ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Swiss Federal Institute of Technology, Lausanne
Signal Processing Laboratory

Multiple feature extraction for robust image sequence segmentation

Diploma project

Diego Santa Cruz

Assistant: Roberto Castagno

February 1997

Abstract

For the second generation based coding of video sequences, like MPEG-4, it is necessary to obtain a robust segmentation where the belonging of regions to objects is preserved across frames.

A feature space approach, in which the local characteristics of the image are treated altogether, should constitute a robust method of accomplishing such segmentations.

In this project several features are studied that can be used in such an approach. The main ones are the local average and local standard deviation.

To avoid the edge blurring that normally occurs when applying an averaging filter, a method is developed, based on the sigma filter, that preserves the edges while eliminating the noise. Also several methods are developed to reduce the border problem that appears in the estimation of the local standard deviation.

The studied features are tested with vector quantization as a basic segmentation algorithm. The ones used are not only based on the gray level but also on the chrominance components, which significantly increases the robustness of the segmentation.

This set of features is also applied to the segmentation of sequences achieving good tracking of regions across the frames without any special strategy. This enables the approach to be used in partially supervised segmentation schemes.

Resumé

Dans le monde d'aujourd'hui, où les télécommunications et le multimédias sont en plein essor, il est très souvent nécessaire de compresser l'information. Un des types de données les plus importants dans ce domaine ce sont les images et plus particulièrement les séquences vidéo.

A présent de nouvelles techniques de compression sont en train d'être developpées, basées dans des techniques dites de deuxiéme génération, telles que MPEG-4. Dans ces techniques l'image est divisée en plusieurs régions de forme arbitraire qui sont codées séparement. Pour augmenter au maximum le taux de compression les régions sont prises comme des aires homogènes à l'intérieur des objets. Dans MPEG-4, dont la standarisation est prévue pour fin 1998, la séparation se fait aussi au niveau d'objets permettant ainsi de les manipuler séparément dans l'image codée.

Pour pouvoir appliquer une telle strategie de codage il est nécessaire de disposer des séquences vidéo segmentés, où les régions à coder sont déjà determinées, de même que l'appartenece de celles-ci aux différents objets. A fin d'obtenir de telles séquences de façon automatique, ou semi-automatique, une méthode de segmentation robuste est nécessaire, où l'appartenance des régions est maintenue entre les différentes trames. Ces dernières exigences font de ceci un problème difficile pour lequel une bonne solution n'a pas encore été trouvée.

Une approche interéssante est celle du *feature space* ou plusieurs caractéristiques locales de l'image sont traitées en parallèle par des méthodes de filtrage vectoriel.

Dans ce projet nous avons cherché les caractéristiques (*features*) qui sont les plus adaptés à une segmentation robuste et à poursuivre les régions entre les trames d'une séquence.

Les caractéristiques utilisées sont la moyenne locale et l'écart moyen local. Néanmoins les méthodes d'estimation de ces paramètres ont été modifiés pour s'affranchir des problèmes inhérents à ces caractéristiques.

La moyenne locale à le grand inconvénient de rendre flous les bords des régions. La méthode employé, basée sur le filtre sigma, selectionne les pixels, sous la fenêtre de calcul, ayant les mêmes caractéristiques communes. Seulement ces pixels rentrent dans le calcul de la moyenne. Comme résultat le moyennage élimine le bruit et rend plus homogènes les régions tout en respectant parfaitement les bords.

L'écart moyen local a pour ça part un autre inconvénient lié aux bords. Lorsque celui-là est calculé pres d'un bord l'estimation de l'écart moyen donne des valeurs beaucoup plus grandes que les normales, mais ne représentant aucune texture. Ceci

est du au saut du niveau de gris qui apparaît dans le bord, qui ne constitue pas vraiment une texture. Une méthode semblable à celle utilisée pour la moyenne locale, mais plus complexe, permet de réduire notablement l'effet des bords sans pour autant affecter la "mesure" des textures.

Ces caractéristiques, apliquées au niveau de gris aussi bien qu'aux chrominances, donnent des segmentations de bonne qualité dans des différents types d'images. Mais plus important encore, elles sont presque invariantes dans le temps, ce qui permet de poursuivre facilement les régions entre les trames.

Les caractéristiques trouvées constituent une bonne base pour devélopper un algorithme de segmentation bien adapté. Pour nos tests de segmentation nous avons utilisé un algorithme de quantification vectorielle que malgré le fait qu'il ne soit pas un vrai algorithme de segmentation était assez bien adapté pour tester la robustesse des caractéristiques étudiés.

De plus une librarie et des programmes, représentant plus de 6000 lignes de code, ont été developpés et seront integrées avec celles du laboratoire pour être utilisées dans le futur.

Acknowledgments

I would like to thank all the people at the Signal Processing Lab for their invaluable advices and support without whom this project would have not been possible.

In particular I would like to thank Roberto Castagno, my teaching assistant, for all his wise advises, good ideas and for the revision of this report.

Also thanks to Pascal Fleury for all the suggestions concerning the source code of the programs, the support of the `LTS`*Platform* (the programming platform of the lab) and for being there whenever problems occurred.

And last I would like to thank Jean-Marc Vesin for his support and for following the progress of the study.

Contents

1 Introduction

1.1 Second generation coding: MPEG-4

In our days data compression is very important for transmission and storage purposes. One of the main types of data to compress, specially concerning transmission, are the images and particularly the video sequences. Several compression standards exist today, some of them being H.261/H.263 and MPEG-2. In these coding standards frames are split in rectangular blocks, which are then coded independently.

Currently another type of coding is being developed, the so called second generation coding. These new schemes work by coding arbitrarily shaped regions of the image separately, and not just rectangular blocks as in the first generation ones. The splitting of the image in uniform regions enables for higher compression rates for given distortion and subjective quality.

Here appears the necessity to make a clear distinction between objects and regions. Objects have a semantic signification, for example a car, a person, a tree, etc. An object is divided into regions, which are uniform areas. A region can be, for example, a part of an object with same light and color.

One can think of a two-level partition of images: the first, in which semantically meaningful objects are separated, is the object layer and the second, where the image is further divided into regions which are homogeneous according to other criteria (e.g. color and/or gray level), is the region layer. A region should then belong to only one object. The regions is what one normally obtains from a segmentation.

MPEG-4 is a standard, to be released by the end of 1998, that is based on second generation coding. This standard works at the object level rather than at the region one. One of the key elements of MPEG-4 is its object scalability: every object is coded separately and can also be decoded independently of the other objects.

The object scalability enables for a multitude of different applications like:

- easy editing of a sequence, even by the end user,
- content based search (i.e automatically search in a sequence the part where a particular object appears),
- graceful degradation for transmission over bandwidth limited channels (i.e different objects are coded with different quality),

In order to achieve this object scalability is necessary to have the sequence segmented at the region and object levels before the coding is done. It is therefore necessary to have an automatic or semi-automatic system that segments a sequence at both levels. This is a problem very difficult to solve, where a lot of research has been done and is still being done.

In this project we will study the possibility of doing the segmentation in the feature space, and also study its robustness.

1.2 Supervised segmentation and object tracking

The process of separating the objects in an image is trivial for humans but it actually requires a lot of previous knowledge, experience and training. We know what the objects are and because of this we are capable of merging several regions into an object without any effort.

In the computers we have nowadays, even if they are extremely powerful, it is impossible to incorporate all the intelligence and all the visual experience humans have. Therefore it is very difficult to come up with a system capable of segmenting a video sequence and merging the regions into objects in a fully automatic way. On the other hand computers are more suited to divide the images in regions with a given criteria (e.g coding efficiency, gray level)

A partially supervised approach appears as a promising intermediate solution to the segmentation problem. It consists in telling the computer, by means of a human interface, which regions belong to the same object. However the assistance to the computer should be minimal so as to make the system usable in practical situations.

Once the computer knows the correspondence between regions and objects for one frame it will repeat the segmentation for the following ones maintaining the same attribution of regions to the semantic objects.

This robustness of the segmentation also achieves object tracking since the correspondence across frames of the regions constituting an object are well established for the whole scene.

1.3 Feature space approach

Some of the existing methods of segmentation take into account the local characteristics of the image such as gray level, motion information or texture, independently,

and eventually merge the results of the different segmentations. Other methods use these local characteristics, or features, according to a competitive or iterative scheme. Because of the processing of each feature being done at a different stage in the segmentation the system is not very robust.

In the feature space we associate a vector of features to each pixel of the image and this vector becomes an entity in its own, instead of having a collection of features for an image that are considered as separate things (as in the other approaches). The processing of the features is then done in parallel by means of vector-based operators, such as vector filtering or vector quantization.

With this approach, and provided appropriate sets of features are selected, it should be possible to obtain a robust method for general purpose segmentation that can achieve object tracking across several frames in a straightforward way.

In the following sections we will first present the features used and how they are calculated. The results of segmentation based on these features is presented in Sec. 4.

2 The feature space

2.1 Local average

As far as segmentation is concerned one of the most significant characteristics of an image is the gray level or luminance. Two drawbacks of directly using the gray level as is found in the image are that it is normally not uniform inside the regions and that it can be noisy. This makes its direct use for segmentation difficult.

Therefore it is important to find a method to eliminate the noise of the gray level so that it can be used for the segmentation. Two different approaches exist to overcome this difficulty. The first one consists in using an averaging filter, or other low-pass filter, which effectively removes the noise. The second one consists in starting the segmentation with the image at a very low resolution and improve it iteratively using higher resolutions. The details of these methods and the problems encountered with them are explained below.

As explained above, the first solution consists in taking the average of the gray level over a small window centered on the pixel of interest, as given in eqn. (1) for a $M \times N$ window, with M and N odd, where (x_{ij}) is the gray level image. Usually one uses odd sized windows so that the center pixel is well defined; most used sizes are 3×3, 5×5 and 7×7. Because averaging is a kind of low-pass filtering the higher spatial frequencies (including noise) are attenuated.

$$\bar{x}_{ij} = \frac{1}{MN} \sum_{\substack{k=i-(M-1)/2 \\ l=j-(N-1)/2}}^{\substack{k=i+(M-1)/2 \\ l=j+(N-1)/2}} x_{kl} \qquad \text{the local average} \tag{1}$$

In the following we will refer to this feature, local average, as **avg**.

In Fig. 1 we can see an image in gray level and its local average calculated with a 5×5 window. The result is much more uniform inside regions, however there is a major drawback: the image is blurred so that the precise edges in the original image are lost. Therefore this information it is not directly usable in a robust algorithm.

In the multi-resolution methods, also called pyramidal methods, one forms low-resolution images from the original image by replacing, for example, a square of four pixels with their average (Fig. 2). Then we start the segmentation at the lowest resolution image and then improve it with the next higher resolution image and continue this process until we get to the original image.

Figure 1: original image (left) and average, *avg*, gray level with 5×5 window (right).

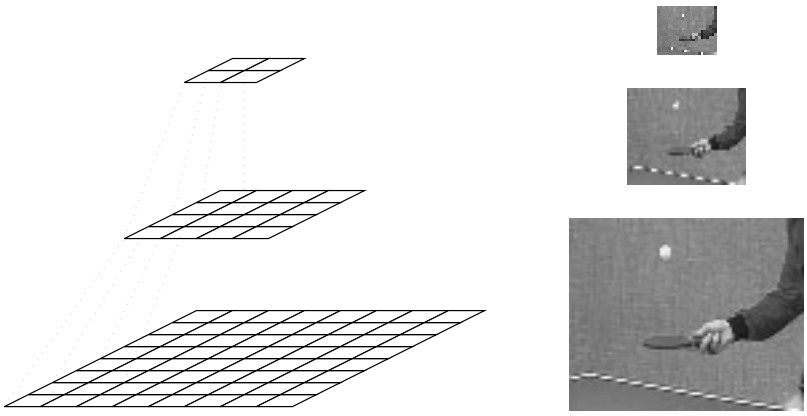

Figure 2: multi-resolution approach example.

These multi-resolution approaches are very complicated: going from a coarse segmentation towards a finer one it is not a trivial problem. Here we want to elude this kind of complications by using the average information at a full resolution level but without loosing the edges. In Sec. 2.4 we will show a method to solve the edge problem based on a sigma filter [1].

The local average characteristics are not only useful when calculated on the gray level, even if this last one is the most important. In fact when the images to process are in color we can use the local average of the chrominance components in the segmentation process. As we shall see having the average on three color components makes the segmentation much more robust.

2.2 Local variance and standard deviation

Besides the average gray level, there are other useful types of information about an image. One of them is the fact that different objects can have different textures. There are several "measures" of this texture information. The simplest one is the local variance.

The local variance is calculated over a $M \times N$ window (M and N odd) centered on (i, j) as given by

$$\mathrm{var}(x_{ij}) = \frac{1}{MN} \sum_{\substack{k=i-(M-1)/2 \\ l=j-(N-1)/2}}^{\substack{k=i+(M-1)/2 \\ l=j+(N-1)/2}} (x_{kl} - \bar{x}_{ij})^2 \tag{2}$$

where $\bar{x}_{ij}$ is the local average calculated as in eqn. (1) and (x_{ij}) is the gray level image on which we calculate the features.

However there are some problems with this kind of feature as seen in Fig. 3, an aerial photography of the San Francisco bay, where the variance value is represented as a gray level (a 0 variance corresponds to black). This actually looks more like the result of an edge detection algorithm than a "texture detection" one.

The problem comes from the fact that when the variance is calculated over a region edge the step change in gray level makes the estimation to take values more than 40 times larger than the values obtained over very textured regions. As a result, the interesting information about textures is lost.

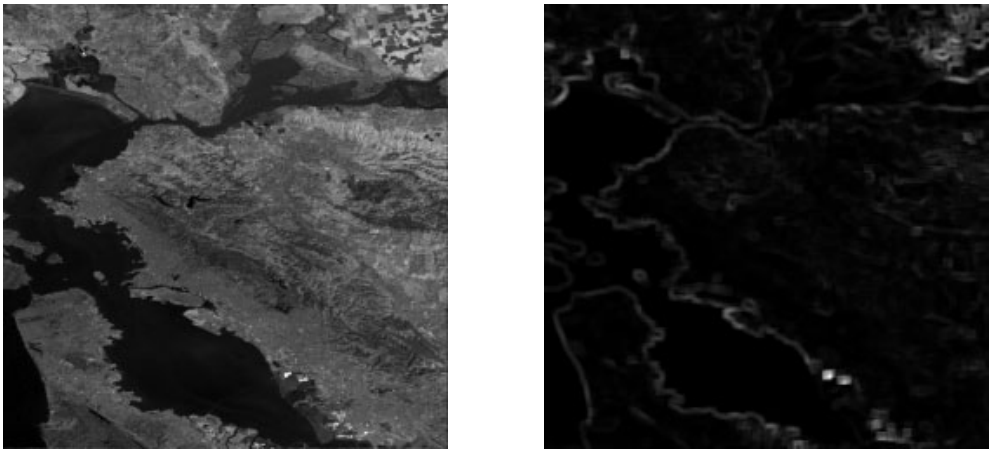

Figure 3: original image (left) and local variance, *var* (right), over a 5×5 window (right).

An alternative measure is the local standard deviation, or local sigma noted σ_{ij}, which is nothing more than the square root of the local variance.

$$\sigma_{ij} = \sqrt{\text{var}(x_{ij})} \tag{3}$$

These features will be designated as **var** for the local variance and as **std** for the local standard deviation.

The standard deviation obtained for the image in Fig. 3 is shown in Fig. 4. As we can see, the land and the water have a clearly different texture and this is reflected in the local standard deviation values. The feature is then representative of the texture but the problems with the borders still remain. This makes this feature not a robust one because the borders are considered as separate from the objects they actually belong to.

Another alternative is to use the logarithm of the local variance as given by eqn. 4. However this has the drawback of making small changes in variance, which are not representative of textures, too important while the border problem is not solved. An example is shown in Fig. 4 for the image of Fig. 3.

$$y_{ij} = \log_{10}(1 + \text{var}(x_{ij})) \tag{4}$$

In the following sections some solutions to the border problem will be shown: one based on the gradient information (Sec. 2.3) and another one based on the sigma filter (Sec. 2.5).

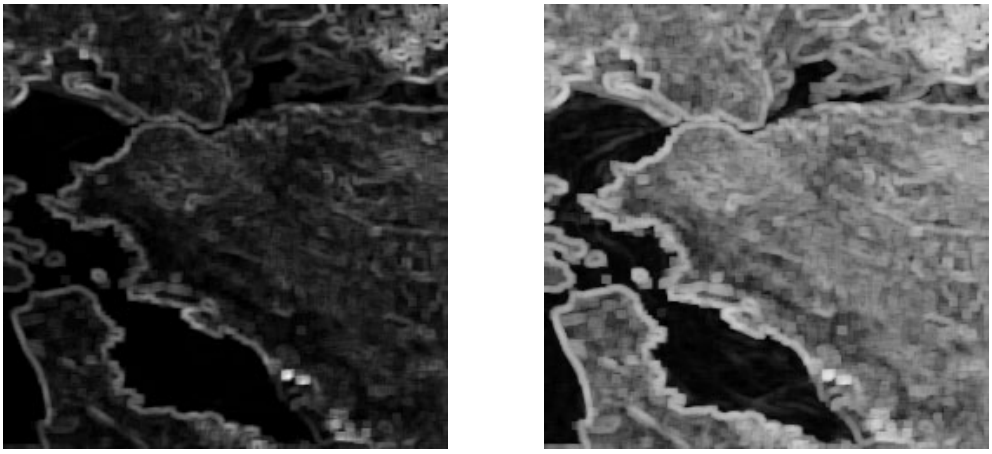

Figure 4: local standard deviation, *std* (left), and logarithm of the local variance, *var* (right). Both calculated over a 5×5 window.

To differentiate between textures with different orientation one could use the variance calculated over lines or columns, or even diagonals, of the window (a directional variance). The horizontal variance, that is the average of the variance along the lines of the window, which would reflect a vertical texture, can be calculated over a $M \times N$ window (M and N odd) as

$$\mathrm{hvar}(x_{ij}) = \frac{1}{MN} \sum_{\substack{k=i-(M-1)/2 \\ l=j-(N-1)/2}}^{\substack{k=i+(M-1)/2 \\ l=j+(N-1)/2}} (x_{ij} - \mathrm{havg}(x_{ij}))^2 \tag{5}$$

where

$$\mathrm{havg}(x_{ij}) = \frac{1}{N} \sum_{l=j-(N-1)/2}^{j+(N-1)/2} x_{ij} \tag{6}$$

The other directional variances can be calculated in the same way but exchanging lines and columns, or even diagonals and anti-diagonals. However the same problems that appear with the non-directional variance or standard deviation appear here, but the solutions are very similar.

2.3 Local variance with gradient information

As we have seen in the previous section the variance and standard deviation information is useful for identifying textures. However the border problem is very

1	2	3	4	5
6	7	8	9	10
11	12	13	14	15
16	17	18	19	20
21	22	23	24	25

21	22	13	4	5
16	17	8	9	10
11	12	13	14	15
16	17	18	19	20
21	22	23	24	25

Figure 5: mirroring across a border: before (left) and after (right).

annoying.

In order to solve the problem one has to calculate the variance only with the pixels lying inside the window that belong to the same region as the center pixel. In that manner the step change in gray level across edges does not affect the variance estimation. The problem is not trivial since knowing to which region each pixel belongs is knowing a segmentation of the image, which is actually our objective.

However, since the windows used are small compared to the size of regions there is normally no more than one edge under the window. One can then use the gradient information to detect the presence of an edge and decide to which side of the center pixel it lies so to separate the two regions. Once these regions are identified we replace the pixels lying on the "bad" region by mirroring across the border the pixels in the "good" region (the one that contains the center pixel). Finally we calculate the variance over the obtained window after the mirroring as explained in Sec. 2.2.

An example of this mirroring technique is shown if Fig. 5 where the border pixels are dark shaded and the mirrored pixels are light shaded.

The algorithm as described above looks quite simple. In reality is much more complicated because it is not clear from the gradient information which pixels are on an edge and which not. A high value of the gradient's magnitude reveals the presence of a border, but what is high enough? One has to fix some thresholds, decide on a majority basis, etc. A full description of the proposed algorithm can be found in the source code of the `feature` program.

In the implementation we used the Sobel operator to obtain an estimate of the gradient. The matrices used for the Sobel horizontal (S_h) and vertical (S_v) operators are

$$S_h = \begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix} \qquad S_v = \begin{pmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{pmatrix} \tag{7}$$

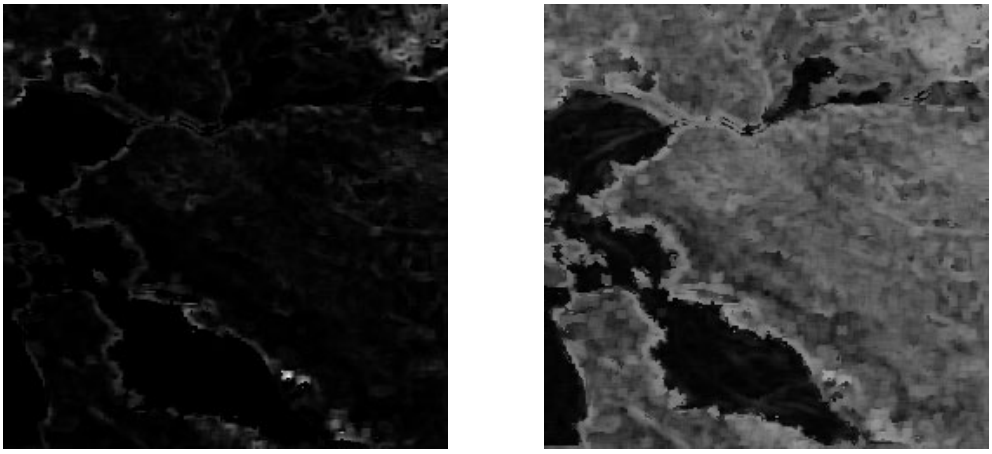

Figure 6: local variance with mirroring, *svar*, in normal (left) and log scale (right), over a 5×5 window.

and the magnitude of the gradient is then obtained by

$$|\text{grad}_{ij}| = \sqrt{x_h(i,j)^2 + x_v(i,j)^2} \tag{8}$$

where $x_h(i,j)$ and $x_v(i,j)$ are the results of filtering the image $x(i,j)$ with S_h and S_v respectively ($**$ is the two-dimensional convolution).

$$\begin{aligned} x_h(i,j) &= x(i,j) * *S_h \\ x_v(i,j) &= x(i,j) * *S_v \end{aligned} \tag{9}$$

In the following we refer to the local variance and standard deviation, calculated with the above method, as **svar** and **sstd** respectively.

We applied the algorithm explained above to the image with the gradient's magnitude obtained from the Sobel operator. An example of the results obtained for the local variance are shown in Fig. 6 for the image of Fig. 3. Another example is shown in Figs. 7 and 8 for another image.

In the first of these examples there is an improvement over the normal calculation (as seen in Sec. 2.2, Figs. 3 and 4), however the result is still not acceptable for a segmentation application since the edge problem remains important. In the second one there is an improvement in some areas but there is an unacceptable deformation of fine details in some other zones (e.g the face).

The reason for the persistence of the edge problem is that the borders between regions are not a step change in gray level but more a gradual one. As a consequence

Figure 7: original image (left) and its local variance, *var* (right), over a 5×5 window.

Figure 8: local variance, *var* (left), and local variance with mirroring, *svar* (right). Both in log scale and calculated over a 5×5 window.

the presence of the pixels lying on the edge unduly increases the value of the variance. A solution could be to shift the detected edge towards the center pixel and to mirror the pixels from further away. This could work for large regions were there are pixels of the same region far way of the edge, but for thin objects this method would be catastrophic because pixels from still another region would be taken into account leading to even more disastrous results.

Another problem of the technique employed here is that small details get destroyed by the mirroring because the axis of mirroring it's not a straight line, and also because sometimes there could be two edges under the window, making the technique to fail.

As we have seen this is not a good solution for the edge problem. Another technique based on a sigma filter, and that works much better, is explained later in Sec. 2.5.

2.4 Local average and sigma filter

As explained in Sec. 2.1 the local average is a very useful feature but has the drawback of blurring the edges. Here we will present a method to calculate the local average without blurring edges that works perfectly well, based on sigma filters [1].

The blurring of edges comes from the fact that the average is taken on all the pixels under the window without any discrimination of regions. The idea of the sigma filter is that only the pixels under the window whose value lie within a predetermined range, the *inliers*, are taken into account while the *outliers* are left out. The range is centered on the center pixel's value. This computation can apply to luminance as well as chrominance values.

In the original version of the sigma filter the range is fixed as $[x_{ij} - 2\sigma, x_{ij} + 2\sigma]$, were σ is the standard deviation of the value across the whole image. The 2σ range comprises, for a Gaussian distribution, the 95.5% of pixels and is large enough to include most of the pixels of the same region, but at the same time small enough to exclude those from other regions.

In the version we use the modified range is $[x_{ij} - 2\sigma_{ij}, x_{ij} + 2\sigma_{ij}]$, where σ_{ij} is the local standard deviation as calculated in eqn. 3. This has the consequence of blurring the edges still less than the standard sigma filter. Nevertheless when we approach very sharp edges, where the local sigma (σ_{ij}) is very high, we have to limit the range so no blurring will occur. The maximum allowed σ_{ij} is then fixed as 2 times the average of σ_{ij} throughout the image.

Mathematically this can be put as

$$\delta_{kl_{(ij)}} = \begin{cases} 1\ , & \text{if} \quad (x_{ij} - \Delta_{ij}) \leq x_{kl} \leq (x_{ij} + \Delta_{ij}) \\ 0\ , & \text{otherwise.} \end{cases} \tag{10}$$

where

$$\Delta_{ij} = \begin{cases} \sigma_{ij} \quad , & \text{if} \quad \sigma_{ij} \leq 2\bar{\sigma}_{ij} \\ 2\bar{\sigma}_{ij}\ , & \text{otherwise} \end{cases} \tag{11}$$

and

$$\bar{\sigma}_{ij} = \frac{1}{IJ} \sum_{\substack{i=0 \\ j=0}}^{\substack{i=I-1 \\ j=J-1}} \sigma_{ij} \tag{12}$$

where I and J are, respectively, the height and width of the image.

In other terms $\delta_{kl_{(ij)}}$ is 1, meaning inlier, when x_{kl} is inside the range relative to x_{ij} as the center pixel, or 0 otherwise, meaning outlier (the $_{(ij)}$ subindex means relative to x_{ij}).

Then we take the average, over a $M \times N$ window (M and N odd), only on the pixels marked as inliers. Or mathematically

$$\hat{x}_{ij} = \frac{1}{S_{ij}} \sum_{\substack{k=i-(M-1)/2 \\ l=j-(N-1)/2}}^{\substack{k=i+(M-1)/2 \\ l=j+(N-1)/2}} \delta_{kl_{(ij)}} x_{kl} \tag{13}$$

where $\hat{x}_{ij}$ is the local average calculated with the sigma filter and S_{ij} the number of inliers inside the window centered on (i, j),

$$S_{ij} = \sum_{\substack{k=i-(M-1)/2 \\ l=j-(N-1)/2}}^{\substack{k=i+(M-1)/2 \\ l=j+(N-1)/2}} \delta_{kl_{(ij)}} \tag{14}$$

In the following we will refer to this feature, the local average calculated with a sigma filter, as the **sigavg** feature.

Figure 9: local average with sigma filter, *sigavg*, over a 5×5 window.

Two examples of the result of this filtering are shown in Fig. 9, taken on the images of Figs. 1 and 7. It is clear that with this method we get the advantage of the average inside the regions without sacrificing the edges, which is very important for our purposes: a robust segmentation. The good performance of the method is demonstrated by the fact that even small details, such as the eyes in one of the examples, are not modified by the averaging. We get smooth regions with sharp borders, ideal for segmentation based on gray and chrominance average values.

Yet another modification that could be introduced in the basic sigma filter is the *K-limit*. When the basic sigma filter is applied sometimes almost all pixels in the window are marked as outliers and thus it does not average some isolated pixels. What one can do to solve this problem is that when the number of inliers is less than a fixed number, the *K-limit*, the result is replaced by the average of the immediate neighbors of the center pixel.

Mathematically eqn. 13 would be replaced by

$$\hat{x}_{ij} = \begin{cases} \frac{1}{S_{ij}} \sum\limits_{\substack{k=i-(M-1)/2 \\ l=j-(N-1)/2}}^{\substack{k=i+(M-1)/2 \\ l=j+(N-1)/2}} \delta_{kl_{(ij)}} x_{kl} \ , & \text{if} \quad S_{ij} \leq K \\ \\ \frac{1}{8} \sum\limits_{\substack{k=i-1 \\ l=j-1 \\ (k,l)\neq(i,j)}}^{\substack{k=i+1 \\ l=j+1}} x_{kl} \ , & \text{otherwise} \end{cases} \tag{15}$$

The K limit has to be carefully chosen so not to wipe out any fine details. Nevertheless, even with a small K (like 2) there are some parts of thin lines that get wiped out. This is a big drawback of this strategy that is not acceptable for our segmentation purposes so it is actually not used.

2.5 Local variance and sigma filter

The application of the sigma filter to the calculation of the local average gives, as we have seen, very satisfactory results. One could then apply the same strategy to the calculation of the local variance and local standard deviation. By analogy it would consist in calculating the local variance or local standard deviation on the pixels marked as inliers only, according to

$$\widehat{\mathrm{var}}(x_{ij}) = \frac{1}{S_{ij}} \sum_{\substack{k=i-(M-1)/2 \\ l=i-(N-1)/2}}^{\substack{k=i+(M-1)/2 \\ l=i+(N-1)/2}} \delta_{kl_{(ij)}} (x_{kl} - \hat{x}_{ij})^2 \tag{16}$$

and

$$\hat{\sigma}_{ij} = \sqrt{\widehat{\mathrm{var}}(x_{ij})} \tag{17}$$

where $\widehat{\mathrm{var}}(x_{ij})$ is the local variance calculated with the sigma filter, $\hat{\sigma}_{ij}$ the local standard deviation, also calculated with the sigma filter, $\hat{x_{ij}}$ the local average calculated with the sigma filter as in eqn. 13, and $\delta_{kl_{(ij)}}$ and S_{ij} are calculated as in eqns. 10 and 14 respectively. In the following we will refer to features calculated in this way as **sigvar** for the local variance and **sigstd** for the local standard deviation.

Here the limitation of the range for high σ_{ij}, as explained in the previous section, is very important. If such limitation is not done, pixels from different regions, under the same window, are taken into account in the calculation and the results are not much better than those obtained in Sec. 2.2.

An example of this calculation is shown in Fig. 10 for the image of Fig. 1. We can see that the border problem has been greatly reduced in comparison with the calculation of the variance over the whole window as in Sec. 2.2, and there are no problems of deformation of the regions as with the calculation with the gradient as shown in Sec. 2.3.

Even though the border problem is greatly reduced using the sigma filter it is not completely solved. Thin lines of high variance are still present. This is due to the fact that when the center pixel lies on the border its value is normally an intermediate one between the values of the two adjacent regions. As a consequence the range of accepted values for the sigma filter includes some pixels of one region and some of the other and the variance estimation is incorrect. Unfortunately if the range is further reduced, so that no pixels of the adjacent regions are included,

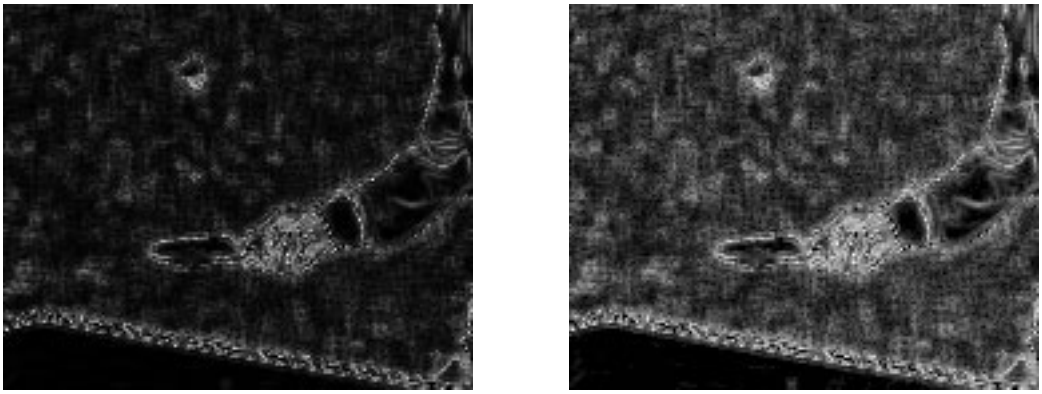

Figure 10: local variance, *sigvar* (left), and local standard deviation, *sigstd* (right), both calculated with a sigma filter over a 5×5 window.

there are no inliers left but the center pixel. As a result the estimated variance would be 0 what it is not a good result either. One solution to this problem that uses post-processing is presented in Sec. 2.6.

Another problem that appears inside some regions with texture, like the background in Fig. 10, is that the variance does not correspond to the one calculated without the sigma filter. This is caused by a large amount of pixels being marked as outliers. In order for them to be marked as inliers we have to increase the limit of the accepted range, however this will have the consequence of increasing the edge problem.

To solve the latter difficulty yet another modification of the sigma filter can be used. It consists in marking the outliers based on the *sigavg* instead of on the image itself (i.e the pixels are considered as inliers if their *sigavg*, and not their original value, lies inside the range). However the calculation of the variance is done on the actual image data (i.e the original values).

This last modification can be stated as

$$\delta^{(a)}_{kl_{(ij)}} = \begin{cases} 1\ , & \text{if} \quad (\hat{x}_{ij} - \Delta_{ij}) \leq \hat{x}_{kl} \leq (\hat{x}_{ij} + \Delta_{ij}) \\ 0\ , & \text{otherwise.} \end{cases} \tag{18}$$

where Δ_{ij} is calculated as in eqn. 11 and $\hat{x}_{ij}$ is the local average calculated with the sigma filter as in eqn. 13 (the superscript (a) states that the marking of outliers is done on the local average calculated with the sigma filter).

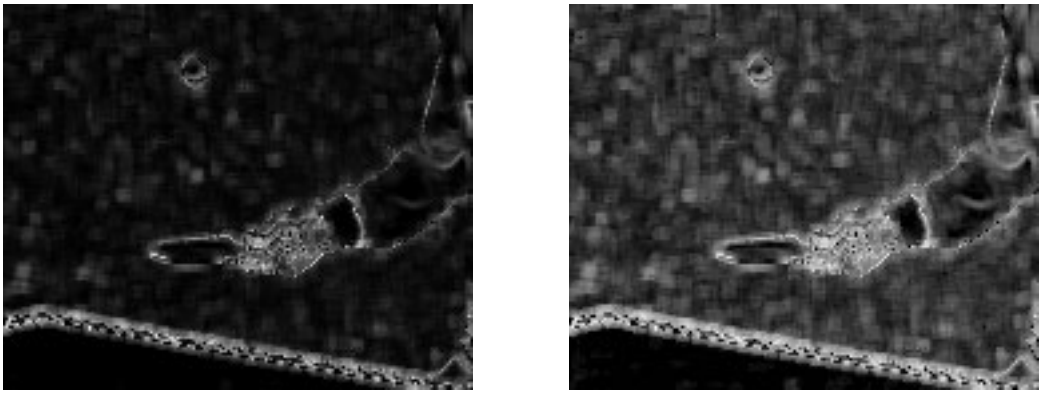

Figure 11: local variance (left) and standard deviation (right), calculated as *sasigstd* and *sasigvar* respectively. The window used is 5×5.

Then the new local variance and standard deviation are calculated as

$$\widehat{\mathrm{var}}^{(a)}(x_{ij}) = \frac{1}{S_{ij}} \sum_{\substack{k=i-(M-1)/2 \\ l=i-(N-1)/2}}^{\substack{k=i+(M-1)/2 \\ l=i+(N-1)/2}} \delta^{(a)}_{kl_{(ij)}} (x_{kl} - \hat{x}_{ij})^2 \tag{19}$$

and

$$\hat{\sigma}^{(a)}_{ij} = \sqrt{\widehat{\mathrm{var}}^{(a)}(x_{ij})} \tag{20}$$

In the following we refer to the features calculated with this method as the **sasigvar** for the local variance ($\widehat{\mathrm{var}}^{(a)}(x_{ij})$) and as the **sasigstd** for the local standard deviation ($\hat{\sigma}^{(a)}_{ij}$).

An example of the result is given in Fig. 11. One can see that now the variance inside the regions is correctly calculated and that the edge problem is reduced in certain areas.

Another problem with these methods to calculate the local variance that has not been mentioned yet is that there are some isolated black spots (i.e variance 0) at some places, like the border of the table in Figs. 10 and 11. This occurs not because the local variance is actually 0 but because all pixels in the window are marked as outliers with the exception of the center pixel, which is always an inlier, and a variance calculated over only one element is always 0.

It is not possible to solve this problem by modifying the calculation process of the local variance because if only one of the pixels marked as outliers is included the

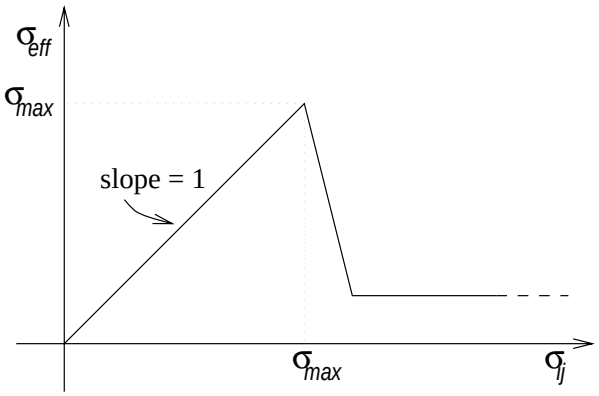

Figure 12: function to calculate effective sigma to use with the sigma filter.

resulting variance is very high due to the difference of the pixels' values. However this problem can be solved with some post-processing as it is explained in the following section.

In addition to what is explained above there were some other modifications of the sigma filter that were tried, to improve the final result. However none of them were satisfactory.

One of those strategies was to use an effective sigma related to the local sigma, σ_{ij}, with a function like that of Fig. 12. That is if σ_{ij} was less than a limit, σ_{max}, the effective sigma, σ_{eff}, was equal to the former, but if it was greater than the limit, σ_{eff} would be much smaller. In this manner when the window would be over edges the range of accepted pixels would be much smaller and less pixels would be taken into account.

Unfortunately this last method doesn't work any better than just limiting the maximum σ_{ij} allowed as it was done before. That's why we abandoned this last idea.

Another idea was to pass the result of calculating the local variance with the sigma filter trough a 3×3 sigma filter. The result is very deceiving since the edges are completely blurred and the improvement on the values of the variance is not significant. So this idea was abandoned too.

As a last example we show in Fig. 13 the result of calculating the standard deviation as *sigstd* and *sasigstd* for the San Francisco bay image. Compared to the simple calculation of the local standard deviation (without sigma filter nor gradient information), shown in Fig. 4, the result is significantly better, specially when the marking of outliers is done on the local average (calculated with the sigma filter).

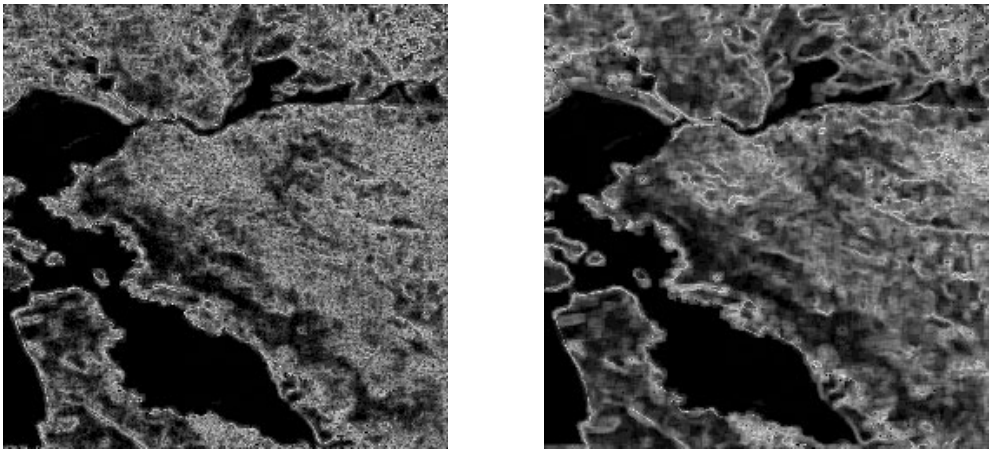

Figure 13: local standard deviation calculated calculated as *sigstd* (left) and *sasigstd* (right). The window size used is 5×5.

The only remaining problem is that some "black spots" appear due to all neighboring pixels being marked as outliers. However this is not a big problem with respect to a segmentation, because a good segmentation algorithm merges isolated pixels into the neighboring regions.

2.6 Median and sigma filters applied to local variance

As we have seen in the previous section the application of the sigma filter to the calculation of the local variance and standard deviation significantly improves the quality of the estimation. Unfortunately this is not enough to completely solve the edge problem, even though this one is greatly reduced.

In order to reduce even more the problem one can use some post-processing. What we have to remove from the result of calculating the local variance with the sigma filter are the thin lines of high variance and the isolated pixels of 0 variance. One useful filter to remove this kind of thing is the median filter.

The two dimensional median filter orders the pixels by their value and then picks up the one lying in the middle. In that way if in the window there are some pixels with a high value it does not affect the result as it would be the case with an averaging filter. The window size need not to be large because the purpose of the median filter here is to remove the thin lines of high variance that may appear. The 3×3 window used is enough to accomplish it.

We will refer to the feature obtained by applying the median filter on *sigstd* as

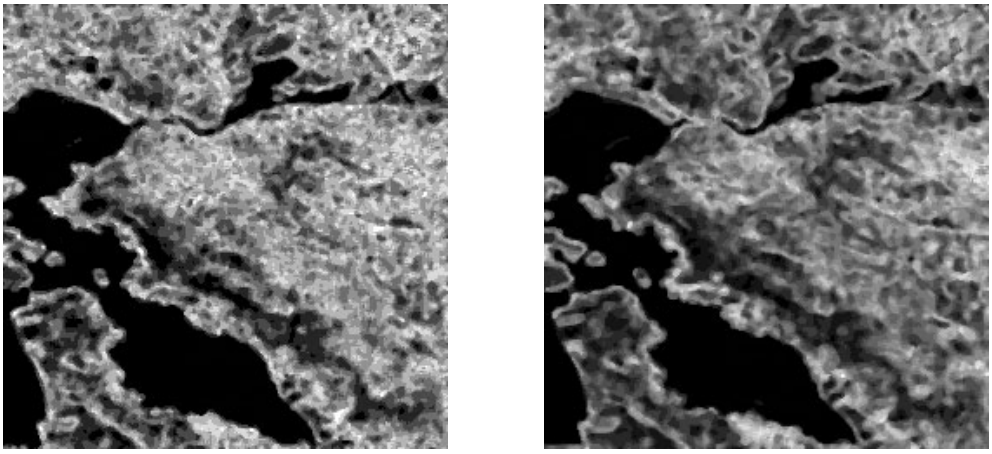

Figure 14: application of a 3×3 median filter: local standard deviation calculated as *sigstdmed* (right) and *sasigstdmed* (left).The window size used is 5×5.

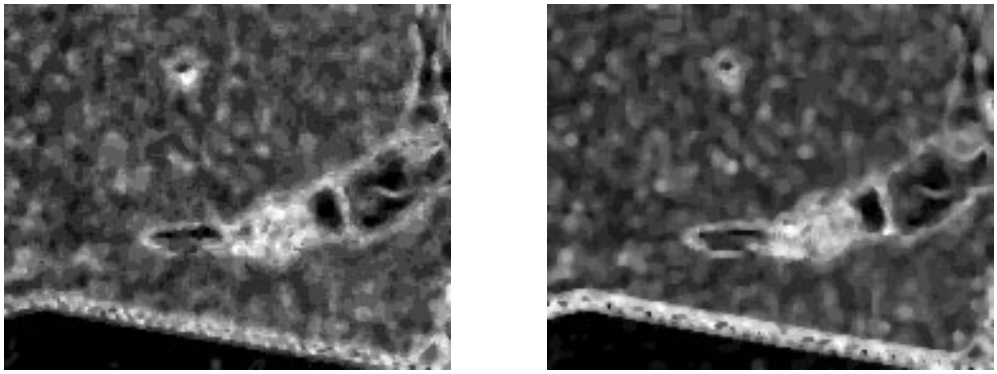

Figure 15: application of a 3×3 median filter: local standard deviation calculated as *sigstdmed* (right) and *sasigstdmed* (left).The window size used is 5×5.

sigstdmed, and to the one obtained by applying the median filter on *sasigstd* as **sasigstdmed**.

Some examples of the results obtained are shown in Figs. 14 and 15. As one can see all the thin lines of high variance, coming from edges, and the "black spots" are removed. However edges are shifted and sometimes small details are lost. It should be up to the user to make a trade-off, based on the application, between applying or not the median filter, accordingly with the wanted precision for the borders.

3 The features implementation

For the features explained in the previous sections some programs and mainly a reusable library were developed in C++ using the LTS*Platform*, a programming platform designed at the Signal Processing Laboratory (LTS) at EPFL.

In the following sections we explain the most important aspects of the developed library and programs, and also some off-the-shelf programs.

3.1 The LTS*Platform*

When designing new algorithms it is necessary to validate them. In the image processing field this is normally done by software simulation. As a consequence it is necessary to write a program to do simulation.

A normal procedure to do that is to write a piece of software from scratch, or almost, debug it and only at the end concentrate in the algorithm part. The fact that the basic parts have to be written and debugged every time represents an enormous waste of time.

As a solution to this problem a programming platform, called the LTS*Platform*, has been developed at the lab in the C++ language. Its object oriented design makes it possible to reuse modules, or objects, just having to know how it can be used and not having to know in detail how it is done.

A detailed explanation of the LTS*Platform* cannot be given here but the brief one given below should suffice to understand the following sections. For more detailed information about the LTS*Platform* refer to [3].

All the basic data structures, like strings, vectors, matrices, several image types and many others, have already been implemented; and also some basic functions like image reading and writing to disk, etc.

For the algorithm implementation part, all objects should derive from `Arrow` or `Box` objects. This is inspired from block diagrams: arrows are data carrying objects and boxes are processing units. As an example an image derives from `Arrow` and a DCT transform from `Box`.

An arrow doesn't do anything, it just stores the data in a convenient form. The box does the processing but doesn't store any data (at least permanently). How can one then change the behavior of a box? In the LTS*Platform* one can create a set of

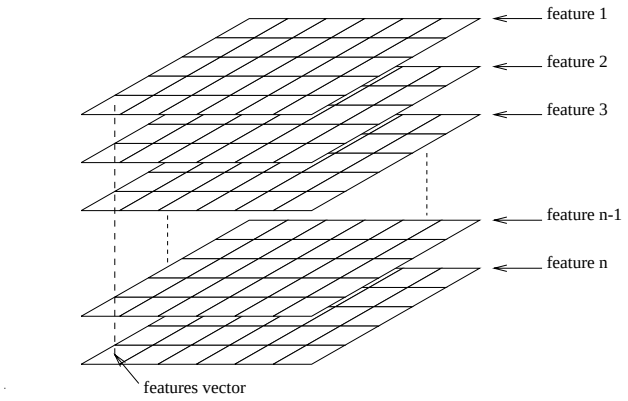

Figure 16: the `FeatureMatrix` structure.

parameters for a box, their values being set before it starts working. In this way one can influence the algorithm.

3.2 The `FeatureMatrix` arrow

First of all what we need in the program is an object to store the features data (like local average, standard deviation, etc.). A class, called `FeatureMatrix` was created for this purpose which inherits from `Arrow` since it is a data storage object.

The `FeatureMatrix` class is mainly a matrix of vectors. Each element of the matrix corresponds to a pixel of the image, so the matrix and the image have the same size. The feature data is stored in the vectors, elements of the matrix, as one feature per position in the vector.

One can alternatively think of this structure of as a stack of "images", each "image" containing the information about one feature, as shown in Fig. 16.

The vector elements are `floats` so there are no overflow or precision problems. However, if more precision is needed, it is possible to change this to `double` by just changing a definition.

This structure is well suited for the task of storing and processing the data. However this is not enough, since it is necessary to know which feature is at which position in the vectors, with what window size it was calculated, on which color component, etc.

For that purpose there are auxiliary data structures in `FeatureMatrix`. These are arrays containing complete feature specifications, one for each position in the

vectors. The information they hold is: the feature type, the size of the window used (horizontal and vertical) and the color component on which it is calculated (luminance Y, chrominance U or V).

This feature specification data it is not directly accessible by the user. It is accessed by some functions that let find a feature in the `FeatureMatrix`, know which feature is at which position, and set the feature information. However the feature data is directly accessible by the user.

A feature type is defined for each feature discussed in the previous sections (i.e local average, local average with sigma filter, local variance, etc.). If one thinks of another feature type it is possible to include it by just adding a type definition for it (detailed instructions are found in the source code). The algorithm is defined in `FeatureCalc`, `FeatureMatrix` just being a data storage object.

As for the features, it is also possible to add other color spaces to the one already defined, so it is possible to use the defined features on other color components. This is done by just adding a color component type definition. The function that does the transformation between one color space and the other is implemented in `FeatureCalc`.

The color components already defined are: Y, U and V, since we work with YUV images.

A `FeatureMatrix` can also be resized (keeping or not the current data), assigned to another `FeatureMatrix`, or created from another one (by its copy constructor). Furthermore a `FeatureMatrix` can be reinitialized: the feature type specifications are all set as uninitialized and the color component specifications reset to Y.

There are also I/O functions in `FeatureMatrix` that allow saving and reading data from disk. One can write the features data, as one file per feature type, in any image file format already defined in the `LTS`*Platform* , or in pure ASCII format (readable by MATLAB™). Otherwise one can also write all feature data, one vector after another, in one binary file without any feature specification.

A file format and the read and write functions for this one have also been defined. This file format saves the whole `FeatureMatrix`, features data and specifications, and has an identification stamp, “or magic number”, to identify the file type.

Finally, given a segmentation, one can write feature data in one file per region, so to isolate the features relative to different regions. All features in a region are written in binary form, one vector after another.

3.3 The `FeatureCalc` box

In addition to the data storage unit with I/O functions, we need another one that calculates the features. This is what the `FeatureCalc` class is intended to do. It inherits from `Box` since it is a processing unit.

This box accepts a `YUV420Image` object as its input and a `FeatureMatrix` as its output. This is the only configuration accepted, or *scheme* in the `LTS`*Platform* jargon. However one can easily add another scheme to accept another image types as input.

The `YUV420Image` is an image format for color images in the YUV color space with the chrominance components, U and V, downsampled by two in the horizontal and vertical directions, the luminance being stored at full resolution.

In the `LTS`*Platform* one can instantiate images of no matter what type of elements: they can be `floats`, `ints`, `unsigned chars`, etc.; but the only one actually used is `unsigned char` (integers from 0 to 255), which is the only one that `FeatureCalc` accepts.

The specifications of the features to calculate are given as parameters. Some parameters are mandatory and some are optional, having a default value.

The mandatory parameters are: the feature type, the horizontal window size and the vertical one. These are actually arrays of parameters, where each position in the arrays corresponds to one different feature.

The optional parameters are: the color component and a "new-image" flag. The color component parameter is an array specifying the color component (Y, U or V) on which the calculation of each feature should be carried out. If it is not given it defaults to the luminance (Y) for all features.

The "new-image" parameter is just a flag. If 0 it indicates that the `FeatureMatrix` given for the output already contains calculated features for the given image, and only those that differ in specification should be calculated. If not given, or if it is not 0, the box assumes that the image at the input is a new image and that all specified features should be calculated. This last parameter makes it possible to add features to the `FeatureMatrix` without recalculating the features that have already been calculated.

When the `FeatureCalc` is called (with the member function `process`) it reads all given parameters and starts calculating every specified feature. To actually calculate these features one function is defined for each feature that implements the algorithms given in Sec. 2.

Once a feature has been calculated the feature specification is written to the `FeatureMatrix` before calculating the other ones. This makes it possible to make a feature calculation dependent on another one, as it is the case of the local average with the sigma filter (*sigavg*) that depends on the local standard deviation (*std*). The calculation function tries to find in the `FeatureMatrix` the needed feature, if it is not found it adds it to the `FeatureMatrix` by first resizing it and then calling the function that calculates the needed feature. The automatically added features are deleted once that all specified features have been calculated.

If a new feature type is introduced all that has to be done is to define a function that implements the algorithm and add the type definition to `FeatureMatrix` as explained in the previous section.

In order to facilitate the implementation of filtering functions a new class has been defined that inherits from `MatrixIO`, called `MtxBlock`. A member function of this class returns a window of an image given the center coordinates and the size. It handles automatically the border problems (i.e when part of the window lies outside the image) by mirroring the inner pixels. Two variants of this last function have also been written: one that implements the marking of outliers for the sigma filter, and one that implements the mirroring of pixels based on the gradient information as explained in Sec. 2.3.

These functions enable the user to easily write new feature calculation functions without having to worry about the border problems, so one can write the code to try out a feature in less time. However there is a penalty: the execution speed is not as high as with a specific implementation for each feature; but this execution time difference have proven to be minimal (except with the function implementing the mirroring with the gradient information).

Finally, as it has been said before, it is possible to add new color spaces. For each new color component a transformation to get it from the image data has to be added and the rest is done automatically (a more detailed explanation can be found in the source code).

3.4 The `feature` program

With the two classes explained above, `FeatureMatrix` and `FeatureCalc`, it is possible to implement the calculation of features in a program and then to use the result. However to make the testing of the features practical it is necessary to write a program that uses those classes and that needs not to be recompiled each time one wants to change the specifications of the features to calculate.

This is why the `feature` program has been written. It is basically a program that calculates the features, as specified in a file, on the given image and that saves the results in several file formats as specified with the command line options.

This program has a simple user interface and an on-line help. It reads any image file format that the `LTS`*Platform* is able to read. For the image formats that don't have a size specification in the file itself, such as YUV or raw gray level, the size can be given in the command line.

The configuration file, where the specifications of the wanted features is given, is a text file. The specifications given are: the name of the feature, the horizontal size of the window, the vertical size of the window and the color component on which to calculate the feature. For example it can contain the following,

```
avg  5  5  y
var  5  5  y
```

The configuration file given above tells the program to calculate the local average and the local variance, both with a 5×5 window and on the luminance of the image, in that order. The possible names for the features currently are,

norm a copy of the image data, no processing is done.

avg the local average, with no special handling.

std the local standard deviation, with no special handling.

var the local variance, with no special handling.

hstd the horizontal standard deviation as explained in Sec. 2.2.

hvar the horizontal variance as explained in Sec. 2.2.

vvar the vertical standard deviation as explained in Sec. 2.2.

vvar the vertical variance as explained in Sec. 2.2.

hsob the result of applying the horizontal Sobel operator.

vsob the result of applying the vertical Sobel operator.

sob the magnitude of the gradient estimated with the Sobel operator.

svar the local variance calculated with mirroring using the gradient information as explained in Sec. 2.3.

sigavg the local average calculated with a sigma filter.

sigstd the local standard deviation calculated with a sigma filter.

sigvar the local variance calculated with a sigma filter.

sighstd the horizontal standard deviation calculated with a sigma filter.

sighvar the horizontal variance calculated with a sigma filter.

sigvstd the vertical standard deviation calculated with a sigma filter.

sigvvar the vertical variance calculated with a sigma filter.

sasigstd the local standard deviation calculated with a sigma filter and with the marking of outliers done on *sigavg*.

sigstdmed the local standard deviation calculated with a sigma filter and passed trough a 3×3 median filter.

sasigstdmed the local standard deviation calculated with a sigma filter and with the marking of outliers done on *sigavg*, passed trough a 3×3 median filter.

sigsigstd the local standard deviation calculated with a sigma filter and passed trough a 3×3 averaging sigma filter.

test a temporary name to test new features. Currently nothing.

The possible names for the color components currently are,

y the luminance or gray level (more precisely the Y component of the YUV space).

u the U chrominance component of the YUV color space.

v the V chrominance component of the YUV color space.

Other feature types or color components could be added in the future, as it was explained in the previous section. It is clear that some of the named features are not useful for the segmentation algorithm but they are needed by other features to be calculated (for example *sigavg* needs *std*).

The configuration can be stored in any file, the name being given in the command line to the `feature` program. For a detailed syntax of command line options and arguments refer to the source code or type '`feature.x -h`' at the command prompt.

3.5 Vector quantization programs

To test the robustness of the features we used vector quantization [2] as it is explained in Sec. 4.

For the vector quantization we used the public domain programs written by Eve Riskin at Stanford University: `stdvq` and `stdvqe`. The first one is the one that does the training and the second one is the coder.

The training is actually the search of a set of vectors that minimizes the mean square error to encode the given data. The resulting set of vectors, or codewords, is called a codebook and its size is fixed by the user.

The coder takes a set of vectors and replaces each one of them by the closest one that is found in the given codebook. Then we can say that each encoded vector belongs to a region, corresponding to a particular codeword.

The `stdvq` program has been taken without any modifications, the input to it are the vectors of features that the `feature` program outputs, in raw binary format.

The `stdvqe` program, the coder, has been slightly modified so that it saves the index of the codeword assigned to each vector in the input data. These can be saved as a gray level image in pgm format (with automatic scaling to enhance the contrast between indices) or as an image in raw format.

More detailed information can be found in the source code of these programs.

3.6 Utility programs

In addition to the programs described above (`feature`, `stdvq`, `stdvqe`) a library of other utility programs was written. These convert between different file formats, extract regions from an image, etc. Also, since the programs written work on one image at a time, UNIX shell-scripts to automate the process with sequences were written.

The utility programs that have been written using the `LTS`*Platform* are:

`mergereg.x` merges feature vectors from several files (saved in raw binary format) in one file. Each input file contains the features belonging to a region of the image. A file defining the regions (segmentation file) must be given.

`extractreg.x` extracts the specified regions from a given image in YUV420 file format over a background (white by default). A segmentation file, defining the regions, must be given.

`fmraw2img.x` converts a `FeatureMatrix` file saved in raw binary format to an image format as one file per feature. The image file formats supported are: pgm, raw gray level and pure ASCII (readable by MATLAB™).

`anytoyuv420.x` converts an image in any file format readable by the `LTS`*Platform* to the YUV420 format in 3 files.

`yuv420toppm.x` converts an image in the YUV420 file format (3 files) to the ppm "rawbits" file format.

Other utility programs were written in straight C, without using the `LTS`*Platform*. They are:

`doub2ascii` converts from binary `doubles` to ASCII to the standard output. Numbers can be grouped in lines with an optional argument. It is used to read the codebooks found by `stdvq`.

`float2ascii` like the above program but converts from binary `floats`.

`endianconv` converts data between little-endian and big-endian byte ordering and vice-versa. It is used to exchange binary data between machines with different byte ordering.

`scalefeat` scales a feature saved in a file in raw binary format by the given scale factor. It can be modified in the input file or copied into another file.

`logscalefeat` like scalefeat but it takes the logarithm of the feature before multiplying it (actually it does $f \cdot \log_{10}(1 + x)$, where f is the scaling factor and x the feature value).

`uchar2float` converts from binary `unsigned char` data to binary `float` data. Used to make the `stdvq` program take an image for the training.

The UNIX shell-scripts (csh or tcsh) written to automate some tasks are:

`mkfrdir` copies a YUV420 sequence to directories as one directory per frame, which is the directory structure used by the other scripts. It can also make symbolic links instead of copying the files.

`fvq` runs the `feature`, `stdvq` and `stdvqe` programs on a given image. It has options to scale or logscale a feature. The result is a segmentation based on the specified features.

`fvqnf` same as `fvq` but skips the `feature` program, it uses the feature data already there.

`fsvq` runs the `stdvq` and `stdvqe` programs on an existing feature data in a per region basis, where the regions are specified in a segmentation file.

`favq` runs the `feature`, `stdvq` and `stdvqe` programs on all the frames of a given sequence. The codebook is calculated for each frame and its used to encode that frame only. It has options to scale or logscale a feature. The result is a segmentation for each frame (it is like running `fvq` on each frame separately).

`ffvq` runs the `feature`, `stdvq` and `stdvqe` programs on all the frames of a sequence. The codebook is calculated for only one frame (specified in the command line) and then it is used to encode all the frames in the sequence. It has options to scale or logscale features. The result is a segmented sequence.

`ffvqmovie` displays the results of `ffvq`, the segmented sequence, as a movie on an X display.

`ffvqext` extracts, using the `extractreg.x` program, the specified regions from a sequence previously segmented with `ffvq`. The result is a sequence displaying the extracted regions on a background (white by default).

`ffvqextmov` displays the results of `ffvqext`, the extracted sequence, as a movie on an X display.

For more detailed information about all the options of the above programs and scripts refer to their online documentation or their source code.

4 Results of segmenting in the feature space

4.1 Segmenting with vector quantization

In order to test the robustness of the features explained in Sec. 2 we need to perform a segmentation in the feature space. That is what vector quantization [2] is used for, since no other available segmentation programs can work in the feature space (i.e in a vector scheme).

Vector quantization is normally used to quantize an image in an optimal way, with minimum distortion. The vectors used in that case are picture elements and the training consists in finding a set of vectors, of fixed size (not necessarily found in the image), that minimizes a given measure of distortion. The set of vectors found is called the *codebook*, and the vectors found in it are called *codewords*.

In our application the goal of the vector quantization is different. We applied the techniques developed for vector quantization in the multidimensional feature space. The main difference resides in the fact that the vectors used are not picture elements but the vectors in the feature space (i.e the vectors of the `FeatureMatrix`). As a consequence we do not obtain an encoded image but an encoded `FeatureMatrix`, and the codewords are vectors of features.

Since in the encoded `FeatureMatrix` there is only a small number of different feature vectors (the ones found in the codebook) we can make up an image where the pixel values are the indexes of these vectors. We obtain then an image representing the distribution of the codewords in the encoded `FeatureMatrix`. We can say then that the original image is dived in a small number of classes.

This classes not only apply to the original image but also to the `FeatureMatrix` associated to it. Each class in that one contains feature vectors that resemble each other because the codebook minimizes the encoding error. If we set the size of the codebook relatively small what we obtain is actually a segmentation, where each class mentioned above is a region of the segmentation.

As it is shown in the following sections this segmentation method works fairly well. However it is not a real segmentation algorithm because it doesn't take into account the physical distance in the image: a class can be divided in two parts that are very far apart. Moreover isolated pixels are not merged to neighboring classes, they remain isolated.

Even if the vector quantization is not a real segmentation algorithm it is very useful to test the robustness of the features because the flaws mentioned in the

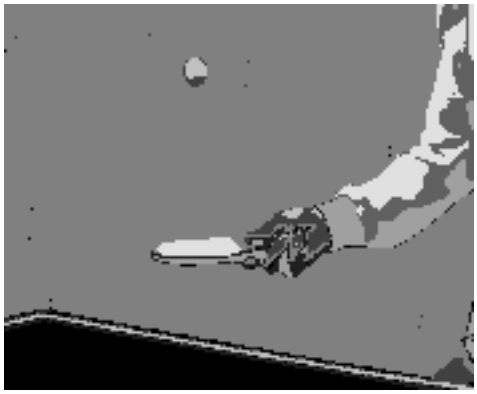

Figure 17: an example of a segmentation obtained by vector quantization with 8 regions (i.e. codebook size of 8).

previous paragraph are not relevant to judge that.

As an early example of what kind of segmentation can be obtained we show, in Fig. 17, one done on the image of Fig. 1. The vector quantization was run with a codebook of size 8, so there are 8 regions in the segmentation.

4.2 Still image segmentation

The goal of this study is to extract objects from sequences. However the features have been tested first on still images to determine the most suitable ones.

After all the tests we have determined that the perfect combination of features does not depend too much on the image type. The local average of the gray level, calculated with the sigma filter, is the most important one. If the image is in color then the use of the local averages calculated with the sigma filter of the chrominance components improves significantly the result.

As an example we can see the segmentations obtained in Figs. 19 and 20 of the image shown in Fig. 18. It is clear from these examples that using the sigma filter to calculate the average improves a lot the precision of the limits between the regions and lets appear fine details in them. Moreover if the sigma filter is not used there appears a border line between some regions that is non-existent in the original image. This is because of the indiscriminate averaging of pixels inside the window that transforms step edges in gradual ones.

The window size used for the calculation is not very critical but the 5×5 window seems to be the best choice, it gives better results than 3×3 but is slower to

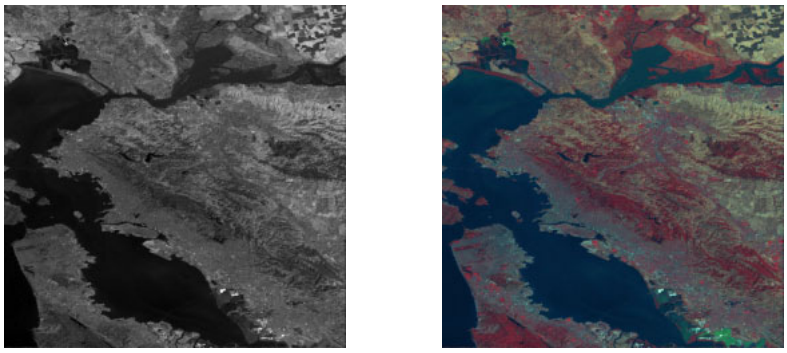

Figure 18: image used for the segmentation examples in gray (left) and color (right).

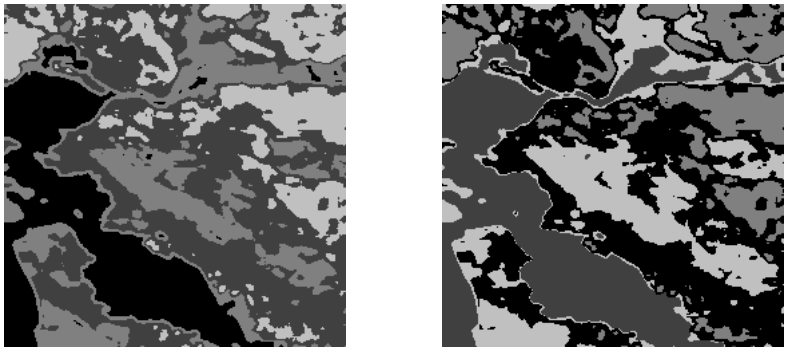

Figure 19: segmentation obtained using the local average, *avg*, of only the gray level (left) and of the three color components Y, U and V (right). The window size used was 5×5 and the number of regions was fixed to 4 (note that no sigma filter was used).

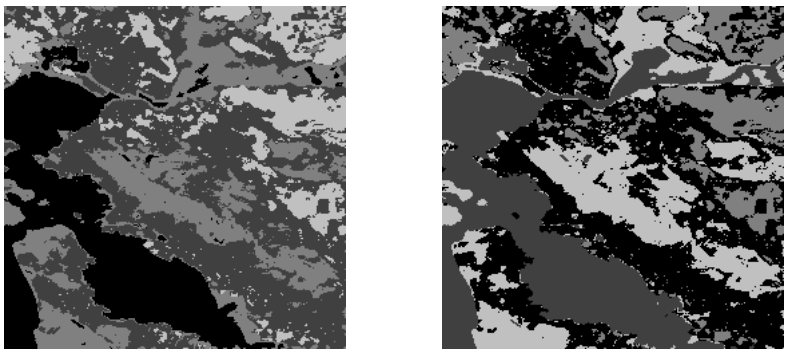

Figure 20: segmentation obtained using the local average with the sigma filter, *sigavg*, of only the gray level (left) and of the three color components Y, U and V (right). The window size used was 5×5 and the number of regions was fixed to 4.

calculate. Larger windows are not useful since the segmentation results do not improve, compared to the 5×5 window, and the calculation time is much longer.

One can also see from the examples given that the use of the chrominance components, U and V, gives significantly better results than the use of the luminance alone. In the different segmentations that are shown one can see that the result is almost perfect when the color information is taken into account and the sigma filter is used. Only some fine lines of the sea are erroneously segmented.

From the discussion above it should be clear that one should always use the local average calculated with the sigma filter on the gray level for the segmentation, and also the averages on the chrominance components if the image is in color. This not only applies to the image shown here but to a wide variety of them. Some other examples are given in the next section.

Concerning the texture information the choice of features is not as straightforward as with the local averages. In Figs. 21–24 examples of segmentations using the local standard deviation calculated in four different forms are given. The local standard deviation is preferred over the local variance because the edge problem mentioned in Sec. 2 is less important in the standard deviation even though it represents the different textures fairly well. In all the examples shown the standard deviation is scaled by 5 so it has a weight similar to those of the local averages in the vector quantization (the dynamic range of the standard deviation is much smaller than that of local averages).

Only the luminance is used for the standard deviation and variance, since the ones of the chrominance components is not very significant.

In the examples given it is clear that, when the chrominance components U and V are not taken into account, using the standard deviation significantly improves the quality of the segmentation. However the differences between the four methods shown are minimal. The best results are obtained when the local standard deviation is as calculated as *sasigstd*. The better performance is due to how the outliers are marked because, since this respects better the fine details. The use of the median filter in the example degrades the segmentation, nevertheless this filter can become useful in images where the edge problem explained in Secs. 2.2 and 2.5 is stronger than in the example given here.

When the chrominance components are taken into account the improvement over the segmentation without the standard deviation is not as dramatic as in the gray level case. The best performance is obtained when the standard deviation is calculated as *sasigstd*, as it is the case when only the gray level is taken into account, even though the differences with the standard deviation calculated with the simple

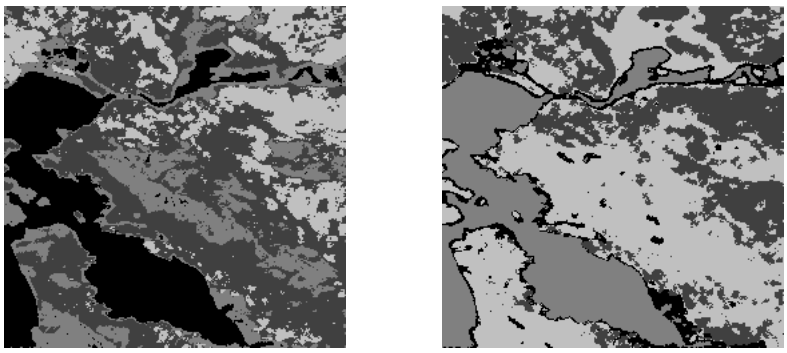

Figure 21: segmentation obtained using the local standard deviation *sigstd* of the luminance and the local average *sigavg* of only the gray level (left) and of all three color components (Y, U and V) (right). The window used was 5×5 and the number of regions was fixed to 4. The standard deviation has been scaled by 5.

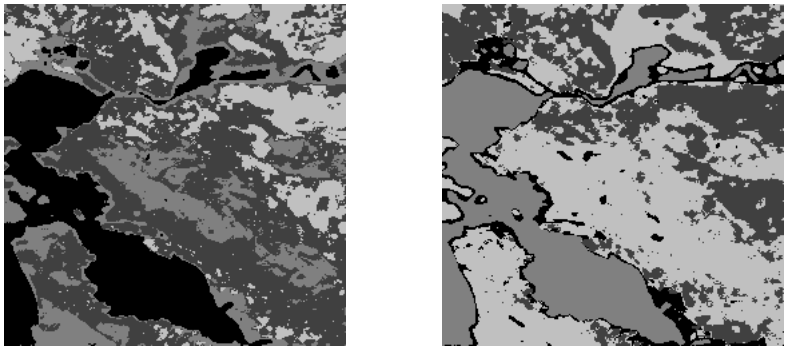

Figure 22: segmentation obtained using the local standard deviation *sigstdmed* of the luminance and the local average *sigavg* of only the gray level (left) and of all three color components (Y, U and V) (right). The window used was 5×5 and the number of regions was fixed to 4. The standard deviation has been scaled by 5.

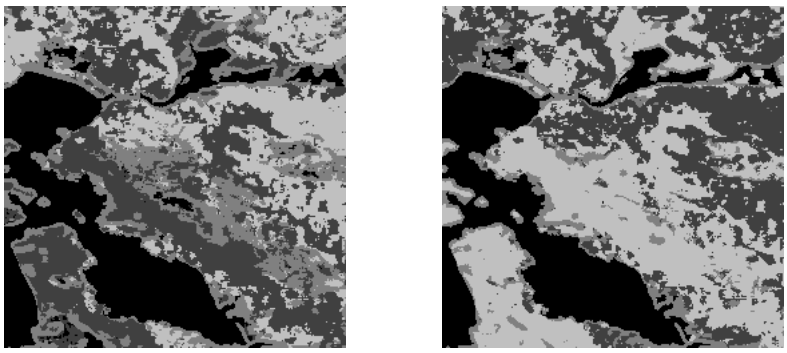

Figure 23: segmentation obtained using the local standard deviation *sasigstd* of the luminance and the local average *sigavg* of only the gray level (left) and of all three color components (Y, U and V) (right). The window used was 5×5 and the number of regions was fixed to 4. The standard deviation has been scaled by 5.

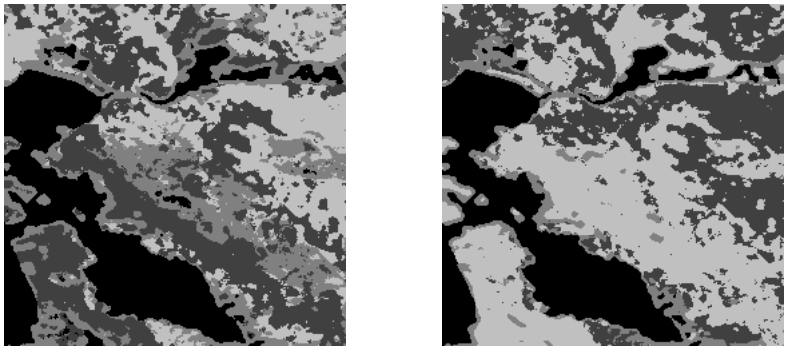

Figure 24: segmentation obtained using the local standard deviation *sasigstdmed* of the luminance and the local average *sigavg* of only the gray level (left) and of all three color components (Y, U and V) (right). The window used was 5×5 and the number of regions was fixed to 4. The standard deviation has been scaled by 5.

sigma filter (*sigstd*) are almost insignificant. As in the previous case the use of the median (*sigstdmed* and *sasigstdmed*) filter deteriorates the segmentation.

Another series of examples is shown in Figs. 26–31 for another image, which is shown in Fig. 25 in gray level and color. The remarks on the local averages made for the previous example remain valid: the segmentation is significantly improved when the sigma filter is used, and the use of the chrominance components are very important, they should be used whenever they are available.

One can see that even when the only feature taken into account is the average gray level calculated with the sigma filter, the arm, the ball and the racket are well segmented. However, the background and the table are oversegmented, specially the background, and the table white border does not come out very well.

Adding the standard deviation to the last case, in any of the four forms, does not significantly improve the segmentation, not as in the San Francisco bay image. The major improvement occurs in the table but it is not very significant. This is because there are not many textured regions, the only one being the background. Moreover the background has a sort of "macro" texture that it is not well reflected by the local standard deviation since the window size is not very big. However a larger window size cannot be taken because the edge problem becomes too important and it renders the estimation unusable.

In the color case one can see that the obtained segmentation is far better than in the gray level case. All regions are well segmented, with the exception of the hand that is an object fairly complicated, with too many different regions. The background is separated in two but this is not very important since we aim to a supervised segmentation, the most important thing is that it should be well separated from the other regions, as it is, and not oversegmented.

When the standard deviation is included in the color case the background comes out as one region and not two or three as in the other cases. Even if the standard deviation is not a good "measure" of the background texture it is enough to make it one region when the color information is present. However this is also enough to divide the ball in two regions.

In both cases, color and gray level, the different methods to calculate the standard deviation give quite different results. The best one is obtained with the use of only a sigma filter.

As a general rule one should not use the standard deviation or variance when the image does not contain any textured regions. In those cases using it degrades the segmentation quality because of the problem with the edges. Also when there

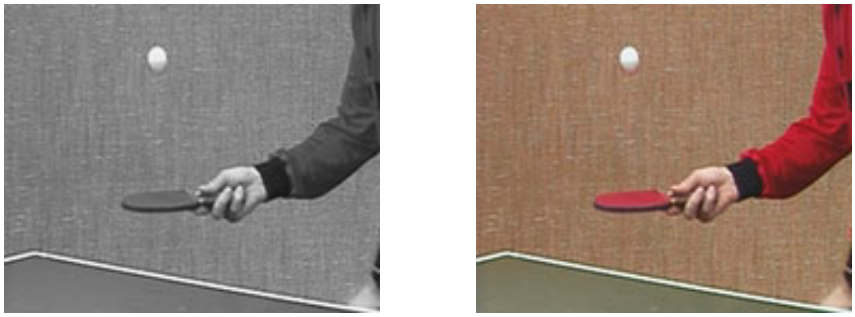

Figure 25: image used for the segmentation examples in gray (left) and color (right).

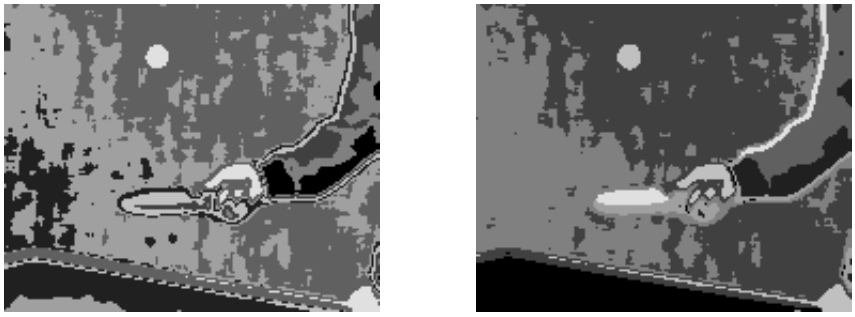

Figure 26: segmentation obtained using the local average *avg* of only the gray level (left) and of the three color components Y, U and V (right). The window size used was 5×5 and the number of regions was fixed to 8 (note that no sigma filter was used).

are very few textured regions using the standard deviation does not help much and sometimes it can even degrade the result, depending on the image type.

It should also be noted that if the standard deviation is to be included among the features used in the segmentation the choice of the best method to calculate it depends on the image too. When there are borders in the objects, like the white one in the table, marking the outliers on the local average calculated using a sigma filter degrades the segmentation, one should only use it when there are no such regions in the image. Moreover, the median filter should be used whenever the edge problem is important at the expense of shifting the borders a little bit.

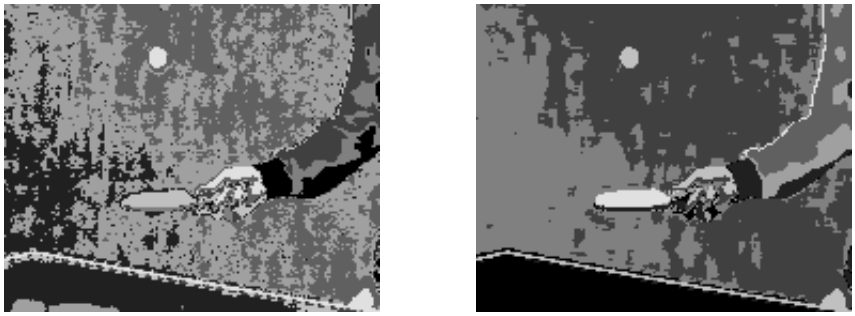

Figure 27: segmentation obtained using the local average *sigavg* of only the gray level (left) and of the three color components Y, U and V (right). The window size used was 5×5 and the number of regions was fixed to 8.

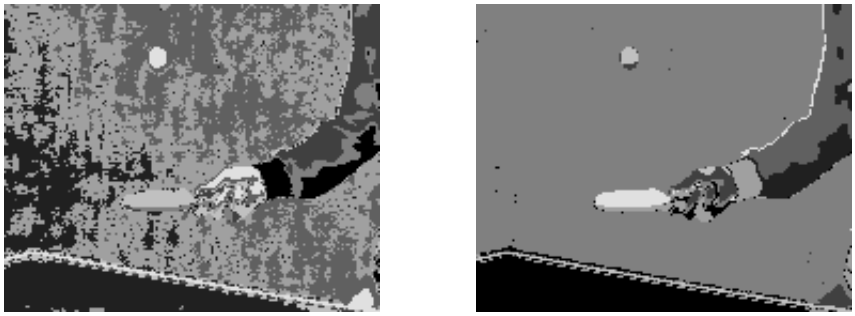

Figure 28: segmentation obtained using the local standard deviation *sigstd* of the luminance and the local average *sigavg* of only the gray level (left) and of all three color components (Y, U and V) (right). The window used was 5×5 and the number of regions was fixed to 8. The standard deviation has been scaled by 2.

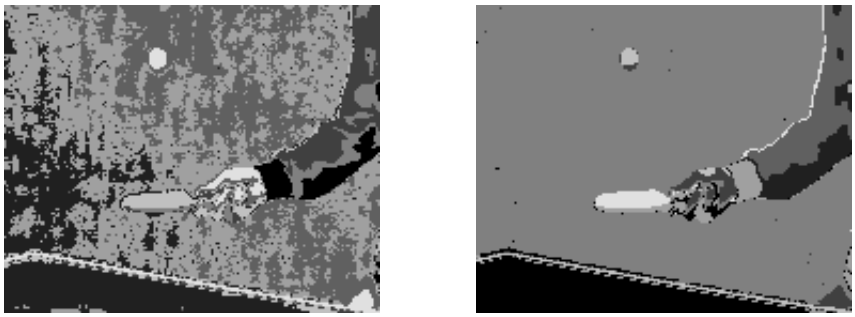

Figure 29: segmentation obtained using the local standard deviation *sigstdmed* of the luminance and the local average *sigavg* of only the gray level (left) and of all three color components (Y, U and V) (right). The window used was 5×5 and the number of regions was fixed to 8. The standard deviation has been scaled by 2.

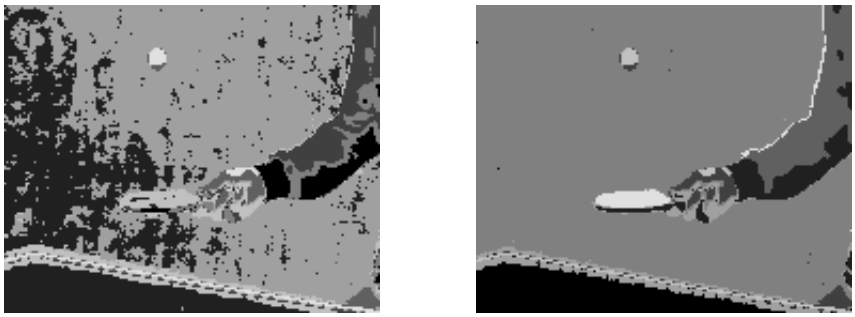

Figure 30: segmentation obtained using the local standard deviation *sasigstd* of the luminance and the local average *sigavg* of only the gray level (left) and of all three color components (Y, U and V) (right). The window used was 5×5 and the number of regions was fixed to 8. The standard deviation has been scaled by 2.

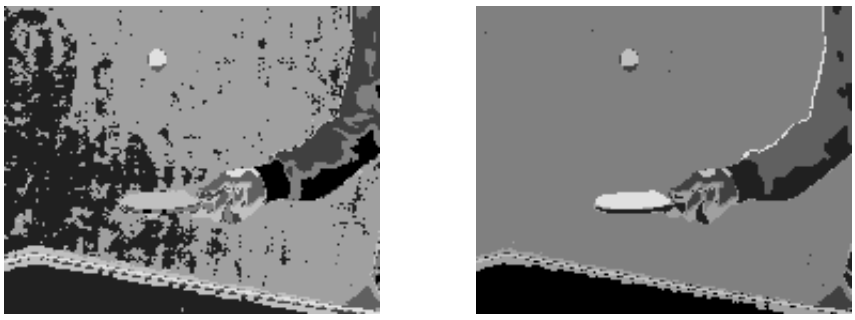

Figure 31: segmentation obtained using the local standard deviation *sasigstdmed* of the luminance and the local average *sigavg* of only the gray level (left) and of all three color components (Y, U and V) (right). The window used was 5×5 and the number of regions was fixed to 8. The standard deviation has been scaled by 2.

4.3 Sequence segmentation and object tracking

In order to segment a sequence it is not enough to perform a good segmentation in each frame, it is necessary that a correspondence between the regions in one frame and the regions in another one can be established.

In the previous section we have shown features that perform well for still image segmentation. But how do these features behave in time? As it is shown later in this section the features that we have found do not vary much from frame to frame. This makes it possible to establish the correspondence of regions across the sequence.

The first series of tests done for the sequences consists in comparing the centroids of the regions, or codewords, obtained from doing a training in each frame. The corresponding segmentations are shown for a few of the frames in Fig. 32. Also the graphics representing the evolution of the centroids across frames for each region are shown in Fig. 33 (the regions are numbered from black to white: 1 is black 8 is white).

The features used for those segmentations are the local averages of the luminance, Y, and chrominances, U and V, calculated as *sigavg*, and the local standard deviation of the luminance, calculated as *sigstd*; all of them on a 5×5 window. The local standard deviation is scaled by 2. This is the best configuration for the table tennis sequence as shown in the previous section.

One can see that the regions are quite stable, they do not vary much from one frame to another. The only major problem is the background that sometimes is split in two and then merged again. From the graphics showing the evolution of the centroids for each region one can see that they remain stable most of the time. The only large variations occur in region 3 when it changes from a part of the ball to a part of the background, and in region 6 when it changes from including the wrist and not including it. Concerning region 8 the centroids vary because the region containing the racket and the upper part of the arm changes in size.

From the above one can see that the features are quite robust, the centroids suffer little disturbance despite the changing regions. However the results above are not stable enough to easily segment a sequence, the splitting and merging of regions makes it difficult to use it.

As a second series of tests we segmented again the whole sequence but using the codebook obtained in the first frame to run the vector quantization on all other frames, instead of doing the training for each frame as in the examples above.

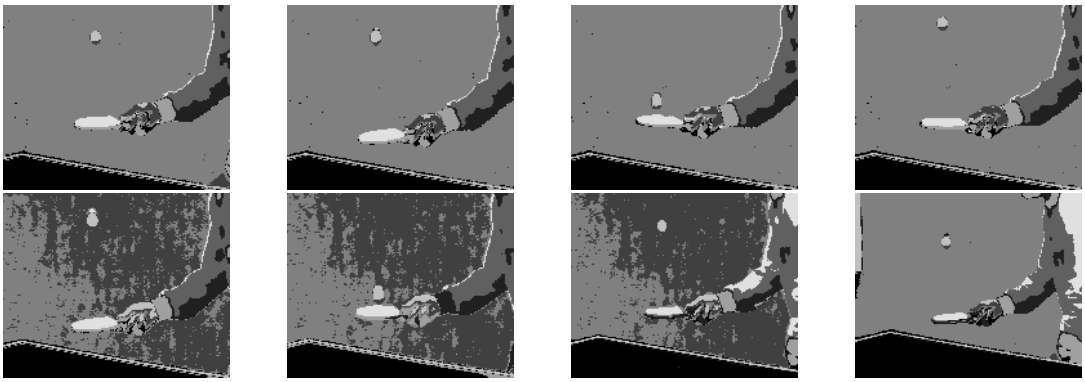

Figure 32: the segmentations obtained in the frames 1, 6, 11, 16, 21, 26, 31 and 36 of the sequence. The training for the vector quantization is done in each frame. The number of regions is fixed to 8.

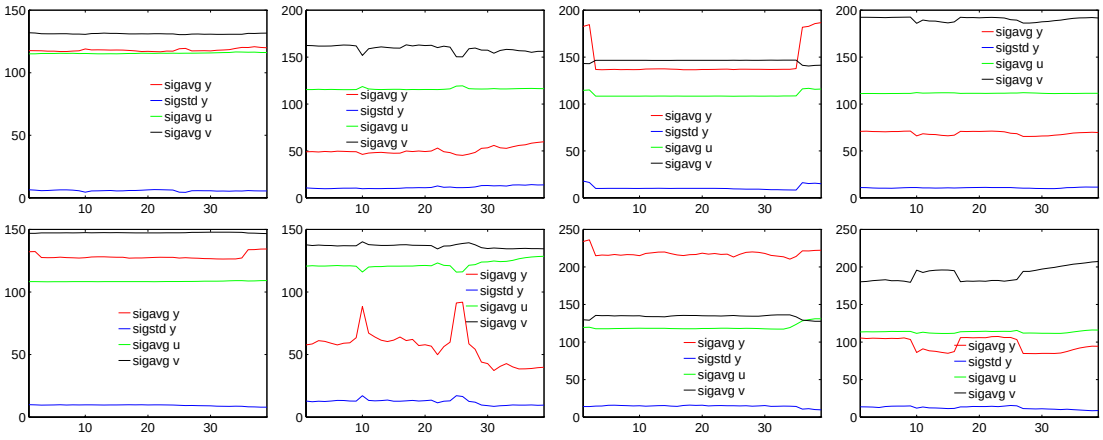

Figure 33: the evolution of the centroids of region 1–8 (from left to right and top to bottom) across the 39 frames tested.

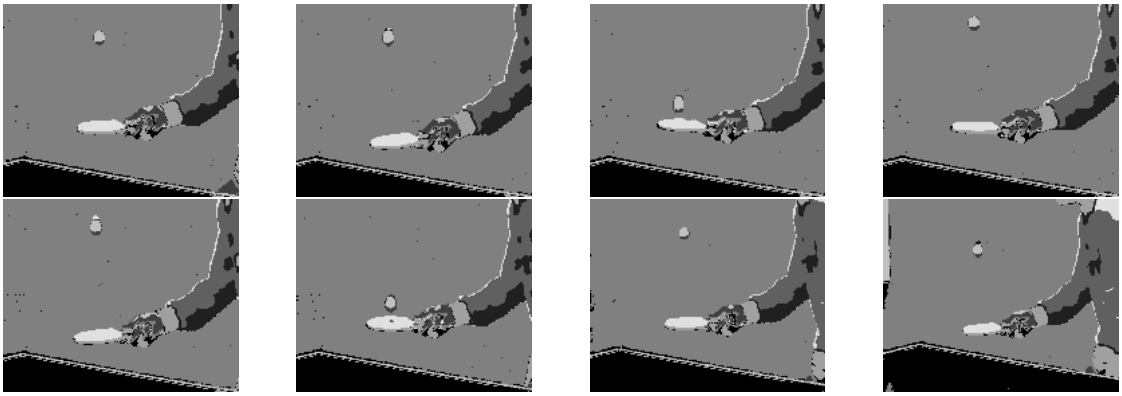

Figure 34: the segmentation obtained for the frames 1, 6, 11, 16, 21, 26, 31 and 36 (from left to right and top to bottom), using the codebook of frame 1.

The results are very encouraging. An example of what is obtained is shown for the same sequence in Fig. 34. The features used are the same as in the previous example.

One can see that the segmented regions do not change across the frames, they remain the same. Nevertheless the segmentation perfectly follows the movements of the different regions. This is really what is wanted: being able to segment a frame and then follow the regions in the other frames of the sequence.

It should be clear that this strategy of using the codebook obtained with the first frame without doing any kind of update as the frames change cannot be pushed too far. If the image changes too much the codebook would not be adapted and then the algorithm could fail. A method to do an update of the codebook has to be found to be able to do this kind of segmentation in longer sequences. However that is not what was intended to do here, we concentrated more in founding the right features and testing their robustness than finding an smart segmentation algorithm adapted to our purposes.

As the last example of this sequence we show the result of extracting, on a white background, the arm, ball, hand and racket using the segmentation shown above. Some of the frames of the result are shown in Figs. 35.

As the very last example we show the same results on another sequence (Figs. 36 and 37). The procedure used is the same as for the table tennis sequence but the features taken into account are only the local averages of the luminance, Y, and the chrominances, U and V, calculated as *sigavg*. The local standard deviation is not used since this sequence does not have any textured regions.

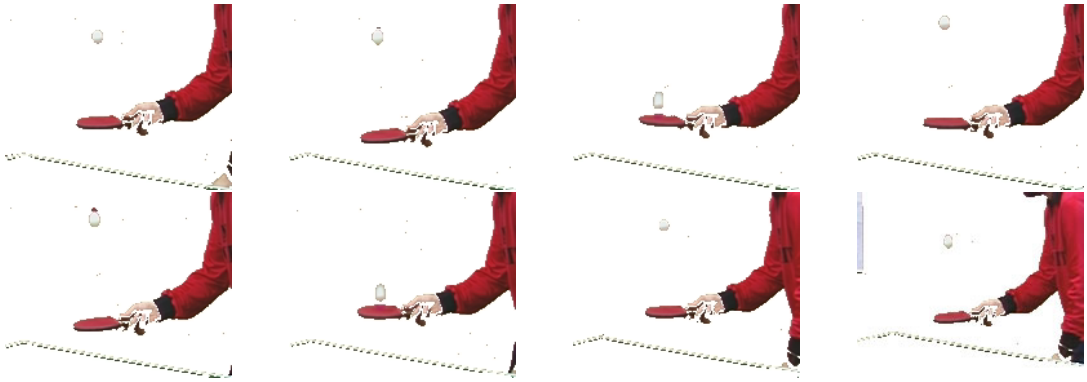

Figure 35: example of the extraction of some regions in the frames 1, 6, 11, 16, 21, 26 and 31 (from left to right and top to bottom). The segmentation used is the one based on the codebook of only the first frame.

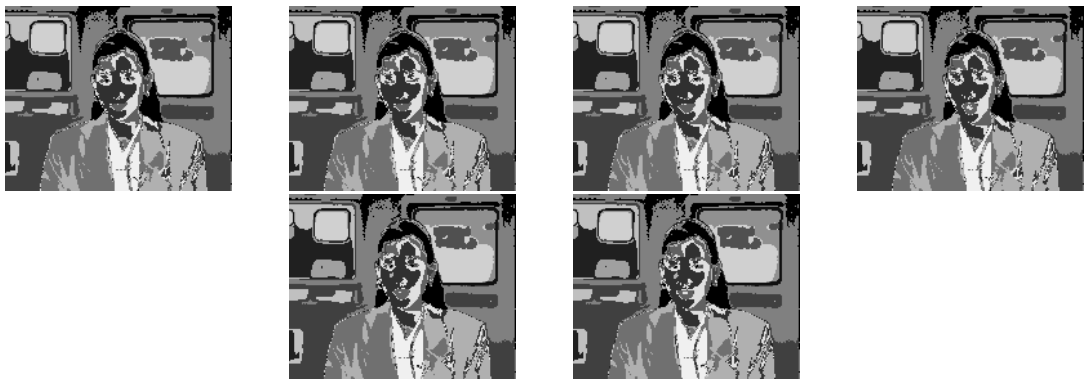

Figure 36: the segmentation obtained for the frames 0, 5, 10, 15, 20 and 25 (from left to right and top to bottom), using the codebook of frame 0.

The number of regions used is 16, what gives an oversegmentation, since 8 regions is not enough for the image. Unfortunately the number of regions with the vector quantization algorithm can only be fixed to a power of two (the reason for this is that it uses binary splitting).

One can see that the algorithm does not work as well as with the table tennis sequence. However the result should be greatly improved by using a smart, or real, segmentation algorithm that works in the feature space.

Figure 37: another example of the extraction of some regions in the frames 0, 5, 10, 15, 20 and 25 (from left to right and top to bottom). The segmentation used is the one based on the codebook of only the first frame.

5 Future improvements

5.1 Motion fields

The main application of the feature space as described here is the segmentation of sequences. Since the objects normally do not suffer from deformation each point in them has the same movement. One can then use the motion information to improve a segmentation, or even base the algorithm entirely on that kind of information. This is used in a wide variety of segmentation algorithms.

Including the dense motion fields in the feature space should significantly improve the segmentation (by dense we understand that is calculated for every pixel). Currently a program to calculate this motion fields is being developed at the lab and will be integrated to the ones use in this project in the near future.

However the integration of this information is not as straightforward as it seems. Just including the motion fields as one more feature in the vector quantization would not work for a sequence because the motion changes in time, even though it is uniform for an object (the motion it is not a constant in time of an object like the color). As a consequence using the codebook of the first frame to do vector quantization on the others would give erroneous results.

One way to integrate the motion fields in the feature space could be to do a separate segmentation for two adjacent frames, using the motion fields in each one, and then try to establish a correspondence of the regions based on the centroids of the constant features only (like local averages) and forgetting the ones for the motion fields.

5.2 New features

The features that one can include in the feature space are obviously not limited to the ones used here.

As other features one can imagine calculating the variance, or standard deviation, in some ways that reflect better the "macro" textures. One idea is to calculate it over a checker board or to calculate the horizontal variance taking into account only 1 every 2 columns, or analogously the vertical one taking into account 1 every 2 lines.

Another features that could be integrated are of a totally different nature. These are features used in other systems to determine the best coding algorithm for each

region with its optimum parameters [8]. If included in the feature space one could obtain a segmentation with the coding parameters all at once.

6 Conclusion

As it has been shown in the previous sections trough several examples the feature space approach constitutes a very interesting and promising method to do a segmentation.

The segmentations obtained show that the system is capable of segmenting and, most important of all, tracking the regions, even tough the segmentation algorithm is not very smart and that no motion information was used. No tradeoff is done between moving and non-moving objects as is the case in other algorithms.

The power of this approach resides on the fact that the features are considered altogether (the vectors are treated as one entity on its own) and not in an iterative or competitive fashion.

Another advantage of this method that has not been mentioned yet is that there are no parameters that one has to fix specifically for an image. The only adjustments that one has to do is to decide the features to include in the vectors and an eventual scaling of some of these features. As we have seen is fairly easy to determine those ones from just knowing the type of regions in the image.

Also a non-negligible part of this project are the programs and software library that were developed. This represents more than 6000 lines of code that will be integrated into the `LTS`*Platform* for the use of other people at the lab.

In my opinion the feature space approach, with the set of features studied here, constitutes a robust mean of doing a segmentation where the user does not have to try many different configurations before finding the most suitable one. It suffices from following some common sense rules mentioned in the previous sections.

Diego Santa Cruz

Lausanne, February 21, 1997

References

[1] Jong-Sen Lee, "Digital Image Smoothing and the Sigma Filter", *Computer Vision, Graphics, and Image Processing*, 24, 1983, pp. 255–269.

[2] R.M. Gray, "Vector quantization", *IEEE ASSP Magazine*, 1, No. 2, pp. 4–29, April 1984.

[3] *LTS Platform Presentation*, http://ltswww.epfl.ch/~fleury/Platform/Platform.html, Signal Processing Laboratory, EPFL, 1996.

[4] M. Kunt, *Traitement Numérique des Images*, Presses Polytechniques et Universitaires Romandes, Lausanne 1993.

[5] J. S. Lim, *Two-dimensional signal and image processing*, Prentice-Hall, 1990.

[6] ISO SC 29/WG11/N998, *MPEG-4 Proposal Package Description*, July 1995.

[7] MPEG4 Requirements Ad-Hoc Group *MPEG-4 requirements - version 1.1* ISO SC29/WG11/N1395, October 1996.

[8] P. Fleury and O. Egger, "Neural Network Based Image Coding Quality Prediction", to appear in *Proceedings of ICASSP* 1997, Munich, April 1997.

[9] Alan Ruegg, *Probabilités et Statistique*, Presses Polytechniques Romandes, Lausanne 1985.

[10] Stanley B. Lippman, *C++ Primer, 2nd edition*, Addison-Wesley, 1993

[11] Richard Petersen, *Introductory C, pointers, functions and files*, Academic Press, San Diego 1992.

[12] Helmut Kopka and Patrick W. Daly, *A Guide to LaTeX 2ε second edition*, Addison-Wesley, England 1995.