Homework 3: State Estimation with Extended Kalman Filter

E. Todorov, CSE P590

Due June 13, 2014 (cannot be extended)

Problem statement

In this assignment you will implement a state estimator based on an extended Kalman filter (EKF) to play ping-pong. The model consists of a 2 degree-of-freedom yellow paddle that moves in the vertical (x-z) plane, and a green 3 degree-of-freedom ball. The game consists of 10 trials in which the ball will fly with a random velocity towards the wall. The goal is to move the paddle as to deflect the ball and prevent it from hitting the wall behind the paddle in each trial. The score of the game is how many paddle hits you get out of the trials.

However, in this homework you do not have direct access to the position of the ball. Instead, there is an eye at the origin that is able to give imperfect (noisy) estimates of where the ball in eye-space is. Your EKF implementation will accumulate and convert these noisy eye-space estimates into a reliable global position of the ball. This estimate can then be used to decide where the paddle should move to deflect the ball.

Part 1: Ball State Estimation

First, refresh yourself on extended Kalman filter theory and equations. See *this tutorial* for a concise derivation and summary of the equations; it is also available on the course website under Readings. Sections 3 and 4 of the tutorial summarize the equations you need to implement. We will use the notation from this tutorial.

In our case, the global state $\mathbf{x} = \left[\begin{smallmatrix}\mathbf{p}\\\mathbf{v}\end{smallmatrix}\right]$ we want to estimate is a 6-dimensional vector containing position and velocity of the ball. The eye observes only the position of the ball in azimuth/elevation/depth coordinates. This is a 3-dimensional observation vector $\mathbf{z}$. In the code you will have a function *mjGetSensor* that gives a noisy observation $\mathbf{z} + \epsilon$, where ϵ is observation noise with zero-mean Gaussian distribution with covariance matrix $\mathbf{R}$.

The mapping from ball position to (noise-less) observation is:

$$\mathbf{z} = h(\mathbf{x}) = \begin{bmatrix} \tan^{-1}(\frac{\mathbf{p}_y}{\mathbf{p}_x}) \\ \tan^{-1}(\frac{\mathbf{p}_z}{\sqrt{\mathbf{p}_x^2+\mathbf{p}_y^2}}) \\ \sqrt{\mathbf{p}_x^2 + \mathbf{p}_y^2 + \mathbf{p}_z^2} \end{bmatrix} \tag{1}$$

The simulator implements this mapping internally and then adds Gaussian noise to it.

The ball moves according to deterministic dynamics given by:

$$\mathbf{x}_k = \begin{bmatrix}\mathbf{p}_k \\ \mathbf{v}_k\end{bmatrix} = f(\mathbf{x}_{k-1}) = \begin{bmatrix}\mathbf{p}_{k-1} + \Delta\mathbf{v}_{k-1} \\ \mathbf{v}_{k-1} + \Delta\mathbf{a}\end{bmatrix} \tag{2}$$

Here $\mathbf{a}$ is the constant acceleration due to gravity, and Δ is the time step, both specified in the XML model file.

We will provide you with methods to calculate $f(\mathbf{x})$ and $h(\mathbf{x})$ as well as their Jacobians $\mathbf{J}_f$ and $\mathbf{J}_h$ mentioned in the tutorial. See code documentaion for more details. We will also provide you with initial state estimate mean $\mathbf{x}_0^a$ and covariance $\mathbf{P}_0$.

1.1 EKF Updates

It is up to you to implement the EKF update equations summarized in the tutorial so at to estimate the current state of the ball $\mathbf{x}_k^a$.

The starter code will visualize your estimate of the ball's position as a blue sphere in the scene. This is implemented as a collision-free geom which you can move anywhere you want using the new function *mjSetEstimator*. It does not affect the simulation in any way; it is merely provided to help you visualize what your estimator is doing.

1.2 Iterated EKF Updates

Once you have implemented basic updates, iterate these updates as described in section 4 of the tutorial. Choose an appropriate number of iterations to balance accuracy and speed of your estimation.

Testing mode – no direct state access

Note that in previous homeworks you simply used the *mjGetState* function to get the exact position of the objects in the scene. You are still welcome to use this function to get the ball position when debugging your algorithm, but this function will be disabled when we test your code. More precisely, when the custom field 'getstate' in the XML model file is set to 0 (as it will be when we test your code), the simulator will return 0 for the ball position and velocity. Also, it will return an error if you call *mjGetBody* or *mjGetGeom*. You can still use *mjGetState* to find out where the paddle is (although you don't really have to, because the paddle is position-controlled).

Part 2: Paddle Control

Now that you have an estimate of the global position and velocity of the ball, you must use it to move the paddle. The paddle is moved by a positional servo: you specify the desired position in the horizontal and vertical directions (with *mjSetControl*) and the simulator implements a virtual spring between the actual and desired positions of the paddle. See starter code for details. For this assignment, if your estimator works properly it is good enough to simply move the paddle to the current estimate of the ball's planar position.

This strategy should consistently get you perfect score on the test trials and noise levels we provided.

For extra credit (but not necessary to get perfect score on this homework): rather than moving the paddle to current ball's position, move it to position where you estimate the ball to hit the wall. In practice, this provides 'anticipation' and would help if the ball is moving very fast or paddle can only move slowly. Here are some hints: if you know the ball's velocity, you can calculate at what time in the future it will reach paddle's depth. The ball moves linearly in horizontal direction and in a parabolic arc vertically. So if you know impact time, you can predict where it will be at that time. It is up to you to figure out how to do this, if you wish.

Starter code

On the Catalyst assignment page you will find a package containing Mujoco physics simulation server and a starter Visual Studio project containing a Mujoco client that you will extend.

After compiling MujocoClient, start *Mujoco.exe* (the server). You will see simulation display window. Then start *MujocoClient.exe*. The client will load the model we provided. Read starter code and comments for more details. Your task is to extend the starter code to complete the assignment.

The code will run 10 game trials and output the final score for the game. Do not remove the scoring output as we will use it to grade your solutions.

Note that you may end up in situations where you are inverting a singular matrix. In this case, the matrix class will return an empty matrix result. Make sure this case is handled appropriately in your code.

You will have to run the code at different estimation noise levels (diagonal entries of $\mathbf{R}$). To change noise level, edit lines

```
R(0,0) = 0.05;
R(1,1) = 0.05;
R(2,2) = 0.30;
```

in *MujocoClient.cpp* and edit line

```
<custom name="noise" data="0.05 0.05 0.30"/>
```

in *estimator.xml* model file.

What to submit

Submit one executable along with your source code. Use whatever number of EKF iterations you found to work best. For submission, use estimation noise level 0.05 0.05 0.30.

Additionally, submit a short writeup (pdf or txt) reporting the score you got for the above noise level. What score range do you get when you set noise level to 1.0 1.0 1.0? What about 1.0 1.00 10.00?

Currently initial state covariance $\mathbf{P}_0$ is set to $\alpha\mathbf{I}$, where α is 1. What happens when you set α to 10^{-2} or 10^2?

What happens when you run EKF updates for extra iterations as opposed to just one? Do the extra iterations of EKF seem to help?

Simulation notes

As you already noticed in homework 2, the simulator is very fast, so you should insert a sleep command in your main loop in order to see the animation.

You can also play with the passive system (without connecting your executable to the simulator) as follows. Load the XML model from the File menu and switch to Sim mode (by pressing space or from the Option/Mode dialog). The ball will simply drop because its initial velocity is 0; note that we have set gravity to 2 instead of 9.81 to make the problem easier. The XML contains a number of keyframes with different initial velocities. PgUp/PgDn in the simulator cycles through the keyframes, and in this case shows the ball flying towards the wall. You can also change the keyframe index from the Mode dialog. Ctrl+Backspace resets the state to the currently selected keyframe.

Finally, if you have nothing else to do, you can play the game with the mouse. Left-DoubleClick the paddle to select it. Rotate the camera so that you get a nearly frontal view of the wall. Now use Ctrl+RightDrag to move the paddle around with the mouse. This may be difficult because the paddle is pulled towards the (0,0) position by the virtual spring. You can make the spring weaker by reducing the kp values in the actuator section of the XML, but make sure you restore the original values for the actual homework.