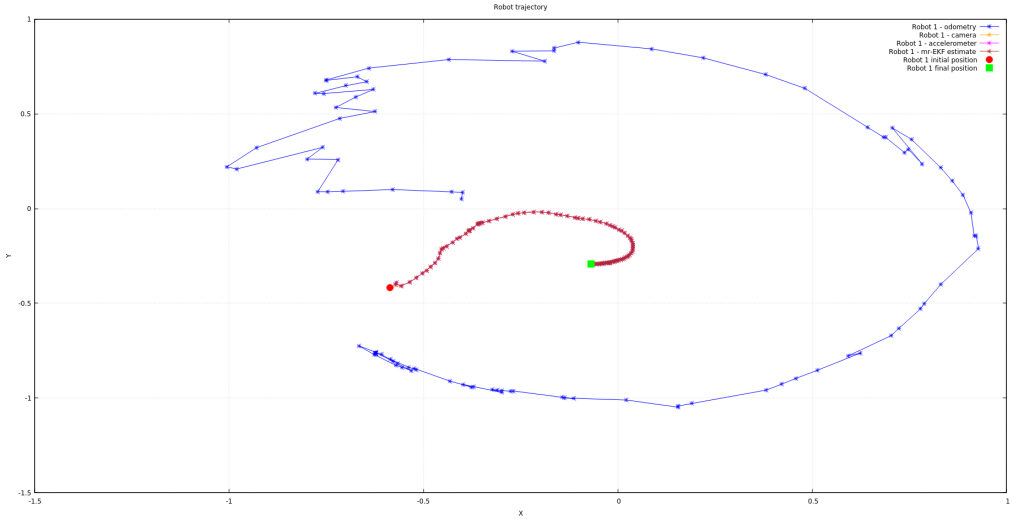

Figure 3.3: Evolution of sensor measurements and mr-EKF estimate for agent 1

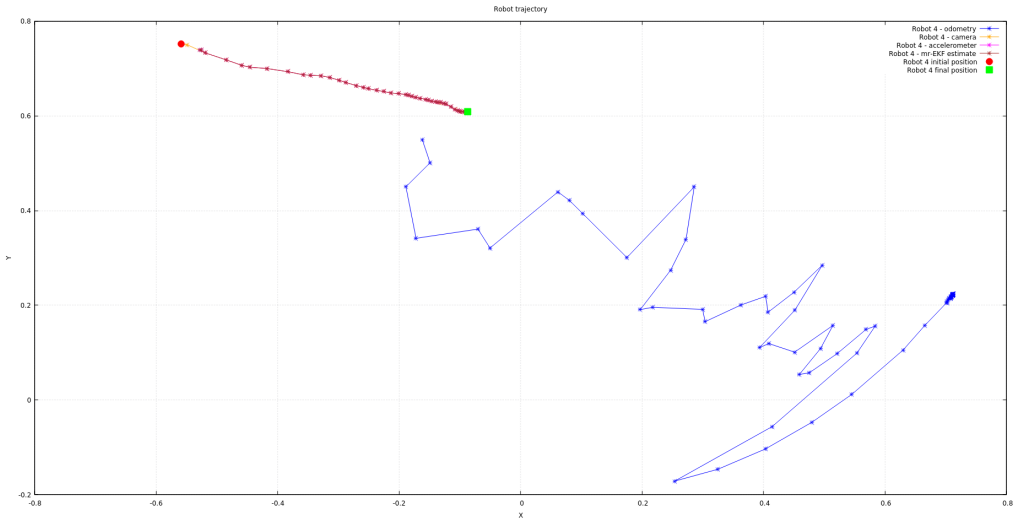

Figure 3.4: Evolution of sensor measurements and mr-EKF estimate for agent 4

3.3. Formation control for two agents

To understand how the number of robots can affect execution time, we ran experiments with only two robots and compared the execution time with the four robot case (see figure 3.5). We can find that on average, the execution time in the case of two robots is slightly faster (by **0.4msec**). From this, we can conclude that increasing the number of agents does increase the execution time, but the increase is very small in this case. Due to lab usage constraints, we were unable to run experiments for more than four agents. This will be deferred to future work.

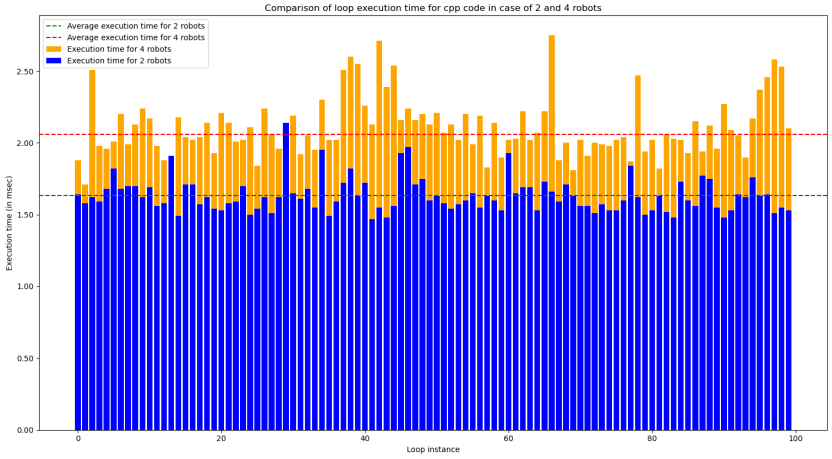

Figure 3.5: Comparison of execution time in the estimator node between 2 agents and 4 agents

The trajectory of the agents for consensus of two robots is seen in figure 3.6. Additionally, figures 3.7 and 3.8 show the evolution the odometry, camera, accelerometer, and mr-EKF estimates for both agents. Similar to the case of formation control for four agents, the odometry readings are highly erroneous, which causes the mr-EKF to assign higher weights to the camera and accelerometer readings. The video of the agents reaching consensus can be viewed via this link

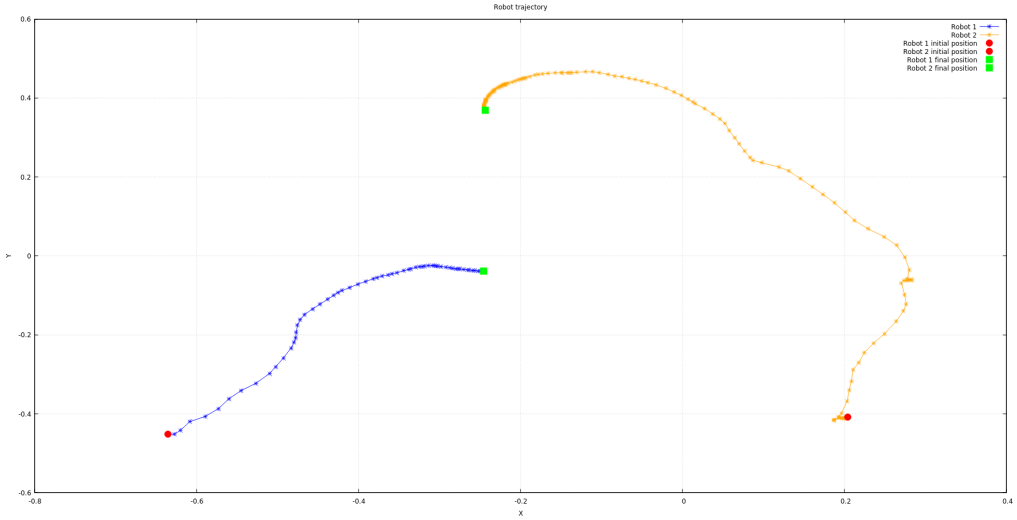

Figure 3.6: Formation control for 2 agents

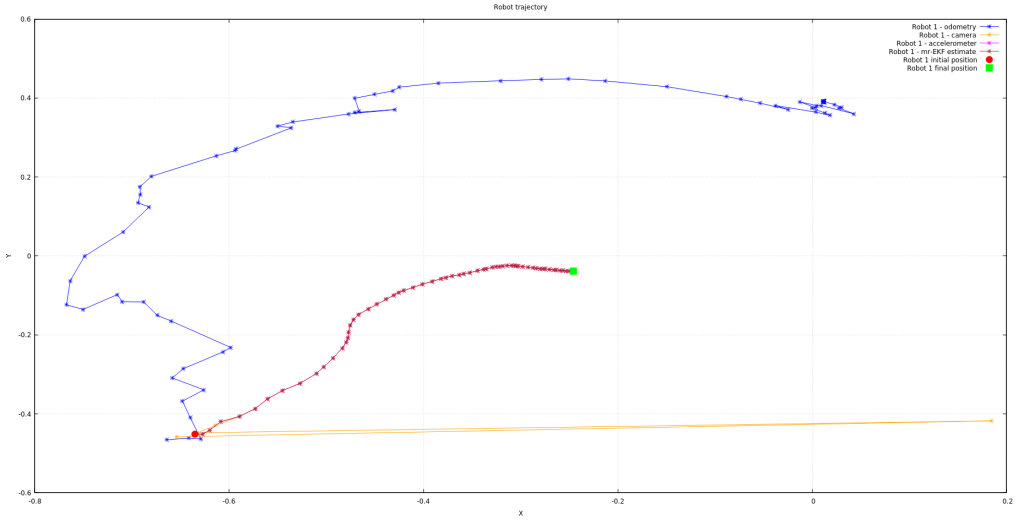

Figure 3.7: Evolution of sensor measurements and mr-EKF estimate for agent 1

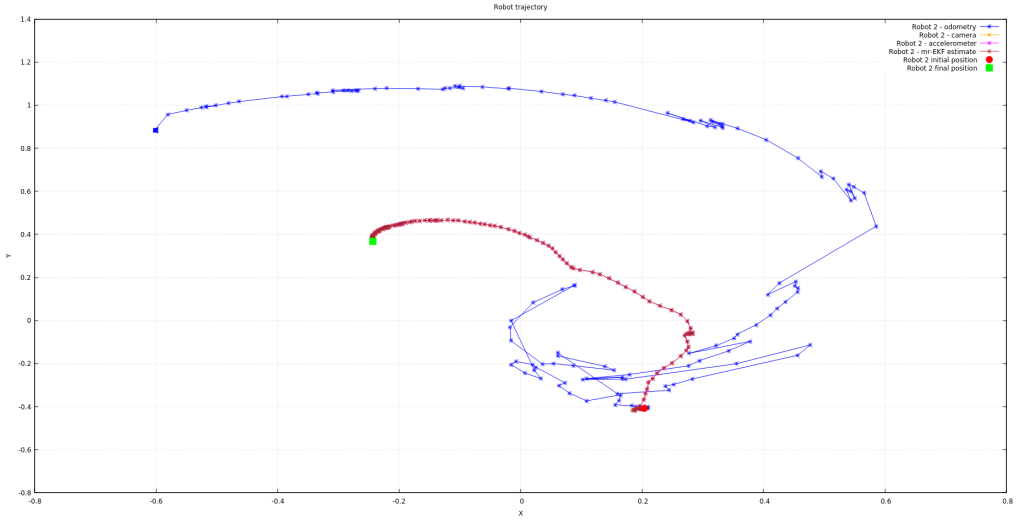

Figure 3.8: Evolution of sensor measurements and mr-EKF estimate for agent 2

4
Discussion & Future work

4.1. Disappearance of markers

The size of the markers used for detection by the OptiTrack system also plays a role in the accuracy of the measurements. In certain regions of the environment (lab test area), the smaller sized markers were not being detected by the OptiTrack system, causing an agent to “disappear” as there are not enough markers to detect the agent (a minimum of 3 markers is required to define a rigid body (or agent) for the OptiTrack system to detect it).

For example, consider four agents with a unique identifier from the set $1, 2, 3, 4$. In the event that agent 3 “disappears”, the OptiTrack system considers agent 4 to now act as agent 3 as it cannot find the original agent 3. This leads to an error in the estimation of agents 3 and 4. We have included a function in the code that helps combat this issue. Figure 4.1 shows an example of different marker sizes for the robots.

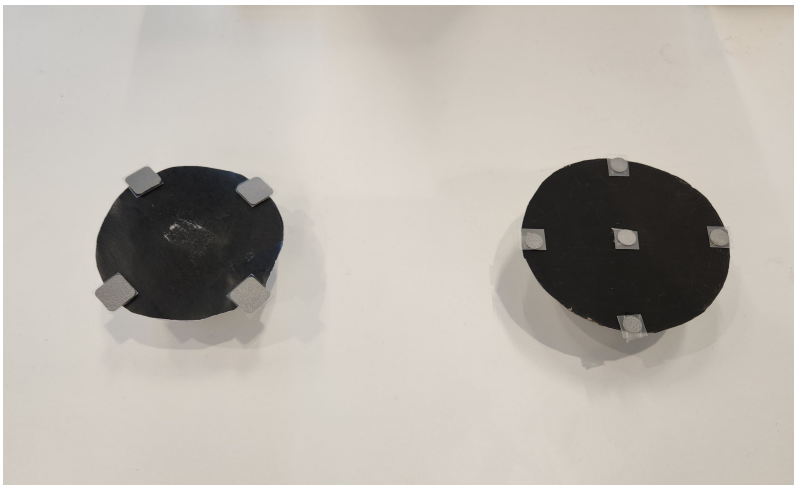

Figure 4.1: On the left: A medium-sized marker; On the right: A small-sized marker. The medium sized is detected better by the OptiTrack system compared to the small-sized marker.

Using bigger markers for detection can reduce such errors, but very big markers may cause the OptiTrack to consider two adjacent markers as a single marker. Hence, it is necessary to find the right balance in marker size.

4.2. Boundary condition

There is a boundary condition so that the robots do not go outside a specified boundary. In case a robot moves outside the boundary, or the estimate is erroneous, the robot stops moving.

4.3. Inherent errors in the robots

Not all robots work as intended. In some robots, one wheel rotates faster than the other wheel when input with the same speed, causing the robot to not travel in a straight line when it is expected to. The top speed of robot ‘A’ can be different than that of robot ‘B’. Despite calibrating all the robots, there are some differences in the speed input to the motors and the actual speed of the

motors. We can introduce closed loop controllers, PID controllers or robust controllers to ensure that the robots move with a set speed, at the cost of increased complexity and computational time. This can be one avenue to pursue for future work.

4.4. Path followed by the robots to reach the goal

As we saw in figures 3.2 and 3.6, the robots do not take the shortest path available to reach the goal. When conducting multiple experiments, a robot's decreasing charge causes it to move slower for the same set of input commands. This causes the robot to veer off in different directions before reaching its goal. Like we mentioned in the previous section, implementing speed control for the robots can help alleviate this issue. Additionally, path planning algorithms can help the robots reach the goal in the shortest path. The code can be improved in the future to include a path planner for all the robots. Another improvement can be to introduce obstacle avoidance schemes to prevent the robots from crashing into each other.

5
Conclusion

The goal of this internship was to improve the performance of of the localization algorithm for swarm robots. We were able to achieve an improvement factor of 200 in the execution time over the original code-base, which enabled us to operate our algorithm at our desired frequency. We also managed to achieve formation control for the case of 2-agent and 4-agent consensus. On the software side, we gained experience in working with motion capture systems, a software library for Elisa3 robots, ROS, C++, and Python. On the theory side, we gained knowledge on single rate and multi rate Kalman filters, and consensus.

A lot of challenges were faced during the conversion of the Python code to C++. Understanding some parts of the original code and their purpose was very tricky. Debugging the errors in our new code was time consuming, as one would expect with changing a large code-base from one language to another. This made us realise the difficulties in modifying already existing code, which is a common practice in industries.

Working with hardware always presents a lot of challenges. We tested a lot of robots to eliminate the defective ones. There was a motor speed mismatch in some robots, whereas in others, it was not possible to upload the firmware. Due to the limited availability of the testing area in the lab, we had to debug the errors as much as possible before testing them in the lab.

Finally, I would like to thank dr. ir. Manuel Mazo Jr. for giving me the opportunity to work on this project.

References

[1] S. Adams, D. Jarne Ornia, and M. Mazo Jr, "A self-guided approach for navigation in a minimalistic foraging robotic swarm," *Autonomous Robots*, vol. 47, no. 7, pp. 905–920, 2023.

[2] S. Huang and G. Dissanayake, "Robot localization: An introduction," *Wiley Encyclopedia of Electrical and Electronics Engineering*, pp. 1–10, 1999.

[3] Y. Li, "Indoor localization with multi-rate extended kalman filter," M.S. thesis, Delft University of Technology, 2023.

[4] *Elisa3 - gctronic wiki*, https://www.gctronic.com/doc/index.php/Elisa-3, Accessed: 27-03-2024.

[5] J. Ghommam, H. Mehrjerdi, M. Saad, and F. Mnif, "Formation path following control of unicycle-type mobile robots," *Robotics and Autonomous Systems*, vol. 58, no. 5, pp. 727–736, 2010.

[6] *Eigen library for c++*, https://eigen.tuxfamily.org/index.php?title=Main_Page, Accessed: 27-03-2024.

[7] E. Zwetsloot, "Consensus algorithms for single-integrator multi-agent systems," M.S. thesis, Delft University of Technology, 2023.

[8] *Localization code for elisa3 robots*, https://github.com/shashank0911/elisa3_code, Accessed: 27-03-2024.