Kalman Filter Lab

Chris Monson, Reid Broadbent, and Mike Goodrich

October 27, 2005

If I make changes to these notes after they are posted and if these changes are important (beyond cosmetic), the changes will highlighted in bold red font. This will allow you to quickly compare this version of the notes to an old version.

The purpose of this lab is to give you an idea of what the Kalman Filter does and under what conditions it works well. (HINT: it doesn’t work well under all conditions.) Your mission, should you choose to accept it, is to do the following:

You will be required to do the following:

• Code up the multivariate (i.e., matrix-based) Kalman Filter update equations.
• Use your Kalman Filter to estimate the position, velocity, and acceleration of a moving BZFlag agent, for which you have noisy observation data; the estimation vector is the x and y positions, x and y velocities, and x and y accelerations.
• Print the estimation vector to the console.
• Use your estimate to trail the enemy agent and tag it.

1 What to Turn In

To pass off this lab, you will:

• Meet with one of the TAs and demonstrate that your code works well; you will need to show the TA the portion of your code that computes the estimation using the Kalman Filter equations.
• Turn in a write-up electronically to the TA. This write-up needs to include (a) a declaration of time spent by each lab partner and (b) a well-written, formatted (e.g., title, section headings, etc.) lab report that includes both a summary of what you did as well as results of some experiments that you conducted.

You will need to demonstrate to the TA that your code works. Your code should print to the console the estimation vector. Using this estimation vector, you will trail, intercept, and tag the enemy agent. You will need to show to the TA your estimation code and your tagging strategy.

The writeup for this lab should include information about what kinds of transition and covariance matrices you used and how they affected your filter’s performance. Do meaningful experiments that test the abilities of the filter, and try to make meaningful and insightful observations. For example, consider experimenting with different values in your $\Sigma_x$ and $\Sigma_y$ matrices and then note how these values affect the filter’s performance.

2 Description

The BZFlag distribution for this lab includes a new enemy agent called RandomWalkAgent, which behaves in a well-defined but randomized manner. More precisely, the agent maintains constant acceleration for a half-second at a time, then switches to a different acceleration for the next half-second, and etc. There are three exceptions to this general rule:
• The velocity is capped which means that the enemy agent will not ever go faster than a pre-defined maximum.

• The server provides a “friction” effect; at each time tick, before applying the new acceleration to change the agent’s velocity, the acceleration is discounted by a constant fraction of the agent’s velocity. (One side note: real friction is actually proportional to velocity squared, but we are implementing it this way because it is easier to encode in a matrix.)

• When the RandomWalkAgent gets close to a wall, its acceleration may change a bit more intelligently to move it away from that wall.

3 Distribution

First, download the code distribution (which is identical to the previous distribution except for a modified version of RobotPlayer.cxx and playing.cxx, plus the addition of some code to support matrix arithmetic). After downloading the code, you will create a ta_bot executable from the RandomWalkAgent.cxx code file. Then, you will create your own Kalman Filter agent to predict the ta_bot’s position on the field.

Your job is to apply the Kalman Filter to the observations of the enemy agent’s position, once every half-second, and print out the estimation vector to the console indicating what the filter thinks are the true x-y positions, x-y velocities, and x-y accelerations. To move your agent towards the location of the enemy, set the x and y acceleration values in the way you did for the Potential Fields lab in such a way that you will intercept the enemy.

4 Example Matrices

To accomplish this lab, it is helpful to understand the “physics” used by the enemy agent. We will represent these physics using matrices as done in the class discussions. You will want to play with the values in these matrices, especially $\Sigma_x$ and $\Sigma_z$, and we encourage you to do so in order to better understand how the Kalman Filter works.

Initially, the enemy agent will be at some unknown position on the playing field, and the velocity and acceleration will both be zero. You can use that information to create your initial estimates of the mean and covariance. The physics are based on the six values in our state vector (in this order, represented as a column vector):

$$x_t = \begin{bmatrix} x_t \\ \dot{x}_t \\ \ddot{x}_t \\ y_t \\ \dot{y}_t \\ \ddot{y}_t \end{bmatrix}, \quad (1)$$

where $x$ and $y$ are the $(x, y)$ position of the enemy agent, $\dot{x}$ is the $x$ component of the agent’s velocity, $\ddot{x}$ is the $x$ component of the agent’s acceleration, and etc.

Given this state vector, the Kalman Filter will produce a mean estimate for this vector $\mu$ and a covariance matrix for this vector $\Sigma$. So, your initial estimates of the mean and covariance could look like these:

$$\mu_0 = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \quad (2)$$
which means that you think the agent begins at the origin with no velocity and no acceleration, and

$$
\Sigma_0 = \begin{bmatrix}
100 & 0 & 0 & 0 & 0 & 0 \\
0 & 0.1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0.1 & 0 & 0 & 0 \\
0 & 0 & 0 & 100 & 0 & 0 \\
0 & 0 & 0 & 0 & 0.1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0.1
\end{bmatrix}, \tag{3}
$$

which means that you are pretty sure that the agent is not accelerating or going anywhere, but that you are
pretty unsure exactly where the agent is.

Once every half-second, the enemy agent will update its state \( x \) as follows:

$$
X_{t+1} = FX_t + \eta_t. \tag{4}
$$

In other words, the enemy agent applies the system transition matrix \( F \) to its previous state and then adds
noise, which is drawn from some distribution. Since the initial state and all subsequent states are random
variables, these variables are capitalized to be consistent with our notes in class. The \( F \) matrix used in this
lab is precisely the one that we derived in class using Newton’s laws of motion (with one exception):

$$
F = \begin{bmatrix}
1 & T & \frac{1}{2}T^2 & 0 & 0 & 0 \\
0 & 1 & T & 0 & 0 & 0 \\
0 & -c & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & T & \frac{1}{2}T^2 \\
0 & 0 & 0 & 0 & 1 & T \\
0 & 0 & 0 & 0 & -c & 1
\end{bmatrix}, \tag{5}
$$

where the \(-c\) indicates that we have a linear friction force working against this agent.

In this lab, the updates will happen every half-second, so \( T = 0.5 \). The friction coefficient \( c \) is set to 0.1.

The noise \( \eta_t \) is actually a vector of noises drawn from a normal distribution. In reality, only \( x \) and \( y \) accelerations have noise (with a standard deviation of 0.5), but since there are some other influences on the
behavior of the agent (such as being pushed away from walls) you will want play with the covariance matrix.
A good place to start is with a covariance matrix that emphasizes acceleration more than position, like the
following:

$$
\Sigma_x = \begin{bmatrix}
0.1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0.1 & 0 & 0 & 0 & 0 \\
0 & 0 & 100 & 0 & 0 & 0 \\
0 & 0 & 0 & 0.1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0.1 & 0 \\
0 & 0 & 0 & 0 & 0 & 100
\end{bmatrix}. \tag{6}
$$

You will actually be provided with two observation values: the true \( x \) and \( y \) positions and noisy measure-
ments of the \( x \) and \( y \) positions. Do not use the true positions to control your agent; these are only provided
to allow you to check to see if your code is working correctly. The noisy measurements of these positions will
have zero-mean Gaussian noised with a standard deviation of 5 units in each dimension. Since you will be
storing these observation values in your estimate vector, \( Z_t \), we can encode this information via the following
equation:

$$
Z_t = HX_t + \nu_t. \tag{7}
$$

As discussed in class, the observation matrix, \( H \), selects out the two “position” values from the state vector.
It looks like this:

$$
H = \begin{bmatrix}
1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0
\end{bmatrix}. \tag{8}
$$
Since these measurements are corrupted by noise, it is important to know the covariance matrix of this noise. Since the standard deviation of the $x$ and $y$ position noise is 5 and since these two noises are uncorrelated, the covariance matrix is given by:

$$
\Sigma_z = \begin{bmatrix} 25 & 0 \\ 0 & 25 \end{bmatrix}.
$$

(9)

## 5 A Few Implementation Hints

- You are more than welcome to implement your own matrix manipulation code. If you don’t wish to do so, you may use the code we have provided, in MatrixStuff.cxx (just include MatrixStuff.h in your agent code file). If you decide to use this code, please look through it and get a basic understanding of how it works. Feel free to change the code if you like, and please let us know if you find any bugs in it.

- Here’s the three Kalman update equations again, copied from the uncertainty notes:

$$
K_{t+1} = (F \Sigma_t F^\top + \Sigma_x) H^\top (H (F \Sigma_t F^\top + \Sigma_x) H^\top + \Sigma_z)^{-1}
$$

(10)

$$
\mu_{t+1} = \mu_t + K_{t+1}(Z_{t+1} - HF \mu_t)
$$

(11)

$$
\Sigma_{t+1} = (I - K_{t+1} H)(F \Sigma_t F^\top + \Sigma_x).
$$

(12)

- Be careful not to get confused with the different $\Sigma$ matrices: $\Sigma_x$, $\Sigma_z$ and the various $\Sigma_t$ matrices (one for each time step $t$).

- Note that these four matrices are constants, so they can be initialized once (like, in the world_init function): $F, \Sigma_x, H, \Sigma_z$.

- Note also that since $H$ and $F$ are constant, $H^\top$ and $F^\top$ are also constant, and can be precomputed just once.

- You can also initialize your $m_0$ and $Sigma_0$ matrices in world_init.

- Don’t be too scared by all the subscripts in the Kalman equations. Just think of $t$ as “last time” and $t + 1$ as “this time.”

- Note also that the expression $(F \Sigma_t F^\top + \Sigma_x)$ occurs three times in the equations, so you may save some time by calculating that first.

- To apply predictions into the future, you don’t make additional observations, so you shouldn’t use the full equations. Instead, try some variations. For example, you can do any of the following:

  - **No prediction:** Just take the current estimate of the enemy’s position and assume that it will not move very much in a half-second.

  - **Just add velocity:** You can take the “filtered” position returned from the Kalman Filter and add the “filtered” velocity. All of that information is stored in the $\mu_{t+1}$ vector after you run through the three Kalman filter equations.

  - **Apply the $F$ matrix:** The $F$ matrix essentially says, “Give me what’s happening now, and I’ll tell you what will probably be happening next time-step.” So, once you get your $\mu_{t+1}$ vector you can apply $F \mu_{t+1}$.

  - **Predicting more than one time step into the future:** This can be done by applying the $F$ matrix several times, producing, for example, $FFFFF \mu_{t+1}$

- Here’s a brief snippet of pseudocode that may help you:

  - If the number of time ticks is divisible by 50, then
* Find the enemy agent and observe its $x$ and $y$ position
* Place the $x$ and $y$ values into your $z$ vector (which has one column, two rows)
* Run through the three equations above
* Print $\mu_{t+1}$ to the console
* Predict where the enemy agent will be and guide your agent toward it.