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 Summer, ETH Zürich

Zürich, March 2012

Abstract

Hier kommt das Abstract ...

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.

2.1 The Playing Field

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`

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

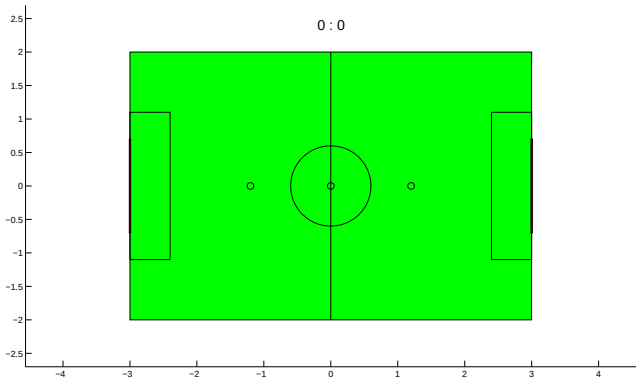

Figure 2.1: Playing field after initialization.

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

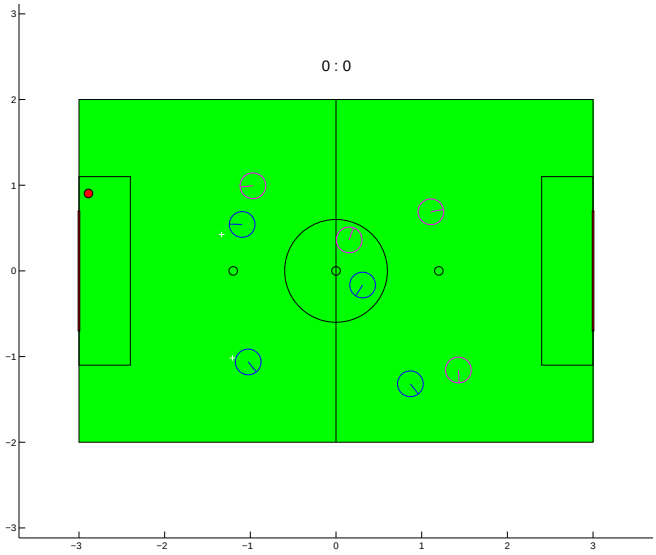

Figure 2.2: Capture of a regular simulation frame.

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.

2.4 A Random Simulation

All functions mentioned above 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 length of a simulation is user configurable and is currently set to 2000 frames.

3 Kalman Filtering: Theoretical Basics

3.1 Linear Kalman Filter

There exists a recursive Kalman filter algorithm for discrete time systems.[1] This ongoing Kalman filter cycle can be divided into two groups of equations

1. Time update equations

$$\hat{x}_k^- = A\hat{x}_{k-1} + Bu_{k-1} \tag{3.1}$$
$$P_k^- = AP_{k-1}A^T + Q \tag{3.2}$$

2. Measurement update equations

$$K_k = P_k^- H^T (HP_k^- H^T + R)^{-1} \tag{3.3}$$
$$\hat{x}_k = \hat{x}_{k-1}^- + K_k(z_k - H\hat{x}_k^-) \tag{3.4}$$
$$P_k = (I - K_k H)P_k^- \tag{3.5}$$

To test this algorithm and to learn something about applying the Kalman filter on linear systems we created an example. A detailed description of the following example can be found in appendix **??**.
We consider a linear, timeinvariant model, given by the following circuit diagram in figure 3.1. First we added process noise and measurement noise to the system output. The aim is to get an estimation of the noisy output. Therefore we applied the Kalman filter algorithm based on the equations 3.1. In the figure 3.2, above we can see the input signal and the ideal measurement of the output signal. That ideal measurement includes process noise, which can obviously not be filtered by the Kalman filtering algorithm. Below one can see the noisy measurement on the left side and the filtered output on the right side.

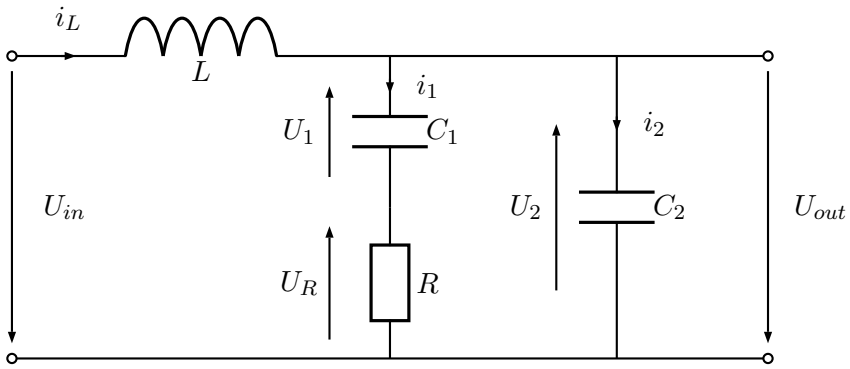

Figure 3.1: Example: Linear electrical circuit.

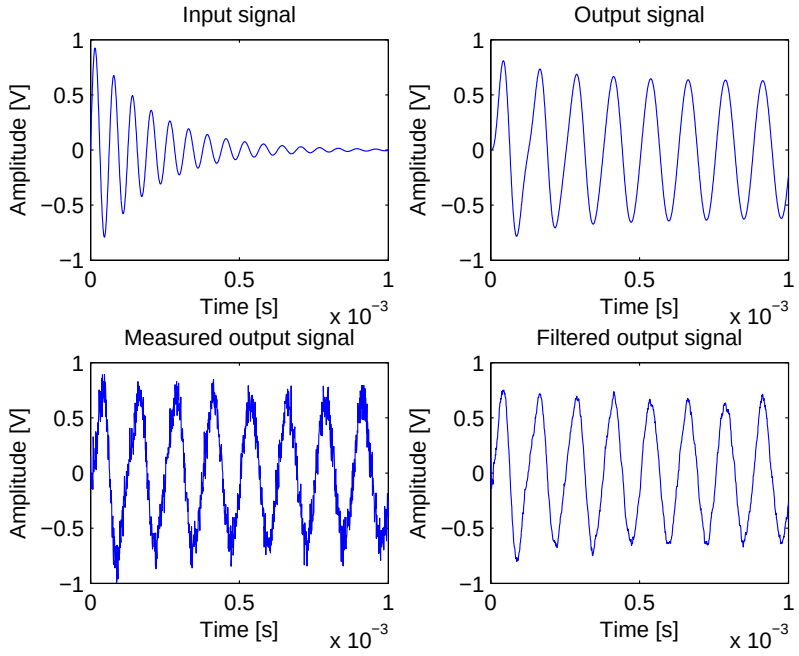

Figure 3.2: Example: Input signal, output signal, noisy output, and filtered output.

3.2 Extended Kalman Filter (EKF)

In the section above we discussed the usage of a Kalman filter on linear systems. Most systems, including the motion equations of our robots, however are nonlinear. Nevertheless linearizing our system around the current estimate still makes our Kalman filter useful and leads to the concept of the extended Kalman filter [1]. The recursive equations are quite similar to those of the linear Kalman filter

1. Time update equations

$$\hat{x}_k^- = f(\hat{x}_{k-1}, u_{k-1}, 0) \tag{3.6}$$
$$P_k^- = A_k P_{k-1} A_k^T + W_k Q_{k-1} W_k^T \tag{3.7}$$

2. Measurement update equations

$$K_k = P_k^- H_k^T (H_k P_k^- H_k^T + V_k R_k V_k^T)^{-1} \tag{3.8}$$
$$\hat{x}_k = \hat{x}_{k-1}^- + K_k(z_k - h(\hat{x}_k^-, 0)) \tag{3.9}$$
$$P_k = (I - K_k H_k) P_k^- \tag{3.10}$$

The function $f(\hat{x}_k, u_k, w_k)$ contains the system's nonlinear dynamics where w_k represents the current process noise and $h(\hat{x}_k, v_k)$ denotes the nonlinear state-to-output relationship with the measurement noise v_k. Furthermore the matrices A_k, H_k, W_k and V_k are linearizations, i.e. partial drivatives, of their respective functions at time step k

$$A_{i,j} = \frac{\partial f_i}{\partial x_j}(\hat{x}_{k-1}, u_{k-1}, 0) \tag{3.11}$$
$$H_{i,j} = \frac{\partial h_i}{\partial x_j}(\hat{x}_{k-1}, u_{k-1}, 0) \tag{3.12}$$
$$W_{i,j} = \frac{\partial f_i}{\partial w_j}(\tilde{x}_k, 0) \tag{3.13}$$
$$V_{i,j} = \frac{\partial h_i}{\partial v_j}(\tilde{x}_k, 0) \tag{3.14}$$

We dropped the fact that all matrices should have a subscript k and that they are allowed be different at each time step. Again, before applying the theory to our simulation, we created a generic example of a nonlinear system. Further details can be looked up in section **??**. We did essentially the same as we did for the linear system: We simulated the ideal system, then added process and measurement noise and in the last step checked whether we could get rid of our artifically added measurement noise. The results for a sample run on MATLAB are shown in figure 3.3 below

The two pictures above show the sinusoidal input on the left and the ideal output, i.e. without process and measurement noise, on the right. Thereunder we can see the

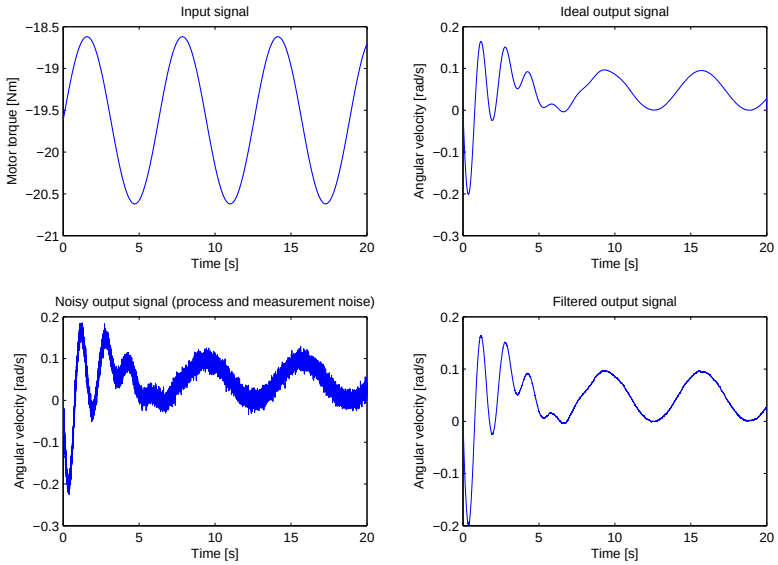

Figure 3.3: Example: Input signal, output signal, noisy output, and filtered output.

plots of the noisy signal and the signal after it has been filtered by the extended Kalman filter. We can conclude that the extended Kalman filter too provides an acceptable performance if our measurement noise is not too big.

4 Kalman Filtering: Implementation in the MATLAB Environment

4.1 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*eye(2), [0;0]; [0 0 Noise.process.dir]];
    R = [Noise.measure.pos*eye(2), [0;0]; [0 0 Noise.measure.dir]];
    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. Therefore 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.

4.2 Estimation of the Ball

5 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]

5.1 Sensors of the Robots

The robots comes with a camera with a defined field of view. There are no other sensors or informations with 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.

5.2 Principles of Sensor Fusion

There are several methods

5.3 Dempster-Shafer

6 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.