Embedded control systems 2010
C.A.T.S.: Cooperative Autonomous Tracking System

Fredrik Wahlberg, Christian Ålander, Nils Törnblom, Martynas Mickevičius, Edvard Zak, Cheewin Pisanupoj

C.A.T.S.: Cooperative Autonomous Tracking System

Fredrik Wahlberg, Christian Ålander, Nils Törnblom, Martynas Mickevičius, Edvard Zak, Cheewin Pisanupoj

***Abstract*—This report describes the work of implementing a team of robots which cooperatively tracks a non-cooperative target using bearings-only measurements. Robots collect data using low resolution cameras and odometers mounted inside the wheel motors. The collected data is then processed and used for robot positioning and target tracking.**

We have chosen to implement two target tracking filters – an Unscented Kalman Filter as well as a particle filter. Both filters were developed side by side to compare the implementation of the differing mathematical models in the embedded device.

Tracking can only be done successfully if robots know their own position. With a known starting position, the current position of the robot can be calculated using odometer readings. Unfortunately, the errors in those readings will be accumulated over time, resulting in a position error steadily increasing over time. This problem was solved by using a set of landmarks standing at known positions. By making bearings measurements to the landmarks, the accumulated error in the position of the robots can be corrected.

***Index Terms*—Bearings-only Tracking, Particle Filter, Unscented Kalman Filter, Robot Platform, Embedded platform**

I. Introduction

With this work we present two ways of autonomous target tracking for a group of mobile cooperating robots using on-line bearings-only measurements. The target is non-cooperative and moving within a bounded environment with potentially occluded areas. Cooperation between the robots is achieved by communication over a wireless network and data storage on a PC working as a base station. Our solution implements two different tracking filters; the Unscented Kalman Filter (UKF) and a distributed particle filter. All calculations are made on the chosen mobile Lego® Mindstorms® NXT robots, which have somewhat limited computational resources. It is important to note that no calculations relating to the actual positioning are made on the PC as it was used for data sharing and visualization only. The bearings only measurements are acquired using a set of video cameras connected to the robots. In addition, the motors used for propulsion and camera rotation also have built in odometers. The desired positions of the cats are determined using the guidance algorithm (guide). Proof of concepts of the suggested solutions were simulated in Matlab before the Lego implementation was started. The robots are hereafter called cats and the target they are tracking is called the mouse.

Following this introduction, an overview of some previous work relating to particle and unscented Kalman filters is given in Section 2. In the following Methodology section, a brief description is given of the mathematical theory behind the filters as well as the guide. We also present some results from the Matlab simulations here. Section 4 describes our implementation, including the overall software architecture, physical design of the cats and the landmarks, control algorithms for the movements of the cats, network architecture, Java implementation of the filters and the guide, the graphical user interface on the PC, as well as the Java simulation framework. The results are given in Section 5, showing benchmarks of the different subsystems. A discussion of the results are given in Section 6, as well as some suggestions for future work and what can be done to improve the performance of the system.

II. Previous Work

Since its published formulation in 1960, the Kalman filter has been a widely used method for estimating the true states of a linear dynamical system from a series of noisy measurements. For linear systems with a Gaussian noise distribution, the Kalman filter is the optimal method to compute the state estimates. Non-linear state models cannot be used directly in the standard Kalman filter. One way to circumvent this is to linearize the model, which is done in the Extended Kalman filter (EKF). The EKF has long been the de facto standard in nonlinear state estimation [1]. The approximation done in the linearization can give significant errors for highly non-linear systems though. An improvement of the EKF is the Unscented Kalman filter (UKF), introduced in 1997 by Simon J. Julier and Jerrey K. Uhlmann [2]. It is based on the unscented transform, which works by propagating a minimal set of sampling points through the non-linear functions. From this, the estimated mean and covariance of the state variables are then recovered. Just like in the EKF, the UKF uses approximations of the non-linear state model, but the errors that arise from them are smaller [3]. Where the EKF captures the mean and the covariance accurately to the first order (Taylor series expansion), the approximations of the UKF are accurate to the third order (assuming Gaussian random variable) [4]. Moreover, the EKF requires the derivation of the Jacobian matrices which can be difficult to find. The UKF does not use the Jacobian [2]. It has also been shown that the UKF consistently achieves a better level of accuracy than the EKF at a comparable level of complexity [4].

Particle filters are a common approach in non-linear tracking and have been proven effective for robot localization [5], [6], [7]. It is a model estimation technique based on a Monte Carlo approach. With a sufficient number of samples (particles) the particle filters approach the Bayesian optimal estimate, so they can be made more accurate than Kalman filter variants [8]. It is a computationally heavy technique and hence there would

have to be a lot of work spent optimizing the code. The target platform was quite limited in computational power, but since we also would try the Kalman filter, which is a standard technique, it was decided that there would be enough time to test both.

III. Methodology

A. Alternative solutions

1) Absolute positioning with electromagnets: An initial idea of robot positioning was to use two electromagnets generating two separate magnetic fields, strong enough to cover a large enough area of interest (maybe 2x2 m). By shifting these fields on and off, we were hoping to extract field-vectors of these fields with a magnetic sensor (compass), and triangulate the position of the robot by calculating the direction of the magnetic source. This ambitious concept proved hard to realize for numerous reasons, and would probably just on its own have been a much larger project than what we were about to execute. To cover the area required with a sufficient field, we first looked on the market for purchasing an already existing electromagnet. However, we soon realized that the strength of the fields they generated was not adequate for our needs. To get enough strength we had to build our own electromagnets. By letting an iron core be equipped with some hundred laps of wire, and letting a strong current flow through it, we figured that a large enough magnetic field for our needs could be obtained. To get a position from the field vectors which we hoped to be able to get from the magnetic sensors, available in the LEGO Mindstorms kit, we needed to switch the two magnetic fields on and off separately. Otherwise these field vectors would have merged into a single one, with a complicated structure. To get a good measurement of the generated field vector we wanted to have them at least twice as large as Earths magnetic field, over the whole arena. This due to enabling us to distinguish it at all. After running some calculations for our hypothetical electromagnets, we were forced to abandon this sub-project, since they showed that the size of the field the electromagnets were required to generate far exceeded what reasonably could have been achieved with available equipment. Another reason to reject this idea was the magnetic sensors from LEGO that did not give good enough measurements to satisfy our ambitions for this application. A bad sensor with fluctuating readings would easily dismantle the purpose of the system, by giving the sensitive system a high rate of wrong values, and make the calculations for the estimated position disastrous.

2) Color coding: Another suggestion to the problem of the cats knowing their own correct position and orientation was using a color coded grid on the floor of the arena. Using an available color sensor the idea was to detect whenever the cat traveled over a colored strip. With four colors printed on a white paper it should be possible to detect in which direction the cat is moving and how far it has gone. By having both horizontal and vertical lines with alternating colors, one can detect the x- and y-component of the cat's velocity. However it turned out that the color sensor sampled its reading from quite a large area, which meant that we had to use rather wide colored stripes for them to be detected correctly. This presented the problem with the intersections of the horizontal and the vertical lines. When the color sensor crossed over such an intersection, a mix of both colors was detected instead of each one separately. This meant that neither the x- or y-component of movement was updated, resulting in an erroneous position estimate. A solution to this could be to limit the cat to move only in four directions, parallel and perpendicular to the lines. That would however require that the cat could correct its orientation from time to time if it got off track.

Another problem was that the color sensor was sensitive to the ambient lighting. This could be accounted for in the software to some extent with some clever programming, but not completely. These limitations made us abandon the color coding altogether after some time, even though we had put quite a bit of work in to it. Instead we chose to use the camera and landmarks to locate the cats. The color codes do however have the nice property of providing both position and orientation very often, without having to look for any landmarks.

B. Kalman filter theory

1) The Kalman filter: The Kalman filter is a method of estimating state variables from noisy measurements. With a provided state model of the system, a predicted state variable estimate of one time step ahead is computed. The uncertainty for this prediction is also calculated using the previous estimate. A weighted average of the actual measurement and the predicted value is then used as the output of the filter. The weights are chosen so that higher uncertainties get smaller weights and vice versa. The Kalman filter models the errors in the measurements and state estimates as Gaussian measurement noise and process noise respectively, with fixed standard deviations. These two standard deviations are user selectable parameters, which should be chosen so that the filter achieves the best performance. A lower measurement noise standard deviation means that the measurements are trusted more and given a bigger weight, which should be the case if the measurements are accurate. The process noise standard deviation works similarly, representing how much the state model should be trusted.

Given the low computational complexity of the nonlinear Kalman filters and the target platform's limited processing speed, we decided that it would be a interesting alternative to the particle filter to try out. The UKF was chosen over the EKF because of its advantage both regarding accuracy and ease of implementation. The full algorithm of the UKF will not be presented here, since our project used a pre-written Matlab implementation [9], later converted to Java. However, an important note is that the square root of the covariance matrix is required in a step of the algorithm. The square root of the matrix B is the matrix A satisfying $B = AA^T$. This can be computed in different ways, such as using the Cholesky decomposition, which requires that the matrix B is both symmetric and positive-definite.

2) State model formulation:

(a) *Tracking filter*

Using the UKF made it possible to use the true non-linear state measurement equation given by

$$z_k = \begin{pmatrix} \theta_1 \\ \theta_2 \\ \theta_3 \end{pmatrix}_k = \begin{pmatrix} h_1(x_k) \\ h_2(x_k) \\ h_2(x_k) \end{pmatrix} + n_k,$$

where θ_i is the absolute measured angle, relative to the arena reference frame, from cat *i.* Subscript *k* henceforth indicate the time step. The states x_k are chosen as

$$x_k = \begin{pmatrix} x \\ y \\ v_x \\ v_y \end{pmatrix}_k$$

where x and y are the position of the mouse and v_x and v_y are its velocities. $h_i(x_k)$ is the non-linear measurement equation for cat *i* given by

$$h_i(x_k) = arctan(\frac{y - y_i}{x - x_i})$$

where y_i and x_i are the position of cat *i*. n_k is the measurement noise with covariance matrix

$$R = r^2 \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

where *r* is the standard deviation of the measurement noise. It is assumed that the noise of the measurements is uncorrelated and has equal power.

The movement of the mouse is modeled with constant velocity. This means that the acceleration of the mouse is modeled as process noise. The process model, or update formula, is

$x_k = Fx_{k-1} + Gw_k$

where w_k is the process noise. Using Newton's laws of motion we easily find that

$$F = \begin{pmatrix} 1 & 0 & T & 0 \\ 0 & 1 & 0 & T \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}, \; G = \begin{pmatrix} \frac{T^2}{2} \\ \frac{T^2}{2} \\ T \\ T \end{pmatrix}$$

where *T* is the sampling time period. The process noise covariance matrix *Q* becomes

$$Q = E[Gw_k(Gw_k)^T] = q^2 \begin{pmatrix} \frac{T}{4}^4 & 0 & \frac{T^3}{2} & 0 \\ 0 & \frac{T}{4}^4 & 0 & \frac{T}{2}^3 \\ \frac{T}{2}^3 & 0 & T^2 & 0 \\ 0 & \frac{T}{2}^3 & 0 & T^2 \end{pmatrix}$$

where *q* is the standard deviation of the process noise.

(b) *Absolute positioning filter*

The basic idea of the absolute positioning filter was that the problem of locating the mouse with several cat angle measurements is very similar to the problem of locating one cat using several landmark angle measurements. The state vector x_k contains the cat position and velocity, while the measurement vector z_k is comprised of the measurements to the different landmarks. There are however some important differences between the two problems. First, with our chosen landmark positions, only at most one landmark would be seen at any given time. Therefore no classical triangulation can be done. The Matlab simulation showed that a filter implemented this way still could correct the position of the cat quite well with only one landmark measurement at a time. More importantly, the absolute positioning filter should also estimate the *orientation* of the cat as it is needed in order to calculate the absolute angular measurements to the mouse from relative measurements, given in the cat's coordinate system. Specifically, the absolute measurement from the cat to the mouse is given by

$$\theta = \theta_{orient} + \theta_{rel}$$

where θ_{orient} is the orientation of the cat and θ_{rel} is the relative measurement to the mouse. The subscript *i* is dropped in the absolute positioning filter since it only considers one cat. The formula above means that errors in the cat's orientation will translate into an erroneous estimation of the mouse's position.

In contrast to the tracking filter, we have other sensors than the camera available, namely the motor odometers. The data from those can be used to calculate the cat's traveled distance and rotation. However, it was quickly realized that only relying on odometer data would give a quite bad estimation of the cat's position after a while, as the measurement errors accumulate. Therefore, the landmark measurements should be used as a way of correcting the cat's position when necessary.

Incorporating velocity measurements in the measurement vector made it possible to use odometer data. The measurement vector now takes the form

$$z_k = \begin{pmatrix} \theta_1 \\ \theta_2 \\ \theta_3 \\ \theta_4 \\ v_x \\ v_y \end{pmatrix}$$

where θ_i is the absolute angular measurement to landmark *i.* v_x and v_y are the x- and y-components of the cat's velocity. This solution was implemented successfully in Matlab. A more appropriate solution would be to use the velocity measurements in polar form, given as speed and orientation (or its derivative) instead of the x and y component of the velocity. That way a higher uncertainty can be assigned to the orientation and a lower to the speed, more closely mimicking the true dynamics of the system. This formulation however yielded undesirable results in the Matlab simulation and was therefore not followed out.

3) Simulation in Matlab: The UKF filter also has a few user selectable parameters (α, β, κ) that were varied to see their effect on performance. The performance of the filter turned out to be almost the same no matter how the parameters were chosen. There was however a valid range for some of the parameters in which the covariance matrix remained positive definite. If the parameters were chosen badly the Cholesky factorization failed.

(a) *Tracking filter*

The measurements in our implementation are intermittent, i.e. the cats cannot provide measurements all the time, and sometimes only a few cats might have measurements available. This has to be handled somehow as the standard formulation of the Kalman filter assumes that the measurements are available all the time. A simple way to take care of this is to set the elements in the measurement covariance matrix corresponding to the unavailable measurements to ∞. That way the measurement is given zero credibility and is weighted away. That is if measurement i is unavailable then set $R_{ii} = \infty$ instead of r^2. The measurement $z_{i,k}$ can then be set to anything. R_{ii} can be set to *inf* in Matlab but since this is not possible in the brick implementation we instead set it a very large number (10^{10}), about as large as double point arithmetic allows. In the Matlab simulation it was verified that this was large enough for the unavailable measurements (and therefore incorrect) to have no actual effect on the state estimates. Some further investigation showed that it is possible to modify the Kalman filter algorithm to only do the update step for the unavailable measurements. This would be a computationally more efficient solution and certainly more elegant. It was however concluded that it was simpler just to keep the previous solution.

It also turned out that the formulation of the measurement function had some problems with the discontinuous nature angles. Only angles in the range $0 - 2\pi$ were used. If a cat was providing angles very close to 0 or 2π the position of the mouse diverged until the cat reported other angles. This seemed to get worse of the reported angles were fluctuating more, so the problem was probably that the reported angles were fluctuating between 0 and 2π. Because this only seemed to happen at a very limited angle, and therefore not very often, the formulation of the measurement function was kept to the Java simulation and the brick implementation.

The performance of the UKF tracking filter is shown in Figure 1 and Figure 2. The cats are shown as black asterisks and their angular readings in the last time step as black lines. The true position of the mouse is shown as a red line, the estimated as a blue line. In both simulations three cats are observing the mouse at all times. The angular measurements to the mouse have an additive Gaussian measurement noise with a standard deviation of 2 degrees. In Figure 1, the mouse is moving with a piecewise constant velocity, abruptly changing direction from one time step to another. We can see that the filter estimates converges to a value quite close to the true position when the mouse is moving straight with constant velocity. When the mouse changes direction, the acceleration is no longer zero as modeled, but very large for a short time. With this chosen process noise standard deviation this leads to an inertial movement in the position estimates.

Figure 2 shows a situation with a constant acceleration. As expected the filter estimate can't converge as close as in the linear case. For comparison, static least squares estimates of the mouse's position are shown as green dots. Clearly, the UKF filter performs better than the latter, albeit the constant process disturbance.

(b) *Absolute positioning filter*

The previously mentioned form of the filter was implemented in Matlab. That implementation used absolute angle

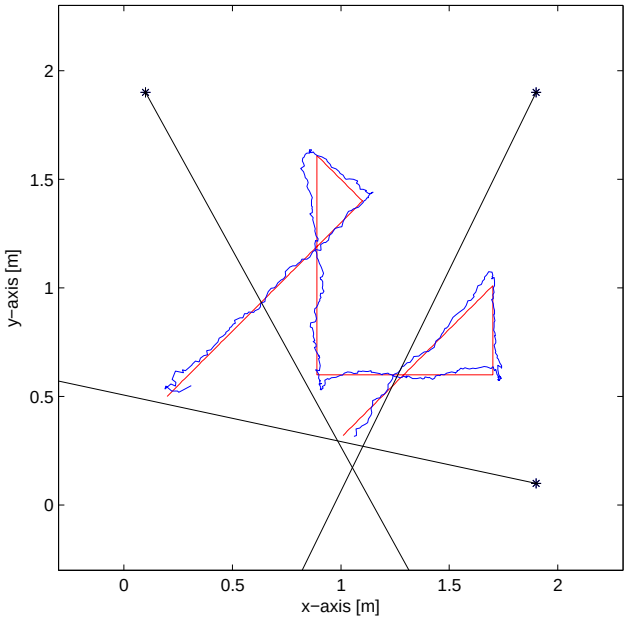

Figure 1. UKF Tracking filter simulation. Mouse is moving in a piecewise linear track.

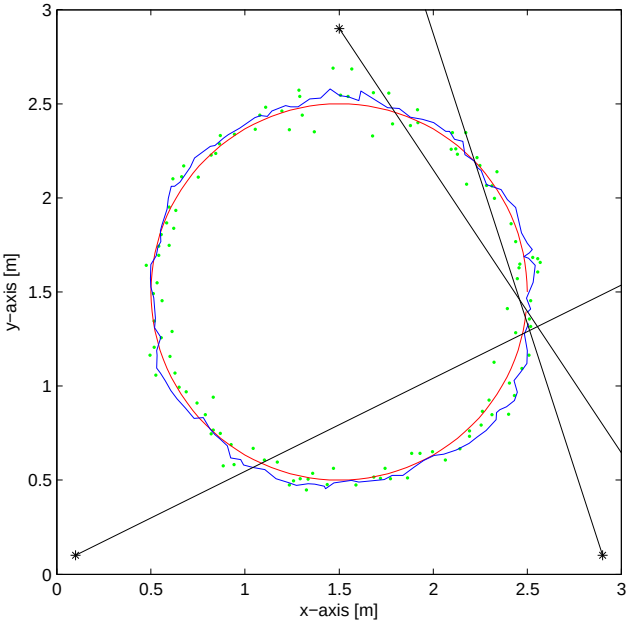

Figure 2. UKF Tracking filter simulation. Mouse is moving in a circular track.

measurements everywhere and could therefore not simulate static errors in the cat's orientation. The absolute positioning filter was also implemented together with the tracking filter in order to get an idea of how the errors in the cat's position would translate into errors in the mouse's position.

C. Particle filter

The basic principle of a particle filter is to by a particle optimization technique filter noisy data. It is a Bayesian recursive method which means it recursively tries to estimate a state vector by a probability distribution using data to evaluate

and update the same distribution. It has been proven to be very useful on bearings-only problems localizing robots [7]. The description of the system is described in equation 1 and 2.

$$x_k = f(x_{k-1}) + w_k \tag{1}$$

$$y_k = h(x_k) + v_k \tag{2}$$

If the function $f(\cdot)$ and $h(\cdot)$ are linear functions that could be seen as a matrix multiplication with respective x_k (representing the state vector) and w and v Gaussian distributed errors, the system would be the same used in the Kalman filter. However in many applications $h(\cdot)$ are non-linear and v are not Gaussian. x_k is the state vector estimate at time k. Around this vector there lies a probability distribution corresponding to the uncertainty of the state vector. The main advantages to particle filters is that they can handle non-linearity's without compromising on accuracy or the need for linearization.

Basically the algorithm looks like this:

Draw random hypotheses about the states in the tracked object within a given probability density function, called and visualized as particles. This first step is only for initialization.

Update particles (integration or similar) according to the estimated objects dynamics (equation 1). Note that different states can have non-linear relations.

Compare the states of each particle with sensor data (equation 1) and give each particle a weight (w_i) indicating how well it conforms to it (equation 3 and 4). θ_{error} is the error between the sensor data θ_{sensor} and the expected angle (i.e. the angle to the particle $(x_{particle}, y_{particle})$ from the sensor given by (x_{sensor}, y_{sensor})).

$$\theta_{error} = \theta_{sensor} - tan^{-1}\left(\frac{y_{particle} - y_{sensor}}{x_{particle} - x_{sensor}}\right) \tag{3}$$

$$w_i = w_i \cdot e^{\frac{\theta^2_{error}}{2\sigma}} \tag{4}$$

Calculate a weighted mean or similar method to get the estimated states of the tracked object (equation 5-7) which is also the parameterization of the probability density. u_i^k signifies the k:th state in the state variable u of the i:th particle.

$$\sum_{i=0}^{N} w_i = 1 \tag{5}$$

$$\mu = \sum_{i=0}^{N} w \cdot u_i \tag{6}$$

$$C_{jk} = \sum_{i=0}^{N} \frac{(u_i^j - \mu^j)(u_i^k - \mu^k)}{1 - \sum_{i=0}^{N} w_i^2} \tag{7}$$

If needed, re-sample the particles based on the estimation of the tracked object (i.e. change the estimated probability density contained in the particle swarm).

The above described steps are repeated with information from the re-sampling step being the input of the update step, hence the name recursive algorithm.

In our specific case there was a need for two particle filters. One type for tracking the target and one type for positioning of the tracking robots. Also since we did not have to take height of a target into account (i.e. the arena for tracking was flat), all filters needed only to take the two dimensional case into account.

The tracking filter needed to estimate the position of the tracked object and, if possible, its speed. A natural choice for state variables was then the position in the x and y direction and velocity in the x and y direction (state vector is shown in equation 8).

$$x_k = \begin{pmatrix} r_x \\ r_y \\ v_x \\ v_y \end{pmatrix} \tag{8}$$

For the positioning particle filter the states were chosen as x, y and heading angle (state vector is shown in equation 9). The camera, mounted on top of the robot, would sweep for targets, giving angles in a local reference system. These sightings could then be compared to the known positions of landmarks and how well a particles states would conform to this data.

$$x_k = \begin{pmatrix} r_x \\ r_y \\ \theta \end{pmatrix} \tag{9}$$

A Gaussian noise model was chosen, hence the ease of parameterizing the distribution in equations 6 and 7. A reason for choosing this model is that the data communication rate should be as low as possible when doing the calculations distributed over the cats. A fairly naive approach was used to do a weighted mean and covariance matrix when merging the information from the different cats. The mean is quite simple as it is obvious that the means can just be weighted since all nodes had the same starting weight. The covariance matrix was also just weighted together in a linear way. This turned out to work but we did not do a formal proof.

1) Simulation: The tracking filter worked very well in simulation and did even estimate the speed quite correctly. The simulation was done in a 3x3 meter arena with 1 to 5 degrees of standard deviation of the angular error of the bearing.

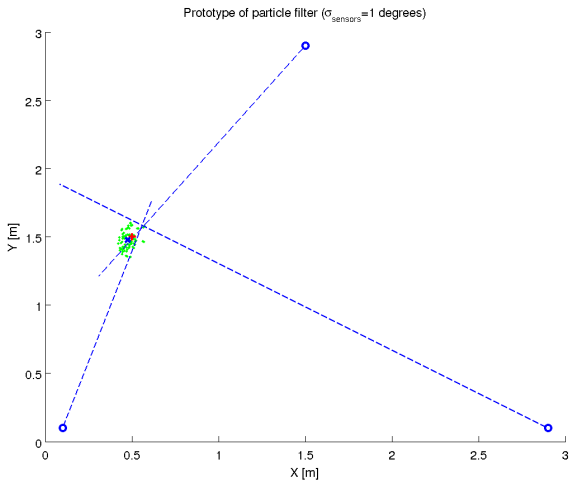

Figure 3. Plot from the simulation of the tracking particle filter

The positioning filter worked very well in simulation even though it, at most, could only see two landmarks at a time. The simulation was done in a 3x3 meter arena with 1 to 5 degrees of standard deviation of the angular error in the bearing.

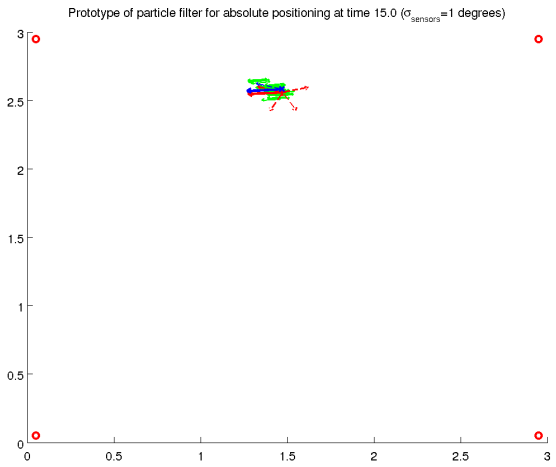

Figure 4. Plot from the simulation of the positioning particle filter

The good results from the simulation was a good indication that this filtering technique was a feasible solution. It was clear though that some optimization would be needed for the filtering technique to work on our desired platform. Also the code in the Matlab implementation was at first structured in a way that was very complex to do in Java. This came naturally since there was a great need to experiment. The last step in the implementation of the Matlab code was to change it so to use only statements which had equivalents in Java.

D. Geometric filter

1) Theory: The need for a naive way of correcting the cat position arouse when implementation of the other approaches to positioning failed. The geometric filter is not really a filter in the ordinary sense. The name is inherited from the class it replaced in the code structure, the positioning Kalman/particle filter. It can not correct much and it is not stable, but it worked good enough during tests to use at the demonstration.

After failing to utilize more modern approaches the basic question which gave rise to the geometric filter was "what would Archimedes have done?".

If the angles between two landmarks are known, the inscribed angle theorem[10] gives that the double angle is the angles between the landmarks when standing in the center of a circle (shown in figure 5). The positions of the landmarks are known, hence the solution to where the cat is lies on a circle. If one more landmark adjacent to the first (which it always is in our case since we use 4 landmarks) another circle can be found. There two circles intersect in two places and one of the solutions is the position of the cat.

A problem with this approach is that it does not filter out any noise from the angle measurements. If there is noise the most probable solution stops being a point and becomes an area, as shown in figure 6. The more noise the more this area

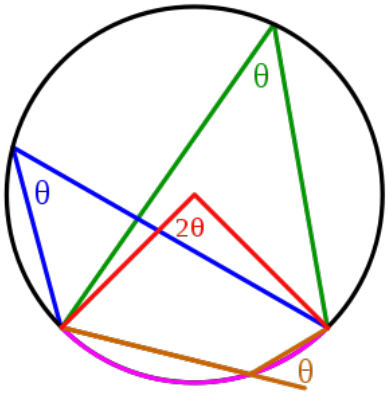

Figure 5. Plot of a circle showing the angles given by the inscribed angle theorem

grows. This could give rise to considerable drift in the filter estimate but would still be better than the accumulated error from inexact motor information.

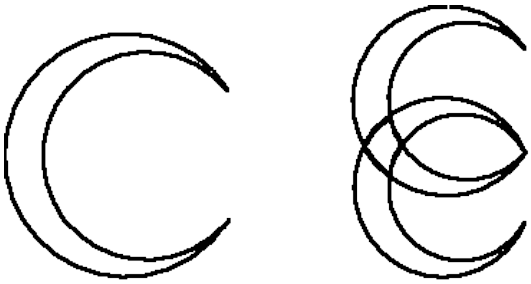

Figure 6. Plot of the probable positions of a cat using noisy measurements to two landmarks (left) to three landmarks giving an overlap of the probable areas (right)

E. Guide

When having mobile robots as in our case the need for positioning them arouse. This was not a part of the given assignment and the used method was more or less developed on a napkin and should be considered as a proof-of-concept.

1) Theory : The basic idea of the guide is that if the rules to follow, like "keep distance from others", can be formulated mathematically then an optimization technique could be used for positioning.

The rules where formulated as functions who could generate a map of where a good place for the cat would be. This method could be compared with letting the functions generate a landscape and telling the cat to try to keep in the valleys. An example of the map of the final function is shown in figure 7. To try to find the maximum (the most desired location) of the map a gradient descent method[11] was used. Such a method is easy to implement and not so expensive to compute. The problem with this kind of method is that it only finds a *local* maximum. The decision was made to not expect the guide to find the optimal position (with the given criterion function) but only to try to correct the position to try to avoid situations

which would make tracking too hard. The method of gradient descent tries to improve an initial value iteratively as shown in equation 10.

$$r_{n+1} = r_n - \nabla f(x, y) \tag{10}$$

Where the derivative is approximated with finite difference scheme shown in equation 11.

$$\nabla f(x, y) \approx \left(\frac{f(x+h,y) - f(x-h,y)}{2h}, \frac{f(x,y+h) - f(x,y-h)}{2h} \right) \tag{11}$$

The criterion function is composed of the multiplication of four functions which each represents a separate "rule" to follow. Those where the following:

1) *Keep an optimal distance to the estimated target position.* The penalty is the distance squared from the optimal distance to the target.
2) *Keep distance from edges.* Where the distance to the edge is penalized linearly within a set minimal distance to an edge.
3) *Keep distance from other mobile objects.* Where the distance to the other object is penalized linearly within a set minimal distance to that object.
4) *Keep distance from another sensors line of sight to the target.* To avoid that cats block each others sight or stands opposite each other (which gives no additional information than having only one observer) the distance to the line of sight is penalized if too small.

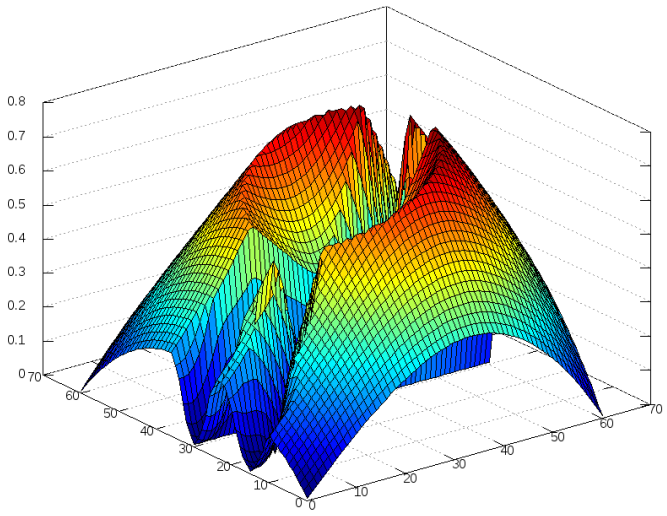

Figure 7. Plot of the combined criterion function used by the guide

2) Simulation: A problem with the gradient descent method is that it in some circumstances starts to oscillate around an optimum. This was avoided by simply not correcting the position less than a couple of centimeters at a time. Convergence is often defined as when the distance between two iterations are below a certain threshold. When oscillating a limit to the number of iterations was used to avoid loops going on for too long. A better solution would be to do a line search[12] but it was not finished due to lack of time.

The first simulation was implemented in Matlab to test the concepts. It worked fine and was ported to Java shortly after that.

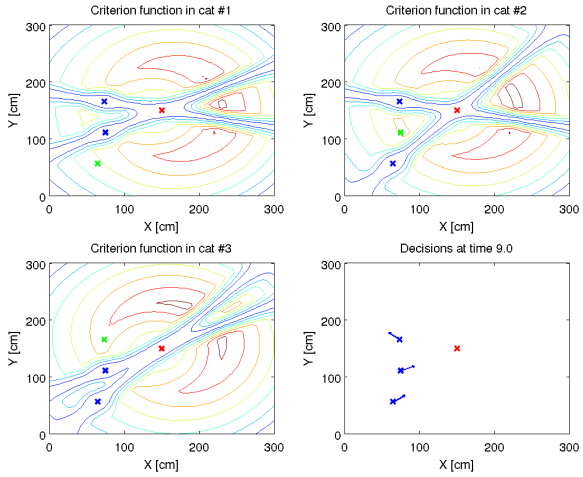

Figure 8. Plot of the Matlab simulation for the guide showing the criterion function for each cat and where they are going (lower right)

IV. IMPLEMENTATION

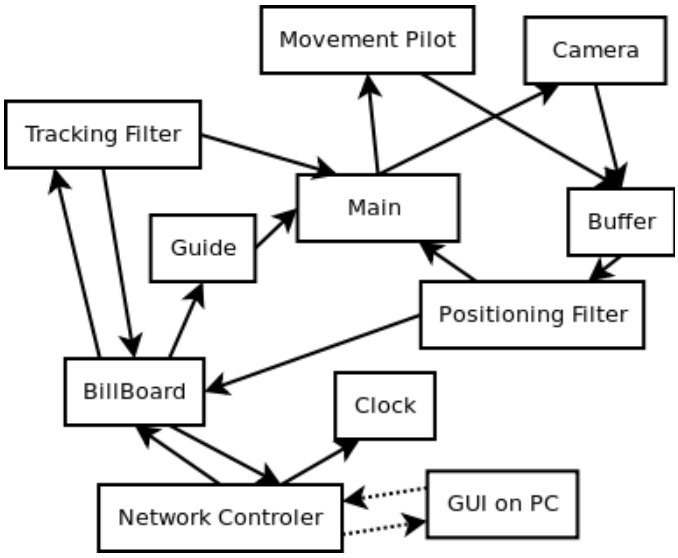

Figure 9. Overview of java software architecture.

A. Cat design

1) Locomotion: We have tried some different designs for the locomotion of the cats, and found a simple construction which was most suited for the task. At first we used two motors with attached wheels placed on both sides of the brick, and a single support wheel which we believed would help ease the rotation. However, we soon found out that the support wheel impaired accurate rotation of the cat. This is because we want to rotate the cat around and axis placed between the two propelling wheels. This is done by rotating the wheels in

opposite direction. Nevertheless, we needed a third supporting point for the cat to be stable. Ideally, this contact point should have a coefficient of friction of zero to the ground. We found a small slippery LEGO-piece to use as a contact point, which preformed better than the support wheel. The final cat design can be seen in Figure 10.

Figure 10. One of the constructed cats.

2) Camera module: As the goal of this project is to track the mouse we wanted the cameras on the cats to be looking on the mouse most of the time. In fact, apart from the landmark sweeps, the camera should be looking at the mouse all the time. The camera was first fitted directly to a motor enabling the camera to rotate and look around the arena. Tests showed that errors in the motor odometer were quite large, up to 10°, resulting in a very bad angular reading. The errors did not drift, so they could in theory be corrected by calibration. However, this was made difficult since the error was not constant but varied over the angular position of the motor. In addition to this, the quite large dead zone of the motor also added to the errors in the angular readings. The solution to this was to gear down the rotation of the camera five times with a large cog wheel. That way the errors were reduced about five times. The maximum speed of the camera rotation was of course also reduced five times, but the speed with the cog wheel showed to be sufficient. In fact, the camera's susceptibility to motion blur prevented any rotation much faster than this anyway.

B. Camera

1) Extracting angle measurements: To allow the mouse's position to be calculated from angular measurements, the angles have to be given in a common coordinate system. We chose to use the fixed coordinate system of the arena. The sensor readings only provide relative angular values however, so the angles have to be converted. This is done using:

$$\theta_{abs} = \theta_{cat} + \theta_m + \theta_{cam} \tag{12}$$

where θ_{abs} is the angle in the arena coordinate system, θ_{cat} is the angle of the cat (i.e. its orientation), θ_m is the angle of the camera motor relative to its starting position. Before program execution, the camera motor was always oriented so that the camera was facing straight forward in the same direction as that of the cat. The cats were also oriented pointing towards 0° (along the x-axis) in the arena coordinate system, every time before program execution. That way we could use the assumed initial values of $\theta_{cat} = 0°$ and $\theta_m = 0°$. θ_{cam} is the angle to the mouse (or landmark) relative to the camera's optical axis. This value has to be computed from the information available in the camera image containing the mouse. The mouse's horizontal pixel offset to the camera's optical axis can be converted to an angular reading. To investigate the relation between the pixel and the angular value, a simple test was performed. A set of markers was placed in the camera's entire field of view, with a fixed angle between each marker. Conveniently, the camera then rendered an image with a fixed horizontal pixel distance between each marker. Therefore, it was concluded that the following linear model to calculate the θ_{cam} would be used:

$$\theta_{cam} = k(e + d) \tag{13}$$

where k is a constant describing the horizontal angular span of one pixel, and e the pixel offset signal. The test showed that the camera's optical axis was not properly aligned with the center of the camera sensor, so a constant pixel offset d was added to the model. This value had to be tweaked to each of the different cameras as they all showed different pixel offsets (up to 19 pixels). The angular field of view of the camera was given in the specifications, and this value could also be verified in the test. Using this value a, k could easily be calculated as

$$k = \frac{a}{p} \tag{14}$$

where p is the number of pixels along the horizontal axis of the image.

Retrieving the horizontal pixel coordinates of the mouse marker in a camera image was easy since the leJOS API already has this feature implemented. Because the camera can detect up to eight targets with different colors, we chose to assign each landmark and the mouse with a unique color marker. That way, we can easily get the correct pixel locations of the mouse and the landmarks in the camera's field of view, even if several should appear at the same time.

2) Modified camera firmware: While observing the camera output using the NXTCamView software we have noticed that tracked object is often split into two, four or even more parts. This effect can be seen even more significantly when lighting conditions are poor. The issue could be solved by taking into consideration all tracked objects and calculating median coordinates. This would have effectively introduced some overhead to the camera regulator code. We chose to modify the camera firmware so that it does not split tracked objects or rather merges them before transmitting data to robot. We have found a modified camera firmware called "MergeBlob" [13] which does exactly that. We have also tweaked the minimum and maximum size of the tracked objects.

Since we have started to modify camera's firmware we thought that it is good idea to put some more filtering logic to the camera so we can keep camera's AVR processor busy, offloading the NXT's CPU. We have added tracking object filtering by the Y coordinate [14]. So objects which are lower than certain minimum and higher then certain maximum are not tracked. This is true since we know the height of the mouse and the height of the landmarks. This helped us to remove any unnecessary noise such as windows and other artificial light sources which could be mistakenly tracked.

3) PD tracking-regulator: We needed a regulator that would follow the movement of the mouse as good as possible to allow for fast mouse movement. Also, the regulator needed to be smart enough not to keep spinning the motor in one direction. Doing so resulted in that the camera cable got tangled up, thereby stopping the motor and hindering any further movement in that direction. The solution was implementing a lower and higher bound of the rotation of the camera. The bounds were chosen as [-180°, 180°] (relative to the starting position of the camera motor), to allow a 360° field of view. Whenever the camera motor tried to go outside this bound, the direction of the motor changed.

When designing a regulator of this kind an error signal is necessary. Since it is desirable to keep the mouse in the middle of the camera image (only horizontal position regarded), the error was chosen to be the pixel difference between the mouse's location in the image and the image center (corrected with an offset, i.e. $e + d$ above). This error signal limited in resolution by the resolution of the camera and it is bounded by the field of view of the camera. If the mouse disappears outside of the picture, the error signal is set to the same as the bound value on the side of the picture of which the mouse was last seen. Also, a control signal is necessary. The natural choice of the speed and direction of the motor, was chosen.

First a simple proportional regulator was tested. It performed quite well, but became a bit oscillatory at high speeds, i.e. at large values of the P-parameter. Then a PID-controller was implemented, which yielded smoother and better performance. The integrating part was later dropped since it did not contribute to the performance, giving the final PD-controller.

C. Landmarks

1) Construction: We encountered several difficulties when selecting a suitable visual marker for the landmarks and the mouse. Because the image processing properties of the camera is rather limited, it can only give the location of a few targets specified by a predetermined color range. The output is the size and the location of the bounding box around the detected targets in view. An ideal visual marker should have a color that is captured by the camera to have a very specific color. Furthermore, this color should be very unusual so that something else that appears in the view isn't detected as the target. First we tried one of the red balls that came with the Lego kit as a visual marker. Unfortunately, the color of the ball was detected differently depending of the lightning. The color was different if the camera saw the side of the ball in shadow compared to if it saw the side which was lit up. One could compensate for this by allowing a bigger range of red color to be considered as the target. This however gave quite many erroneous detections of the target. It was clear that a better visual marker was needed to allow acceptable tracking of the mouse.

The solution adopted involved the use of Cold Cathode Fluorescent Lamps (CCFLs) of various colors. As opposed to light emitting diodes (LEDs) they can be viewed from any direction in a horizontal plane if placed standing. LEDs on the other hand can only be viewed from one side. Furthermore, the CCFLs have a much larger luminous area than the point-like appearance of LEDs. This is necessary if the visual marker should be detectable at large distances. This criterion proved quite difficult to satisfy even with the CCFLs. The size of the lamp in the picture decreases approximately with the inverse of the distance [15]. In combination with the limited resolution of the camera, this made the CCFL no larger than a few pixels or not visible at all at distances much larger than a meter. Also, the CCFLs were quite bright compared to the ambient lighting which made CCFLs overexposed. Thus, the camera saw a white target with only a small colored halo instead of a fully colored object. As we had no direct control over the camera's exposure, this could not be corrected in it's software. The solution was to wrap the CCFL with a semi-translucent plastic sheet. Choosing the plastic to be colored the same way as the CCFL gave the best appearance to the marker. This not only made the lamp appear to be less bright, and therefore receive a correct exposure, but also made the marker larger, enabling detection at larger distances. The final four landmarks had the colors blue, green, purple and white. One of them can be seen in Figure 11. A similar red marker was affixed to the mouse. Red was chosen because it was the easiest color to detect.

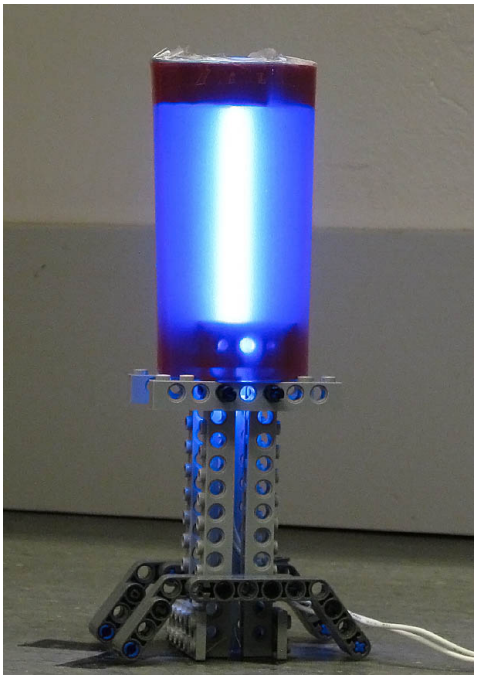

Figure 11. One of the constructed landmarks viewed under standard operating ambient lighting.

D. Cat movement regulator

It was decided that a simple movement regulator would be sufficient for the cats. Therefore, we limited the cats to travel in straight paths, with turning in between. The cats were constructed so that they could rotate on a spot, i.e. around a vertical axis between the wheels. These choices also made it easier for the absolute positioning filter.

1) Calibration: As it turned out, leJOS had a built in Pilot class, aiding the programming of the movement regulator. Given the diameter of the wheels, the traveled distance can be calculated from the motor odometer data. Furthermore, the distance between the wheels is used to calculate how much the cat has rotated. It turned out that the left and right motors weren't equally strong which caused the cat to drift off course when trying to travel straight. However the specified size of the wheels was tweaked to compensate for this. The actual size of them was the same.

2) Smooth acceleration: The provided leJOS movement pilot caused the cat to travel and rotate with a more or less constant velocity, resulting in an abrupt start and stop of each movement command. This hard acceleration caused the wheels to slip, which in turn fooled the odometer. The leJOS pilot was therefore modified to allow smooth acceleration without any wheel slip. Both the travel and rotation velocity was implemented to have an ascending and descending ramp.

E. Network

1) Data synchronization: In order to make the autonomous robot system cooperative, some kind of communication between the robots had to be introduced. Our choice was Bluetooth connection since every robot had such capability. A connection to the PC can also be achieved with a simple Bluetooth dongle. The choice of BT communication limits the distance that the robots can be apart from each other. However, the robots were operating fine with distances up to 3 meters between them.

Two different kinds of data were defined that was going to flow through the network.

1) All the data that is vital to the robots internal systems belong to the first group.
2) All the additional data that can be used by the GUI on the PC for the visualization and debugging of these systems.

The synchronization rate of the first group actively limits the rate on which the tracking filters can run. It also limits the accuracy that the guide can give us. All the remaining bandwidth can be used for data visualization and debugging. The first group can be split further into two parts:

- measurements of the mouse and robot position from the positioning filter. This is needed for the tracking filter and the guide.
- the results of the tracking filter (the position of the mouse with uncertainties). This is needed for the guide.

The synchronization of the data mentioned above was achieved using a billboard model. Every robot had a spot on the billboard. Whenever any robot changes a value on its spot, an update is sent to all of the robots. No updates are concatenated. Every update has a timestamp and there is also a timestamp stored in the billboard for every spot, so if a lost update comes after newer updates were already applied, it is not used to update the spot with the old value. What is more, a read from the billboard is never blocked. Therefore there is no mechanism to prevent old data to be read, so even if new data is being transmitted through the network at that time, old data will still be read. There were 19 float point variables stored in the billboard for every robot which gives 152 bytes for every robot.

2) Time synchronization: Since measurements are time-stamped there had to be a means of synchronizing clocks on every robot. We used the simplest, however not fool-proof protocol for time synchronization. One robot has to be declared as a master, and other robots which sync their clocks with the master are called slaves. The protocol then is as follows:

- A slave initiates a sync request by sending a timestamp TS to the master.
- On the data retrieval the master timestamps it $TS2$ and sends it back to the slave.
- The slave then calculates the current network latency.

$$latency := (currentTime - TS)/2$$

- Then time offset delta is calculated and added to the current time offset.

$$currentDelta := TS2 - latency - currentTime$$

$$timeOffset = timeOffset + currentDelte$$

It is clear that the network latency is not constant and this kind of time synchronization does not take into account latency values from the previous synchronizations. Thus every synchronization can be equally good or bad. Tests showed that most of the time no more than three synchronizations had to be performed in order to achieve preferable clock synchronization.

For the purpose of seeing the clock synchronization in action we implemented a music function into the robots. When the clocks are synchronized all robots can perform the same music piece in parallel where every next note is played by a different robot.

3) Data packets: We wanted to have a robust network which can be easily expanded. We chose to implement packet system on top of the provided low level network functions. This allowed us to easily send and receive different kinds of data for different purposes. It also allowed us to change the underlying low level network implementation (you can access network from leJOS using Java buffered readers/writers or low level functions) without any need to change any of the higher level code. Tests showed that Java buffered network readers/writers are more convenient to use but unstable when dealing with multiple connections.

All of the packets had a type field which lets to distinguish between different packets and a source field which tells where the packet originally came from. Creating a new packet type means adding a new packet class definition and writing a

handler for that packet class. Then this new packet can be sent from anywhere in the code and the robot on the other end will execute actions specified by the packet handler.

We had 13 different packets in total. All of them were used in many different situations including - but not limited to:

- billboard updates
- time synchronization
- orders from the GUI
- visualization data for the GUI
- changing various internal settings on-the-fly

F. Network Architecture

1) Limitations: The robot's Bluetooth stack is limited to one incoming connection and three outgoing connections. Any robot can initiate and accept connections simultaneously as long as the above mentioned limit is not reached. We have also noticed huge delays (sometimes up to one or more seconds) when a robot was dealing with more than one connection. We have also measured that it takes about two and a half seconds to successfully open a connection. The bandwidth tests showed that the maximum transmission speed achieved was 7 kilobytes per second and only when streaming (sending data from one robot to another without waiting for any response) big chunks (512B) of data. All of these details were important while designing the network architecture for the robots.

2) Network with a robot hub: Two kinds of network architecture solutions were considered. The first one is using one robot as a network hub. That way, a small network consisting of three slaves connected to one master can be constructed. There can be no more than four robots in such a network because of the limitations mentioned above. However there could be a possibility to interconnect these small networks and then have a network of any size.

This was our first network architecture of choice and we started implementing it. After major parts were completed we encountered some technical problems. First of all we noticed delays when a master was dealing with multiple outgoing connections. It took a great amount of time (sometimes up to two seconds) to switch from one connection to another. This was unacceptable if we wanted to keep the network fast and responsive.

Another issue coherent with network robustness was small network interconnection. We wanted to have a network architecture witch supports any number of robots (even though we only had three robots). This meant that there would have to be a routing system implemented since the network stack in the robot is not suitable for a fully interconnected peer to peer network (where every robot gets every data even if it is not addressed to it). This would have introduced quite an overhead which is not desirable on slow networks.

3) Network with a PC hub: Our solution was to move the network hub from a robot to the PC. A PC can manage many connections with no noticeable delays. There are practically no limits to the number of robots connected to one PC so no small interconnected networks are needed. Every robot has to manage only one connection to the PC eliminating multiple connection managing delays. Another advantage is that all the network traffic is going through the PC and the data can be used for visualization on-the-fly without any special code in the robots. Next to data visualization PC was also managing connections to the robots and forwarding packets from one robot to another.

G. Kalman filters

1) Java simulation: The 2π-discontinuity issue was detected in the Java simulation too. It seemed to occur a bit more often than in the Matlab simulation, though still not often enough to be a big problem.

The Java simulations showed that the absolute positioning filter as implemented in Matlab performed quite badly since it could not handle large errors in the cat orientation. An ad hoc solution to this was therefore tested. The idea was to correct the cat's orientation using one landmark measurement, assuming the cat's position is known and correct. This correction should only be made while the cat has turned but not yet begun traveling. If the position of the cat was more or less correct before the rotation, the error in the orientation could be corrected. The correct orientation should mean that the position of the cat remained accurate, enabling a correction of the orientation after the next turn. This solution seem to perform reasonably well if the measurement errors weren't too big. Criticism of this circular reasoning were however rightly risen. Therefore, the geometric filter was chosen in the brick implementation. It also used the above method of correcting the orientation from one landmark measurement. It could however correct both the position and the orientation at once if three landmark measurements were provided. This seemed like a good method for "getting back on track" if the errors got too big.

2) Brick implementation: Testing the tracking filter on the brick showed that the filter was somewhat sensitive to input data. Some input data seemed to result in a situation where the covariance matrix no longer remained positive definite, thereby failing the Cholesky factorization and crashing the filter. This happened in Matlab if the filter diverged, which happened only very seldom or not at all in the final code. In the Java simulation it happened somewhat more often, but it didn't seem to be a problem once we sorted out the last bugs. The brick implementation seemed to get this problem when the measurements weren't available for some time, which happened more often than in the Java simulation. It is believed that this caused an illegal covariance matrix to form. In normal operation however, this did not occur.

Again the 2π-discontinuity issue was detected but not deemed a critical issue.

An important element in the brick implementation is how often the filter can run, or how long each iteration takes. A faster filter will be able to better estimate the position and movement of a quickly moving and accelerating mouse. Each iteration of the tracking filter was measured to take about 400 ms. This was faster than the particle filter that took well over a second. Still, it was not as fast as we had hoped. Profiling the Java code both on the brick and the PC showed that matrix operations, mostly reading and writing, took a lot of time.

H. Particle filter

1) Java simulation: Simulations in Java where implemented only to weed out bugs. This approach was very successful since the filter, when on the brick, worked as intended. The filter could also be profiled to find where optimizations would do most good. Even though profiling did give a better insight on how the filter worked in practice the benefits were limited due to the vast differences of hardware between a PC and a NXT.

2) Brick implementation: A problem with the particle filter approach was that it is very CPU intensive. It was clear from the beginning that some platform and problem specific optimizations would be needed.

A common approach to speeding up non-linear functions is look up-tables. The up side of these is that they take little time to check. The only operations necessary is a rounding to the tables precision and then a look up in an array. The cons of this approach is that a lot of memory is needed to give the table accuracy. Since we did not know until very late in the implementation phase how much memory would be available, look up tables were not preferred.

The steps who needed optimizations badly [6] where:

1) Some mathematical functions would be to slow on the NXT.
   a) When comparing particles with sensor data one arc-tan operation is needed per particle (i.e. $\theta_{error} = \theta_{sighting} - arctan(\frac{y_{particle}-y_{est.position}}{x_{particle}-x_{est.position}})$).
   b) Criterion function is a Gaussian PDF (i.e. $w = C_1 e^{C_2 \theta^2_{error}}$).
2) Re-sampling needs one random sample per state variable and particle from a probability distribution in each iteration.
3) Floating point precision is not native to the NXT but implemented in firmware

Since the handling of floating point arithmetic is not native to the NXT, but emulated in the firmware, a classic approach to embedded mathematics was used, the so called fixed point arithmetic [16]. Fixed point uses only integer operations which are much faster than floating point, usually they are even so on processors with native floating point support. The basic idea is that all numbers are multiplied by a constant and then truncating the floating point number into an integer. If the constant is chosen as a power of two many important operations collapse to a bit shift. When doing a multiplication the result looks like the following.

$c \cdot d = ((a \cdot d) \cdot (b \cdot d))/d$

Where d is the constant all numbers are multiplied by. Now if d is chosen as 2^{BASE} then and all numbers are represented as fixed point at all times the above equation collapses to:

$c = (a \cdot b) \gg BASE$

Where the symbol $\gg$ represents a bit shift effectively dividing with the factor 2^{BASE}. The base used in the implementation was 20 and the format is called 12.20 fixed point.

When normalizing vector represented in fixed point some kind of square root algorithm was needed. Since floating point is used on many embedded devices there have been a lot of research on different algorithms. A simple algorithm which searches for the right square was implemented [17] and modified to suit the fixed point setup.

For sine and cosine a simple look up table was used. Because of symmetry in and between the two functions a high resolution can be obtained using little memory.

To replace the arc-tan and the exponential function a linearization technique was investigated. The data given are coordinates on a plane which lead to the though of the definition of inner product in $\mathbb{R}^2$

$$u \cdot v = \frac{cos\theta}{\|u\| \, \|v\|} \Longrightarrow \theta = acos\left(\frac{u \cdot v}{\|u\| \, \|v\|}\right) \tag{15}$$

$$w = f_{weight}\left(\frac{v_x}{\|v\|}\right) \tag{16}$$

If the evaluation function was plotted against $cos\theta$ from the vector product as in equation 15 the result could be approximated with a piecewise linear function. After some work trying to find the best segments to linearize the resulting function is plotted against angular deviation in Figure 12.

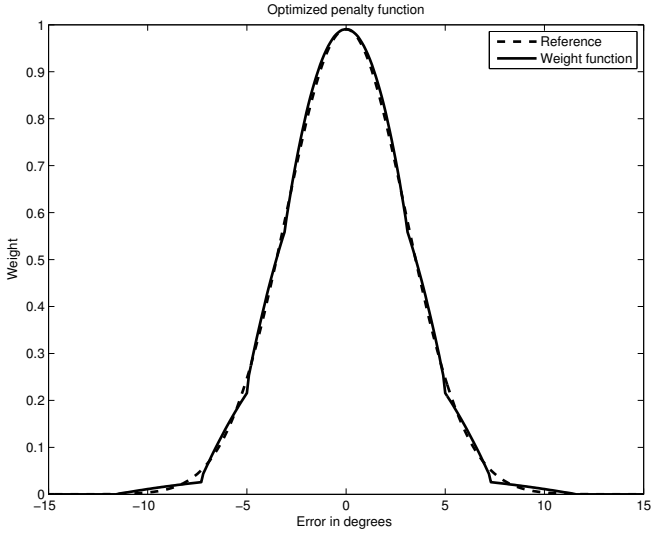

Figure 12. Plot of the linearized weight function and analytical values of the same as reference

Re-sampling the particles is not a hard or heavy task but can still be improved significantly. Most of the time the particles are good and the number of particles which really need to be re-sampled is small. If only the worst particles are re-sampled the filter would converge slower but would not need to do so much random sampling. However this method might need some overhead in sorting the particles. The method worked but was never compared with respect to accuracy.

I. Geometric filter

1) Java simulation: The filter was first implemented in our Java simulation framework and shown to work. There were some problems though.

The errors in the angle measurements had to be handled in some way. An easy method to do this to some extent

is using the assumption that the noise was a zero mean Gaussian process (same assumption as for the other filters). This was implemented as a pre-filter who would mean all the measurements of each landmark seen. So if there would be many sightings of a certain landmark all information would be used. This method is not very robust because the mean of the samples from a Gaussian process with mean zero would only go to zero with a large number of samples. This was not the case but the measurements did become better when running the filter with this pre-filtering method. All data would be invalidated if the cat would move and to further remove errors at least three landmarks had to be seen before correcting position.

An other problem could be that the circles would not intersect or that there would be no obvious choice with of the intersections would be the correct one. The filter always tried to go for the closest intersection. If no decision was obvious the filter would make the decision not to correct.

2) Brick implementation: Since the simulation framework works almost the same way as the brick code, from the filters point of view, the brick implementation was painless.

One difference between the brick and the simulation framework is how the errors in turned angle work. The discovery was made early that looking for three landmarks to correct the turned angle would be tedious. It would simply take too much time to look for three landmarks every time the cat had turned.

The compromise was made that the filter would correct its angle estimate with just one landmark, but only a very small amount at a time. Most of the position error was found to come from errors in the angle turned at the start of the execution of a move command. Error in traveled distance and turning drift while going straight was found to be very small. Hence this method gave us a lot less error in the position after the cat had traveled longer distances. The error in the turning did not have as much opportunity to propagate with distance traveled as before. The correcting was one degree per second and landmark seen. To complement this the decision was made to make the cat look for landmarks with even intervals but only if no other cat was sweeping for landmarks.

This approach turned out to work most of the time. Some of the time the filter would converge to a point far away from the actual position. Sadly there was no time to really test this or find the error.

J. Guide

The communication between the cats would most of the time be slow and hence the delay in information on the others positions would lag considerably. To avoid collisions the cats where not allowed to move too far in one go, to increase the chances of each cat having time to filter and communicate the information on position. Also they waited a while after a move to let the other cats have time to receive the new coordinates.

When implemented on the brick it was found out that the calculation was painfully slow. Sadly the implementation on the brick was late and the speed problems did not show in the simulation framework. When running in the environment (as the lowest priority thread etc) the calculation could take four seconds. Some last minute optimization then made it work in only one second even though that can not be called fast. It was decided to time slot the movement to avoid collisions. Since all cats have a synced clock it was not hard to implement. Each cat was given three seconds while the other ones waited for it to move. A faster implementation would not have been hard to do but working (however inefficient) code should never be changed at the last minute.

K. Graphical user interface

The filter data going from one cat to another was successfully used also for visual representation of the inner workings of the system. A GUI written in Java was created for this purpose. It had a 2D representation of fixed beacons in the corners, a red dot for the mouse, and three black dots with different ID numbers for identification of the cats. The 2D representation of the area automatically updated when a new packet arrived to the PC.

Robust network implementation allowed robots to send other useful information to the PC. One could monitor the landmark and/or mouse measurements that the robots took and used for further calculations. One could also monitor the decisions from the guide, which were sent to the GUI. Guide and other parts of the robots, such as which filter to use, were also able to be controlled from the GUI.

L. Java simulation framework - GSim

It was clear from the start that the work of implementing and experimenting with different filters would be carried out parallel to implementing the base system on the brick (networking, motors, camera etc.). Since this would result in long waits for the filter implementation a simulation framework would be needed. This would also give an opportunity to weed out bugs and test robustness in a controlled environment with good debugging tools available. A screen shot of the simulation is shown in figure 13.

The simulation is written in a way that the filter code would work with no or minor modifications (i.e. importing other libraries) on the brick. During the work with the implementation the simulation was changed to reflect the architectural decisions made for the brick.

The class structure was built using a base class for each type of filter that was slightly different on the brick and in the simulation but with all interfaces looking the same on both platforms. In the simulation a draw method was called at each graphics update to give feedback to the user. All noise in the simulation was, as often in simulations, ideal, but still gave a opportunity to test the basic robustness of the filtering methods.

Moving the code from the simulation to the brick was easy and development of the different parts of the project could be made parallel. At the end the simulation contained many thousands of lines of code. It still saved time since almost no bugs where encountered when the filters were on the brick.

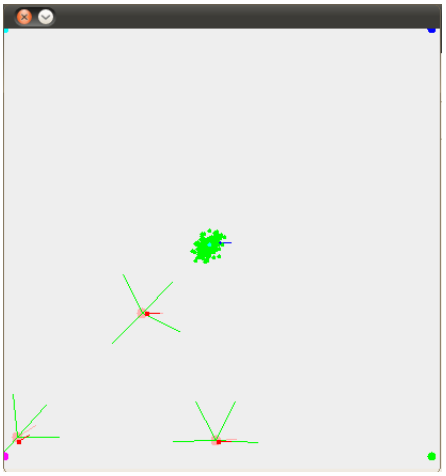

Figure 13. Java simulation framework in action showing three cats (red dots) tracking a target with the particle filter (blue dot covered with green particles)

V. Results

A. Mouse tracking

1) Setup: All tests were conducted within a 2.5 by 2.5m arena. The arena limits were also entered into the settings file of the robots, meaning that the filters could limit estimates to 0.0 or 2.5m if outside this area. The period of the UKF was set to 500ms and the period of the particle filter to 1000ms.

2) Stationary mouse: Both tracking filter implementations of the final system were first tested on a stationary mouse with the cats being set up in different formations. Figure 15 shows the result of the UKF when three cats are standing on a line as shown in Figure 14(a) while Figure 16 shows the corresponding result for the particle filter. It can be seen that both filters converges to a similar static error level, although the particle filter is more noisy. Running the same tests but with the cats placed in optimal positions as shown in Figure 14(b) did not significantly improve the performance, as can be seen in Figure 17 and 18. Tracking the static mouse with only two cats as shown in Figure 14(c) gave approximately the same result for the particle filter, see Figure 19.

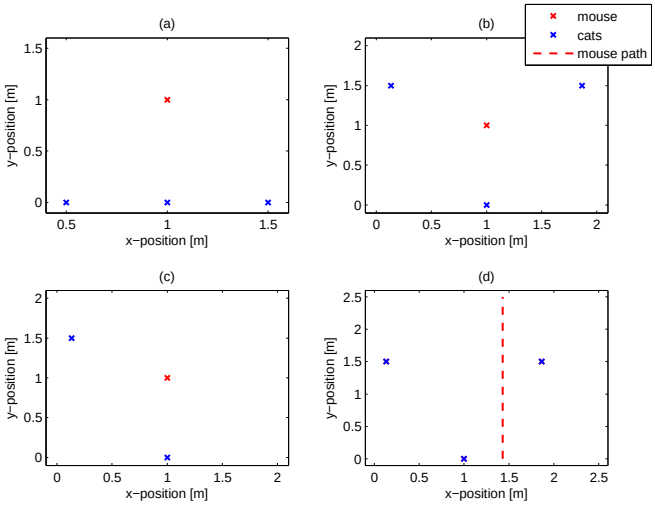

Figure 14. An illustration of the test arrangements.

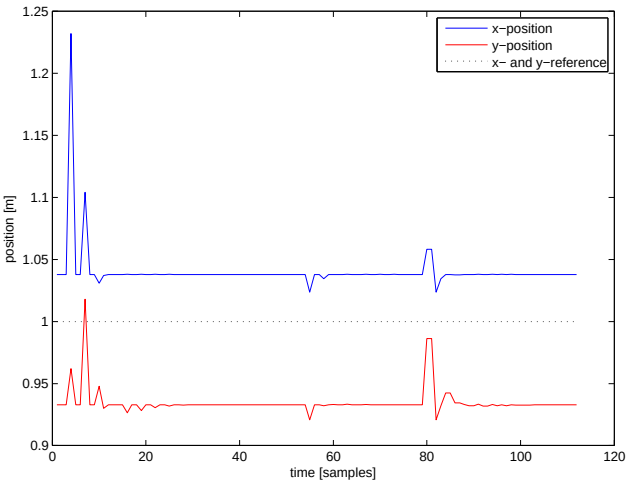

Figure 15. Plot of three cats standing on a line tracking a stationary mouse with the Unscented Kalman Filter. See Figure 14(a) for further details on the test arrangement.

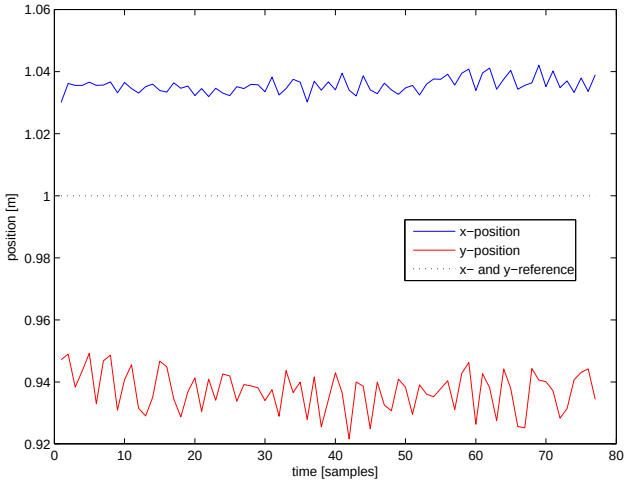

Figure 16. Plot of three cats standing on a line tracking a stationary mouse with the particle filter. See Figure 14(a) for further details on the test arrangement.

3) Moving mouse: The results for three stationary and decently placed cats tracking a mouse moving at a constant speed of roughly 0.19m/s as illustrated in Figure 14(d) is shown in Figure 20 for the UKF, Figure 21 for the particle filter and Figure 22 for the particle filter again but this time with the mouse moving at a lower speed.

B. Cat positioning

The geometric positioning filter was tested against positioning solely based on odometer data. The robot was programmed to travel around a square of size one by one meter three times and the error between the estimated and the actual position was then measured. The robot only turned at the corners of the square and it estimated its own position after each turn. The result of positioning without a filter and the corresponding result for the geometric positioning filter can be seen in Table I . The result shows that the magnitude of the positioning error is more then halved when using the geometric filter, although it differs greatly in x- and y-position.

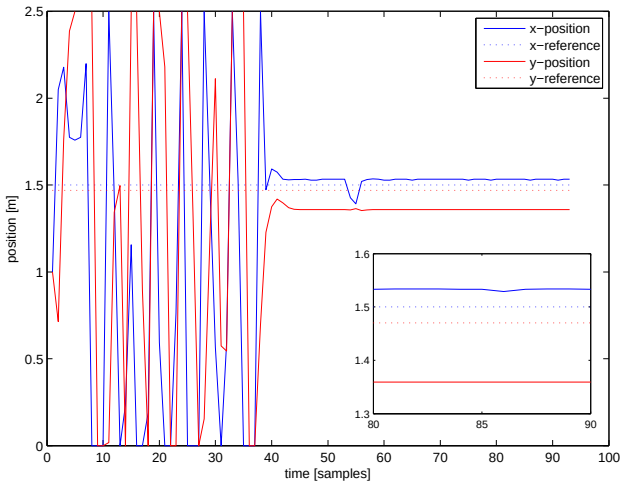

Figure 17. Plot of three cats standing on optimal positions tracking a static mouse with the Unscented Kalman Filter. The graph is cut at 0.0 and 2.5m because the data on the bricks were not allowed to exceed those values as that would be outside the arena. See Figure 14(b) for further details on the test arrangement.

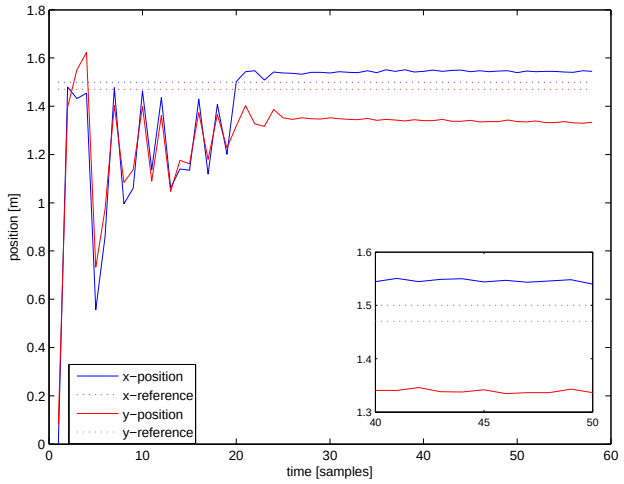

Figure 18. Plot of the final system with three cats standing on optimal positions tracking a static mouse with the particle filter. See Figure 14(b) for further details on the test arrangement.

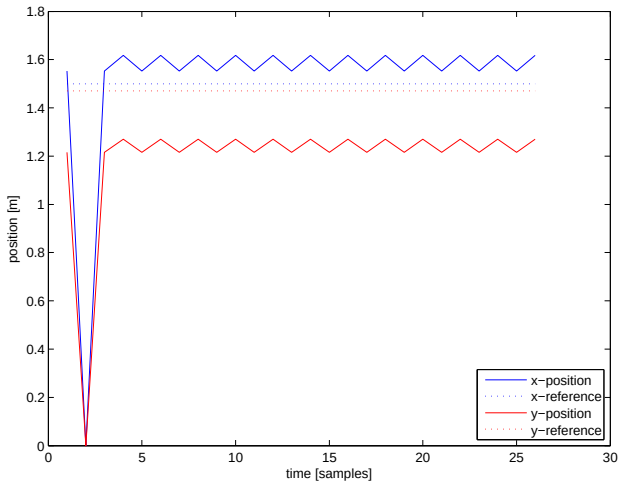

Figure 19. Plot of the final system with two cats tracking a static mouse with the particle filter. See Figure 14(c) for further details on the test arrangement.

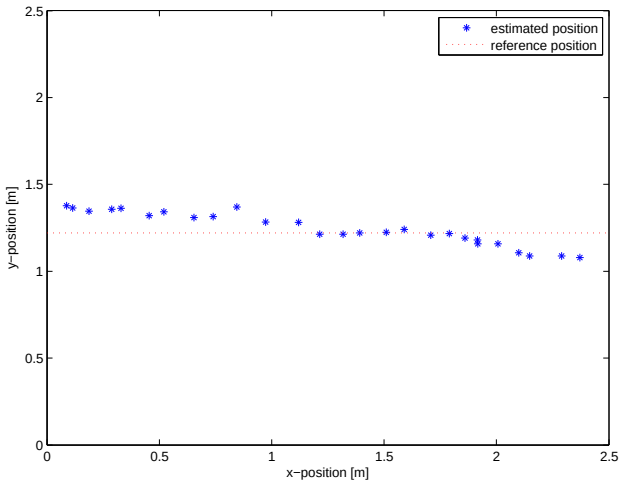

Figure 20. Plot of the final system with three stationary and decently placed cats tracking a mouse moving at a constant speed of roughly 0.19 m/s with the Unscented Kalman Filter. See Figure 14(d) for an illustration of the test arrangement.

VI. Future work

In this project we completed our goal of tracking a moving target using an autonomous team of mobile robots, allowed to take bearings-only measurements. All of the included subsystems are working on the designated Lego® Mindstorms® NXT platform. Some of the subsystems needs more work to achieve a good overall tracking performance. Primarily, the guide system should be considered a proof of concept. A faster computation time of the guide would make the cats' movement much more responsive and better adopted to tracking a fast target. This should also allow us to drop the time slotted movement as well, without risking collisions between the cats.

Secondly, the filters can be improved substantially by decreasing their computation time. When tracking the target during a limited time period, the tracking filters limit the overall performance to the greatest extent. The computation time of both the particle filter and the UKF can be reduced, allowing better tracking performance. Especially the UKF should be possible optimize greatly, given its relatively low computational complexity. In particular, the brick implementation can be improved by switching to a faster matrix class library to reduce the iteration time. One could switch to using single point arithmetic instead of double, as the former has more than enough of accuracy for our application anyway. Even a fixed point solution could be adopted, as was done with the particle filter. That could probably have made the iteration time a lot faster. Ultimately, the network transmission overhead

Table I
Positioning without a filter compared to positioning with a geometric filter.

Average error (magnitude) [m]	Without a filter	With geometric filter
x-position	0.21	0.18
y-position	0.11	0.02

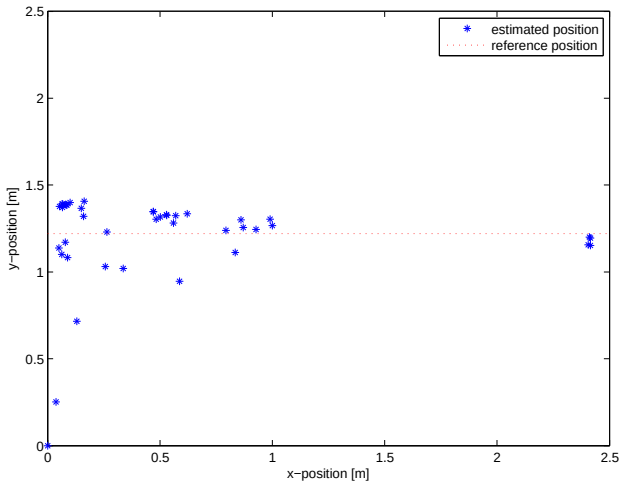

Figure 21. Plot of the final system with three stationary and decently placed cats tracking a mouse moving at a constant speed of roughly 0.19m/s with the particle filter. See Figure 14(d) for an illustration of the test arrangement.

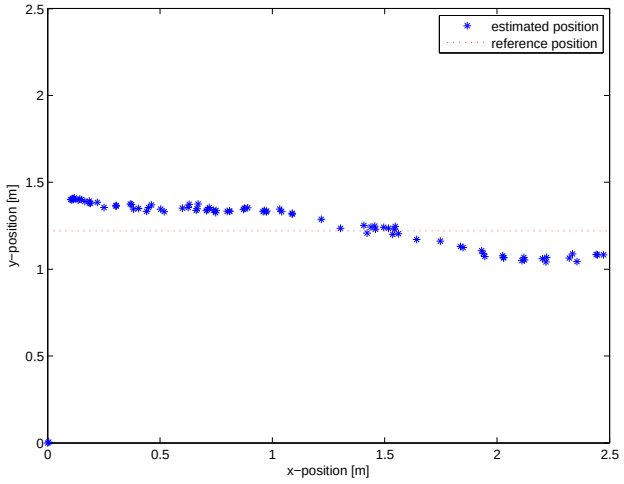

Figure 22. Plot of the final system with three stationary and decently placed cats tracking a mouse moving at a constant speed significantly slower than 0.19m/s with the particle filter. See Figure 14(d) for an illustration of the test arrangement.

puts a limit on how much of a increase in overall system performance we can expect by reducing the filter iteration times. But since the network overhead for one cycle was about 50 ms, there is quite a large room for improvement.

The UKF can be used in conjunction with the particle filter to improve its performance. This was demonstrated by *Yong Rui* and *Yunqiang Chen* in [18]. By using the UKF for proposal distribution, the generated particles are placed closer to the true states. The increased accuracy can also be traded for a faster iteration time by using fewer particles. This should be an interesting and quite straightforward continuation of our work with the filters.

During longer tracking periods the absolute positioning filter becomes more important. A working UKF or particle filter of this kind should improve the performance greatly. With the used landmark positions, each cat should have landmark in its field of view at all times while tracking the mouse. This is assuming the mouse does not travel too close to the edges of the arena, and that the mouse does not occlude the landmarks. It should be possible to minimize these cases by adding appropriate terms to the criterion function of the guide. Under these circumstances, a well implemented absolute positioning filter should be able to position the cats accurately at all times, even given a large error in the odometer readings. Ideally, the filter should be able to correct the cats' position and orientation while they are moving. This was achieved in the Matlab filter simulations, so a brick implementation with this feature should be within the realm of possibility.

An interesting possibility is to enable feedback from the tracking filter to the absolute positioning filter. When three or more cats are tracking the mouse the conformity of their bearing readings is likely to vary with the accuracy of their own estimated position. The covariance matrix of the estimated target position can be used to measure this conformity. If the conformity becomes too low, the cats should improve their own position estimate, e.g. by sweeping for landmarks, thus enabling on-demand absolute positioning. This can be extended when four (or more) cats are used. By examining the conformity of all the different subsets containing three cats, it should be possible isolate a cat that has a bad absolute position estimate (assuming the others are more or less accurate). This is similar to cross-validation in statistics.

Another way of improving the tracking performance is by using better and/or more sensors than the ones we had access to. The camera we used had a very low resolution, making long range target detection difficult. Also, using predetermined color ranges to detect the target and the landmarks has its limitations – it becomes very sensitive to changes in the surrounding scene and the ambient lightning. An alternative could be to use patterns instead of colored lights as beacons. This would however be computationally more complex and probably require a higher resolution camera.

What could be interesting is to replace the camera totally with something else, for example an IR-detection module, and use different IR-codes (i.e. blinking in different speeds) as beacons. Another suggestion could be to use an ultra sonic sensor to detect object close to the brick for localization of the other cats and mouse. Some improvements to the absolute positioning filter could possibly be achieved by adding data from a compass. Regarding the movement pilot, it could be interesting to achieve arc-movement for the robot implemented with the guidance system. This could result in smoother movement, but would ideally require Ackermann's steering correction, to avoid wheel slip and incorrect odometer readings [19]. Unfortunately, such a construction would be hard to build with Lego.

References

[1] Wikipedia, "Extended kalman filter." Viewed 2011-03-04.
[2] S. Julier and J. Uhlmann, "A new extension of the Kalman filter to nonlinear systems," in *Int. Symp. Aerospace/Defense Sensing, Simul. and Controls, Orlando, FL*, 1997.
[3] Wikipedia, "Unscented kalman filter." Viewed 2011-03-04.
[4] E. A. Wan and R. Van Der Merwe, "The Unscented Kalman Filter for Nonlinear Estimation," *Adaptive Systems for Signal Processing, Communications, and Control Symposium 2000. AS-SPCC. The IEEE 2000*, pp. 153–158, 2002.

[5] M. Coates, "Distributed particle filters for sensor networks," in *IPSN '04: Third international symposium on information processing in sensor networks*, pp. 99–107, 2004. 3rd International Symposium on Information Processing in Sensor Networks, Berkeley, CA, APR 26-27, 2004.
[6] S. o. I. T. Miodrag Bolic and U. o. O. Engineering, "Theory and implementation of particle filters," 2004-11-12.
[7] I. M. Rekleitis, "A particle filer tutorial for mobile robot localization," tech. rep., Technical report TR-CIM-04-02, Center for Intelligent Machines, McGill Univeristy, 3480 University St., Montreal, QuÃľbec, CANADA H3A 2A7, 2004.
[8] Wikipedia, "Particle filter." Viewed 2011-03-04.
[9] Wikipedia, "Learning the unscented kalman filter." Viewed 2011-03-04.
[10] Wikipedia, "Inscribed angle theorem." Viewed 2010-08-02.
[11] Wikipedia, "Gradient descent." Viewed 2010-08-02.
[12] Wikipedia, "Line search." Viewed 2010-08-02.
[13] X. Soldaat, "Mergeblob v0.2."
[14] M. Mickevičius and N. Törnblom, "Filter patch for mergeblob."
[15] Wikipedia, "Magnification." Viewed 2011-01-16.
[16] R. Yates, *Fixed-Point Arithmetics: An Introduction*. Digital Signal Labs, pa6 ed., 2009-07-07.
[17] K. Turkowski, "Fixed point sqare root," *Apple Technical Report*, vol. 96, 1994.
[18] Y. Rui and Y. Chen, "Better proposal distributions: Object tracking using unscented particle filter," in *Computer Vision and Pattern Recognition, 2001. CVPR 2001. Proceedings of the 2001 IEEE Computer Society Conference on*, pp. 786–793, 2001.
[19] Wikipedia, "Ackermann steering geometry." Viewed 2011-03-24.

VII. Individual contribution

Fredrik Wahlberg
Project leader. Worked on the Particle filter (both tracking and positioning) and Geometric "filter" theory, simulation and implementation. Participated in software architecture and design. Wrote most of the Java simulation framework and the AI for robot positioning (guide, maximizing sensor reading information and avoiding collision).

Christian Ålander
Wrote most of the GUI code.

Nils Törnblom
Worked on the Kalman filters; modeling, simulation and implementation. Also the person charge of the movement pilot regulator, camera tracking regulator, landmark design and detection. Involved in some of the robot design and construction. Co-developed the color coding.

Martynas Mickevičius
Network communications implementation. Robot and GUI software architecture design.

Edvard Zak
Worked on the Unscented Kalman filters for tracking and positioning, mainly the Java implementation and its testing. Active in robot software architecture and design. Worked on the final testing.

Cheewin Pisanupoj
Worked on building the robots.