Implementation and evaluation of a partially decentralized Cooperative Localization algorithm in ROS with a swarm of TurtleBot robots

Master Thesis

In partial fulfillment of the requirements for the degree

"Master of Science in Engineering"

Study program:

Mechatronics - Electrical Engineering

Management Center Innsbruck

Supervisor:

Sonia Martínez, Ph.D.

Assessor:

Benjamin Massow, M.Sc.

Author:

Joanthan Hechtbauer

1410620009

Declaration in Lieu of Oath

Acknowledgment

This work was supported by a fellowship within the FITweltweit program of the German Academic Exchange Service (DAAD) and a fellowship of the Austrian Marshall Plan Foundation. Furthermore, special appreciation is given to Sonia Martínez from UCSD and Solmaz S. Kia from UCI for the professional help during the realization of the master thesis and to Leena Saurwein and Benjamin Massow from MCI for the support in the preparation phase. Finally, the author would like to thank his parents for their support during the whole study program.

Abstract

The thesis describes the implementation of a *Cooperative Localization* algorithm, applied on a group of *TurtleBot* robots. External measurements between team members are introduced and the acquired data is fused with the internal estimation of the robots. The purpose is to improve the localization of the individual agents for autonomous operations in dynamic and unpredictable environments.

This project investigates a novel algorithm of a partially decentralized *Extended Kalman Filter* scheme. Thus, it is a hybrid of the conventional systems that are either fully centralized or decentralized. The investigated states of every robot are the position in x- and y-direction and the orientation. The according technical term is called pose. On the one hand, every agent calculates the propagation stage of the filter locally. This is executed distributed and no communication is required. They establish a belief with the velocity measurement of the wheel encoders and the method of odometry. On the other hand, the update stage is carried out in a centralized manner. An innovation of the pose is obtained via visual measurements. Afterwards, the master of the team is informed, which consequently processes the information and broadcasts an update-message to the whole team. Finally, the agents apply this message to enhance their predicted pose.

The cooperative software architecture of the project is developed on basis of the *Robot Operating System*. The implementation consists of multiple programs that run distributed over the whole team. The exteroceptive measurements are achieved by the on-board Kinect camera and an image processing library, called *ArUco*. The library can distinguish the pose and identity of a certain observed fiducial markers. Therefore, a referencing cube was developed, which is placed on top of every robot's rack. It presents on every side two markers with different size. Thus, the robots can be detected from all angles and a wide distances range. There is one further marker on top of the cube, which is used by a camera that is mounted to the ceiling of the room. It retrieves absolute measurements that can also be fused with the *Kalman Filter*.

At the end, the novel algorithm is investigated with the group of *TurtleBot*s in real world test scenarios. The first runs validate the individual stages isolated of the filter. Gradually, more robots are included and the final test run comprises a comprehensive measurement sequence that also features the incidence of message drop-outs. The results proofs that the filter is correctly implemented and it supports the theoretical statement that the additional sensor information improves the localization. The relative measurements ensure an consistent team formation over time. However, the global localization of the whole team is limited to the individual member with the best belief. Sporadic absolute measurements enhance the quality of the system and prevent a drift of the estimated team formation within the global world frame.

Keywords: cooperative localization, partially decentralized, EKF, ROS

Kurzfassung

Die Thesis beschreibt die Implementierung eines kooperativen Lokalisierungsalgorithmus, welcher auf eine Gruppe von *TurtleBot* Roboter angewendet wird. Hierfür werden externe Messungen zwischen den Teammitgliedern eingeführt und die erlangten Daten werden mit der internen Schätzung der Roboter fusioniert. Das Ziel ist die Verbesserung der Lokalisierung der individuellen Mitgliedern für autonome Tätigkeiten in dynamischen und unberechenbaren Umgebungen.

Dieses Projekt untersucht einen neuartigen Algorithmus eines teilweise dezentralen *Extended Kalman Filters*. Er ist somit ein Hybrid der herkömmlichen Systeme, welche entweder vollständig zentral oder dezentral aufgebaut sind. Die untersuchten Zustände jedes Roboters sind die Position in x- und y-Richtung und die Orientierung. Der entsprechende Fachbegriff nennt sich Pose. Auf der einen Seite berechnet jeder Agent die Prognose des Filters lokal. Dies wird dezentralisiert ausgeführt und folglich ist keine Kommunikation erforderlich. Sie erstellen eine Schätzung mit Hilfe der Geschwindigkeitsmessung der Radencoder mittels Odometrie. Die Korrekturphase andererseits wird in zentraler Weise durchgeführt. Ein Messwert der Pose wird über optische Mittel erzielt. Danach wird der Kapitän des Teams informiert, welcher folglich die Informationen verarbeitet und eine Update-Nachricht an das gesamte Team versendet. Schlussendlich verwenden die Agenten diese Botschaft um ihre geschätzte Pose zu verbessern.

Die kooperative Software-Architektur des Projekt ist auf Basis des Betriebssystems *Robot Operating System* entwickelt. Die Implementierung besteht aus mehreren Programmen, die über das gesamte Team verteilt ausgeführt werden. Die exterozeptiven Messungen werden durch die integrierte Kinect-Kamera und einer Bildverarbeitungsbibliothek namens *ArUco* erzielt. Die Bibliothek kann die Pose und die Identität einer beobachteten Markierung wahrnehmen. Daher wurde ein Würfel zur Referenzierung entwickelt, welcher auf der Oberseite jedes Robotergestells platziert wird. Er weist auf jeder Seite zwei Markierungen mit unterschiedlicher Größe auf. Somit kann der Roboter von allen Winkeln und einem weiten Entfernungsbereich erkannt werden. Es gibt eine weitere Markierung auf der Oberseite des Würfels, welche von eine Kamera, die an der Decke des Raumes montiert ist, verwendet wird. Sie erzielt absolute Messungen, die ebenfalls in den *Kalman Filter.*eingebracht werden können.

Am Ende wird der neue Algorithmus mit der Gruppe von *TurtleBot*s mit Testszenarien in der realen Welt untersucht. Die ersten Durchläufe dienen der Validierung der einzelnen Stufen des Filters. Nach und nach werden weitere Roboter hinzugefügt und der letzte Testlauf beinhaltet eine umfassende Messsequenz, die auch Nachrichtenstörfälle aufweist. Die Ergebnisse belegen, dass der Filter korrekt implementiert wurde und sie unterstützen die theoretische Behauptung, dass die Lokalisierung durch die zusätzlichen Sensorinformationen verbessert wird. Die relativen Messungen gewährleisteten über die Zeit eine beständige Teamformation. Allerdings ist die globale Lokalisierung des gesamten Teams auf das einzelne Mitglied mit der besten Schätzung begrenzt. Sporadische absolute Messungen werten die Qualität des Systems auf und verhindern einen Drift der geschätzten Teamformation innerhalb des globalen Weltkoordinatensystems.

Schlagwörter: kooperative lokalisierung, partiell dezentralisiert, EKF, ROS

Contents

1 Introduction

1.1 Motivation

Having broken free from its origins in industry, robotics got a significant impact on today's life. Driven by the era of Industry 4.0 and by autonomous cars its development is progressing exponential. The consumer industry reduces the costs of mechatronical components and by that enables low-priced robotic platforms. As a consequence, the research with multi-robot systems arise more and more attention.

Among others, the term Cooperative Localization has been established, with the goal to enhance localization of a network of agents by combining the fields communication and perception. This is in particular appealing for application that have to deal with unknown environments and sporadic localization measurements. To name one example, there is a huge trend in autonomous underwater robots. In this scenario, it is not possible to receive e.g. persistent GPS measurements. However, one member can temporary approach the surface and afterwards, provide the acquired information to the rest of the team. In order to incorporate the update, the team must feature a coupling of the individual poses. An accurate pose-estimation is essential for robotics, since high-level tasks like navigation and general interactions with the environment rely on it.

1.2 Objective

The Multi-Agent Robotics (MURO) laboratory at UC San Diego is designing and analyzing cooperative algorithms on autonomous multi-agent robotic networks. In this context, the thesis implements a novel Cooperative Localization algorithm. The acquired results serve for an ongoing paper as prove of concept.

The new algorithm is applied on a group of robots, called *TurtleBot*, which are powered by the *Robot Operating System*. The members are coupled via relative measurements. Therefore, an onboard camera and an image processing library, called *ArUco*, are utilized to retrieve the pose of detected agents with respect to the frame of the observer. The acquired information is then fused with the proprioceptive sensor-data of the robots, by the help of a partly distributed *Extended Kalman Filter*. This hybrid filter inheres the innovation of the paper and is promising, since it combines the advantages of conventional approaches. Finally, several test runs are executed to evaluate the algorithm and to derive the intended result.

1.3 Structure

At first, Chapter 2 explains the fundamentals. It describes the terms mentioned in the objective, including the *Robot Operating System*, the *TurtleBot*, the *ArUco* Marker, the *Extended Kalman*

Filter for localization and the *Cooperative Localization*. Afterwards, Chapter 3 introduces the prior arts that lead to the novel algorithm and besides, highlights the procedure itself. The following Chapter describes the methods of the implementation and gives an overview of the intended cooperative architecture. Then, the thesis continues with the actual realization (5) of the filter. At first, it describes the set-up of the vision system and then the individual C++ scripts. In Chapter 6 the test runs are illustrated and the according results are displayed with the help of Matlab plots.

2 Fundamentals

2.1 Robot Operating System

The robots, which are used to invest the multi agent algorithms are powered by the *Robot operating System* (ROS). This chapter introduces the basics of the system and further provides details about the platform itself.

The system is established by the laboratory *Willow Garage* in 2007 and has evolved as the standard tool among robotics researchers. The general idea behind, is the following: ”The Robot Operating System is a loosely-defined framework for writing robot software. It is a collection of tools, libraries, and conventions that aim to simplify the task of creating complex and robust robot behavior across a wide variety of robotic platforms.” [1] Thus, the framework gives a head start for developing new robotic functionalities, by making use of pre-developed infrastructure, drivers and software. The operating system is not depending on specific robot platforms, but rather encourages to animate new ones. One reason for its success is deduced from the open-source philosophy. The permissive BSD license enables to network and collaborate with world class researchers.

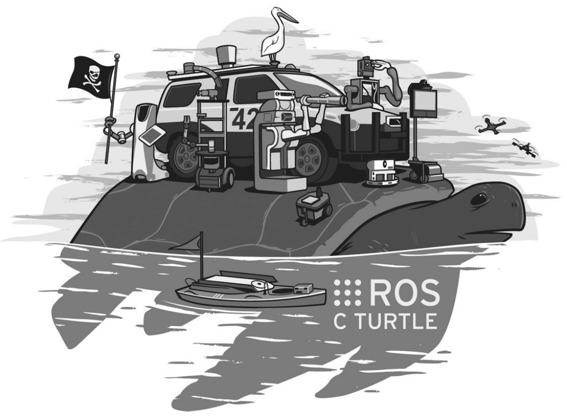


Figure 2.1: ROS poster of an early distribution highlighting popular robots.

Figure 2.1 shows the poster of an early distribution, which gives an overview of popular ROS robots. Among them, are common research platforms like the *TurtleBot*, which will be described in more detail in Chapter 2.2.

ROS is intended to be used in *Ubuntu Linux* and can be interpreted as a second layer above the operating system. By establishing global ROS paths and variables, the system can automatically find libraries by its names and link dependencies among themselves. Besides, it support multi-lingual programming. Popular are the scripting language *Python*, which is convenient for prototyping; and the compilable *c++* language, which yields enhanced performance. The resulting program of the script resembles a ROS node. Usually, a complex robot system is decomposed in a large network of individual nodes. Some examples for sub tasks might be sensing, image processing and manipulation. They can communicate with each other by sending predefined messages. Furthermore, it would be straight forward to exchange a robot with another or a simulated one. Cause of the universal interface, one could still apply the same processing programs.

A prerequisite to use the system is to run a so-called *ROS Master*, which ensures the allocation of the individual nodes. By referencing one IP address with the variable *ROS_MASTER_URI*, it is possible to extend the network over multiple machines. After running the master in a terminal by the command `roscore`, a node can be started in a new terminal by typing `rosrun` and the name of the linked executable. It's further possible to start a master and multiple nodes at once by listing them in a so-called *launch* file, which for its part can be run with the command `roslaunch`.

The group of files belonging to a project are usually stored in a so-called *package*. By default it consists of the basic folders */src* and */launch*, the *CMakeList.txt* file for the compiling description and the *package.xml* file for the package details. Every node that requires compiling, e.g. programs that are written in C++, must be listed in the text file. An example is provided in the following:

```
add_executable( given_name src/filename.cpp)
target_link_libraries( given_name ${catkin_LIBRARIES})
```

Listing 2.1: Referencing of a C++ node for compiling.

The content of the package can be extended with further data types which is shown in later chapters regarding the implementation.

A further crucial point in the ROS system is the communication, because robotic applications are distributed over a multi-node network and individual scripts can be of different programming languages. Therefore, strict conventions are required. The communication architecture is asynchronous, which means that messages are not send directly in between the nodes, but are stored in message flows. ROS offers three principle methods of conversation. The most typical one is the *message*, which uses *topics* as flow channels for a one-way transport. To start an exchange, one has to initialize a topic instance with a unique name and a ROS message type. Afterwards, nodes can publish (write) or subscribe (read) to this topic. If a subscriber receives a messages on the specified topic a callback function is invoked, which can be used to process the incoming data. A more complex method is the *service*. Its special attribute is that it consists of a pair of messages, which are used for a request and reply. Thus, it resembles a typical function, which can be called from an external node. The last message type is called *action*. It is similar than a service, but has the additional features to get periodic feedback and the ability to cancel the request. Thus, it is better suited for tasks that require long executing time as e.g. picking up an object. Further information on communication is given in Chapter 4.3, which deals with the creation of own message types and their implementation.

2.2 TurtleBot

The algorithms of this thesis are investigated with a group of *TurtleBot* robots (Figure 2.2), which are specially designed for the use of ROS. Its mobile base is derived from a robot vacuum cleaner. Thus, the wheel configuration is established with a two-wheel differential drive and two additional points of contact. This enables the *TurtleBot* to move forward, backward and to turn around its own axis. By launching its *bringup* node, one can move the base by publishing a pose message on the dedicated topic. In addition, the robot continuous publishes its odometry with the help

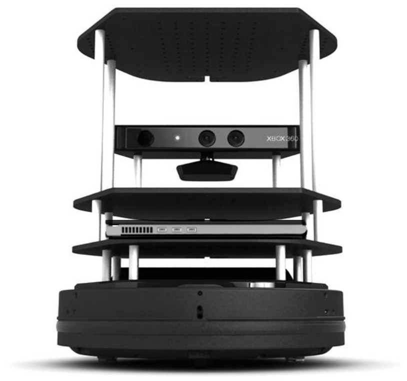


Figure 2.2: The TurtleBot robot. [2]

of wheel encoders. The measurement estimates change in position over time with respect to the starting location.

The rack, mounted on top of the platform, contains a *Kinect* camera. This optical sensor enables the robot to get information about its surrounding environment. On the one hand, the node publishes a RGB live stream, which can be used e.g. for teleoperation or further processing in OpenCV. On the other hand, it publishes a *depth map*, which contains the 3D information of the captured spot. However, since the depth sensor of the Kinect relies on a projected IR light pattern, this functionality is lost outdoors.

To sum up, the *TurtleBot* contains all basic hardware required for an autonomous robot and is at the same time very inexpensive. Thus, it is an ideal platform for multi-agent research.

2.3 ArUco Marker

The *ArUco* library is developed by Rafael Muñoz and Sergio Garrido in their paper "Automatic generation and detection of highly reliable fiducial markers under occlusion" [3]. A fiducial marker, which is placed in the field of view of an imaging system serves as a point of reference. This qualifies the library, as stated by the paper, to be "specially appropriated for camera pose estimation in applications such as augmented reality and robot localization". The library is implemented in the popular image processing software *OpenCV*, which can be accessed within ROS.

The marker itself consists of a square with a pattern of black and white pixels, surrounded by a white padding. It can be created by the library itself or with a web application [4] and has parameters for its ID-number, size and padding. Figure 2.3 displays the markers of the first three IDs with a size of 40 mm and 5 mm padding.

The position of the *ArUco* marker can be interpreted by detecting the four corners of the square. Moreover, the orientation and identity can be found with the binary code of the pattern, which consists of a 7x7 pixels grid. The boarder, which has to be composed of only black pixels is a first rejection test. The series of the remaining five bits per row are based on a Hamming code. To be precise, three pixels are used for error detection and the other two carry the actual information. Thus, with the given five rows it is possible to encode up to 1024 different IDs. By extracting the code of the marker, one can obtain four identifiers, one for every rotation. If any of them is found in the marker's dictionary the candidate is considered as a valid marker. Finally, a bounding box

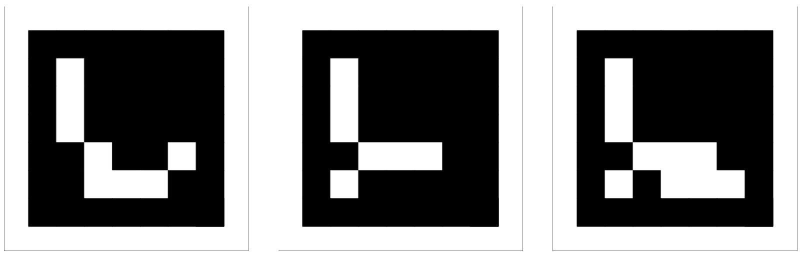


Figure 2.3: Examples of ArUco markers with Size of 40 mm and Padding of 5 mm. From left to right: ID = 1, ID = 2, ID = 3.

and a coordinate system can be drawn into the camera image (compare Figure 2.4) to indicate the pose of the marker.

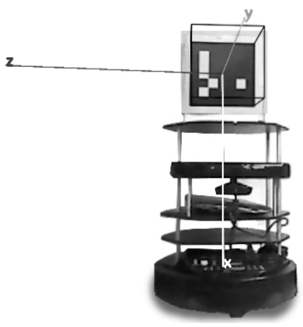


Figure 2.4: Bounding box and coordinate system, drawn in the image for a detected ArUco marker.

According to [3], the steps for the algorithm can be summarized as the following:

1. Generate a gray-scale image
2. Image segmentation by thresholding
3. Contour extraction and filtering
4. Marker code extraction
5. Marker identification and error correction
6. Corner refinement and pose estimation

2.4 Extended Kalman Filter for localization

The book *Probabilistic Robotics* [5] is the major dictionary for probabilistic problems dealing with uncertainty. It introduces the basic techniques and provides examples, based on robotic applications. They name the *Kalman Filter* among the different approaches for estimation and describe it as ”[...] a technique for filtering and prediction in linear systems [...]” with the help of ”[...] belief computation for continuous states.” A belief computation has the form of a Gaussian distribution and thus, can be described by the mean value and the standard deviation.

The project’s application of the filter is the localization of a robot. Thus, the observed states define the pose of the mobile platform. The corresponding mean vector $X_t = [x_t, y_t, yaw_t]^T$ consists of the position in x and y direction and the orientation regarding the z-axis. The size of the vector is equal to the degree of freedoms N and its entries depend on time, as indicated by the subscript t.

The robots must rely on sensors to determine its position in the real world. Since every sensor is affected by noise, one can increase the accuracy with the help of multiple ones. Thereby, it is the task of the filter to fuse the data from the different sources and to establish the belief of the current pose. The standard deviation provides information about the certainty of the estimated states and is expressed by the covariance matrix P_t, with dimension $[NxN]$. Its diagonal entries contain the variances of the individual states and the off-diagonal values describe their correlations.

Since the movement of a robot is rarely of linear behavior, the filter is in fact not practicable for localization applications. This problem can be overcome with the *Extended Kalman Filter* (EKF), which is designed for nonlinear systems. The procedure of the filter can be seen in Algorithm 1 [5]. It can be divided into a propagation and update routine.

The first part (line 1-2) estimates the states of the robot based on the previous position x_{t-1} and the new relative movements. The robot has to integrate the velocity data u_t over time according to the state transition matrix G to determine its new pose. In practice, the applied sensor is a wheel-encoder and errors might occur because of the slip of the tires. Therefore, the propagation increases the uncertainty of the prediction for every iteration with the noise covariance matrix *R*.

The second routine (line 3-5) uses a measurement z_t and the corresponding observation model H to update the propagated states. One possible measurement event might be that the robot recognizes a known object, also called *landmark*. By knowing the absolute position of the landmark, the robot can apply this measurement via the Kalman gain K and improve its own absolute localization. Consequently, the updates decrease the uncertainty. It is restricted by Q, the noise covariance matrix of the measurement. The innovation of the extended algorithm can be seen in the nonlinear functions $g()$ and $h()$.

```
Algorithm 1: Extended Kalman Filter
  Input  : $x_{t-1}, P_{t-1}, u_t, z_t$
  Output: $x_t, P_t$
1 $\bar{x}_t = g(x_{t-1}, u_t)$
2 $\bar{P}_t = G_t P_{t-1} G_t^T + R_t$
3 $K_t = \bar{P}_t H_t^T (H_t P_t H_t^T + Q_t)^{-1}$
4 $x_t = \bar{x}_t + K_t(z_t - h(\bar{x}_t))$
5 $P_t = (I - K_t H_t)\bar{P}_t$
```

2.5 Cooperative Localization

As the name *Cooperative Localization* (CL) implies, the technique makes us of a team of robots that exchange information in order to improve their location estimation. Usually, the members are coupled with each other through their relative pose. This pose can be acquired by an exteroceptive sensor, which is able to measure the distance and the orientation of detected robots. If one agent e.g. has an outstanding certainty about its absolute pose, it can provide this information to the team. The members can then use this knowledge and combine it with the known relative pose to improve their own estimation.

It is important to maintain the relations of the team. Therefore, the term *cross-covariance* is introduced according to Equation 2.1. There is one matrix required for every combination of *Agent i* and *j*, with each index representing one element out of the set of team members *N*. The variable *k* represent the current time step and thus, the following one is stated as *k+1*.

$$\mathbf{P}_{ij}(k+1) = \mathbf{F}^i(k)\mathbf{P}^+_{ij}(k)\mathbf{F}^j(k)^T \tag{2.1}$$

However, new challenges are arising with the CL algorithm regarding communication, memory and processing costs. One particular example is the maintenance of the cross-correlations in a way, so that one can avoid an all-to-all communication for every time-step. This demanding has led to two principal approaches. The first one uses a centralized and the second one a decentralized architecture.

The centralized architectures is the most straight forward approach.The team is coordinated from one central processing unit. Consequently, this master has to process the information for the entire group of robots for every time step. As such, the *Handbook of Robotics* assesses the technique in the following way: "[...] often practically unrealistic due to their vulnerability to a single point of failure, and due to the difficulty of communicating the entire system state back to the central location at a frequency suitable for real-time control." [6]

The decentralized approach has no central control unit and every agent has to execute the filter locally. Thus, all members have to maintain the correlative team information through a network of agent-to-agent communication. The according review is: "Decentralized control architectures are the most common approach for multirobot teams [...] This control approach can be highly robust to failure, since no robot is responsible for the control of any other robot. However, achieving global coherency in these systems can be difficult [...]" [6]

One recent work on decentralized CL is highlighted in the prior arts of the thesis in Chapter 3.1. Besides this paper, the thesis also describes the novel hybrid approach, which is a combination of the two conventional techniques.

3 Prior art

3.1 Decentralized algorithm based on Kalman filter decoupling

For the estimations in CL systems the individual member of the team depend on each other. On the one hand they are coupled because of the cross-covariance equation of the propagation stage. After a relative measurement between two agents occurs, the maintenance of the term requires the information of both agents for every successive time step. As a consequence, both robots have to communicate high frequently, even if no new measurement occurs. On the other hand, the coupling is even more present in the update stage. In the paper of [7], Solmaz S. Kia and her team describe a novel decentralized algorithm and propose the idea to decouple the Kalman filter. It is based on the observation that the cross-covariance $\mathbf{P}_{ij}$ consists of three parts. As demonstrated by Equation 2.1, there is the $\mathbf{F}_i$ matrix, which is local to Agent i; $\mathbf{F}_j$ which is local to Agent j; and the cross-covariance matrix of the previous time step. Out of this observation the paper proposes to split the term into three intermediate variables (compare Equation 3.1).

$$\mathbf{P}_{ij}(k+1) = \mathbf{\Phi}^i(k+1)\mathbf{\Pi}_{ij}(k)\mathbf{\Phi}^j(k+1)^T \tag{3.1}$$

The term $\mathbf{\Phi}$ can be propagated by every agent locally. Therefore, the variable is available locally and $\mathbf{\Phi}^j$ must be obtained by communication with the observed agent.

The intermediate variable Π_{ij} can be calculated according to Equation 3.2). Therefore, every agent has to keep a local copy of the corresponding matrices. Because of symmetry, it is enough to save the upper triangle of the matrix. A group of four robots, requires for example every agent to maintain the following matrices $\mathbf{\Pi}_{12}, \mathbf{\Pi}_{13}, \mathbf{\Pi}_{14}, \mathbf{\Pi}_{23}, \mathbf{\Pi}_{24}, \mathbf{\Pi}_{34}$.

$$\mathbf{\Pi}_{ij}(k+1) = \mathbf{\Pi}_{ij}(k) - \mathbf{\Gamma}_i\mathbf{\Gamma}_j^T \tag{3.2}$$

The decomposition enables the system to process updates in a centralized manner. Therefore, the novel algorithm introduces an *Interim Master*, which involves a variable central processing unit. To be precise, every member of the team becomes the master, as soon it executes a robot-to-robot measurement. Similar to central architectures, it acquires the information from the observed agent in order to calculate and communicate the updates for the other agents of the team. Subsequently, the members can assemble the Kalman gain with the obtained message and the local information and update their states and covariance matrix. Thus, the update process is fully decoupled and there is no need for an all-to-all communication. Furthermore, one can include absolute measurements into the system. The agent, which acquires the information, becomes again the *Interim Master*. One important advantage is that there is no communication required, if no measurement takes place.

So far, only the main equations of the split theorem are presented. The further intermediate steps of the algorithm are described in more detail in the following chapter. Besides, it gives additional information to the lately introduced variables $\mathbf{\Phi}$, $\mathbf{\Pi}$ and $\mathbf{\Gamma}$.

It should be noted that the paper [7] itself also features an substantial outlook. While doing so, it describes previous works that lead to the novel algorithm.

3.2 Partially decentralized EKF

The *Partially decentralized EKF algorithm* [8] is based on the split theorem of the previous algorithm. The difference is that the filter has this time a partially decentralized architecture. The main focus of the thesis is on this algorithm, since it is the one, which is implemented with the real robot systems.

The propagation of this hybrid approach is still fully decentralized, but the update is carried out by one central processing unit. Hence, the $\mathbf{\Pi}$ matrices, which accounts for team correlations, has no longer be maintained by every individual member and the system requires less storage. Nevertheless, the hybrid still benefits from the decentralized advantages of low processing cost and the robustness to message dropouts.

This chapter lists all the mathematical equations that are required for the algorithm. As such, it describes the essential steps of the filter in the order: Initialization, propagation and update. The implementation of the partly decentralized EKF in ROS is described later in the Chapters 5.3.2 and 5.3.3.

First of all, one has to initialization the filter by the following. Every agent has to hold some initial values for its state vector $\mathbf{x}$ and the error covariance matrix $\mathbf{P}$. If the states are not known, one can use random values and at the same time, set the covariance to high values, representing a large uncertainty. Furthermore, they have to initialize the cross-covariance fragment $\mathbf{\Phi}$ as identity matrix.

The central processing unit is responsible for the variable $\mathbf{\Pi}_{ij}$. Therefore, it has to initialize a zero matrix for every combination of the teams' members. $\mathbf{\Gamma}$, on the other hand, must not be initialized, since it is not required beforehand. Instead, it can be calculated out of previously determined variables in the update stage.

The propagation of the *Kalman Filter* is executed decentralized. Every agent calculates for itself the local variables according to Equation 3.3. This has to be done every time step. The time difference between two consecutive propagations is defined as Δt, which is equal to one over the filter's frequency.

$$\begin{aligned}
\mathbf{X}^i &= \mathbf{X}^i + \dot{\mathbf{X}}^i \Delta t = \mathbf{X}^i + \begin{bmatrix} v^i cos(\phi^i) \\ v^i sin(\phi^i) \\ \omega^i \end{bmatrix} \Delta t \\
\mathbf{P}^i &= \mathbf{F}^i \mathbf{P}^i \mathbf{F}^{i^T} + \mathbf{G}^i \mathbf{Q}^i \mathbf{G}^{i^T} \\
\mathbf{\Phi}^i &= \mathbf{F}^i \mathbf{\Phi}^i
\end{aligned} \tag{3.3}$$

The equation reveals the impact of the *Extended* filter. The $g()$ term of Algorithm 1 uses trigonometric functions to describe the nonlinear movement of the x-y-plane. Moreover, the variables $\mathbf{F}$ and the $\mathbf{G}$ are derived through linearization of the motion model and can be calculated with Equation 3.4, respectively, 3.5. The new covariance is composed by two components, which have the following cause: "[...] ; one estimating uncertainty due to the initial location uncertainty, the other estimating uncertainty due to motion noise." [5]

$$\mathbf{F}^i = \begin{bmatrix} 1 & 0 & -v^i sin(\phi^i)\Delta t \\ 0 & 1 & v^i cos(\phi^i)\Delta t \\ 0 & 0 & 1 \end{bmatrix} \tag{3.4}$$

$$\mathbf{G}^i = \begin{bmatrix} cos(\phi^i) & 0 \\ sin(\phi^i) & 0 \\ 0 & 1 \end{bmatrix} \Delta t \tag{3.5}$$

The term $\mathbf{G}$ considers the motion noise and is defined by the matrix 3.6. The variables σ implies the measurement uncertainty of the respective velocity.

$$\mathbf{Q}^i = \begin{bmatrix} \sigma_v^2 & 0 \\ 0 & \sigma_w^2 \end{bmatrix} \tag{3.6}$$

The update is, as already mentioned, executed in a centralized manner. There is no communication required, assuming that no measurement events occur. Then, every agent can updates its variables by Equation 3.7 and the central unit by Equation 3.8.

$$\begin{aligned} \mathbf{x}_{i+}(k+1) &= \mathbf{x}_{i,}(k+1) \\ \mathbf{P}_{i+}(k+1) &= \mathbf{P}_{i,}(k+1) \end{aligned} \tag{3.7}$$

$$\mathbf{\Pi}_{i+}(k+1) = \mathbf{\Pi}_{i,}(k+1) \tag{3.8}$$

If a relative measurement is reported, the central processing unit receives the vector $\mathbf{Z}_{ab}$, which consists of the relative distance in x- and y-direction and the observed orientation. As a next step it has to gather the *landmark-message* from the involved agents, which consists of the three terms: $\mathbf{x}$, $\mathbf{P}$ and $\mathbf{\Phi}$.

Consequently, the master can ascertain the residual $\mathbf{r}^a$ by the difference of the measured and the estimated relative pose (compare Equation 3.9).

$$\mathbf{r}^a = \mathbf{Z}_{ab} - \mathbf{Z}_{est} \tag{3.9}$$

The estimation can be established by the difference of the propagated poses. In order to compare it with the measurements, they have to be transformed into the frame of the observer.

$$\mathbf{Z}_{est} = \begin{bmatrix} cos(\phi^a)\Delta x + sin(\phi^a)\Delta y \\ -sin(\phi^a)\Delta x + cos(\phi^a)\Delta y \\ \Delta\phi \end{bmatrix} \tag{3.10}$$

The unit proceeds to calculate the variable $\mathbf{S}_{ab}$ by the following equation to determine the gain of the new measurement.

$$\mathbf{S}_{a,b} = \mathbf{R}^a + \tilde{\mathbf{H}}_a\mathbf{P}^a\tilde{\mathbf{H}}_a^T + \tilde{\mathbf{H}}_b\mathbf{P}^b\tilde{\mathbf{H}}_b^T + \tilde{\mathbf{H}}_a\mathbf{\Phi}^a\mathbf{\Pi}_{a,b}\mathbf{\Phi}^{b^T}\tilde{\mathbf{H}}_b^T + \tilde{\mathbf{H}}_b\mathbf{\Phi}^b\mathbf{\Pi}_{b,a}\mathbf{\Phi}^{a^T}\tilde{\mathbf{H}}_a^T \tag{3.11}$$

There is the variable $\mathbf{R}^a$, which includes the measurement error. The $\tilde{\mathbf{H}}_i$ matrices are required to respect the transformation of the estimation. They are derived by the following:

$$\begin{aligned} \tilde{\mathbf{H}}_a &= \mathbf{Rot}^T * \begin{bmatrix} 1 & 0 & -\Delta y \\ 0 & 1 & \Delta x \\ 0 & 0 & 1 \end{bmatrix} \\ \tilde{\mathbf{H}}_b &= \mathbf{Rot}^T \end{aligned} \tag{3.12}$$

With:

$$\mathbf{Rot} = \begin{bmatrix} cos(\phi^a) & -sin(\phi^a) & 0 \\ sin(\phi^a) & cos(\phi^a) & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{3.13}$$

The master has to be calculate two more variables in order to accomplish the update messages, depicted in Equation 3.14. It should be remembered that he will provide one message for every member of the team.

$$\begin{aligned} \mathbf{Update1} &= \mathbf{\Gamma}_i\bar{\mathbf{r}}^a \\ \mathbf{Update2} &= \mathbf{\Gamma}_i\mathbf{\Gamma}_i^T \end{aligned} \tag{3.14}$$

The central processing unit weights the residual to receive $\bar{\mathbf{r}}^a$:

$$\bar{\mathbf{r}}^a = \mathbf{r}^a\mathbf{S}_{ab}^{-1/2} \tag{3.15}$$

Additionally, it calculates $\mathbf{\Gamma}$ for every robot from the given variables. However, one has to distinguish different cases. As depicted in 3.16 there is one equation for the observer (*Agent A*), one for the detected agent (*Agent B*) and one for the remaining members.

$$\begin{aligned}
\mathbf{\Gamma}_i &= (\mathbf{\Pi}_{ib}(k)\mathbf{\Phi}^{b^T}\tilde{\mathbf{H}}_b^T - \mathbf{\Pi}_{ia}(k)\mathbf{\Phi}^{a^T}\tilde{\mathbf{H}}_a^T)\mathbf{S}_{ab}^{-1/2} \\
\mathbf{\Gamma}_a &= (\mathbf{\Pi}_{ab}(k)\mathbf{\Phi}^{b^T}\tilde{\mathbf{H}}_b^T - \mathbf{\Phi}^{a^{-1}}\mathbf{P}^a\tilde{\mathbf{H}}_a^T)\mathbf{S}_{ab}^{-1/2} \\
\mathbf{\Gamma}_b &= (\mathbf{\Phi}^{b^{-1}}\mathbf{P}^b\tilde{\mathbf{H}}_b^T - \mathbf{\Pi}_{ba}(k)\mathbf{\Phi}^{a^T}\tilde{\mathbf{H}}_a^T)\mathbf{S}_{ab}^{-1/2}
\end{aligned} \tag{3.16}$$

At this point the update message (compare 3.14) is fully determined and can be published to the team. Finally, every agent has sufficient information to calculate the updated state and covariance matrix locally. It therefore applies the following equations:

$$\begin{aligned}
\mathbf{X}^i &= \mathbf{X}^i + \mathbf{\Phi}^i * [\mathbf{\Gamma}_i\bar{\mathbf{r}}^a] \\
\mathbf{P}^i &= \mathbf{P}^i + \mathbf{\Phi}^i * [\mathbf{\Gamma}_i\mathbf{\Gamma}_i^T] * \mathbf{\Phi}^{i^T}
\end{aligned} \tag{3.17}$$

Last but not least, the master updates the $\mathbf{\Pi}_{ij}$ matrices according to Equation 3.2, which was proposed in Chapter 3.1.

4 Methods

4.1 Multi-robot architecture

First of all one has to develop the architecture for the multi robot system. Figure 4.1 gives an overview of the according program structure. Following the ROS practice, the system for the CL algorithm is divided over several instances. To be precise every robot is running two nodes locally on its computer and two further ones are located at central processing unit.

The local programs comprise a relative-measurement and a sensor-fusion script for the EKF. Additionally, every robot must run several prerequisites. One node is required for launching the mobile platform and one for starting the Kinect video stream. By introducing *namespaces* (discussed in Chapter 4.2) it is possible to differ between individual agents, even though they run the same nodes. The advantage is that one can use the same program for every agent and thus, one must maintain only one script per task. This makes the system modular, regarding the number of team members.

The centralized part of the algorithm implies the requirement that every agent is able to communicate with a central processing unit. ROS is well suited for this requirement, because there is no difference, if a node is running on the same computer or on another one within the same network. Though, it is important that every device uses the same roscore. The master can be established using one of the *TurtleBot*'s computer or on another processing unit with ideally more power. This central instance will run the global-measurement and the central-processing script, which handles computation and storage for the centralized part of the filter.

For the content of the project, a new ROS package with the name *Coop_Localization* is created. All the mentioned CL nodes are stored in its */src* folder in the form of C++ scripts. Furthermore, the package contains the */include* folder for the header files and the */msg* and the */srv* folders for message type definitions. The */launch* folder contains the final *.launch* files for the agents A.1 and for the master A.2. Consequently, a package for a multi robot architecture is established, which is modular and capable of distributed CL tasks.

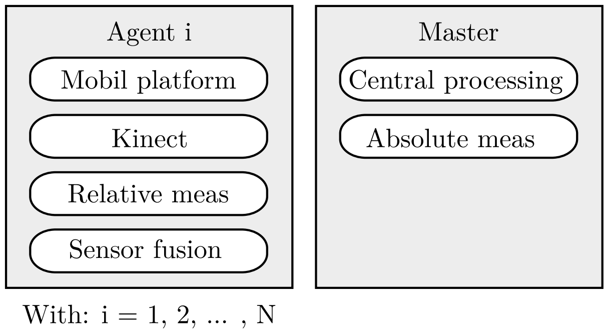


Figure 4.1: Overview of the ROS nodes of the implementation.

4.2 Namespace

In this chapter the concept of ROS namespaces is introduced and their application for the project is described.

ROS provides a hierarchical naming structure for its nodes and topics. Table 4.1 demonstrates some examples. As one can see, the namespaces are used on the one hand to group related content and on the other hand to make multiple instances of the same sources individual. This is crucial for nodes, since it is not possible to run different nodes with the same name. Multiple topics would be able to subscribe or publish to the same name. However, the source of the data would be lost. Thus, the namespaces ensure that nodes, allocated to one robot, can be handled without interfering the communication of other robots.

Table 4.1: Example of namespaces.

Nodes:
/Robot_Name1/sensor_fusion
/Robot_Name2/sensor_fusion
Topics:
/Robot_Name1/update
/Robot_Name2/update
/Robot_Name2/camera/rgb/image_mono
/Robot_Name2/camera/rgb/image_color

The best way to create namespaces is with the help of the *launch* file. As already mentioned, it is used to run several nodes at once. Listing A.1 from the attachments shows the final file, which runs on every robot. With the optional *group* tag one can wrap all nodes and assign them a common namespace via its *ns* argument (line 9). The variable *turtlebot_name* is received from the environment variable *TURTLEBOT_NAME*, which can be set in the configuration file */.bashrc* of every computer. In the laboratory, every robot has a unique name and its respective variable is set accordingly. This group tag affects every node within the brackets of the tag and their referenced topics. However, there is an way to exclude specific topics by defining their name globally, referred by a forward slash as shown in Table 4.2.

Table 4.2: Difference between global and local topic names.

	Without namespace	With namespace
Local	update	/Robot_Name1/update
Global	/update	/update

Apart from starting the nodes, the launch file also specifies ROS parameters. There is one used to hand over information of the robot name to certain nodes. Furthermore, there are scalar parameters, which are used for the initialization.

Thus, with namespaces one can encapsulation individual agent of a multi-robot system which consists of similar machines.

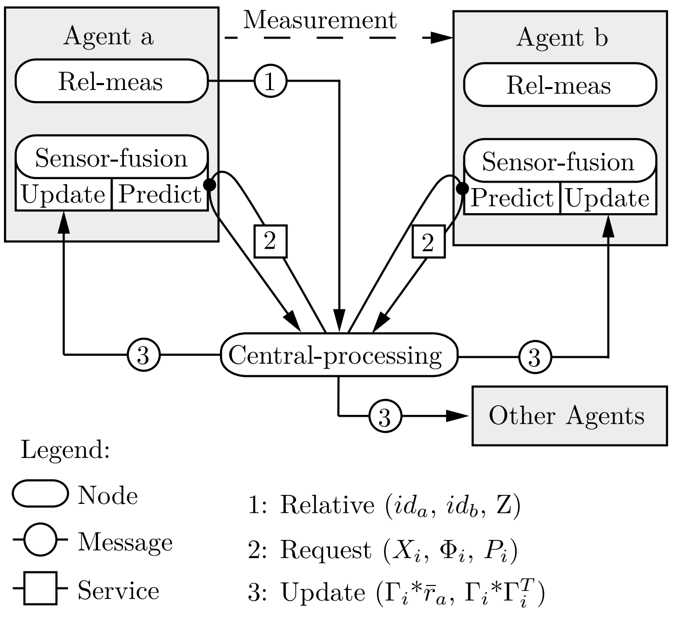


Figure 4.2: Sequence plan of communication for the processing of a measurement between two robots by a central processing unit.

4.3 Communication

As already mentioned, communication is a crucial point of the project, because of the distributed architecture. Therefore, the function of this chapter is to give an overview of the interactions between the nodes and the established message types.

All necessary communication is related to the processing of a measurement. Therefore, Figure 4.2 shows the scheduling for the case of a relative measurement event. Assuming, that *Agent a* detects *Agent b*, the *relative-measurement* node will inform the master by sending a message of type *relative*. Subsequently, the master sends a services of type *request* to the involved agents in order to gather all necessary information. Finally, an *update* message is addressed back to the *sensor-fusion* node of every agent within the team.

As explained in the the fundamentals of ROS (Chapter 2.1), every message requires a certain topic name. For the first communication there is a topic called */central_processing/rel_meas*, which is dedicated to the *central-processing* unit. One topic is sufficient, since the name of the publishing robot is included within the message.

Besides, a new message type, depicted in Listing 4.3, is created in particular for this task. In order to enable their usage for every programming language, their structure must be predefined within the */msg* folder of the package. For compiling of the new message the following features have to be added to the package:

One has to add the following lines into the *package.xml* to link the required dependencies.

```
<build\_depend>message\_generation</build\_depend>
<run\_depend>message\_runtime</run\_depend>
```

Listing 4.1: Changes in the package.xml script, required for creating a new ROS message.

Moreover, one has to adopt the code of the *CMakeLists.txt*. First of all, one has to link the *message_generation* (a) and the *message_runtime* (b) libraries. Then, one has to state the actual file (c) that contains the new message format and finally, one has to uncomment the message generation (d). This processes are important for the ROS system to get to know about the new message type. [1]

```
find_package(catkin REQUIRED COMPONENTS ... message_generation)#(a)

catkin_package(CATKIN_DEPENDS ... message_runtime)#(b)

add_message_files(FILES  rel.msg)#(c)

generate_messages(DEPENDENCIES  std_msgs)#(d)
```

Listing 4.2: Changes in the CMakeLists.txt script, required for creating a new ROS message.

The final step is to actually create the *rel.msg* text file. The first line serves only as description. The, the definition of the message's content follows, line by line, starting with the data type and followed by a given name. Apart from the familiar types like e.g. *string* and *integer*, one can include other predefined message types. The given example uses for example for the *meas* variable the type *Twist* (Listing 4.4), which in turn uses the type *Vector3* (Listing 4.5).

```
# This message type is created for relative measurements.
time                 current_time
string               agent_a
string               agent_b
geometry_msgs/Twist  meas
```

Listing 4.3: Overview of the new created message type for the relative measurement.

```
# This expresses velocity in free space broken into its linear and angular parts.
geometry_msgs/Vector3  linear
geometry_msgs/Vector3  angular
```

Listing 4.4: Overview of the ROS message type Twist.

The second exchange of data is done by the use of a service. In Listing 4.6 one can see the prepared type with two modules that are separated by the indicated horizontal line. As explained in the ROS fundamentals, the first part is the *request* of the message. Since no input is required, this is defined empty. The *reply* however, is populated with the information of the requested agent and thus, the master obtains the current variables of targeted members. One more thing to notice are the square brackets behind the data type. They indicate that the variable is an array with the size of the stated number.

The third message is sent to a topic named *update*. In this case, the allocation to the individual robot's topic is important. Thus, the master initializes for every agent of the team one publisher instance. The name of every robot is added in front of the topic to respect the associated namespaces. The subscribing nodes define their topic name locally and so, the namespace is automatically added. The according type (depicted in Listing 5.1) is specially created, to transport the result of the central processing to the individual members.

```
# This represents a vector in free space.
float64 x
float64 y
float64 z
```

Listing 4.5: Overview of the ROS message type Vector3.

```
# This service type is created to request the current information of a targeted agent.
---
float64[3] X
float64[3] PHI
float64[9] P
```

Listing 4.6: Overview of the new created message type request.

```
# This message type is created for the update msg.
float64[3] GAMMA_r
float64[9] GAMMA_GAMMA_T
```

Listing 4.7: Overview of the new created message type update.

4.4 Prototyping in Matlab

In order to get a better understanding of the algorithm, the CL is first implemented in a Matlab script. The scripting language has advantages for prototyping, since it is very easy to debug and one does not need to declare the type of variables. This includes also the fact that the dimensions of matrices are fitted automatically. Matlab in particular is very handy for the calculation with matrices and does not require any additional libraries.

In total, two different scripts are created. One represent the conventional EKF algorithm and one the decentralized *Split EKF* (compare Chapter 3.1). In this way, one can verify the implementation of the novel algorithm. This in turn, can be used to debug, step by step, the partly distributed EKF of the ROS system. The example scripts are narrowed down to two robots and the update stage. There is no need for the modeling of the motion propagation, since this can be verified easily.

The final two scripts can be seen in Appendix A.3 and A.4. They are divided in an initialization part and a continuously running loop. At first, the states and the variances of the two robots are defined. Also the fragments of the cross-covariance (Π and Φ) are initialized and the accuracy of the sensor is set. The relative measurement is emulated with the vector Z_{meas}. Afterwards, the program enters an infinity loop that contains the processing of the update according to Chapter 3.2. Every iteration of the loop corresponds to a new time step. The loop can be terminated with the command `break` after a desired number of iterations.

The Table 4.3 is intended to give an overview of the variable's dimension. It summarizes their meaning and points out the variable names, which are used in the scripts. For most of the variables, two dimensions are not enough and so, the matrices are nested in a one, respectively, two dimensional array. The first square brackets correlates to the size of the matrix and the second one to the size of the array.

Table 4.3: Listing of important variables of the algorithm. With: D = Degree of Freedoms, N = Number of Agents and [matrix size]x[array size].

Name	Dimension	Variable	Meaning
X	[Dx1]x[N]	x	State Vector
P	[DxD]x[N]	P	Covariance
Φ	[DxD]x[N]	PHI	Decomposition of Cross-Covariance (1)
Γ	[DxD]x[N]	GAMMA	Decomposition of Gain
Π	[DxD]x[NxN]	PI	Decomposition of Cross-Covariance (2)
Z	[Dx1]x[NxN]	Z_meas	Measurement
r^a	[Dx1]	r_a	Residual
$S_{a,b}$	[DxD]	S_ab	Covariance of Measurement
R	[DxD]	R	Accuracy of Sensor

5 Realization

5.1 Preparation of the vision system

The ArUco library (described in Chapter 2.3) is applied to produce relative and absolute measurements. It enables the *TurtleBot* to obtain relative pose information of other agents with the ordinary 2d-image stream of the Kinect. Likewise, an overhead camera is used to detect the pose of the agents in an absolute manner. It is mounted on the room's ceiling and points downwards towards the floor. Motivations for the library are the facts, that it recognizes also the identity of detected robots and that it is suitable for outdoor environments.

First of all one has to install the OpenCV library. This is not described in more detail, cause there are many related tutorials. Less intuitive is the set-up of the ArUco library. It must be downloaded [9] and extracted in a local folder. Then, one has to use a terminal and change the path to the according directory, create a */build* folder and change further into this one.

```
$ cd PATH/TO/ARUCO/LIBRARY
$ mkdir build
$ cd build
```

Inside the */build* path the library can be compiled and linked with the following commands.

```
$ cmake ..
$ make -j4
$ sudo make install
$ sudo updatedb
```

For ROS distributions older then *Indigo* one has to add the following line to the *CMakeLists.txt* file to use an improved compiler.

```
set(CMAKE_CXX_FLAGS "-std=c++0x ${CMAKE_CXX_FLAGS}")
```

The text file also has to include the library and connect it to the nodes that depend on it.

```
include_directories(
include
coop_localization/include
...
${OpenCV_INCLUDE_DIRS}
${aruco_INCLUDE_DIRS}
...
)

add_executable( rel_meas src/rel_meas.cpp)
target_link_libraries( rel_meas
${OpenCV_LIBS}
${catkin_LIBRARIES}
$ENV{aruco_LIBRARIES}
)

add_executable( abs_meas src/abs_meas.cpp)
target_link_libraries( abs_meas
${OpenCV_LIBS}
${catkin_LIBRARIES}
$ENV{aruco_LIBRARIES}
)
```

Listing 5.1: Linking of the ArUco library for compiling.

In order to receive correct measurements, it is required to calibrate the Kinect and the webcam. Therefore, a checkerboard with black and white squares is placed in front of the camera and a series of pictures are captured showing the board in several different poses. They differ in orientation, distance and position. An example of a close position is shown in Figure 5.1. The images and the given data of the checkerboard are consequently used for an OpenCV related library, called *opencv_cam_calibration* [10] to determine the distortion of the lens. For starting the executable, one has to change in its */bin* directory and execute the following command:

```
~/catkin_ws/src/opencv_cam_calibration-master/build/bin$ ./cam_calib ../../data/
    ↪ camera_conf.xml
```

The resulting parameters of the camera are saved in a text file and can later be read by the ArUco image processing program. However, the resulting file matches not exactly the required format. The parser is case sensitive for capital letters and one has to rewrite most of the key words with lower case letter. As template, one can use the sample configuration file from the ArUco library.

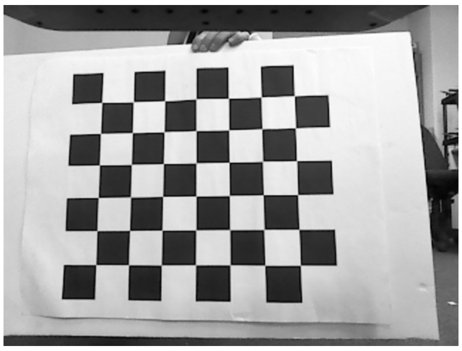


Figure 5.1: One example of the image series, which is recorded for the calibration of the Kinect.

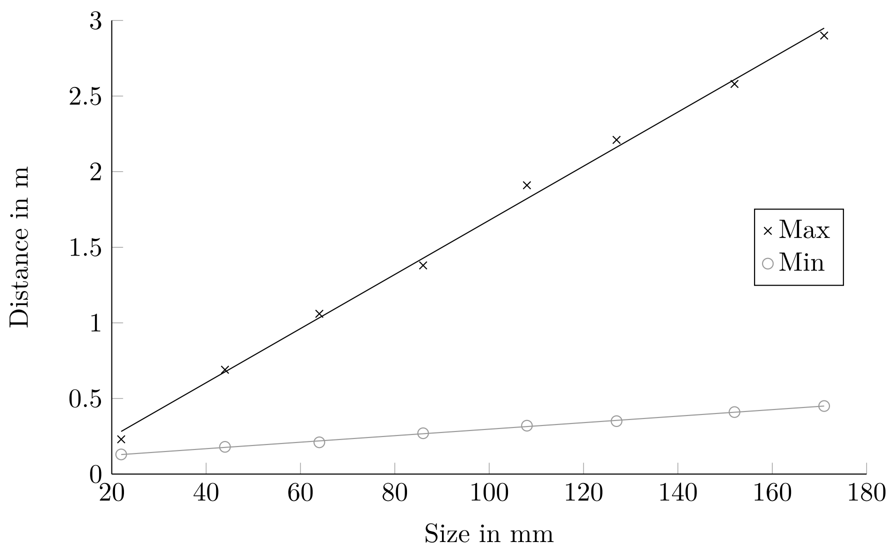


Figure 5.2: Minimal and maximal detectable distance between the centers of two TurtleBots for different sizes of ArUco marker.

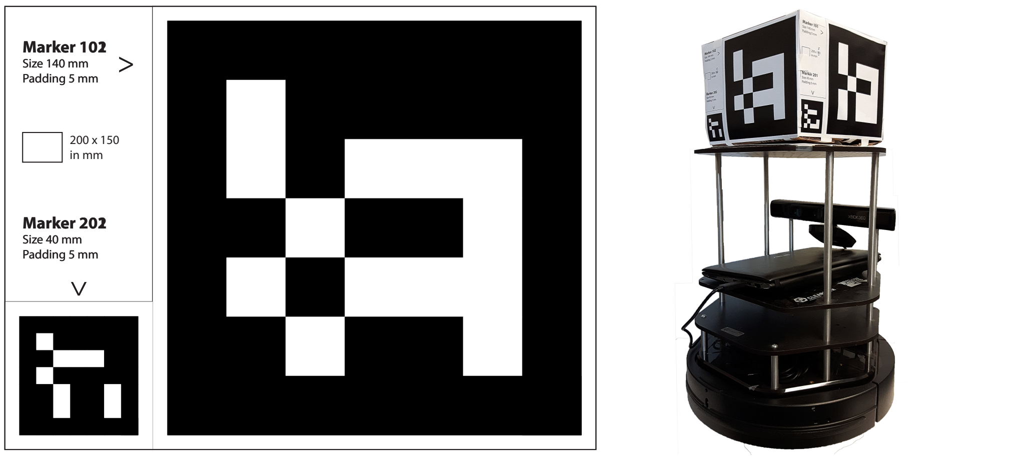
5.2 Fixture for ArUco markers

It has been agreed to attach the marker on top of the *TurtleBot*'s rack, because there is enough space and it does not interfere the view of the own camera. A cube is especially suitable, since one can provide tags in every horizontal direction and also on top of the cube.

In order to find the optimal size of the tags, a series of measurements with different marker-widths is developed. This study is used to determine the range between two *TurtleBot*s, in which the marker can be detected. Close distance is on the one hand limited, since the camera has to capture the entire marker; far distance on the other hand, since the resolution of the marker in the image becomes too low. Thus, it is required to find the right trade-off for the marker size.

For this experiment, a series of ArUco markers, generated according to Chapter 2.3, are printed and, one after each other, attached on the cube. A second agent is used to measure the limits. The maximal and minimal distances between the center of two *TurtleBot*s are plotted in Figure 5.2. One can see a direct proportional behavior of the resulting values to the size of the marker. However, the slope of the max-curve is stronger than the one of the minimal range. Consequently, it would be beneficial to have a marker as large as possible.

In order to achieve a wide range and to not lose the ability to detect close agents, a concept is developed, which uses for every horizontal side two markers of different size (See Figure 5.3). The cube on top of the rack is a modified cardboard box and its footprint is a square with 200 mm to optimally utilize the space of the *TurtleBot*'s top. Every horizontal side has the dimensions 200 mm x 150 mm and contains one marker of 40 mm and one of 140 mm next to each other. Both tags have a padding of 5 mm and thus, cover the full length of the side. Besides, the remaining space on the paper is used to displays the markers' information. According to the study, this concept enables to measure distances from approximately 0.2 to 2.2 m. The possibility that more than one marker is detected for one certain relative measurement is considered in the implementation.



(a) Sign with two ArUco tags of different size. (b) Final marker box on top of the robot.

Figure 5.3: Fixture for the ArUco marker.

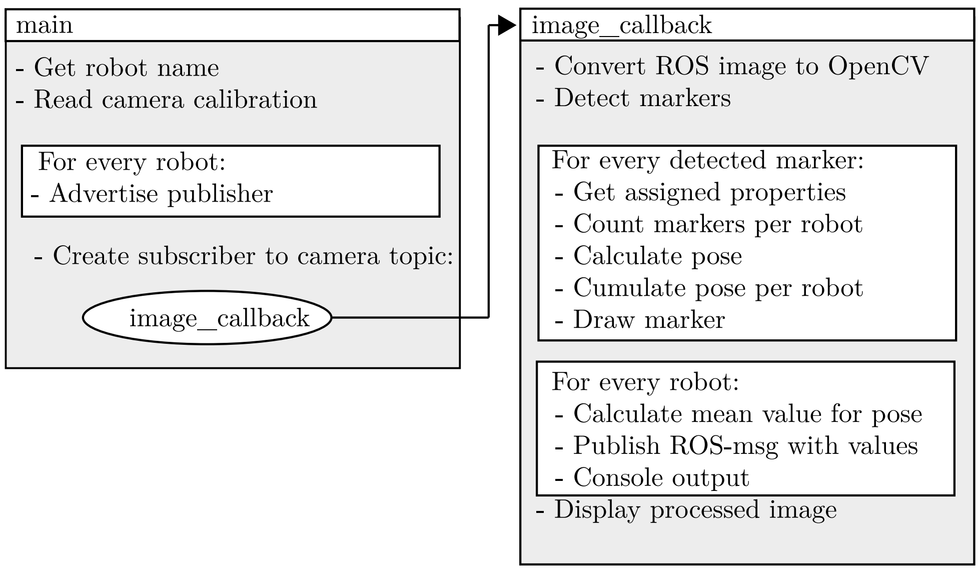

main
- Get robot name
- Read camera calibration
For every robot:
- Advertise publisher
- Create subscriber to camera topic:
image_callback
image_callback
- Convert ROS image to OpenCV
- Detect markers
For every detected marker:
- Get assigned properties
- Count markers per robot
- Calculate pose
- Cumulate pose per robot
- Draw marker
For every robot:
- Calculate mean value for pose
- Publish ROS-msg with values
- Console output
- Display processed image

Figure 5.4: Overview of the script for relative measurements.

5.3 Software nodes

5.3.1 Measurement

The implementation is as well established inside the */src* folder of the new ROS package. There is one C++ script for the relative and one for the absolute measurements. Additionally, the header file *name_map.h* is created in the /include folder of the package. Its purpose is to list the names of the team member and to provide information for every tag identity. Most of the scripts will include this database and in this way, one can maintain it from one point.

As depicted in Figure 4.1 the relative script will run locally on every agent. Its task is to execute the relative measurements. Figure 5.4 gives an overview of the program structure, which is divided in two functions.

The first one is called *main* and is the designated start of the C++ program. First of all, the name of the robot is received via the ROS parameter, which is set by the launch file. The next step is to read the previously created calibration file of the Kinect, which is introduced in the function for the ArUco detection. Then, a loop advertises a ROS publisher for the central processing unit. Lastly, one subscriber is created for the image topic of the Kinect camera. Thus, every time the camera publishes a new frame the image-callback function, which will be explained in the following, is invoked to processes the incoming picture.

The callback function first converts the data into an OpenCV format. Afterwards, it runs the an ArUco function to detect all markers and store them in the dynamic list *Markers*. Its length is equal to the number of detected tags and contains the measurement of every instance.

A first loop runs through every detected marker. The ID of every marker is used as a key for the *robot_name* map to receive the assigned properties, which consists of three levels. The first value provides information about the name of the owner, the second one about the side of the cube and the third one about the size of the tag. At the beginning, it is not evident how many of the detected tags belong to a certain robot-to-robot relation. Therefore, the tags are counted during the run of the loop in one variable per robot. The vector of the relative pose is accumulated and as well saved for every detected agent.

For every iteration of the loop the pose of the corresponding marker is calculated according to equation 5.1. The variables are described in the following.

$$
\begin{aligned}
x_i &+= x_t * s_t + s_{box}/2 * cos(\gamma_t) - o_{sign} * sin(\gamma_t) - o_{kin} \\
y_i &+= y_t + s_{box}/2 * sin(\gamma_t) + o_{sign} * cos(\gamma_t) \\
\gamma_i &+= \gamma_t + side_{box} * \pi/2 \\
&\quad \text{With subscripts: i = agent, t = tags.} \\
&\quad \text{and "+ = " = accumulation.}
\end{aligned}
\tag{5.1}
$$

The pose of the tag (x_t, y_t and γ_t) is received via a code snipped of the *ros_aruco* script [11]. Only two translational variable and one rotatory variable are required, since the state vector has three degrees of freedom. Nevertheless, the coordinates have to be reassigned in a different order to respect the ROS conventions, which state that the x-axis points to the front of the robot, the y-axis to the left and the z-axis to the top. Furthermore, the code snippet is intended for an outdated distribution of the ArUco library. In the newer version the arrangement of the rotation matrix changed and thus, the snippet has to be adopted to select the correct index.

Then, the properties of the *robot_name* are applied to get the transformation from the center of one to the other robot. The orientation can be corrected with an angular offset, depending on which side ($side_{box}$) of the cube the tag is located; the translation by the relative position of the tag inside the sign (o_{sign}) and the size of the box (s_{box}). Furthermore, one has to respect the offset from the Kinect (o_{kin}) to the center of the robot and the size of the tag (s_{tag}), which scales the normal distance between the two robots.

Hence, for every iteration of the loop and therefore every detected marker, a pose is attained, given in the coordinate system of the observer and composed of the position (x, y) and the orientation (γ). As already mentioned, this values are accumulated during the loop, separately for every robot.

Afterwards, a second loop is used, which runs through every robot of the team. At this point it is known, how many tags per agent (n_i) have been detected and thus, the mean values for the pose

($\overline{x}_i$, $\overline{y}_i$ and $\overline{\gamma}_i$) can be calculated according to Equation 5.2.

$$\begin{aligned}\overline{x}_i &= x_i/n_i\\ \overline{y}_i &= y_i/n_i\\ \overline{\gamma}_i &= \gamma_i/n_i\\ &\text{With subscript: i = agent.}\end{aligned} \tag{5.2}$$

Finally, the results are stored in a ROS message and published on the advertised topic. It also includes the robot's name and the one of the detected agent. After both loops of the callback function are finished, the processed image is displayed in an OpenCV window. The displaying is however not required and could be skipped to save processing power.

The entire c++ script is attached in Listing A.5, which gives a more detailed insight of the programming.

It is favorable to run the script directly on the agent's computer. In this way, one can reduce the lag of the image processing. The information has not to be sent across the Wi-Fi network, since the performance of the image stream depends on the subscribing nodes. Likewise, there would be a higher delay, if multiple programs would read out the same topic.

The script for the absolute measurement is constructed similar as the previous program. However, some parts can be simplified. First of all, the detected center of the marker is legit and does not require any further offsets. Besides, it is not possible to detect more than one tag per robot and so, there is no need to calculate a mean value. Therefore, one can skip the process of accumulation and the second loop. Finally, the name of *Agent A* is set to the constant string *abs* to indicate that the measurement is derived from the overhead camera.

5.3.2 Sensor fusion

The beginning of the c++ script includes all necessary header files and libraries. One in particular is the template *Eigen*, which is applied to make the handling of matrices more convenient. It has to be installed on the system and included in the CMakeLists.txt file (Compare Listing 5.2). The features of this template are additional class types for matrices and vectors. One is also able to apply basic mathematical operations on this systems and functions like inverse or transpose. Furthermore, a header file called *ekf.h* (Compare Appendix A.6) is included, which includes most of the programs function.

```
include_directories(
...
${EIGEN_INCLUDE_DIR}
)
```

Listing 5.2: Changes in the CMakeLists.txt script to apply the Eigen library.

As usual for ROS, the script has a main function, which starts by initializing the communication interface. The sensor fusion subscribes to two topics with individual namespace. On the one hand, there is the topic for odometry (*/odom*), which is used for the propagation stage. The odometry of the robot is published with a very high frequency, but the according query of the sensor fusion is limited to 10 Hz. The callback function reads out a part of the message called *Twist* in order to acquire the current linear and angular velocities. This values are then handled in the *motionPropagation* function of the main loop. With the current orientation and trigonometry, the velocity is transformed in the global coordinate system to get the values for x and *y-direction*. Finally, they are multiplied with the duration of the time step and added to the previous states. Thus, the movement is propagated with a constant velocity for the time increment of 0.1s. Also, the covariance is adjusted, as described in Equation 3.3 - 3.6 of Chapter 3.2.

Equation 5.3 is used to calculate the error constants of the proprioceptive measurements retrieved from the wheel encoder. The formula is proposed by paper [12]. Whereby, variable a is the distance between the wheels and σ is the standard deviation of the wheel encoder.

$$\begin{aligned} \sigma_v &= \frac{\sqrt{2}}{2}\sigma \\ \sigma_w &= \frac{\sqrt{2}}{a}\sigma \end{aligned} \tag{5.3}$$

On the other hand there is the */update* topic. An update is only executed if the node could receive a new measurement on the subscribed topic. In general, the nodes for the relative and absolute measurement have a smaller frequency, then the one of the sensor fusion. The given message contains the update, which is published for every member by the central processing unit, as soon as a measurement occurs. The message is composed of two parts as pointed out by the algorithm, given in Equation 3.14. The according callback function rearranges the incoming data into the classes of the *Eigen* library. Two loops, one for the row and one for the column, transform for example the one dimensional array of *update_2* in a two dimensional matrix. Afterwards, the *sensorUpdate* function is executed to apply the received data for the update stage. Each of the updates gets combined with the matrix Φ. Then, the result of *update_1* is added to the state vector and the one of *update_2* to the covariance matrix.

In the last step of the main loop, the outcome of every iteration is published on a topic called */PDEKF*. In advance to communication, the matrices must again be rearranged into arrays to be suitable for the ROS message. Besides, the data is recorded into a text file. Every iteration represents a line of the file and contains the following data per time step:

- Time stamp
- States
- States with pure propagation
- Covariance matrix
- Linear and angular Velocity
- Flag, indicating if the agent is involved in a measurement

Last but not least, the script offers the ability for other nodes of the system to request the current states and covariance of the agent via a ROS service. The function returns a boolean to express the success of the service. Apart of this, it sets the flag *involved_in_rel_meas_this_update* to indicate, as already mentioned, that the agent is involved in a measurement.

To sum up, every robot is supposed to run the sensor fusion script, in order to estimate its states and the connected certainty. The filter requires an additional central node, which is described in the following chapter.

5.3.3 Central processing

The script is highlighted in Appendix A.7. Similar to the last script, there is a header file included, which is called *sensor_fusion.h*.It holds the required variables and functions. The initialization of the main function comprise the ROS communication and the definitions of several matrices. Furthermore, it creates again a text file for recording with the following content.

- Time stamp
- Agent A
- Agent B
- Measured relative pose
- Estimated relative pose

The following loop runs with a frequency of 2 Hz and mainly consists of the function *calculate_updates*. The execution of the function is skipped if the responsible flag does not indicate a new measurement.

There is one essential callback function, evoked by the */central_processing/rel_meas* topic. It reads out the involved agents and the measurement itself. Thereby, the robot's name that took the measurement is *Agent_a* and the one that was been observed *Agent_b*. Likewise, there are the integers *a* and *b*, which are more suitable for indexing of the variables. If the data arrives from an absolute measurement, both indexes are set to the value of *b*.

To get all of the required data, the program makes use of the *request* service. Depending, on weather an absolute or relative measurement arrived, one or two agents are addressed. With regard to this, it should be noted again that Figure 4.2 provides an overview of the nodes, as well as the sequence of actions for an measurement event. To improve the performance of the filter there are some more flags for the different levels of the data acquisition. *Calculating* prevents the node to receive a measurement, while being busy with calculating; *gathering_data* skips incoming messages, while being busy with the gathering of data; and *state_already_received* avoids redundant request for the case that one robot is involved in more than one measurement. In this way, it is assured that a variable is not used simultaneously for different callback invocations.

One last boolean, called *success* returns *true* if the requests have been successful communicated. The variable would become *false*, if a robot is not started or not reachable. Only, if all this logical barriers are fulfilled, the actual measurement values get stored. The designated variable *Z* has the type vector and is a two dimensional array. Thus, one can save for every possible combination of agents (*a* and *b*) a vector containing the three states. The special cases that have the same indexes, resemble absolute measurements.

The function *calculate_updates*, which is called by the main program, subsequently processes the measurements. With two loops it checks every entry of matrix *Z*, if values are available and calculates an update per present measurements. Therefore, two different case are distinguished. The first one is for absolute and second one for relative measurements. Fur this purpose, an *if* statement retrieves, if the two indexes are equal.

For the case, that they are unequal, the equations for the relative processing are selected. First of all, the delta of two agents' states is calculated. Afterwards, the delta can be used to determine the current relative pose. It is transfered in the reference frame of *Agent A* by Equation 3.10. In order to keep the angle in the range of $[0, 2\pi]$ a modulus of 2π is applied on the resulting value. Then, the residual can be calculated by subtracting the new relative pose vector from the measurement. At this point it is important again to look at the angle. By introducing the calculations in Listing 5.3 one can respect that the angular range is arranged in a circle. The basic idea is that the correction of the orientation should always work in the shortest direction. If the residual is e.g. 300 deg, it is faster to minimize the difference by going in the other direction. The resulting value is $300\,\text{deg} - 360\,\text{deg} = -60\,\text{deg}$ and thus, the absolute difference is reduced tremendous. But even more important is the prevention of discontinuous jumps from either 0 deg to 360 deg, or the other way round. Since, all of the three states are in correlation, this would also lead to a strong change of the other two degree of freedoms and the whole filter would become instable.

```
if (abs(r_a[2])>pi) {
  sign = r_a[2]/abs(r_a[2]);
  r_a[2] = r_a[2] - 2*pi*sign;
}
```

Listing 5.3: Incorporate that the range of the angle is arranged in a cricle.

Afterwards, one can calculate $\mathbf{S}_{a,b}$ with the Equations 3.11 - 3.13. By comparing the update message in 3.14 and Equations 3.15 and 3.16 one can see that calculations can be simplified. One can avoid the nontrivial implementation of the matrix-exponentiation $\mathbf{S}^{-1/2}$. This term is presented twice in each of the update message and can consequently be combined to $\mathbf{S}^{-1}$, which is equal to a simple inverse-operation. Therefore, $\mathbf{\Gamma}$ and $\bar{\mathbf{r}}^a$ are calculated with neglecting the $\mathbf{S}$ term. Instead, the updates of 3.14 is changed to the following:

$$\begin{aligned} \mathbf{Update1} &= \mathbf{\Gamma}_i \mathbf{S}^{-1} \bar{\mathbf{r}}^a \\ \mathbf{Update2} &= \mathbf{\Gamma}_i \mathbf{S}^{-1} \mathbf{\Gamma}_i^T \end{aligned} \tag{5.4}$$

If both indexes are referencing to the same agent, the absolute procedure is initiated. One more time, it is the simplified form of the relative one. The residual can be directly calculated with the measured and the estimated pose, since only one agent is involved. This also simplifies the calculation of $\mathbf{S}_{a,b}$, $\mathbf{\Gamma}$ and $\bar{\mathbf{r}}^a$.

Afterwards, the case differentiation is removed and both types of measurement take the same actions. The next step is to renew the Π matrix according to Equation 3.2, which is saved in a two dimensional array (compare Table 4.3) and is referenced by the variables a and b.

In the end of the valid loop cycle, the update is calculated and published to every agent. After running through every index of the measurement arrays, the *calculate_updates()* function is finished. Last but not least, the measurement arrays get cleared, so that it can be filled up again in the next time step.

With the *sensor_fusion* and the *central_processing* nodes, the EKF is finally completely implemented. For debugging, measurements have been suppressed and overwritten with constant values. In this way it is possible to compare the result one-to-one with the Matlab script.

5.3.4 Visualization

The purpose of the last node is to picture the results during the run in the dedicated program Rviz, which is the default visualizer of ROS. It has many tools for displaying sensor data and state information in a virtual 3D environment. Compared to the other scripts so far, the new node is written in the language *Python*. However, each agent runs again the same program. With the help of namespaces, they are subscribing to their final *PoseWithCovariance* message on the according topic. Thus, the callback function retrieves the information of the pose and the covariance. For a start, one has to transform the covariance matrix into the form of a standard deviation (sigma). One could determine sigma by the according entry of the covariance matrix. The square root of the first value would for example be equal to the sigma of the position in x-direction. However, this is in fact not correct, since the correlations between the stats are neglected. Hence, one has to transform the matrix to its principal axis by calculating its eigenvalues. In this way a diagonal matrix is generated and one can finally retrieve the correct standard deviation through the square root of their entries.

The confidence of the current position is then usually presented with the 3-sigma method. This means that $3 * \sigma$ is once added to the corresponding state and once subtracted from it. By doing so, one create two curve that are enveloping the plot of the states over time. This method states that the real position is situated with a certainty of 99.7% inside this limits. Applied to both dimensions of the position yields the so called circle of confidence. In e.g. a x-y-plot this ellipse could be drawn theoretically around any position and gives again a certainty of 99.7% that the real value is within this area.

The goal of the node is therefore, to visualize the current position and the ellipse of confidence. Hence, the Python script calculates the 3-sigma values as described. Afterwards, three so-called *markers* are published, which can be interpreted by the Rviz viewer. The first one is of type *cylinder*. The parameter of the position/orientation is set according to the state values; and the one of the scale according to the 3-sigma value. The next *marker* is of type *arrow* and is used to highlight the current orientation of the robot. Last but not least, there is one of type *text*, which is applied to show the robot's names next to ellipse. The final result is shown in Figure 5.5. On the left side is the camera image of the top mounted camera and on the right side the according visualization in Rviz.

Moreover, the described 3-sigma methods are used for the post-processing of the results from the experiments in Chapter 6.

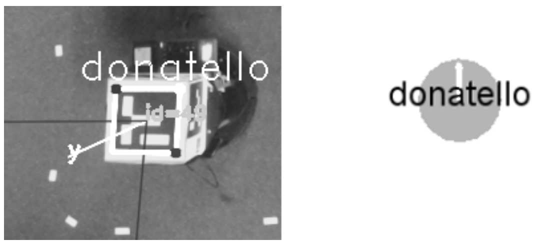


Figure 5.5: The left side shows the image from the top mounted camera and the drawing of the detected marker. The right side shows the visualization of the resulting pose in Rviz.

6 Tests and Evaluation

6.1 Propagation stage

The first test run is without any measurement updates to test the propagating part of the filter, which runs with a frequency of 10 Hz. The mobile platform starts with a predefined pose, received beforehand, via the top mounted camera. This values are manually set to the launch file. Additionally, the variances are initialized with a value of 0.000001 in order to represent a very high certainty about the initial position. As shown in Figure 6.1, there is one robot, which is moving on a square line. The resulting trajectory is different, since only the corners of the square are sent as goal positions. In between this key points the path is undefined and linear and angular movements are executed simultaneously.

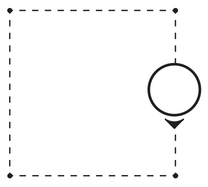


Figure 6.1: Test run to examine the propagation stage.

Figure 6.2 is a plot of the x and y coordinates and shows the propagated position of the robot in comparison to the reference measurements of the top mounted camera. The two given paths start similar, but while moving, they dive in the course of time. Furthermore, the plot features some confidence ellipses to highlight the variance of the position at certain points in time. The axis of the ellipse are determined by the 3-sigma method. Further information about the meaning and the calculation of sigma can been found in Chapter 5.3.4.

The plots in Figure 6.3 reveal the values for all three states and their 3-sigma interval. Also, Figure 6.4 gives an overview of the different sigmas. One can see that the uncertainty is increasing continually over time and that the measured values are situated within the confidence limits.

The test run for the propagation state indicates that the pose measurements through the wheel encoder are not flawless. The reasons therefore are e.g. the slip of the tire and the Linearization. Since, the uncertainty sigma is connected to the velocity, the error is steady increasing, while the robot is moving. This connection can be seen in Equation 3.3, which indicates that the calculation of the covariance matrix consists of two terms, both relying on the velocity of the robot. The test run highlights that the parameter $sigma_{encoder}$ is chosen correctly, since the measurements are bounded within the intervals of both, the ellipse of confidence and the enveloping 3-sigma curve.

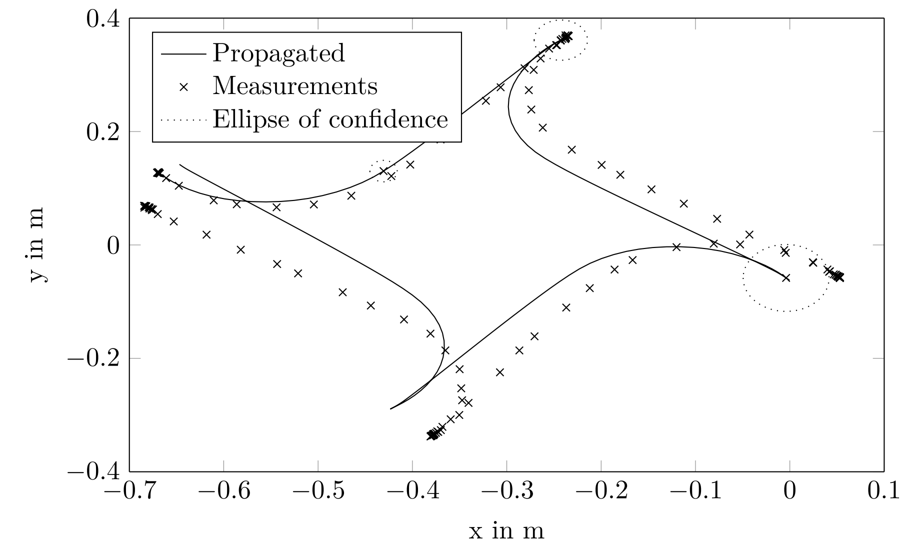


Figure 6.2: X-y-plot for the test run of the propagation stage. One robot is moving to the corners of a square. An ellipse of confidence is given for some positions of interest.

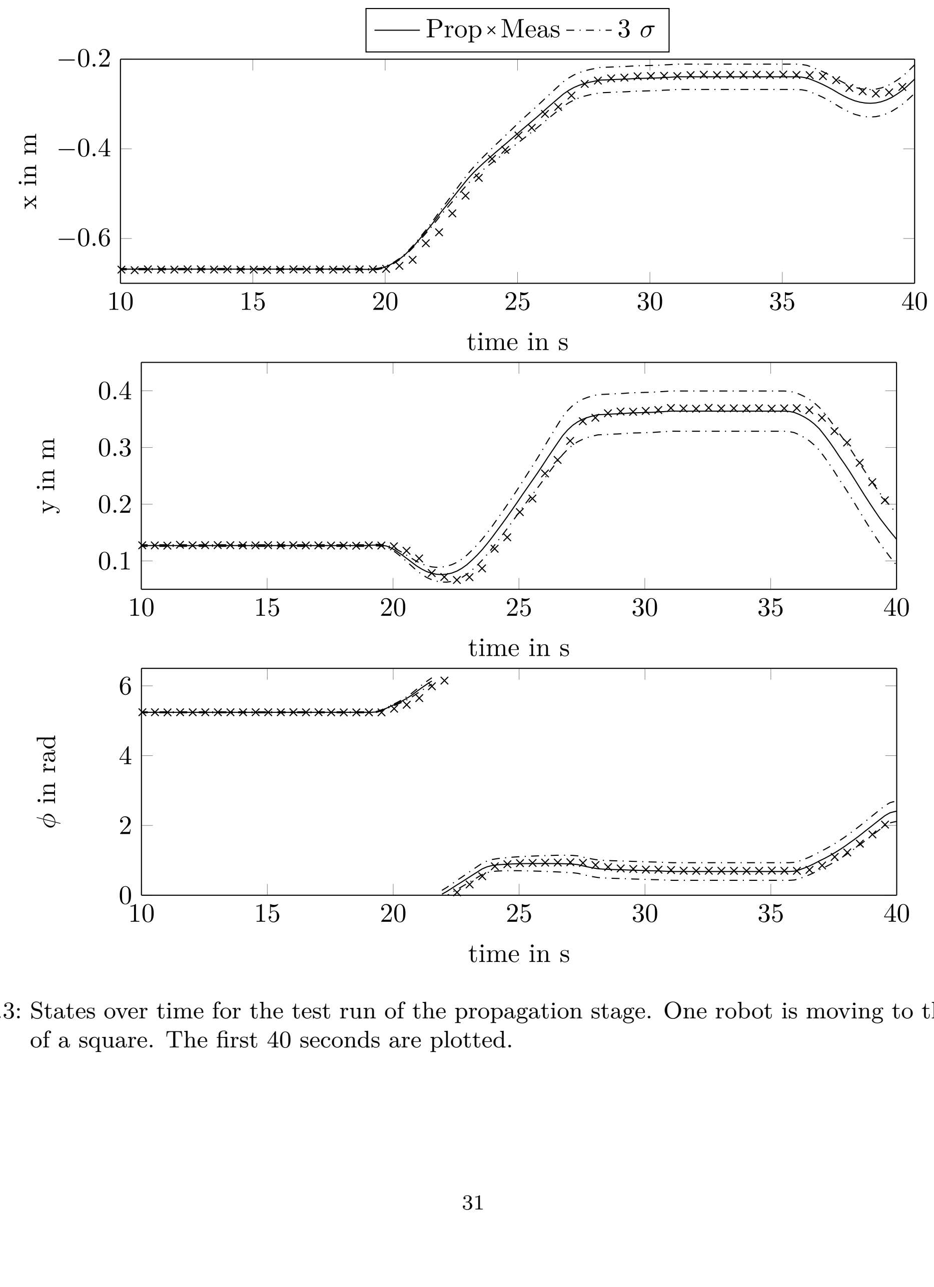


Figure 6.3: States over time for the test run of the propagation stage. One robot is moving to the corners of a square. The first 40 seconds are plotted.

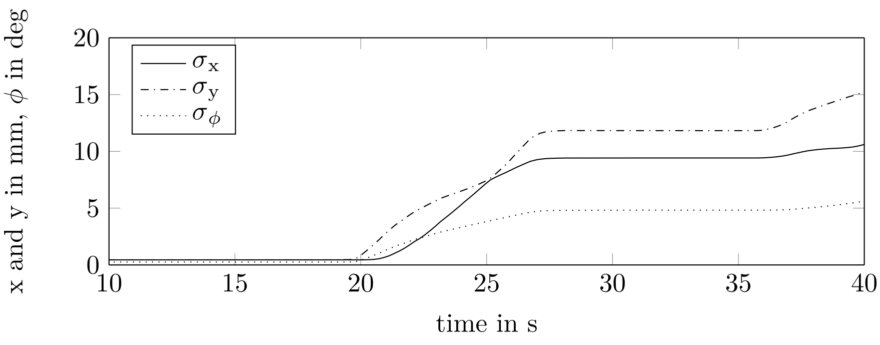


Figure 6.4: Sigma over time for the test run of the propagation stage. One robot is moving to the corners of a square. The first 40 seconds are plotted.

6.2 Global update

In the next step the incorporation of the global update is examined. As illustrated in Figure 6.5 one robot is again moving in a square. However, this time the global measurements of the top mounted camera are applied to the EKF filter. Two test runs with variations of the update frequency are executed. There is one with 2 Hz and another one with 0.2 Hz, while the frequency for the propagation is still left by 10 Hz. Besides, the initial states are no longer hard-coded, but they are all set to zero with a high uncertainty (Variance of each state is 10000). Afterwards, with the first update the estimated position jumps to the real values and the uncertainty decreases promptly. But it would need a certain amount of updates to reach the value of 0.000001, which is chosen for the first test run.

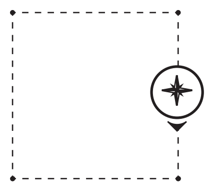


Figure 6.5: Test run to examine the propagation stage with global updates.

The first plot (Figure 6.6) shows again the position in the x-y-plane. Additionally, to the two previous curves, there is a new one for the localization, including the global updates. It can be seen that the trajectory keeps close to the measured points over the whole run. The case with smaller update frequency (Figure 6.9) in comparison is more loosely following the real path, since the update are processed discontinuously.

The x-y plots present that the estimated path is very close to the real path and the end position presents no mismatch. Furthermore, the covariance is no longer continuously increasing. By fusing regular measurement updates the covariance stays bounded, which can be well seen for the states (Figure 6.7) and the sigma overview (Figure 6.8). The second case with low update frequency (Figure 6.9) demonstrates how the 3-sigma interval shrinks for an occurring discrete measurement event. The path is not following the real one consistently, yet also ends up at the right position. The impact of the update can be influenced with the *R(abs)* matrix. It is set to the value 0.0001, which is equal to a measuring accuracy of 10 mm. This is a appropriate quantity, since one has also to consider delay in the system, which has the order of around 0.1 s.

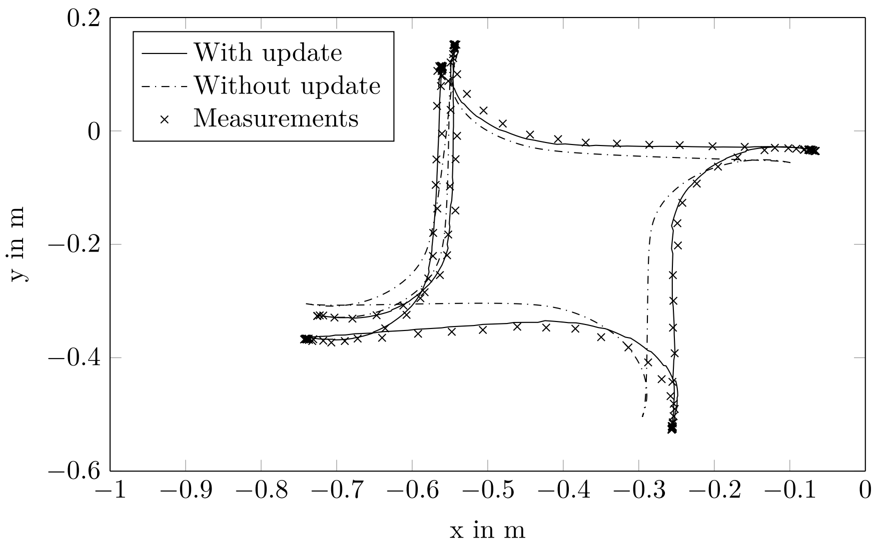


Figure 6.6: X-y-plot for the test run of the global updates. One robot is moving to the corners of a square and receives global updates with 2 Hz.

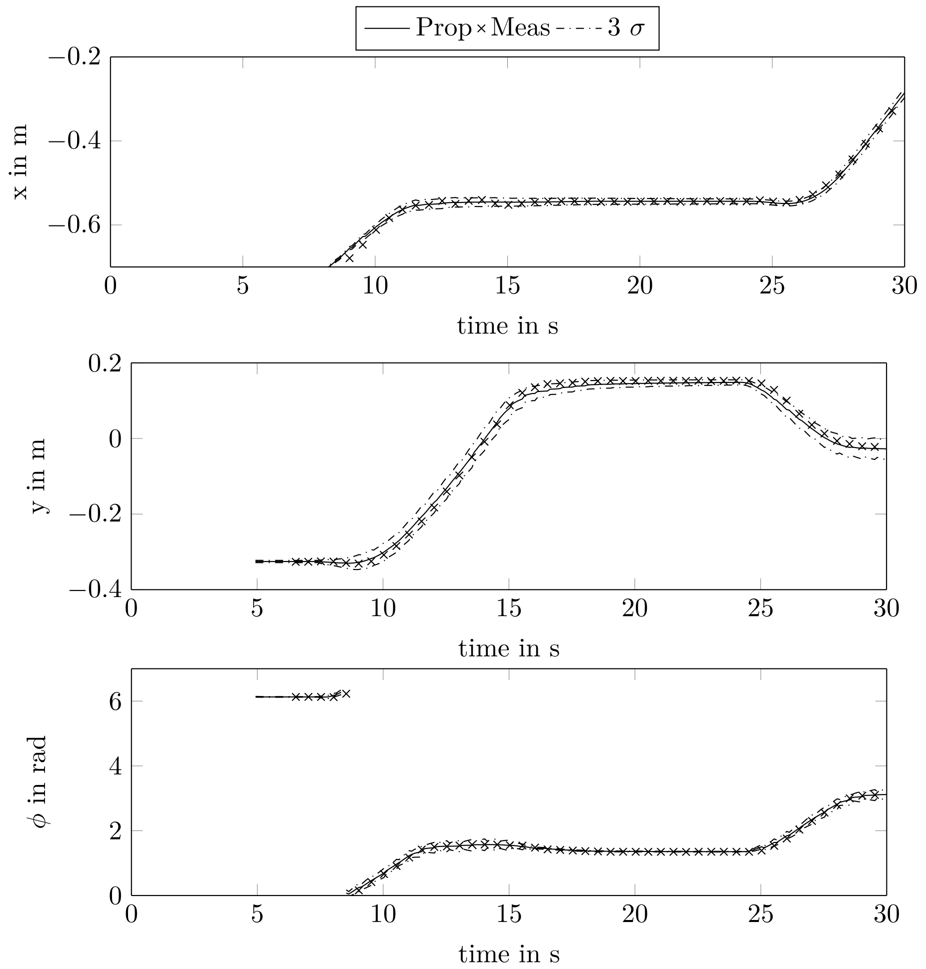


Figure 6.7: States over time for the test run of the global updates. One robot is moving to the corners of a square and receives global updates with 2 Hz. The first 30 seconds are plotted.

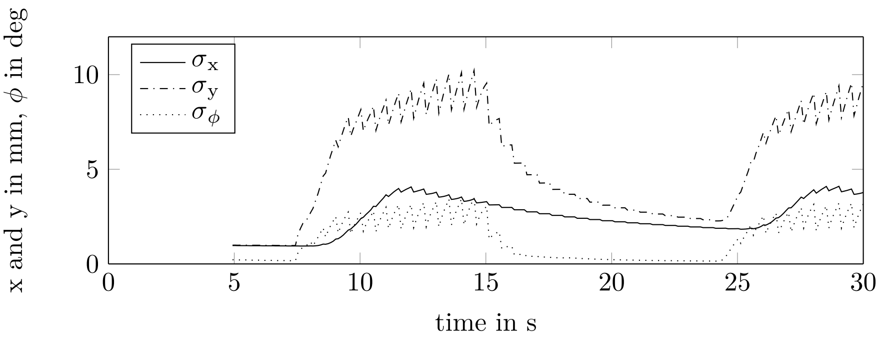


Figure 6.8: Sigma over time for the test run of the global updates. One robot is moving to the corners of a square. The first 40 seconds are plotted.

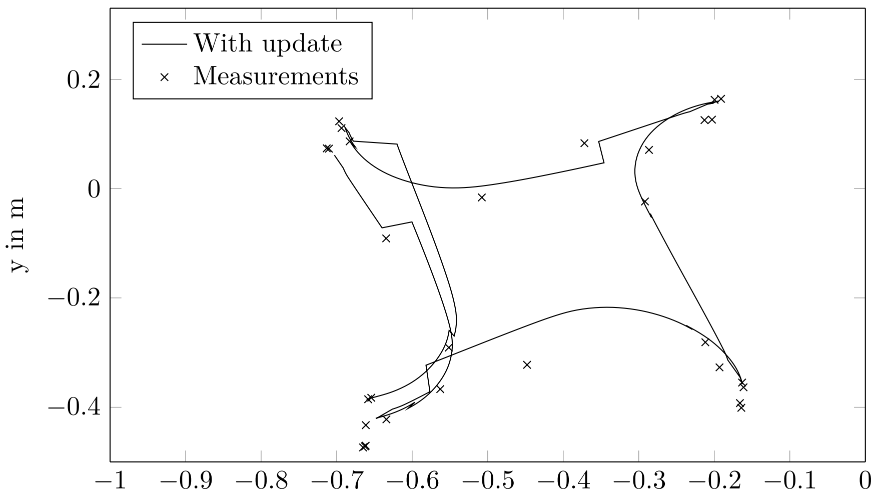


Figure 6.9: X-y-plot for the test run of the global updates. One robot is moving to the corners of a square and receives global updates with 0.2 Hz.

6.3 Relative update

The set up for this test consists of two stationary TurtleBots with a distance of approximately one meter. One robot, *Agent A* is facing another agent, which itself is orientated to the side (Compare illustration of Figure 6.10). Both agents are again initialized with zero values for every state. Consequently, the observing robot initially has a wrong orientation. It beliefs to look in the same direction as *Agent B*, but in reality they have a 90 degree offset. Measurements are performed separately. First, *Agent A* is receiving a global update. After that, the second robot takes the relative measurement.

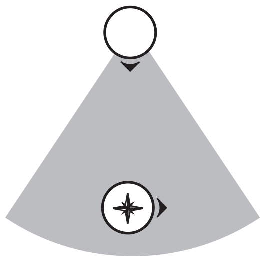


Figure 6.10: Test run to examine the relative update.

As soon as the global update occurs (time: 4 s), the pose of *Agent A* is determined (Figure 6.11) and sigma, demonstrated by the x-direction (Figure 6.12), drops to a value of around 0.01. It keeps decreasing and reaches for this time period a final value of around 0.00025. *Agent B* is not influenced so far. However, by executing the relative measurement (time: 14s), its sigma values decrease, too. After processing the first update the state for the orientation is defined correctly. Nevertheless, the position estimations in x and y target a wrong value and approach the right values only with the subsequent updates. The relative measurement is given in the frame of the observer, but for plotting it is transformed into the world frame, which has the same orientation as the global pose. The relative update reduces the covariance of *Agent B* to a value similar than the one of the observed robot. Meanwhile, *Agent A* acts like a fixed landmark and is hardly influenced by the measurement.

The filter exposes problems with a wrong initial orientation. The detected pose is relative to the observer's frame and consequently, the first position estimation is mistaken. The filter however beliefs that the measurement is correct and reduces its uncertainty. After that, the variance is already so low, that the real value can only be approached by small steps, which is time-consuming and leads to a steady state error. This process shows the typical behavior of a *Kalman Filter*. Once a state is certain, it can hardly be influenced by the update stage.

In order to address the problem of a wrong orientated reference frame, a new feature is created. The first two entries of the diagonal R-matrix for relative measurements are no longer constant, but depend on the current variance of the angle. If the robot is uncertain about its orientation, the position measurement are given less weight. The accuracy is restrained to a constant value of 10 mm after reaching an orientation-certainty of around two degrees. On the one hand, the plots in Figure 6.13 reveal that the position initially still jumps to a wrong value, yet it immediately turns back to the right value and has no steady state error. The variance of the position on the other hand reveals a damped behavior for the settling as one can see in Figure 6.14.

This time the variance is not significantly decreased by the first update. Therefore, the error can be corrected as soon as the second measurement takes place. This is sufficient, since the angle is defined correctly after the first update. Additional measurements will further decrease the variance of the angle and consequently the one of the position. Since, the variable accuracy for position shows an improved localization, it is kept for the following experiments.

There is one more test run with the same setup. The difference this time is that the relative measurement occurs earlier then the global one. As result of the first update, one can observe that the two agents are changing their pose symmetrically around the original values (Compare Figure 6.15). The absolute distance in x-direction of 1 m for example is splitted up on both *TurtleBot*s. One robot changes its location to plus and the other one to minus 0.5 m. Likewise, the angle difference is divided on both agents. In the next sequence, one robot, referenced as *Agent A*, receives a global update, whereupon the estimated pose of both robots changes. The sigmas and therefore the variances are reduced through the relative detection, as one can see in Figure 6.16 by the half and then drop rapidly as the global update is handled.

After the relative interaction, the variance of the resulting belief is smaller than the variances of the individual robot's initial belief. This can be explained with the following: "This is natural, since integrating two independent estimates should make the robot more certain than each estimate in isolation." [5]. But so far, both agents stay uncertain about their pose. Only with the global update event, their sigmas decrease. It is not possible after the relative update that both agents are located in the origin. The estimations change symmetrically in relation to each other, since both agents present the same uncertainty. The final value of *Agent B* looks at first glance different then the indicated relative measurement. After the interaction with the team member, the robot only knows the relation with respect to this agent, but stays unsure about its own global orientation. Therefore, measurements are transferred in a wrong world frame. Compared with the results of the former experiences one can recognize that the steady state values of *Agent B* reach the same values and thus, it is proven that the filter also works in the given order. The filter remembers the correlation of the two agents, which has been gained through their relative measurement. Since, the central processing node is publishing an update to every team member, also agents, which have been previously involved in relation to the robot with the new measurement, improve their localization. The correlations are stored in the Π matrix. Itis initialized undetermined as identity matrix and gets further on adjusted for every relative measurement.

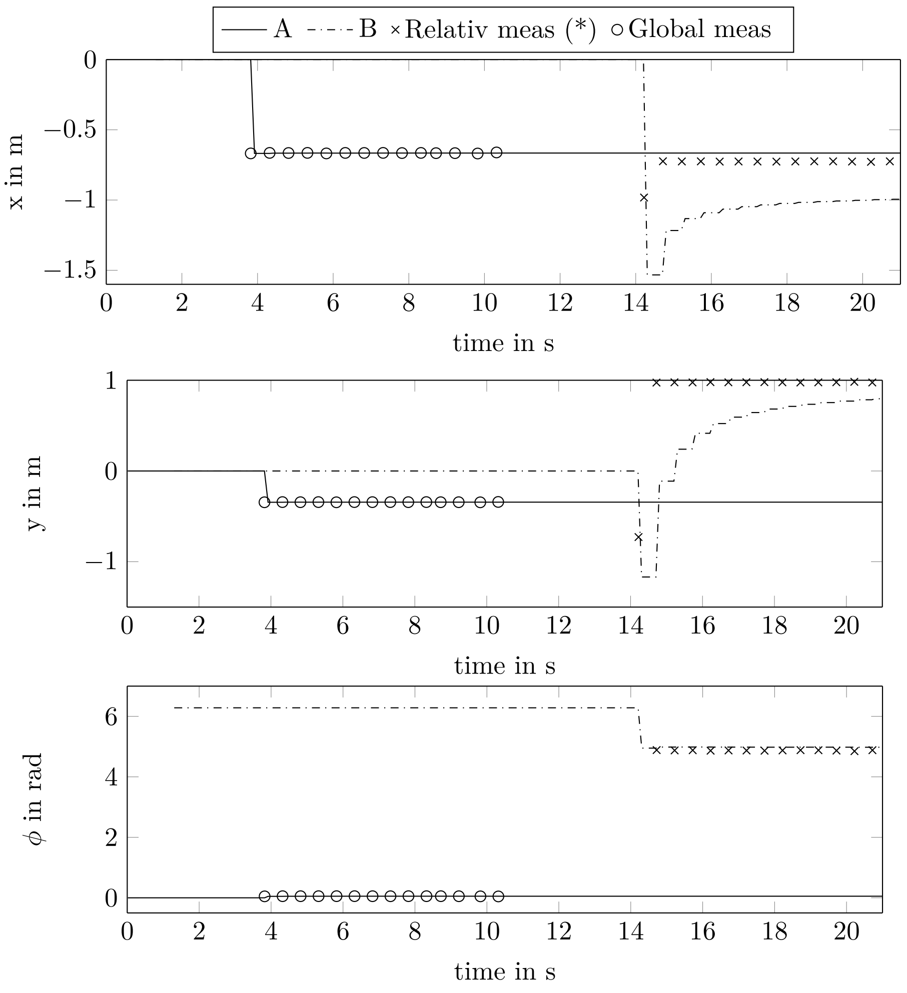


Figure 6.11: States over time for the test run of the relative updates. A global update is followed by a relative one. The accuracy for position measurements is constant. (*)Relative measurements are transformed in the global coordinate system.

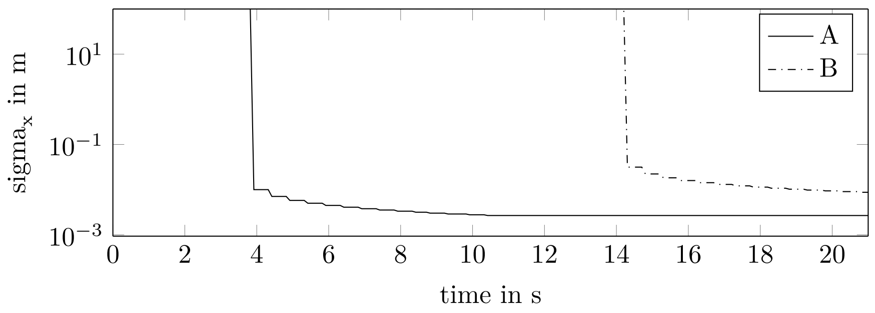


Figure 6.12: Sigma for the x-position over time for the test run of the relative updates. A global update is followed by a relative one. The accuracy for position measurements is constant. (*)Relative measurements are transformed in the global coordinate system.

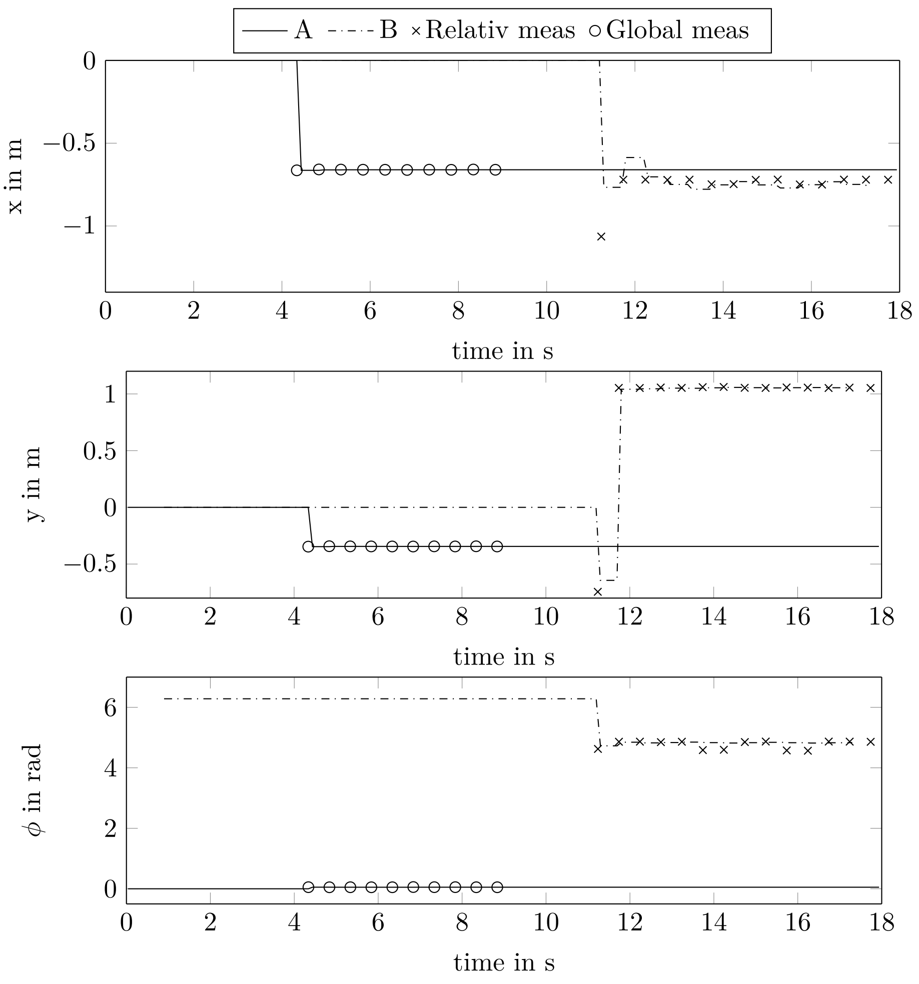


Figure 6.13: States over time for the test run of the relative updates. A global update is followed by a relative one. The accuracy for position measurements is depending on the certainty of the own orientation. (*)Relative measurements are transformed in the global coordinate system.

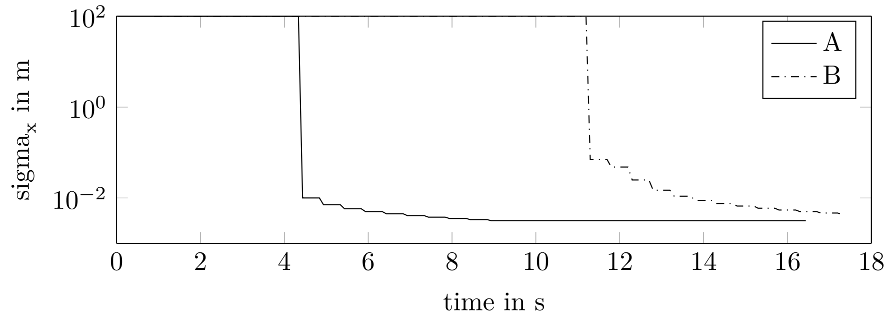


Figure 6.14: Sigma for the x-position over time for the test run of the relative updates. A global update is followed by a relative one. The accuracy for position measurements is depending on the certainty of the own orientation. (*)Relative measurements are transformed in the global coordinate system.

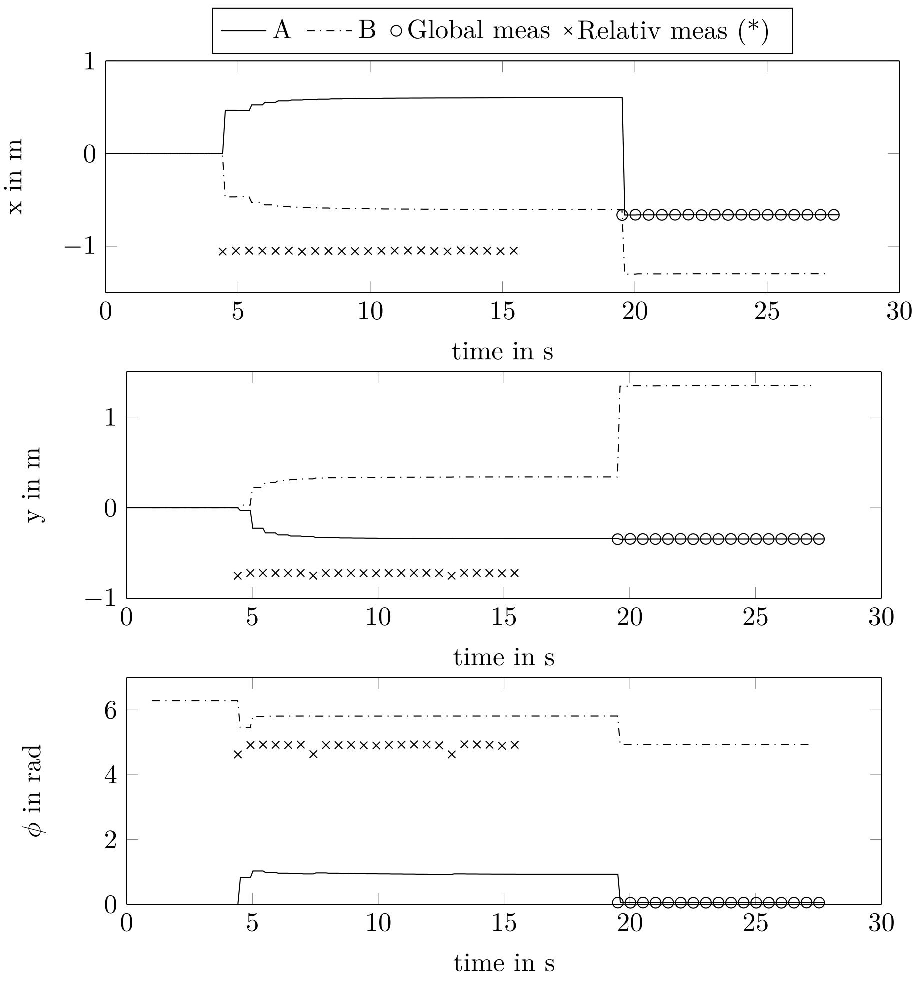


Figure 6.15: States over time for the test run of the relative updates. A relative update is followed by a global one. The accuracy for position measurements is depending on the certainty of the own orientation. (*)Relative measurements are transformed in the global coordinate system.

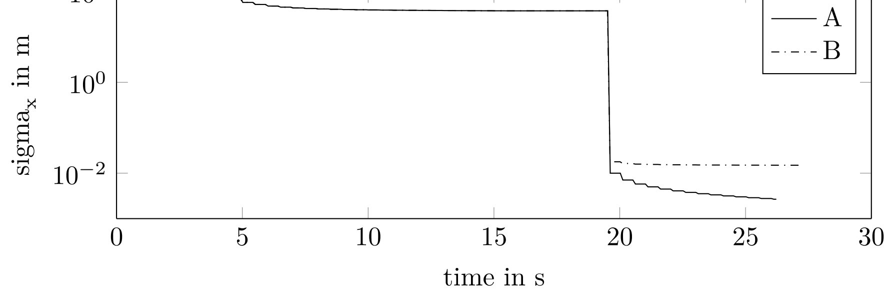


Figure 6.16: Sigma for the x-position over time for the test run of the relative updates. A relative update is followed by a global one. The accuracy for position measurements is depending on the certainty of the own orientation. (*)Relative measurements are transformed in the global coordinate system.

6.4 Entire filter

The next test run investigates the *Kalman Filter*, by combining the relative update stage with the propagation. Therefore, one robot is set in motion by sending a sequence of goal positions, which again reference to the corners of a square. Another agent with a defined pose, previously established by a global measurement, is observing the movements (Compare illustration of Figure 6.17) and is sending relative pose measurements to the central processing unit with a frequency of 2 Hz. Furthermore, there is one more run with a frequency of 0.2 Hz.

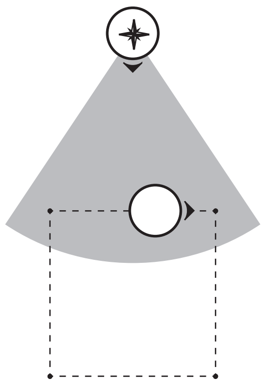


Figure 6.17: Test run to examine the entire filter.

Figure 6.18 displays the trajectory of the moving robot in the x-y-plain. Highlighted is the predicted path with and without relative measurements and the one recorded by the top mounted camera. In the beginning, the observed robot leaves the origin and reaches its real position. Further on, one can see that the result of the filter follows the real positions. There is an offset from the end to the start position, which is neglected by pure propagation. Figure 6.19 shows the estimated states for the progression of the robot. In general both curves lie close together. Nevertheless, the plot reveals some weak points. For example at the time of around 20 s all the curves present some offset to the measured pose values. The most noticeable error is in y-direction with a difference of around 0.15 m.

The result of fusing relative measurements appears not as confident as fusing global ones, yet it brings a significant improvement for the estimated pose. Starting from the initial impact, it continuously corrects the propagation. It is worth to mention that it is more difficult for relative measurements to be permanent present. The observed robot might move out of the point of view or the tag is not detected because of light reflections at unfavorable locations. By that, one can explain the loop at the lower corner. The robot is not detected at the resting position and only gets update during the movements. The test run with lower frequency for the measurements, displayed in Figure 6.20, shows a similar behavior. For the time without cooperation, the estimation might deviate from the real track. However, the present updates are enough for correcting the pose and finally, the robot ends up at the right location.

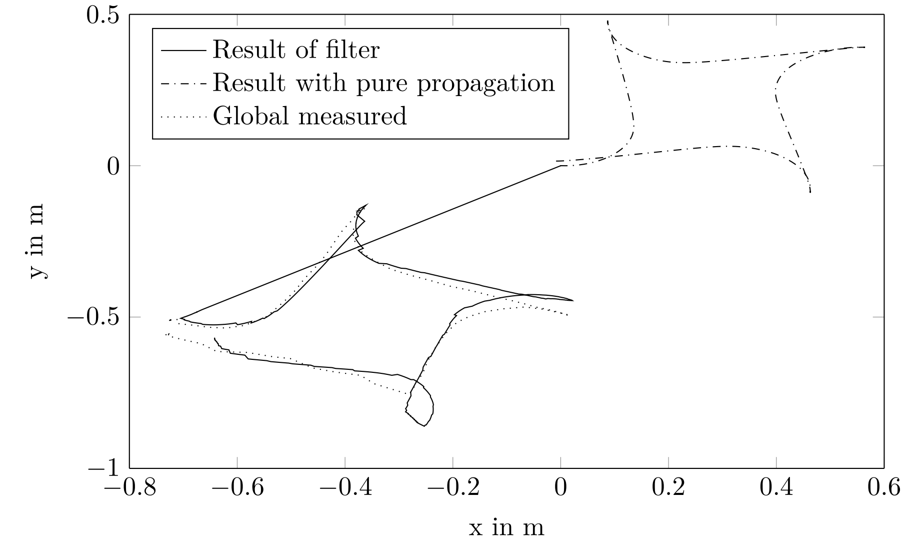


Figure 6.18: X-y-plot for the test run of the entire filter. One robot is moving to the corners of a square and is observed by another agent with a frequency of 2 Hz.

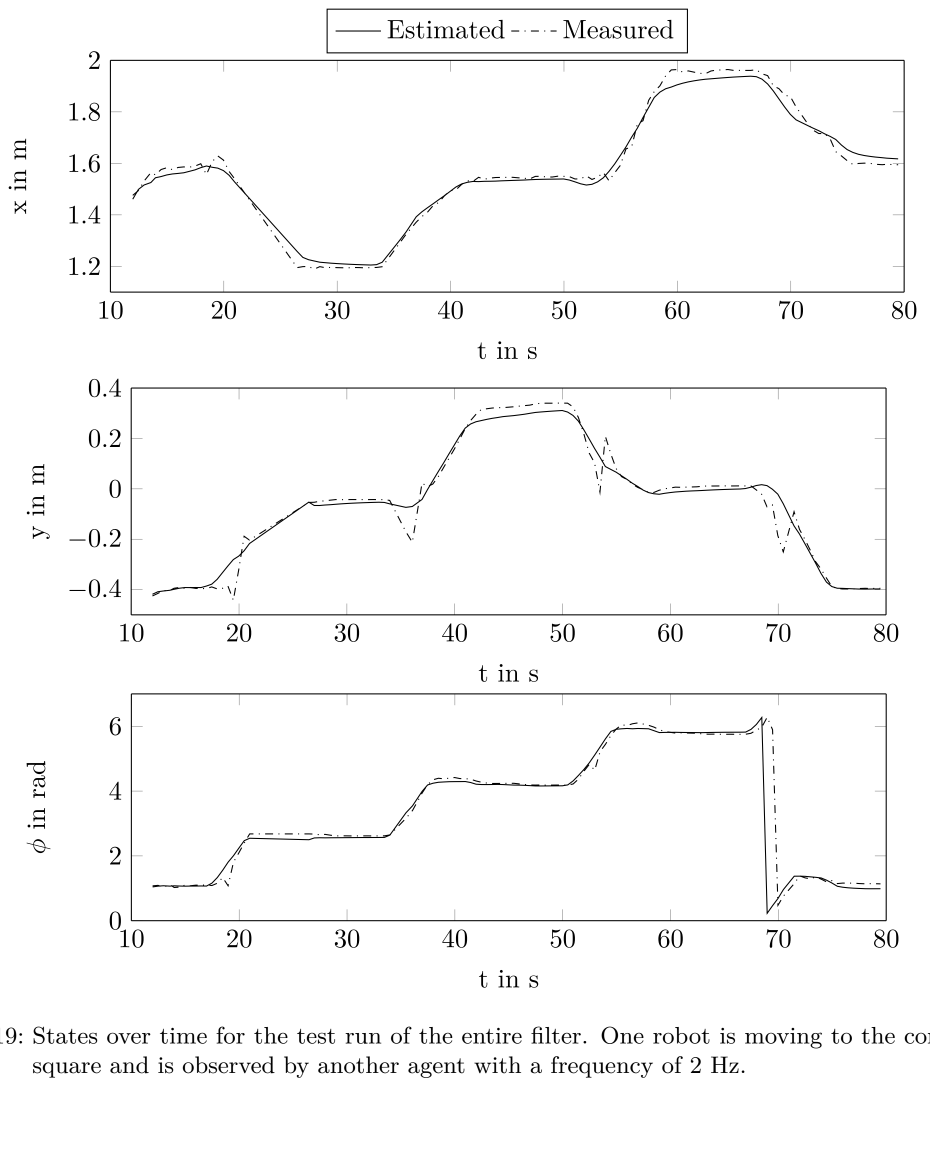


Figure 6.19: States over time for the test run of the entire filter. One robot is moving to the corners of a square and is observed by another agent with a frequency of 2 Hz.

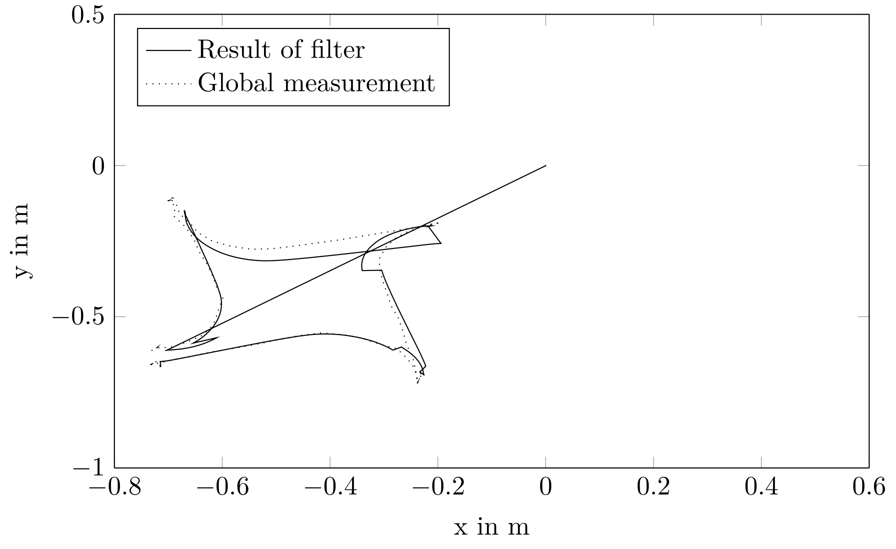


Figure 6.20: X-y-plot for the test run of the entire filter. One robot is moving to the corners of a square and is observed by another agent with a frequency of 0.2 Hz.

6.5 Update of a third passive agent

The setup is extended with a third *TurtleBot* robot (Compare illustration in Figure 6.21). All three members are line up on the y-axis of the world frame and facing in the same direction. In the first sequence *Agent A*, located at the front, receives a global update (at 6 s) and *Agent C*, the robot in the back, detects the pose of the middle one (at 12 s). After stopping this actions, a new measurement takes place from *Agent B* to *Agent A* (at 25 s). The purpose of this experiment is to investigate the updates for members that are not directly involved in a measurement event, though are connected via previous correlations.

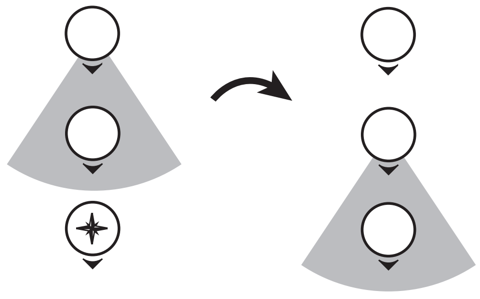


Figure 6.21: Test run to examine the update of a third passive agent.

As displayed in Figure 6.22 the orientations are hardly affected over the full-length, because every robot faces in the same direction. The position of *Agent A* is updated by the global measurement and stays constant for the rest of the time. Furthermore, the relative measurement causes the estimated states of the two involved robots to deflect symmetrically around the original position. The third measurement is not affecting the involved *Agent A*, but changes the states of the remaining members, *Agent B* and *C*. In the y-direction, for example, one can observe that both estimations shift by 1 m, while preserving a consistent relative distance. The sigmas are not much affected by the first relative measurement and only drop significantly as soon as the second relative measurement occurs. Finally, all members are virtually aligned and thus, estimated correctly.

For the first sequence, *Agent B* and *C* change their position in a relative manner, without any knowledge of their global pose. Thus, the certainty of their location is not increasing. The pose of *Agent A* is defined correctly by the absolute measurement. In the second sequence, the estimation of the robot located in the middle is corrected by interaction with *Agent A*, which acts like a landmark. At the same time, *Agent C* profits from this measurement and also improves its estimation by an update from the central processing unit. Even though, the first relative measurement is not useful at first glance, at this point it achieves a correct pose estimation of the passive robot, *Agent C*. Besides, the sigma values change and indicate that the agents have become confident about their location. The previous correlation has been stored in the Π matrix and are respected in the future calculations of the Γ matrix. Responsible for updating passive agents is in particular Equation 3.16. Hence, it is proven that the filter is as well implemented properly for applications with more than two robots.

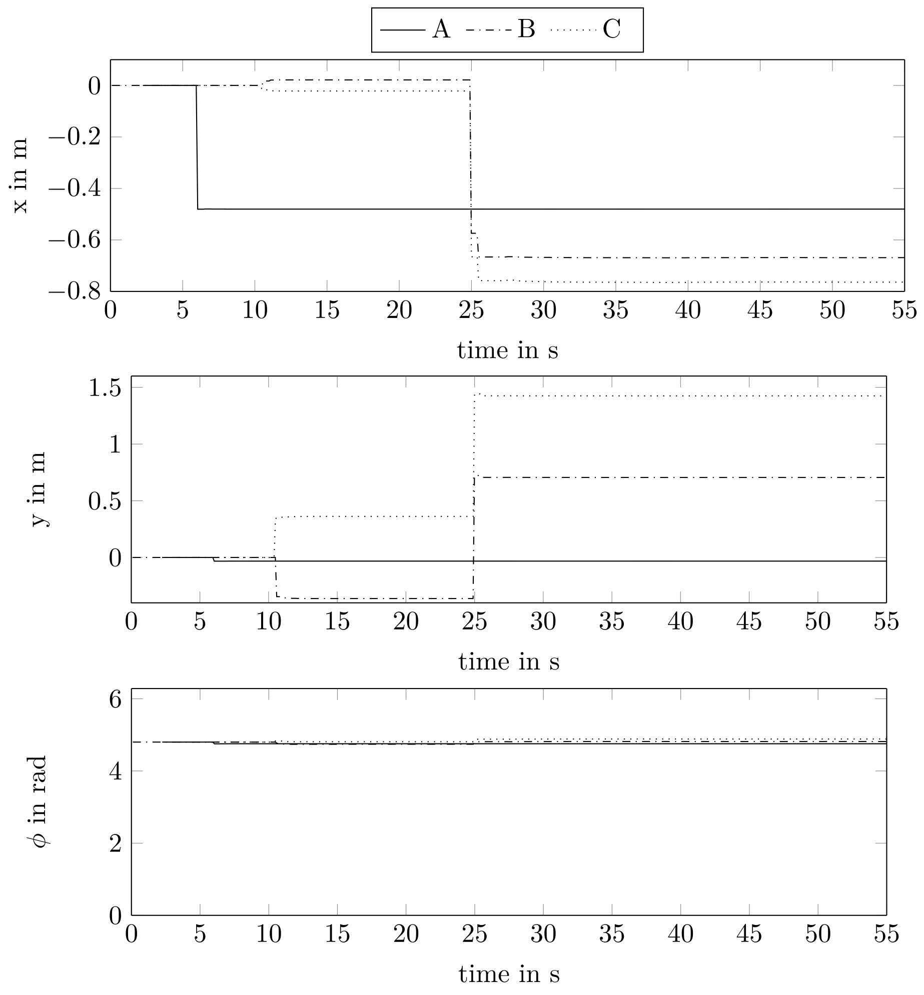


Figure 6.22: States over time for the test run of three agents. First the robot at the front (Agent A) receives a global update and the robot in the back (Agent C) detects the pose of the middle one (Agent B). Afterwards, Agent B is measuring the pose of Agent A.

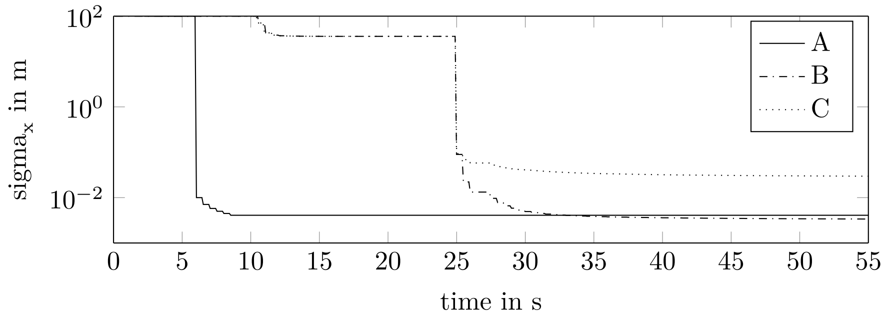


Figure 6.23: States over time for the test run of three agents. First the robot at the front (Agent A) receives a global update and the robot in the back (Agent C) detects the pose of the middle one (Agent B). Afterwards, Agent B is measuring the pose of Agent A.

6.6 Multi robot test scenarios

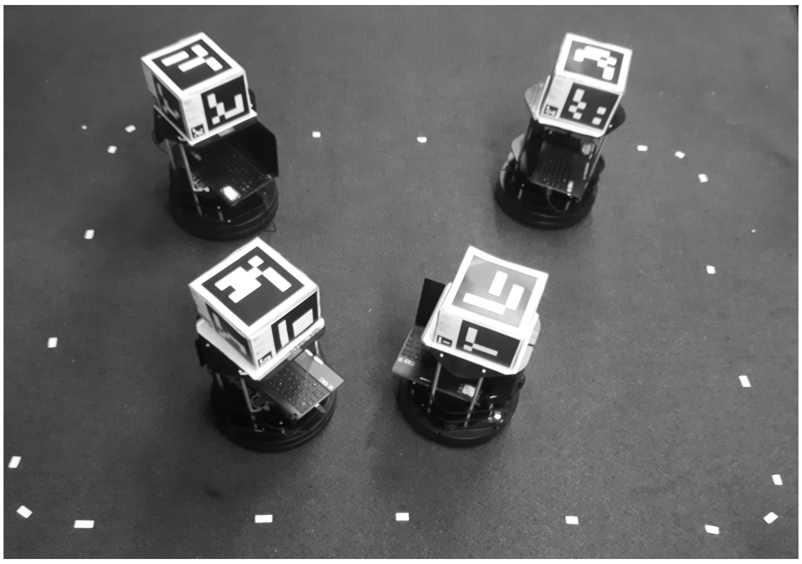


Figure 6.24: Set-up for multi robot test scenarios.

The experiments so far serve for the validation of the individual steps of the partial distributed EKF. In the following, the algorithm should be demonstrated in multi-robot scenarios, with a group of four *TurtleBot* agents. In order to evaluate the experiments, it is required to record some kind of real pose during the run. The most promising sensor is the top mounted camera. However, this results that robots can only operate in the field of view of the camera. Thus, the work space is limited to a area of 2m x 1.5m, indicated by the white marking in Figure 6.24. The movements of the *TurtleBot*s are chosen to be along a square. In the beginning, every member is positioned at one of the corners and then, movements are executed simultaneously in clockwise direction. This time the robots are moved by hand, whereby the wheels stay in contact with the ground. Thus, the wheel encoders operate in the same way and enable an equal propagation. The manual operation is required to achieve a collision-free movement of the team in the small area.

For the first test run, *Agent A* receives persistent updates from the absolute measurement. The analysis from the top mounted camera of the other agents in contrast, is only used for recording the real positions, but is not issued to the central processing unit. Therefore, their estimations rely on relative pose measurements. This actions are executed in a discrete manner in order to make the impact of the updates distinguishable. For every interaction two updates are processed.

Table 6.1: Overview of measurements for the multi robot test run with one landmark robot. I $\rightarrow$ J indicates that Agent I takes a relative measurement from Agent J. I $\rightarrow$ I indicates that Agent I receives an absolute measurement.

Corner	1	2	3	4	5
Measurements	A $\rightarrow$ A	A $\rightarrow$ A	A $\rightarrow$ A	A $\rightarrow$ A	A $\rightarrow$ A
	B $\rightarrow$ A	B $\rightarrow$ A			B $\rightarrow$ A
	C $\rightarrow$ B	C $\rightarrow$ B	C $\rightarrow$ B		C $\rightarrow$ B
	D $\rightarrow$ C	D $\rightarrow$ C	D $\rightarrow$ C		D $\rightarrow$ C

This is achieved by a manual shutter for the Kinect camera. The three robots without absolute measurements drive along the sides of the square with pure propagation and take the measurements at the corners. The relative image processing script runs with a frequency of 0.2 Hz.

The measurement sequences of the first experiment are listed in Table 6.1. Initially, every robot determines its absolute position. On the one hand, *Agent B* establish a direct connection to *Agent A*, which possesses the absolute pose information. *Agent C* and *Agent D* on the other hand, make further relative measurements and therefore, depends on the previous estimation. This sequence is repeated a second time after moving along one side of the square. After the next movement, measurements are taken again, but this time, the chain has no connection to *Agent A*. Then, all robots continue to drive the remaining part of the round and finally, repeat the initial measurements.

The result of the test run is highlighted in Figure 6.25. There are two curves per plot. The solid line represents the estimated path; and the dash-dotted the measurements of the top camera. In the beginning, the two curves start, more or less, at the same position. Consequently, one can derive that the starting position of *Agent A* is the top-right corner, the one of *Agent B* in the top-left, the one of *Agent C* in the bottom-left and the one of *Agent D* in the bottom-right.

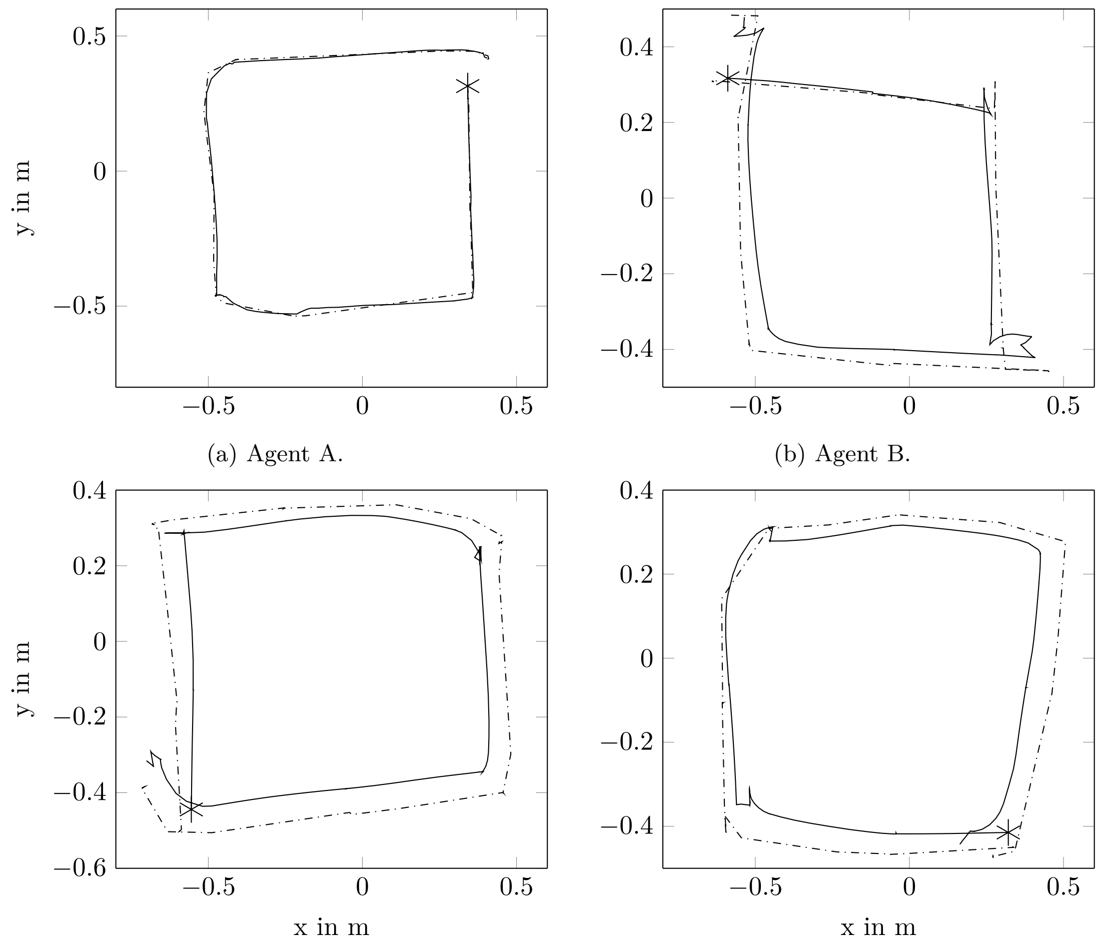
First of all, one can see that *Agent A*'s estimation follows the measured path ideally. The other estimations have some mismatch. The final measurements improve the estimation of *Agents B* significantly by the link to the landmark robot. *Agent C* and *D* is also improving its estimations, but its final error is higher. One can observe that the estimation of *Agent B* has a smaller offset to the reference compared to the following two members of the measurement chain. Figure 6.26 displays the state in x-direction of every member and in addition their 3-sigma confidence limits. The interval is increasing in the time of pure propagation and decreasing for the relative measurement events. The absolute measurements, indicated by the crosses, are lying, with some exceptions, inside this area of 99.7% trust. One should note that the absolute measurement must not perfectly reproducing the real situation. There are still some issues, which are discussed in the following paragraph.

Since *Agent A* receives high frequent absolute updates, its estimation is flawless for the whole run. Besides, it uses exactly the same sensor information as the referencing curve. The other members do not apply their absolute position measurement. Their referencing curve is from different origin and an offset is present between the two unconnected measurement systems. The error is caused by inaccuracy of measurement. Even so, every camera is calibrated, the distortion of the picture is not completely removed. Furthermore, the marker-boxes have some misalignment to the ideal center. Last but not least, errors are caused by the resolution of the *ArUco* library. In particular, the offset at the starting position can be attributed to this issues.

The agent that takes direct measurements from the landmark robot follow as well closely the real curve, but has a certain offset to the reference. The following agents will adopt this error and add up the one of the new relative measurement. Thus, the error regarding the absolute reference lane is increasing as longer the measurement chain becomes.

The relative measurement that are not connected to an absolute origin also improve the estimation. This can be for example seen in Figure 6.25 at the bottom-right position of robot *B*'s plot. *Agent C* on the other hand, is not improving evidently regarding the world frame. The relative measurements are useful to retain a similar estimation accuracy for every member. It is proper for the relative relation, but regarding the world frame, it is always limited to the most accurate estimation within the team. Thus, including an absolute measurement in the chain enhance the correction of every member significantly. This can be seen at the final update event, which push all the estimations towards the right direction.

To sum up, one can observe that the measurement updates make sure that every estimated pose stays bounded to the real values. The improvement via relative measurements is limited to the team member with the best certainty and supports in particular to retain relative formations. By adding absolute measurements, the whole chain of cooperative estimations is corrected in an absolute manner. The measurement system has also some inaccuracy, which is causing an offset to the reference line.



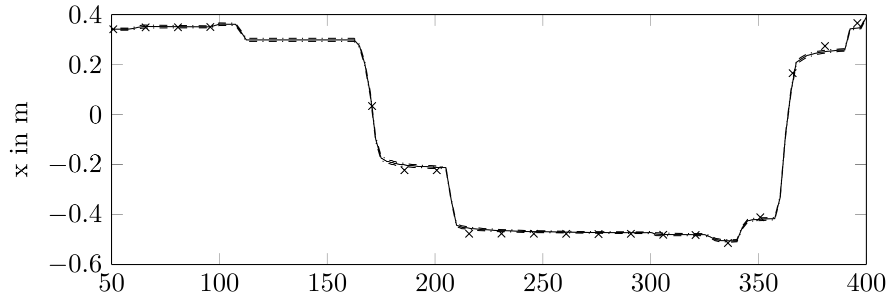
(a) Agent A.

(b) Agent B.

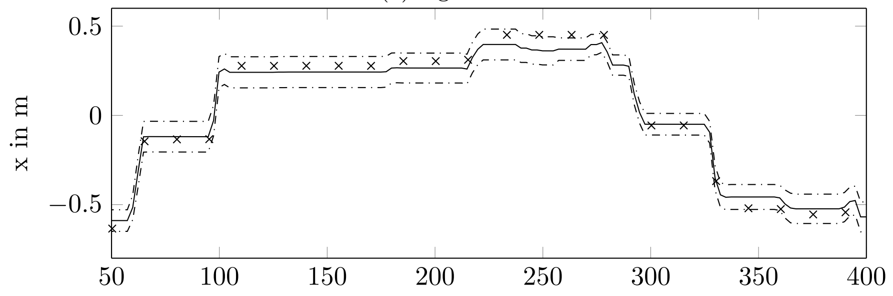
(c) Agent C.

(d) Agent D.

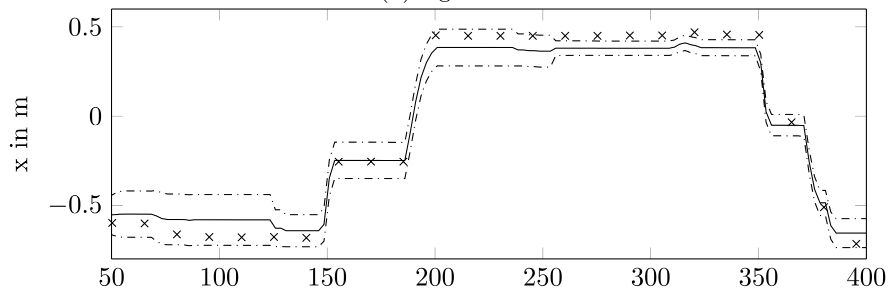
Figure 6.25: X-y-plots for the multi robot test run with one landmark robot. Four robot are moving in a square and execute measurements according to Table 6.1. The solid line represents the estimated path; the dash-dotted one the measurements of the top camera; and the star symbol the start position.



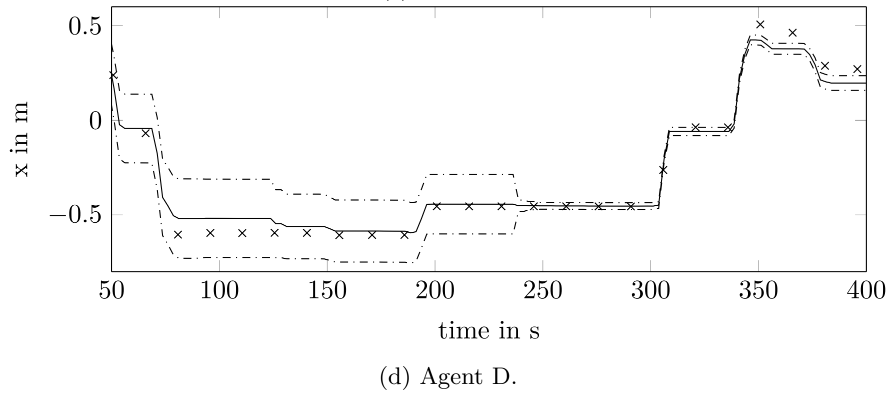
(a) Agent A.



(b) Agent B.



(c) Agent C.



(d) Agent D.

Figure 6.26: States in x-direction for the multi robot test run with one landmark robot. Four robot are moving in a square and execute measurements according to Table 6.1. The solid line represents the estimated state; the dash-dotted ones the 3-sigma interval; and the crosses the measurements of the top camera.

The next test run eliminates the landmark robot and treats every member equal. With the help of ROS parameters, it is achieved that absolute measurements are only applied in determined points of time. As Table 6.2 points out, they can be set for every member individually. Initially, a similar measurement chain as in the previous run is established. Afterwards however, *Agent A* does no longer profit of absolute information. At the next corner, there is no global update; and then *Agent B*, *C* and *D* get successive absolute pose information. The relative measurement chain remain unchained throughout all the sequences. The starting positions are comparable with the previous run.

In Figure 6.27 it can be seen that the absolute measurement updates correct the estimations effectively towards the reference frame. The whole team is influenced significantly per update, because of the established relative connections. However, the upgraded orientation is still defective. This can be seen in the x-y-plot, since the consecutive propagation depart in an angle compared to the reference line. Even so, the absolute information is applied far less as in the previous run, it is enough to bound the estimated states to the right values (compare Figure 6.28). The first update is worsen the initialized positions compared to the reference line. Again, one can see that the error increases as longer the measurements chain gets. But in a practical case there are no hard-coded starting positions available and thus, the approximation yields benefits for the localization.

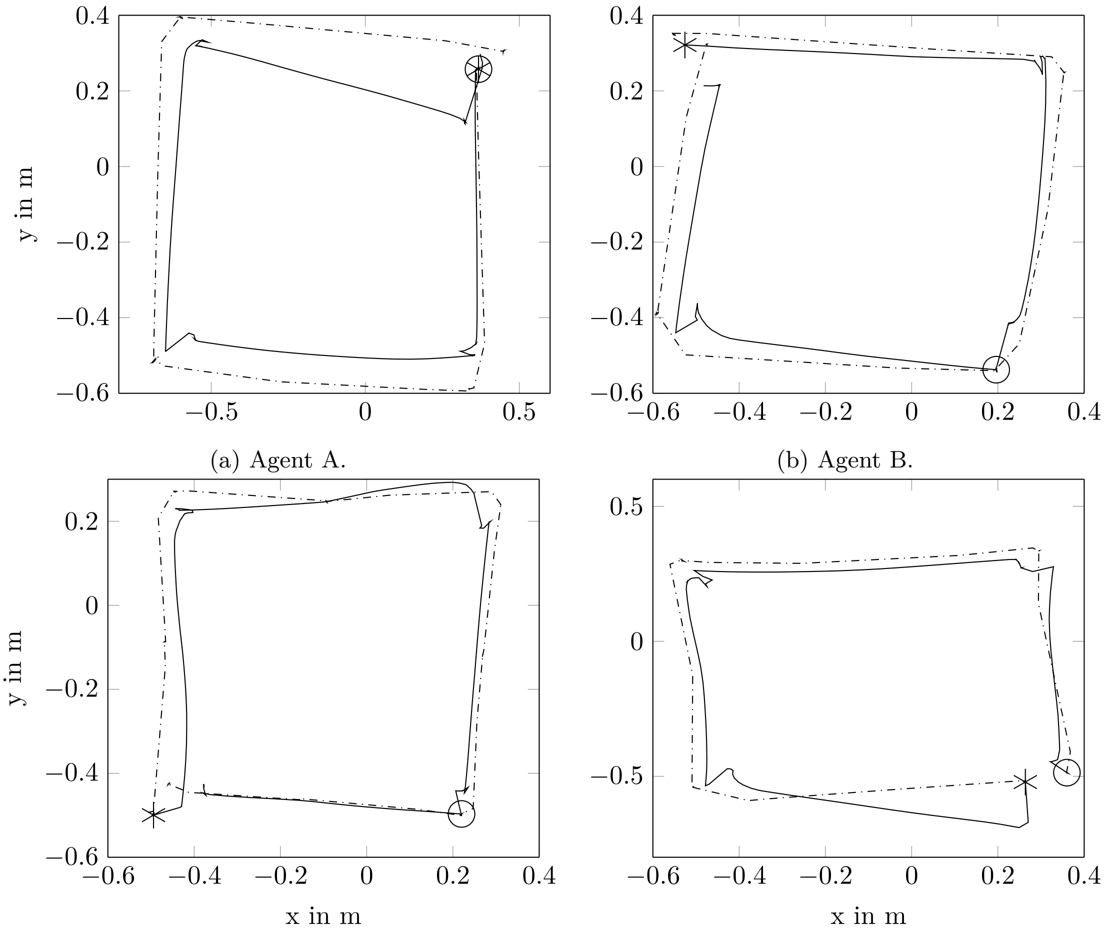
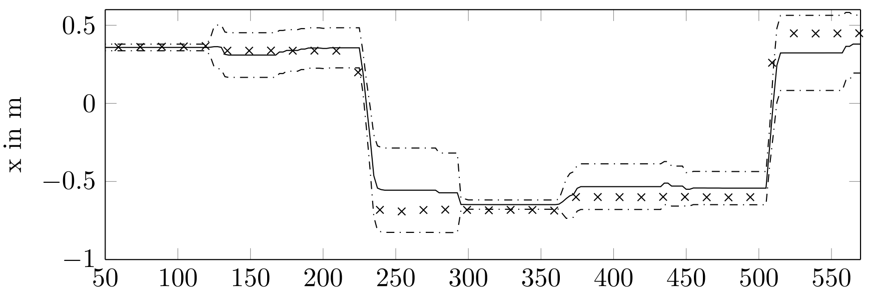
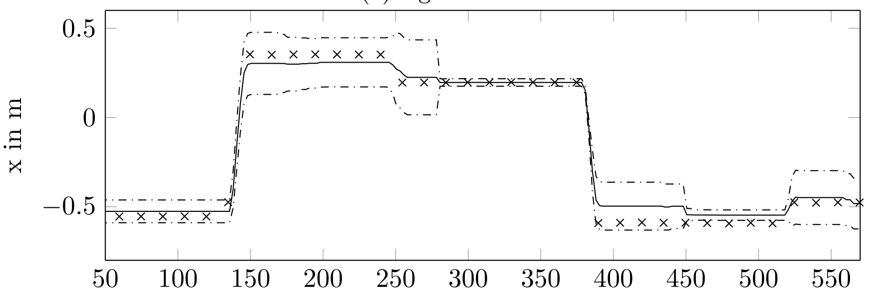


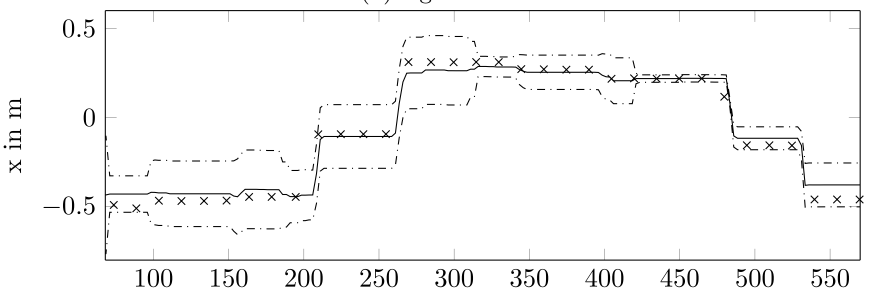
Figure 6.27: X-y-plots for the multi robot test run. Four robot are moving in a square and execute measurements according to Table 6.2. The solid line represents the estimated path; the dash-dotted one the measurements of the top camera; the star symbol the start position; and the circle the absolute measurements.



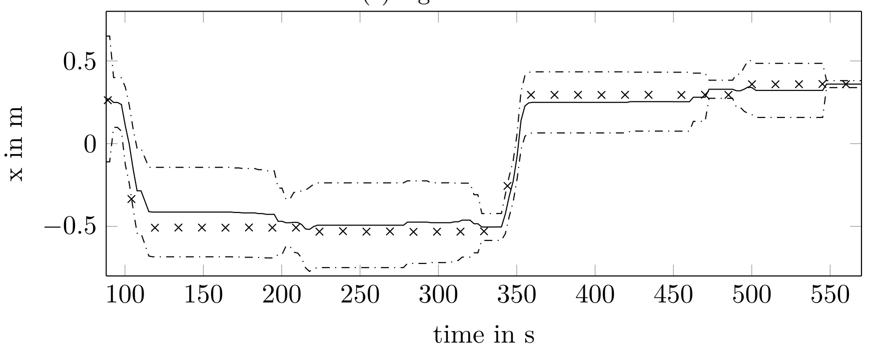
(a) Agent A.



(b) Agent B.



(c) Agent C.



(d) Agent D.

Figure 6.28: States in x-direction for the multi robot test run. Four robot are moving in a square and execute measurements according to Table 6.2. The solid line represents the estimated state; the dash-dotted ones the 3-sigma interval; and the crosses the measurements of the top camera.

Table 6.2: Overview of measurements for the multi robot test run. I → J indicates that Agent I takes a relative measurement from Agent J. I → I indicates that Agent I receives an absolute measurement.

Corner	1	2	3	4	5
Measurements	A → A		B → B	C → C	D → D
	B → A	B → A	B → A	B → A	B → A
	C → B	C → B	C → B	C → B	C → B
	D → C	D → C	D → C	D → C	D → C

6.7 Multi robot test scenario with message drop outs

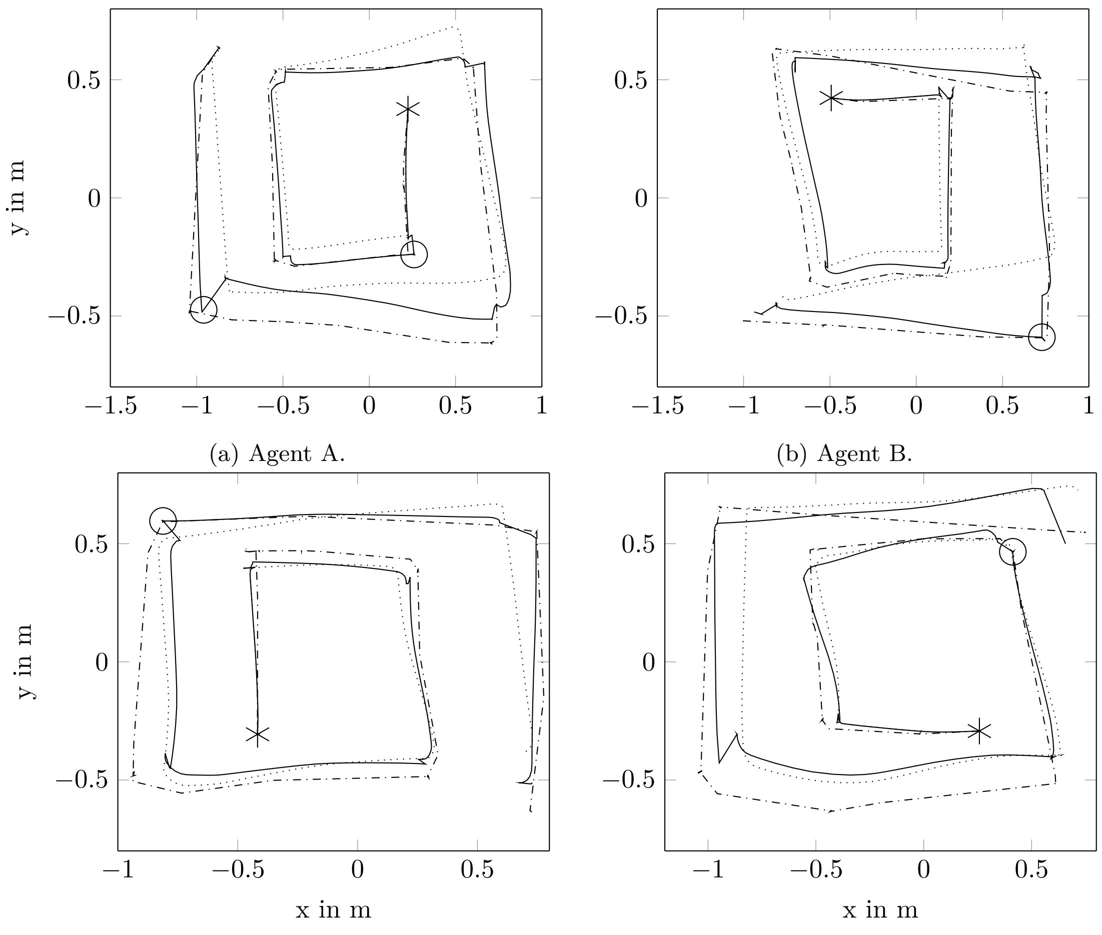
In the final scenario, message drop outs are introduced to the system. One agent is disconnected for certain measurements from the team and does not receive the according updates. This evaluation is required to proof the stability of the novel filter.

In order to include more sequences of measurements, the test run is extended with the help of a helical path. This leads to three additional corners, as one can see in Table 6.3. Every column of the table features an individual sequence. They differ in the absolute and relative measurement actions. The chain of measurements is kept as short as possible to minimize errors. Furthermore, *Agent D* is disconnected for every second event from the team. For the times, in which the agent is separated, it is not involved in any measurement. Thus, the central processing unit can calculate the updates as usual. The only consequence is that the resulting update is not applied to *Agent D*. The according cross-covariance matrix is however updated. It is worth to mention that the algorithm is intended to drop a measurement for the case that one of the involved robots is disconnected.

Figure 6.29 shows the resulting x-y-plots of all four robots. The total traveling way is longer as the previous experiments. This results in a worse behavior of the filter with pure propagation (dotted line). Here one can see the benefits of the updated version (solid line), which follows the reference (dash-dotted line) closer. The dotted line might end up well, but has great deficits during the run. The final deflection could be better demonstrated with a non-symmetric path, so that errors in opposite directions cannot compensate each other. In the following, the plot of *Agent D*, which drops some of the updates, is examined in more detail. The robot is disconnected at the two upper-left corners and the lower-right one. The plotted line shows on no correction at this points and behaves analogous to the pure propagation. The important part to look at however, is the consecutive updates. No matter whether the robot receives an absolute or relative one, the reaction is stable and results in a benefit for the estimation. The resulting improvement through this update is even stronger, since it includes the previous ones. Thus, it is proven that the algorithm can also be applied in difficult conditions, which might present communication losses. For the time that one robot is disconnected, the master accounts the respective updates in the cross-covariance matrix. After returning back to the team, the agent can fully benefit of the previous actions.

Table 6.3: Overview of measurements for the multi robot test run with dropouts. I $\rightarrow$ J indicates that Agent I takes a relative measurement from Agent J. I $\rightarrow$ I indicates that Agent I receives an absolute measurement.

Corner	1	2	3	4	5	6	7	8
Measurements		A $\rightarrow$ A		D $\rightarrow$ D		C $\rightarrow$ C	B $\rightarrow$ B	A $\rightarrow$ A
		B $\rightarrow$ A	B $\rightarrow$ A	A $\rightarrow$ D	B $\rightarrow$ A	B $\rightarrow$ A	B $\rightarrow$ A	B $\rightarrow$ A
		C $\rightarrow$ B	C $\rightarrow$ B	B $\rightarrow$ A	C $\rightarrow$ B	C $\rightarrow$ B	C $\rightarrow$ B	C $\rightarrow$ B
		A $\rightarrow$ D		D $\rightarrow$ C		D $\rightarrow$ C		A $\rightarrow$ D
Disconnected Robots			D		D		D	



(a) Agent A.

(b) Agent B.

(c) Agent C.

(d) Agent D.

Figure 6.29: X-y-plots for the multi robot test run with message dropouts. Four robot are moving in a square and execute measurements according to Table 6.3. The solid line represents the estimated path; the dash-dotted one the measurements of the top camera; the dotted one the pure propagation; the star symbol the start position; and the circle the absolute measurements.

7 Conclusion and Outlook

7.1 Conclusion

The following chapter sum ups the encountered problems during the realization and the conclusions of the evaluation.

During the realization of the implementation, several effects have been causing problems. At first, it has been intended to investigate the CL without any absolute measurements, but it turned out to be insufficient. Over time, the individual estimations would drift away in terms of position and orientation, while the relative relations would stay correct. Besides, the absolute measurement is very beneficial for the initialization. The most fatal bug has been the jump of the orientation from 0 to 2pi. The discontinuity changes all states abruptly and leads to an instability. By considering that the values are arranged in a circle this problem is solved and additionally, the time to minimize the residuals is reduced. Crucial has also been the measurement accuracy for the x- and y-position, while having a false estimation of the orientation. Compared to a *Particle Filter*, the *Kalman Filter* cannot influence its estimated position, once it has established an estimation with a high certainty. It would be trapped in a wrong belief and the only way of rectification would be to reduce the certainty by e.g. movements of the robot. Hence, one has to be very carefully to prevent any wrong updates.

The expected result has been that the novel algorithm obtains a high perception of the robots localization and by the same time reduces communication costs significantly. It can be seen straight away that the partially architecture reduces the amount of interactions. The propagation stage requires a high frequency to minimize the error, which is caused by Linearization. By outsourcing this calculations to the individual agents, one can economize the communication considerably. The perception of the filter is investigated in the evaluation. The first test run, whose estimation relies on pure propagation highlights that the error and the covariance are increasing continually while moving. Every measurement system exhibit some inaccuracy, in this case it is cause by the slip of the tire. The subsequent experiments prove that the error can be reduced by applying absolute or relative measurements. Thus, it keeps the uncertainty bounded over time, which is mandatory for every robotic application. Afterwards, more agents are involved in the test run to show that the architecture is able to process multiple measurements. Besides, one can observe that every member of the team profits of a measurement, even so they are not directly involved. Therefore, the cross-covariance term is responsible to maintain the coupling between the agents. On can say in general that the perception of every member tends to be in a similar magnitude and thus, they are denoted by the best localization estimation within the team. In the end, the stability of the algorithm against message dropouts is proven. One agent is not involved for certain updates, but the according cross-covariance is still updated. As soon as the robot is reconnected to the team, it can benefit of all the previous updates. There is one more advantage, which should not be underestimated. The disconnected agent can to some extent estimate its own pose, since the propagation stage runs locally on its processor. Thus, the individual is not entirely lost without the master, compared to a centralized system.

Returning to the example of the swarm of autonomous underwater robots, the novel algorithm yields evident benefits. Apart of the improved perception through the CL, the partly decentralized filter features considerably less communication costs and a stability against message dropouts. Consequently, it economize costs and at the same time reduces the risk of accident through a higher localization perception. Especially, in close formations, which is a crucial character of robotic swarms, it is very likely to have collision without any relative measurements.

7.2 Outlook

Regarding the implementation of the relative measurements, there is still one problematic issue with the acquired distance. The x- and the y-position of the detected robot are referenced in the frame of the observer and therefore depend on its orientation. As discussed in the chapter *Tests and Evaluation* this leads to a wrong belief for the case of a corrupt sense of direction. The solution of the thesis is to account the current orientation-variance for the update of the distance. However, the system would be modeled even better by switching from the Cartesian to a Polar coordinate system. The innovation of the relative measurement would be reduced to two states. This would solve the problem, since the resulting parameters distance and angle are independent from each other.

Until this point, the only available sensor of the MURO laboratory has been the top-mounted camera. With its help it was possible to create some faked relative measurements. However, most of the influences of a real world scenario are bypassed. Moreover, the implemented CL architecture and the relative measurement system enables the team to extend the research to outdoor environment. This is encouraging, also in the context of flying vehicles. They laboratory possesses a group of Quadrocopter, which can hardly operate within the limits of the room and not to mention within the field of view of the camera. Likewise, current projects like formation control and human-robot-interaction can profit from the enhanced perception.

The outlook is to further improve the relative measurements. The laboratory purchased in this context a set of 360-degree cameras. With this new gadgets, the CL does no longer rely on the on-board Kinect of the *TurtleBot*. In the same way, as an ordinary camera, the new one can be calibrated and subsequently, be used to retrieve relative pose measurements. It yields the advantage that the equipped robot can detect agents isotropically within a spherical area around the camera. This boosts the application of the measurement system and finally, enables more complex test scenarios.

References

[1] M. Quigley and B. Gerkey, *Programming Robots with ROS.* O'Reilly, 2015.

[2] *TurtleBot robot*, (accessed June 10, 2016). [Online]. Available: http://www.active-robots.com/turtlebot-2-complete-and-options-package

[3] S. Garrido-Juradoo, R. Muñoz-Salinas, and F. Madrid-Cuevas, *Automatic generation and detection of highly reliable fiducial markers under occlusion*, 2014.

[4] *Aruco Marker Generator*, (accessed June 1, 2016). [Online]. Available: http://terpconnect.umd.edu/~jwelsh12/enes100/markergen.html

[5] S. Thrun, W. Burgard, and D. Fox, *Probabilistic Robotics*, 2000.

[6] B. Siciliano and O. Khatib, *Handbook of Robotics.* Springer, 2008.

[7] S. Kia, S. Rounds, and S. Martínez, *Cooperative localization for mobile agents: a recursive decentralized algorithm based on Kalman filter decoupling*, 2015. [Online]. Available: http://arxiv.org/abs/1505.05908

[8] S. Kia and S. Martínez, *A partially decentralized EKF scheme for cooperative localization over unreliable communication links*, 2016.

[9] *Augmented reality library based on OpenCV*, 2016- (accessed June 1, 2016). [Online]. Available: https://sourceforge.net/projects/aruco/

[10] *Easy to use opencv camera calibration*, 2014 (accessed June 1, 2016). [Online]. Available: https://github.com/warp1337/opencv/cam/calibration

[11] *Code snippet for using the ArUco Library in ROS*, 2014 (accessed June 1, 2016). [Online]. Available: https://github.com/warp1337/ros_aruco

[12] P. G. Huang, N. Trawny, I. A. Mourikis, and R. I. S., *Observability-based Consistent EKF Estimators for Multi-robot Cooperative Localization*, 2010. [Online]. Available: http://www-users.cs.umn.edu/~stergios/papers/AR-CL-Consistency-2010.pdf

List of Figures

List of Listings

List of Tables

List of Acronyms

BSD	Berkeley Software Distribution
CL	Cooperative Localization
EKF	Extended Kalman Filter
GPS	Global Positioning System
ID	Identifier
MURO	Multi-Agent Robotics
ROS	Robot Operating System

A. Additional materials

Launch files

```xml
<?xml version="1.0"?>
<launch>
   <arg name="name" default="$(env TURTLEBOT_NAME)" />
   <arg name="x_initial"      default="0" />
   <arg name="y_initial"      default="0" />
   <arg name="yaw_initial"    default="0" />
   <arg name="covar_initial"  default="0.01" />

   <group ns="$(arg name)">
    <param name="tf_prefix" value="$(arg name)"/>

      <!-- TurtleBot Driver: minimal_bringup-->
      <include file="$(find turtlebot_bringup)/launch/minimal.launch"/>
      <param name="mobile_base/base_frame" value="$(arg name)/base_footprint"/>
      <param name="mobile_base/odom_frame" value="$(arg name)/odom"/>

      <!-- Kinect:    3dsensor_bringup-->
      <include file="$(find turtlebot_bringup)/launch/3dsensor.launch">
          <arg name="depth_registration" value="false" />
          <arg name="depth_processing" value="false" />
      </include>

      <node name="rel_meas" pkg="coop_localization" output="screen" type="rel_meas">
         <param name="RobotName" type="string" value="$(arg name)" />
      </node>

      <node name="sensor_fusion" pkg="coop_localization" type="sensor_fusion">
         <param name="RobotName"    type="string" value="$(arg name)" />
         <param name="x_initial"    type="double" value="$(arg x_initial)" />
         <param name="y_initial"    type="double" value="$(arg y_initial)" />
         <param name="yaw_initial"  type="double" value="$(arg yaw_initial)" />
         <param name="covar_initial" type="double" value="$(arg covar_initial)" />
      </node>

      <node name="cov_plot" pkg="coop_localization" type="cov_plot.py">
         <param name="RobotName"   type="string" value="$(arg name)" />
      </node>
   </group>
</launch>
```

Listing A.1: ”cl_agent.launch”

```xml
<?xml version="1.0"?>
<launch>
   <node ns="camera" pkg="uvc_camera" type="uvc_camera_node" name="uvc_camera_node">
      <param name="width" type="int" value="640" />
      <param name="height" type="int" value="480" />
      <param name="fps" type="int" value="30" />
      <param name="frame" type="string" value="webcam" />
      <param name="device" type="string" value="/dev/video0" />
      <param name="auto_focus" type="bool" value="0" />
   </node>

   <!-- Absolute Localization -->
   <node name="abs" pkg="coop_localization" type="abs_meas"/>

   <node name="central_processing" pkg="coop_localization" type="central_processing"/>
</launch>
```

Listing A.2: ”cl_master.launch”

Matlab files

```
clear,clc
X_a = [1 0 pi]';
P_a = [0.1 0 0; 0 0.1 0; 0 0 0.1];
PHI_a = [1 0 0; 0 1 0; 0 0 1];

X_b = [0 0 0]';
P_b = P_a;
PHI_b = PHI_a;
P_ab = zeros(3,3);

Z_meas = [1 0 pi]';

R = [0.1*0.1, 0,   0,
   0,  0.1*0.1, 0,
   0,  0,   15/180*pi*15/180*pi];
PI_ab = zeros(3,3);

i = 0;
% Loop
while 1
    % no Propagation
    % Update
    phi = X_a(3);
    X_delta = X_b - X_a;
    if X_delta(3)<0
      X_delta(3)=X_delta(3)+2*pi;
    end

    % S-ab
    Rot = [ cos(phi), -sin(phi), 0,
         sin(phi),  cos(phi), 0,
         0,  0,   1];
    H_tilde_a = [    1, 0, -X_delta(2),
               0, 1,  X_delta(1),
               0, 0,  1];
    H_tilde_a = Rot'*H_tilde_a;
    H_tilde_b = Rot';
    temp_S1 = H_tilde_a*P_a*H_tilde_a';
    temp_S2 = H_tilde_b*P_b*H_tilde_b';
    temp_S3 = H_tilde_a*PHI_a*PI_ab*PHI_b'*H_tilde_b';
    temp_S4 = H_tilde_b*PHI_b*PI_ab'*PHI_a'*H_tilde_a';
    S_ab = R + temp_S1 + temp_S2 - temp_S3 - temp_S4;
    S_ab = (S_ab + S_ab') *0.5; % To make sure that it is symmetric

    GAMMA_a = PI_ab*PHI_b'*H_tilde_b'   - inv(PHI_a)*P_a*H_tilde_a';
    GAMMA_b = inv(PHI_b)*P_b*H_tilde_b' - PI_ab'*PHI_a'*H_tilde_a';

    % r-a-overline
    Z_est = [cos(phi)*X_delta(1) + sin(phi)*X_delta(2),
         -sin(phi)*X_delta(1) + cos(phi)*X_delta(2),
         X_delta(3)];
    r_a = Z_meas - Z_est;
    r_a_overline = r_a;

    % For comparison
    K_a = PHI_a*GAMMA_a*inv(S_ab);
    K_b = PHI_b*GAMMA_b*inv(S_ab);

    % Update for every agent
    X_a = X_a + PHI_a*GAMMA_a*inv(S_ab)*r_a_overline;
    P_a = P_a - PHI_a*GAMMA_a*inv(S_ab)*GAMMA_a'*PHI_a';
    X_b = X_b + PHI_b*GAMMA_b*inv(S_ab)*r_a;
    P_b = P_b - PHI_b*GAMMA_b*inv(S_ab)*GAMMA_b'*PHI_b';
    update1_b = GAMMA_b*inv(S_ab)*r_a;
    % no motion -> PHI-i is constant

    PI_ab = PI_ab - GAMMA_a * inv(S_ab)* GAMMA_b';

    i=i+1;
    if (i==0)
      % ... read out variables
    end
    if (i==1)
      % ... read out variables
      break
    end
    pause(1)
end
```

Listing A.3: ”split_ekf.m”

```
clear%clc

X_a = [1 0 pi]';
P_a = [0.1 0 0; 0 0.1 0; 0 0 0.1];

X_b = [0 0 0]';
P_b = P_a;
P_ab = zeros(3,3);
Z_meas = [1 0 pi]';

R = [0.1*0.1, 0,        0,
     0,       0.1*0.1, 0,
     0,       0,        15/180*pi*15/180*pi];
PI_ab = zeros(3,3);

i=0;
% Loop
while 1
    % Update
    phi = X_a(3);
    X_delta = X_b - X_a;
    if X_delta(3)<0
        X_delta(3)=X_delta(3)+2*pi;
    end

    % S-ab
    Rot = [cos(phi), -sin(phi), 0,
           sin(phi),  cos(phi), 0,
           0,         0,        1];
    H_tilde_a = [ 1, 0, -X_delta(2),
                  0, 1,  X_delta(1),
                  0, 0,  1         ];
    H_tilde_a = Rot'*H_tilde_a;
    H_tilde_b = Rot';
    temp_S1 = H_tilde_a*P_a*H_tilde_a';
    temp_S2 = H_tilde_b*P_b*H_tilde_b';
    temp_S3 = H_tilde_a*P_ab*H_tilde_b';
    temp_S4 = H_tilde_b*P_ab'*H_tilde_a';

    %r-a
    Z_est = [cos(phi)*X_delta(1) + sin(phi)*X_delta(2),
            -sin(phi)*X_delta(1) + cos(phi)*X_delta(2),
             X_delta(3)];
    r_a = Z_meas - Z_est;
    S_ab = R+temp_S1+temp_S2-temp_S3-temp_S4;

    K_a = (P_ab*H_tilde_b'-P_a*H_tilde_a')*inv(S_ab);
    K_b = (P_b*H_tilde_b'-P_ab'*H_tilde_a')*inv(S_ab);

    X_a  = X_a  + K_a*r_a;
    P_a  = P_a  - K_a*S_ab*K_a';
    X_b  = X_b  + K_b*r_a;
    P_b  = P_b  - K_b*S_ab'*K_b';
    P_ab = P_ab - K_a*S_ab*K_b';

    i=i+1;
    if (i==0)
        % ... read out variables
    end
    if (i==1)
        % ... read out variables
        break
    end
    pause(1)
end
```

Listing A.4: ”convent_ekf.m”

Script files

```
//Jonathan Hechtbauer, UCSD, MURO, 2016

#include <ros/ros.h>
#include <image_transport/image_transport.h>
#include <cv_bridge/cv_bridge.h>
#include <sensor_msgs/image_encodings.h>
#include <opencv2/imgproc/imgproc.hpp>
#include <opencv2/highgui/highgui.hpp>
#include <opencv2/calib3d/calib3d.hpp>
#include <aruco/aruco.h>
#include <aruco/cameraparameters.h>
#include <aruco/cvdrawingutils.h>
#include <aruco/markerdetector.h>
#include <sstream>
#include <fstream>
#include <iostream>
#include <string>
#include <stdlib.h> // getenv
#include <map>
#include "coop_localization/rel.h"
#include <geometry_msgs/Twist.h>
using namespace std;
using namespace cv;

map<string, float> n,x_cum, y_cum, roll_cum, x_old, y_old;
#include "name_map.h" //List that contains the information for every marker
//map name: name_map

float xx, yy, rollroll;
float x_pxl, y_pxl;
const double pi = 3.14159265;
string home, yml_file, RobotName;
ros::Publisher rel_meas_pub;
coop_localization::rel rel;
bool new_loop=false;

// ROS
image_transport::Publisher pub;
// ArUco
aruco::MarkerDetector MDetector;
vector<aruco::Marker> Markers;
aruco::CameraParameters CamParam;
// OpenCV
namespace enc = sensor_msgs::image_encodings;
static const char WINDOW[] = "MULTIAGENT LOCALIZATION";// For the OpenCV-window.

string getEnvVar(string const& key)
{
   char const* val = getenv(key.c_str());
   return val == NULL ? string() : string(val);
}

//This function is called every time a new image is published
void image_cb(const sensor_msgs::ImageConstPtr& original_image)
{
   //Convert ROS-image-msg to a CvImage
   cv_bridge::CvImagePtr cv_ptr;
   cv_ptr = cv_bridge::toCvCopy(original_image, enc::BGR8);

   //Call to ArUco to identify markers
   MDetector.detect(cv_ptr->image, Markers,CamParam,1); //Size respected later.

   //
   //                   START LOOP for the markers
   float x_new, y_new, z_new;
   float roll,yaw,pitch;
   n.clear();
   x_cum.clear();
   y_cum.clear();
   roll_cum.clear();
   for (unsigned int i=0;i<Markers.size();i++) { // Runs through every detected marker


      int id = Markers[i].id;
      // Checks if the detected marker is defined in the name-map
      map<int,boost::tuple<string,int,string> >::iterator j=name_map.find(id);

      //    DRAW
      aruco::CvDrawingUtils::draw3dAxis(cv_ptr->image,Markers[i],CamParam);

      x_pxl = Markers[i].getCenter().x;
      y_pxl = Markers[i].getCenter().y;
      //skip if the marker is not defined or if it is close to the boarder
      if ((j != name_map.end()) && (140<=x_pxl) && (x_pxl<=500))
      {
         string   name = j->second.get<0>(), size = j->second.get<2>();
         int   side = j->second.get<1>();
         double size_box = 0.200, offset_kinect = 0.09, size_, offset;

         if (size=="s")       {size_=0.04; offset=0.075;}
         else if (size=="m")  {size_=0.14; offset=0.025;}
         else if (size=="l")  {size_=0.17; offset=0;}

         // Calculate the detected pose
         // According to https://github.com/warp1337/ros_aruco
         x_new =  Markers[i].Tvec.at<Vec3f>(0,0)[2];
         y_new = -Markers[i].Tvec.at<Vec3f>(0,0)[0];
         cv::Mat rot_mat(3,3,cv::DataType<float>::type);
         cv::Rodrigues(Markers[i].Rvec,rot_mat);
         roll = acos(rot_mat.at<float>(2,2)) - CV_PI/2;

         x_new = x_new*size_ + size_box/2*cos(roll) - offset*sin(roll) - offset_kinect;
         y_new = y_new*size_ + size_box/2*sin(roll) + offset*cos(roll);
         cout<<endl<<"ROLL without side: "<<roll*180/pi;
         roll = roll + side*pi/2;
         if (roll<0){roll=roll+2*pi;}
         cout<<endl<<"ROLL: "<<roll*180/pi;

         if ((abs(x_new-x_old[name])<0.5*abs(x_new)+0.2) && (abs(y_new-y_old[name])<0.5*
    ↪ abs(y_new)+0.2)){
         //check that the difference between two measurements is not too huge.
         //If measurement is at a new position it needs two measurement events.
            n[name]++; // Increase number of detected markers. Saved relative to robot.
            // Cumulate the measured values for every marker. Saved relative to robot.
            x_cum[name] += x_new;
            y_cum[name] += y_new;
            roll_cum[name] += roll;
            cv::putText(cv_ptr->image,name,cv::Point(x_pxl-50,y_pxl-30),2,1, cv::Scalar
    ↪ (0,255,0));
            Markers[i].draw(cv_ptr->image,cv::Scalar(0,255,0),2);
         } else{
            cv::putText(cv_ptr->image,"Bad measurement!",cv::Point(x_pxl-50,y_pxl-30)
    ↪ ,2,1, cv::Scalar(0,0,255));
            Markers[i].draw(cv_ptr->image,cv::Scalar(255,0,0),2);
         }
         x_old[name] = x_new;
         y_old[name] = y_new;
      } else{
         cv::putText(cv_ptr->image,"Close to boarder!",cv::Point(x_pxl-50,y_pxl-30),2,1,
    ↪  cv::Scalar(0,0,255));
         Markers[i].draw(cv_ptr->image,cv::Scalar(0,0,255),2);
      }

   }//                 END LOOP for the markers

   //
   //                  START LOOP for every detected robot
   typedef map<string, float>::const_iterator MapIterator;
   for (MapIterator j = n.begin(); j != n.end(); j++) {

      string name = j->first;
      xx = x_cum[name]/n[name];
      yy = y_cum[name]/n[name];
      rollroll = roll_cum[name]/n[name];
      cout<<endl<<"ROLL-"<<name<<": "<<roll*180/pi;

      //    ROS MSG
      rel.current_time = ros::Time::now();
      geometry_msgs::Twist twist_msg;
      twist_msg.linear.x = xx;
      twist_msg.linear.y = yy;
      twist_msg.angular.z = rollroll;
      rel.meas = twist_msg;
      rel.agent_a = RobotName;
      rel.agent_b = name;
      if (new_loop)
      {
         rel_meas_pub.publish(rel);
         new_loop=false;
   }} //                 END LOOP for the robots

   //Display the tracked robots
   cv::imshow(WINDOW, cv_ptr->image);
   cv::waitKey(1); //Add some delay in milliseconds.
}


int main(int argc, char **argv)
{
   ros::init(argc, argv, "rel_meas");
   ros::NodeHandle nh;
    image_transport::ImageTransport it(nh);
   rel_meas_pub = nh.advertise< coop_localization::rel>("/central_processing/rel_meas",
    ↪ 0,true);
   image_transport::Subscriber sub = it.subscribe("camera/rgb/image_mono", 1, image_cb);
    ↪      //Kinect
   ros::param::param<string>("~RobotName", RobotName, "noname");
   home = getEnvVar(string("HOME"));
   yml_file = home + string("/catkin_ws/src/ucsd_ros_project/coop_localization/config/
    ↪ cal.yml");
   CamParam.readFromXMLFile(yml_file.c_str());
   cv::namedWindow(WINDOW, CV_WINDOW_AUTOSIZE);
   MDetector.setThresholdParams(7,7);
   cv::destroyWindow(WINDOW);

   ros::Rate loop_rate(4);
   while (ros::ok())
   {
      ros::spinOnce();
      new_loop=true;
      loop_rate.sleep();
}}
```

Listing A.5: ”rel_meas.cpp”

```
/**
 * Author: Jonathan Hechtbauer
 * Modified version of Akhil Kumar Nagariya
 * */

#include "ros/ros.h"
#include "std_msgs/String.h"
#include<Eigen/Dense>
#include<string>
#include<fstream>
#include<iostream>
#include<cmath>
#include <geometry_msgs/PoseWithCovariance.h>
#include <nav_msgs/Odometry.h>
#include "coop_localization/update.h"
#include "boost/bind.hpp"
#include <tf/transform_broadcaster.h>
#include "coop_localization/request.h"

using namespace Eigen;
using namespace std;

class Ekf{
  public:
   Ekf();              //constructor call to initialize the filter
   ofstream record_file;
   bool involved_in_meas_this_update;

   void motionPropagation_dropouts();
   void sensorUpdate_dropouts();

   Matrix2d Q;
   Vector3d X, update_1;
   Matrix3d PHI, P, F, update_2;

   int members, id;
   double v,w,delta_t;
   double sigma_encoder;
   double update_frequ;

   void motionPropagation();
   void sensorUpdate();
   void publishPose();
   void motion_cb(const geometry_msgs::Twist);
   void record();
   ros::Publisher pose_pub;
   double time;
   ros::Time start_time;
};

Ekf::Ekf(){
    PHI = Matrix3d::Identity();
    Q << 1, 0,
         0, 1;
}

void Ekf::record(){
   time =ros::Time::now().toSec()-start_time.toSec();
   record_file <<time<<" "<<X(0)<<" "<<X(1)<<" "<<X(2)<<" "<<P(0,0)<<" "<<P(0,1)<<" "<<P
    ↪ (0,2)<<" "<<P(1,1)<<" "<<P(1,2)<<" "<<P(2,2)<<" "<<involved_in_meas_this_update<<
    ↪ " "<<v<<" "<<w<<"\n";
   involved_in_meas_this_update = false;
}

void Ekf::motionPropagation(){

  double phi = X(2);
  Matrix3d P_temp;

  // http://www-users.cs.umn.edu/ stergios/papers/AR-CL-Consistency-2010.pdf, page 13
  double a = 0.25;//wheel-distance
  double sigma_v = sqrt(2)/2 * sigma_encoder*v;
  double sigma_w = sqrt(2)/a * sigma_encoder*v;
  delta_t = 1/update_frequ;

  F <<  1, 0, -v*sin(phi)*delta_t,
         0, 1,  v*cos(phi)*delta_t,
         0, 0, 1;
  MatrixXd G(3,2);
  G <<  cos(phi)*delta_t, 0,
         sin(phi)*delta_t, 0,
         0,                 delta_t;
  Q <<  sigma_v*sigma_v, 0,
        0,       sigma_w*sigma_w;

  X <<  X(0) + v*delta_t*cos(phi),
        X(1) + v*delta_t*sin(phi),
        X(2) + w*delta_t;

  PHI = F*PHI;

  P_temp = F*P*F.transpose() + G*Q*G.transpose();
  P = (P_temp+P_temp.transpose())*0.5;       // To make sure it is symmetric
}

void Ekf::sensorUpdate()
{
   X = X + PHI*update_1;
   P = P - PHI*update_2*PHI.transpose();

   //round -> get rid of small errors close to zero
   X[0] = (float)((int)(X(0)*pow(10,6)))/pow(10,6);
   X[1] = (float)((int)(X(1)*pow(10,6)))/pow(10,6);
}

void Ekf::publishPose(){

    static tf::TransformBroadcaster br;
    tf::Transform transform;
    transform.setOrigin( tf::Vector3(X(0), X(1), 0));
    tf::Quaternion q;
    q.setRPY(0, 0, X(2));
    transform.setRotation(q);
    br.sendTransform(tf::StampedTransform(transform, ros::Time::now(), "map", "
    ↪ camera_link"));

    geometry_msgs::PoseWithCovariance msg;
    msg.pose.position.x = X(0);
    msg.pose.position.y = X(1);
    msg.pose.orientation.z = q[2];
    msg.pose.orientation.w = q[3];
    ros::Rate loop_rate(10);
    msg.covariance[0] = P(0,0);
    msg.covariance[1] = P(0,1);
    msg.covariance[2] = P(0,2);
    msg.covariance[3] = P(1,0);
    msg.covariance[4] = P(1,1);
    msg.covariance[5] = P(1,2);
    msg.covariance[6] = P(2,0);
    msg.covariance[7] = P(2,1);
    msg.covariance[8] = P(2,2);
    pose_pub.publish(msg);
}
```

Listing A.6: ”ekf.h”

```
//Jonathan Hechtbauer, UCSD, MURO, 2016
// Central Control Unit alias Interim Master

#include "central_processing.h"
bool meas_available = false;
bool calculating = false;
bool gathering_data = false;
bool success_[members][members];
bool state_already_received[members];
string name_a;
string name_b;
int a;
int b;
ros::Time start_time;
ofstream record_file;

// Publishes the update message to every robot
// The message of consists of GAMMA*r-a-overline and GAMMA*GAMMA-T
void publish_updates()
{
   cout<<"Updates send to:";
   for(int i=0;i<members;i++)
   {
      for(int rowrow=0;rowrow<3;rowrow++){
         update_msg.GAMMA_r[rowrow]   = update1_[i](rowrow);
         for (int colcol=0;colcol<3;colcol++){
             update_msg.GAMMA_GAMMA_T[3*rowrow+colcol] = update2_[i](rowrow,colcol);
      }}
      pub_map[robot_names[i]].publish(update_msg);
   cout<<" "<<robot_names[i];
   }
   cout<<endl<<endl;
}


// Function is called for every received relative measurement
// The incoming message includes the measurement Z and the names of the two agents
// It starts ROS services to receive X, PHI and P from the two involved agents
// Finally it saves all the data in a "Measurement" object
// The object is pushed back in a vector, since there might occur
// more measurements/callbacks per time step
void meas_event_cb(const coop_localization::rel msg)
{
   if ((calculating == false) && (gathering_data == false))
   {
      gathering_data = true;
      name_b = msg.agent_b;
      b = get_index_of_agent_name(name_b);
      name_a = msg.agent_a;
    if (name_a == "abs"){
         a = b;
         name_a = name_b;
      } else{
      a = get_index_of_agent_name(name_a);
      }
      success_[a][b] = true;

   if(Z_[a][b] == Vector3d::Zero(3))
   {
      //Call two ROS services to request information from agent-a and agent-b
      if (state_already_received[a] == false){
          if (client_map[name_a].call(srv)){
            PHI_[a](0,0) = 1;
            PHI_[a](1,1) = 1;
           for (int row=0;row<3;row++){
                X_[a](row)   = srv.response.X[row];
                PHI_[a](row,2) = srv.response.PHI[row];
                for (int col=0;col<3;col++){
                   P_[a](row,col) = srv.response.P[3*row+col];
         }}ROS_INFO("agent a");} else{ROS_ERROR("Measurement occurred, but failed to contact agent a."); cout<<name_a<<a;success_[a][b] = false;}
      }
      if (state_already_received[b] == false)
      {
         if (client_map[name_b].call(srv)){
            PHI_[b](0,0) = 1;
            PHI_[b](1,1) = 1;
            for (int row=0;row<3;row++){
               X_[b](row)   = srv.response.X[row];
               PHI_[b](row,2) = srv.response.PHI[row];
               for (int col=0;col<3;col++){
                  P_[b](row,col) = srv.response.P[3*row+col];
         }}ROS_INFO("agent b");} else{ROS_ERROR("Measurement occurred, but failed to contact agent b."); success_[a][b] = false;}
      }
    }

      if(success_[a][b] == true){
         ROS_INFO("Measurement occurred and successfully gather required information.");
         Z_[a][b] <<    msg.meas.linear.x,
                     msg.meas.linear.y,
                     msg.meas.angular.z;
         meas_available = true;
         state_already_received[a] = true;
         state_already_received[b] = true;
      }
      gathering_data = false;
}}

// First loop runs through every measurement, that are received in this time step.
// The function calculates r_a_overline and GAMMA
// with the help of x_delta, Rot, H_tilde_a, H_tilde_b, S_ab, Z_est and r_a
// The 3 Degree of Freedoms are x,y,rot-z
void calculate_updates()
{
   int N_meas = 0;
   for(int aa=0;aa<members;aa++)
   {
      for(int bb=0;bb<members;bb++)
      {
      if(Z_[aa][bb] != Vector3d::Zero(3)) //Check if measurements is available
      {
      N_meas++;
        if (aa == bb){
         //Further on the variables are choose with the name b (which in this case is equal to a)
         Z_est = X_[bb];
         // Wrap around the range of 0 and 2pi
         Z_est[2] = Z_est(2) - (2*pi) * floor( Z_est(2) / (2*pi) );
            //residual
            r_a = Z_[aa][bb] - Z_est;  //residual
            //respect that the values 0-360deg are on a circle
            if (abs(r_a[2])>pi) {   r_a[2] = r_a[2] - 2*pi*(r_a[2]/abs(r_a[2]));}
               H_tilde_b = Matrix3d::Identity();//not required
               S_ab = R_abs + H_tilde_b*P_[bb]*H_tilde_b.transpose();

               //GAMMA-Vector
               for( int UID=0;UID<members;UID++){
                  if (UID == bb){
                     GAMMA_[bb] = PHI_[bb].inverse()*P_[bb]*H_tilde_b.transpose();
                  } else{
                     GAMMA_[UID] = PI_[UID][bb]*PHI_[bb].transpose()*H_tilde_b.transpose();
        }}} else{
         double phi = X_[aa](2), c_phi = cos(phi), s_phi = sin(phi);
         X_delta = X_[bb] - X_[aa];
         if (X_delta(2)<0){
            X_delta[2]=X_delta(2)+2*pi;
         }
         Z_est <<    c_phi*X_delta(0) + s_phi*X_delta(1),
                  -s_phi*X_delta(0) + c_phi*X_delta(1),
                  X_delta(2);
         // Wrap around the range of 0 and 2pi
         Z_est[2] = Z_est(2) - (2*pi) * floor( Z_est(2) / (2*pi) );
         //residual
         r_a = Z_[aa][bb] - Z_est;
         cout<<"DEBUG r: "<<r_a[2];
         //respect that the values 0-360deg are on a circle
         if (abs(r_a[2])>pi) {
            ROS_ERROR("CHANGE DIRECTION");
            r_a[2] = r_a[2] - 2*pi*(r_a[2]/abs(r_a[2]));
         }
         Rot <<   c_phi, -s_phi, 0,
               s_phi,  c_phi, 0,
               0,    0,     1;
         H_tilde_a <<   1, 0, -X_delta(1),
                     0, 1,  X_delta(0),
                     0, 0,  1;
         if (P_[aa](2,2)<0.001) {R_dist = 0.001;}
         else{R_dist = P_[aa](2,2);}

         R_rel <<    R_dist, 0,     0,
                  0,       R_dist, 0,
                  0,    0,     0.01;
         double time =ros::Time::now().toSec()-start_time.toSec();
         record_file <<time<<" "<<aa<<" "<<bb<<" "<<Z_[aa][bb][0]<<" "<<Z_[aa][bb][1]<<"
    ↪  "<<Z_[aa][bb][2]<<" "<<Z_est[0]<<" "<<Z_est[1]<<" "<<Z_est[2]<<"\n";
         H_tilde_a = Rot.transpose()*H_tilde_a;
         H_tilde_b = Rot.transpose();
         temp_S1 = H_tilde_a*P_[aa]*H_tilde_a.transpose();
         temp_S2 = H_tilde_b*P_[bb]*H_tilde_b.transpose();
         temp_S3 = H_tilde_a*PHI_[aa]*PI_[aa][bb]*PHI_[bb].transpose()*H_tilde_b.
    ↪ transpose();
         temp_S4 = H_tilde_b*PHI_[bb]*PI_[aa][bb].transpose()*PHI_[aa].transpose()*
    ↪ H_tilde_a.transpose();
         S_ab = R_rel + temp_S1 + temp_S2 - temp_S3 - temp_S4;
         //GAMMA-Vector
         for( int UID=0;UID<members;UID++){
            if (UID == aa) {
               GAMMA_[aa] = PI_[aa][bb]*PHI_[bb].transpose()*H_tilde_b.transpose() -
    ↪ PHI_[aa].inverse()*P_[aa]*H_tilde_a.transpose();
            }
            else if (UID == bb){
               GAMMA_[bb] = PHI_[bb].inverse()*P_[bb]*H_tilde_b.transpose() - PI_[bb][aa
    ↪ ]*PHI_[aa].transpose()*H_tilde_a.transpose();
            } else{
               GAMMA_[UID] = PI_[UID][bb]*PHI_[bb].transpose()*H_tilde_b.transpose() -
    ↪ PI_[UID][aa]*PHI_[aa].transpose()*H_tilde_a.transpose();
        }}}
      double time =ros::Time::now().toSec()-start_time.toSec();
      record_file <<time<<" "<<aa<<" "<<bb<<" "<<Z_[aa][bb][0]<<" "<<Z_[aa][bb][1]<<" "
    ↪ <<Z_[aa][bb][2]<<" "<<Z_est[0]<<" "<<Z_est[1]<<" "<<Z_est[2]<<"\n";
      cout<<"Measurement from "<<robot_names[aa]<<" to "<<robot_names[bb]<<":"<<endl;
      cout<<"Z_meas: "<<Z_[aa][bb][0]<<", "<<Z_[aa][bb][1]<<", "<<Z_[aa][bb][2]<<endl;
      cout<<"Z_est: "<<Z_est[0]<<", "<<Z_est[1]<<", "<<Z_est[2]<<endl;
      cout<<"r_a : "<<r_a[0]<<", "<<r_a[1]<<", "<<r_a[2]<<endl;
      //PI
      for (int ii=0;ii<members-1;ii++){
         for (int jj=ii+1;jj<members;jj++){
            PI_[ii][jj] = PI_[ii][jj] - GAMMA_[ii] *S_ab.inverse()* GAMMA_[jj].transpose
    ↪ ();
         PI_[jj][ii] = PI_[ii][jj].transpose();
      }}
        //Update
      for(int i=0;i<members;i++)
      {
         update1_[i] = GAMMA_[i]*S_ab.inverse()*r_a;
         update2_[i] = GAMMA_[i]*S_ab.inverse()*GAMMA_[i].transpose();
         X_[i] = X_[i] + PHI_[i]*update1_[i];
         P_[i] = P_[i] - PHI_[i]*update2_[i]*PHI_[i].transpose();
      }
      cout<<"DEBUG x: "<<X_[3][2];
      publish_updates();
   }}}
   cout<<endl<<"This time step "<<N_meas<<" measurement(s) occured."<<endl;
}
```

```

int main(int argc, char **argv)
{
   ros::init(argc, argv, "interim_master");
   ros::NodeHandle nh;
   ros::Subscriber meas_event =  nh.subscribe("/central_processing/rel_meas", 100,
    ↪ meas_event_cb);
   record_file.open ("/home/jony2/measurements.txt");
   start_time =ros::Time::now();
   record_file <<start_time<<"\n";
   // Initialize
   R_abs <<  0.00001,   0, 0,
             0,   0.00001, 0,
             0,   0, 0.00001;
   PI_initial = Matrix3d::Zero();
   for (int ii=0;ii<members-1;ii++){
      for (int jj=ii+1;jj<members;jj++){
         // PI is a [4x4] matrix (if number of members = 4). Only the upper triangle is
    filled up
         // At each position it has a [3x3] sub matrix
         PI_[ii][jj] = PI_initial;
         PI_[jj][ii] = PI_[ii][jj]; //So that one can also index to the lower triangle
   }}
    ROS_init_subscriber_publisher_and_service_clients();
   ros::Rate loop_rate(2);
   while (ros::ok())
   {
      ros::spinOnce();
      if (meas_available == true)   // If any measurement occurred in this time step
      {
         calculating = true;
         calculate_updates();
         clear_Z_();
         calculating = false;
         meas_available = false;
         for (int iii=0;iii<members;iii++){
            state_already_received[iii] = false;
      }}
      loop_rate.sleep();
}}
```

Listing A.7: ”central_processing.cpp”