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 **??**.

But why do we actually need estimation of movements, is it not enough to simply take the measurements we get and use them as estimates? The answer to that question is no for several reasons. Since the measurements are gathered by the built-in cameras of the robots, they are subject to measurement noise. This noise is certainly much bigger than the noise of a global eye which would see the whole playing field. So one reason to use estimation is the need to get rid of this measurement noise. An other reason concerns the availability of measurements. It is easily possible that objects on the field will be measured more than one time or not at all within one timestep. For the former case it is important that we have a meaningful measurement fusion such that we can take advantage of the multiplicity of measurements. For the later case we would like to keep track of objects even if measurements are not available. Therefore we need some kind of prediction. For the prediction task on the other side we have the problem of asymmetric information distribution. That means that input parameters, like the velocity of objects, are perfectly known for own team players but not for the opposite team and the ball. A filter task gets even harder with these constraints.

Our approach is now to solve all of these problems with a single estimation and prediction module. One can think of a block model with an input and an output. The input is the data coming from all four robot cameras which we have at our disposal. This data is then processed by the estimation unit; we estimate the position of our own robots as well as the position of other robots and the ball. This step consists of fusing and filtering all incoming measurements. The output will finally be a global map of the current situation on the playing field, so we try to simulate a global eye by using the equipment and possibilities our robots have. The following chapters of this documentation will explain step by step how we tried to solve this task.

2 Simulation

One main aspect of this group work among the design of a Kalman filter was the simulation of Nao soccer match in MATLAB. It don't have to simulate a real scenario, but should provide data to test the filter. Therefore we made four independent modules which build the framework for the simulation. The four parts contain functions concerning the dynamics of the robots and ball, the measurements and the plots. Since these parts are constructed in a modular fashion, we can change one module without influencing the others. 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. Every step the in 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 dynamics
      Ball dynamics
      Robot measurement
      Ball measurement
      Robot filtering
      Ball filtering
      Plot environment and objects
      // Debugging tool

End Loop
```

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

about plots is mainly a documentation of the draw functions for the field and objects we made to illustrate the simulation, while the section about robots and ball also covers the mathematical background.

2.1 Plots

Plotting a step in the simulation can be divided essentially in two parts. One part contain the static part, such as the field or goals and the other part all dynamic objects, e.g. the robots and the ball.

Plot of the field - `plot_env()`

All graphical features concerning the playing field are implemented by the function `plot_env()`. It is subdivided in two functions, which draw the field and, as a neat add on, the scorecounter seperately. The size of the field and of the goals we took from the RoboCup Nao Rule Book 2011 and are defined in the global Struct `Field`.

```
global Field;
    Field.width = 6; %[m]
    Field.height = 4; %[m]
    Field.penaltyAreaWidth = 0.6; %[m]
    Field.penaltyAreaHeight = 2.2; %[m]
    Field.goalHeight = 1.4; %[m]
    Field.goalWidth = 0.02; %[m]
    Field.centerCircleRadius = 0.6; %[m]
    Field.pointRadius = 0.05; %[m]
    Field.penaltyPointLocation = 1.8; %[m]
```

To draw the field we mostly use built-in MATLAB commands such as `rectangle()` or `line()`. Only for drawing circles we made the function

```
draw_circle(x,y,r,color,isFilled)
```

where

`x,y` is position of the origin of the circle

`r` is radius of the circle

`color` is a string for the color of the circle, e.g. `'k','r'`

`isFilled` is a boolean variable,
`= 0` only the contour of the circle will be plotted in `color`,
`= 1` the circle will be plotted filled in `color`

The playing field after the execution of `plot_env()` is displayed in figure 2.1.

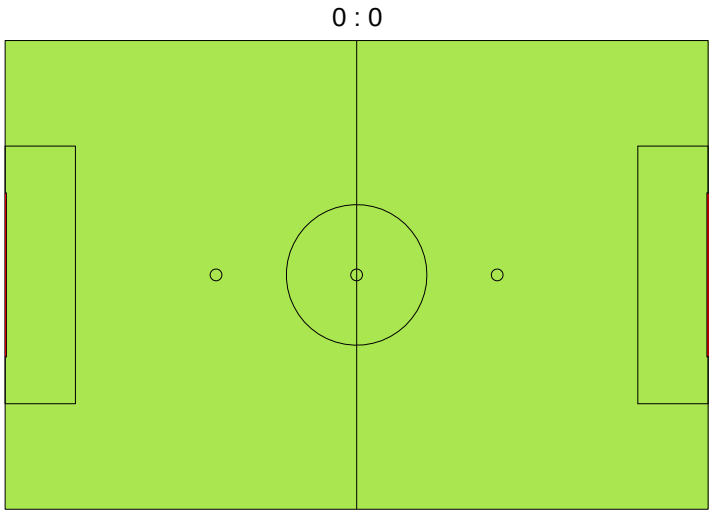


Figure 2.1: Playing field with `plot_env()`.

Plot of the robots and ball - `plot_objects(Robot,Ball,style)`

To plot robots and ball we made the function

`plot_objects(Robot,Ball,style)`

where

`Robot` is a struct of robotstructs as described in section 2.2

`Ball` is a ballstruct as described in section 2.3

`style` is a string defining the rooter shape, which contain an arbitrary number and order of following characters

`'O'` Draws circles in true size
`'@'` Same as above, but filled
`'-'` Includes the direction indicator
`'+'` Draws crosses
`'#'` Enumerate Robots
`'V'` Sight of view for blue team only
`'Va'` Sight of view all robots

and one of the following character to specify the robot color
`'t'` Team specific colors (blue and magenta)
`'r','k','b',...` other MATLAB colors

This different shape styles allows to display different robots, e.g. the real and the filtered robots, in a clear manner. Figure 2.2 shows the playing filed after initialization and execution of `plot_env()` and `plot_objects(Robot,Ball,'@-t')`. Where `Robot` and `Ball` are the standard initialization structs as described in Section 2.2 and 2.3.

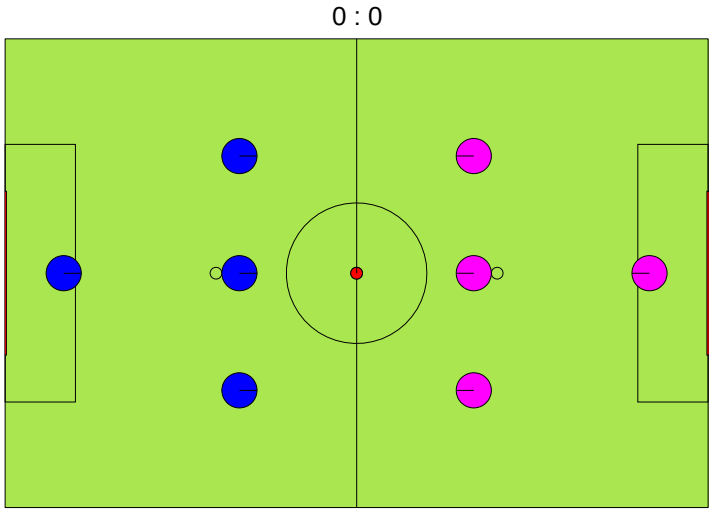


Figure 2.2: Playing field after initialization and `plot_objects(Robot,Ball,'@-t')`.

2.2 Robots

A important part of the simulation is the adequate depiction and behaviour of all eight robots. The task can be separated in two components: After the robots are initialized, their new positions on the field, according to their motion equations, are computed.

Initialization

The initialization of the robots is quite simple. The function `robot_standard_init()` creates a struct of eight structs with the essential informations for every robot, i.e. its horizontal and vertical position, its direction and its team affiliation. Following tabular depicts the structure of such a struct.

Robot(1)	Robot(2)	···	Robot(8)
.color	.color		.color
.x	.x		.x
.y	.y		.y
.dir	.dir		.dir

Furthermore the initialization function defines some robot specific parameters in the global struct `RobotParam`. These are characteristics such as the robot's radius or its maximum possible angular change for one time step. 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 additionally parameters of sight distance as well as its angle of sight are set in the initialization function. The MATLAB code below shows our assumptions for these parameters

```
global RobotParam dt;
RobotParam.radius = 0.15; %[m]
RobotParam.velocity = 0.1 * dt; %[m/step]
RobotParam.changeOfDir = 0.1 * 2*pi * dt; %[rad/step]
RobotParam.sightDistance = 2.5; %[m]
RobotParam.sightAngle = pi./6; %[rad]
```

Dynamics of the robots - Mathematical model

To compute the next step of the robots we used the following simple non-linear discrete time motion equations

$$x_{i_{k+1}} = x_{i_k} + v_{i_k}\cos(\varphi_{i_k})\Delta t \tag{2.1}$$
$$y_{i_{k+1}} = y_{i_k} + v_{i_k}\sin(\varphi_{i_k})\Delta t \tag{2.2}$$
$$\varphi_{i_{k+1}} = \varphi_{i_k} + d\omega_{i_k}\Delta t \tag{2.3}$$

with the boundary conditions, that the robots have to be on the playing field and can't covering the same disk as another robot:

$$(x_{i_k}, y_{i_k}) \in [-3 + r_r, 3 - r_r] \times [-2 + r_r, 2 - r_r] \quad \forall k, \forall i \in \{1, 2, \ldots, 8\}$$

$$\sqrt{(x_{i_k} - x_{j_k})^2 + (y_{i_k} - y_{j_k})^2} =: d_{ij} \geq 2r_r \quad \forall k, \forall i, \forall j \neq i$$

Where (x_i, y_i) is the position and φ_i the direction of robot i and r_r the radius of a robot. The velocity v_i and the change of direction $d\omega_i$ are inputs to the system.

Random Collision Roboter To satisfy the boundary conditions we first used robots which rebounce when they collide into the sidelines and into each other. If they hit each other they swap direction and if they touch a sideline, they reflect after the law of reflection

$$\Theta_{\text{in}} = \Theta_{\text{out}}.$$

Those dynamics are implemented in `robot_rand_step(Robot)` with a random inputs $d\omega_{i_k} \sim \mathcal{N}(0, 0.2\pi \cdot dt)$ and constant velocities $v_{i_k} = v = 0.1m/s$.

But 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 a normal Kalman filter.

Random Potential Roboter The function `robot_randpot_step(Robot,Ball)` eliminates both issues because it doesn't use collision detection but collision avoidance. If the distance of two robots falls below a given radius r_a, 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. The force in a electric field is given by

$$F = m\ddot{x} = C\frac{1}{x^2}.$$

Integration results in

$$\dot{x} = \frac{dx}{dt} = \tilde{C}\frac{1}{x}.$$

Or the same described in vector notation:

$$\vec{F} = C\frac{\vec{r}}{|\vec{r}|^3} \qquad \Rightarrow \qquad \dot{\vec{r}} = \frac{d\vec{r}}{dt} = \tilde{C}\frac{\vec{r}}{|\vec{r}|^2}$$

As you can see in the code 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
           % Distance between robot i and j
           r = sqrt((Robot(j).x-Robot(i).x).^2+(Robot(j).y-Robot(i).y).^2);

           % Potential function to compute repulsion between robots.
           if ( i~=j && (r <= ra) )
               dx = dx + C./r.^3*(Robot(i).x-Robot(j).x);
               dy = dy + C./r.^3*(Robot(i).y-Robot(j).y);
           end
        end
```

A related potential is also assigned to the sideline, if a robot is under a given distance r_f to it. With those potentials, the change of directions is now continuous, which makes tracking possible for our Kalman filter. Additionally to the collision avoidance the robots are attracted to the ball if it's near them, like it had a negative charge. This leads to some more ball contacts than before.

2.3 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 different than the robot also the velocity.

Ball
.x
.y
.dir
.velocity

These parameters are set by executing `ball_init()`. This function also defines the ball's radius, its initial velocity and the friction towards the ground in the global struct `BallParam`. In our simulation we used following parameters for the ball

```
global BallParam dt;
BallParam.radius = 0.05; %[m]
BallParam.velocity = 0.4 *dt; %[m/step]
BallParam.friction = 0.995;
```

Dynamics of the ball - Mathematical model

The dynamics of the ball are fairly analog to those of the robots. Only that the velocity is no longer an input but now a state of the system, there is some friction between the ball and ground and the direction is constant. The the non-linear motion equation for the ball are therefore

$$x_{k+1} = x_k + v_k \cos(\varphi_k)\Delta t \tag{2.4}$$
$$y_{k+1} = y_k + v_k \sin(\varphi_k)\Delta t \tag{2.5}$$
$$\varphi_{k+1} = \varphi_k \tag{2.6}$$
$$v_{k+1} = \varrho \cdot v_k \tag{2.7}$$

With the boundary conditions, that the ball has to stay on the playing field and can't be under a robot

$$(x_k, y_k) \in [-3 + r_b, 3 - r_b] \times [-2 + r_b, 2 - r_b] \quad \forall k,$$

$$\sqrt{(x_k - x_{ik})^2 + (y_k - y_{j_k})^2} =: d_{bi} \geq r_r + r_b \quad \forall k, \forall i$$

To satisfy these conditions, we used for the sidelines again the law of reflection. And if the ball collides with one of the robots, it regains its initial velocity and bounces away, perpendicular to the robot's position. Like the robot had kicked the ball. These dynamics are implemented in the function `ball_step(Ball)`.

2.4 Noise

Up to this point we computed the behavior of an ideal robot and ideal ball. 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. Also we added some process noise to the dynamics of robots and ball, which can't be filtered. Hence the motion equations get extended as followed

$$\begin{aligned} x_{k+1} &= f(x_k, u_k) + w_k \\ z_k &= x_k + v_k \end{aligned}$$

where x is the state, z the output, $f()$ the nonlinear system of equations in (2.1-2.3) or (2.4-2.7), w the process noise vector and v the measurement noise vector.

The process noise w we added in the specific step functions for robots and ball as white Gaussian noise. The measurement noise is white Gaussian, too. But additionally we made two different implementations, which determine if there is a measurement at all or the corresponding parameters are just dropped.

Random measurement The function `robot_random_measure(Robot)` simply adds white Gaussian noise to the position and the direction of every robot. Additionally with a certain possibility there is no measurement at all. `ball_random_measure(ball)` does the same for the ball. 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. But this scenario is not realistic in a RoboCup soccer match, because only the robots themselves can gain visual information. Figure **??** shows random measurement as black crosses.

Sight of view measurement The functions `robot_sight_of_view_measure(Robot)` for the robots and `ball_sight_of_view_measure(Ball)` for the ball solve this problem. According to a sight of view, 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. Figure **??** shows measurements generated with these functions.

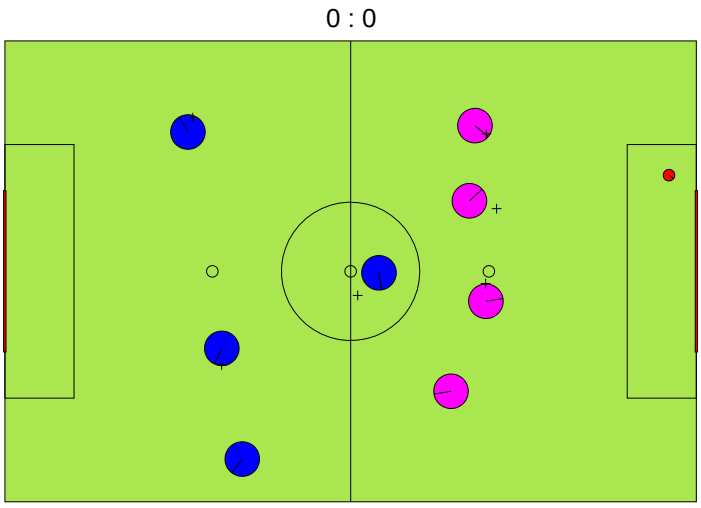


Figure 2.3: Robots on field and Random measurement (+).

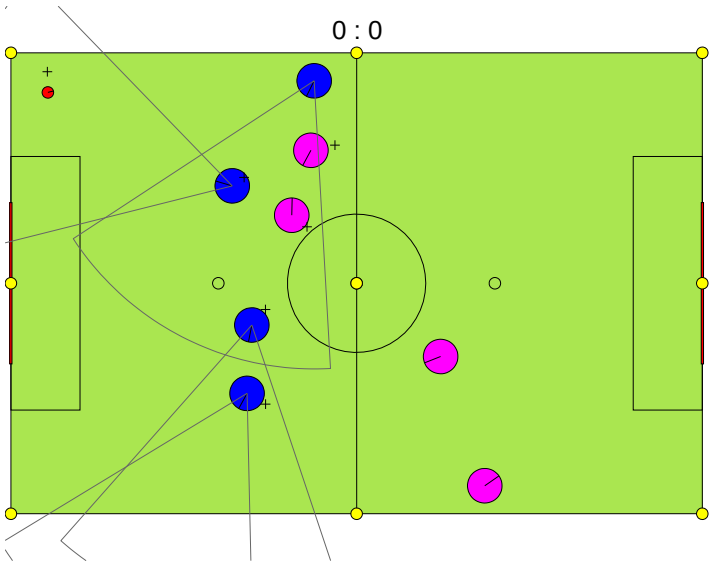


Figure 2.4: Robots on field and measurement in sight of view.

3 Theoretical Basics of Kalman Filtering

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

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

2. Measurement update equations

$$\begin{aligned} K_k &= P_k^- H^T (HP_k^- H^T + R)^{-1} \\ \hat{x}_k &= \hat{x}_{k-1}^- + K_k(z_k - H\hat{x}_k^-) \\ P_k &= (I - K_k H)P_k^- \end{aligned} \tag{3.2}$$

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, time invariant 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 and 3.2. 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 algorithm. Below one can see the noisy measurement on the left side and the filtered output on the right side.

3.2 Extended Kalman Filter (EKF)

In the section 3.1 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

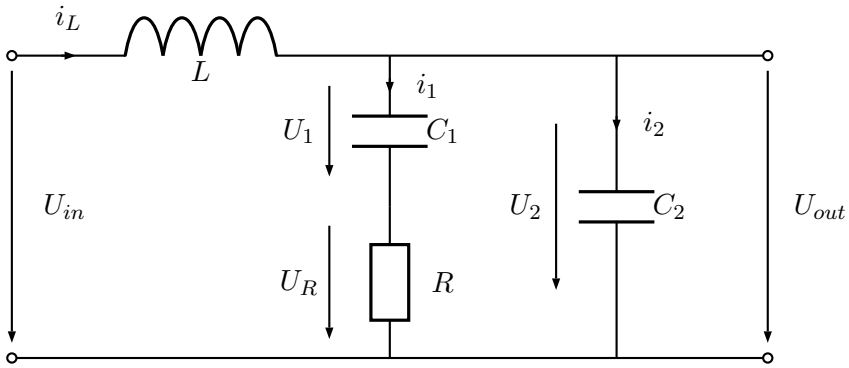


Figure 3.1: Example: Linear electrical circuit.

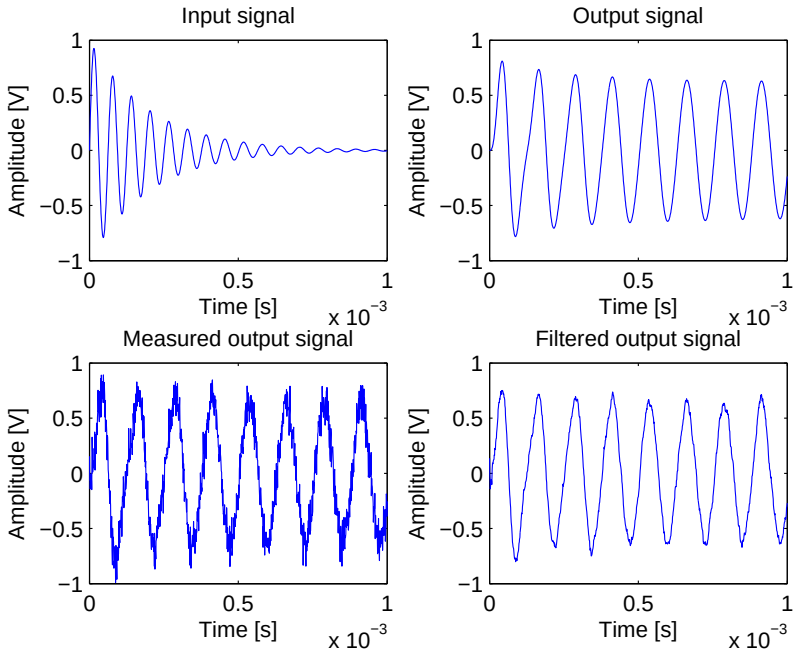


Figure 3.2: Example: Input signal, output signal, noisy output, and filtered output.

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

$$\begin{aligned} \hat{x}_k^- &= f(\hat{x}_{k-1}, u_{k-1}, 0) \\ P_k^- &= A_k P_{k-1} A_k^T + W_k Q_{k-1} W_k^T \end{aligned} \tag{3.3}$$

2. Measurement update equations

$$\begin{aligned} K_k &= P_k^- H_k^T (H_k P_k^- H_k^T + V_k R_k V_k^T)^{-1} \\ \hat{x}_k &= \hat{x}_{k-1}^- + K_k (z_k - h(\hat{x}_k^-, 0)) \\ P_k &= (I - K_k H_k) P_k^- \end{aligned} \tag{3.4}$$

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.

$$A_{i,j} = \frac{\partial f_i}{\partial x_j}(\hat{x}_{k-1}, u_{k-1}, 0) \tag{3.5}$$

$$H_{i,j} = \frac{\partial h_i}{\partial x_j}(\hat{x}_{k-1}, u_{k-1}, 0) \tag{3.6}$$

$$W_{i,j} = \frac{\partial f_i}{\partial w_j}(\tilde{x}_k, 0) \tag{3.7}$$

$$V_{i,j} = \frac{\partial h_i}{\partial v_j}(\tilde{x}_k, 0) \tag{3.8}$$

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.

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 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.

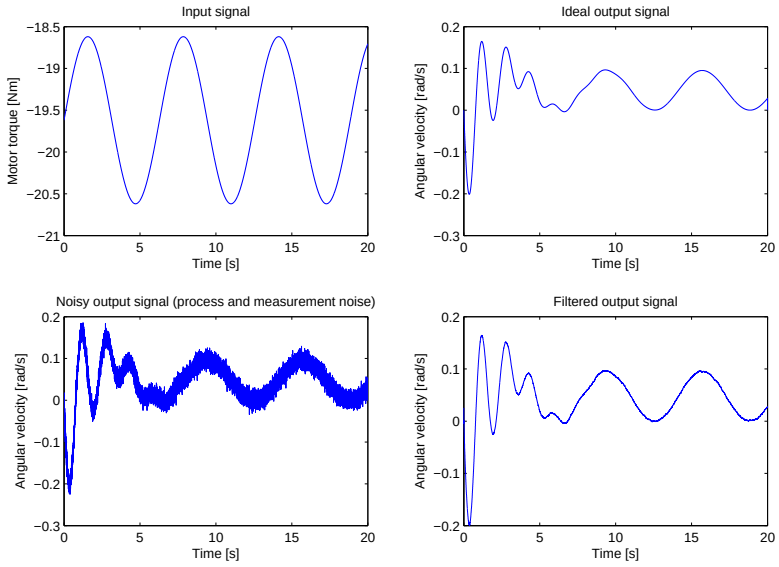


Figure 3.3: Example: Input signal, output signal, noisy output, and filtered output.

4 Implementations of the Kalman Filter in MATLAB

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 instead of a linear one. The function `robot_extended_kalman_filter(robotMeasure,robotEstimate, mValues,eValues, dOmega,v,vPink,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.

```
                              % team's steering inputs.

%———————— 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);
    xApriori = [0;0;0];
```

4.1.1 Estimation with known inputs (blue robots)

The core of our filter is the prediction of the future states (based on the current states and inputs) and their correction (based on measurements with white noise). In our case, the estimated states are x, y and dir.

```
      % Time update (predict)
      xApriori(1) = robotEstimate(i).x+cos(robotEstimate(i).dir)*v(i);
```

```
xApriori(2) = robotEstimate(i).y+sin(robotEstimate(i).dir)*v(i);
xApriori(3) = robotEstimate(i).dir+dOmega(i);
Pstep(:,:,i) = A*P(:,:,i)*A'+W*Q*W';

% Measurement update (correct)
if isnan(robotMeasure(i).x * robotMeasure(i).y * robotMeasure(i).dir)
    estimates = xApriori;   % Measurement drop
    Knorm(i) = NaN;
else
    z = [robotMeasure(i).x; robotMeasure(i).y; robotMeasure(i).dir];
    K(:,:,i) =  (Pstep(:,:,i)*H')/(H*Pstep(:,:,i)*H'+V*R*V');
    Knorm(i) = norm(K(:,:,i));
```

In order to handle dropped measurements (mapped to NaN), the states are predicted even if there is no correction data available. The arising problems from this approach are discussed in the following section.

4.1.2 Quickly regaining confidence in measurements

Missing measurements over a large time span will cause divergence between estimated values and actual values, since the state cannot be corrected. Regaining visual information from robots would 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:

```
xAbs = abs(eValues(1,1,i) - mValues(1,1,i));
yAbs = abs(eValues(2,1,i) - mValues(2,1,i));
dirAbs = abs(eValues(3,1,i) - mValues(3,1,i));
probX(i) = erfc(xAbs/(sqrt(RobotMeasure(i).sigma)));
probY(i) = erfc(yAbs/(sqrt(RobotMeasure(i).sigma)));
probDir(i) = erfc(dirAbs/(sqrt(RobotMeasure(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.

4.1.3 Estimation without input information (pink robots)

Whereas the blue team's "steering input" is assumed to be known, we have to estimate it for our opponent, the pink team. This task is done by `input_approximation(RobotMeasure,mValues,v).m`. The goal is to reconstruct the change of the angular position and the velocity by looking only 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.

```
dOmegaStep(i-4) = RobotMeasure(i).dir - mValues(3,1,i);
vNew = dt*sqrt((RobotMeasure(i).x - mValues(1,1,i)).^2 + ...
    (RobotMeasure(i).y - mValues(2,1,i)).^2);
vStep(i-4) = (v(i-4)*(windowSize-1)+vNew)/windowSize;
```

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.

4.2 Estimation of the Ball

The estimation of the ball is done by `ball_ekf(BallOe, BallMeasure, Poe).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. Whereas robots can turn slowly, the ball moves in straight lines (within process noise) across the playing field. Whenever it bounces, the balls direction changes rapidly. Since such a "discrete event" can't be covered by the ball's dynamic equations, we handle it through the following if-statement:

```
xOe = [BallOe.x ; BallOe.y ; BallOe.dir];
xMeasure = [BallMeasure.x ; BallMeasure.y ; BallMeasure.dir;];
```

```
if(norm(xOe(1:2) - xMeasure(1:2))>0.5 || abs(xOe(3) - xMeasure(3))>pi/2)
    Poe = eye(4);
end
```

Thus, 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 a boundary.

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 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.

5.2 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

$$\hat{X}_i = X + w_i. \tag{5.1}$$

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

$$Y \quad = \quad a_1\hat{X}_1 + a_2\hat{X}_2 + \ldots + a_n\hat{X}_n, \tag{5.2}$$

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

$$E[Y] = X \qquad \text{and} \qquad E\left[(Y - X)^2\right] \rightarrow \text{minimal.} \tag{5.3}$$

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

$$a_1 + \ldots + a_n = 1 \qquad \Longleftrightarrow \qquad G(a_1, \ldots, a_n) = a_1 + \ldots a_n - 1. \tag{5.4}$$

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 (5.4) into account leads to the second condition

$$F(a_1, \ldots, a_n) = a_1^2\sigma_1^2 + \ldots + a_n^2\sigma_n^2 \quad \rightarrow \quad \text{minimal.} \tag{5.5}$$

The generic approach is now to solve equation (5.5) with $\nabla F(a_1, \ldots, a_n) = 0$. This attempt fails, because we also have to meet the side condition from equation (5.4). Therefore we use the method of Lagrange multipliers [5]. The equations we have to solve, are shown below

$$\nabla F(a_1, \ldots, a_n) = \lambda \nabla G(a_1, \ldots, a_n), \qquad G(a_1, \ldots, a_n) = 0. \tag{5.6}$$

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

$$a_i = \frac{\sigma_i^{-2}}{\sum_{j=1}^{n} \sigma_j^{-2}}. \tag{5.7}$$

In a final step, we are interested in the statistical properties of our fused measurement. The mean is already given by condition (5.3). Since Y is a Gaussian random variable we only need the covariance σ_Y^2 to fully describe Y. We get

$$\sigma_Y^2 = E\left[Y^2\right] - E[Y]^2 = \frac{1}{\sum_{j=1}^{n} \sigma_j^{-2}}. \tag{5.8}$$

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. The effect on the covariances with fusion is similar to the effect of a parallel connection of several resistors in an electrical circuit. Regardless of the amount of resistors we connect in parallel: The overall resistance of the whole circuit will always be smaller than the smallest resistance in the circuit.

5.3 The implementation in MATLAB

The sensor fusing methods for our simulation are embedded in the measurement functions `robot_sight_of_view_measure.m` and `ball_measure.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

```
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);
    end
end
```

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.

```
if k ~= 0
    RobotMeasure(i).x = x./k;
    RobotMeasure(i).y = y./k;
    RobotMeasure(i).dir = dir./k;
    RobotMeasure(i).sigma = 1./k;
end
```

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.

6 Challenges and Outlook

Several problems which can occur in a real Nao soccer match are covered by our design of the model such as measurement drops, measurement fusion or the fact that inputs of the opposing team are unknown. However there are still practical challenges which we do not encounter in our work. Maybe the most important challenge is the provision of measurements and their related covariances. Up to this point we simply assumed that there is an algorithm which computes the positions and directions of objects on the playing field. We didn't care about the fact whether this can be done in a reliable way or within a reasonable space of time. This task was beyond the scope of this work and certainly could be the subject of another whole group work for the people in the ETHZ RoboCup Team which are occupied with perception.

Other problems arise from the restrictions of our chosen model, for example the assumption that there are only three scenarios under which measurements are gathered. In reality we are probably faced with much more possibilities, that do not only depend on whether a robot sees a landmark or not but also on the actual sight distance to an object or the head angle of a robot. Furthermore the measurement noise may be coloured and not white, so our model would be inaccurate in describing the uncertainty in the given system. Those kind of issues can be summarized as "lack of detail", the approximation to reality is not close enough. This can be solved by a better and deeper analysis of the system.

Last but not least there are still several things, that were only partly or not at all considered in our work and which have huge potential for the further localization task. One main aspect here is certainly the practical application of the simulation on the Nao Platform. Not only that it is of course the main goal to have a working estimation and prediction for a real Nao soccer match but also that further improvements of the estimation algorithms will only be possible if they are tested in real environment with real constraints. The modularity of our simulation allows improvements in specific areas concerning estimation. For example the tracking of the ball may be improved too by replacing the extended Kalman filter by a so called particle filter, so switching from deterministic to probabilistic methods. It is also imaginable to have hybrid forms like it is partly done for the extended Kalman filter of the robots (see Ch. 4.1). As a third point one could mention the vast improvements which are possible in estimation if the robots are not acting randomly but if they have a strategy that governs their movements. In terms of localization that would mean that blue robots actively seek getting good measurements and that for example losing the current position of the ball for long

periods is not possible anymore. More extensions of the current basic configuration such as better input approximation of enemy robots, the prediction of enemy movements or better handling of discrete events are also thinkable and can have positive effects on the overall performance.

A Organisation

A.1 Project description

A.2 MATLAB-Code Style Guidelines for our project

In the MATLAB-Script `RoboCupSim.m` and in the associated functions we tried to use following naming convention:

- **Variables**
  - in general: *mixed case starting with lower case*
    e.g. `goalHeight, dt`
  - representing a number of objects: *prefix n*
    e.g. `nSteps`
  - iterator variables: *prefix i,j,k*
    e.g. `iStep`
  - boolean variables: *prefix is*
    e.g. `isValid`
- **Constants**: *all uppercase using underscore to separate words*
  e.g. `MAX_ITERATIONS`
- **Structures**: *begin with a capital letter*
  e.g. `Field.width, Noise.Process.pos`
- **Functions**: *lower case with underscore to separate words*
  e.g. `plot_objects, ball_step`

B Examples

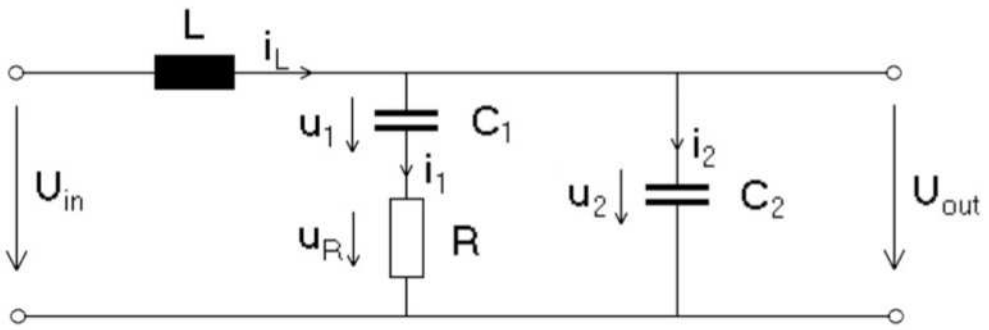
B.1 Kalman Filtering of Linear System

Group project - Linear model

Daniel Gilgen, David Lehnen, Matthias Roggo, Fabio Marti

March 12, 2012

We are considering a linear, timeinvariant model, given by the following circuit diagram



This is obviously a system of order 3 with the states $x_1 = i_L$, $x_2 = u_1$ and $x_3 = u_2$, the input $u = U_{\text{in}}$ and the output $y = U_{\text{out}}$. The system's equations are given by

$$L \cdot \frac{di_L}{dt} = u_L = U_{\text{in}} - u_2$$

$$\Longrightarrow \qquad \frac{di_L}{dt} = \frac{U_{\text{in}}}{L} - \frac{u_2}{L}$$

$$C_1 \cdot \frac{du_1}{dt} = i_1 = u_R \cdot R = (u_2 - u_1) \cdot R$$

$$\Longrightarrow \qquad \frac{du_1}{dt} = \frac{R}{C_1} \cdot u_2 - \frac{R}{C_1} \cdot u_1$$

$$C_2 \cdot \frac{du_2}{dt} = i_2 = i_L - i_1 = i_L - (u_2 - u_1) \cdot R$$

$$\Longrightarrow \qquad \frac{du_2}{dt} = \frac{i_L}{C_2} - \frac{R}{C_2} \cdot u_2 + \frac{R}{C_2} \cdot u_1$$

This leads to the following state space representation

$$\begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \\ \dot{x}_3 \end{bmatrix} = \begin{bmatrix} 0 & 0 & -\frac{1}{L} \\ 0 & -\frac{R}{C_1} & \frac{R}{C_1} \\ \frac{1}{C_2} & \frac{R}{C_2} & -\frac{R}{C_2} \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + \begin{bmatrix} \frac{1}{L} \\ 0 \\ 0 \end{bmatrix} \cdot u$$

$$y = \begin{bmatrix} 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$$

What we have now is a continuous time state space model of the form

$$\dot{x}(t) = \bar{A}x(t) + \bar{B}u(t)$$
$$y(t) = \bar{C}x(t)$$

Now we can assume, that all electrical devices are not ideal or that there are external disturbances like fluctuations in temperature or air moisture, such that their behaviour is not ideal. These factors will lead to process noise $w(t)$. Furthermore we will measure the output voltage U_{out} with a voltmeter, which will not measure the values exactly or which has an inappropriate resolution. This will add some measurement noise $v(t)$ to our model. Our new state space representation will be

$$\dot{x}(t) = \bar{A}x(t) + \bar{B}u(t) + w(t)$$
$$y(t) = \bar{C}x(t) + v(t)$$

with

$$E[w(t)w(\tau)] = Q_e\delta(t-\tau)$$
$$E[v(t)v(\tau)] = R_e\delta(t-\tau)$$

so $w(t)$ and $v(t)$ are white Gaussian noise processes. Since we assume that our model is quite reliable and there are only few external disturbances, the process noise will be much smaller that the measurement noise. Reasonable choices for Q_e and R_e could be

$$Q_e = \begin{bmatrix} 10^{-4} & 0 & 0 \\ 0 & 10^{-4} & 0 \\ 0 & 0 & 10^{-4} \end{bmatrix}, \qquad R_e = 10^{-1}$$

The last step is to convert this system into a discrete time linear system. The state space representation for such a system is given by

$$x_{k+1} = Ax_k + Bu_k + w_k$$
$$y_k = Cx_k + v_k$$

The matrices Q_e and R_e will stay the same. All other matrices are given by

$$A = e^{\bar{A}T}, \qquad B = \int_0^T e^{\bar{A}(T-\tau)}\bar{B}\, d\tau, \qquad C = \bar{C}$$

where T represents the sampling rate of our discrete time model. By choosing appropriate values for L, C_1, C_2 and R, we finally have enough information to build a Kalman filter for this linear system.

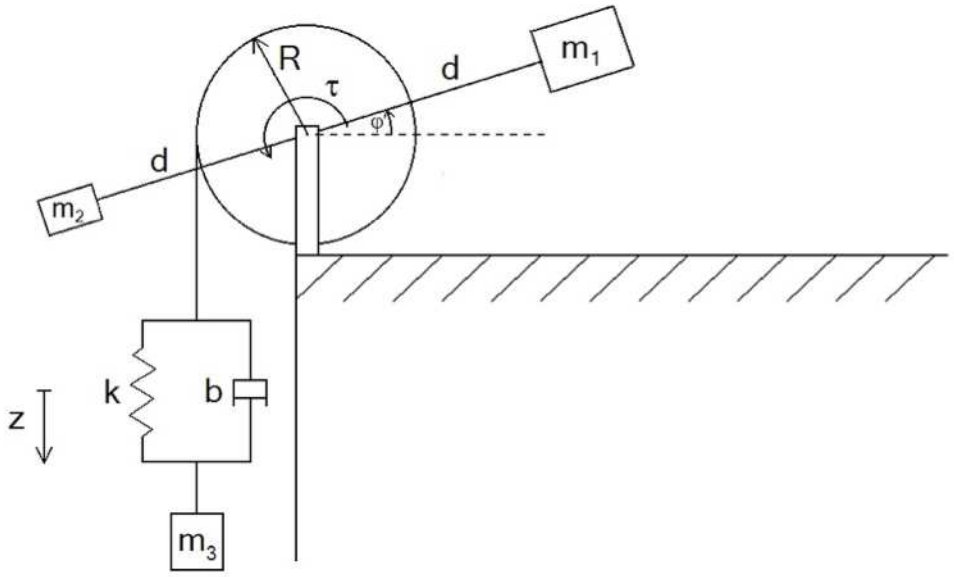
B.2 Kalman Filtering of Nonlinear System

Group project - Nonlinear model

Daniel Gilgen, David Lehnen, Matthias Roggo, Fabio Marti

April 22, 2012

Consider the following mechanical system



We assume now that the wheel doesn't have a mass, that m_1 has a moment of inertia of $J_1 = c_1 m_1$ and that m_2 has $J_2 = c_2 m_2$ with respect to the wheel's hub. Both masses are modelled as dots with distance d to the hub. Furthermore we are assuming that the force of the damper is given by $F_b = b \cdot \Delta z$. Using Newton's laws we obtain the equations

$$m_3 \ddot{z} = m_3 g + k(\varphi R - z) + b(\dot{\varphi} R - \dot{z})$$

$$(J_1 + J_2)\ddot{\varphi} = \tau + gd\cos(\varphi)(m_2 - m_1) + k(z - \varphi R) + b(\dot{z} - \dot{\varphi} R)$$

By definig the states $x_1 = z$, $x_2 = \dot{z}$, $x_3 = \varphi$ and $x_4 = \dot{\varphi}$, the input $u = \tau$ and the output $y = \dot{\varphi}$ we find the following differential equations

$$\dot{x}_1 = x_2$$

$$\dot{x}_2 = g + \frac{k}{m_3}(x_3 R - x_1) + \frac{b}{m_3}(x_4 R - x_2)$$

$$\dot{x}_3 = x_4$$

$$\dot{x}_4 = \frac{1}{c_1 m_1 + c_2 m_2}\left(u + gd\cos(x_3)(m_2 - m_1) + k(x_1 - x_3 R) + b(x_2 - x_4 R)\right)$$

$$y = x_4$$

In a next step we transform our continuous time system into a discrete time system. We use a simple Euler step for this task

$$\dot{x} \approx \frac{x_{k+1} - x_k}{T}$$

where T is the sampling period. The discrete system's equations now look as follows

$$\begin{aligned}
x_{1,k+1} &= x_{1,k} + T \cdot x_{2,k} \\
x_{2,k+1} &= x_{2,k} + T \cdot \left(g + \frac{k}{m_3}(x_{3,k}R - x_{1,k}) + \frac{b}{m_3}(x_{4,k}R - x_{2,k}) \right) \\
x_{3,k+1} &= x_{3,k} + T \cdot x_{4,k} \\
x_{4,k+1} &= x_{4,k} + T \cdot \left(\frac{1}{c_1 m_1 + c_2 m_2} (u_k + gd\cos(x_{3,k})(m_2 - m_1) + k(x_{1,k} - x_{3,k}R) + b(x_{2,k} - x_{4,k}R)) \right) \\
y_k &= x_{4,k}
\end{aligned}$$

Since we are dealing with nonideal effects, caused by model uncertainties or measurement errors, we add process noise w_k and measurement noise v_k to our model. We will get equations of the form

$$\begin{aligned}
x_{k+1} &= f(x_k, u_k, w_k) \\
y_k &= h(x_k, v_k)
\end{aligned}$$

Note that x_{k+1} as well as w_k and v_k are column vectors. In our nonlinear model, w_k and v_k are realizations of white Gaussian noise processes

$$\begin{aligned}
E[w_k w_n] &= W_k Q_k W_k^T \delta[k-n] \\
E[v_k v_n] &= V_k R_k V_k^T \delta[k-n]
\end{aligned}$$

where $W_k = \frac{\partial f}{\partial w_k}(x_{k-1}, u_{k-1}, 0)$ and $V_k = \frac{\partial h}{\partial v_k}(x_k, 0)$. Note that Q_k and R_k could change every timestep k. Possible values for them are

$$Q_k = \begin{bmatrix} 10^{-6} & 0 & 0 & 0 \\ 0 & 10^{-6} & 0 & 0 \\ 0 & 0 & 10^{-6} & 0 \\ 0 & 0 & 0 & 10^{-6} \end{bmatrix}, \qquad R_k = 10^{-2}$$

We have a full description of our nonlinear system and are now able to implement an extended Kalman filter by choosing reasonable values for the system's parameters.

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.