Parallel algorithms for target tracking on multi-core platform with mobile LEGO robots

Fredrik Wahlberg
Abstract

Parallel algorithms for target tracking on multi-core platform with mobile LEGO robots

Fredrik Wahlberg

The aim of this master thesis was to develop a versatile and reliable experimental platform of mobile robots, solving tracking problems, for education and research. Evaluation of parallel bearings-only tracking and control algorithms on a multi-core architecture has been performed. The platform was implemented as a mobile wireless sensor network using multiple mobile robots, each using a mounted camera for data acquisition. Data processing was performed on the mobile robots and on a server, which also played the role of network communication hub. A major focus was to implement this platform in a flexible manner to allow for education and future research in the fields of signal processing, wireless sensor networks and automatic control. The implemented platform was intended to act as a bridge between the ideal world of simulation and the non-ideal real world of full scale prototypes.

The implemented algorithms did estimation of the positions of the robots, estimation of a non-cooperating target’s position and regulating the positions of the robots. The tracking algorithms implemented were the Gaussian particle filter, the globally distributed particle filter and the locally distributed particle filter. The regulator tried to move the robots to give the highest possible sensor information under given constraints. The regulators implemented used model predictive control algorithms. Code for communicating with filters in external processes were implemented together with tools for data extraction and statistical analysis.

Both implementation details and evaluation of different tracking algorithms are presented. Some algorithms have been tested as examples of the platforms capabilities, among them scalability and accuracy of some particle filtering techniques. The filters performed with sufficient accuracy and showed a close to linear speedup using up to 12 processor cores. Performance of parallel particle filtering with constraints on network bandwidth was also studied, measuring breakpoints on filter communication to avoid weight starvation. Quality of the sensor readings, network latency and hardware performance are discussed. Experiments showed that the platform was a viable alternative for data acquisition in algorithm development and for benchmarking to multi-core architecture. The platform was shown to be flexible enough to be used a framework for future algorithm development and education in automatic control.
Parallel algorithms for target tracking on multi-core platform with mobile LEGO robots

Fredrik Wahlberg, jf.wahlberg@gmail.com

23rd June 2011

Abstract

The aim of this master thesis was to develop a versatile and reliable experimental platform of mobile robots, solving tracking problems, for education and research. Evaluation of parallel bearings-only tracking and control algorithms on a multi-core architecture has been performed. The platform was implemented as a mobile wireless sensor network using multiple mobile robots, each using a mounted camera for data acquisition. Data processing was performed on the mobile robots and on a server, which also played the role of network communication hub. A major focus was to implement this platform in a flexible manner to allow for education and future research in the fields of signal processing, wireless sensor networks and automatic control. The implemented platform was intended to act as a bridge between the ideal world of simulation and the non-ideal real world of full scale prototypes.

The implemented algorithms did estimation of the positions of the robots, estimation of a non-cooperating target's position and regulating the positions of the robots. The tracking algorithms implemented were the Gaussian particle filter, the globally distributed particle filter and the locally distributed particle filter. The regulator tried to move the robots to give the highest possible sensor information under given constraints. The regulators implemented used model predictive control algorithms. Code for communicating with filters in external processes were implemented together with tools for data extraction and statistical analysis.

Both implementation details and evaluation of different tracking algorithms are presented. Some algorithms have been tested as examples of the platforms capabilities, among them scalability and accuracy of some particle filtering techniques. The filters performed with sufficient accuracy and showed a close to linear speedup using up to 12 processor cores. Performance of parallel particle filtering with constraints on network bandwidth was also studied, measuring breakpoints on filter communication to avoid weight starvation. Quality of the sensor readings, network latency and hardware performance are discussed. Experiments showed that the platform was a viable alternative for data acquisition in algorithm development and for benchmarking to multi-core architecture. The platform was shown to be flexible enough to be used a framework for future algorithm development and education in automatic control.

Keywords: Control Education; LEGO; Benchmarking; Real Data; Bearing-only Target Localisation; Model Predictive Control; Particle Filtering; Sensor Network; Parallelisation; Mobile Robots; Multi-core
## Contents

### I Introduction

1 Populärvetenskaplig introduktion (popular introduction in Swedish)  

2 Specifications and background
   2.1 Project description  
   2.2 Mobile robots  
   2.3 Bearing-only tracking problem  
   2.4 Context

### II Implementation

3 Overview of the implemented system
   3.1 Code conventions

4 Hardware design
   4.1 Sensor node design  
   4.2 Server design

5 Software design
   5.1 Overview  
   5.2 Operating system - leJOS  
   5.3 Graphical user interface  
   5.4 Network and communication  
   5.5 External filters

6 Simulation framework

7 Arena

8 Interfaces for implementing filters and regulators
   8.1 Preprocessing filter - filter on brick  
   8.2 Filtering phase 1 - estimation of sensor node positions  
   8.3 Filtering phase 2 - estimation of target position  
   8.4 Regulator

### III Implemented filters and regulators

9 Particle filtering
   9.1 Particle filter theory  
   9.2 Implemented filters

10 Regulators
   10.1 Model Predictive Control  
   10.2 Heuristic regulator  
   10.3 Formation regulator  
   10.4 Search tree regulator  
   10.5 Implementation

### IV Experiments and results

11 General performance
   11.1 Sensor accuracy  
   11.2 Moving target  
   11.3 Test scenarios  
   11.4 Tools for data analysis
12 Phase 1 filter performance
12.1 Test of drift ................................................................. 29
12.2 Test of correction over time .............................................. 30

13 Phase 2 filter performance
13.1 Performance measures and data sets ................................. 30
13.2 Accuracy test ............................................................... 31
13.3 Scalability test ............................................................. 31

14 Discussion
14.1 Conclusions ............................................................... 35
14.2 Problems during implementation ...................................... 35
14.3 Filtering and real data .................................................... 36
14.4 Future work ............................................................... 36

References ................................. 37
Figur 1: Överblick över arenan. Det eftersökta målet är roboten i mitten, som åker på en bana. Målet är omgivet av tre robotar, s.k. sensornoder, som utför sökandet. I övre vänstra hörnet står ett s.k. landmärke som används för att uppskatta sensormodernas positioner i arenan.

Part I
Introduction

1 Populärvetenskaplig introduktion (popular introduction in Swedish)


Delmål för funktionaliteten är följande:

• Robotarnas positioner ska kunna uppskattas relativt ett antal fasta landmärken.
• Positionen för ett icke samarbete mål ska kunna uppskattas.
• Robotarna ska kunna förflytta sig för att öka noggrannheten i mätningarna.

I denna rapport kommer jag beskriva detaljer kring det implementerade systemet och ge exempel på hur plattformen kan användas.


Problemet med att hitta en uppskattning av var målet befinner sig löses här med s.k. partikelfilter. Partikelfilter kan användas för att välja samman information från många typer av sensorer och klarar av att ta hjälp av mycket komplicerade modeller över det system vars tillstånd man försöker uppskatta. Nackdelen är att de är beräkningskraftiga. De partikelfilter som används här utnyttjar flera processorkärnor för att råda bot på problemet med brist på beräkningskraft.

Robotarna har till uppgift att ge realistisk rå-data till partikelfiltren. Detta ska ses i perspektivet att simulering med data genererad med en dator ofta blir väldigt tillrättalagd. Å andra sidan är en fysisk prototyp av ett system väldigt dyr. En medelväg är att skapa en enkel prototyp i t.ex. LEGO som kan generera realistisk data. Med mer realistisk data kan prövandet hos algoritmerna undersökas närmare och med högre relevans i resultaten än med helt simulierad data.

Arbetsgången under köring för den implementerade plattformen kan sammanfattas i följande steg:

1. Varje robot sveper med sin kamera och söker efter landmärken och mål.
2. Den information som varje robot samlar in skickas via ett trådlöst nätverk till en central dator.

3. Den centrala datorn använder informationen från varje robot, tillsammans med tidigare kunskap om robotarnas positioner, för att göra en ny uppskattning av robotens position.

4. När uppskattningen av robotarnas positioner är tillräckligt bra ligger de till grund för en uppskattning av målets position som görs av den centrala datorn.

5. När både robotarnas och målets position är kända med tillräckligt hög noggrannhet kan robotarna förbättra noggrannheten i uppmätta data.

Examensarbetet utförs som en del i det värvetenskapliga forskningsprojektet "Computationally Demanding Real-Time Applications on Multi-core Platforms" som undersöker datorsystem med möjlighet att köra realtidsprogram parallellt. Fokus ligger på att hitta sätt att utnyttja flerkärningsprocessorer effektivt för signalbehandling och reglerteknik.

2 Specifications and background

2.1 Project description

The goal was to implement a flexible framework for implementing and testing different tracking and regulator algorithms applicable to a wireless sensor network of mobile robots equipped with bearings-only sensors (i.e. sensor that only gives a direction to a specified target). The robots performed data acquisition and some data processing. Most sensor data was sent upstream via a bluetooth network to a central multi-core computer doing the main parts of the data processing. The following tasks were to be solved in real time:

- The position estimation of individual robots with respect to a set of stationary navigation marks.
- The position estimation of a non-cooperating target.
- The evaluation and command of changes in the positions of mobile sensor stations in order to improve the target position estimate.

To achieve these goals, each robot collected information on the target and navigation marks (called landmarks for here on) and sent it upstream to the server. The server filtered the data to determine each robot's position. Having positioned each robot the server could determine the position of the non-cooperating target. Tracking the target over time gave the possibility to estimate its velocity. When the position and velocity of the target were determined, its probable future positions could be estimated. Using the estimated future positions of the target the robots was sent movement commands to potentially increase the quality of future sensor readings.

The server was responsible for the bulk of the calculations and data processing, but some were also performed on the sensor nodes. The platform was implemented to give room for different approaches to the tracking and control loop, either distributed or centralized. Visualization, storage and data analysis was also done on the server.

The set-up consist of the following (image in figure 1):

- Three mobile robots, called the sensor nodes, collecting data using a camera.
- One non-cooperating target sought for by the sensor nodes. This target was also build as a LEGO robot and is described in section 11.2.
- Four landmarks, one placed at each corner of the square arena.
- One server where the estimation and control algorithms were run.

2.2 Mobile robots

The theme of this thesis work is multiple mobile robots solving tracking and regulator problems in a limited and controlled space. This kind of set-up can be seen in many other articles, books, competitions and commercial applications[1, 2, 3]. There are of course many military applications of robotics and tracking spanning from medic support[4] during war time to UAVs tracking vehicles[5]. Bearings-only tracking is also important in computer vision and used in surveillance, human computer interaction and lately also games. There is some work on making generic robotics platforms[6] but since there is a limitation in hardware, re-using this work was not realistic.

The field of research concerning sensor networks is wide. Problems considered here only concern tracking and control with the exception of unavoidable dependent problems e.g. synchronisation. Many interesting and important problems like network topology, energy consumption etc. fell outside the scope.
Figure 2: The bearings-only problem. In both sub-figures the curved arrows show the angles measured from a reference angle. Straight dashed arrow show the direction of the bearing. **Left:** Estimation of the positioning of a sensor node. The node (stick figure) sees four landmarks and can from their bearings estimate it’s position and heading. **Right:** Positioning of the non-cooperating target (common problem solved by all nodes together). Three nodes have bearings to a target from which it’s position and, over time, velocity can be estimated.

The mobile robots here were built using the LEGO Mindstorms robotics kit and a low resolution camera designed for Mindstorms, called the NXTCam. The camera gives the robot vision capabilities. One motor were used for turning the camera and two more propel the robot. The reason for using a LEGO Mindstorms set-up was mainly that it was relatively cheap. Robotics set-ups can be quite expensive but the LEGO kit is intended as a toy. Combined with the open source firmware leJOS[7], a Java virtual machine directly on top of the hardware (section 5.2), it can be used as a robotics platform suitable for, but not limited to, bearings-only problems. However, the hardware chosen was very limited in ways such as sensor resolution, computational power and network bandwidth.

The main advantage of using a robotics platform for researching tracking algorithms instead of simulation is to let the algorithms meet reality. In simulation, many design decisions must be made to give a realistic environment for the algorithm. If one of these assumptions is wrong, the simulation might give results impossible to match in reality. Using real data is crucial to prove the effectiveness of an algorithm. However, implementing a full prototype is expensive and time-consuming. This work tries to mitigate these contradictions.

### 2.3 Bearing-only tracking problem

What defines a bearings-only problem is the type of information given by the used sensors. The only data given is a direction to a tracked object relative to the position of a sensor and a reference angle. This is called a bearing, as in navigation. In this case, it meant that the sensor nodes did a sweep with their cameras to find the navigation marks and the target, trying to find the bearings to the observed objects. The bearing data was given in a local reference system relative to each robot. A relation between the different sensors local reference systems and the global reference system was sought for to be able to get the bearing of the non-cooperating target needed in the common tracking problem. In figure 2, this process is illustrated. Each sensor node sees four navigational marks with known positions. From this information, its own position can be known to an accuracy constrained by the sensor quality. All three sensor nodes cooperate to find the position of the target. Tracking the target over time opens for the opportunity to estimate the target’s velocity. The certainty of the tracking of the target is also subject to the accuracy of the sensors.

There is a lower bound on how well the position of a target can be estimated. If the sensors are noisy the estimation will be more uncertain. The estimation certainty is bounded by the Cramer-Rao bound. This is a result from information theory that gives a bound on the variance of an estimate depending on the sensor accuracy.

A common approach to solving the problem of finding the positions of mobile objects with bearings-only sensors are particle filters[8, 9, 10]. It is a method for recursively determine the states of the tracked object that is suitable for non-linear tracking (described in more detail in section 9). An important advantage of particle filters is their ability for handling sensor fusion and non-linearity. The standard way of solving non-linear estimation problems is to use the extended Kalman filter (EKF). The EKF depends on a linearization of the problem, where the particle filter does not. The particle filter has been shown to be more accurate and not suffer from instabilities caused by linearization[11].

Some of the tracking done here was based on the work of Olof Rosén[12]. He has evaluated different particle filter algorithms suitable for parallel execution as part of his master thesis, and tested them in Matlab simulation. The parallel C++ implementation of the particle filters (section 5.5) used to estimate the non-cooperating target have been developed together with him. Filtering performed to estimate each sensor node’s position is implemented in Java using standard techniques modified here to suit the specific problem and hardware.
2.4 Context

The topic of this master thesis is related to the multi-disciplinary research project "Computationally Demanding Real-Time Applications on Multi-core Platforms"[13] funded by Swedish Foundation for Strategic Research. It is being carried out at the department of Information Technology at Uppsala University and deals with real-time applications on multi-core platforms. It aims to analyse the parallelization of both control and signal processing algorithms on one hand and the usage of computer resources like cache and inter-core communication on the other.

A strong argument for using multi-core platforms is the energy efficiency. This is important both for computers connected to electricity grid and those running on battery, but definitely more important in the case of the latter. In many embedded systems, the computing power needs are great and the energy consumption limits strict. Multi-core technology is a way of reconciling these constraints. While energy consumption scales roughly quadratic with CPU clock frequency, it only scales linearly with the number of cores. This of course assumes that the multi-core architecture can be efficiently used. This is not a trivial task and an algorithm might even perform worse on multi-core than on a single-core machine if not correctly implemented. The implemented robot platform in this thesis is able to test these kinds of parallel algorithms in a more realistic environment than when using only simulation. Support for parallelism using both a single computer and many different nodes was implemented and used.

Wireless sensor networks consist of many spatially separate embedded computer systems, called nodes, equipped with sensors of some kind and communicating through wireless links. They sense e.g. temperature or radiation and are usually used for monitoring large industrial structures. Using wireless communication with a non-hierarchical network topology a sensor network can communicate information on the environment over large distances at a low energy cost. Equipping the nodes with motors makes it possible for the network to reposition itself and monitor very dynamical environment such as burning buildings. The sensor networks can have a central controlling node or be completely distributed (i.e. not dependent on any specific node). The work done here is connected to this field by using multiple sensor equipped mobile nodes for estimation problems.

The code for the implemented system was based on the C.A.T.S.[14] (Cooperative Autonomous Tracking System) project for which the author was the project leader. It solves similar problems and were built as a part of a project course in Embedded systems[15] at Uppsala University. The project aimed at solving a tracking bearings-only problem cooperatively between three LEGO robots but with much less functionality than the work presented here. The inherited code base has here been reused, where possible, and greatly extended. Considerable time was spent on integration, documentation and re-factoring the code base.

Part II
Implementation

3 Overview of the implemented system

The implemented system consisted of two types of subsystems connected via a bluetooth piconet. The server side subsystem did data processing, data representation, regulating and user interface. The sensor side consisted of many mobile sensor nodes, each with its own instance of the sensor subsystem. It also did some data processing but most importantly sensor control and movement.

A view of the system as a control loop is shown in figure 3. The following steps are done in real time (numbers below correspond to numbers in the figure):

1. The sensors collected data on their environment and motor status. This gave a bearing to a tracked object, angles turned by the robot and distance travelled. (section 4.1.2 and 4.1.3)
2. The preproccessing filter kept a local position estimate and prepared sensor data for upload to the server. (section 8.1)
3. The server received the data and passed it to the phase 1 filter. There was one instance of this filter for each sensor node. These filters tried to position each node relative to the known landmarks finding a relation between the bearings in the local and global reference systems. (section 4.1.2 and 8.2)
4. The phase 2 filter used information from all sensor nodes to find the tracked object’s position and velocity. (section 8.3)
5. The regulator issued commands to the sensor nodes on how to move or search to get more information on the landmarks and tracked target. (section 8.4)
Figure 3: An overview of the implemented system illustrated as a control loop

All steps will be discussed in more detail below. The connection between the motors and sensors in figure 3 is dashed since it does not signify an information flow but rather changes the possibility of sensor information gain. Every block in the control loop was run separately, virtually connected via buffers. Some of the buffers were connected via Bluetooth and all data were sorted by time. The network bandwidth, network latency, the error in the distributed clock and thread periods created delays in the data relaying and processing. These delays could at maximum become the period time of each block added to the maximum network latency. This theoretical maximum loop time sets constraints on system performance. Mostly the system gave a loop time of under 600 ms.

All filters and the regulator components could easily be replaced or rewritten to allow for algorithms not implemented here. The interfaces that in that case needs to be implemented are described below.

3.1 Code conventions

The project includes approximately 16500 lines of Java code and 800 lines of Matlab code. Making the code base readable was an important goal. Usually this entails hours for writing documentation. Still the interfaces often evolve and bugs may be fixed in a way changing functionality. Documentation separate from the code is never a good idea and even Javadoc can get quite out of date if the specifications change over time. Using a good code convention[16] and self documenting code was probably the best of the easy ways to keep the code in good shape and was hence used.

Some of the code was written by others than the author and was not included in the work on code quality. However, all code was object oriented as required by Java so all functionality was wrapped in a standard and easy way. Most of the code has been re-factored to some extent but there is still work needed to make the code as good as the goals set.

To make extending the functionality simple, all components included in the control loop have a standard structure. All the communication with the surrounding system was inherited from base classes, one for each component. When implementing any new components, one does not need detailed knowledge of the underlying system. Everything from buffering to threading is already taken care of. Most information is however public inside the respective subsystems to make reading of it possible.

4 Hardware design

4.1 Sensor node design

Every sensor node was build using a LEGO Mindstorm[17] robot kit and a Mindsensors[18] NXTCam camera.

4.1.1 NXT Hardware overview

The Mindstorm kit consisted of (among other parts) one computer block with Bluetooth chip and sensor and actuator sockets. The CPU was clocked at 48MHz[19], which is not much by today's standards but was enough for running the system. Sadly it was discovered that the Bluetooth stack was somewhat buggy and that the transmission speeds was at maximum between 3-7 Kb/s to each node (compared to 90 Kb/s to 7 nodes and up to 255 nodes in total counting the passive ones, described in the Bluetooth standard[20, 21]). The top transmission speeds could only be achieved when sending large packets without delivery confirmation. Network latency was also considerable.
Three motors with tachometers were capable of giving a fairly accurate reading of the motors absolute rotational position relative to the starting point, but not speed. The tachometers were accurate to one degree. The angular momentum of the motor was considerable and a regulator was needed to get accuracy in rotational speed and position.

The standard sensors delivered with a Mindstorm kit were not suited for bearing-only tracking and could not be used. In the end the best choice of sensor, considering data collection speed and field of view, was a camera. The standard operating system did not really allow for programming anything advanced and had to be replaced (section 5.2).

### 4.1.2 NXTCam

The acquired camera was an NXTCam, a low resolution “web” camera[22, 18] attached to a micro controller. It communicated with the NXT computer via an i2c bus. This bus was too slow to transfer a whole image, and the NXT computer not really suited for image analysis, hence all processing of the image data was done on the micro-controller. The firmware on the micro-controller tried to find regions of predefined colours and returned their positions along with a value symbolising what colour the region had. Since this parametrisation of the important information was small, the measurements per second could be high in spite of the limited hardware. Camera configuration needed to be done on a PC where the colours were defined and the new firmware compiled.

The horizontal resolution of the camera was 176px and the field of view 42 degrees. This gave a best case accuracy of $\frac{42}{176} \approx 0.24$ degrees. The coloured regions in the image could vary a lot around the edges depending on light and noise in the image acquisition. Determining the centre of a region was done one frame at a time and the centre point could vary somewhat. The accuracy of 0.24 degrees must therefore be seen as theoretical lower limit.

Calibration of the cameras needed to be performed with respect to the horizontal bias. The cameras did not acquire the image straight in front of them which led to a measurement error of many degrees. Calibration was done by shifting the image centre to account for this error. Geometric aberrations in the image because of the lens were not accounted for but have shown to be small enough. Only the non-cooperative target was searched for in a way that puts it in the centre of the image. The errors from geometric aberrations would mostly occur in the sighting of the landmarks since they were mostly spotted at the sides of the image while looking for the non-cooperating target. Including this error in the model of the filters was shown to compensate for it.

The angle of the camera was given by the tachometer in a motor that the camera was mounted on. It was important that the camera faced straight forward at start-up since the tachometers were reset to zero then. The motor was also fitted with gears to increase the accuracy of the tachometer data. The resolution of the tachometers were one degree and this was geared down to 0.2 degrees. The theoretical largest bias was then 0.44 degrees. The turning of the motor was controlled by a PD-controller with deviation in pixels from the target to the calibrated centre as the error measure. When a target was seen, the information was committed to a buffer, which was read from by the preprocessing filter.

### 4.1.3 Movement

The base of each robot was the central computer block. Driving was done by two motors controlling one wheel each, positioned on both sides of the robot and slightly in front of the centre of mass. An easy sliding part
lifted the rear of the robot. The motor placement gave the robot the capability of simple movement like tuning or driving straight. The implemented motor controller could be given coordinates of where it was and where to go. From this data, it calculated how much to turn and then drove forward or backward. The controller took acceleration into account to minimize sliding. The tachometers in the motors and the knowledge of wheel diameter gave information on distance travelled and degrees turned. Information on movement was committed to the buffer connected to the preprocessing filter on each sensor node. Move advanced movement was hard to implement since the constraints were too tight concerning network bandwidth, assuming the regulator was implemented on the server side. The round trip time could be so large that the sensor node would have to be moving really slow for the server to keep up. It was decided that each movement command should have a definite end, ensuring that the robot nodes did not go too far in case of a network congestion.

The drift calibration was done by letting the nodes turn 90 degrees and drive 3 meters, measuring the error in the final position. Some calibration constants were added to the movement regulator to account for the errors. Turning was seldom more than a couple of degrees wrong, but still enough for there being a need to correct it by using the phase 1 filter. Driving was very accurate and on a straight track of 3 meters the node was seldom more than 10 cm off. The calibration however depended on that the parts of the robots did not move during a running session. If a motor would be just slightly loose from the robot frame, the drift could increase considerably. During a run with the robots, this problem would occur and the only way to fix it was to stop the experiment to press the robots back together again.

4.2 Server design

The server was a HP Z800 workstation with two processors having 6 cores each and about 24 gigabytes of memory. The operating system was set up so that while benchmarking no other processes should be running above a negligible CPU consumption level. Giving all computing power to the parallel estimation problem. The server was not delivered with an internal bluetooth chip so an external antenna was bought. Using a level 1 bluetooth antenna[21] the theoretical range would be up to 100 meters, which was enough. Setting up the leJOS communication scripts took some time but was worth it since uploading the firmware to the sensor nodes in parallel could be achieved, which saved a lot of time compared to uploading it via USB.

5 Software design

5.1 Overview

Figure 5 shows an overview of the main components of the system. The sensor side information was collected through work of the camera and motors. The preprocessing filter sends information upstream. A global clock tags all relevant transmissions with a time stamp, which follows a piece of data through the system all the way to the phase 2 filter. The global clock was implemented like a component living on all machines involved.

The server side had a graphical user interface as a front end for the whole system. There manual orders could be given and data visualized. The filter data handler dispatched data to the relevant filters and built a configurable pipeline of data processing and regulating of the connected sensor nodes. The phase 2 filter component could also be connected with pipes to an external process, which gave considerable freedom in the implementation of the filtering.

5.2 Operating system - leJOS

The operating system on the NXT was changed, from the one pre-installed by LEGO, to leJOS[7]. It was a Java virtual machine implemented directly on top of the hardware. leJOS was in beta stage but still very stable. Most of the applicable Java libraries were already implemented and it could handle an almost full set of core functions. During the implementation, the limitations of the Java motor were not a large problem despite some functionality not being implemented in a full Java engine and some small memory management bugs. Instead of the lacking support for dynamic data structures the necessary containers and buffers were written during implementation, with the advantage of being customized to the current needs.

The leJOS interfaces for the sensors and actuators behave like Java objects usually do. When using a motor for instance, a controller class was instantiated and assigned a port on which the motor was connected. Then, methods in the controller object were easily called setting speed and the direction in which the motor should be running. Information could be easily read the same way.

Choosing Java as the main language of implementation was a natural choice when comparing to the alternatives available. There were many possible operating systems for the NXT and the one used had to be easy to work with, easy to debug and fairly stable. Using anything with C or C++ as the development language would probably give fast code, when it was bug free. Using Java would make the job of writing bug free code much
Figure 5: Overview of the complete implemented system separated into two sub-systems, the server and sensor sides. The sub-systems communicated via bluetooth. The sensor side system was run as one instance on each sensor node.

easier and the risk of memory management bugs small. The potential loss of speed was less important than the development time aspect and hence Java was the best choice. Another important feature of leJOS was that it had an open source license. If the system needed patching it could be done, which is usually not the case with proprietary software.

5.3 Graphical user interface

To visualize what was happening inside the system and to give some interactivity a graphical user interface was implemented, see figure 6, using the standard Java libraries for graphics. The user interface design was mostly inherited from cats and only slightly modified. There were different tabs with possibilities to give orders, view the latest data etc. Under the orders tab the sensor nodes can be ordered to play some music as a clock synchronisation test. The human ear is very sensitive and can hear very small differences in time like if rhythms are of by fractions of a second. Since no external time source was available, this method was used to hear if the robots were synchronised.

At start-up, the network links to the robots were initiated by pressing the connect buttons for each robot. After a connection was established, the clocks would synchronise and the start order could be given in the order tab. The robots would then start to move if a regulator was used.

5.4 Network and communication

5.4.1 Sensors node communication

Communication between the sensor nodes and the server was implemented using bluetooth. There was no other realistic means of communication available. All sent data was first serialized and then transmitted as byte data over the bluetooth serial interface, which emulates a RS232 connection between the devices. The bandwidth was quite low but so was also the minimum latency. The round trip of a time synchronization request (one package per communication direction) was lower than 50 ms that should approximately be double the time for sending a network package one way. The size of packets vary very much but this should not have much impact on response time since the overhead seemed to outweigh it.

The range of bluetooth can be a problem. Some low power implementations can have seriously degraded performance already at a couple of meters. Using a more powerful antenna the effective range can be more than 10 meters (100 meters theoretical maximum in specifications), which was more than enough for this platform. However, some problems were encountered when trying to send packages with sightings and motor data. For some reason, a large latency and much lower bandwidth than earlier was measured. Solving this problem turned out to be both important and time consuming (more in section 5.4.3).

5.4.2 Global clock

It was obvious from the beginning that a shared time reference would be needed. This is a common problem in distributed systems. The standard approach used in time synchronization protocols such as the Berkeley algorithm and NTP is using Christian’s algorithm[23].
Figure 6: View of the main tab in the graphical user interface on the server side. The centre part of the window shows line of sight from three sensor nodes as red lines, surrounded by coloured landmarks. Note the connect buttons in the lower left corner.

Figure 7: Figure of the control flow in Christian’s Algorithm. $T_0$ and $T_1$ are the nodes local time when it sends and received the synchronisation package. $T_{\text{server}}$ is the servers local time at approximately at the nodes time $T = \frac{T_0 + T_1}{2}$.

In Christian’s algorithm it is assumed that the transmission time of sending a request from one network node to another is approximately the same as the transmission time of receiving the answer, and that the requests processing time is much lower than the latency. This makes the best accuracy of the synchronisation a function of the difference in network latency. Since there was only a few nodes on the piconet used here, these were valid assumptions. Basically the algorithm works as follows (shown in figure 7):

1. The sensor node prepares and sends a message requesting the servers time to the server.
2. The server parses the request, prepares an answer containing the current server time and sends the message.
3. The sensor node receives and parses the server’s answer.

If the running time of part 2 is assumed to be small compared to the running time of part 1 and 3 the sensors node can know the approximate network delay (equation 1). Knowing the delay the node can find what the server time is at time $T_1$ (equation 2). These equations are not exact and there will be an error, however small, in the estimation.

$$\frac{T_1 - T_0}{2} = d_{\text{network}} \quad (1)$$

$$T_{\text{server}} + d_{\text{network}} = T_{\text{global}}(T_1) \quad (2)$$

The accuracy of the synchronisation between the server and a node must be a value lower than the total request time, but it was not known how much. The round trip time was chosen as the accuracy measurement.
since it is known and always larger than the actual accuracy. In experiments, this was shown to be lower than 40ms (using only one sensor node, the accuracy was definitely much lower but how much was hard to prove without some kind of second source of time synchronisation). To test the accuracy, the sensor nodes played a C major chord together every other second (the human ear can easily hear the difference in time if relevantly high).

To make the accuracy as good as possible, each sensor node tried to synchronise with the server 10 times with a random delay between each request of $225 \pm 75$ ms. Only the best result was used, i.e. the one with the lowest round trip time. To account for a restart of the server clock, which occurred if the used interface was restarted, periodic synchronisation was performed. Every 10 seconds a synchronisation request was sent from each sensor node. If the difference in time was large or the accuracy better than the current one the sensors local time was updated.

5.4.3 Network latency

The network latency turned out to be a big problem. Testing of the given code turned out to have not only a low bandwidth but latency could rise to 15 seconds or more. The large latency lead to lags in the control loop beyond reasonable functionality. A strange phenomenon was also that the latency at times could rise beyond 15 seconds and stay there even though the sensor nodes stood still only sending a minimum of data.

The latency could easily be measured as the global clock often was a magnitude more accurate than the average latency. All information from the cameras and motors were given a time stamp, which could be compared to the server clock on arrival to the server.

Considerable time was spent on trying to improve the network throughput and latency. A plot of the network latency is shown in figure 8 where it can be seen both before and after the improvements. Lowering the need for bandwidth (explained below) was one part. The other was to redesign the bluetooth communication layer somewhat. Figure 8 shows the improvements of the latency testing with a similar stress on the network link. The error turned out to be the bluetooth code in leJOS, which did not behave as specified. The main solution was to request bytes from the bluetooth layers using a combination of interface methods for polling both individual bytes in a blocking manner and then pull larger chunks.

5.5 External filters

5.5.1 Rationale for external filters

Some filtering techniques were not suitable for implementation in Java. Java adds an extra layer between the user and the hardware, so if one wants to try a hardware dependent technique it is often complicated or impossible to do. Using an external filter, run as a separate process, gave the possibility to plug in any piece of code the user wanted. Also there was no middle man when doing calculations, as the Java memory management, and no garbage collector execution at random intervals. The restriction was that if the filter was to be used in real time, it had of course to be run at the same time as the Java engine. The data was however easy to store and do off-line tests later on, with a recorded session. This turned out to be useful for comparing different techniques using identical input data (this was used in section 13).
5.5.2 Inter process communication

The server software was written in Java. Some of the implemented filter code was written in C++ and was run as separate processes. There are many ways to achieve the inter process communication needed. Many include a mediator software package to be used, some use network sockets and Java has its own platform dependent ways (JNI) etc.

The filter code were launched by the Java application so that both standard input and standard output streams were accessible. If the input buffer of the filter could be written to in a way to export the data needed perhaps the output buffer could export filtered data back to the server. This turned out to work very well on a Unix system. The approach was very simple and it did not require any platform specific libraries or network interfaces. Reading from standard input is fairly simple in any programming language. Also since the data streams did not go both ways the data sent was trivial to store for later analysis.

The server launched the filters as external processes and wrote binary data containing information on bearings, positions etc. of the sensor nodes. The filters processed this data and wrote to its standard output new binary data containing estimates. The server then read the data from the filters and decoded the estimate. To get other data than tracking data from the filter process, the standard error output stream of the filters was used. Among other things for exporting information about execution time and run time parameters.

Using this set-up gave the opportunity to record data acquisition sessions, as the binary data given to the filters, by just piping it to a file. Using these recorded sessions the different filters performances could be compared with each other and with different run time parameters (number of particles/threads etc.) using the same input data by again piping it to the process.

5.5.3 Protocol buffers

The binary data protocol used in the inter process communication was built on Google protocol buffers[24]. This package generated code in both Java and C++ from a protocol packet definition. The work of encoding and decoding messages are completely done by this package. The only thing left to do was to get the message across. This was done by simply writing it in binary to the appropriate pipe (as described above). To know how many bytes a message contained, all messages was sent starting with a byte telling the length of the package.

The protocol buffer definitions used are shown in algorithm 1. Bandwidth economy was taken care of by the generated code by simply not sending empty data fields and the resulting packet size turned out to be very small. Some fields were marked as repeated meaning that they could dynamically be repeated any number of times in the same package.

The data chosen to be a part of the packages were all information relevant to the tracking. The names the packages SensorData and Estimate should be read as input data and output data. Using the protocol buffers to generate code made extending the protocol easy with full backwards compatibility.

5.5.4 OpenMP

OpenMP[25] offers a simple way of turning a program written in C++ into a threaded application. It is a parallelization framework that works by re-factoring the code, instructed by preprocessor directives. In figure 9, an example is shown. The compiler identifies the preprocessor directive that define the different parallel tasks, whether they be a for-loop with independent iterations or wholly free code segments that are independent. OpenMP will insert code to spawn the independent tasks in different threads. If the execution time of a task is significantly higher than the overhead needed for spawning a thread there will be a speed-up. Parallel tasks 1-3 in figure 9 are split into task blocks are run concurrently.

It is probable that the kernel scheduler will place an applications different threads on different cores (given that the utilization of the CPU is low). An application can then get access to the full (or at least close to full) potential of the CPU with only minor changes to the original code, with given system kernel scheduler constraints.

OpenMP was not used in the Java implementation of filtering but only in the C++ code described below.

6 Simulation framework

The Lego NXT with leJOS lacks reasonable debugging tools. The screen on the NXT is too small for any real debugging information, if an exception occurs only a reference to where and what kind of error is given. Some of the components, like camera control, naturally had do be tested and debugged on the NXT. This was not the case for some other components, like filters and data structures. Hence a simulation environment enclosing the easily portable components would make testing and debugging considerably easier. A bonus was also that compiling a single Java file and then running it locally took milliseconds, compared to the 20 seconds needed just for uploading the new software to the NXT.
Algorithm 1 The protocol buffer definitions used in the communication with the external filter processes. (section 5.5.3)

```plaintext
package trackingProtos;

option java_package = "GSim";
option java_outer_classname = "TrackingProtos";

message SensorData {
  required int32 SourceId = 1; // Sender ID
  required int32 TimeStamp = 2; // Global time stamp in ms
  optional double PositionX = 3; // Position of sensor
  optional double PositionY = 4;
  optional double SightingAngle = 5; // Angle (relative/absolute)
  optional int32 SightingType = 6; // Mouse/Landmark type
  optional double TurnedAngle = 7; // Radians turned
  optional double DrivenDistance = 8; // Distance driven in meters
}

message Estimate {
  required int32 timestamp = 1; // Time of latest used data
  optional double MeanX = 2; // Estimate mean
  optional double MeanY = 3;
  optional double VarX = 4; // (Co)Variance of position
  optional double CoVarXY = 5;
  optional double VarY = 6;
  optional double MeanXv = 7; // Estimate mean velocity
  optional double MeanYv = 8;
  optional double VarXv = 9; // (Co)Variance of velocity
  optional double CoVarXvYv = 10;
  optional double VarYv = 11;
  repeated double weight = 12; // Weight (one for each thread)
}
```

Figure 9: Figure of parallel tasks in different threads. Each task block is defined as such and then run in different threads using OpenMP to spawn threads and re-factor functionality. (Image from Wikipedia) (section 5.5.4)
The simulation was implemented using standard graphics libraries (result shown in figure 10) and emulated the same data structure and internal communication used on the NXT. A filter or regulator could easily be developed without access to the NXT hardware or tedious waits for uploads or compiles. When an exception occurred, it could be caught and analysed using any available debugging tool on a Linux machine. The basic structure of the simulation system is shown in figure 11.

The graphics code in the simulation drew the arena with landmarks, sensor nodes and target. All sensor nodes, called cats in the simulation, and the target, called mouse in the simulation, were separate objects with their own simulated motors and sensors. All moving objects implement the Actor class (figure 12) and lived in an environment similar to the NXT system.

The system iteration time was 10ms and all go() methods in the actor objects were called at each iteration, but graphics were updated less often to save CPU power. All filter and regulator components experienced time as they did in the real implementation. Noise was added in the motor controller (the class name is MotorControl) and sensor controller (the class name is SensorHandler). A limitation of this simulated environment was of course that all noise were ideal. Biases were also added for more realism.

7 Arena

The arena was the stage on which all tracking took place. It consisted of a restricted area in the shape of a square with a side of 3 meters in which the sensor nodes could move freely. To improve movement, the floor type needed to give good grip for the wheels but low enough friction for the rear sliding piece.

In each corner of the arena a landmark was placed, shown in figure 13. It could be seen by the camera on the sensor node and was used for positioning the node inside the arena. The landmarks were built from LEGO together with a light originally intended for decorating a computer box. The light was refracted through some plastic to make the bright area look bigger. The camera adjusted for average light intensity in an image but small light sources could overflow the intensity and be hard to find. Using larger and less intensive lights solved this problem. Still seeing the lights from too great a distance would generate the same problem. The arena is not big enough for this to happen in the current configuration.
Figure 12: UML sketch of the actor classes. (section 6)

Figure 13: Image of one of the landmarks used. (section 7)
8 Interfaces for implementing filters and regulators

A common feature of all base classes, for both filters and regulators, were the interfaces with their environment. To be able to use code both in simulation and in the real system, all communication with other components needed to be done by calling inherited methods or clearly specified buffers. The UML below describes the interfaces that was implemented for the respective type of component to work properly.

Since this platform was not intended for commercial use, most data in system components were not hidden. Making data accessible increases flexibility but can also create instability if not used properly. This implementation was a closed system so defining data as public does not give rise to any security concerns as it otherwise would.

8.1 Preprocessing filter - filter on brick

The name of the preprocessing filter is somewhat misleading. As it has been implemented for preprocessing, the name is correct. However, there is no real limitation (except system resources) of how much work this filter object can do.

Early during the implementation the need for a preprocessing filter was obvious. The data from the motors and camera were written to a buffer. This data has a high degree of temporal resolution. It was not productive to send all this data through the network since the bandwidth was low and the filters upstream often did not use all information anyway. Instead of aggregating the data after it had been sent it was better to aggregate it first and then send it. The result was the same but network communication was more sparse. In figure 15, the data flow through the preprocessing filter is shown. The data from the buffer comes in order of time stamp to the filter that choses what and when to send data upstream. The filter is also the most appropriate component to keep track of where the sensor node is at any given time, since it has access to all the motor data. When new information on position is received from the server, it is updated in the preprocessing filter using the initData method.

Extending this component to a combined regulator and filter or letting it spawn extra threads or classes is possible. The preprocessing filter specification was written as to be a black box, from the rest of the systems point of view. Only the interfaces needed to be implemented correctly (shown in figure 14). For example the regulator data received from the network were written to a data exchange component that was public to all objects running on the virtual machine. If the preprocessing filter wrote a movement command to the data exchange component the command would have been handled by the motor controller in the same manner as if had it come from the network.

The reason for implementing this filter in Java and not in something more close to the hardware had to do with the operating system on the NXT. Writing some extra component in C would demand detailed knowledge of leJOS to be able to guarantee stability of the system. Information would still have to be handed over to the C part and then back to the Java code, which would transmit it upstream. Using Java, as mentioned earlier, also decreased the risk of bugs compared to C.

8.2 Filtering phase 1 - estimation of sensor node positions

The first phase of the filtering on the server tried to position the sensor nodes. Every type of phase 1 filter implemented the abstract class shown in figure 17. The inputs to the filters were provided through a buffer written to by the networking code. The buffer provided information on bearings and movements using data containers suitable for the respective type of data. This data was in essence a reconstruction of the output
from the preprocessing filter on each NXT. The filter handler that instantiated the phase 1 filters sorted the incoming data by time and provided only the data from one sensor to each buffer (as shown in figure 16).

All data was sent through the node-server network interface and extending the transfer for new demands would be trivial. The bearing data was given in each sensor node’s own referencing system where the zero angle signified forward from the node’s point of view. The sighting input data always came with a time stamp, id of source, angle of sighting and type of sighting (tracked object or landmark type as defined in the settings file). The movement input data always contained a time stamp, id of source, turned angle (even if zero) and distance travelled (even if zero and negative if node had driven backwards).

The output data from the filter was set in the common referencing system. A crucial point here is that the data in the given data containers were only modified, allowing for extending the data contained. New fields in the data containers would only be passed on to the output buffer. Output data was written to the outputBuffer component and in that way passed on to the phase 2 filter.

Each instance of the phase 1 filter spawned would be assigned an id to work with. The output data always contained a time stamp, id of node, estimated position of node and angle of sighting. Period time \( T \) was given in milliseconds. For compatibility, a co-variance matrix was estimated even if it is not a good representation of the distribution of certainty.

The filter was implemented as its own thread and the \texttt{update()} method was called once each period specified in the settings file.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure16.png}
\caption{Data flow on the server.}
\end{figure}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure15.png}
\caption{Chart of the data flow on the NXT with the preprocessing filter. (section 8.1)}
\end{figure}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure17.png}
\caption{Base class UML diagram of the phase 1 filter used for sensor node position estimation. (section 8.2)}
\end{figure}
8.3 Filtering phase 2 - estimation of target position

After the phase 1 filter had estimated the position of the sensor nodes, the phase 2 filter tried to estimate the position of the non-cooperating target. All information to the phase 2 filter passed through a data buffer. The bearings were given in the common referencing system, to the best of the phase 1 filters ability.

Estimates from the tracking were polled from outside using the appropriate methods in figure 18. When a filter iteration was done, the output from these methods reflected the new information given to the filter. The filter was implemented as its own thread and the update() method was called once each period as specified in the settings file.

For the estimation of the non-cooperating target, some filters were written in C++. Using these filters required some compiling and then loading a phase 2 filter called TrackingExternalFilter. This Java filter spawned the chosen C++ filter in a separate process from the Java engine.

8.4 Regulator

The regulator component had access to the information from the filters knowing the estimates of the states of the sensor nodes and the non-cooperating target. Using this information it predicted the future and made decisions on how to move the sensor nodes. For the regulator there was an easy interface for feedback to the sensor nodes implemented, as a front for the network code and inherited from the Regulator base class. A UML diagram of the base class is shown in figure 19.

All sensor nodes had a state accessible by the regulator that said if the node was moving, sweeping for landmarks or waiting for a command. There was a slight delay in the feedback for obvious reasons. Every issued move order contained the current position of the sensor node, this was the feedback mechanism for the phase 1 filter to the preprocessing filter on each NXT.
Part III
Implemented filters and regulators

9 Particle filtering

9.1 Particle filter theory

The problem consists of finding the position of the sensor nodes and the non-cooperating target using only noisy sensor measurements. To achieve this, a basic model of the tracked units must be defined and the relevant information found in the noisy data. The type of algorithm used to solve this task is called a particle filters. Here follow a short overview of the theory behind particle filters, for more details see references.

9.1.1 State space model in estimation

To describe a dynamic system, it is modelled mathematically by using differential equations and a number of variables representing the state of the system[26]. The state variables used should be the smallest subset of the possible system variables that describe the system completely or, for the application, to a sufficient degree. The vector containing the state variables is called the state vector \( x \in \mathbb{R}^d \), where \( d \) is the number of tracked state variables.

The basic equations of a state space model can be seen in equations 3 and 4, which also are the equations used in the estimation algorithms described below. The model is subject to some initial condition \( x_0 \in \mathbb{R}^d \), which in the case of tracking corresponds to the initial guess of the systems state.

Equation 3 describes the relation between the system state at the current time \( k \) and the state at the next time step \( k + 1 \). The vector \( x_k \) is the state vector at time \( k \), the function \( f(\cdot) : \mathbb{R}^d \rightarrow \mathbb{R}^d \) represent the relation between the state at time \( k \) and time \( k + 1 \) (i.e. a prediction) and \( w_k \in \mathbb{R}^d \) is the process noise at time \( k \).

\[
x_{k+1} = f(x_k) + w_k
\]  

The model can be continuous, in which the left hand side of equation 3 represent the derivative, but is here discreet. Equation 4 shows the relation between the state vector and sensor data. The vector \( y_k \in \mathbb{R}^e \) is the sensor data at time \( k \), the function \( h(\cdot) : \mathbb{R}^d \rightarrow \mathbb{R}^e \) represent the relation between the state vector and sensor data, the vector \( v_k \in \mathbb{R}^e \) is the sensor noise at time \( k \). Note that the state vector is of dimension \( d \) and the sensor data of dimension \( e \).

\[
y_k = h(x_k) + v_k
\]  

The state vector can be estimated using the standard Kalman filter when the functions \( f(\cdot) \) and \( h(\cdot) \) are linear (i.e. matrix multiplications) and \( w_k \) and \( v_k \) are Gaussian noise. However, when solving a bearing-only problem the function \( h(\cdot) \) is non-linear and the vector \( v_k \) is not necessarily Gaussian. How the function \( h(\cdot) \) is used is shown below, for more detail see references.

One way of solving this kind of problem is to use a Kalman filter that is able to handle non-linearities, such as the extended or unscented Kalman filter. These filters both depend on a linearization of the problem, which can introduce errors and stability problems. Here a particle filter will be used which can handle the non-linearities of the function \( h(\cdot) \), as described below.

The state vector chosen for the problem of tracking the non-cooperating target is given in equation 5.

\[
x_k = \begin{pmatrix} r^k_x \ r^k_y \ \nu^k_x \ \nu^k_y \end{pmatrix}^T \in \mathbb{R}^4
\]  

The variables \( r^k \) represent position in the \( x \) and \( y \) directions of the robot at time \( k \). The state vector for the non-cooperating tracked object is its position and velocity. Acceleration is not estimated since the velocity is known to be small enough for not including higher order derivatives.

The state vector chosen for the problem of tracking the sensor nodes is given in equations 6.

\[
x_k = \begin{pmatrix} r^k_x \ r^k_y \ \theta^k \end{pmatrix}^T \in \mathbb{R}^3
\]  

The variables \( r^k \) represent position in the \( x \) and \( y \) directions and \( \theta^k \) is the heading angle of the robot in radians at time \( k \). The reason for choosing position and heading angle as states for the sensor nodes are that distance travelled and angles turned are the only information delivered upstream. Including velocity in the model would not add any accuracy and it is hence not estimated.
9.1.2 Particle filter

The particle filter\cite{27, 28, 29}, also known as sequential Monte Carlo methods, is a state estimation algorithm that solves the estimation problem recursively using Monte Carlo simulation. The possible future states are simulated from an input distribution, called the \textit{apriori} estimate, that is updated with time and compared to sensor input. Through a Bayesian model a new distribution, called the \textit{aposteriori} estimate, is formed containing the sensor information from all previous iterations. The simulated possible states are called particles and it can be proven that if the number of particles are sufficiently high the estimate approach the Bayesian optimal estimate. This also means that for good estimation accuracy a large number of particles are required, which makes the algorithm computationally expensive. The algorithm works as follows:

1. Draw new particles from \textit{apriori} distribution  
2. Update particle (i.e. prediction as shown in equation 3)  
3. Compare particles with sensor data (example in section 9.1.4, equation 8)  
4. Calculate new \textit{aposteriori} distribution (example in section 9.1.4, equations 9 and 10)  
5. Re-sample (if needed) (section 9.1.7)  
6. GOTO 2

In the state space model, equation 3 corresponds to the update step and equation 4 the comparison step. The particle filter has been proven to handle non-linear/non-Gaussian dynamic systems, such as in a bearings-only problem, successfully. By using the appropriate weighing functions in the comparison step most types of sensors can be used together, making the particle filter very suitable for sensor fusion. The estimation given is a probability distribution and it is the mean of this distribution, which here will be used as the estimated state vector.

9.1.3 Parallel implementation

All algorithms are, by definition, expressed as a finite list of well defined instructions. These instructions should be performed sequentially and in order. Often though an algorithm's different steps can be clustered into independent blocks or steps might be repeated where every iteration is independent. This structure lets a programmer implement the algorithm in a way to allow independent blocks to be executed concurrently.

A common, and easy, way to achieve parallelization is to fork off slaves from the main process and letting them return with a computationally heavy function's result, letting the operating system dispatch the threads to different cores. Another, more network friendly, way would be to have dedicated processes waiting for work, on the same or different machines. Which one to use depends on the nature of the problem, the hardware available and the programmer's taste. The more of the algorithm that can be run in parallel the better it can utilize the processing power of a multi-core machine. An important difference between different approaches and hardware are the constraints on communication, and hence on shared information between parts of an algorithm. The algorithm might be very easy to run in parallel but communication then needed might cost execution time outweighing the benefits of parallelization. To achieve a good parallelization the algorithm should, in general, have independent and computationally heavy blocks with little input and output data. Reference data that does not change much over time can however be large.

In the particle filter procedure described above, the independent steps are the propagation of particles and comparing particles to sensor data. Tasks calculating state vector estimate and probability distribution are not always independent but often the calculations can be split up into independent subtasks, leaving little work for the main thread. Some approaches to particle filtering are more similar to using several different filters in parallel with some information exchange instead of one filter. One such method is described in section 9.1.6.
9.1.4 Gaussian Particle Filter

The Gaussian particle filter is a very basic approach to particle filtering. The algorithm is briefly described below, for a more detailed description and derivation of this filter see [12].

1. A set of particles are sampled from an *a priori* Gaussian distribution described by a mean (i.e. the current leading hypothesis of what the state vector look like) and a covariance matrix. The particle weights are uniform (i.e. set to $\frac{1}{N}$, where $N$ is the sample size).

2. All particles are updated according to the given model of the system, updating the samples from the *a priori* distribution to incorporate the fact that some time has passed. Each particle is updated individually. Process noise is added symbolising the uncertainty of the model.

3. All particles are compared to measured data in the current time frame, and their weights are updated. The sensor error is here assumed to be Gaussian so the weight update function (equation 8), which comes from Bayes’ theorem, is similar to the equation describing a Gaussian probability density function. In equation 8 variable $\sigma$ corresponds to the standard deviation of the expected error in the sensor bearing measurement, $w_i$ to the weight of the $i$th particle before the weight update and $\hat{w}_i$, to the weight after the update, $\theta_{error}$ to the angular error of the particle. There are usually many sensor readings during a time frame and all of those can be used in the particle evaluation. In a real world application, it can be important not to make too many evaluations before re-sampling since weights can get too small for the machine to handle correctly.

$$\hat{w}_i = w_i \cdot e^{-\frac{{\theta_{error}}^2}{2\sigma^2}}$$  \hspace{1cm} (8)

The weight update in equation 8 must be performed once for each particle and sensor reading (where $w_i$ is the weight associated with the $i$th particle, $i = 1...n$, from the vector $w \in \mathbb{R}^n$ and $\hat{w}_i$ is the same updated weight).

4. The last step of the algorithm is to calculate the new estimate. The mean and co-variance matrix is estimated using equations 9 and 10, where $k$ and $j$ are vector or matrix element indices, $x_i$ is the state vector of the $i$th particle, $w_i$ the weight of the $i$th particle.

$$\mu = \frac{\sum_{i=0}^{N} w_i \cdot x_i}{\sum_{i=0}^{N} w_i}$$  \hspace{1cm} (9)

$$C_{jk} = \frac{\sum_{i=0}^{N} w_i \cdot (x_i^j - \mu^j)(x_i^k - \mu^k)}{\left(\sum_{i=0}^{N} w_i\right)^2 - \sum_{i=0}^{N} w_i^2 \sum_{i=0}^{N} \sum_{m=0}^{N} (x_i^j - \mu^j)(x_i^k - \mu^k)}$$  \hspace{1cm} (10)

In equations 9 and 10, $\mu$ is the new mean vector (i.e. the *a posteriori* state vector), $C$ is the covariance matrix.

The new estimate incorporates both the new sensors readings and the old information given before the iteration. The only data that is carried over from one iteration to the next is the mean and covariance matrix of the estimated distribution. This makes this filter ideal for being distributed over many nodes with heavily constrained communication. If implemented correctly most of the algorithm might even be run in the processor cache, since there is no need to store the particles for the following iterations.

9.1.5 Globally distributed particle filter

The globally distributed particle filter is a filter for a shared memory environment. All particles are shared between the working processes but are divided between them at some steps of the algorithm. The advantage being that parallelization is easily implemented since all particles are available at any given time. The mean is calculated using equation 9 just as with the Gaussian particle filter.

Since this filter does not try to make the particles conform to any special distribution, it can by its nature estimate any distribution. The re-sampling method used is the so called systematic re-sampling, which is a common approach to re-sampling (described in section 9.1.7). It can re-sample any distribution of particles and is not confined, as in the case of the sampling of the Gaussian particle filter, to a Gaussian distribution.

9.1.6 Locally distributed particle filter

The locally distributed particle filter is a filtering method suitable for distributed calculation without a shared memory. Every node is independent and creates its own particle set on which to perform propagation, evaluation, re-sampling and calculation of an estimate. This approach is very suitable if there for example are many physically separate nodes, connected by a network, that do the filtering.
When not being able to communicate any information desirable to other processes, there is a risk of weight starvation. This happens when a large number of particles does not fit with the sensor data. Calculations on a particle with a low weight cost as much CPU time as calculations on a particle with a high weight. Weight starvation hence leads to a lot of wasted processing power. To keep the threads together as one filter and avoid starvation, each node exchanges a portion of its particles every iteration with other nodes. The possible exchanging of particles is constrained by the network capabilities (bandwidth, latency etc.). This leads to a trade-off where the filter needs to communicate enough particles to avoid starvation but still keeping network traffic low. The minimal exchange ratio needed to avoid weight starvation in this implementation will be investigated below.

The re-sampling method used is systematic re-sampling, the same as with the globally distributed particle filter. To form a global estimate, local estimates are transmitted to a main thread/node together with a weight.

### 9.1.7 Systematic re-sampling

Systematic re-sampling is a common approach to re-sampling. In this algorithm, the weights of the evaluated particles are used to give each particle a replication factor relative to its normalized weight. Particles with low weight can receive a replication factor of zero, which means that the particle should be removed. If some particles have relatively very high weights, they might be replicated many times and after re-sampling dominate the set. The sum of the replication factors are often the same as the total number of particles, which is used here, but could be adaptive if necessary, resulting in a changing size of the particle set. This approach is used in both the locally and globally distributed particle filters. Since good particles are replicated using only the weights, any distribution of particles can be re-sampled using this technique. After re-sampling all weights are reset to \( \frac{1}{N} \) (where \( N \) is the total number of particles), i.e. all particles have equal importance.

### 9.2 Implemented filters

Below, filters for preprocessing and both phases of the filtering are described. The C++ implementation of the phase 2 filters were developed together with Olof Rosén as a continuation of his work on developing filters for estimation on multi-core platforms. The important addition here is to make the idealised models work in a real-world environment.

#### 9.2.1 Basic preprocessing filter

This filter tried to aggregate the data on the NXT. Many motor data posts were read from the input buffer and aggregated into one post before being sent upstream. In this way, a lot of bytes were saved but most of the information contained was kept (temporal resolution was lowered though). This method saved up to 80% of the otherwise used bandwidth.

The class was called `DataAggregator` and was enabled with the `PREPROCESSING_AGREGATOR` flag in the settings file.

#### 9.2.2 Preprocessing filter with classifier

The class was called `DataClassifier` and was enabled with the `PREPROCESSING_CLASSIFIER` flag in the settings file.

This preprocessing filter contained a classifier for the bearings of landmarks. Some bearings were clearly erroneous and came from the camera micro-controller interpreting a colourful object (e.g. a reflection from the sunlight) as a landmark. These bearings could easily be removed using information on sensor node position and landmark positions and colour. If the angle to a landmark sighting was off by too many degrees it could safely be assumed that it was erroneous and should be considered as noise. Using this method assumed the sensor nodes own estimate of its position to be fairly correct and proceed to work well.

Classifying bearings on the brick made the data aggregation easier. Since some sightings could be removed, data could be aggregated even more. A reduction both in extreme errors and bandwidth demand of slightly more then 80% (compared with no preprocessing) was observed using this method.

#### 9.2.3 Preprocessing filter with aggressive classifier

The class was called `DataClassifierAggressive` and was enabled with the `PREPROCESSING_CLASSIFIER_AGGRESSIVE` flag in the settings file.

The aggressive classifier aggregated data even more than the preprocessing filters described above. All changes of positions and bearings of landmarks were aggregated without any regard to their temporal relation. This approach assumed that changes to the robots position were relatively small during the period of the filter.
9.2.4 Phase 1 Gaussian particle filter in Java

This filter was the first implemented filter for positioning of the sensor nodes. The class was called `AbsolutePositioningParticleFilterGaussian` according to a naming convention that had grown out of continual changes. It was enabled by setting the `USE_POSITIONING_PARTICLE_FILTER_GAUSSIAN` flag in the settings file.

This filter tried to position the sensor nodes using the approach of the Gaussian particle filter as described above. The mean of the sampled particles varied slightly around the ideal mean, since a small number of particles was used, which led to a drift in the estimate. However the drift was shown to be small. Some drift could also sometimes be good if the sensor nodes position was off and it needed correcting.

9.2.5 Phase 1 particle filter with systematic re-sampling in Java

This filter used the same approach as the globally distributed particle filter to estimate sensor node position. It was enabled by setting the `USE_POSITIONING_PARTICLE_FILTER_SYSTEMATIC` flag in the settings file. In Figure 20, a session using this filter in the Java simulation environment is shown.

9.2.6 Phase 2 Gaussian particle filter in Java

This filter implemented the Gaussian particle filter technique in Java. This filters was enabled by settings the `USE_TRACKING_FILTER_GAUSSIAN` flag in the settings file.

9.2.7 Phase 2 Gaussian particle filter in C++

This filter implemented the Gaussian particle filter technique in a C++ implementation using OpenMP for parallelization. It was enabled by setting the `USE_TRACKING_FILTER_EXTERNAL` flag in the settings file. Additionally, the external filters path had to be set in the `EXTERNAL_TRACKING_FILTER_CMD` String constant in the settings file.

9.2.8 Phase 2 globally distributed particle filter in C++

This filter implemented the globally distributed particle filter technique in a C++ implementation using OpenMP for parallelization. It was enabled by setting the `USE_TRACKING_FILTER_EXTERNAL` flag
Figure 21: Model Predictive Control scheme. The reference signal and the predicted system state converge in the simulated future, using some set of control signals. (image from wikipedia) (section 10.1)

in the settings file. Additionally, the external filters path had to be set in the $EXTERNAL\_TRACKING\_FILTER\_CMD$ String constant in the settings file.

9.2.9 Phase 2 locally distributed particle filter in C++

This filter implemented the locally distributed particle filter technique in a C++ implementation using OpenMP for parallelization. It was enabled by setting the $USE\_TRACKING\_FILTER\_EXTERNAL$ flag in the settings file. Additionally, the external filters path had to be set in the $EXTERNAL\_TRACKING\_FILTER\_CMD$ String constant in the settings file.

10 Regulators

10.1 Model Predictive Control

Model predictive control is a common technique used for controlling systems where future estimates can be calculated from current state information using a model of the system and effects of changes from control signals can be predicted.

The idea[31, 26] is to predict what will happen to a system a limited number of time steps into the future, the so called prediction horizon. During some of these time steps, control signals can be inserted into the system limited in time by the so called control horizon. From the control signals inserted, the future states of the system can be predicted. The MPC regulator tries to find the set of control signals that minimizes some criterion function designed to make the system converge to some reference signal.

The optimal control signals are usually determined by using optimization techniques. There is no general cost function. Instead a function must be found for any new system that a regulator is designed for. The cost function will be some combination of what happens to the system during the prediction horizon. Usually there are also some constraints on the range of the control signals (e.g. maximum output power). Using this information of the system the regulator can predict the future development of a particular control signal set to find an optimal control strategy.

An example of this kind of control strategy is shown in figure 21. The system is dynamic and the model might be very complex. Some control signal is inserted into the system at every time step trying to get the reference signal and the predicted future to converge. The strength of this strategy is that if the model is approximately correct a very complex system can be controlled solving a problem a human could not.

MPC tries to find an optimal solution to the control problem but is always limited by the quality of the model. This kind of problem is usually solved using sequential quadratic programming or similar optimization technique but other methods were tried here as described below.

10.2 Heuristic regulator

The most basic approach to the regulator problem given was to make a simple set of rules solving the regulator problem. Solving a problem in this way can be quite tricky since stability and other desirable features of a solution can be hard to prove. The kind of problem that was solved here however was very intuitive.

For the cameras to give precise bearings on the target they need to be close to the target, but if they are too close a collision might occur. This pointed to that there would probably be some kind of optimal range of
distance. Another concern was that the robots needed to make sure they did not collide with each other even when the estimated positions were off (within reason). Lastly, the intersection of the sensors lines of sight to the target should not be at too sharp an angle. The information gained with small angles of intersection would almost be comparable to using just one sensor but increase rapidly when the intersection angles were increased.

A set of rules was constructed where a desirable distance between the target and each sensor were set and the intersection angles of the sensors line of sight would be maximized (possible only if there were three or more sensor nodes). Also any order issued would not be able to move the sensor node outside the arena, with a safety buffer at the edges to allow for some errors in the estimation of their positions.

The regulator was somewhat conservative when issuing movement commands since moving introduce errors in the estimated position of the node. All movement was subjected to a threshold keeping the sensor nodes from moving too often. Every movement, however small, would have to be weighed against the possible advantage of gaining a better position for the camera.

10.3 Formation regulator

For demonstration purposes and testing the long term correction capacity of the phase 1 filter, a regulator was used to make the robots drive in formation. It was built on the heuristic regulator but made the sensor nodes go around the centre of the arena in a circle.

Each sensor node moved between a number of points around the circle. The number of points was the double of the number of sensor nodes. Using this regulator with three sensor nodes, as done here, gave a hexagon formation to move around. The results from using this regulator is shown in section 12.

10.4 Search tree regulator

A search tree approach to finding the optimal control signals have been shown effective[32] when controlling robot movements. This is an approach in which the movement options at each time step were searched through using a recursive function. This formed a tree of possible paths to take where the one with the highest information gain was chosen.

The estimation of the possible information gained by each movement was a merge between many estimated probability density functions representing what each robot would be able to contribute at a given time step. Assuming the sensor error was a Gaussian distribution the information gained by each sensor can be easily parametrized. The sensor fusion was done by analytically merging the covariance matrices of the distributions[3]. In this case, maximum information gain correspond to minimizing the estimated covariance matrix.

10.5 Implementation

Both the heuristic regulator and the formation regulator worked very well in simulation and were ported to the production repositories. They both performed as they should using the robots. A risk of a collision using the formation regulator was observed but small.

The search tree regulator did not perform well in simulation and was due to time constraints not ported to production repositories.

Part IV

Experiments and results

11 General performance

11.1 Sensor accuracy

A test on the sensor accuracy was performed. The robots where placed at coordinates (0.7, 1.7), (1.9, 1.2) and (0.7, 0.7) in the arena and given orders to scan for landmarks. All data from the sensor nodes where recorded and imported via Matlab to give the histogram in figure 22. The histogram shows how the sensor nodes perceived their environment. The red colour is the tracked object, which was turned off shortly after the data collection began. It is included here to illustrate the risk of noise as the red bars can be seen far from the mean of the red sightings. In reality, some bearings might be completely off. It is very clear in the histogram that the blue landmark show up as two peaks next to each other. This was probably because of the light conditions during the test. The camera micro-controller was looking for blue regions in the image and has probably seen the outer edges of the light and gotten over exposed in the centre.
Figure 22: Histogram of sightings for each sensor node when scanning for landmarks. The sensor nodes were placed at different places in the arena and did not move. The white landmark is shown as black. Numbers on peaks and samples can be found in table 1. (section 11.1)

Figure 23: **Left:** Image of the line tracker robot used as non-cooperating target. **Right:** Sketch of the track that the line tracker followed (vector graphics of the printed version). (section 11.2)

In table 1, some statistics on the resulting data are shown sorted by id of sensor and colour of landmark. In the last column, the estimated standard deviation of the sensor error is shown. It is also clear that a Gaussian approximation of the error profile is a reasonable assumption. These result were used to improve the model for the sensor noise in the particle filters.

### 11.2 Moving target

#### 11.2.1 Line tracker as target

As a non-cooperating target that could be tracked by the sensor nodes, a line tracking robot was used (show in figure 23). A track, also shown in figure 23, was printed on a large paper with a colour gradient as track. The gradient let a calibrated light sensor act as the input signal to a PID-controller letting the robot drive in the middle of the track. Both the sensor nodes and the server knew nothing of the track geometry. How far along the track the line tracker had moved was communicated to separate software on the server and by knowing the position and geometry of the track, the position of the line tracker at a given time could be estimated. During a data collection session, this data was saved as reference data for the accuracy tests.
<table>
<thead>
<tr>
<th>Id</th>
<th>Colour</th>
<th>Number of sightings</th>
<th>Variance</th>
<th>Standard deviation [rad]</th>
<th>Standard deviation [degrees]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Red</td>
<td>86</td>
<td>2.0998</td>
<td>0.2291</td>
<td>13.1263</td>
</tr>
<tr>
<td>0</td>
<td>Purple</td>
<td>103</td>
<td>0.000323894</td>
<td>0.017908</td>
<td>1.0312</td>
</tr>
<tr>
<td>0</td>
<td>White</td>
<td>163</td>
<td>0.0026947</td>
<td>0.05191</td>
<td>2.9742</td>
</tr>
<tr>
<td>0</td>
<td>Green</td>
<td>96</td>
<td>0.00010651</td>
<td>0.010032</td>
<td>0.59131</td>
</tr>
<tr>
<td>0</td>
<td>Blue</td>
<td>212</td>
<td>0.0075033</td>
<td>0.086022</td>
<td>4.9631</td>
</tr>
<tr>
<td>1</td>
<td>Red</td>
<td>89</td>
<td>0.024834</td>
<td>0.15759</td>
<td>9.0292</td>
</tr>
<tr>
<td>1</td>
<td>Purple</td>
<td>104</td>
<td>0.00036348</td>
<td>0.019065</td>
<td>1.0923</td>
</tr>
<tr>
<td>1</td>
<td>White</td>
<td>280</td>
<td>0.002276</td>
<td>0.047707</td>
<td>2.7334</td>
</tr>
<tr>
<td>1</td>
<td>Green</td>
<td>98</td>
<td>0.00023169</td>
<td>0.015221</td>
<td>0.87213</td>
</tr>
<tr>
<td>1</td>
<td>Blue</td>
<td>301</td>
<td>0.01731</td>
<td>0.13157</td>
<td>7.5384</td>
</tr>
<tr>
<td>2</td>
<td>Red</td>
<td>91</td>
<td>0.070847</td>
<td>0.26617</td>
<td>15.2505</td>
</tr>
<tr>
<td>2</td>
<td>Purple</td>
<td>31</td>
<td>0.0001969</td>
<td>0.014032</td>
<td>0.8097</td>
</tr>
<tr>
<td>2</td>
<td>White</td>
<td>43</td>
<td>0.054277</td>
<td>0.073673</td>
<td>4.2211</td>
</tr>
<tr>
<td>2</td>
<td>Green</td>
<td>166</td>
<td>0.0020488</td>
<td>0.045275</td>
<td>2.5941</td>
</tr>
<tr>
<td>2</td>
<td>Blue</td>
<td>217</td>
<td>0.031866</td>
<td>0.17851</td>
<td>10.2279</td>
</tr>
</tbody>
</table>

Table 1: Table of results from sweeping sensor test also shown as histogram in figure 22. Red is the target colour while other colours correspond to landmarks. The Id column correspond to different ids of the robots. Note that there is a large difference between standard deviations depending both on colour and robot id. This is because some colours are more visible to the cameras, assuming lighting conditions, and also because of the distance between each robot and landmark. (section 11.1)

### 11.3 Test scenarios

Several test scenarios were run using the complete platform. Combinations of moving or stationary sensors and target were decided on.

- With both target and sensor nodes stationary the tracking performance without regulator or phase one filtering could be performed. The sensors were set up close to each other to test tracking capabilities in a very non-linear situation.
- With a moving target and stationary sensors the tracking capabilities of the non-linear estimation set-up (as above) could be tested. Here the estimate would move, making the problem harder to solve and also testing the estimation of velocity.
- With the target stationary and moving sensor nodes the capabilities of the implemented platform could be tested. This test case is good for seeing if all components work as they should disregarding time.
- With both sensors and target moving, the full capabilities of the platform could be tested. This would be the hardest case demanding reasonable functionality from every component.

### 11.4 Tools for data analysis

A number of tools for data analysis were developed.

All data available on the server side was saved in the binary format provided by the Google protocol buffers. To be able to read the data and do analysis, tools were written in Matlab using Java hooks for reading the binary data. The export of information on latency was written by the Java server software as a CSV file and could easily be imported into Matlab.

The benchmarking was done using python as a code generator running shell commands. Output from the filters were saved as binary files and given reference numbers. There reference numbers were also written to a CSV file with meta data (such as filter type etc.). All data amounted to many gigabytes and analysis was automated needing little supervision.

### 12 Phase 1 filter performance

#### 12.1 Test of drift

It was important that the filter did not show significant drift while the position of the sensor nodes were fixed. To test this, the sensors were placed inside the arena and ordered to scan for landmarks. The landmark data was then processed through the phase 1 filters (estimating the position of each node) and the regulator tried to keep the robots at fixed positions. The non-cooperating target was not used during this test.
Figure 24: Plot of error for three sensor nodes set to place themselves at different fixed positions in the arena. The error is measured as estimated distance from the reference point. Most importantly the drift in the estimating filter is shown as a slight change in the errors over time. Sharp changes are due to changes in the sensor node's position. (section 12.1)

Two filters were implemented for estimating each robot's position. A Gaussian particle filter and a particle filter with systematic re-sampling. Only the Gaussian particle filter was tested. The filter with systematic re-sampling added noise to the particles when turning or moving, the Gaussian particle filter adds noise with time, making drift over time impossible.

Since at re-sampling a limited number of particle values are drawn from a Gaussian distribution, some drift may occur and the mean will move slightly. The errors, measured as distance from a reference position, for three sensor nodes are shown in figure 24.

In the dataset generated for this test the sensor nodes were at start-up placed at the side of the arena, hence during the first 30 seconds of the plotted data the error is very high. The set points of the sensor node positions were the coordinates (0.625, 1.875), (1.250, 1.250) and (1.875, 0.625). Essentially they were set to place themselves on a diagonal line through the arena. The figure shows what the filters believed the errors to be and was confirmed by manually measuring the error. The drift in the estimating filter is shown as a slight change in the errors over time. Sharp changes are due to changes in the sensor node's position.

This test was performed many times with similar results but only one dataset is presented here. The manual measuring shows that the estimation were accurate up to less than a decimetre. The drift was not significant compared to the size of the arena.

12.2 Test of correction over time

The correction test consisted of making the sensor nodes drive in a formation for an extended period. There was no independent way of getting reference data to compare the accuracy of the phase 1 filter. However, if the correction did not work the drift in the position was enough to make the server lose track of the position of the sensor nodes. For this test, the formation regulator was used to give orders of formation driving in a hexagon shape. The data series was generated over some time to give the sensor nodes the opportunity to fail. This test was not intended to show the accuracy at any given time but to show the capabilities over time. Results using the filters are shown in figure 25.

The correction was good enough not only to keep the robots on the arena but also to track the non-cooperating target and compensate for large errors.

13 Phase 2 filter performance

13.1 Performance measures and data sets

The accuracy of the filters were measured by the euclidean distance (equation 11) from the reference point at a given time.

\[ d_{error} = \sqrt{(x - \hat{x})^2 + (y - \hat{y})^2} \]
Figure 25: **Left:** Plot of the estimated path driven by the sensor nodes using the phase 1 Gaussian particle filter. **Right:** Plot of the estimated path driven by the sensor nodes using the phase 1 filter with systematic re-sampling. Lines connecting data points show relations in time. Note that this shows the estimated path and not the real path. This test shows that the phase 1 filter can compensate for errors in the movement to a sufficient degree to perform tracking of the non-cooperating target. (section 12.2)

The reference information came from knowing where on the track the line tracker robot was at a given time.

The scalability of a filter was measured as speedup\[^{[33]}\). The speed-up \((S_p)\) is defined in equation 12 where \(T_1\) and \(T_p\) is the execution time with 1 and \(p\) number of processors, respectively.

\[
S_p = \frac{T_1}{T_p}
\]  

The speedup is a measure of how much faster a piece of code runs if split up into parallel parts. In the ideal case, the speedup should be linear i.e. if a program is allocated \(N\) processors it runs \(N\) times faster than when using one processor. The ideal case assumes that an algorithm can be split up into \(N\) parts with equal execution time. This is seldom attainable since most algorithms have a significant part that must run sequentially or with uneven load. Another limitation if overhead cost (e.g. from moving data) that might limit the performance of the hardware.

The tests of the second phase filtering was done using all three external filters implemented in C++.

13.2 Accuracy test

For the accuracy test each filter were run with 2000 particles 50 times changing the number of threads from 1 to 12. For the locally distributed particle filter a particle exchange ratio of 15\% were used, as recommended by \[^{[12]}\).

Accuracy of the Gaussian particle filter is shown in figure 26 and 27. Accuracy of the globally distributed particle filter is shown in figure 28 and 29. Accuracy of the locally distributed particle filter is shown in figure 30 and 31.

The figures show the mean of the tracking error in blue and \(\pm 2\) standard deviations in grey using 1, 4, 8 and 12 threads to the left and a histogram of the distribution of errors using 12 threads to the right. The blue areas are quite compact since the time series is long and the estimate moves when a new data is received. Data packets were received a couple of times per second and most resulted in a new estimate.

Overall performance was satisfactory. Note that the accuracy of the phase 2 filters are always limited by the accuracy of the phase 1 filters. The filters could all keep track of the non-cooperating target, as shown in the figures referenced above.

13.3 Scalability test

The speed-up of the external particle filters were investigated and are shown in figure 32. All filters were run with the same data sets and the execution time was accumulated and compared. Note that the lines in the figure are only for making the data clearer, measured points are shown with symbols.
Figure 26: Accuracy of the Gaussian particle filter on the first dataset. Plots of estimation errors while using different number of threads (left) and plot of error distribution as a histogram (right). (section 13.2)

Figure 27: Accuracy of the Gaussian particle filter on the second dataset. Plots of estimation errors while using different number of threads (left) and plot of error distribution as a histogram (right). (section 13.2)

Figure 28: Accuracy of the globally distributed particle filter on the first dataset. Plots of estimation errors while using different number of threads (left) and plot of error distribution as a histogram (right). (section 13.2)
Figure 29: Accuracy of the globally distributed particle filter on the second dataset. Plots of estimation errors while using different number of threads (left) and plot of error distribution as a histogram (right). (section 13.2)

Figure 30: Accuracy of the locally distributed particle filter on the first dataset. Plots of estimation errors while using different number of threads (left) and plot of error distribution as a histogram (right). (section 13.2)

Figure 31: Accuracy of the locally distributed particle filter on the second dataset. Plots of estimation errors while using different number of threads (left) and plot of error distribution as a histogram (right). (section 13.2)
Speedup of the external particle filters

The speedup was almost linear for the Gaussian particle filter and slightly less for the other filters. This shows the potential of using multi-core for estimation algorithms. The curve for the Gaussian particle filter could probably be pushed upwards by re-implementing the algorithm to not place particles in RAM but only in the cache of each processor core.

The speedup of the locally distributed particle filter could be pushed closer to linear speedup by lowering particle exchange as much as possible, being careful not to raise the probability for the weight starvation too much. Also the shared memory could be used differently to avoid false sharing, which could not happen in reality since the filter should not be run using shared memory. The filter does however perform very well here, both in terms of scaling and accuracy.

The globally distributed particle filter probably suffers from needing to place all particles in RAM. It has the least good speedup, which could depend on it being a very simple method. It is also easy to implement but not able to use as much processing power as the other filtering methods.

13.3.1 Weight distribution in locally distributed particle filter

The locally distributed particle filter used different particles on each calculation node. All operations on the global particle set were done in the local bins of particles. If the particles of one of these bins would be off by too much the weights would tend to zero, but calculations would still be performed costing as much CPU power as with high weights. To stop the effects weight starvation from happening and keep the filter together, a portion of each local bin’s particles were exchanged with other bins at each iteration. The higher the ratio of exchange the lower the risk of weight starvation. Since this filter was not designed for use in a shared memory environment and communication bandwidth was an important constraint, keeping the exchange ratio as low as possible was important.

In figure 33, the results of running the locally distributed particle filter with different combinations of the exchange rate and number of used cores is shown. Each combination of 1 to 12 threads and exchange ratio of 0% to 95% (in steps of 5%) are tested 50 times using 2000 particles. The error measure is the ratio of time steps the globally normalized weight of a thread is below 20% of the threads ideal weight $\left(\frac{1}{N_{\text{threads}}}\right)$.

Experiments showed that only a small portion of the particle needed to be exchanged to avoid weight starvation. In the scenarios tested here, only a 5% exchange ratio was needed. A lower ratio than 5% might give the same results but lower values was not tested.

Figure 32: Speed-up of the external particle filters. Note the dashed line representing linear speedup. The downturn at 12 cores on all filter are probably because of false sharing (i.e. invalidated cache due to two cores writing to the same data). (section 13.3)
14 Discussion

14.1 Conclusions

The platform has proven itself to be reliable and accurate enough for solving the bearing-only tracking problem given in the specifications for this master thesis. It can track both the sensor nodes and the non-cooperating target along with changing the positions of the sensor nodes to achieve better accuracy. As shown in part II, the platform is very flexible and re-usable. Writing new parts for any control loop component is fairly simple and detailed knowledge of the underlying system is not needed.

The operating system leJOS is very stable for being in the beta stage of development, but in beta none the less. leJOS is a very active open source project and work is being done continuously to improve the code base and it is definitely stable enough.

The implementation described above is a complete system in contrast to the most projects realised using the LEGO Mindstorms. The usual approach is to implement the minimal number of components needed to solve the given problem leaving flexibility for a speedy implementation. Here the system is capable of solving many types of control and signal processing problems with little modification. If routing in wireless sensor network are to be tested, the network component should be replaced. When other components are kept as they are, the platform will still solve the same bearings-only tracking problem. If more sensors are fitted on the robots, the sensor controller object can be attached to the same internal buffering structure as the camera controller and data will reach the server with little or no modification. In all, the implemented system is a complete robotics base system implemented on top of the Java operating system. It can be used as it is or as a framework for future projects. By exchanging the relevant plug-in components, research in control and signal processing can be performed with the advantage of using real data to investigate different problems. The platform is also ideal for being used in education since little or no knowledge is needed about the underlying system. When implementing a solution, the student would not have to re-invent the wheel but rather could focus on the specific problem given.

14.2 Problems during implementation

A major problem during the implementation was the quality of the network links. At one time during a session, latency reached 21 seconds and stayed there. Without packet loss 21 seconds of network data must have been in hiding somewhere along the communication line. After giving this a weeks time, the latency and bandwidth could be improved considerably (as shown in section 5.4.3). Still a latency of many hundreds of milliseconds needs to be investigated more closely. The speeds at which both the sensor nodes and the mobile target was working could have been lowered to simulate more real time per packet sent, but since the platform finally worked this fall-back was not used.

Using web cameras with a low resolution were cheap, but for a reason. The accuracy of the camera was very low and often reported wrong colours of pixels. Since the landmarks were identified on colour, this kind of error turned out to generate a lot of headache. Adding the classifier, as described in section 9.2, solved this somewhat
but also had its cost since each sensor node needed to have an estimate of its position with a reasonable error. If a sensor node would drift too much, the classifier would start to remove too much information on landmarks and the estimation of position would be hard to do. Knowing on the server side when the estimation was off or if the robot simply did not see some landmarks was very hard. It would be interesting to classify the landmarks on a per particle basis, especially when having bearings to many landmarks. Such a method would probably give better results since more information could be used. However, it would also need more bandwidth since the aggressive classifier could not be used.

To mitigate the problem with sensor inaccuracies and low bandwidth in the bluetooth link, some strategies were tested as described in section 8.1. These strategies worked very well but had their own problems. Aggregating the data and just using one time stamp for multiple data posts from the sensors lowers temporal resolution considerably. Using multiple sensor readings from the camera and averaging them decreases noise (though not bias) of the reading but might not be desirable. Although some of the sensor information is lost, it was not a problem since it would not be used anyway upstream because of how the phase 1 filters were written.

14.3 Filtering and real data
The filters performed as expected, however the model initially used was a bit too idealised for real implementation. Early in the implementation the filters did not take constraints to the range of the state variables into account. As an effect of this, the filters could start to give estimates that were very far of from any possible position if sensor data quality dipped. Some safe guards were implemented to account for situations where weights would go to zero or constraints were violated. This improved performance significantly as expected.

The testing of the locally distributed particle filter show that weight starvation is avoided effectively at a low rate of 5% particle exchange. The risk also increased with the number of threads. This is probably because the size of the local particle bin is lowered with more threads, increasing the chance of few particles containing important information.

The importance of real data was indeed shown in experiments. The noise in sensor readings was not necessarily Gaussian, even though it did work as an approximation. Extreme noise is hard to compensate for, even though the Gaussian model should account for it. The filters risked weight starvation leading to an estimate that were completely off, if extreme noise was not filtered out before reaching the particle filters on the server. A surprising error was that if one sensor node reported too many bearings before any other sensor node did, the estimate could move away fast from the sensor node giving the data. After some investigation this was not so surprising, since the sensor only gave bearings and the estimated probability distribution is then not constrained in the direction of the sensors line of sight.

There are many parts of the filtering that cannot be parallelized and hence the linear speedup is not attainable. The speedup was however satisfactory and confirm earlier findings. Improvements are probably possible since the filter code was very general and not optimized for the current platform, more that using the multi-core capabilities.

14.4 Future work
The platform can be improved in a number of ways. The obvious future work however is to start implementing and testing different filtering and regulator techniques. Some interesting projects would be:

- Testing different routing protocols for distributed networking (i.e. remove dependence on the server).
- Cooperatively mapping the sensors environment, perhaps using range-finders such as lasers, then solving a tracking problem in a mapped area with occlusion.
- Implementing a larger model of a factory process with many control loops and hierarchical controller structure, as used in the processing industry.
- Changing robot design to allow for more advanced movement like smooth turning, height differences in the arena or just faster movement. Perhaps racing with different designs to find the fastest and most reliable design.
- Building a set of robotic arms, as used in an automotive industry, for testing or teaching more advanced controller and estimation algorithms on multi-core architecture. A complete arm needs more than one NXT and if more than one arm were to be built the controller problem would be very interesting.

To get better reference data (independent estimations of positions) for accuracy tests, an independent source if position data would be needed. A camera in the ceiling tracking each robots would solve this problem and would not be technically too hard to implement. Since the image background would be constant, it could be filtered out (e.g. using median filtering on each pixel over time). When this is known, some tracking technique
(a particle filter perhaps) could be used to estimate the position of each robot. Using a camera with high resolution and known geometric aberrations would give a very high accuracy.

To improve on the estimation of the positions of the sensor nodes, some strategies would be interesting to look at. Having better calibrated motors would probably improve movement a lot. Adding an accelerometer would give additional information on movement and errors such as motor drift could be compensated for. Accelerometers for the Mindstorms platform have been shown to be accurate enough to be the only source of movement data and still give a very good estimate. This kind of sensor would have to be placed on the rotational axis of the robot to avoid complicated models in the tracking. To improve accuracy in turning, a compass might be used. Even though a compass might be slow to find the earth magnetic fields direction and many sources for errors can often be found where computers are involved, it would still be very useful if the robot only turned around its axis. Assuming the magnetic field at a point is constant the compass is not really dependent on the earth’s magnetic field to deliver useful information on the angle turned. Using information from both compass and accelerometer to achieve online calibration of the motors would also increase accuracy in the moving of the node. More relevant sensors can only give better a estimate of what has happened to the position, not make the movement better in the first place. Online calibration could also compensate for LEGO parts separating during a data collection session, which currently can not be compensated or calibrated away.

The motors controlling the cameras needs higher accuracy than 1 degree but also needs to be fast. With better tachometers this would be possible, and the gearing that is used to improve on accuracy might be removed. Improving the camera controller to move fast between know positions of tracked objects could also increase the number of bearings, and thereby accuracy, by not looking for data where there is none. Some kind of feedback from the preprocessing filter to the camera controller would be needed to achieve this.

Currently the bandwidth only allows for only the absolutely necessary data to be transmitted. If a higher temporal resolution or more status information is desired, it would be hard to get it through the network pipeline. It is very likely that improving the bandwidth is possible since the bluetooth specifications state that there should be a lot more available. To achieve this, a rewrite the code that send and receives the data would be needed along with studying the network layer in leJOS more closely. It seemed however that the bandwidth limit was not global but on a per sensor node basis. This suggests that the number of nodes could be increased without reduced bandwidth. The latency would increase when adding more nodes though.

The filters could only estimate a uni-modal probability distribution. Extending the platform to be able to track many non-cooperating objects would be interesting. If the network layer could be improved and some sensors added it would even be possible to make the robots map the arena and introduce advanced occlusion problems.

References


