École Centrale Paris

# Motion Tracking with Kalman Filter

# Computer Vision Project

*Teacher*
Olivier Juan

Pierre Gergondet
Joël Lopes Da Silva

June 13, 2008

**Abstract**

Kalman Filter is a common algorithm based on two basic steps to track objects.

After reviewing quickly the theory behind the Kalman Filter, we will see how it can be implemented in C++ with the library `CImg`.

Finally, we will see how such a motion tracking program can be enhanced with some bonus features.

# Contents


**Introduction** **3**

**I An introduction to Kalman Filter** **4**
- 1 General presentation . . . . . 4
- 2 Prediction step . . . . . 4
- 3 Correction step . . . . . 5

**II Kalman Filter for constant speed** **6**

**III Kalman Filter for adaptive speed (acceleration)** **8**
- 1 Adaptation of the algorithm . . . . . 8
- 2 Observations . . . . . 9

**IV New features** **11**
- 1 Improving occlusion resistance . . . . . 11
  - 1.1 Finding when we loose track of the target . . . . . 11
  - 1.2 What to do when we loose track of the target . . . . . 12
  - 1.3 Observation . . . . . 13
- 2 Color support . . . . . 14
- 3 Multi-tracking . . . . . 15
  - 3.1 Goals . . . . . 15
  - 3.2 Principle . . . . . 15
  - 3.3 Implementation . . . . . 16
- 4 Trying to find automatically the targets . . . . . 17

**Conclusion** **19**

# Introduction

Motion tracking is a major application of Computer Vision. For example, it can be used to monitor some place and check every movement in this place. Kalman Filter is a common algorithm for motion tracking.

While the theoretical aspects of the Kalman Filter may be quite complicated, its implementation in C++ with the library `CImg` is not too difficult, at least when we only take position and speed into account. But when we try to use the acceleration too, then the results can be very bad.

Fortunately, adding some new features (some of them are in fact *tricks*) to the program can enhance tremendously our results.

Since the features we implemented were most of the time needed by bad results of our previous attempts, we prefer mixing the process explanation and the corresponding results. Our progression in this document will be quite close to our progression while implementing the Kalman Filter, and those features.

One last thing about the computation part: in order to prevent having to recompile everything each time we had to change some parameter, we used a configuration file for most of the useful parameters we introduced. This configuration file is also useful to choose the dataset. We used two very simple libraries for the configuration file. They are called `ConfigFile` and `Chameleon`, and can be found here:
http://www.adp-gmbh.ch/cpp/config_file.html.

# Part I

# An introduction to Kalman Filter

## 1 General presentation

Kalman Filter is to the so-called data linear filtering problem, which in practice leads to simply tracking the state of an object. It has been the subject of extensive research and applications and had a lot of success due to the fact that it's very powerful, resistant to system's model's lack of precision, and yet quite simple both mathematically and to implement.

The algorithm is divided into two steps, though we could mix them into a one step operation, but we need two steps to take a clear look at our doings.

## 2 Prediction step

We first have *prediction step*, which is a very primal guess which doesn't take into account process noise, movement variations, etc.

The operation is:

$$x_t^- = Ax_{t-1}P_t^- = AP_{t-1}A^T + Q$$

$A$ relates the state at $t-1$ to the state at $t$ without considering the presence of process noise, movement variations, etc. It's simply called *transition state matrix*. In practice, it may change with each step, however we will consider it constant.

$Q$ is the *process noise covariance*. It also may vary with each step, however we also consider it constant to solve the problem.

# 3 Correction step

After *prediction step* comes *correction step*, the objective here is to correct errors taking into account noise and initial variations compared to the movement modelised by the chosen matrix $A$.

The operations are:

$$K_t = P_t^- H^T \times (H P_t^- H^T + R)^{-1} x_t = x_t^- + K_t(z_t - H x_t^-) P_t = (I - K_t H) P_t^-$$

The state $z$ is a *measurement* of the current state, for example, in our implementation we chose to use cross-correlation to determine a measurement $z$ of the state $x$. The goal of the correction step is to *blend* this measurement and the prediction to obtain an accurate actual position.

The matrix $K$ is chosen to be the *gain* or *blending factor* that minimizes the *a posteriori* error covariance.

The matrix $H$ relates the state to the measurement. It may vary through the step of the algorithm but we assume it constant.

# Part II

# Kalman Filter for constant speed

Given the general version of Kalman Filter algorithm we can notice that it only requires the determination of a few parameters to be able to use it to predict the states of a system through the time.

Those factors are $A$, $Q$, $R$ and $H$.

$Q$ and $R$ will be fixed by the algorithm and determined empirically by experimenting the algorithm with various datasets. Also, $R$ will sometimes be adjusted in order to make the algorithm more resistant to occlusion, when we estimate that the object may be hidden, we will strongly increase this factor.

$H$ will be considered as a representant of the observables of the system. Through our work we only included position as an observable but extending the kalman filter to take account of, let's say, the textures of the object, by adding data into the state vector for example, we may extend it to both position and texture informations.

$A$ is directly linked to the nature of the movement we want to track, that's why, given that for a movement at constant speed we have:

$$x_t = x_{t-1} + v_{t-1}\delta t$$

And that the state vector is:

$$\begin{pmatrix} p \\ v \end{pmatrix}$$

We choose $A$ to be:

$$A = \begin{pmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Implementation is then simply a matter of translating the mathematical equations into `CImg` operations. This part of the project was done as a lab exercise and didn't need any extensive improvements.

# Part III

# Kalman Filter for adaptive speed (acceleration)

## 1 Adaptation of the algorithm

Explanation given in the previous section about the choice of $Q$, $R$ and $H$ are still true when we want to track an adapative speed movement. Indeed, the only adaptation we theorically have to do is on $A$. Although we will see that it indeed had a huge impact on the quality of the algorithm, especially when comparing its occlusion resistance compared to the one of the constant speed algorithm.

To choose $A$ we need to model physically the movement of an object at adaptive speed. A good approximation is to consider it as a "constant" speed movement where the "constant" speed is growing according to a really constant acceleration.

That's why we model it with the following equations:

$$x_t = x_{t-1} + v_{t-1}\delta t v_t = v_{t-1} + a_{t-1}\delta t$$

With a state vector being:

$$\begin{pmatrix} p \\ v \\ a \end{pmatrix}$$

This leads to the following matrix $A$:

$$A = \begin{pmatrix} 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

This was implemented quite easily given the original work for constant speed. The code only required minor modifications:

- Modify $A$ according to our formula
- Modify the state vector to include acceleration
- Modify $R$, $Q$ and $H$ dimensions

And so was the job done, although we observe some issues compared to the constant speed algorithm.

# 2 Observations

Taking into account the acceleration turns out to be quite complicated. Basically, when the target gets invisible for a certain amount of time, the classic prediction and correction steps lead to wrong results which get accumulated.

For example, when a cyclist gets hidden underneath some trees, the position guessed after the Kalman Filter process may not correspond to the actual position of the cyclist: we just pick the position corresponding to the best patch (for the maximum Cross Correlation). This kind of error occur as long as the cyclist stays hidden. However, what might be the difference between the precedent case, when we did not take into account acceleration?

The problem is that when we make our first error concerning the position of the target, its speed (as a vector, i.e. including its direction) may vary a lot. Therefore, the new value of the acceleration of the target may be very different from what it should be. It can be much higher (in norm), and it can have a very different direction. Add to this the influence of that new value of the acceleration on the future value of the speed, and it becomes clear that the prediction step in the future frame will give very bad results.

In other words, within two or three frames, when the target is not visible, the acceleration tends most of the time to modify the speed in a wrong way, due to the error made in the position of the target in the successive correction steps. This results of course in a tremendous error in computation of the new position of our target, within very few frames, and getting even worse as the target stays hidden longer.

Obviously, we needed to use some trick to prevent such a bad guess in the new postion make the found positions inevitably derive, just because of the acceleration influence.

# Part IV

# New features

## 1 Improving occlusion resistance

Somehow, we still had to fix our implementation of the Kalman Filter with acceleration because of its very bad resistance to occlusion.

### 1.1 Finding when we loose track of the target

The idea we implemented is actually quite simple. Instead of believing blindly in the result of the Kalman Filter process, we should check its pertinence. Indeed, when we actually loose track of the target, the best match we find must be far worse than the previous ones. But what does “worse” means?

It simply has to do with the value of the maximum Cross Correlation in the search box. Since it is the criteria used to choose the best patch in the search box (the one which looks the most like the previous one), we can think that a low value of the maximum Cross Correlation means a bad match between the two patches.

The idea we chose was to measure the relative error between the current maximum Cross Correlation and the mean of the previous maximum Cross Correlation. If the current maximum Cross Correlation is $m$ and the mean of the previous ones is $\bar{m}$, then our criteria is:

$$\frac{\bar{m} - m}{\bar{m}} \leq 1 - \lambda$$

where $1 - \lambda$ is the maximum relative error we tolerate. When the current relative error exceeds this value, then we consider that we just lost track of the target.

We actually can simplify the precedent criteria:

$$m \leq \lambda \cdot \bar{m}$$

Of course, $\lambda$ represents the minimum percentage of the resemblance of new patches compared to previous ones.

This is the version we implemented in the code (but it is strictly equivalent to the previous one). This parameter is referred as `comparisonFactor` in the section `mean` of the configuration file.

Of course, when the maximum Cross Correlation is low, we don't consider this bad value for the computation of $\bar{m}$. We also have to wait some time to initialize the value of $\bar{m}$. The parameter `minimumFrames` in the section `mean` of the configuration file is the number of frames to wait until we use the previous criteria to know wether the match is good or not.

## 1.2 What to do when we loose track of the target

The idea is simple: the most probable reason why we would loose track of the object is the presence of some obstable which prevented us to see it. Therefore, no matter how hard we look around for some resembling patch, we won't actually find the target. That's why in such cases, the correction step actually messes everything up, and we would probably get better results doing only the prediction step.

However, the trajectory might be weird, even when the target is hidden. Therefore we must progressively increase the size of the search box as long as we don't find the target again. The results showed that increasing the size of the search box tended to slow down the computation. We decided therefore to limit the growth of this box to a certain factor of the size of the target. This factor is referred as `maximumSizeFactor` in the section `searchBox` of the configuration file.

Moreover, we must control the speed of this increase. If the search box becomes too big too fast, another potential target, quite resembling but not the actual one we were tracking, can be taken for the good one, while it is not. On the other hand, if its growth is not fast enough, then the target can become out of reach (outside the search box). We chose to quantify this speed with another factor, referred as `increaseFactor` in the section `searchBox` of the configuration file. This is the factor of the size of the target which will be added to the size of the search box as long we loose track of the target.

When we find again the target, we must of course decrease the size of the search box, to speed up the computation, and to prevent it to be too big (which could result in two similar objects present in the search box).

Finally, another problem must be solved. When two objects quite similar are moving in a video, and the one we are tracking gets hidden behind the other one for one or two seconds, then the program can be mistaken, and consider the second one as the target. What we see in such cases is the position of the target jumping from one object to the other. We used one more trick to help in such cases. Before beginning to decrease the size of the search box, we have to wait some time, to be sure we actually find again our target. The corresponding parameter is referred as `minimumGoodMatchesToDecrease` in the section `searchBox` of the configuration file.

## 1.3 Observation

The parameter $\lambda$ referred above is actually very important, and the program is very sensitive to its value. If it is too low, then we realize too late that we just lost track of the target, whereas if is too high, the program thinks it lost the target all the time, and gets rid of the correction step, which is after all useful most of the time.

For example, to track the first cyclist, we had to choose about 0.6, whereas to track one of the people walking, we had to choose 0.8. Indeed, since all the walking people are wearing the same clothes, they look a lot like one another. Therefore we must say to the program that we don't tolerate a relative error superior to 20 %, to prevent the target from jumping from one person to another.

To better observe the process, we decided to show the target in a red box, and the search box in a black box.

Figure IV.1: Search box resizing when loosing track of the target

# 2 Color support

The algorithm which was originally written in lab only worked with grayscale image. It had the advantage of making the computation faster but made the algorithm not very resistant on some particular datasets.

For example, let's consider a dark green square moving at constant speed above a dark red background. Theoretically it's a perfect case for our algorithm: images are clean (noise could be considered to be equal to zero), the tracked object has a constant and simple form and the movement is at constant speed which is the simplest case. Yet, if we work with grayscale images we will fail as dark green and dark red are very similar (most certainly exactly similar indeed) in greyscale. That's why color support is needed to improve the algorithm.

Its implementation only required to modify the step of the algorithm where you measure the position of the object, in our case, it implied a modification of the cross correlation measurement function. This modification was quite simple since we only had to modify the computation of the mean matrix of a given matrix $u$.

Indeed, in the first implementation we had:

$$\bar{u} = \frac{u.sum()}{u.dimx() \times u.dimy()}$$

Which is not accurate for a multiple color channels image since `CImg` will use three different images of size $x \times y$ to represent a three color channels image. That's why it had to become:

$$\bar{u} = \frac{u.sum()}{u.dimx() \times u.dimy() \times u.dimv()}$$

It then worked quite well, and we also observed some deviations due to the color tracking:

- Computation is a bit slower, this is not surprising since `CImg` implementation implies that working with color images implies at least 3 times (4 times if an alpha channel is present in the image) more operations because the matrix contains at least 3 times more factors.
- On greyscale images, the tracking seems a bit less stable (target oscillates a little around the real position of the object) although this is maybe more of a feeling than a tangible problem.

We also think that in some cases we could get better results by using only greyscale images or by focusing on a particular channel of the images or one other image factor such as luminosity. For example, one of the provided datasets showed a fish in a see. The images were not very clear and tracking is almost impossible (plus the fish is going back and forth out of the image). Yet, we feel that using luminosity channel instead of all the channels may lead to interesting results since the fish is very brilliant. Unfortunately, given the time table, we couldn't verify our idea.

# 3 Multi-tracking

## 3.1 Goals

The main objective of multi-tracking is obviously to track multiple targets. However, simply doing this is quite easy but kind of useless though it's funny to see a multi-tracking in action. That is why we can use the different targets to make the algorithm more resistant to occlusion but mostly to prevent the algorithm to be mistaken when the target looks like other possible targets (which we will multi-track) in the scene. For example, that is the case with the cyclists video or the walking guys video.

## 3.2 Principle

We are going to present the principle of this method with only 2 targets because it's simpler to describe with words and in the meantime easy to extend to multiple targets.

Let's say that we have 2 targets: target 1 and target 2. Let's imagine that they are following each other at different speed. For example, target 2 is following target 1 and then gets ahead target 1. Let's also suppose that they have a very similar appearence.

With the classic kalman filter, while we're tracking target 1 we may lost him and get him confused with target 2 while target 2 is going ahead of target 1 and then we may continue to follow target 2 and basically, target 1 would be lost.

Let's now say that we are tracking target 1 and target 2 at the same time. When target 2 goes ahead of target 1 and we confuse target 1 with target 2, we can use the tracking of target 2 to realize that we confused 1 with 2. Indeed, if target 2's tracking is good, event if target 1 algorithm's

prediction is better than target 2 algorithm's prediction, the prediction (i.e. step 1 of kalman filter) of target 2's position will probably score a lot more better with the algorithm predicted position of 2 than the prediction of 1. In this case, we will assume that measurement failed and so that correction step was mistaken so we will skip this step for target 1's position.

## 3.3 Implementation

Given a set $[1, n]$ of targets, we define a matrix $D$ which coefficients are defined by:

$$\lambda \in [0, 1], D_{i,j} = \lambda correlation(x_{i_t}^-, x_{i_t}) + (1 - \lambda) correlation(x_{i_t}^-, x_{j_t})$$

Where $x_{i_t}^-$ is the state at prediction step for step $t$ of the algorithm and $x_{i_t}$ is the state at correction step for step $t$ of the algorithm.

To choose the matching patch for target $i$, we choose the patch $j$ for which we have a highest $D_{i,j}$, except if $D_{j,j}$ is the highest one for $j$. In this case, the matching patch for target $i$ is the prediction one (i.e. we skip correction step).

$\lambda$ was typically chosen around 0.7 and above which is a way to say that we trust the algorithm result and that has proven efficient as long as the targets are not too mixed together. In the opposite case you may want to decrease this value to make sensitive to target's switching.

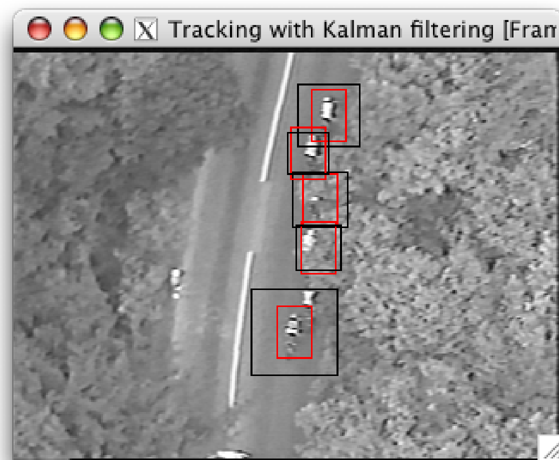


Figure IV.2: Multi-tracking

# 4 Trying to find automatically the targets

One of the annoying things with our implementation of the Kalman Filter was that we had to manually initialize the position of our target. A much better program would find by itself were the objects in motion are.

To do such a thing, we have two ideas. If the video camera does not move, then we can use a static frame of the background, and substract the current frame to the background frame. That way, we can distinctly see the potential targets.

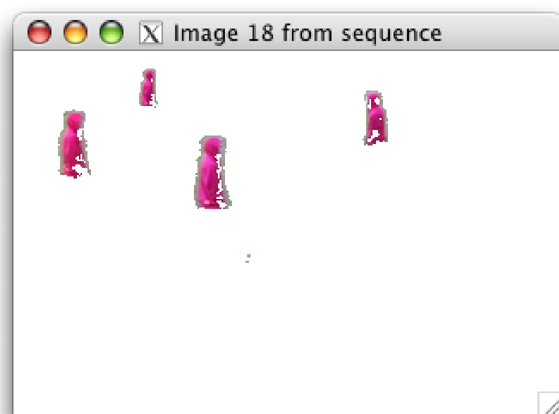


Figure IV.3: Abstracting background to find the potential targets

But this is actually impossible to do when the video camera also moves.

Another idea would be to measure the displacement of each pixel in the frames computing the optical flow. Then, when the optical flow value is high, something is moving. We must then choose a threshold to decide the limit above which a pixel is considered as moving.

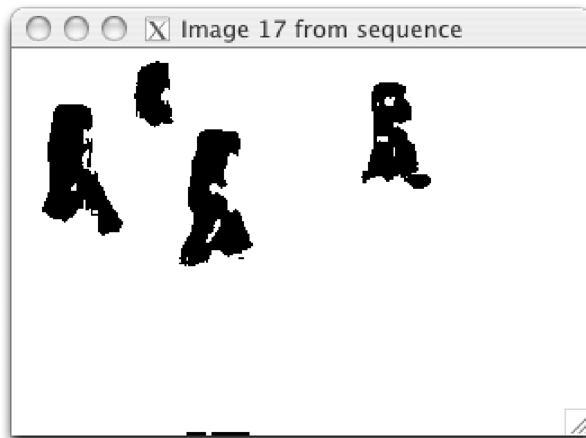


Figure IV.4: Using the optical flow to find the potential targets

Such tricks would have to be done in a pre-process computation, only on the first frames, to detect the positions and the dimensions of the potential targets. Then, we would be able to use the classic program.

Unfortunately, we did not have time enough to complete this, computing thanks to those images the positions and dimensions of the targets. This might be a good idea, but I'm truly sorry we could not make it on time for this project.

# Conclusion

This project clearly showed to us the limits of the Kalman Filter. Indeed, while a simple implementation of it with only position and speed in the state vector worked nearly out of the box, when we added acceleration, the correction step tended to worsen errors which occurred when the target got hidden.

Skipping this correction step and adapting the size of the search box when we loose track of the target gave us a solution for this problem. But this solution relies in fact in some very important parameters, which values vary tremendously from one video to another. To be more accurate, the important factors are the speed of the objects in motion, the typical distance between two objects in motion, and their percentage of resemblance.

Some other interesting features have been implemented too, such as color support, with a limited impact, and multi-tracking which was a real breakthrough, even though this feature was not perfect yet.

Unfortunately, a good idea about finding automatically the target couldn't be implemented in time, but we could foresee the kind of solution we could try to implement.

In the end, this project has been full of teachings, since it allowed us to see how powerful the Kalman Filter can be, how limited it can be at the same time and what kind of solutions we could bring to enhance its results.