(1,1)

Fabio Marti, Matthias Roggo, David Lehnen, Daniel Gilgen

Robocup: Cooperative estimation and prediction of player and ball movement

Group Project

Department:
IfA – Automatic Control Laboratory, ETH Zürich

Supervising Professor:
Prof. Dr. John Lygeros, ETH Zürich

Supervisor:
M.S., Ph.D. Student Sean Summers, ETH Zürich

Zürich, March 2012

Abstract

Reliable estimation and prediction are important components for the ETHZ RoboCup Team to win the annual world cup of the Standard Nao Platform League in near future. This paper presents the most important ideas and results of our Group Work in estimation and prediction of a robot soccer match on the Standard Nao Platform. In a first part the simulation of robot and ball movements and the underlying implementation in MATLAB are described and discussed. The second part provides a short theoretical insight in Kalman-filtering and the application of this theory in the framework of the simulation. The last part of this work deals with the multiplicity of measurements and the need of sensor fusion which comes along with this. This Group Work claims to provide a good basis for further investigations in the area of estimation such as improvements in Kalman filtering or sensor fusion and the migration of the implementation on the Nao Platform.

Contents

List of Figures

1 Introduction

RoboCup ("Robot Soccer World Cup") is an international ongoing robotics competition founded in 1997. The official goal of the project: "By mid-21st century, a team of fully autonomous humanoid robot soccer players shall win the soccer game, complying with the official rule of the FIFA, against the winner of the most recent World Cup." [3] There are a lot of topics of automation and control contained in this project. One is the topic of estimation, which is a part of our group work. The goal of our work is to build a cooperative estimation and prediction of player and ball movement. A detailed description and further informations about the goal of our group work can be found in the appendix **??**. Therefore we have to estimate the position of the robot on it's own and the position of the other robots and the ball. In a second step we have to fuse all the measurements together to a big point of view and then to decide whether the data from the robots are reliable. .....

2 Simulation

One main aspect of this group work among the design of a Kalman filter was the construction of a graphical environment for our work, so we decided to implement a simulation of a Nao soccer match in MATLAB. Therefore we created three independent modules which build the framework for the simulation. The three parts contain functions concerning the playing field, the robots and the ball respectively. Since these parts are constructed in a modular fashion, we can change one module without influencing the other two. All functions mentioned below will build the core of a simple random simulation of a RoboCup soccer match. The simulation is called random because the input paramters of the robots, i.e. their velocity and their change of angular direction, are chosen randomly. The script `RoboCupSim.m` is essentially a finite loop with the described functions in it and every step in the loop generates a new frame of our simulation. Most of the global variables are defined in `RoboCupSim.m` such as the dimensions of a real RoboCup playing field or all noise-relevant parameters. Also variables which cannot be initialized in a function, such as the covariance matrices for the Kalman filters which will be discussed later, are defined in this script. The pseudocode below shows the general structure of our simulation framework and emphasizes the modularity of our code

```
General settings
Initialize Field Parameters
Initialize Noise Parameters
Initialize Robot and Ball Parameters

Begin Loop

      Robot computations
      Ball computations
      Robot filtering
      Ball filtering
      Plot environment and objects
      // Debugging tool

End Loop
```

This chapter focuses on the implementation of the computation and plot functions. The filter task of the simulation will be discussed in chapter 4.

2.1 The Plots

All graphical features concerning the playing field are implemented by the function `plot_env.m`. It is subdivided in two functions, which draw the field and, as a neat add on, the scorecounter seperately. We mostly use built-in MATLAB commands for this task such as `rectangle()` or `line()`. The following short code excerpt shows for example how the center point of the playing field is drawn

```
    draw_circle(0,0,Field.centerCircleRadius,'k',0);
```

where `draw_circle(x,y,r,color,filled)` is a custom-build function to draw marker circles of the field, but also circles representing the robots and the ball. All functions are designed for fast calculations since we want as many frames as possible if the simulation is running. Figure **??** shows the playing field after the execution of `plot_env.m`

Figure 2.1: Playing field after initialization.

2.2 The Robots

Maybe the most important part of the simulation is the adequate depiction and behaviour of all eight robots. The latter task is split in three components: In a first step the robots are initialized, after that, their new positions on the field, according to their motion equations, are computed and in the end we are adding measurement noise for our filtering task. The initialization of the robots is quite simple. The function `dummy_init().m` just creates eight structs with the essential informations for every robot, i.e. its horizontal and vertical position, its direction and its team affiliation. Furthermore the function defines the robot's radius and its maximum possible angular change for one timestep as global variables. In a latter stage of development we dropped the simplification of a global eye and assumed that a location of a robot's position is only possible if it is in the sight of view of at least one other robot. So in the second version `robot_init().m` of this function, we additionally defined global variables for the robot's velocity, its distance of sight as well as its angle of sight. Once initialized we use the functions `dummy_step(Robot).m` and `robot_step(Robot,Ball).m` respectively to compute the attributes of all robots for every timestep. The key issue of both functions however is the recalculation of the position, i.e. the motion of the robots

```
%------------ Motion equations for robots  ------------%

    for i=1:8
```

```
        RobotStep(i).color = Robot(i).color;
        RobotStep(i).x = velocity(i) * cos(Robot(i).dir) + Robot(i).x;
        RobotStep(i).y = velocity(i) * sin(Robot(i).dir) + Robot(i).y;
        RobotStep(i).dir = d_omega(i) + Robot(i).dir;
    end
```

Later on the non-linearity of these equations will make it necessary that we use an extended Kalman filter instead of a simple linear one. The addition of process noise and the collision handling of robots also happen in these functions. The process noise we use for our model is always white Gaussian noise with seperate covariances for the positions and the directions. The handling of collisions, between robots and the field's border, however is different for both step-functions. In `dummy_step(Robot).m` we used a very simple model, assuming that if two robots collide, their directions swap such that they do not run into each other

```
                RobotStep(i).dir = Robot(j).dir;     % Swapping the
                RobotStep(j).dir = d;                % directions
```

This solution was not optimal for two reasons: First we had the problem that there was a bug which made it possible that two robots became wedged together and their further behaviour was messed up after they had contact. The second problem was, that our Kalman filter didn't work properly since the swapping of directions is a rapid change in states, which cannot be handled by our Kalman filter. The function `robot_step(Robot,Ball).m` eliminates both issues because it doesn't use collision detection but collision avoidance. If the distance of two robots falls below a given radius, we assign a potential to both robots. The effect is now similar to this of a positive charge in an electrostatic field: The closer two robots are, the bigger is their mutual repulsion. As you can see below, we use the same r^{-2}-relation for the repulsing force as it is known from the analysis of electrostatic fields

```
        for j=1:8
           % Length of the vector
           r = sqrt((Robot(j).x-Robot(i).x).^2+(Robot(j).y-Robot(i).y).^2);

           % Potential function to compute repulsion between robots. If
           % the distance between two robots is r_a, we have unit repuls-
           % ion.
           if (i~=j && (r<=r_a))
               x_v = x_v + (Robot(i).x-Robot(j).x)./r.^3.*r_a.^2;
               y_v = y_v + (Robot(i).y-Robot(j).y)./r.^3.*r_a.^2;
           end
        end
```

The change of directions is now continuous, which makes tracking still possible for our Kalman filter. Additionally to the collision avoidance, the robots are now attracted to the ball if it near them. Up to this point we computed the behaviour of an ideal robot. Since our goal is to filter out the uncertainty of motion of the robot parameters, we artifically have to add some measurement noise which we can filter later on. The functions `dummy_measure(Robot).m` and `robot_measure(Robot).m` are implemented for that purpose. `dummy_measure(Robot).m` simply adds white Gaussian noise to the position and the direction of every robot. Additionally with a certain possibillity there is no measurement at all, so the corresponding parameter is dropped. What we are assuming with this model is, that there is some global eye available which measures the position and direction of every robot with some given resolution. Since this scenario is not realistic in a RoboCup soccer match, because only the robots themselves can gain visual information, we developed the function `robot_measure(Robot).m` to solve this problem. According to this philosophy, a measurement of a robot is only available if it stands in front of a characteristic point on the field or if it is within the distance of sight of another robot. Note that a robot only locates other robots if it is aware of its own position and that we get information from only one team, which will be the case for official RoboCup matches. If more than one measurement is available for a robot, the mean of all measurements is computed. After this computational part, the robots are added to the graphical environment by using the function `plot_robot(Robot,Ball,style).m`. It provides several features such as the coloration, the shape and the label of the robots on the field. A sample output on the graphical interface after several timesteps is shown in figure 2.2 below

Figure 2.2: Capture of a regular simulation frame.

Note that measurements (white crosses) are not available for most robots (blue and magenta circles).

2.3 The Ball

The treatment of the ball is quite similar to that of the robots. The ball object is also represented by a struct containing its horizontal and vertical position, its direction and the velocity. These parameters are set by executing `ball_init().m`. This function also defines the ball's radius, its initial velocity and the friction towards the ground. The function `ball_step(Ball,Robot).m` does, like the equivalent function for the robots, the computations of the ball's next position and direction on the field. These parameters however do not only depend on the ball's dynamics but also whether it collides with one of the robots. The following MATLAB code shows the algorithm for this collision detection

```
%------------ Checking collision with robots ------------%

    for i=1:8
        dX = Ball.x - Robot(i).x;
        dY = Ball.y - Robot(i).y;
        if (dX.^2 + dY.^2 < (RobotParam.radius + BallParam.radius).^2)
            BallStep.dir = angle(dX + dY*j);
            BallStep.velocity = 1;
            BallStep.x = BallParam.velocity * Ball.velocity * cos(BallStep.dir) + Ball.x;
            BallStep.y = BallParam.velocity * Ball.velocity * sin(BallStep.dir) + Ball.y;
        end
    end
```

In our simple model we assume that if the ball collides with one of the robots, it regains its initial velocity and bounces away, perpendicular to the robot's position. Since the ball too is a subject of measurement, we also needed a measurement function for this object. `ball_measure(Ball).m` adds measurement noise or, as the case may be, drops the measurement completely. Again, a measurement is only available if it is in the sight of view of at least one blue robot that knows its own position. For more than one measurement the mean value of all measurements is taken. After all computations are done, the ball is drawn on the field, together with the robots, with the function `plot_robot(Robot,Ball,style).m`. All features of this function like coloration and drawing of different shapes are also available for the ball.

3 Kalman Filtering: Implementation in the MATLAB Environment

3.1 Estimation of the Robots

As we have seen before, the motion equations of the robots are nonlinear, which makes it necessary to use an extended Kalman filter for the estimation of the robots. The function `robot_ekf(robot_m,robot_e,m_values,e_values,d_omega,v,P).m` is dedicated for this task. As stated in the theory above, we will need the old estimates and the measurements in order to compute new estimates. Most matrices like the covariance matrices or the Jacobian matrices are the same for all eight robots and hence have to be defined only once. The error covariance P and the estimator K on the other side have to be stored for every robot individually, therefore we will use cell arrays for this task. The initialization in MATLAB of these parameters is shown below

```
%———————— Init of covariance matrices and linearized matrices ————————%

    Q = [Noise.process.pos.^2*eye(2), [0;0]; [0 0 Noise.process.dir.^2]];
    H = eye(3);
    V = eye(3);
    W = eye(3);
    x_apriori = [0;0;0];
    K = zeros(3,3,8);
    P_step = zeros(3,3,8);
```

In a next step we calculate the Kalman estimate for every robot. In a former version of the function, the two cell arrays `m_values` and `e_values` contained a history of the last measurements and estimates of the robots. They were used to improve the performance of the extended Kalman filter. The collision detection of former versions of the simulation made it necessary for the filter algorithm to be responsive to large changes of the direction of the robots. Since the Kalman filter itself couldn't handle these rapid changes, we needed a function that indicates that the measurements are much more reliable if there is a huge difference between them and the estimates over

a certain space of time. The essential functionality was to reduce the matrix R, which caused the extended Kalman filter to heaviliy trust the incoming measurements. For this task a history of former measurements and estimates was necessary. But since these problems disappeared with the introduction of collision avoidance, the used methods are obsolete. The function `i_measurement(robot_m,m_values,e_values).m` however is still needed for another reason: Measurement drops over a large time span will cause divergence between estimated values and actual values. Regaining visual information from robots will eventually lead to convergence of both values, but this usually happens way too slow. Therefore we needed a method to get the estimates quickly back on track, if there is a huge difference between them and the measurements. The following computations are done for every robot

```
x_abs = abs(e_values(1,1,i)-m_values(1,1,i));
y_abs = abs(e_values(2,1,i)-m_values(2,1,i));
dir_abs = abs(e_values(3,1,i)-m_values(3,1,i));
prob_x(i) = erfc(x_abs/(sqrt(robot_m(i).sigma)));
prob_y(i) = erfc(y_abs/(sqrt(robot_m(i).sigma)));
prob_dir(i) = erfc(dir_abs/(sqrt(robot_m(i).sigma*2*pi)));
```

The current implementation uses probabilistic methods, to decide how to reduce the matrix R. The key idea behind this method is, to compute how probable it is, that the measurements and estimates differ even more, than they actually do. For example: If the estimation of a variable is equal to its measurement, we have probability 100% that their difference is even bigger. In this case R is multiplied by 1; there is no need to trust the measurement more than usual because our estimation is pretty good. Another challenge of the estimation was how to get information about the inputs of enemy robots. This task is done by `input_approximation(robot_m,m_values,v).m`. The goal is to reconstruct the change of the angular position and the velocity only by looking at the measurements. Since the change of the angular position is a random process which produces statistically independent values for every timestep, a prediction of this value is impossible. The best one can do is therefore to form the difference between the robot's measured direction in time step k and $k-1$, so we get a delay of one timestep in our calculations. The input variable will now contain some error, but this can be handled by the Kalman filter as long as the error is small enough. For the velocity on the other side we use a different method. The velocity is computed out of the position measurements, which are heavily affected by noise. Therefore the computation of the velocity in only one timestep is not very reliable. We solved this problem by using a sliding window, meaning that the velocity, we use as an input, is the mean value of velocities of several timesteps. This is a valid approach, since contrary to the change of angular position, the velocity can't change very rapidly. The corresponding code excerpt is shown below

```
d_omega_step(i-4) = robot_m(i).dir - m_values(3,1,i);
v_new = dt*sqrt((robot_m(i).x-m_values(1,1,i)).^2+(robot_m(i).y-m_values(2,1,i)).^
```

```
            v_step(i-4) = (v(i-4)*(window_size-1)+v_new)/window_size;
```

If there are no measurements at all, we assume maximal velocity and zero change in direction. After facing the problems presented above, we could implement the time update and measurement update equations just as they were stated in the theory. The MATLAB-code below forms the core of the extended Kalman filter for the robots

```
% Time update (predict)
x_apriori(1) = robot_e(i).x+cos(robot_e(i).dir)*v(i);
x_apriori(2) = robot_e(i).y+sin(robot_e(i).dir)*v(i);
x_apriori(3) = robot_e(i).dir+d_omega(i);
P_step(:,:,i) = A*P(:,:,i)*A'+W*Q*W';

% Measurement update (correct)
if isnan(robot_m(i).x * robot_m(i).y * robot_m(i).dir)
    estimates = x_apriori;   % Measurement drop
else
    z = [robot_m(i).x; robot_m(i).y; robot_m(i).dir];
    K(:,:,i) =  (P_step(:,:,i)*H')/(H*P_step(:,:,i)*H'+V*R*V');
    estimates = x_apriori+K(:,:,i)*(z - x_apriori);
    P_step(:,:,i) = (eye(3)-K(:,:,i)*H)*P_step(:,:,i);
end
```

The prediction of the robot's position and direction can be done for every time step, but the correction is only possible if all measurements are available. This is not the case for example if robots don't get visual information of other robots. With the if-statement we accomodate this fact. So the typical Kalman cycle is only executed if we have measurement on the positions and the direction. Otherwise we drop the measurement update, i.e. our new estimate is simply a simulation of the robot's motion with the former estimates.

3.2 Estimation of the Ball

The estimation of the ball is done by `ball_kf(Ball_oe, Ball_m, P_oe).m`. This function has essentially the same functionality as the Kalman filter for the robots. If there are measurements of the ball's position and direction, the time update and measurement update equations are computed, otherwise only the time update equations are calculated. The only difference between the ball and the robots is, that large changes of its motion are possible. Such discrete events are handled with the if-statement shown below

```
x_oe = [Ball_oe.x ; Ball_oe.y ; Ball_oe.dir];
x_m = [Ball_m.x ; Ball_m.y ; Ball_m.dir;];
```

```
if(norm(x_oe(1:2)-x_m(1:2))>0.5 || abs(x_oe(3)-x_m(3))>pi/2)
    P_oe = eye(4);
end
```

So the Kalman filter for the ball is switch triggered: If a certain threshold of acceptable error in position or direction estimation is exceeded, we reset the Kalman filter. This has the effect that the tracking of the ball is quickly reestablished after a collision with a robot or with some of the boundaries.

4 Sensor Fusion

In RoboCup every robot works autonomous except a WLAN connection between the teammates. So the whole team can share information to optimize their game. Through his field of view every robot can bring in some informations about the playing field, other robots and about the ball. Now to optimize the estimation and to exploit the informations from the robots, the team can share this informations in form of a sensor fusion algorithm. Sensor fusion offers a great opportunity to overcome physical limitations of sensing systems.[4]

In the case of RoboCup we need as so called High-level fusion (decision fusion). Methods of decision fusion do not include only the combination of position, edges, corners or lines into a feature map. Rather they imply voting and statistical methods. [4].

4.1 Sensors of the Robots

The robots come with a camera with a defined field of view. There are no other sensors or informations which can be used for estimation of the positions. There are three types of sensor configuration regarding to sensor fusion. [4]

- Complementary
- Competitive
- Cooperative

In the RoboCup case all the types can occur. The sensor configuration is complementary if a region is observed by only one robot or camera. And if there are two or more cameras the sensor configuration can be competitive or cooperative.

4.2 Mathematical model for Sensor Fusion

We have chosen a situation-based statistical model for our purposes. The covariance of a measurement depends on the situation at which a robot gains the positon and the angle of another robot. Differently said: There are several scenarios under which a robot can

get visual information and each scenario is characterized by the quality of a measurement, implying the value of the covariance. In our case, the measurement of the i-th robot looks as follows

$$\hat{X}_i = X + w_i$$

X is the deterministic variable we want to know (i.e. position or direction) and w_i is white Gaussian noise with a certain covariance σ_i^2, depending on the condition under which a robot got the information, so $w_i \sim \mathcal{N}(0, \sigma_i^2)$. A fusion of n measurements now looks as follows

$$Y = a_1\hat{X}_1 + a_2\hat{X}_2 + \ldots + a_n\hat{X}_n \tag{4.1}$$

where n in our case can be at most 4. The main task of sensor fusion is to find the weights a_i. To find them, we make two reasonable assumptions: The mean of the fused measurement has to be equal to X. On the other side the mean of the squared error has to be minimal. Formally we want

$$E[Y] = X \qquad \text{and} \qquad E\left[(Y - X)^2\right] \rightarrow \text{minimal} \tag{4.2}$$

The first condition leads to the simple fact, that all weights have to sum up to 1

$$a_1 + \ldots + a_n = 1 \qquad \Longleftrightarrow \qquad G(a_1, \ldots, a_n) = a_1 + \ldots a_n - 1 \tag{4.3}$$

For the second condition, we are assuming that all w_i are mutually independent, so we have $E[w_i w_j] = 0,\ i \neq j$. Taking (4.3) into account leads to the second condition

$$F(a_1, \ldots, a_n) = a_1^2\sigma_1^2 + \ldots + a_n^2\sigma_n^2 \quad \rightarrow \quad \text{minimal} \tag{4.4}$$

The generic approach is now to solve equation (4.4) with $\nabla F(a_1, \ldots, a_n) = 0$. This attempt fails, because we also have to meet the side condition from equation (4.3). Therefore we use the method of Lagrange multipliers [5]. The equations we have to solve, are shown below

$$\nabla F(a_1, \ldots, a_n) = \lambda \nabla G(a_1, \ldots, a_n), \qquad G(a_1, \ldots, a_n) = 0 \tag{4.5}$$

The system of equations above consists of $n+1$ equations and $n+1$ unknowns, so there is a unique solution which is shown below

$$a_i = \frac{\sigma_i^{-2}}{\sum_{j=1}^{n} \sigma_j^{-2}} \tag{4.6}$$

In a final step, we are interested in the statistical properties of our fused measurement. The mean is already given by condition (4.2). Since Y is a Gaussian random variable we only need the covariance σ_Y^2 to fully describe Y. We get

$$\sigma_Y^2 = E\left[Y^2\right] - E[Y]^2 = \frac{1}{\sum_{j=1}^{n} \sigma_j^{-2}} \tag{4.7}$$

A short analysis reveals, that $\sigma_Y^2 < \sigma_i^2$, $\forall i$; the covariance of the fused measurement is smaller than the covariance of every single measurement. Or in other words: The fused measurement is, from a statistical point of view, always more precise than the measurement of only one robot, even if we fuse a very good and a very bad measurement.

4.3 The implementation in MATLAB

The sensor fusing methods for our simulation are embedded in the measurement functions `robot_measure(Robot).m` and `ball_measure(Robot, Ball).m`. There are three scenarios under which visual information is obtained: In the first case a blue robot determines its own position by recognizing a characteristic point on the field. This kind of measurement has the lowest possible covariance in our model. In the other two scenarions, blue robots measure the position and direction of other robots or the ball, either with or without the knowledge of their own position. It is clear that the covariance of the former case must be smaller, so the worst measurement we can get is one we get from a blue robot which is not aware of its own position. All measurements and the covariances which are associated with them, are stored in an array of structs for the blue robots. After the measuring is done, every blue robot presents its records, if there are some, of a certain robot or the ball and adds it to the fused measurement. The code shows how this is done for the robots

```
        dir = 0;
        k = 0;
        for j = 1:4
            if (~isnan(RobotAllMeasure(i).x(j)))     % Check for measurement
                sigma2 = (RobotAllMeasure(i).sigma(j));
```

```
            x = x + RobotAllMeasure(i).x(j)./(sigma2.^2);
            y = y + RobotAllMeasure(i).y(j)./(sigma2.^2);
            dir = dir + RobotAllMeasure(i).dir(j)./(sigma2.^2);
            k = k + 1./(sigma2.^2);
```

Note that `sigma2` is the standard deviation and not the covariance. After this step all variables are simply divided by the sum of the inverse values of all covariances as stated in the theory above.

```
        % Compute the weighted mean of all measurements
        if k ~= 0
            RobotMeasure(i).x = x./k;
            RobotMeasure(i).y = y./k;
            RobotMeasure(i).dir = dir./k;
```

For the ball we are essentially doing the same and we only distinct whether we get a measurement from a robot who knows its own position or not. The subsequent fusion of all records works after the same principle as the fusion for the robots.

5 Flaws and Outlook

Bibliography

[1] G. Welch and G. Bishop, "An Introduction to the Kalman Filter," UNC-Chapel Hill, 2006.

[2] RoboCup Technical Committee, "RoboCup Standard Platform League (Nao) Rule Book," RoboCup, 2011.

[3] `http://www.robocup.org/about-robocup/objective/`
Webpage of the RoboCup Organisation, last access: 08.05.2012

[4] Wilfried Elmenreich, "An Introduction to Sensor Fusion," Institut fr Technische Informatik Vienna University of Technology, 2001.

[5] Christian Blatter, "Ingenieur Analysis 1 und 2," Springer, 1996.