# Computer Vision using MatLAB and the Toolbox of Image Processing

## Technical Report B-05-09

Erik Cuevas[1,2], Daniel Zaldivar[1,2], and Raul Rojas[1]

[1]Freie Universität Berlin, Institut für Informatik
Takusstr. 9, D-14195 Berlin, Germany
[2]Universidad de Guadalajara, CUCEI
Av. Revolucion No. 1500, C.P 44430, Guadalajara, Jal, Mexico.
{cuevas, zaldivar, rojas}@inf.fu-berlin.de
November 12, 2003.

## Abstract

During the implementation of computer vision algorithms the manipulation of pointers, memory administration and some other resources are expensive in time even for friendly programming language. All these problems can be resolved if the implementation test is carried out in MatLAB using its toolbox of image processing with it the time of implementation becomes the minimum with the trust of using algorithms scientifically proven and robust. In this work we show the form in which can be used matlab and its toolboxes to solve common problems of computer vision efficiently.

## Introduction

During the implementation of computer vision algorithms the manipulation of pointers, memory administration and some other resources are expensive in time even for friendly programming language. To implement it in C++ language (due to their characteristics of high and low level it is appropriate for the implementation of computer vision algorithms) it would suppose the lost of valuable time that which is very important in those cases in which we do not have the security of what we want to implement finally will work. Also, to use C++ for the test period demands a correction time due normal errors in the implementation process of the algorithm, that is to say programmatic errors made to the moment to multiply two matrix or any other resemblance. All these problems can be resolved if the implementation test is carried out in MatLAB using its toolbox of image processing with it the time of implementation becomes the minimum with the trust of using algorithms scientifically proven and robust.

The toolbox of image processing implements a group of well-known algorithms to work with binary images, geometric transformation, morphology and color manipulation that together with the functions already integrated with matlab allows to implement analysis and transformations of images in the domain of the frequency (Fourier and Wavlets transform).

This work is divided in 3 parts, the first one treats the basic concepts of the images how they are represented in matlab as well as an introduction to the basic operations of handling of files. The second part covers the common and representative image processing functions in the area of computer vision; explaining the use of these

functions through examples. Finally the third part explains the use of the tool **vfm** used to capture images of devices installed in the computer such as video cards and USB Webcams.

**Basic concepts of the images and manipulation functions**

In matlab an **grayscale** image is represented by a two-dimensional matrix of *m* x *n* elements where *n* represents the number of pixels of wide and *m* the number of pixels of long. The element $v_{11}$ corresponds to the top left element. Where each element of the image matrix can have the value from 0 (black) to 255 (white). The figure 1 shows in detail these concepts.

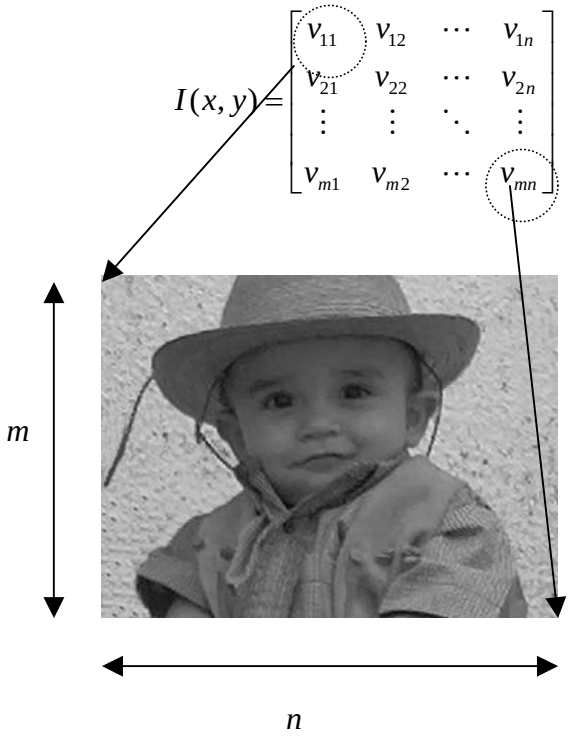



**Figure 1**. Grayscale image representation in MatLAB.

On the other hand a **RGB** color image (several types exist however it is the broadly used for computer vision, besides representing for matlab the default option) it is represented by a three-dimensional matrix *m* x *n* x *p*, where *m* and *n* have the same significance for the case of the grayscale images while *p* represents the plane that can be 1 for the red, 2 for the green and 3 for the blue. The figure 2 shows details of these concepts.

*Reading and writing images from a file*

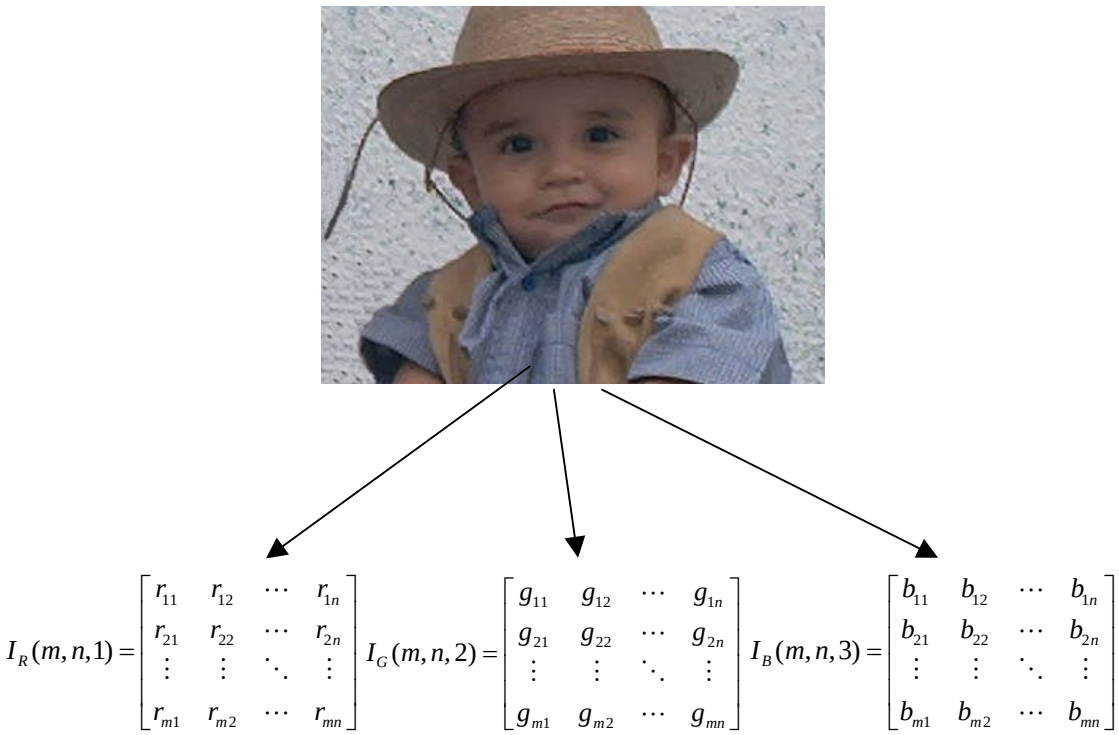
To read images in matlab contained in a file the function `imread` is used whose syntax is

```
imread( ‘name of the file’)
```

Where, ‘name of the file’ is a chain of characters containing the complete name of the image with their respective extension, the formats of images that matlab supports are shown in the table 1.

| Format | Extension |
|---|---|
| TIFF | .tiff |
| JPEG | .jpg |
| GIF | .gif |
| BMP | .bmp |
| PNG | .png |
| XWD | .xwd |

**Table 1.** Formats and extensions supported by MatLAB.



$$I_R(m,n,1)=\begin{bmatrix} r_{11} & r_{12} & \cdots & r_{1n} \\ r_{21} & r_{22} & \cdots & r_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ r_{m1} & r_{m2} & \cdots & r_{mn} \end{bmatrix} \quad I_G(m,n,2)=\begin{bmatrix} g_{11} & g_{12} & \cdots & g_{1n} \\ g_{21} & g_{22} & \cdots & g_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ g_{m1} & g_{m2} & \cdots & g_{mn} \end{bmatrix} \quad I_B(m,n,3)=\begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1n} \\ b_{21} & b_{22} & \cdots & b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{m1} & b_{m2} & \cdots & b_{mn} \end{bmatrix}$$

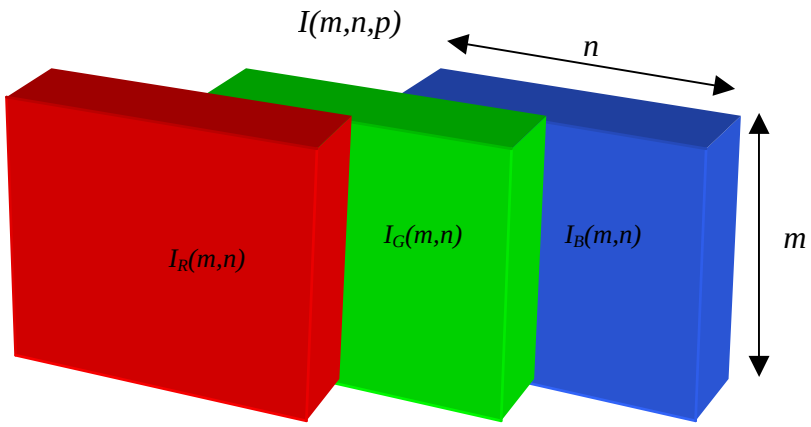



**Figure 2**. Representation of an RGB image color in MatLAB.

To introduce an image of a file with some of the formats specified in the table 1 it is only necessary to use the function imread and to assign its result to the variable that will represent to the image with the structures shown in the figure 1 or 2. For example that if it is wanted to introduce the image contained in the file data.jpg to a variable for its later processing in matlab we have to write in the command line window

```
>>image=imread( 'data.jpg' );
```

with this command the image contained in the file data.jpg was stored in the variable image.

Once the image is contained in a matlab variable then is possible to use the functions to process the image. For example a function that allows to find the size of the image is size(variable)

```
>>[m, n]=size(image);
```

where *m* and *n* will contain the values of the dimensions of the image.

To record the content of an image in a file the function imwrite(variable, ' name of the file' )is used, where *variable* represents the variable that contains the image and *name of the file*, the name of the file with its respective extension according to the table 1. Supposing that the variable image2 contains the image to be record in the file data2.jpg in this case we have to write

```
>>imwrite(image2, 'data2.jpg' ) ;
```

It is always necessary when we carry out some processing with the image to plot the obtained result, the function imshow(variable) allows to plot the image in a window in the matlab environment. If the variable to plot is face then we have to write in the command line

```
>>imshow(face);
```

we would obtain the image of the figure 3.

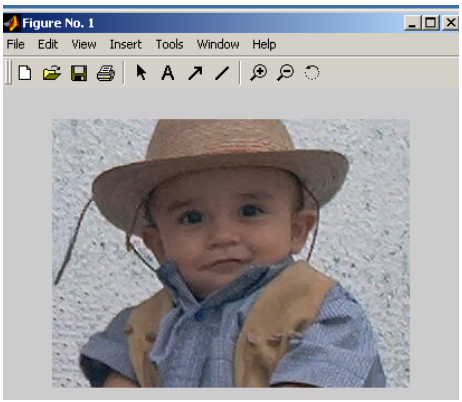



**Figure 3**. Example of the use of the function `imshow`.

*Access to pixel and planes in the images.*

The access to a pixel of an image is one of the most common operations in computer vision and this is totally automatic in matlab; only it is enough to index the interested pixel in the structure of the image. Let us consider that we have an image image1 in grayscale and we want to obtain its intensity value in the píxel specified by *m*=100 and *n*=100; we would have to write

```
>> image1(100,100)
ans =
   84
```

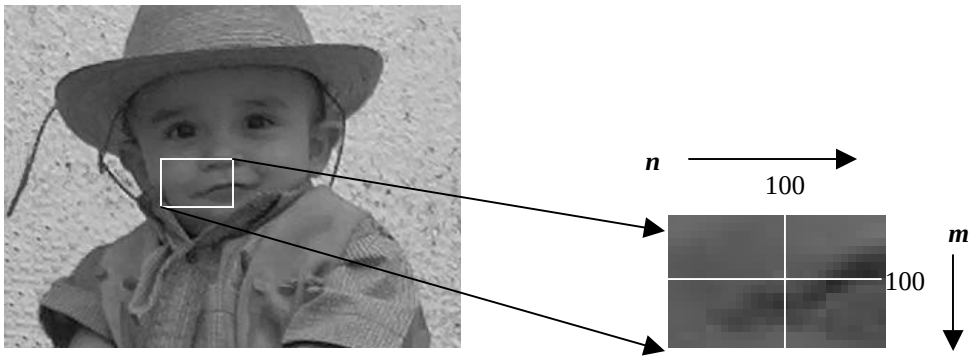



**Figure 4**. Obtaining the value pixel an image2.

Thus, if we want to change the pixel value to black, that is to assign it the value of 0 what we have to writte in command line is

```
>>image1(100,100)=0 ;
```

In the case of grayscale images we have to process only a plane (constituted by the matrix *m* x *n* that contains the values of intensity for each index). However the color images are composed for more of a plane. In the case of images RGB (just as it was explained above) these have 3 planes one for each color. Let us consider the image RGB contained in the variable image2 (shown in the figure 3), and we want to obtain each one of its planes. Then we have to write

```
>>planeR=image2( :, :,1) ;
>>planeG=image2( :, :,2) ;
>>planeB=image2( :, :,3) ;
```

The resulting planes of the previous commands are shown in the figure 5.

(a) (b) (c)

**Figure 5**. Planes of the image a) red, b) green and c) blue.

If we want to manipulate a pixel from an RGB color image then we have to assign a value for each plane. Let us suppose that we have the image RGB contained in the variable image2 and we want to obtain the value of the pixel *m*=100 and *n*=100 for each plane R, G and B. we have to write

```
>>valueR=image2(100,100,1) ;
>>valueG=image2(100,100,2) ;
>>valueB=image2(100,100,3) ;
```

That will give three values as a result. The same case as with the grayscale example we can change this pixel value to some other color, for the appropriate determination of each of their respective planes; for example

```
>>image2(100,100,1)=255;
>>image2(100,100,2)=255;
>>image2(100,100,3)=255;
```

In occasions it is preferable to know the color or the intensity (the pixel value) in an active way , that is to have the possibility to select a pixel in a homogeneous region and to obtain the value of it. This possibility is offered by the function `impixel`, which actively gives the value (one or three) of the selected pixel that appears in the window ploted by the function `imshow`. The format of this function is

```
value=impixel;
```

Where **value** represents a escalar (in case that the image is grayscale) or a vector of 1 x 3 (with the values corresponding to each one of the RGB plane).

Before to use this function is necessary to call the function used to plot the image (with the function `imshow`). Once ploted it is called the function and when the mouse cursor is on the surface of the image changed to +. When the left button of the mouse is pressed the pixel ist selected, we can select again the pixel in case that we have made a mistake positioning the mouse, since the function will be activated until the enter key is pressed. The figure 6 shows an image with the operation here described.

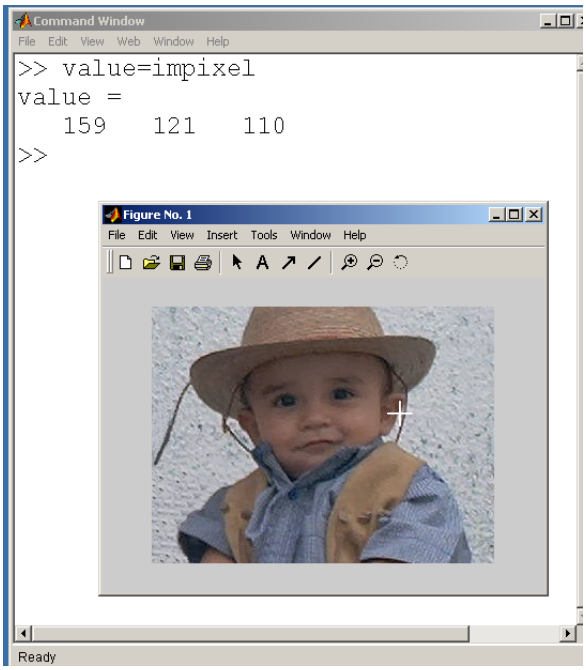



**Figure 6**. Use of the function `impixel`.

An important operation in computer vision is determining a image profile; that is to convert a segment from the image to a one-dimensional signal to analyze their changes, this is of special meaning in stereo vision where ist analyzed, for the algorithms epipolar segments of each camera. Matlab has the function `improfile` that allows to trace the segment interactively with the mouse, deploying the profile of the image in a different graph. This function needs the image on which is wanted to obtain the profile to be ploted previously with the function `imshow`. It should consider that if the image is grayscale the profile will show only a signal corresponding to the fluctuations of the image intensities, however if the profile is of RGB color, this will show one signal segment for each plane. For the use of this function it is necessary to write on command line

```
>>improfile
```

`improfile` is a iteractiv function so, as soon as the mouse is on the surface of the image the pointer changed the symbol to a +, of this way we can configure the wanted profile by the establishment of a line in the image. The figure 7 shows an image of the described operation.

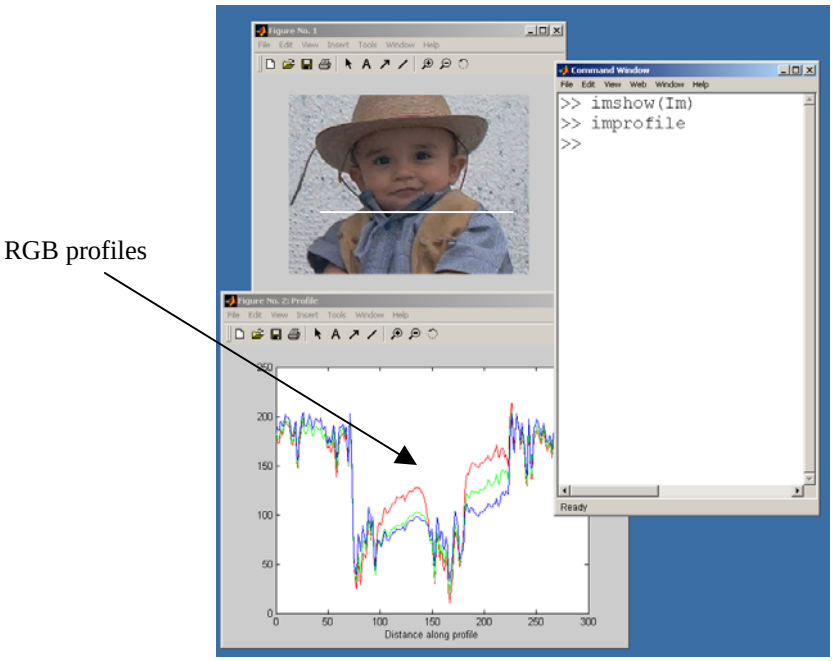


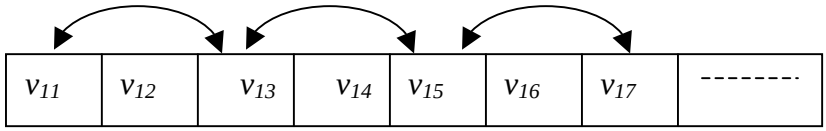

**Figure 7**. Utilización de la función `improfile`.

*Sub-sampling of images*

In occasions it is necessary to make calculations that require to process the image completely, in these cases make it on the original resolution will be expensive. An alternative is the sub-sampling of the image. Sub-sampling means to generate an image from periodic samples of the original image in such a way that this will be the representation of the original image but with a small size. If it is considered the image *I(m,n)* defined as

$$I(m,n)=\begin{bmatrix} v_{11} & v_{12} & v_{13} & \cdots & v_{1n} \\ v_{21} & v_{22} & v_{23} & \cdots & v_{2n} \\ v_{31} & v_{32} & v_{33} & \cdots & v_{3n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ v_{m1} & v_{m2} & v_{m3} & \cdots & v_{mn} \end{bmatrix} \quad (1)$$

and it is wanted sub-sampling the image in a half of the original the new image would be composed by the elements “taking one” element and “other not” of the original image as it is shown below




$$I_{S_2}(m,n)=\begin{bmatrix} v_{11} & v_{13} & v_{15} & \cdots & v_{1(n-2)} \\ v_{31} & v_{33} & v_{35} & \cdots & v_{3(n-2)} \\ v_{51} & v_{53} & v_{55} & \cdots & v_{5(n-2)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ v_{(m-2)1} & v_{(m-2)3} & v_{(m-2)5} & \cdots & v_{(m-2)(n-2)} \end{bmatrix} \quad (2)$$

Considering a RGB image contained on the variable image2 and we want to acoplish a Sub-sampling in half, we have to write the following code on command line

```
>>imageSub2=image2(1:2:end,1:2:end,1:1:end);
```

*Data type of the image elements*

The image elements in matlab have the format integer `uint8`, that is a data type that can vary from 0 to 255, without being able to support float data type and values outside of that range. This is a obstacle in those cases where algorithms to be implemented work

with float data type. In these cases it is necessary to transform the data type image from uint8 to double. It is important to observe that if the function imshow is used to plot the images; this function does not have the capacity to plot images of double data type, for that once accomplished the operations of floating point are necessary to convert the data type to uint8. Let us suppose that we have a grayscale image represented in the variable imagegray and we want to reduce their intensities in half, then we have to write

```
>>imagegrayD=double(imagegray) ;
>>imagegrayD=imagegrayD*0.5 ;
>>imagegray=uint8(imagegrayD) ;
>>imshow(imagegray) ;
```

**Image processing functions**

The functions that the image processing toolbox implements are very diverse, without to count the multiple offer of functions generated by other users and available in the Internet, however in this chapter only some functions considered as importat in computer vision will be treated.

*Spatial filtering*

Spatial filtering is one of the common operations in computer vision either to carry out effects of noise elimination or border detection. In both cases the pixel determination of the new image depends of the original image pixel and their neighbors. For it is necessary to configure a matrix (mask or window) that considers which neighbors and in what forms they will influence in the determination of the new pixel. Let us consider an image $I_S(m,n)$

$$I_S(m,n)=\begin{bmatrix} vs_{11} & vs_{12} & vs_{13} & \cdots & vs_{1n} \\ vs_{21} & vs_{22} & vs_{23} & \cdots & vs_{2n} \\ vs_{31} & vs_{32} & vs_{33} & \cdots & vs_{3n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ vs_{m1} & vs_{m2} & vs_{m3} & \cdots & vs_{mn} \end{bmatrix}$$

that will be the image to be filter and $I_T(m,n)$

$$I_T(m,n) = \begin{bmatrix} vt_{11} & vt_{12} & vt_{13} & \cdots & vt_{1n} \\ vt_{21} & vt_{22} & vt_{23} & \cdots & vt_{2n} \\ vt_{31} & vt_{32} & vt_{33} & \cdots & vt_{3n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ vt_{m1} & vt_{m2} & vt_{m3} & \cdots & vt_{mn} \end{bmatrix}$$

the resulting image; if we also consider *w(r,t)* the matrix that is used to carry out the filtering

$$w = \frac{1}{h}\begin{bmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \\ w_{31} & w_{32} & w_{33} \end{bmatrix}$$

where $r$=3 and $t$=3, and each element of $I_T(m,n)$ is calculated as

$$vt_{ij} = \frac{1}{h}[w_{22}vs_{ij} + w_{11}vs_{(i-1)(j-1)} + w_{12}vs_{(i-1)j} + w_{13}vs_{(i-1)(j+1)} \cdots$$

$$+ w_{21}vs_{i(j-1)} + w_{23}vs_{i(j+1)} + w_{31}vs_{(i+1)(j-1)} + w_{32}vs_{(i+1)j} + w_{33}vs_{(i+1)(j+1)}]$$

The figure 8 shows these details.

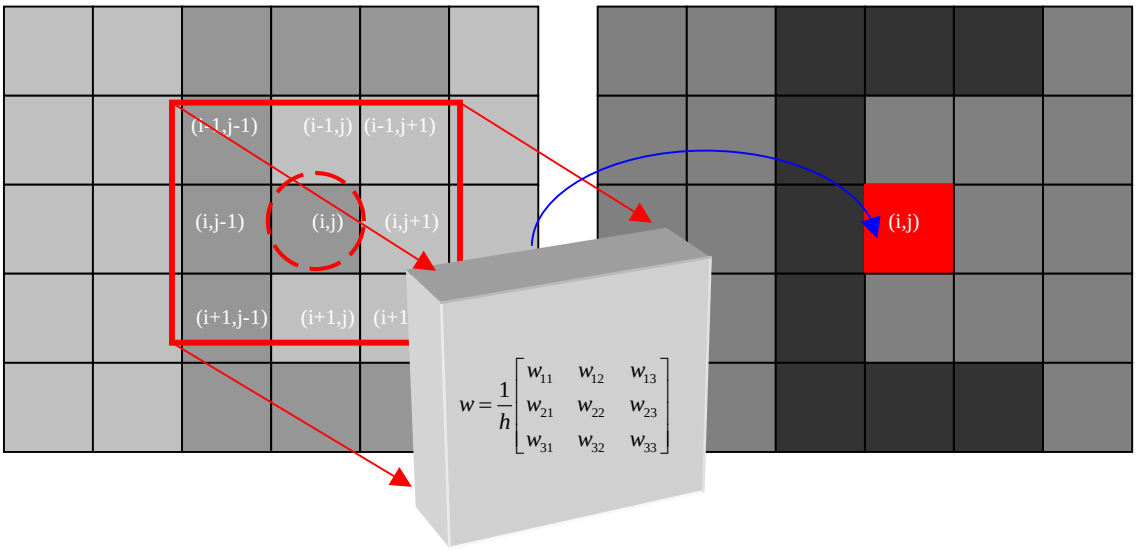



**Figure 8**. Spatial filtering for one mask of 3 x 3.

To develop in matlab this operation the function `nlfilter` is used, whose structure is the following

```
IT=nlfilter(IS,[i j],fun);
```

where ***IT*** is the variable that contains the resulting image of the operation, ***IS*** it is the variable that contains to the original image, **[*i j*]** are the dimensions of the mask that defines the influence of the neighbors for the calculate of the new pixel, finally **fun** represents the function that develops the calculate on the elements in a defined neighborhood dimension *i* x *j*.

The function ***fun*** receives as input a matrix **x** of *i* x *j* data corresponding to the pixel neighbors of the image which are processed by the function and then returning the value that corresponds to the element centered in the mask.

To implement a low pass filter on the image of the figure 1 this should be represented in the variable image3, having as mask

$$w = \frac{1}{9}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

For it is implemented a **.m** file with name **myfunction** and whose content will be integrated by the following lines

```
function  result=myfunction(x)
result=1/9*(x(1,1)+x(1,2)+x(1,3)+x(2,1)+x(2,2)+x(2,3)...
            +x(3,1)+x(3,2)+x(3,3));
```

To carry out the filtering we have to write on command line

```
>> image3=double(image3);
>> imageR=nlfilter(image3,[3 3],@myfunction);
>> imageR=uint8(imageR);
>> imshow(imageR);
```

The result is shown in the figure 9.

**Figure 9**. Resulting image of the spatial filtering.

*Edge detection functions*

In computer vision is of utility for the object recognition or to segment regions, to extract the borders of objects that define in theory their sizes and regions. The function `edge` gives the possibility to obtain the borders of the image. The function `edge` allows to find the borders by means of two different algorithms that can be chosen, canny and sobel. The format of this function is

```
ImageT=edge(ImageS, 'algorithm');
```

Where ***ImageT*** is the image obtained with the extracted borders, ***ImageS*** is the variable that contains the grayscale image which we seek to recover its borders, while ***algorithm*** can be one of the two canny or sobel. Thus, if we want to recover the borders of the grayscale image contained in the variable `imagegray` using the canny algorithm we have to write

```
>>ImageR=edge(imagegray, 'canny');
```

The figure 10 shows an example of the function.

**Figure 10**. Resulting image of the canny algorithm.

*Binary images and segmentation for threshold.*

A binary image is an image in which each pixel can have only one possible values 1 or 0. An image under those conditions is easier to find and to distinguish structural characteristic.

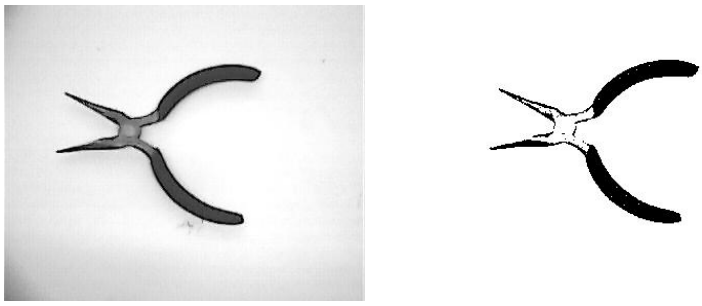
In computer vision the processing with binary images is very important either to carry out segmentation for intensity image in reconstruction algorithms or for structure recognition.

The common form of generating binary images is by means of the use of the threshold value ; that is to say a limit value is chosen (or an interval), starting from which all the bigger intensity values will be coded as 1 while those that are for under it will be coded to zero. In matlab this type of operations are carried out in a simple way using the properties of overload of the relation operators.

For example if we want on the image sample to carry out this operation, that the pixels bigger at 128 is considered as 1 and those that are smaller or similar to 128 as zero, it would be written on command line as

```
>>result=sample>=128;
```

The figure 11 shows the original image and the result of applied the previous instruction.



(a) (b)

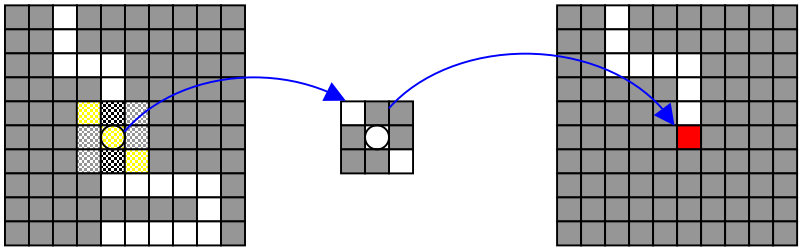
**Figure 11**. a) Original image and b) Resulting image of the application of a threshold of 128.

*Morphological operations*

One of the common operations in computer vision on binary images are the morphological operations. The morphological operations are operations carried out on binary images based on forms. These operations take as input a binary image returning as a result also an image binary. The value of each pixel of the binary resulting image is based on the value of the corresponding pixel of the binary original image and its

neighbors. Then choosing appropiately the neighbors structure can be carry out morphological operations sensitive to a particular form.

The main morphological operations are the dilation and erosion. The dilation operation adds pixels in the border object, while the erosion removes them. In both operations a structure is used that determines which neighbors of the central element of the mask will be taken for the determination of the resulting pixel. The structure is an square arrangement that contains ones and zeros, in the places that contains ones will be the neighbors of the original image considering the central píxel that will be taken to determine the resulting pixel of the image, while the places that have zeros are not taken into account. The image 12 shows graphically the effect of the mask on the original image and its result in the final image.



(a) (b)

**Figure 12**. a) Original image and b) resulting image of the application of the morphological operation considering the structure of 3 x 3.

As the figure 12 shows only the pixels of yellow color in the original image participates in the determination of the red píxel of the resulting image.

Once assigned the size of the structure and their configuration, then the morphological operation is applied. In the case of the dilation if some of the pixels of the structure configured as ones coincides with *at least one* of the image, then the pixel is set to one. On the contrary in the erosion, all the pixel of the structure configured as ones must coincide with *all* those of the image, if this does not happen the pixel is set to 0.

In matlab the functions used to carry out these two morphological operations are `erode` and `dilate`. The format of both functions are

```
ImageR=erode(ImageS,w);
ImageR=dilate(ImageS,w);
```

Where ***ImageR*** is the variable that receives the resulting image, ***ImageS*** is the original binary image to which we want to apply the morphological operation and ***w*** it is a matrix of ones and zeros that determines the format and structure of the mask.

Let us consider that we want to apply to the binary image shown in the figure 11.b the morphological operation of the erosion considering as structure the represented in the

figure 12. Then we would have to write supposing that the binary image is contained in the variable imagebinary the following

```
>>w=eye(3);
>>imageR=erode(imagebinary,w);
```

where eye(3) generates a identity matrix of 3 x 3 that it is used as structure for the function erode. The figure 13 shows the obtained result.

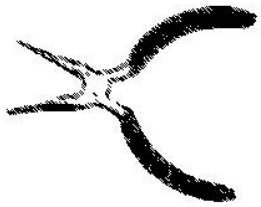


**Figure 13**. Resulting image of the function erode.

*Object-Based Operations*

In a binary image can be defined an object as a group of connected pixels with value 1. For example the figure 14 represent a binary image containing a square object of 4 x 4. The rest of the image can be considered as the background.

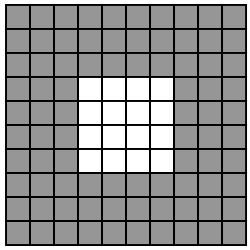


**Figure 14**. Binary image containing an object.

For many operations the distinction of objects depends of the connectivity convention used that is the form in which is considered if two pixel are part of the same object. The connectivity can be of two types, connection-4 or connection-8. In the figure 15 both connectivities are schematized.


(a) (b)

**Figure 15**. (a) Connectivity connection-8 and (b) connection-4.

In the connectivity connection-8 the red pixel belongs to the same object if exists a pixel of value one in the positions 1,2,3,4,5,6,7 and 8. On the other hand the connectivity connection-4 relates only to the pixels 2,4,5 and 7.

For the operations that consider connectivity as parameter is important to take into account that this determines strongly the final processing result, because can be considered objects where if another connectivity is chosen they will not exist. To explain the above-mentioned we consider the figure 16. As we can see if we choose the connectivity connection-8 the figure contained in the image is considered as only one element but if it is chosen the connectivity connection-4 we have 2 different objects.

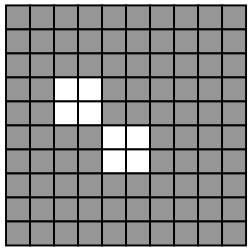


**Figure 15**. The connectivity problem.

The function bwlabel carries out a labeled of the existent components in the binary image, which can be considered as a form of to find how many elements (considering as elements groupings of ones with some of the two connectivity approaches) are present in the image. The function has the following format

```
ImageR=bwlabel(ImageS, connectivity);
```

Where ***ImagenR*** is the image that contains the labeled elements with the number corresponding to the object, ***ImagenS*** is the binary image that we want to find the number of objects and ***connectivity*** can be 4 or 8 (corresponding to the connectivity type previously explained).

The effect of this function can be explained easily if it is analyzed the original image and the result of applying the function `bwlabel`. This is shown in the figure 16.

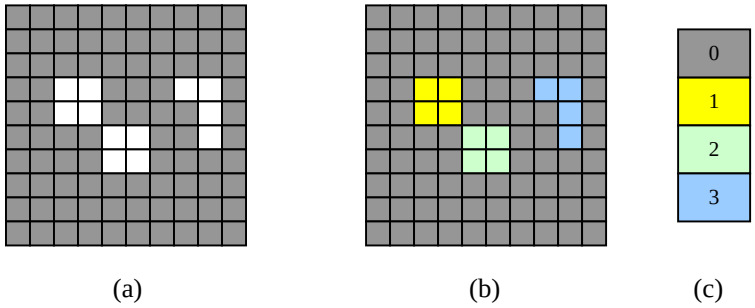



**Figure 16**. a) binary original Image, b) Resulting image of the operation `bwlabel` Considering as connectivity connection-4 and (c) the color code.

Let us suppose that the image (a) is the original binary image (***ImagenR***) and (b) it is the resulting image (***ImagenS***) of executing the sentence

```
>>ImagenR=bwlabel(ImagenS,4);
```

the resulting image assigns to each pixel belonging to a certain object according to its connectivity the label of the object number while the pixels in zero does not have effect in the operation, in the figure 16 this corresponds to the color code shown in (c).

The resulting image is of the type `double`. Also due to their content (the values are very small) cannot be ploted by the function `imshow`. An useful technique for the visualization of this type of matrices is the pseudo-color in form of an index image. An index image is an image that can represent color images like RGB but it uses a different format, instead of using three planes as RGB images, it uses a matrix and a table, the matrix contains in each pixel an integer number corresponding to the table index, while each index of the table corresponds 3 values corresponding to the RGB planes, with it is possible to reduce the image size when reducing the number of different colors. The figure 17 shows an example of index image.

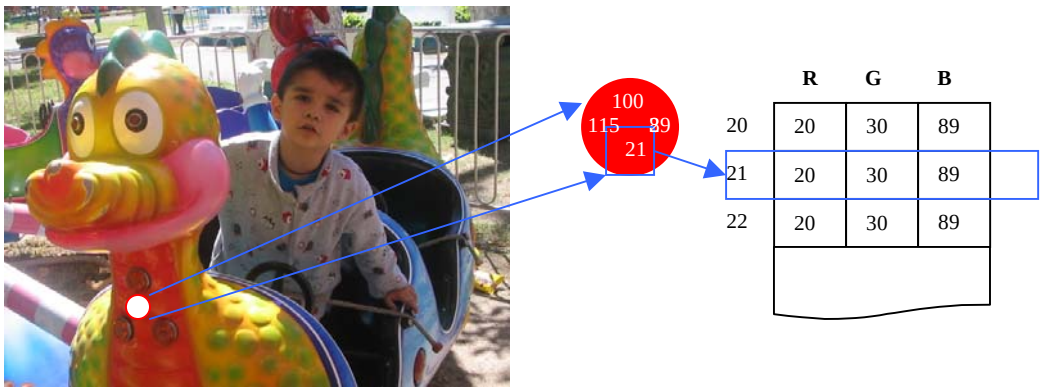
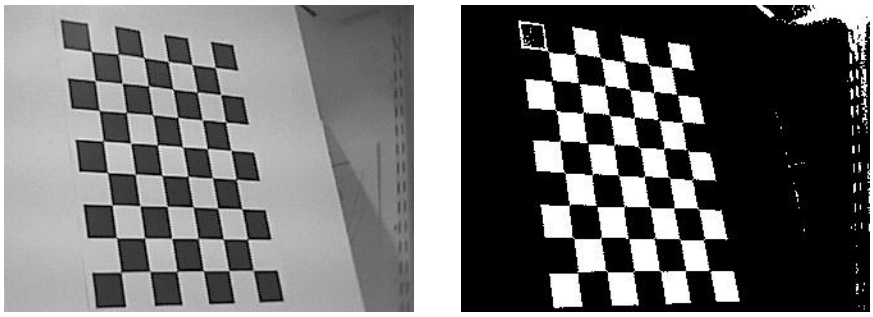



**Figure 17**. Example of index image.

Using an index image each object of the resulting image of applying the function `bwlabel` can be ploted in a different color and identified quickly. It is important to emphasize that the necessary number of index to form the image is similar to the number of labels found by the function `bwlabel` plus one, because the background constituted of zeros is also an index.

Let us consider an example to illustrate the previous technique. Let us suppose that we have the grayscale image represented by the figure 18.a and we convert it to binary applying a threshold of 85 obtaining in this way the image 18.b.

```
>>imagebinary=imagegray<85;
```



(a) (b)

**Figure 18**. a) Grayscale image and b) binary image.

The function `bwlabel` is applied to obtain the contained objects in the binary image considering the connectivity connection-8

```
>>Mat=bwlabel(imagebinary,8);
```

We find the number of contained objects in the image

```
>>max(max(Mat))
ans=
      22
```

We generate the index image with 23 elements

Label number + 1

```
>>map=[0 0 0;jet(22)];
>>imshow(Mat+1,map,' notruesize' )
```

Giving the image represented in the figure 19 as result.

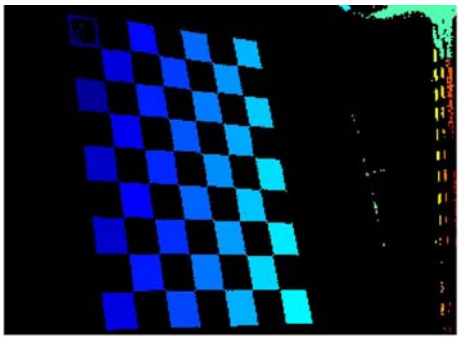


**Figure 19**. Result of the example of the function `bwlabel`.

*Object Selection*

In computer vision is of special utility to be able to isolate objects of a binary image with a quick and interactive method. The function of matlab `bwselect` allows interactively to select the binary object to segment with only to point out it in the window previously ploted by means of the function imshow. The format of the function is

```
ImageR=bwselect(c);
```

Where ***ImageR*** is the image containing the selected object while c represents the connectivity approach used. Likewise that other used interactive functions in this document it is necessary to select with the mouse the object in the binary image to isolate, pressing the right button and later the enter key. The following example illustrates the use of the function. Let us consider that we have an grayscale image as the exemplified by the figure 20.a and we convert it to binary with a threshold of 140 to obtain this way the figure 20.b

```
>>imagebinary=imagegray<140;
```

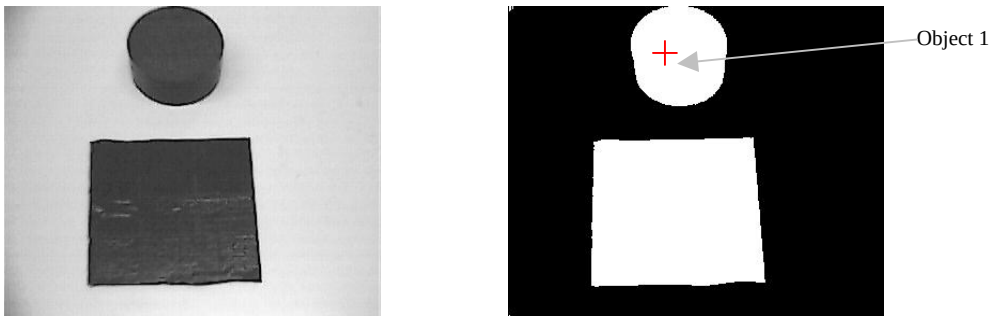



**Figure 20**. a) Grayscale image and b) binary image.

Now if we write

```
>>imagesegment=bwselect(8);
```

and we select the object 1 as it shows in the figure 20b. Of the above-mentioned we would have as resulting image ***imagesegment*** represented in the figure 21.

**Figure 21**. Representation of the variable ***imagesegment***.

*Feature measurement*

In computer vision is of particular interest to find feature such as the area, centroid and others of previously labeled objects by the function `bwlabel` with the objective of identifying their position in the image. Matlab has the function `imfeature` to find such feature. The format of this function is

```
Stats=imfeature(L,' measurement');
```

Where Stats is a structured type that contains the measurement value indicated in the chain text ***measurement***, ***measuremen***t is a text chain that indicates to the function the measurement to carry out on the contained objects in the image. The group of possible measurements is represented in the table 2.

| Area | Image | EulerNumber |
|---|---|---|
| Centroid | FilledImage | Extrema |
| BoundingBox | FilledArea | EquivDiameter |
| MajorAxisLength | ConvexHull | Solidity |
| MinorAxisLength | ConvexImage | Extent |
| Orientation | ConvexArea | PixelList |

**Table 2.** Measurement carried out by the function `imfeature`.

It is important to notice that **Stats** is a structured type that contains the measurement of all contained objects in the binary image, for what this structured type is indexed; that

is to say we can control an index the measurement of the object that we want. For example if the image contains 4 elements, we can access to the measurement of each object writing

```
stats(1).Medicion
stats(2).Medicion
stats(3).Medicion
stats(4).Medicion
```

Let us consider that we have the binary image (contained in the variable imagebinary) represented in the figure 22 and we try to find the centroid of both figures contained an the image.

**Figure 22**. Image used to exemplify the use of the function `imfeature`.

As first step we use the function `bwlabel` and we verify the object number

```
>>imageR=bwlabel(imagebinary,8);
>>max(max(imageR))
 ans=

       2
```

Now the function `imfeature` is used to find the centroid of both figures

```
>>s=imfeature(imageR,' Centroid' );
>> s(1).Centroid
ans =
   81.9922   86.9922
>> s(2).Centroid
ans =
  192.5000   85.0000
```

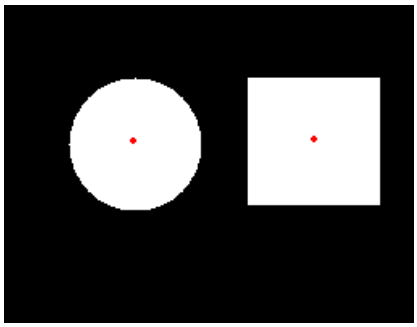


**Figure 23**. Centroid identification.

*The conversion function of images and color models*

The format of color representation offered by the images RGB is not appropriate for applications in which the illumination change an is a problem for example the color segmentation. Another type of less sensitive color formats to the illumination change is the HSV model. Matlab has special functions to carry out conversions between color models and to change color images to grayscale; some of those functions will be treated in this section.

The function `rgb2gray` changes an image in format RGB to grayscale, the format of this function is

```
imagegray =rgb2gray(imageRGB);
```

On the other hand the function `rgb2hsv` changes the RGB colormodel to the HSV model, this function takes as input an RGB image compound of three planes and returns the HSV image composed of three planes corresponding to the H, S and V. The format of this function it is

```
Imagehsv=rgb2hsv(imageRGB);
```

The contrary conversion ist carried out with the function hsv2rgb.

To exemplify the use of these functions we consider the image represented in the figure 24. The idea is to transform this image to a less sensitive model to the illumination changes as it is the HSV model and segment it.

**Figure 24**. RGB image.

Writing on command line we obtain as a result the HSV image that if we represent it considering the RGB model it would have the aspect of the figure 25.

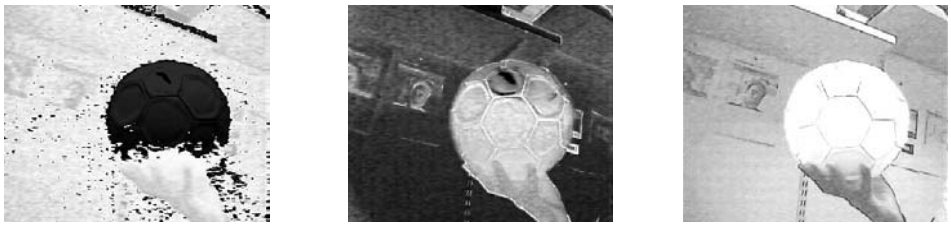
```
>>imageHSV=rgb2hsv(imageRGB);
```

**Figure 25**. HSV image.

If we divide the image in their respective HSV planes, we will write

```
>>H=imageHSV(:,:,1);
>>S=imageHSV(:,:,2);
>>V=imageHSV(:,:,3);
```

we obtain as a result the figures 26.a, 26.b and 26.c.



(a) (b) (c)

**Figure 26**. Image planes a) H, b) S and c) V.

Of the previous images it is evident that can be used the plane S (saturation) to segment the object.

## The vfm tool

However an important problem in the previous versions to the release 14 (in this version with the incorporation of the toolbox of image acquisition the problem was resolved) is the lack connection between an image taken by video card ( or simply a Web-cam) and the processing carried out by matlab. To have this advantage allows to implement the vision algorithms with the real characteristics of camera without any additional effort as to implement a program to record the captured image in a file (with a specify format), for later to use the normal toolbox commands to open the file.

The vfm tool [3] allows to resolver this problem, vfm is a group of dynamic libraries that allow directly to access the device controllers registered by windows. Thus USB Web-cams of low cost can be used as well as other devices to capture an image and later to use it with matlab and its toolboxes.

The tool once installed allows to be used in a simple way. To use this it is only necessary to write

```
>>vfm
```

Then a window will appear like the shown in the figure 27.


**Figure 27**. Window of the vfm tool.

The tool has three menus to control the capture. In the menu **Driver** is allowed to select one of several installed video sources, in the menu **Configure** can configure the format and characteristic of the previously selected driver, finally in the menu **Window** is allowed to control the flow of the presentation of the image, the mode *Preview* is usually preferred so that the image is shown in all moment in the window.

*Image capture in matlab*

The captured images using the vfm tool are stored in RGB format.To capture the images surrendered by the drivers in matlab is used the function **vfm** that has the following format

```
ImageCaptured=vfm( 'grab' ,n);
```

Where ImageCaptured is the variable that contains the image RGB, and **n** it is the number of frames to capture, in the case that **n** is more as one then the variable of the captured image is indexed where each index represents the frame number.

In such a way that if we want to capture the current frame seen by the camera we will write

```
>>ImageRGB=vfm( 'grab' ,1);
>>imshow(ImageRGB);
```

We will obtain as a result the shown in the figure 28.

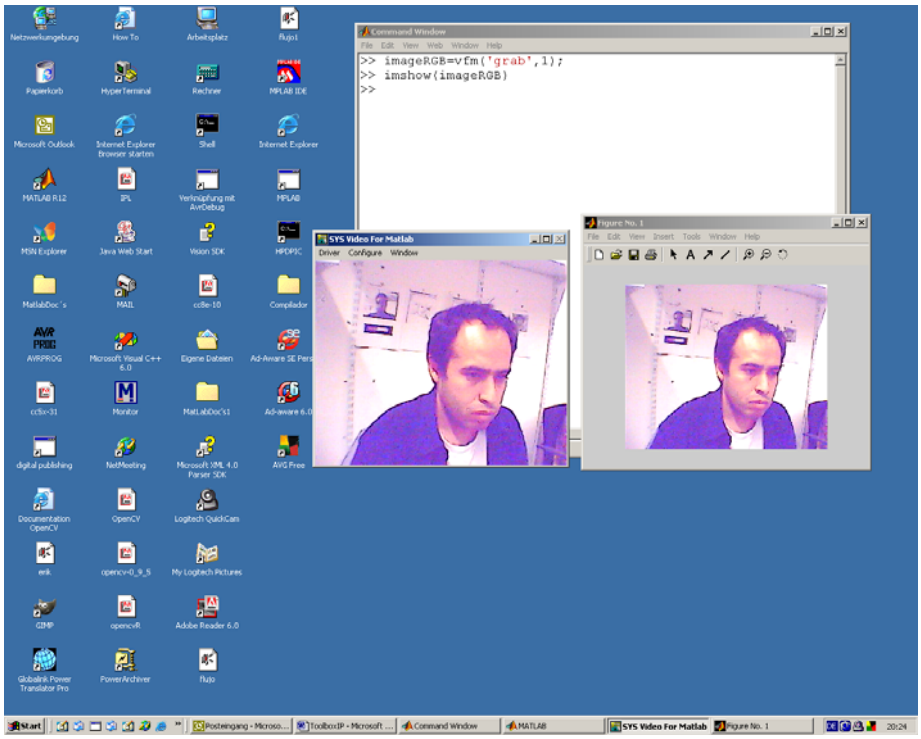



**Figure 28**. Capture process of vfm tool.

## References

**[1]** Gonzales C. Rafael, Woods E. Richard, Digital image Processing, Prentice Hall, 2002 New York.

**[2]** Gonzales C. Rafael, Woods E. Richard, Eddins L. Steven, Digital Image Processing using MATLAB, Prentice Hall, 2002 New York.

**[3]** http://www2.cmp.uea.ac.uk/~fuzz/vfm/default.html.

**[4]** Marchand Patrick, Holland O. Thomas, Graphics and GUIs with MATLAB, Chapman & Hall/CRC, 2002.