Kalman Filter Simulation

Università degli Studi di Pisa

Information Engineering Department

Master's Degree in Robotics and Automation Engineering

Real Time Systems course project

Di Luzio Fulvio
492897
fulvio.diluzio@gmail.com

Contents

List of Figures

1 Introduction

This report shows the developed work for the *Real Time Systems* course project. The paper is organized as follows. Section 2 illustrates the goal of the project. Section 3 provides brief information about the Kalman filter. In Sections 4 and 5, GUI and tasks are respectively described. Section 6 shows files and commands for the execution. Conclusions are in the last section.

2 Course project

The goal of this course project is to develop a Kalman Filter simulation in **C language** under **Ubuntu OS** (version 17.04) using the **gcc** compiler and mainly two libraries: the **pthread library**, which lets to handle tasks, and the **Allegro library** (version 4.4), which lets to build a GUI.

3 Kalman Filter

3.1 LTI Systems

Physical systems can be modelled as LTI (linear time invariant) systems using the following discrete-time state-space model:

$$\begin{cases} x(k+1) = Ax(k) + Bu(k) \\ y(k) = Hx(k) + Du(k) \end{cases}$$

where:

- $\mathbf{x} \in \mathbf{R}^n$ is called the *state vector*;
- $\mathbf{y} \in \mathbf{R}^q$ the *output vector*;
- $\mathbf{u} \in \mathbf{R}^p$ the *input*(or *control*) *vector*;
- $\mathbf{A} \in \mathbf{R}^{n \times n}$ the *state matrix*;
- $\mathbf{B} \in \mathbf{R}^{n \times p}$ the *input matrix*;
- $\mathbf{H} \in \mathbf{R}^{q \times n}$ the *output matrix*;
- $\mathbf{D} \in \mathbf{R}^{q \times p}$ the *feedforward matrix*.

Most of times the state vector can not be measured directly, due to the fact that it is inaccessible, or, if sensors can be applied, the measurement error is too big. Therefore **state obserbers** have been designed.

3.2 State Observer (Estimator)

A State Observer is a dynamical system capable to determinate the state of a system (the vector $\mathbf{x}$) by the measurements of the same system (the vector $\mathbf{y}$).
The term **Observer** is used in a deterministic view. However measurements sensors are affected by noise due to the sensor itself, the system and the environment. "Noise" is an element that can not be predicted so the approach to build this dynamical system passes from deterministic to stochastic one. Instead of observer, we have to use the term **Estimator**.

Therefore the previous state-space form can be rewritten as:

$$\begin{cases} x(k+1) = Ax(k) + Bu(k) + w(k) \\ y(k) = Hx(k) + Du(k) + v(k) \end{cases}$$

where $w(k)$ and $v(k)$ are noises, respectively, on the input and on the output.

3.3 The Filter

The discrete-time Kalman filter is a recursive estimator. This means that only the estimated state from the previous time step and the current measurement are needed to compute the estimated current state.
The state of the filter is represented by two variables:

- $\hat{x}_{k|k}$ is the a posteriori state estimate at time k given observations up to and including at time k;
- $P_{k|k}$ is the a posteriori error covariance matrix (a measure of the estimated accuracy of the state estimate).

The filter is conceptualized into two different phases: **Prediction** and **Correction**. The predict phase uses the state estimate from the previous time step to produce an estimate of the state at the current time step. This predicted state estimate is also known as the *a priori state estimate*. In the correction phase, the current a priori prediction is combined with current observation information to refine the state estimate. This improved estimate is termed the *a posteriori state estimate*.

Prediction phase

- $\hat{x}_{k|k-1} = A_k\hat{x}_{k-1|k-1} + B_k u_k$ predicted (*a priori*) state estimate
- $P_{k|k-1} = A_k P_{k-1|k-1} A_k^T + Q_k$ predicted (*a priori*) estimate covariance

Correction phase

- $z_k = y_k - H_k\hat{x}_{k|k-1}$ *innovation* (or measurement)
- $S_k = R_k + H_k P_{k|k-1} H_k^T$ *innovation* covariance
- $K_k = P_{k|k-1} H_k^T S_k^{-1}$ *Optimal* Kalman gain
- $\hat{x}_{k|k} = \hat{x}_{k|k-1} + K_k z_k$ updated (*a posteriori*) state estimate
- $P_{k|k} = P_{k|k-1} - K_k H_k P_{k|k-1}$ updated (*a posteriori*) estimate covariance

Where $Q_k \geq 0$ and $R_k > 0$ are, respectively, the covariance matrices of $w(k)$ and $v(k)$. It is important to denote that $w(t)$ and $v(t)$ are supposed to be noises with a Gaussian *probability density function*(PDF). That is a very important condition for the correct work of the filter.

Furthermore, with the given conditions on the system, the Kalman filter is an optimal filter respect to the minimisation of the covariance matrix P_k.

4 GUI

The *Graphic User Interface* (GUI) is build with the **DIALOG** object contained in the **Allegro** library. It has several features, like buttons and sliders, and lets also to define own object types for particular necessities. The interface is shown in figure 1.

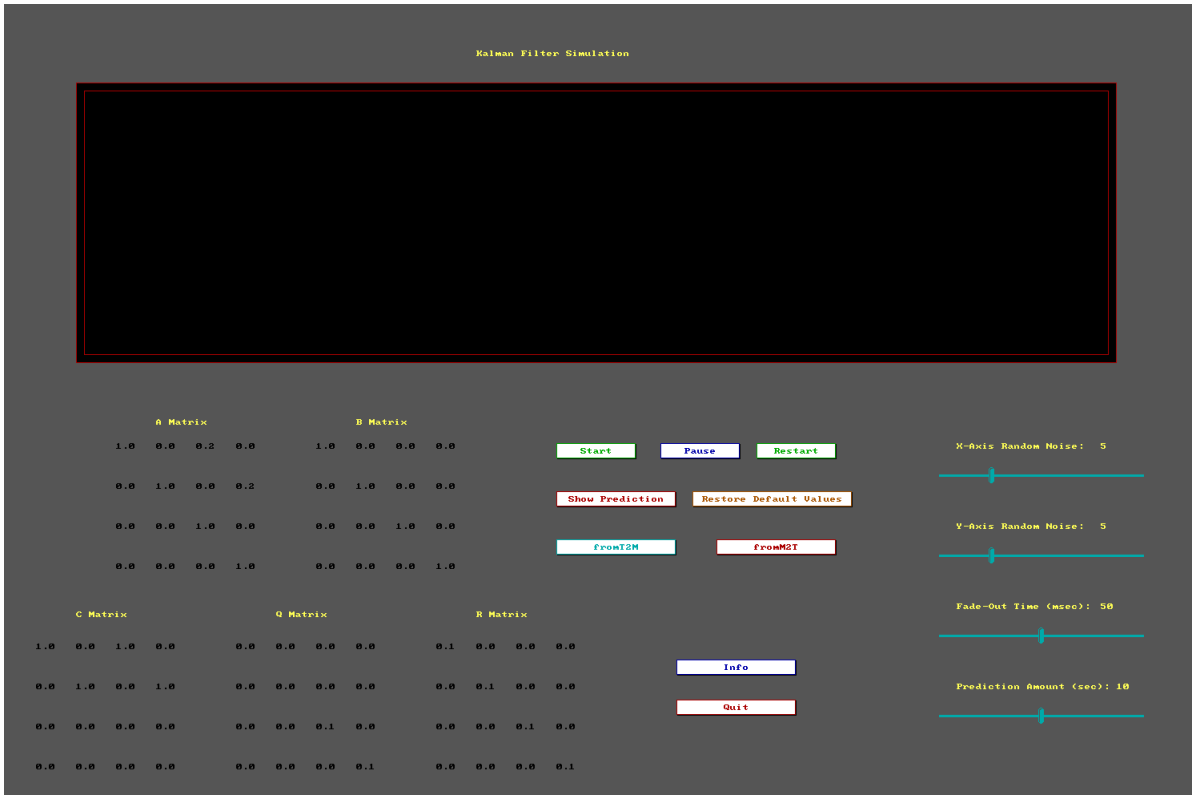

Figure 1: The GUI.

As explained above, this program simulates the behaviour of a Kalman filter. The position, which should be estimated by the filter, can be generated in two different ways:

- The user can draw any trajectory moving the mouse;
- Using a trajectory generated by the program.

Whatever the choice, it should look out as in figure 2. The red line represents the predicted trajectory by the Kalman filter, the green point is the estimated position by the filter. The blue points show the mouse position, corrupted by noise. The simulation works only inside the black window of the GUI (figure 1).

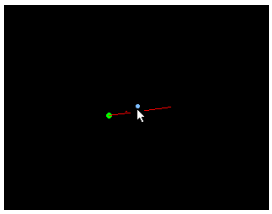

Figure 2: Screenshot of the simulation.

Looking always to figure 1, three groups of objects appear:

1. One of matrices, that describe the system model;
2. One of buttons;
3. One of sliders.

In the next sections, a description of these groups is given.

4.0.1 System Model

The GUI shows five matrices: A, B, H, Q and R. Their meaning was already explained in sections 3.1 and 3.3. The user has the possibility to change any value of these matrices.

4.0.2 Buttons

There are nine buttons and everyone has an own function.

- **Start**: it starts the simulation and places the mouse inside of simulation window;
- **Pause**: it blocks the simulation;
- **Restart**: it resumes the simulation;
- **Show Prediction**: it activates the drawing of the predicted trajectory;
- **Restore Default Values**: it resets the matrices and sliders to their default values.
- **fromM2T**: the trajectory is committed to the program;
- **fromT2M**: the trajectory is committed to the user;
- **Info**: it shows how to change a matrix element;
- **Quit**: it lets to close the simulation.

4.0.3 Sliders

There are four sliders that let the user to change the gain of some aspects of the simulation.

- **X-axis random noise**: a scalar gain that increases/decreases the noise along the x-axis;
- **Y-axis random noise**: a scalar gain that increases/decreases the noise along the y-axis;
- **Prediction time**: how far the prediction goes on;
- **Fade-Out time**: it makes the simulation more/less smooth.

Important Note: the modification of a matrix element, the buttons and sliders activations are all handled by Allegro.

5 Tasks

5.1 Task description

Here a brief description of the tasks is presented. The scheduler **FIFO** (First-In-First-Out) is adopted.

5.1.1 Task *masterTask*

It is a high priority, aperiodic task that activates the tasks *takePosition*, *drawPoint*, *Kalman*, *drawPrediction* and *restoreBackground*.
It is an aperiodic task that executes only when the *Start* button is clicked.

5.1.2 Task *takePosition*

It is a periodic task that records the computer mouse position (x,y) and its velocity (vx,vy) in an array (i.e. y_k of section 3.3). The task works only when the mouse is inside the simulation window. Otherwise the task keeps the last recorded data. The measurement is affected by noise with a gaussian DPF, that can be handled by the sliders.

This task shares a semaphore with the task *drawPoint* to maintain consistent data.

5.1.3 Task *drawPoint*

It is a periodic task that draws a point where *takePosition* has recorded the mouse position. As the previous one, this tasks works only when the mouse is inside the simulation window.

In order to have synchronisation between *takePosition* and *drawPoint*, they share a semaphore.

5.1.4 Task *Kalman*

It is a periodic task that computes the steps of **prediction** and **correction** seen in section 3.3.

5.1.5 Task *drawPrediction*

It is a periodic task. It is activated by *masterTask*, but draws the predicted trajectory only when the button *Show Prediction* is clicked.
It computes the **prediction** factor, seen in section 3.3, as many times as indicated by the slider *Prediction Time*.

5.1.6 Task *restoreBackground*

It is a periodic task that restores the background into a black window, in order to delete passed trajectories.
The period of this task can be handled by the user.

5.1.7 Task *masterTask2*

It is a high priority, aperiodic task. It is activated when *fromM2T* or *fromT2M* is clicked. In the first case, it disables the tasks *takePosition* and *drawPoint* and activates *trajectory* and *drawTraj*. In the second case, the result is the opposite.

5.1.8 Task *trajectory*

It is a periodic task that computes a parabolic trajectory. Then these values (position and velocity) are corrupted with a Gaussian PDF noise and loaded into an array (i.e. y_k of section 3.3).
Since this trajectory is generated by the program, its initial values cannot be modified by the user.

This task is synchronised with *drawTraj* by a semaphore.

5.1.9 Task *drawTraj*

It is a periodic task that draws the points of the trajectory generated by *trajectory*, with whom it shares a semaphore.

5.2 Task parameters

Table 1 shows the parameters *priority*, *period* (ms) and *deadline* (ms) chosen for every task.

Tasks parameters			
Task	***Priority***	***Period***	***Deadline***
masterTask	90	-	150
takePosition	50	25	150
drawPoint	50	25	150
Kalman	55	25	150
drawPrediction	55	25	150
restoreBackground	50	100*	150
masterTask2	90	-	150
trajectory	50	25	150
drawTraj	50	25	150

Table 1: Table of task parameters.

The parameters choice lays on two aspects: 1) all the tasks are activated simultaneously; 2) they have to be enough fast to follow and elaborate the movements of the computer mouse, moved by an user.
Based on this last one, the periods have been chosen by a trial-and-error approach. In order to have a simultaneous activation, *masterTask* and *masterTask2* have very high priority, since they are responsible for the activation of all the other ones. The remaining tasks have the same priority level.

*: The default period is $100ms$ and can be changed by the user by the slider *Fade-Out time* in a range of $[50;150]ms$.

6 The Program

The program is divided into three files:

- **mylib.h**: it contains the prototypes of useful functions;
- **mylib.c**: here functions are specified;
- **kalman_simulation.c**: it is the main code where tasks and GUI are handled.

The commands from terminal are:

1. sudo gcc kalman_simulation.c mylib.c ‘allegro-config –libs‘ -lpthread -lm
2. sudo ./a.out

7 Conclusions

The so structured program lets to simulate a Kalman filter, which is a state estimator. The trajectory to estimate can be drawn by the user simply moving the computer mouse inside the dedicated space, or generated by a function clicking the appropriate button of the GUI. The user has the possibility to change some parameters of the simulation via the GUI.

The simulations did not show any missing deadline by tasks.

All the work has been developed in **C language**, under **Ubuntu OS**, with the support of libraries **Allegro** and **pthread**.

References

[1] G. C. Buttazzo, *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*, Third Edition, Springer, 2011.