Object tracking in embedded systems: component labeling and trajectory prediction

Stefano Varotti, Alessandro Lenzi

September 20, 2014

Project report for Sensor Network course, Scuola Superiore Sant'Anna, Pisa, Italy

1 Introduction

1.1 Problem statement

Embedded computing systems are a unfavorable platform to perform computer vision tasks, due to the stringent memory and processing power constraints. Traditional algorithms must be modified to accommodate to the hardware requirements, so the feasible computer vision tasks are limited. In this particular project we focused on two tasks that are fundamental for the objective of real time object tracking in a video stream. These two tasks are component labeling and trajectory prediction.

Component labeling is the process of splitting the pixels that have been detected containing some moving object into separate shapes grouping them by adjacency. The labeling part is improved upon a previous implementation in the paper "SCOPES: Smart Cameras Object Position Estimation System", that uses a union-find algorithm. The prediction tasks correlates the shapes detected at a certain point in time with the past events, so it recognizes previously seen objects. The prediction part uses a Kalman filter calculated upon the centroids coordinates of the detected moving shapes.

For testing we used a SEED-EYE board, that consists of a PIC32 CPU without FPU, with 128 KB of available RAM and 512K of ROM. The SEED-EYE board has support for image acquisition via camera, but for the experiment it received images through UART serial line, processed them and passed back results and benchmark information back to the PC. As a source of images we used the Fudan Pedestrian data-set. This data-set comes from a camera mounted in front of the entrance of a office building in Shangai. It consists of 5 sets of 300 QQVGA images of people entering and exiting the building. In the images of the dataset the motion detection part needed for the tracking task is already done, and ground truth black and white images are provided along with the video

images. So the SEED-EYE received already partially processed images where motion detection algorithm outlined moving object over the static background.

This project aim has been to lower the service time as much as possible in order to increase the possible frame-rate of a video stream for tracking, all while respecting the tight memory constraints.

1.2 Report structure

The first chapter is this and is a general introduction to the project, its goals and the current state of affairs. The second chapter delves into the specifics of the connected components labeling tasks, how does it work, how does it respect the constraints and how it performs on a real platform. The third chapter explains how Kalman filter works, how and why it has been used, and how it performs. The last chapter contains overall considerations of the tracking task on the SEED-EYE and proposals to further improve the metrics of the algorithms.

2 Labeling

2.1 Introduction

Connected component labeling is the process that splits pixel where motion has been detected into groups related to the shape they belong. So the input is a 320x240 black and white image where for each pixel we get two possible values, white if motion has been detected, or black if not. As a result of our labeling algorithms we get a set of blobs, each composted exclusively of adjacent pixels.

For a black and white image, using 1 bit per pixel, 9600 bytes are needed to store one image, while having a 8 bit label for each pixel implies reserving a buffer 76800 bytes per single image. On embedded system we can suppose to have from 256k to 64k of total available memory for the whole tracking task, so the memory requirements are so stringent that only few bits per pixel can be afforded. In the following pages we show some developed algorithms that allow component labeling to be performed using less than 20k of memory.

2.2 Reference implementation

The first algorithm that was taken into consideration is from the paper "SCOPES: Smart Cameras Object Position Estimation System" . The algorithm performs an union find, where a different label is assigned to each motion pixel and adjacent pixel are joined in a single set. The image is scanned from the top left and each pixel is assigned a label according to the pixel on the top and the pixel on the left according to the rule outlined in table 1.

Top pixel	Left Pixel	Action
-	-	New label
L1	-	Assign label L1
-	L2	Assign label L2
L3	L3	Assign label L3
L4	L5	Union(L4, L5); Assign L4

Table 1: Rules for union-find connected component labeling

The result of this is a buffer where each pixel is assigned a label, that ties it to a specific blob. When the last pixel has ben processed, all the pixels are labeled and all blobs have been detected.

The main problem of this algorithm is that has high memory use, mainly caused by the high number of temporary labels required, and so the number of connected components is limited by the space used. In table 2, we can see that to label the blobs of a complex image the memory requirements grows above the available space on a typical embedded system. Moreover, according to the data structure used to do the union find algorithm, the need temporary labels may lower even more the number of detectable blobs.

bits for pixel	bytes total	% of RAM used for a SEED-EYE (128 KB)	max blobs
1	9600	7.5%	1
2	19200	15%	4
3	28800	22.5%	8
8	76800	60%	256

Table 2: Memory usage and blob limits for union-find algorithms

The fastest union-find algorithms, for N pixels, have a $O(N + NlogN)$ complexity; though the theoretical space required is very high as it uses a number of labels equal of the number of pixels. Indeed if the programmer attempts to keep temporary labels to a minimum, each merge operation will need to modify each pixel belonging to one of the merged groups and so will make the merge more cpu intensive, while if the programmer attempts to avoid it the number of needed labels grows quickly. This is not a problem on a computer where 32 bits labels would be a non-issue, while on embedded systems is a concern. In figure 1 we can see an example blob that is hard to parse for a union find algorithms that tries to limit the maximum number of blobs to save memory. Indeed as it scans the second row it will need a label for each pixel, before joining all the labels while scanning the first row.

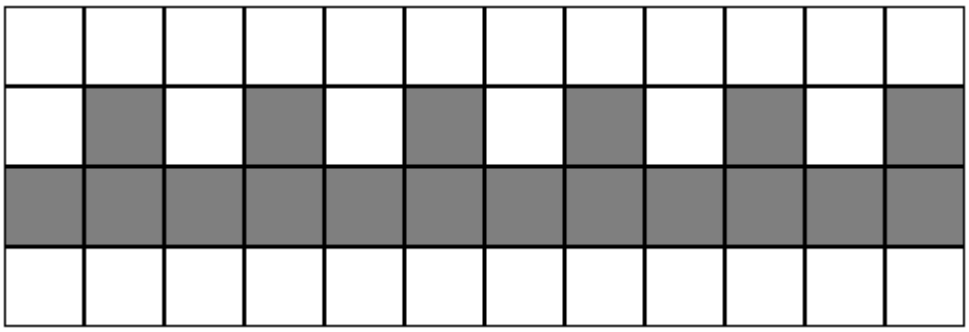

Figure 1: Worst case shape for union-find algorithms: high number of temporary labels

2.3 First improvement: multi-pass algorithm

The first modification to the union-find algorithm allows to use only two black and white buffers. This is possible by altering the behavior of the algorithm when a pixel that is disconnected from the first connected blob is detected. Instead of createing and assigning a new label, the new pixel is ignored and it stays unprocessed. Eventually the algorithm will find that the single blob currently being processed is still connected to previously ignored pixels. So when

this happens it performs a further scan of the raster in the opposite direction, so from bottom to top, right to left. The algorithm keeps scanning back and forth the image in loop until no more ignored pixels connected to the blobl are left.

As a result we obtain a buffer containing a single connected shape, that can then be processed separately, and that is deleted from the original mask image in order to allow the extraction of the successive blob. Figure2 shows an example where we can see the starting image and the detected connected components after the first and the second pass.

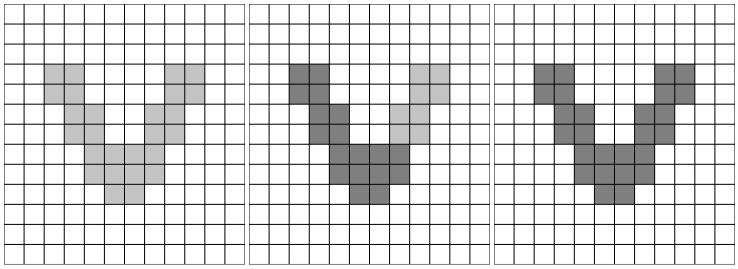

Figure 2: Example of multipass algorithm

This changes greatly the behaviour of the union find algorithm, that becomes a incremental algorithm as the connected components are extracted from the original mask one at a time. So at the price of increased CPU usage we can separate the connected components with a low memory usage. The duration of the algorithm for a single blob depends both on the size and on its shape, and the number of raster scans can be very high in worst case scenario. In figure 3 we can see an image where the pixels can be in such a setting that the image is scanned a very high number of times, requiring as much passes as pixels, so potentially thousands of passes. By fine-tuning the scanning rules the downsides can be relatively be limited but keeping a reasonable worst case timing constraint is difficult.

For example, intermediate steps can be interrupted early when the pixel scanned are at more than one row distance from the previously detected pixels. Another way to limit the number of raster scans could be to increase the number of intermediate buffers for pixel labels, so by utilizing an hybrid union-find algorithm. This implies using a low number of labels, and using multiple passes when they are not enough. This create a time-memory trade off scenario, but yet does not solve the unpredictability of the number of passes.

Even if this implementation turned off to be infeasible for the implementation of labeling on the SEED-EYE, it provided inspiration for further improvements and demonstrated that an incremental approach, where blobs are extracted from the image one at a time, is the best approach to limit the memory usage instead of using a buffer where the number of blobs is limited. Still in order to have a working connected component labeling algorithm a new approach is required.

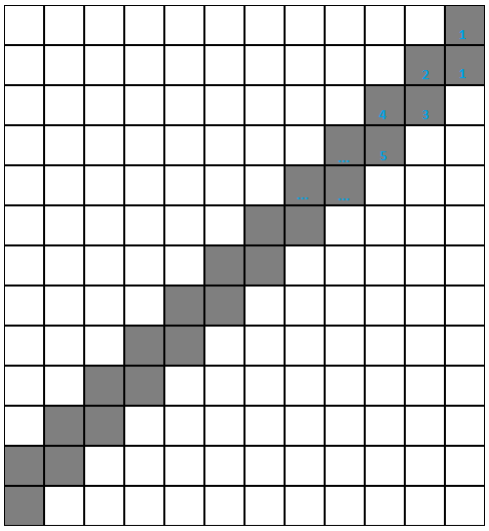

Figure 3: Worst case scenario for multipass algorithm

2.4 Second improvement: "labyrinth algorithm"

A better time bound can be obtained indeed by changing approach. The main inspiration for this has been the algorithm to explore a labyrinth, where by visiting the rightmost corridor at every intersection the whole labyrinth can be visited. So the essence of the algorithm is that as soon as the first pixel is found, the algorithm starts moving in a direction inside the shape trying to visit the next rightmost unvisited point. Figure 4 show what direction the algorithm takes while exploring a sample blob.

In practice we can consider a black and white image as a graph where each white pixel is a node and where adjacent pixels are connected with an edge. So a blob is a single connected subgraph, and after the first pixel is found the algorithm traverse a spanning tree starting from it. As each pixel has at most 4 adjacent pixels, the traverse algorithm tests for unvisited pixels at most 4 times, then it never visit it again, so we have a strong time bound. In figure 5 we can see an example of a spanning tree that is traversed by the algorithm.

This algorithm requires two black and white buffers, and two bits per pixel in a temporary buffer on the stack. First it raster scans the image to find the first pixel and then begin traversing the shape. When a new pixel is found, the output buffer is updated and the current scanning direction is stored in the temporary buffer. This is required in order to remember which was the adjacent node that was the parent in the spanning tree. Then the algorithms finds the rightmost unvisited pixel continues from that. For example when the search direction is RIGHT, it checks the pixel on directions DOWN, RIGHT, UP in this order. When no more unvisited pixels are found, the algorithm moves back

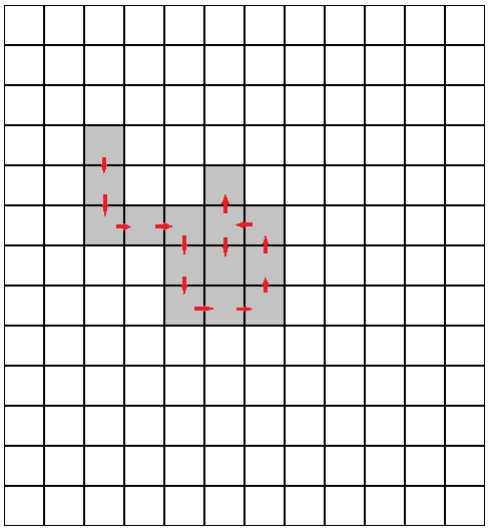

Figure 4: Visiting a shape keeping the right

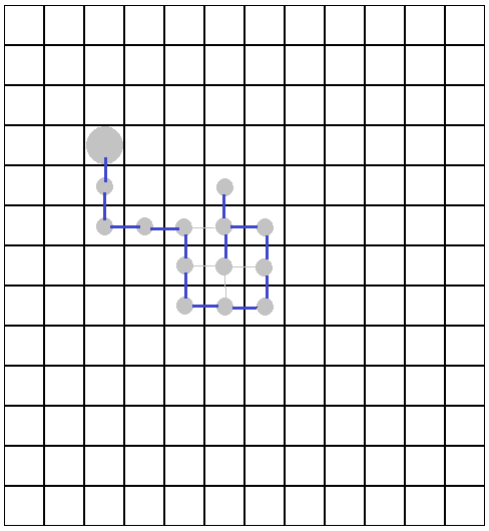

Figure 5: Connected pixels as graph with spanning tree

to the parent pixel, and when finally the root pixel is reached the algorithm terminates.

Compared to the previous approac it uses more memory, but it has the advantage of having a more predictable behaviour. Indeed if we get back to the graph representation of the image, we can see that for a N blobs if we visit the spanning tree we will move over N-1 edges and no loop occurs. So the required space is bounded by the number of pixel of the image, and time is bounded by the number of foreground pixels.

2.5 Correctness analysis

To test the correctness of the three algorithms we compared the results obtained over the images of the pedestrian data-set. As all of the three algorithms are fully deterministic, comparing the three different implementations should provide identical results. Indeed comparing the output for all images in the data-set shows that all the results are identical. So the only difference between the reference algorithm and the two improvements are the memory requirements and the timing.

2.6 Performance analysis

To measure the performance on real hardware, we used a SEED-EYE board connected via serial line to a PC. Ground truth images were sent to the board, and benchmark data was returned back and collected from the PC Timing measurement were collected from the elaboration for the extraction of a single connected component and for total image processing. The measurements plotted in figure 6 show that the time that the algorithm require to process a single blob is linear to the quantity of connected pixel, with a small but not negligible constant time part. It is interesting to see a small quantity of measurement in the area in shortly left to a thousandth of pixels and above 10 milliseconds, that defy the linear relation. Even if it consists of only eight data points, these outliers deserve a further future investigation.

Furthermore, the measurements showed that the total time for processing an image is a gamma distribution with average of about 30 milliseconds. From figure 7 we can see the cumulative probability to elaborate a whole image in a certain time. We can see that 50% of the runs are below 29 milliseconds, 90% below 44 milliseconds with rare worst cases above 60 milliseconds. This shows that for a high frame per second video stream, a soft real time algoritm is feasible provided with a mechanism could be devised in order to abort the elaboration of a frame in case of the rare excessive delay.

2.7 Optimizations

Since the full shape of the blob is not needed for the subsequent steps, instead of saving the blob into a buffer, the synthetic values effectively used such as the bounding box and the centroid coordinates can be calculated while each pixel

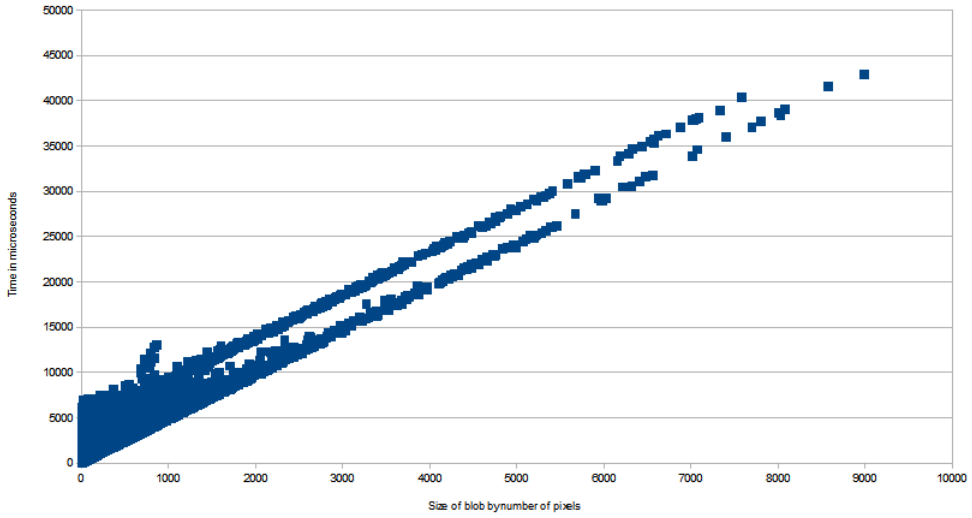

Figure 6: Scatter plot of timings for parsing a single blob

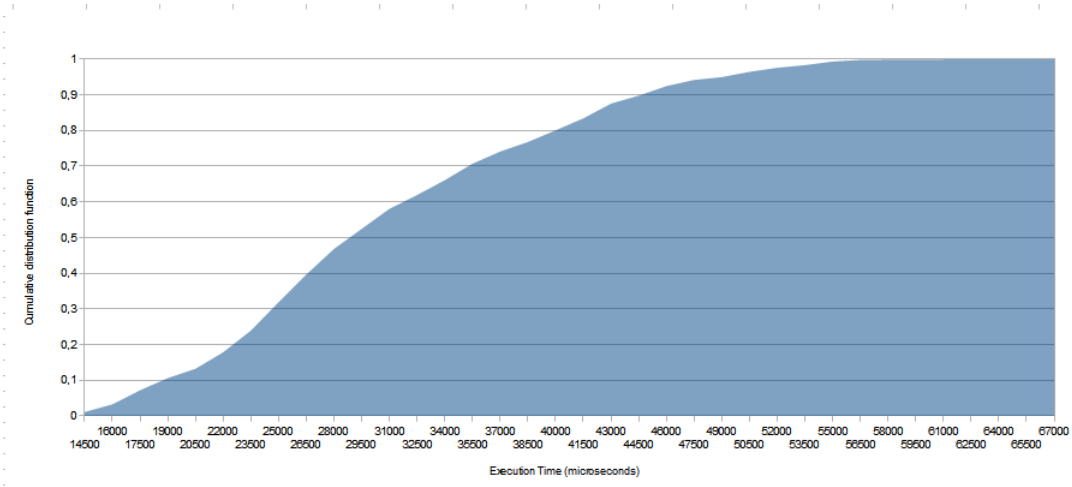

Figure 7: Cumulative distribution function of total labeling execution times

is being visited. This allows to cut off a 9.6 KB buffer used for the intermediate representation of each blob, and the relative buffer initialization and scan time.

Moreover, when updating the synthetic values some calculations done on each pixel can be simplified. For example the bounding box can be updated only when moving in a particular direction, so that when the next visited pixel is on the right only the right side of the bounding box will be checked and updated, and so on. This allows to reduce four checks performed for each pixel into a single check.

Another thing that can be optimized is the first part of the algorithm where we scan the image for the first pixel of a blob. In the first implementation each time a new blob is searched the image is scanned starting from the top left pixel. By storing the last position of the previous scan in two static variables, the subsequent scans avoid to check already checked pixels, and this using only 4 more bytes of RAM.

3 Trajectory Prediction

The task described in this part is to foresee the trajectory of objects, so that they can be identified in a consistent manner across several frames given as input in a stream fashion. In this part, we used the Kalman Filter in order to perform prediction on future positions basing on the previous observations. Input of this phase are the centroids coming from the previous object identification, while the output is an assignment of the objects to a consisent label allowing to identify them for the time they spend in the scene.

For the prediction step, we used the Kalman Filter, an optimal, recursive data processing algorithm. Kalman filter is considered optimal under some criterias, and is a quite simple and well performing algorithm that has been chosen given the constraints of the target architecture. In fact, one of the most interesting features of such filter is its recursive nature, which allows to store only a limited amount of information thanks to an assumption of Markovianity. This is particularly advantageous in all cases, like ours, in which memory and processing constraint devices must be used to perform any kind of data processing.

This filter can be used in several situations, but is especially good in estimating behaviors of noisy systems, in which the measurements taken cannot be trusted entirely. Also notice that Kalman Filter doesn't pose any limitation on the complexity of the state analyzed, thus allowing to incorporate several parameters in the system.

In the following, we will give a short introduction on how the Kalman Filter works, to move to briefly depict and explain the algorithm. Then we will move to explain how the trajectory prediction step has been implemented, both referring to the Kalman Filter implementation and to the assignment algorithm used to assign objects to predictions.

We will then explain the need of an optimization for the Kalman Filter and move to present the compared performances of the two versions, to finally show an overview of the quality of the system developed.

3.1 Introduction to Kalman Filter

As previously stated, Kalman Filter is an optimal, recursive data processing algorithm. The need of a filter for data processing is due to several factors, among which the fact that possibly some measurements of interest of the "real" whose data is to be analyzed cannot be sampled by the system or are to noisy to be reliable; even worst, often we don't have complete knowledge of the inputs driving the system of interest.

The Kalman filter provides a way to combine all available data of the system together with the prior knowledge of the system to produce an estimation of the desired state variables.

Assume we're given a state $y_t$varying in time, and a noisy measure $x_t$of such state. We can devise, by studying accurately the system, a model of how the state evolves. Such model will be a probabilistic one, since as we assumed before we might not know everything about the system; thus the likeliness that

a new realization takes a certain value will be in the form $p(y_{t+1}|y_t, y_{t-1}, ...)$. Notice that in general this model of state evolution needs to take into account all previous states.

However, in the Kalman Filter a Markovianity Assumption is performed, meaning that we assume that the Markovian Property (in words, given the present, the future is conditionally independent of the past) holds in our system. More formally stated: $\forall t, y_{t+1}, p(y_{t+1}|y_0, y_1, ..., y_t) = p(y_{t+1}|y_t)$. This will allow to have a great simplification of the system, since to produce an estimation of future value it will not be needed to take into account all the previous states.

Sadly, however, it won't be possible to measure y_t directly: instead, the filter will be fetched it's noisy correlated measure, x_t. The measurements performed will be related to the real value, but more noisy. Such values are also assumed to be conditionally independent given the actual state y_{t+1}. Formally: $p(x_{t+1}, x_t, ...|y_{t+1}) = p(x_{t+1}|y_{t+1})p(x_t|y_{t+1})...$

This means that the actual prediction of the future state at time t+1 will have to be performed using a conditional probability density function, in which the conditioning part are the sequence of noisy measurements fetched in the past to the filter. The basic idea under the hood of the Kalman Filter is to propagate such conditional probability density function, whose aim is to describe given the previous noisy measurements, what is the likeliness that the actual state had a certain value.

Then we can define on such conditional probability density function an "optimal estimation". Some common choices are the mean, the mode or the median.

Other reasonable assumptions performed by the Kalman Filter are:

- linearity of the system (or better, of the model used to describe it)
- the noise is white and Gaussian

While the first assumption is clearly made for the sake of tractability, the second is more difficult to explain. It is enough to know that white noise simplifies the mathematical tractability of the problem and it will just add a constant power on all frequencies, while looking exactly as the real wideband noise within the signal bandpass. Gaussianness is related to the amplitude of the signal, assumed to have a bell shape. Of course noise is in reality caused by several small sources, which may be assumed as independent random variables, that added together - as known - can be described very closely with a Gaussian probability density function.

In the following, how the Kalman Filter works will be explained by using a very simple example, to then move to the explanation of the actual generalized algorithm.

3.1.1 A classical example

This example is taken from Chapter 1 of Stochastic Models, Estimation and Control, Vol.1 by Peter S. Maybeck, and here written in a simplified and more abstract fashion.

Assume you want to measure at time t the location of a point $x(t)$, that you cannot know with absolute certainty. Somehow, you take an approximated measure z_1, whose precision is such that the standard deviation is σ_{z_1}. It is now possible to define the conditional density of the actual position based on the measured value as a Gaussian bell $N(z_1, \sigma^2_{z_1})$. Of course, the larger is the standard deviation the larger will be the uncertainty.

At the actual point, the best estimation that can be performed of the actual position $\hat{x}(t_1)$ will be z_1, while the variance of the error in the estimation will be $\sigma^2_x(t_1) = \sigma^2_{z_1}$.

When a second measurement z_2 is taken with a standard deviation σ_{z_2} in a point in time very close to t_1, an analogous conditional of position with respect to measurement can be calculated as $N(z_2, \sigma^2_{z_2})$. But we wish to combine the knowledge coming from both measurements together, in order to get a better estimation of the actual position of the measured point.

It is possible to show that, under the assumptions made above by the Kalman Filter, we have again a normal conditional density function $N(\mu, \sigma)$, whose parameters are:

$$\mu = \frac{\sigma^2_{z_2}}{\sigma^2_{z_1} + \sigma {z_2}^2} z_1 + \frac{\sigma^2_{z_1}}{\sigma^2_{z_1} + \sigma {z_2}^2} z_2$$

and

$$1/\sigma^2 = (1/\sigma^2_{z_1}) + (1/\sigma^2_{z_2})$$

Intuitively, the new average is a weighting of the measurements taken by their certainty. In fact, larger variance of the first measurement with respect to second means that the former measurement should be trusted - and thus weighted - less as we do by weighting it for $\sigma^2_{z_2}$.

Notice also how the new combined variance is littler than both the two measured ones, making the uncertainty of the estimate decrease as we would expect.

At this point, the best estimation $\hat{x}(t_2)$of the actual position of the point taken into consideration is obviously μ. Notice that this is both the mode and the mean of the conditional density function defined above.

We can now give a first definition of the Kalman gain at step t_2 as

$$K(t_2) = \frac{\sigma^2_{z_1}}{\sigma^2_{z_1} + \sigma {z_2}^2}$$

And, by simple manipulation of the previous formula for calculating the best estimation in a dynamic manner, which can be seen as a correction of the previous one:

$$\hat{x}(t_2) = \hat{x}(t_1) + K(t_2)[z_2 - \hat{(}x)t_1$$

However, in the previous we implicitly assumed that the point whose position we are trying to estimate has no dynamic, meaning that it doesn't move. What happens if, instead, we introduce motion into the system?

We can assume that the velocity of our system is in its simplest form, $\frac{dx(t)}{dt} = u+w$, where u is the velocity that we assume to have while w takes into account the uncertainty of u and is a white Gaussian noise with zero mean and variance σ_w^2.

As time proceeds, the estimation becomes less and less accurate: we can imagine the previous $\hat{x}(t_2)$ to move according to the speed u, while the conditional probability functions "spreads", meaning that its variance increases as we go farther in time from measurements, as the position is less and less certain with the passing of time. Just before a previous measurement z_3 is taken at time t_3, we will have a density described by the following estimations and variances:

$$\hat{x}(t_3^-) = \hat{x}(t_2) + u[t_3 - t_2]\sigma_x^2(t_3^-) = \sigma_x^2(t_2) + \sigma_w^2[t_3 - t_2]$$

We can updated our condition probability density function parameters with the new measure just as we did before for the static case:

$$\hat{x}(t_3) = \hat{x}(t_3^-) + K(t_3)[z_3 - \hat{x}(t_3^-)\sigma_x^2(t_3) = \sigma_x^2(t_3^-) - K(t_3)\sigma_x^2(t_3^-)$$

Where now the Kalman Gain $K(t_3)$ is defined as:

$$K(t_3) = \sigma_x^2(t_3^-)/[\sigma_x^2(t_3^-) + \sigma_{z_3}^2]$$

3.2 Kalman Filter in Practice

When moving to the simple example described above to a more complex one, we need to have a more complex state. Notice that the Kalman Filter theoretically doesn't pose any limit on the complexity of the state or of the way it evolves, as long as the evolution can be described by a linear system.

We will now have a vector $\hat{x}(t)$ representing the state of the system we are willing to measure, that will have as many dimensions as the components that we wish to use to represent the system. In a practical example, a simple model for a moving object in a 2D plane could be $\hat{x}(t) = \mid \ posX \quad posY \quad velX \quad velY \ \mid$

where the posX and posY are the components of the position and velX and velY are the components of the speed at time t.

However, such state can be more complicated and could, as an example, take into consideration also the acceleration.

In the previous example, the acquired knowledge of the system was propagated through the system with a couple, composed by the previous estimation and by the variance of the measurement, as a measure of the "uncertainty" contained in the system. When moving to a more complicated state, this is no longer just a value, but becomes represented as a matrix $P = cov(x - \hat{x})$, which is the error covariance matrix, and contains an estimation of the accuracy of the inferred state. This matrix will contain on the diagonal the variance of the measure w.r.t the actual value, and is thus an estimation of the uncertainty of measurement process.

In order to address the evolution in time of the system, we also need a model describing it, in terms of a linear operator, usually represented by a matrix F. Going on with the previous example, such matrix can be represented as follows:

$$F = \begin{bmatrix} 1 & 0 & \Delta t & 0 \\ 0 & 1 & 0 & \Delta t \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

As it is possible to see immediately, such matrix represents the evolution of the state in terms of position and speed.

In the previous, we implicitly assume to have some kind of knowledge of the system (i.e. the speed u at which the point taken into consideration moves) and some observations. In this case, we distinguish between observed values and modeled values. The observation matrix H is used in order to define which values can be observed in the system. In the case in which all components of the state are observable, the matrix will correspond to the identity. However, in general the matrix will be of rectangular shape; going on with our example, if we assume that we can only observe the x and y component of the position, such matrix will be made as follows:

$$H = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}$$

A more complex characterization is necessary in order to infer the noise of the system. In general, it will be different depending on the system actually measured. We will assume to have a vector Q representing the measured process noise covariance, which will be white and thus distributed as $N(0, Q)$. Another noise is used in the filter, and is the so-called observation noise, also white and with covariances in R. Values of Q and R can be found either by an accurate studying of the measurement equipment, by fine-tuning the filter or by some learning mechanism. In the following, we will assume that such values are given. In general these values may vary in time, however we will assume them to be constant.

Let's now proceed to explain how the algorithm actually works. First, let there be known that, although such distinction was not highlighted in the previous example, we can divide the filter behavior in two parts:

- the *predict* phase is responsible of providing an a-priori estimation based on the actual knowledge of the system. In this phase, the system evolution model F is applied to the current state to update the estimation as well as the P matrix. As previously explained, this will make the uncertainty of the process grow.

- the *correct* or update phase is responsible of updating the knowledge of the system using new measurements.

Predict This phase, is represented by two simple equations that respectively provide an update estimation and update the uncertainty measure. In the most general form, this part also contemplates the possibility of having a control model and a control vector, which is however neglected in the following since this feature has not been used in the described project and for the sake of simplicity.

The prediction at discrete time k is calculated as follows:

$$\hat{x}_p(k) = F\hat{x}(k-1)P_p(k) = FP(k-1)F^T + Q$$

Correct This is the most complicated phase in the calculation of the system state, as it performs a new calculation of the conditional probability density function carried along by the Kalman Filter and used for subsequent estimations.

The first component to be calculated is the so-called *measurement innovation*, a vector that, for every visible value of the state, calculates the difference between the estimation performed in the previous predict state and the actual measured value. This is done with the following equation:

$$y(k) = z(k) - H\hat{x}_p(k)$$

The subsequent step is calculating the innovation of the covariance matrix. Notice how to every component observed the measurement noise is added, thus representing the variance of the new measurement:

$$S(k) = HP_p(k)H^T + R$$

The most important step is calculating the Kalman Gain:

$$K(k) = P_p(k)H^T S(k)^{-}1$$

This matrix will then be used in order to update the state and uncertainty estimation with the following two equations:

$$\hat{x}(k) = \hat{x}_p(k) + K(k)y(k)$$
$$P(k) = (I - K(k)H)P_p(k)$$

Notice how the calculations performed in this step somehow recall the behavior described in the previous single dimension example.

3.3 Using Kalman Filter for Tracking

What has been described in the previous part refers to the optimal estimation of the state of a single object, or more precisely of its position. However, starting from this, it is possible to think to generalize such problem in order to perform tracking over a stream of images. The aim of this part is to describe how is possible to consistently track objects along a stream of images using the estimations provided by the Kalman Filter.

In the following, we will call tracks the path composed by all discrete time estimation produced by the filter for a single object along with some accessory information, while we will call blobs the input of this phase, that is more specifically a representation of a blob in terms of its centroids and bounding box. In this case, such centroids collection will be seen as a collection of measurements, which will be used in order to correct the estimations in subsequent step.

The object recognition part is assumed to be integrated in the system using a different functions, which returns a list of measurements and is explained in detail in the previous section.

This poses several challenges:

1. New objects might enter the scene, leading to the need to find a way to recognize such object and to initialize a new track for each of them
2. Objects might leave the scene, with the subsequent need to delete the tracks
3. It is necessary to find a way to assign measurements to predictions in order to perform corrections along the stream

When the stream of images starts, the first image fetched to the tracking system is "taken as is" and is used in order to initialize the tracks. In practice, for every blob measurement in the system, a new track is initialized, each one with its own id. Accessory information contained in a track state are:

- age
- identifier
- total visible frames
- consecutive invisible frames

The age of the track is used in order to decide whether to take into account for displaying or not. This is done in order to avoid noisy blobs appearing for just one frame to overly complicate the visualization of the system. The consecutive invisible frames component is used in order to delete tracks that have not been assigned to measurements for too much time, so that they don't perturbate the measurement of assignments in the following steps.

The system starts by an initialization phase, in which the output of a first object detection phase is used to initialize the tracks in the order in which objects are detected, thus without the assignment step.This allows to start the subsequent stream analysis, which is made of three main functions: predict, assign and correct. Such procedures have been developed in two ways, for reasons and in ways that will be clarified later.

Let us first start by the description of the prediction step, which is in fact the first step executed on every frame. This step is implemented, for every previously detected track in a very straightforward manner which accurately reflects the equations shown before. The tracks which have been invisible for too

much time (thus, referring in our assumptions to object that left the scene) are deleted, and their prediction is not performed. Another variation is the increment of the consecutive invisible frames, which will eventually be decremented in case of successful tracking in the correct phase executed later. Although the estimation is time dependent, we assumed that no frames are skipped and that the images are sampled at regular rate. The output of this phase is an updated estimation of the state of each object tracked, namely updated position and uncertainty.

Now that we have such values, the issue is assigning them to the correct objects detected in the scene, always taking into consideration that new objects might appear and other may leave.

To do so, it is necessary to perform some kind of assignment, which is responsibility of the assignment function. What we want is to minimize the cost of the assignment, which is represented by the difference between the measured position and the detected, noisy object position. To be more precise, there are several valid measurements for distance, like the taxi cab one, the euclidean distance or its more computationally friendly equivalent square value.

There are some notable algorithms achieving a global minimum cost of assignment, like the Hungarian one. This algorithm has complexity O(n^4), which is indeed an high one for constrained devices, as the one on which we operated. While in the original formulation it wouldn't allow to avoid assignments, it can be enriched so that it takes into consideration the cost of not performing an assignment, thus seeming suitable enough for the problem we are dealing with, at the cost of increased complexity.

In practice, such algorithm is overly complicated for our task, since we are not really interested in achieving a global minimum assignment: by assuming that estimations are quite accurate, what we really want to achieve is to assign every track to the closest new measurements, up to a certain threshold. This threshold shall be a parameter of the assignment algorithm and must be accurately calculated depending on the environment, of the nature of the tracked objects and on the precision of the measurements.

The algorithm we developed for this purpose is briefly shown in Figure 8.

Very briefly, the algorithm generates all the possible tracks n for blobs m assignment couples, and calculates their cost. Then it sorts such values in increasing order. Once the sorting has been performed, the vector containing all assignments and their cost is scanned linearly, saving the assignment whenever the flags for both the blog and the track involved are free. This cycle ends whenever the cost is greater than the unassignment cost, meaning that on average we expect the number of iterations of this step to be way smaller than the maximum $O(nm)$.

We can see that the complexity will be dominated by the cost of the Quicksort, thus leading to an average overall complexity of $O(nmlog(nm))$. While the first cycle will be executed completely, the second one iterating over the sorted values will probably skip most of them (unless all predictions and measurements are very near), thus making the algorithm very lightweight in practice. The occupied space is $O(nm)$ - a very small value in practice - plus the two flags

```
Assign(Tracks, Blobs, unassignmentCost) {
    distances[|Tracks| x |Blobs|];
    assignment[|Blobs|];
unassigned[|Blobs|];
    for t in Tracks:
        for b in Blobs:
            distances.add(t, b, distance(t,b));
    QuickSort(distances)
    for d in distances:
        if(d.cost > unassignmentCost) break;
        if(assignedTracks[t] == assignedBlobs[b]  == 0)
            assignedTracks[t] = assignedBlobs[b] = 1;
            assignment[b] = t;
    for b in Blobs:
        if (assignedTracks[t] == 0) unassigned.add(b);

    return <assignment, unassigned>;
}
```

Figure 8: Pseudo-code of the assignment algorithm.

arrays, occupying respectively m and n bytes. The space of the latter vectors could be greatly reduced by using a vector of flags in a bit fashion. However we decided that, due to the low space occupation of such vectors with respect to the vector containing all assignments and their cost, this was an unworthy space optimization which could have affected negatively the execution time.

Notice that this algorithm for assignment is, on average, more costly than the simplest one, in which for every track (or blob, alternatively), the smallest-distance blob is selected given that the cost of unassignment is not overtaken. However, we considered this approach to be more effective since the simplest one would induce a bias due to the scanning order in the assignment, which is particularly problematic in the case in which a track previously assigned to an object that has now left the scene is checked for assignment.

Output of this phase is a vector in which, for every blob, the position of the corresponding assigned track is stored. Another vector, containing all unassigned blobs is returned in this phase; elements in this vector are used to initialize new tracks, initally marked as invisible and that will try to seek the currently unidentified blob in the next element of the stream.

Finally we can move to the part in which the measurement performed on a blob is used in order to correct the position estimation of every assigned track, performed in the correct procedure.

In the correction step, the value of the consecutive invisible steps is placed to zero for the tracks on which it is called, thus avoiding the track to be deleted when the next frame is analyzed.

The correction for every track is performed in a straightforward manner: in the following we will briefly outline how it was optimized to avoid dealing with matrix multiplications, exploiting the knowledge of the system to reduce the amount of time to spend to compute such updates.

At first, the matrix of innovation of covariance (the matrix S in the previous section) is calculated. This is easily done, as we know that it is a two by two matrix because of the structure of matrix H. S will contain on each of its components the value contained in the corresponding position of $P_p(k)$, plus the measurement noise.

Also the calculation of the gain is made easier, thanks to the fact that the multiplication of H^T for $P_p(k)$ will result in a matrix of two columns and four rows. Inverting the S matrix is now easy, since we know that it is a two by two. Thus the Kalman Gain will have a 2 columns and 4 rows, exactly as we expected since we need it to weight the contribution of each measurement to the update of the state and of the uncertainty. It is now possible to update the state and the matrix P and to return updated tracks position, corrected velocity estimation and uncertainty and to proceed with the analysis of the next frame.

In the following, we will describe the results achieved by this tracking strategy from a performance point of view. Finally, we will move on to present a qualitative analysis of the outcome of two different implementations of the above described functions.

3.3.1 Performance Analysis

As some experimental results show, using floating points on our target architecture, on the SeedEye board, can be up to ten times more costly than using integer. However, the probabilistic nature of the Kalman Filter seems to suggest that indeed some non integer values should be taken into account for a better precision. This is the reason behind our choice to implement two versions of the various function previously described.

In the first version, we used a straightforward floating point implementation of the state, in which the nominal velocity was represented by two floats as well as the covariance, while the position of the tracked object was represented using short integers. In the latter optimized version, we reduced the precision by using a fixed point approximation, resulting in a worst precision in the identifying the tracks, but still achieving good results while speeding up the prediction and correction process. We also decided to implement a variation of the assignment procedure, which was developed by merely optimizing the process of metric calculation.

To develop the integer version of the Kalman Filter, we merely used a shift in base, making the standard unity worth $1/precision$. However, the matrix multiplication process involved in the correction step poses a severe limitation by this point of view. When assigning a variance to a measurement, in fact, the common suggestion is to use a very high value, in order to represent the high uncertainty of the process. However, when the Kalman Gain is calculated, for some values this could lead to intermediate values for which $K * precision^2$

are bigger than 2^32, the maximum value that we can represent using an integer number. To minimize this problem, a careful ordering of the operations is necessary (i.e. whenever a division or a change in base is involved, it is necessary to anticipate it as much as possible) and a careful tailoring of the precision and of the variance initial value is necessary. We will now proceed to evaluate the performance of the three main functions adopted in the tracking process, to then compare the two versions developed for each from a performance point of view, starting from the prediction.

Prediction The prediction step is one the most computationally light involved in our tracking solution. In fact, for a single track this part merely consists of a (small, 4x4) matrix multiplication and an update of the position values by the velocity, namely two multiplications and assignments. This part was further optimized thanks to the knowledge of the model, allowing to reduce to less than a quarter the number of operations involved, by encoding directly the matrix multiplication.

In fact, as we expect, the prediction time is very small in the floating point version as it will need on average 16.6 μs to be executed. Looking at the plot in 9 , we see that the execution time looks exponentially distributed, and we can see that, over the about 50000 values sampled, the experimental probability of terminating before 19μs is about 97%, and in no case we observed values greater than 24μs.

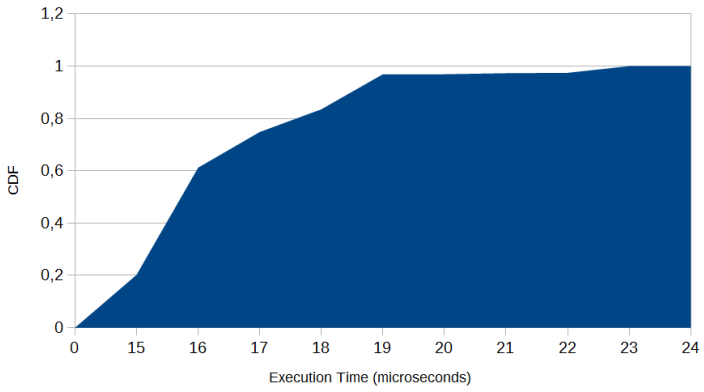

Figure 9: Cumulative Distribution Function of the prediction (floating point version) function performed on a single track

The version with using integers to emulate fixed point numbers performs a lot better, but still the advantage is not very clear in this very lightweight phase. In this case, for every track we were able to achieve an execution time which is on average of 7.8μs. As it is possible to see in 3.3.1, we actually have a probability greater than 50% of completing this phase in 7μs, while with a probability of about 99% the computation will take less than 9 seconds. In no

case we were able to observe values greather than 13 μs, which is actually less than the minimum (15 μs) achieved with the floating point version.

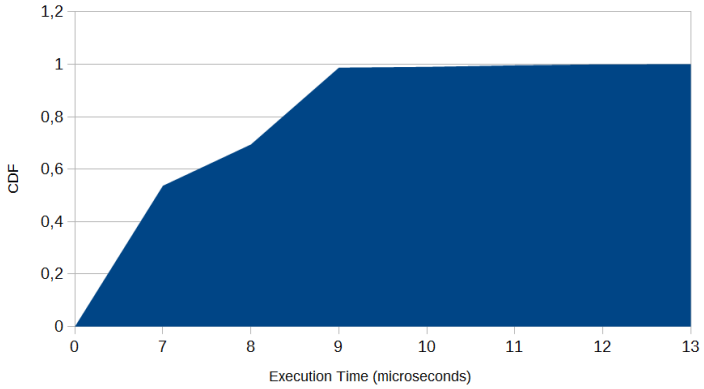

Figure 10: Cumulative Distribution Function of the prediction (fixed point version) function performed on a single track.

Overall, the prediction performed on all tracks behaves as we expect, assuming the shape of a gamma distribution, being the sum of the contribution of several exponentials. The two total prediction times are show in 11, depicting the fixed point execution time probability density function (in blue) and the floating point one (in red). From the plot, it is possible to see how the mass of probability of the optimized version performs significantly better, having an average of 94μs against 183μs of the unoptimized one.

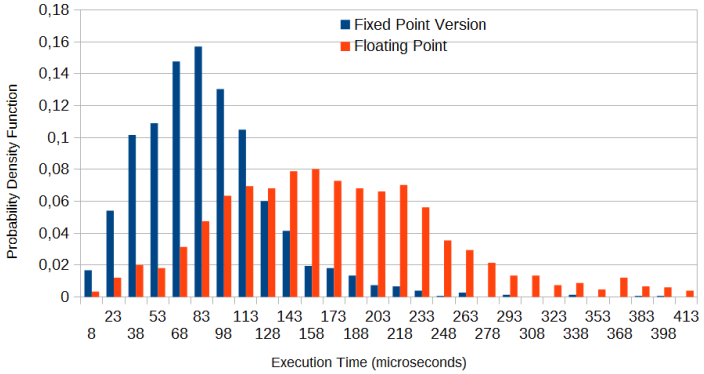

Figure 11: Comparison between the execution time probability density function of the fixed point and floating point version of the prediction step on all the states.

Assignment The assignment algorithm has been developed in two fashion: in the former, we used as distance measure the classical Euclidean Distance between the predicted point of each track and the actual measurements performed in each step; in the latter we exploited the fact that the square root operator is strictly monotone to avoid to use the costly `sqrt` operator, just returning the square of the Euclidean Distance.

As said before from a simple average analysis of the assignment algorithm, its complexity in time is $O(nmlog(nm))$, where n is the number of tracks and m is the number of blobs. Alternatively, it has complexity $O(klogk)$ if k is the number of possible assignments. We can see in 12 that the plot, in which we confronted the execution time of a whole assignment procedure against the number of possible couples, supports our hypothesis. Another interesting feature shown in 12 is that values are very concentrated near to small execution times, while cases in which k - and thus the execution time - are huge are very rare, as confirmed by 13, which shows how, with very high probability, the execution time of this algorithm will be under two milliseconds.

The average execution time of this version of the assignment algorithm is about $1704\mu s$, which could still be improved, as in fact we did in the other version.

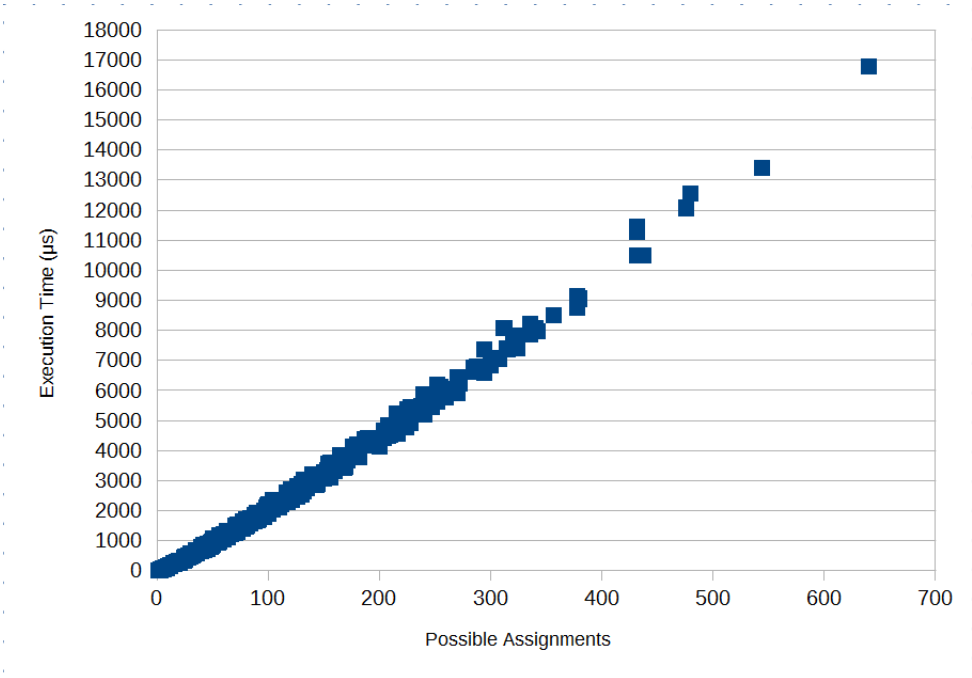

Figure 12: Execution time in μs of the assignment procedure using euclidean cost metrics versus the number of possible assignments

In the case in which the `sqrt` was avoided, we were able to achieve better

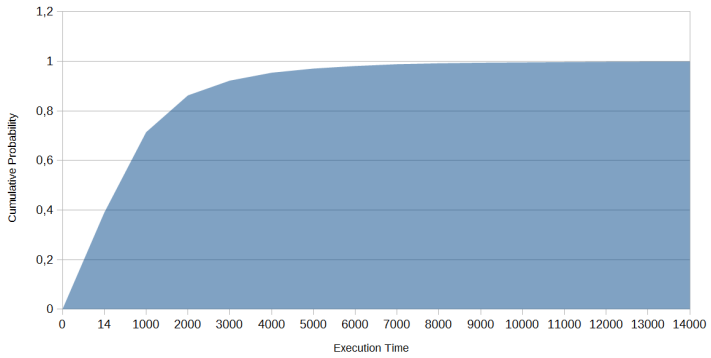

Figure 13: Cumulative distribution function of assignment execution time using the Euclidean Distance metric, resembles an exponential distribution. Time is in microseconds.

average time for the assignment process, which is however more irregular with respect to the previous case as we shown in 14. However, this could be explained with the fact that the complexity $O(nmlog(nm))$ is an average case one, dominated by the Quick Sort. Still, it is possible to have several variations in the actual complexity of the algorithm depending on the input, thus explaining the noise.

A better view of the behavior of this procedure is given by the cumulative probability distribution function of the execution time, which once again assumes the shape of an exponential one as shown in 15. With very high probability, the execution time for this version will take less than 600 μs, thus improving of an order of magnitude the previously explained version. The average execution time is, in this case 662μs.

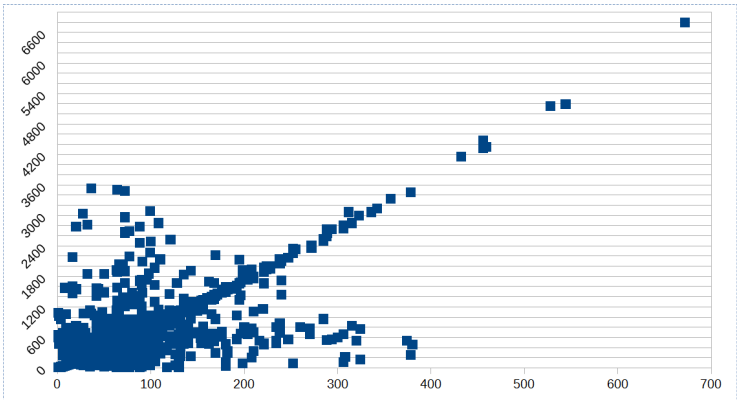

Figure 14: Execution time in μ s of the assignment algorithm versus the total number of admitted couples in the case in which the distance metric was the square of the Euclidean Distance.

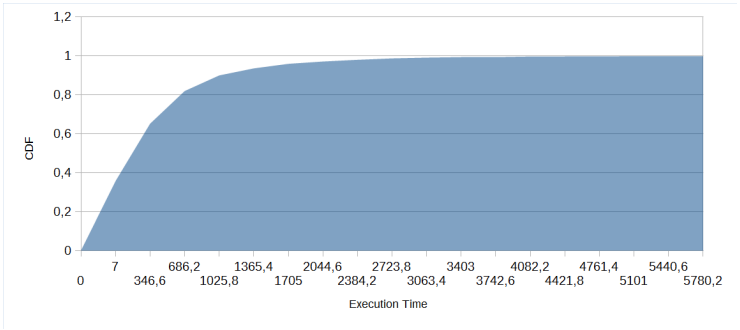

Figure 15: Cumulative Distribution Function of the assignment execution time.

Finally, in 17 it is possible to see how this second version outperforms significantly the first one, as the values will be concentrated near smaller values with a much higher probability.

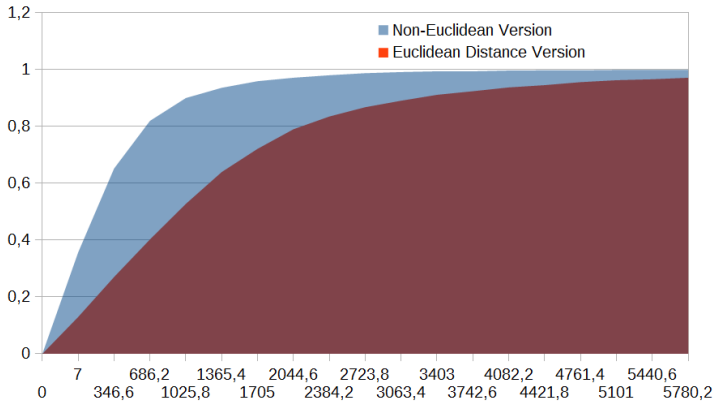

Figure 16: Comparison between the two assignment versions developed, in terms of cumulative distribution function of their execution times (on the x-axis, in microseconds).

Correction The correction step is the most costly one involved in the Kalman Filter calculation. This is due, in both versions, to the huge amount of operations that must be performed in order to calculate the gain and to update the matrix of the error covariances.

The correction execution time, in the floating point version of the filter and for a single couple composed by a track and a blob, takes on average 130.26μs. From the experimental results, it is not clear whether the distribution of the execution time behaves like a Gamma function, as can be seen in ... . For sure, we can state that the probability that the execution time will take less than 133μs is above 80% in our sample of more than 36000 calculations.

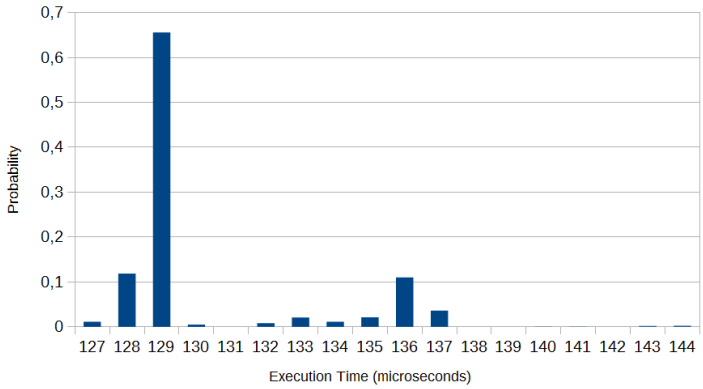

Figure 17: Probability density function of the execution time of a single update step in the floating point implementation

Although these numbers don't seem impressive, specially when compared to the ones resulting from the assign step, we have to take into count the fact that they are for a single track-blob couple, and thus may result in a execution time of the whole correction phase way more costly.

The fixed point version, instead, behaves better exactly as we expected, achieving an average time for correct function of 46μs, and assumes a more reasonable shape of the density of an exponentially distributed density function, as it is possible to see in 18. As we can see, the maximum calculation time experienced is less than half of the minimum of the previous implementation.

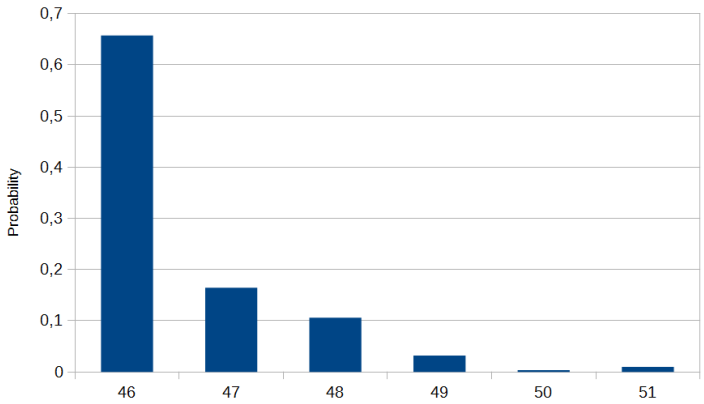

Figure 18: Probability density function of the execution time of a single update step in the fixed point implementation

From the point of view of the whole phase of correction, thus performed on all tracks and measures, we can show how the fixed point implementation led to great benefits, moving from an average value of 958μs to a more reasonable one of 330μs. In 19 is possible to see the density of the execution times for the two implementations of this phase.

We can see how in the fixed point case, the probability that the phase takes less than 500μs is of about 70%, while in the floating point case this execution time will be met in less than 15% of the cases. Both the curves resemble a gamma distribution, as we could expected given the nature of the smaller correction functions involved in the calculation.

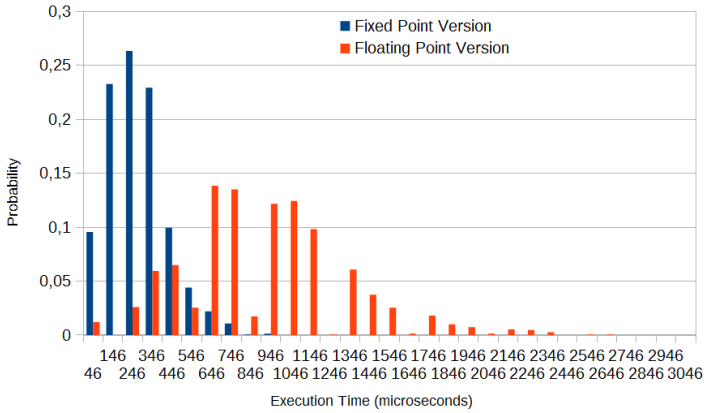

Figure 19: Comparison between execution times of the floating point(red) and fixed point(blue) probability density functions.

Combined Analysis In this part we will briefly discuss the performance of the previous three parts combined, in order to understand what kind of problems the part of the system responsible of consistently tracking previously extracted objects along frames may address. We intentionally avoid here to discuss the performance of the labeling part, as this process might be performed separately.

In 20 it's possible to see the variation of the execution time for all the operations on a frame plotted against the number of tracks and the number of blobs recognized in it. As it was already clear from the previous analysis, we can see that the slowest component, dominating the execution time is the one in the assignment process, as it is possible to see from the quadratic execution time behavior [1].

[1] Previously, we stated that the assignment time has complexity $O(nmlog(nm))$. However, if we assume that $n \simeq m$, as it is in our application, this complexity becomes $O(2n^2log(n)$

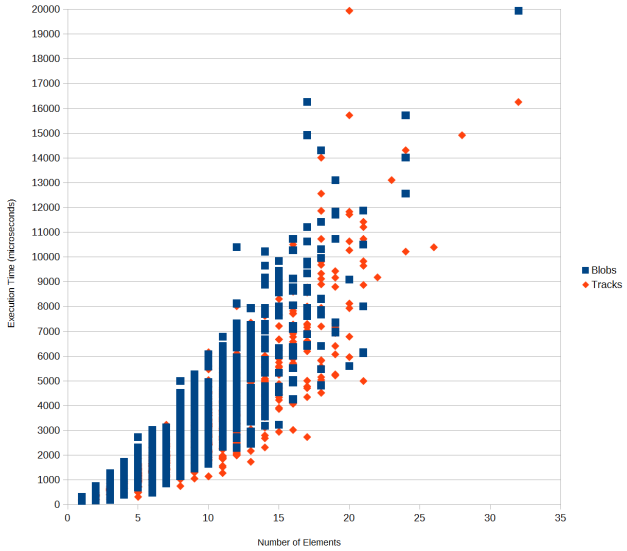

Figure 20: Number of tracks and number of blobs in the current tracking step against the execution time of the whole step

This poses a limitation in the maximum number of tracks and objects that can be dealt with, as an increase in the order of magnitude will notably affect our service time T_s. An analysis can be performed in a easier manner by taking into consideration the number of possible assignments k in each frame as a normalizing factor of the overall service time.

We can assume that the completion time, with respect to the number of couples, will behave as

$$\hat{T}_s(k) = T_{couple} * k * log_2(k)$$

The T_{couple} parameter will depend on several factors, and thus its estimation has been done experimentally on the gathered samples, finding that a good value for this parameter could be in the interval of $[4, 7.5]\mu$s.

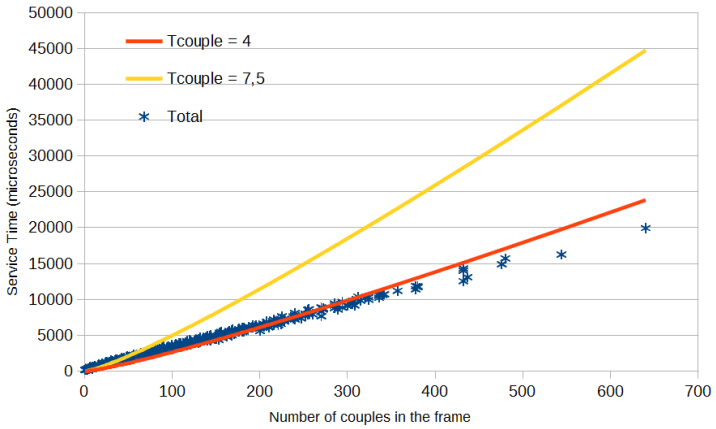

Figure 21: Experimental service time (in blue) against the theoretical cost model with different processing time for each couple.

In the plot 21, we show the experimental total execution time for the number of couples in the assignment against the theoretical cost function, with different values of T_{couple}. Even though values around 4 minimize the Mean Square Error, we will use $T_{couple} = 7.5$, which has a greather (but still comparable to the variance) error, but gives a good overestimation of the service time also for a small number of couples.

With such value, we can see how, if we wish to process at a frame rate of 30fps, we can track approximately up to 4400 couples; since we expect the number of blobs and tracks to be almost the same, we can process images containing no more than 66 objects.

In the case of the optimized version, the distribution of the costs is more fuzzy, making difficult to investigate a precise cost model. However, we postulate that it will be approximately similar, but with a lower T_{couple}, due to the minor impact of integer operations on the completion times. Further investigation will be required to analyze this behavior.

On the testing data set, in which the maximum number of couples is 640, we were able to find that the density of the service time shown in 22 resembles one of a Gamma distribution function. These results are considered very satisfying, as the overall probability of using less than 2000μs is about 0.9, meaning that we expect our service time to be within this range, which would allow to process images at a very high frame rate, however taking into account that this will hold only for a number of objects in the order of 25 units.

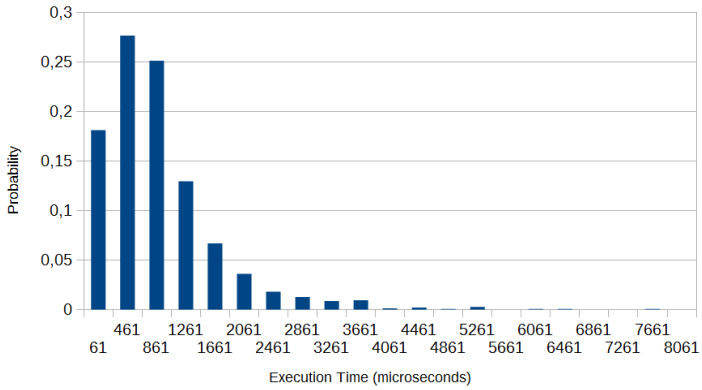

Figure 22: Density function of the total service time in the fixed point version

3.3.2 Qualitative Analysis of Tracking

To assess the quality of the tracking procedure we developed, we needed to create some sort of convention. The following rules have been followed and used in order to create a (small) gold standard to be used in order to compare the tracking in different processes:

- We neglect the non-recognition of objects appearing/tracked only for one frame, that we assumed to be noise as we did in the algorithm
- We don't consider an error when a change in label occurs after separating tracks: while the human eye will be able to understand easily to which of the two "separating" parts the label should be assigned to be consistent, without more precise assumptions about the nature of the object tracked
- We also don't consider an error when a track is lost due to the merge in the images of different, previously separated blobs

To assess the precision of the algorithm, we used these assumptions in order to create a small gold standard calculated over 50 frames, against which we compared the precision of our tracking procedure, using both the float and the integer approximated version. When evaluating "errors", we followed the criteria of evaluating the overall goodness of the detected track with respect to the one detected by the human eye.

This means that, as an example, if an object previously tracked is assigned to a new track or to a different track, and however this track is kept constantly, this will be counted as a single error. Consistently, as it in fact occurs in our data set, if the object previously wrongly assigned is then re-assigned after some steps to its previous track, this will count as another error.

The resulting precisions are depicted in the following table:

Notice how such analysis is not very accurate, as it is carried on on a very small sample. This is justified by the fact that the creation of the gold standard

	Integer Version	Float Version
Number of blobs for the analysis	240	240
Number of images	50	50
Wrong Images[2]	20	2
Wrong Assignments	34	2
Error probability	14,17%	<1%

has revealed himself to be very costly. Also, we noticed that errors, especially when involving wrong assignments, tend to propagate and must be hand-checked for correctness in several cases, as under the assumptions we used the correctness check is very costly.

However, a better evaluation of the performances of these two versions of our tracking system can be seen by giving a look at the videos that can be found here:

`https://dl.dropboxusercontent.com/u/5545859/FloatVideo.mp4`
`https://dl.dropboxusercontent.com/u/25611509/VideoInt.mp4`

4 Conclusion and Future Works

We now wish to analyze fully our procedure, composed of the labeling part with the task of identifying objects in the frames and the Kalman Filter + Assignment part with the task of consistently track the objects along the stream.

The most computationally heavy part of the whole system is the labeling one, as it is possible to see in 23, in which the total service times (in light red) are shown decomposed by their 4 constituting parts, namely: the labeling and the calculation of centroids (in blue), the assignment (in yellow), the correction (in green) and the prediction (in red). As obvious, the part of the process dealing with the analysis of the images and the identification of objects is the heaviest one, thus predominating on all the other components by orders of magnitude. Of course this applies both in the case in which the filtering part was carried on with floating point and with fixed point approximation.

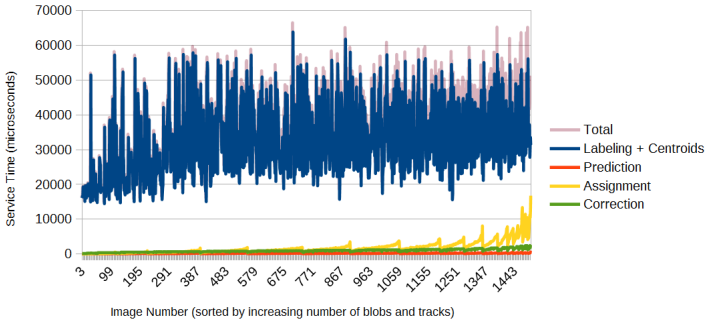

Figure 23: Contribution to the total service time of every phase in the tracking system

The service time density function shown in 24 has the typical shape of a Gamma one, as we would expect given the nature of the labeling process and its dominance over the other components. Even though we experience huge peaks in completion time, looking also at the cumulative distribution function in 25 we can see that in a soft real-time environment we are able to support a frame rate of 20fps with a probability of about 91%, meaning that the probability of skipping a frame is less than 10%. Even in the worst case, given our simulation a 10fps image stream can be for sure elaborated, given that the maximum total service time we experienced has been of 66564 μs.

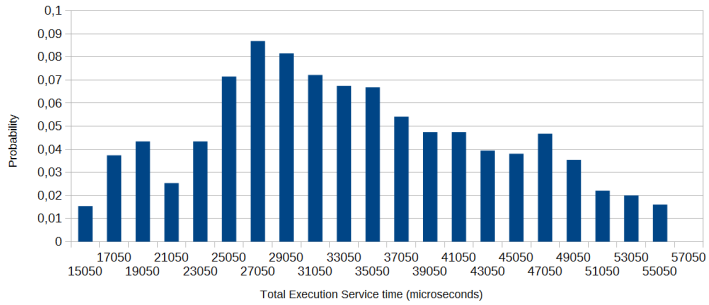

Figure 24: Density function of the total service time.

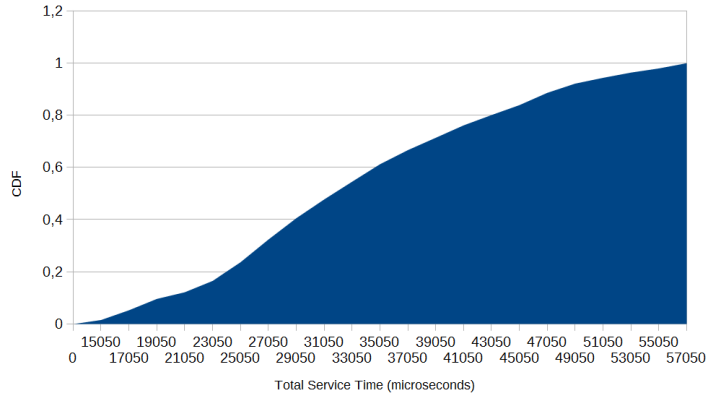

Figure 25: Distribution function of the total service time.

Higher frame rates could be also supported, provided that the interruption of a frame elaboration can be tolerated by the final application.

Of course, the introduction of this possibility undermines one of the assumptions used in the Kalman Filter implementation, for which measurements inter-arrival times are supposed to be constant, thus possibly affecting the precision of the procedure. Still, given the littler contribution of the Kalman Filter parts in the implementation of the whole tracking mechanism, we can imagine more complicated models for this part, as an example taking into account also the bounding box in the object in the state model. Also the assignment part could be improved, by introducing some more complex metrics allowing possibly to cope with cases in which a blob separates: assuming that the tracked bodies won't change very fast their shape, we could take into consideration also the area of the bounding box in evaluating the cost of the assignment.

This could be especially good in the case in which the second part is seen as a separated component, receiving directly a stream of measurements: being this

part not the bottleneck, the remaining time would be spent waiting for input, thus lowering the efficiency. A more complicated filter and assignment could be devised for this purpose.

From the point of view of the labeling, it seems a good idea to lower further the execution time of the labeling part. At the present time, we did not exploit the 32-bit architecture on which we operated, and instead we only accessed and operated on the memory containing the images in a bit wise fashion. This could possibly led to a performance increase, thanks to a better cache exploitation and to an enhanced usage of the 32-bit CPU. This would need a further approach change, as the image would be checked by blocks of 32 pixels instead of by single pixels.

A further enhancement of this part that we seek to investigate in the future is the usage of down scaled images, with a lower resolution and thus a smaller service time for the process of retrieving the blobs: since subsequent steps only need coarse grain information, a loss in resolution wouldn't be critical and, instead, it could also enhance the filtering part by removing some noise filtering out some useless blobs caused by noise in the previous steps.

In conclusion, we advocate that tracking is definitely possible in a high frame per second embedded systems in a soft real time fashion, when low resolution images and very high precision is not critical.