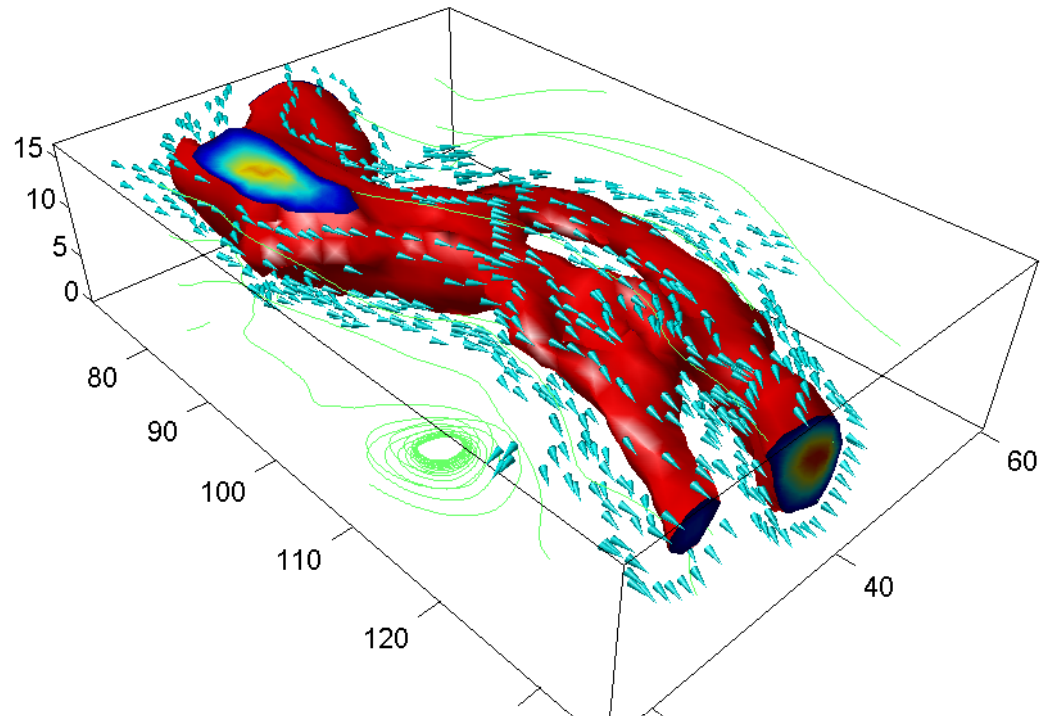
Visión por Computador utilizando MatLAB Y el Toolbox de Procesamiento Digital de Imágenes



Erik Valdemar Cuevas Jimenez
Daniel Zaldivar Navarro

Índice general

Índice de figuras

Capítulo 1

Introducción

La implementación de algoritmos en visión por computador resulta muy costoso en tiempo ya que se requiere de la manipulación de punteros, gestión de memoria, etc. Hacerlo en lenguaje C++ (que por sus características compartidas de alto y bajo nivel lo hacen el mas apropiado para la implementación de algoritmos de visión computacional) supondría la inversion de tiempo y sin la seguridad de que lo queremos implementar funcionará. Además utilizar C++ para el periodo de prueba exige un tiempo normal de corrección de errores debidos al proceso de implementación del algoritmo, es decir errores programáticos efectuados por ejemplo al momento de multiplicar dos matrices, etc. Todos estos problemas pueden ser resueltos si la implementación de prueba es realizada en MatLAB utilizando su toolbox de procesamiento de imágenes con ello el tiempo de implementación se convierte en el mínimo con la confianza de utilizar algoritmos científicamente probados y robustos.

El toolbox de procesamiento de imágenes contiene un conjunto de funciones de los algoritmos mas conocidos para trabajar con imágenes binarias, trasformaciones geométricas, morfología y manipulación de color que junto con las funciones ya integradas en matlab permite realizar análisis y trasformaciones de imágenes en el dominio de la frecuencia (trasformada de Fourier y Wavlets).

Este documento esta dividido en 3 partes, en el capitulo 2 se trata los conceptos básicos de las imágenes y como son representadas en matlab así como una introducción a las operaciones básicas de manejo de archivos. En el capitulo 3 se aborda el procesamiento de imágenes mas comunes y representativos en el área de visión computacional ; explicando el uso de estas funciones a través de ejemplos. Por ultimo en el capitulo 4 se explica el uso de la herramienta *vfm* utilizada para la captura de imágenes captadas por dispositivos instalados en la computadora tales como tarjetas captadoras y USB Webcams.

Capítulo 2

Conceptos básicos de las imágenes

En matlab una imagen a **escala de grises** es representada por medio de una matriz bidimensional de m x n elementos en donde n representa el numero de píxeles de ancho y m el numero de píxeles de largo. El elemento v11 corresponde al elemento de la esquina superior izquierda (ver figura 1.1), donde cada elemento de la matriz de la imagen tiene un valor de 0 (negro) a 255 (blanco).

Por otro lado una imagen de **color RGB** (la mas usada para la visión computacional, además de ser para matlab la opción default) es representada por una matriz tridimensional m x n x p, donde m y n tienen la misma significación que para el caso de las imágenes de escala de grises mientras p representa el plano, que para RGB que puede ser 1 para el rojo, 2 para el verde y 3 para el azul. La figura 2.2 muestra detalles de estos conceptos.

2.1. Lectura y escritura de imágenes a través de archivo

Para leer imágenes contenidas en un archivo al ambiente de matlab se utiliza la función *imread*, cuya sintaxis es

```
imread('nombre del archivo')
```

Donde nombre del archivo es una cadena de caracteres conteniendo el nombre completo de la imagen con su respectiva extensión, los formatos de imágenes que soporta matlab son los mostrados en la tabla 2.1.

Para introducir una imagen guardada en un archivo con alguno de los formatos especificados en la tabla anterior solo tiene que usarse la función *imread* y asignar su resultado a una variable que representará a la imagen (deacuerdo a la estructura, Figura 2.1 para representar escala de grises y Figura 2.2 para

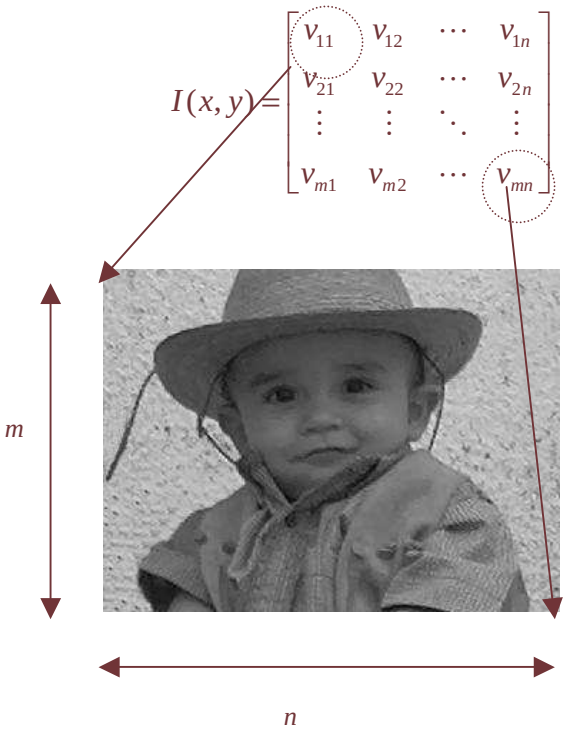


Figura 2.1: Representación de una imagen a escala de grises en MatLAB.

Formato	Extensión
TIFF	.tiff
JPEG	.jpg
GIF	.gif
BMP	.bmp
PNG	.png
XWD	.xwd

Cuadro 2.1: Formatos y extensiones soportadas por MatLAB.

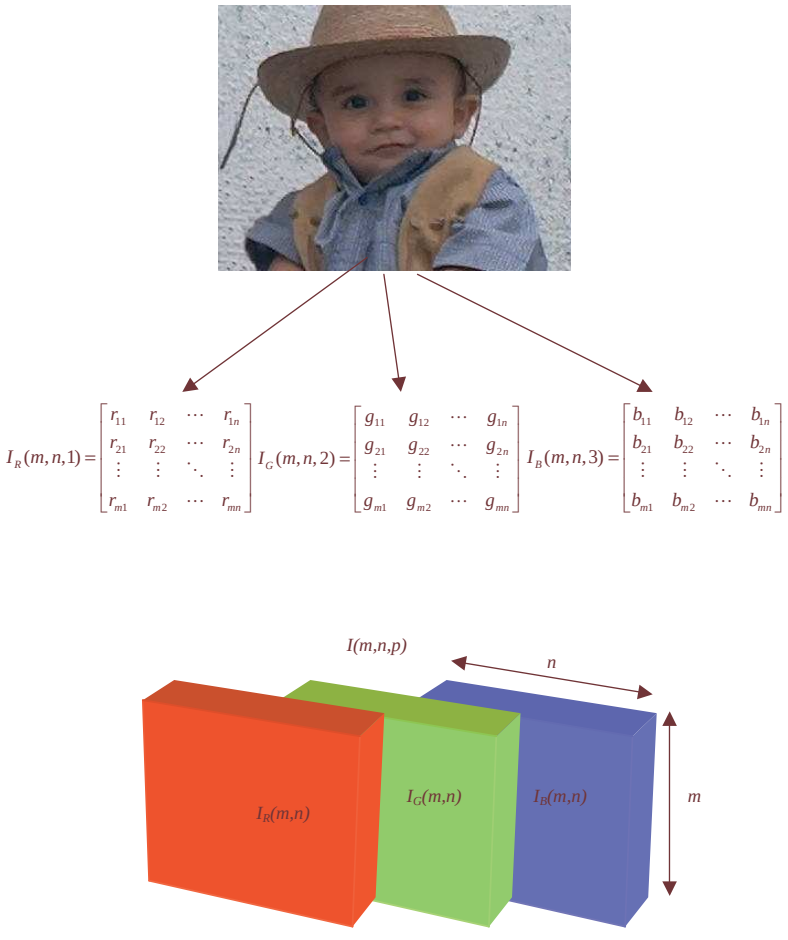


Figura 2.2: Representación de una imagen a color RGB en MatLAB.

RGB). De tal forma que si se quisiera introducir la imagen contenida en el archivo data.jpg a una variable para su procesamiento en matlab, entonces se tendría que escribir en línea de comandos:

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

con ello la imagen contenida en el archivo *data.jpg* quedará contenida en la variable *image*.

Una vez que la imagen esta contenida en una variable de matlab es posible utilizar las funciones para procesar la imagen. Por ejemplo, una función que permite encontrar el tamaño de la imagen es size(variable)

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

en donde m y n contendrán los valores de las dimensiones de la imagen.

Para grabar el contenido de una imagen en un archivo se utiliza la función `imwrite(variable,'nombre del archivo')`, en donde *variable* representa la variable que contiene a la imagen y *nombre del archivo* el nombre del archivo con su respectiva extensión de acuerdo a la tabla 2.1. Suponiendo que la variable *image2* contiene la imagen que nos interesa grabar en el archivo *dato2.jpg* tendríamos que escribir:

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

Después que realizamos un procesamiento con la imagen, es necesario desplegar el resultado obtenido, la función `imshow(variable)` permite desplegar la imagen en una ventana en el ambiente de trabajo de matlab. Si la variable a desplegar por ejemplo, es *face* al escribir en la línea de comandos:

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

obtendríamos la imagen de la figura 2.3.

2.2. Acceso a píxel y planos en las imágenes

El acceso a píxel de una imagen es una de las operaciones mas comunes en visión computacional y en matlab esta sumamente simplificado; solo bastará con indexar el píxel de interés en la estructura de la imagen. Consideremos que tenemos una imagen *image1* en escala de grises (Figura 2.4) y deseamos obtener su valor de intensidad en el píxel especificado por m=100 y n=100 ; solo tendríamos que escribir

```
>>image1(100,100)

ans =

  84
```

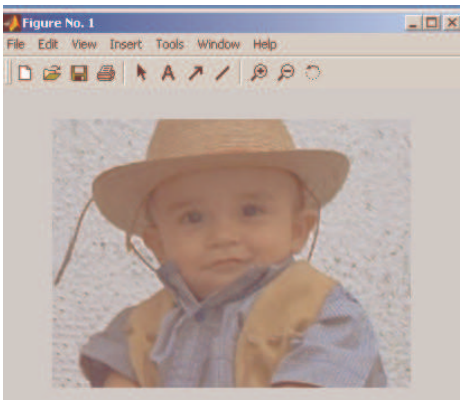


Figura 2.3: Imagen mostrada al utilizar la función `imshow`.

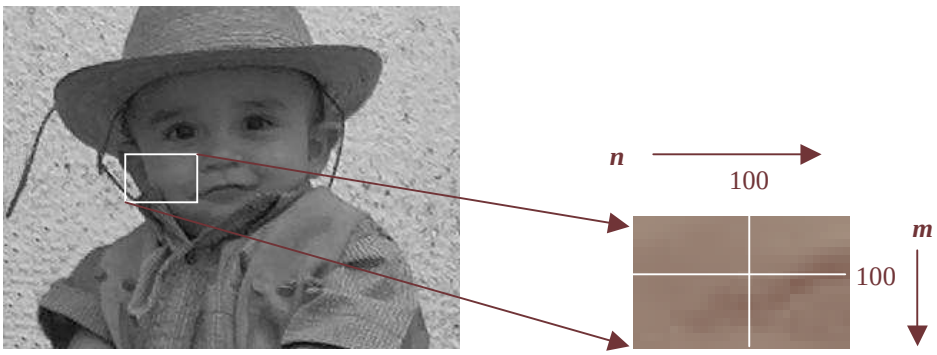


Figura 2.4: Obtención del valor de un píxel de image2.

(a) (b) (c)

Figura 2.5: Planos de la imagen a) rojo, b) verde y c) azul.

De igual forma si se desea cambiar el valor de este píxel a negro, es decir asignarle el valor de *0* lo que tendría que escribirse en línea de comandos es:

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

En el caso de imágenes a escala de grises estas solo tienen un plano, constituido por la matriz m x n que contiene los valores de intensidad para cada índice. Sin embargo las imágenes de color cuentan con mas de un plano. En el caso de imágenes RGB (tal como se explico arriba) estas cuentan con 3 planos uno para cada color que representa. Consideremos ahora que la imagen RGB contenida en la variable *image2* es la mostrada en la figura 2.3, y deseamos obtener cada uno de los planos que la componen. Entonces escribiríamos:

```
>>planeR=image2( :, :,1) ;

>>planeG=image2( :, :,2) ;

>>planeB=image2( :, :,3) ;
```

Los planos resultantes por los anteriores comandos son mostrados en la figura 2.5.

Si se deseará manipular un píxel de una imagen a color RGB este tendrá un valor para cada uno de sus planos correspondientes. Supongamos que tenemos la imagen RGB contenida en la variable *image2* y deseamos obtener el valor del píxel m=100 y n=100 para cada uno de los diferentes planos R, G y B. Tendríamos que escribir:

```
>>valueR=image2(100,100,1) ;

>>valueG=image2(100,100,2) ;

>>valueB=image2(100,100,3) ;
```

Lo cual dará como resultado una tripleta de valores. De igual forma que con el caso de escala de grises podemos modificar este píxel a otro color mediante el cambio de su valor en cada uno de sus respectivos planos; por ejemplo un cambio a color *blanco* mediante:

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

>>image2(100,100,2)=255;

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

En ocasiones resulta preferible saber el color o la intensidad de gris (el valor del píxel) de forma iteractiva, es decir tener la posibilidad de seleccionar un píxel en una región y obtener el valor de este. Esta posibilidad es ofrecida por la función impixel, la cual iterativamente entrega el valor (uno o tres) del píxel seleccionado que aparezca en la ventana desplegada por la función imshow. El formato de esta función es:

```
value=impixel;
```

Donde **value** representa un escalar, en el caso de que la imagen sea a escala de grises o bien un vector de 1 x 3 con los valores correspondientes a cada uno de los planos RGB.

Para utilizar esta función es necesario antes, desplegar la imagen con la función *imshow*. Una vez desplegada se llama a la función y cuando el cursor del ratón este sobre la superficie de la imagen cambiara a una +. Cuando se presione el botón izquierdo del ratón se seleccionara el píxel, el cual podemos seleccionar otra vez en caso de que se allá cometido un error a la hora de posicionar el ratón, ya que la función seguirá activada hasta que se presione la tecla de enter. La figura 2.6 muestra una imagen de la operación aquí descrita.

Una operación importante en visión computacional es el determinar un perfil de la imagen; es decir convertir un segmento de la imagen a una señal unidimensional para analizar sus cambios. Esto es de especial significado en la visión estereo en donde se analizan para los algoritmos segmentos epipolares de cada camara. Matlab dispone de la función *improfile* que permite trazar el segmento interactivamente con el ratón, desplegando después el perfil de la imagen en una grafica diferente. Esta función necesita que la imagen original sea previamente desplegada mediante la función *imshow*. Debe de considerarse que si la imagen es a escala de grises, el perfil mostrara solo una señal correspondiente a las fluctuaciones de las intensidades de la imagen, sin embargo si la señal es de color RGB esta mostrara un segmento de señal para cada plano. Para la utilización de esta función solo es necesario escribir en línea de comandos

```
>>improfile
```

como es una función iteractiva en cuanto el ratón se encuentra en la superficie de la imagen el puntero cambiara de símbolo a una +, de esta manera podemos mediante el establecimiento de una línea en la imagen configurar el perfil deseado. La figura 2.7 muestra una imagen de la operación descrita.

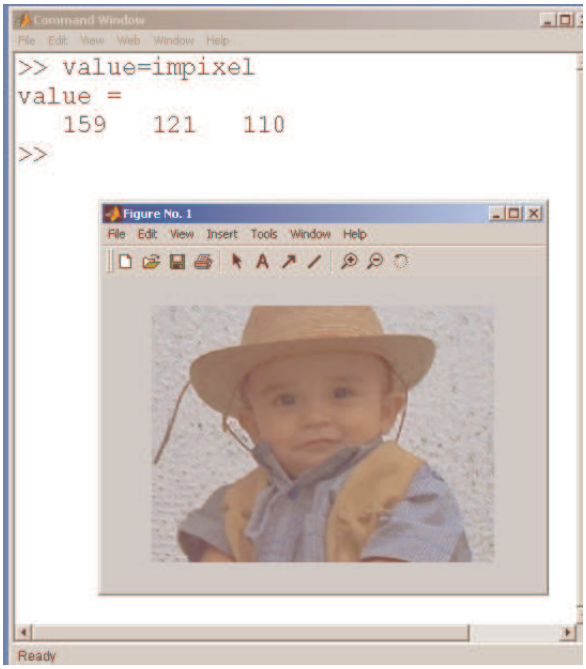


Figura 2.6: Utilización de la función `impixel`.

Figura 2.7: Utilización de la función `improfile`.

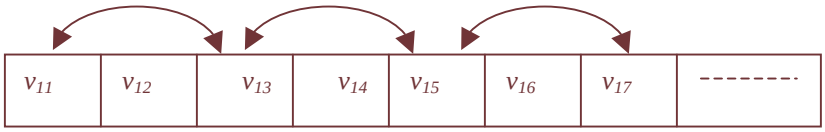


Figura 2.8: Ejemplo de submuestreo

2.3. Sub-muestreo de imágenes

En ocasiones es necesario hacer cálculos que requieren procesar por completo la imagen, en estos casos hacerlo sobre la resolución original de la imagen seria muy costoso. Una alternativa mas eficiente, resulta el sub-muestreo de la imagen. Sub-muestreo significa generar una imagen a partir de tomar muestras periódicas de la imagen original, de tal forma que esta quede mas pequeña. Si se considera la imagen *I(m,n)* definida como:

$$I(m,n) = \begin{bmatrix} v_{11} & v_{12} & v_{13} & \cdots & v_{1n} \\ v_{21} & v_{22} & v_{23} & \cdots & v_{2n} \\ v_{31} & v_{32} & v_{33} & \cdots & v_{3n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ v_{m1} & v_{m2} & v_{m3} & \cdots & v_{mn} \end{bmatrix} \tag{2.1}$$

y se desa sub-muestrear la imagen para obtener la mitad de su tamano original, asi la nueva imagen quedaría compuesta por los elementos tomando *uno si y otro no* de la imagen original:

$$I_{S_2}(m,n) = \begin{bmatrix} v_{11} & v_{13} & v_{15} & \cdots & v_{1(n-2)} \\ v_{31} & v_{33} & v_{35} & \cdots & v_{3(n-2)} \\ v_{51} & v_{53} & v_{55} & \cdots & v_{5(n-2)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ v_{(m-2)1} & v_{(m-2)3} & v_{(m-2)5} & \cdots & v_{(m-2)(n-2)} \end{bmatrix} \tag{2.2}$$

El mismo concepto se explica graficamente en la figura 2.8.

Considerando que se tiene la imagen RGB en la variable image2 y la Sub-muestreamos a la mitad tendríamos que escribir el siguiente código en la línea de comandos:

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

2.4. Tipo de dato de los elementos de una imagen

Los elementos que constituyen una imagen en matlab tienen el formato entero uint8, que es un tipo de dato que puede variar de 0 a 255, sin poder

soportar decimales y valores que salgan fuera de ese rango. Lo anterior resulta una desventaja principalmente en aquellos casos donde se implementan algoritmos que trabajan con este tipo de datos para realizar operaciones de división o multiplicación por tipo de dato flotante. En estos casos es necesario trasformar la imagen de tipo de dato *uint8* a *double*. Es importante tener en cuenta que si se utiliza la función *imshow* para desplegar las imágenes; esta no tiene la capacidad de poder desplegar imágenes del tipo *double* por lo que una vez realizado las operaciones de punto flotante es necesario después convertir al tipo de dato *uint8*. Supongamos que tenemos una imagen de escala de grises representada en la variable *imagegray* y queremos reducir sus intensidades a la mitad, entonces escribiríamos:

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

Capítulo 3

Procesamiento de la imagen.

El numero de funciones que implementa el toolbox para el procesamiento de imagen es muy diverso, sin contar la múltiple oferta de funciones ya generada por otros usuarios y disponibles a través del Internet, sin embargo en este tutorial serán tratadas algunas consideradas como las mas usadas y útiles para la visión computacional.

3.1. Filtraje espacial

El filtraje espacial es una de las operaciones comunes en la visión computacional ya sea para realizar efectos de eliminación de ruido o bien detección de bordes. En ambos casos la determinación de los píxeles de la nueva imagen dependen del píxel de la imagen original y sus vecinos. De esta forma es necesario configurar una matriz (mascara o ventana) que considere cuales vecinos y en que forma influirán en la determinación de el nuevo píxel. Consideremos una imagen $I_S(m,n)$.

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

que será la imagen a la cual pretendemos filtrar espacialmente y $I_T(m,n)$:

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

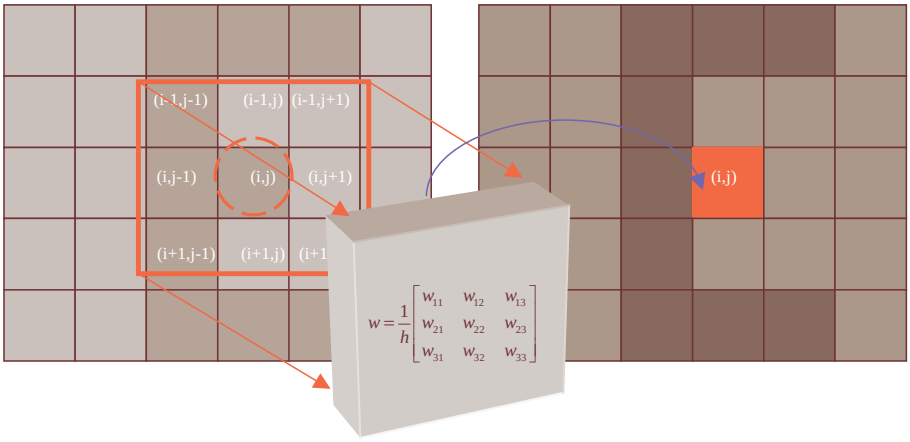


Figura 3.1: Filtrado espacial por una mascara de 3 x 3.

como la imagen resultante ; si además consideramos a *w(r,t)* la matriz que se utiliza para realizar el filtrado :

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

donde r=3 y t=3,tendremos que cada elemento de $I_T(m,n)$ es calculado:

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

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

La figura 3.1 muestra estos detalles.

Para desarrollar en matlab este tipo de operaciones se utiliza la función `nlfilter`, cuya estructura es la siguiente:

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

donde **IT** es la variable que contiene la imagen resultado de la operación, **IS** es la variable que contiene a la imagen original, **[i j]** son las dimensiones de la mascara que define la influencia de los vecinos para el calculo del nuevo píxel por ultimo **fun** representa la función que desarrolla el calculo sobre los elementos de la vecindad definidos de dimensión i x j.

La función **fun** recibe como entrada una matriz **x** de i x j datos correspondientes a los vecinos de la imagen los cuales son procesados por la función devolviendo el valor que corresponde al dato centrado en la mascara.

Figura 3.2: Imagen resultado del filtraje espacial.

Si se deseara implementar un filtro pasabajas sobre la imagen de la figura 2.3 que esta representada en la variable *image3*, teniendo como estructura la mascara:

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

entonces se implementa un archivo **.m** con nombre myfunction y cuyo contenido estará integrado por las siguientes líneas

```
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));
```

Para desempeñar el filtrado se escribiría en línea de comandos

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

El resultado es desplegado en la figura 3.2.

3.2. Funciones para la extracción de bordes

En visión computacional es de utilidad para hacer reconocimiento de objetos o bien para segmentar regiones, extraer los bordes de objetos (que en teoría delimitan sus tamaños y regiones). La función *edge* da la posibilidad de obtener los bordes de la imagen. La función permite encontrar los bordes a partir de dos diferentes algoritmos que pueden ser elegidos, *canny y sobel*. El formato de esta función es:

Figura 3.3: Imagen resultado de la aplicación del algoritmo de canny.

```
ImageT=edge(ImageS, algoritmo);
```

Donde **ImageT** es la imagen obtenida con los bordes extraídos, **ImageS** es la variable que contiene la imagen en escala de grises a la cual se pretende recuperar sus bordes, mientras que **algoritmo** puede ser uno de los dos canny o sobel. De tal forma que si a la imagen en escala de grises contenida en la variable imagegray se le quieren recuperar sus bordes utilizando en algoritmo canny se escribiría en línea de comandos:

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

La figura 3.3 muestra un ejemplo del uso de esta función.

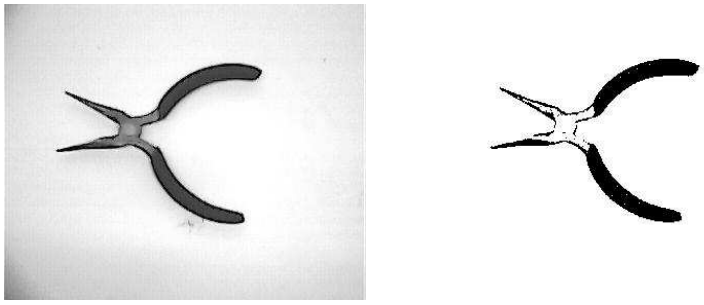
3.3. Imágenes binarias y segmentación por umbral.

Una imagen binaria es una imagen en la cual cada píxel puede tener solo uno de dos valores posibles 1 o 0. Como es lógico suponer una imagen en esas condiciones es mucho mas fácil encontrar y distinguir características estructurales.

En visión computacional el trabajo con imágenes binarias es muy importante ya sea para realizar segmentación por intensidad de la imagen, para generar algoritmos de reconstrucción o recoconocer estructuras.

La forma mas común de generar imágenes binarias es mediante la utilización del valor umbral de una imagen a escala de grises; es decir se elige un valor limite (o bien un intervalo) a partir del cual todos los valores de intensidades mayores serán codificados como 1 mientras que los que estén por debajo serán codificados a cero. En matlab este tipo de operaciones se realizan de forma bastante sencilla utilizando las propiedades de sobrecarga de los símbolos relaciónales.

Por ejemplo si de la imagen **sample** quisiera realizarse este tipo de operación de tal forma que los píxeles mayores a 128 sean considerados como 1 y los que son menores o iguales a 128 como cero, se escribiría en línea de comandos como



(a) (b)

Figura 3.4: a) imagen original y b) imagen resultado de la aplicación de un umbral de 128.

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

La figura 3.4 muestra la imagen original y el resultado de haber aplicado la anterior instrucción.

3.4. Operaciones morfológicas

Una de las operaciones mas utilizadas en visión sobre imágenes previamente binarizadas es las operaciones morfológicas. Las operaciones morfológicas son operaciones realizadas sobre imágenes binarias basadas en formas. Estas operaciones toman como entrada una imagen binaria regresando como resultado una imagen también binaria. El valor de cada píxel de la imagen binaria resultado es basado en el valor del correspondiente píxel de la imagen original binaria y de sus vecinos. Entonces eligiendo apropiadamente la forma de los vecinos a considerar, puede construirse operaciones morfológicas sensibles a una forma en particular.

Las principales operaciones morfológicas son la dilatación y la erosión. La operación de dilatación adiciona píxeles en las fronteras de los objetos, mientras la erosión los remueve. En ambas operaciones como se menciono se utiliza una rejilla que determina cuales vecinos del elemento central de la rejilla serán tomados en cuenta para la determinación del píxel resultado. La rejilla es un arreglo cuadricular que contiene *unos y ceros*, en los lugares que contiene *unos* serán los vecinos de la imagen original con respecto al píxel central, los cuales serán tomados en consideración para determinar el píxel de la imagen resultado, mientras que los lugares que tengan *ceros* no serán tomados en cuenta. La imagen 3.5 muestra gráficamente el efecto de la rejilla sobre la imagen original y su resultado en la imagen final.

Como muestra la figura 3.5 solo los píxeles de color amarillo en la imagen original participan en la determinación del píxel rojo de la imagen resultado.

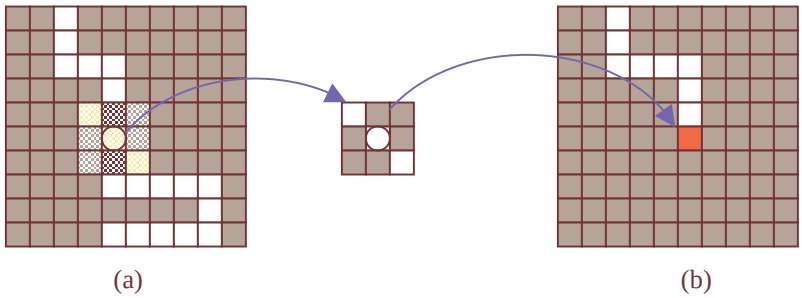


Figura 3.5: a) imagen original y b) imagen resultado de la aplicación de la operación morfológica considerando la rejilla de 3 x 3.

Una vez determinado el tamaño de la rejilla y su configuración. Se aplica la operación morfológica. En el caso de la dilatación, si alguno de los píxeles de la rejilla configurados como *unos* coincido con *al menos uno* de la imagen original el píxel resultado es uno. Por el contrario en la erosión todos los píxel de la rejilla configurados como *unos* deben coincidir con *todos* los de la imagen si esto no sucede el píxel es 0.

En matlab las funciones utilizadas para realizar estas dos operaciones morfológicas son *erode y dilate*. El formato de ambas funciones es:

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

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

Donde **ImageR** es la variable que recibe a la imagen resultado, **ImageS** es la imagen binaria origen a la que se desea aplicar la operación morfológica y **w** es una matriz de unos y ceros que determina el formato y estructura de la rejilla.

Consideremos que quisiéramos aplicar a la imagen binaria mostrada en la figura 3.4 la operación morfológica de la erosión considerando como rejilla la representada en la figura 3.5, entonces tendríamos que escribir suponiendo que la imagen binaria es contenida en la variable imagebinary lo siguiente:

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

donde eye(3) crea una matriz identidad de 3 x 3, que es usada como rejilla para la función erode. La figura 3.6 muestra el resultado obtenido.

3.5. Operaciones basadas en objetos

En una imagen binaria, puede definirse un objeto como un conjunto de píxeles conectados con valor 1. Por ejemplo la figura 3.7 representa una imagen

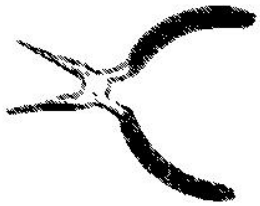


Figura 3.6: Imagen resultado de la función erode.

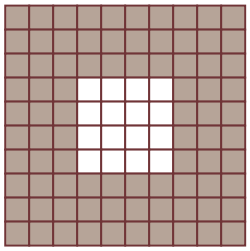


Figura 3.7: Imagen binaria conteniendo un objeto.

binaria conteniendo un objeto cuadrado de 4 x 4. El resto de la imagen puede ser considerado como el fondo.

Para muchas operaciones la distinción de objetos depende la convención utilizada de conectividad, que es la forma en la que se considera si dos píxel tienen relación como para considerar que forman parte del mismo objeto. La conectividad puede ser de dos tipos, de conexión-4 ó bien conexión-8. En la figura 3.8 se esquematiza ambas conectividadades.

En la conectividad conexión-8 se dice que el píxel rojo pertenece al mismo objeto si existe un píxel de valor uno en las posiciones 1,2,3,4,5,6,7 y 8. Por su parte la conectividad conexión-4 relaciona solo a los píxel 2,4,5 y 7.

Para las operaciones que consideran conectividad como un parámetro es im-

Figura 3.8: (a) Conectividad conexión-8 y (b) conexión-4.

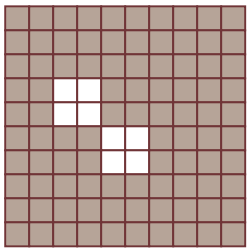


Figura 3.9: Problema al elegir la conectividad.

portante tomar en cuenta que esta determina fuertemente el resultado final del procesamiento puesto que puede dar origen a objetos nuevos en donde si se hubiera elegido otra conectividad no existirán. Para explicar lo anterior consideremos la figura 3.9. Como puede verse si se elige la conectividad conexión-8 la figura contenida en la imagen seria considerada como una sola pero si al contrario se elige la conectividad conexión-4 seria vista como 2 objetos diferentes.

La función bwlabel realiza un etiquetado de los componentes existentes en la imagen binaria, la cual puede ser considerada como una forma de averiguar cuantos elementos (considerando como elementos agrupaciones de unos bajo alguno de los dos criterios de conectividad) están presentes en la imagen. La función tiene el siguiente formato

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

Donde **ImagenR** es la imagen resultado que contiene los elementos etiquetados con el numero correspondiente al objeto, **ImagenS** es la imagen binaria que se desea encontrar el numero de objetos y **conectividad** puede ser 4 o 8 (correspondiendo al tipo de conectividad anteriormente explicado.

El efecto de esta función puede explicarse fácilmente si se analiza la imagen original y el resultado de aplicar la función `bwlabel`, como se muestra en la figura 3.10.

Supongamos que la imagen 3.10 (a) es la imagen binaria original (**ImagenR**) y 3.10 (b) es la imagen resultado (**ImagenS**) de ejecutar la función

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

La imagen resultado asigna a cada píxel perteneciente a un determinado objeto según su conectividad la etiqueta perteneciente al numero de objeto mientras que los píxeles en *cero* no tienen efecto en la operación, en la figura 3.10 esto corresponde a la codificación de colores mostrada en 3.10 (c).

La imagen resultado es del tipo double. Además debido a su contenido (los valores son muy pequeños) no puede ser desplegada por la función imshow. Una técnica útil para la visualización de este tipo de matrices es la de utilizar seudo color en forma de una imagen indexada. Una imagen indexada es una imagen

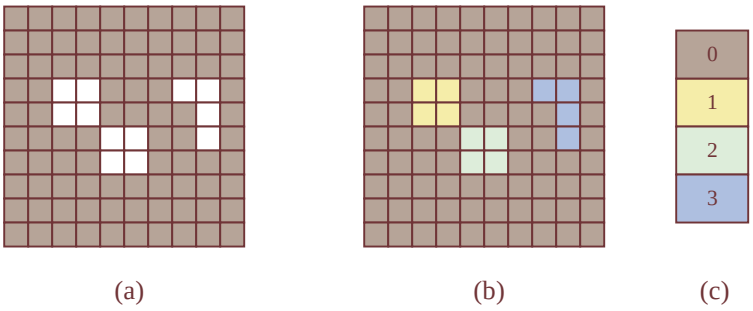


Figura 3.10: a) Imagen original binaria, b) imagen resultado de la operación bwlabel Considerando como conectividad conexión-4 y (c) la codificación de color.

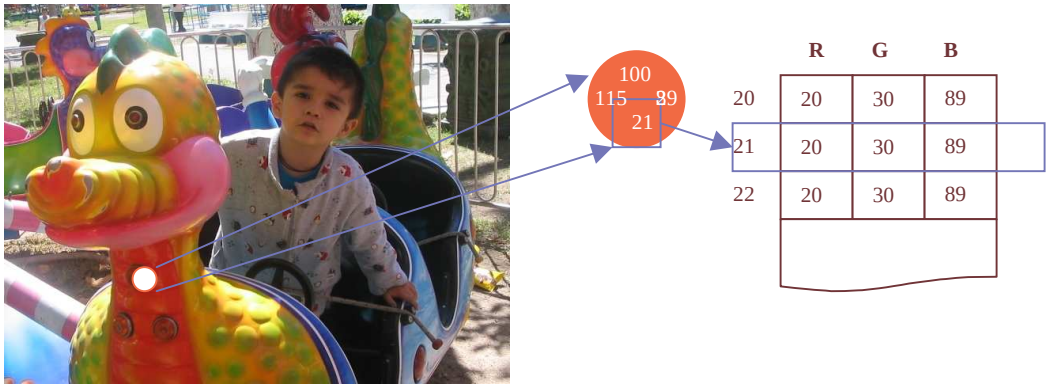
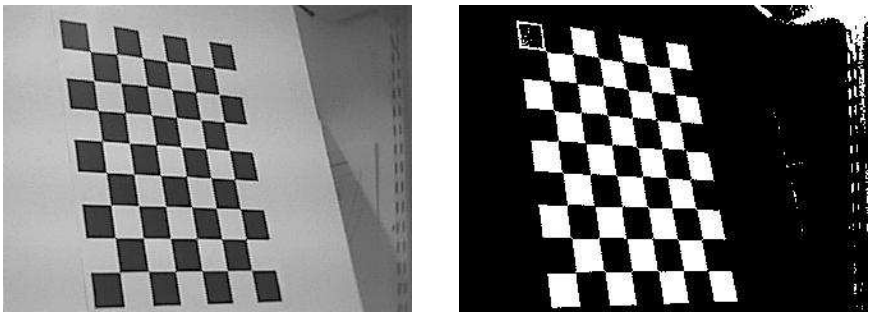


Figura 3.11: Ejemplo de imagen indexada.

que puede representar imágenes a color al igual que las RGB pero utiliza un formato diferente, en lugar de utilizar tres planos como lo realiza las imágenes RGB utiliza una matriz y una tabla, la matriz contiene en cada píxel un numero entero correspondiente al índice de la tabla, mientras que cada índice de la tabla corresponden 3 valores correspondiente a los planos RGB, con ello es posible reducir el tamaño de las imágenes al reducir el numero de diferentes colores. La figura 3.11 muestra un ejemplo de imagen indexada.

De esta forma cada objeto de la imagen resultado al aplicar la función bwlabel puede ser desplegado en un diferente color y ser identificado mas rápidamente. Es importante recalcar que el número de índices necesarios para formar la imagen es igual al número de etiquetas encontradas por la función *bwlabel* mas uno, ya que el fondo constituido de solo *ceros* también es un índice mas.

Consideremos un ejemplo para ilustrar la anterior técnica. Supongamos que tenemos la imagen a escala de grises representada por la figura 3.12 (a) y la



(a) (b)

Figura 3.12: a)Imagen a escala de grises e b) imagen binarizada.

binarizamos aplicando un umbral de 85 obteniendo así la imagen 3.12 (b).

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

Se aplica la función bwlabel para obtener los objetos contenidos en la imagen binarizada considerando la conectividad conexión-8

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

Se encuentra el numero de objetos contenidos en la imagen

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

Se genera la imagen indexada con 154 elementos

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

Dando como resultado la imagen representada en la figura 3.13.

3.6. Selección de objetos

En visión por computador resulta de especial utilidad de poder aislar objetos de una imagen binaria con un método rápido e interactivo. La función de matlab bwselect permite interactivamente seleccionar el objeto binario a segmentar con tan solo señalarlo en la ventana (previamente desplegada mediante la función *imshow*). El formato de la función es

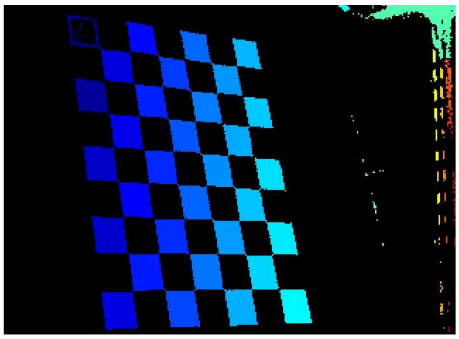
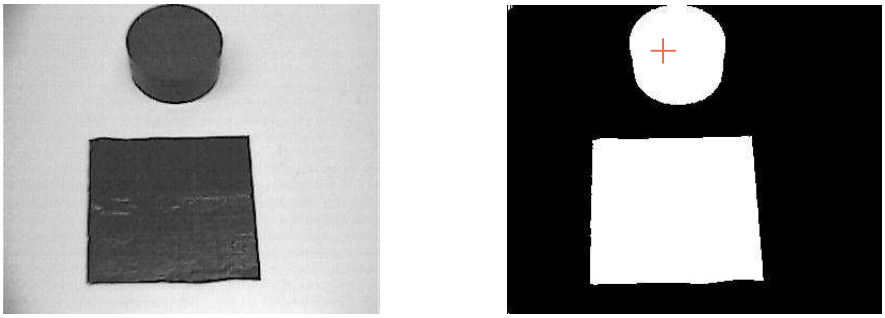


Figura 3.13: Resultado de el ejemplo de la función `bwlabel`.



(a) (b)

Figura 3.14: a) Imagen a escala de grises e b) imagen binarizada.

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

Donde **ImageR** es la imagen conteniendo al objeto seleccionado mientras que **c** representa el criterio de conectividad usado. Al igual que otras funciones iteractivas utilizadas en este documento es necesario seleccionar con el apuntador del ratón el objeto en la imagen binaria a aislar, pulsar el botón derecho y después la tecla enter. El siguiente ejemplo ilustra la utilización de la función. Consideremos que tenemos una imagen a escala de grises como la ejemplificada por la figura 3.14 (a) y la binarizamos con un umbral de 140 para obtener así la figura 3.14 (b)

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

Ahora si escribimos en línea de comandos

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

Figura 3.15: Representación de la variable **imagesegment.**

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

Cuadro 3.1: Mediciones realizadas por la función `imfeature`.

y seleccionamos el objeto 1 como lo muestra la figura 3.14 (b). De lo anterior resultaría como resultado la variable **imagesegment** representada en la figura 3.15.

3.7. Medición de características

En visión computacional es de particular interés encontrar mediciones de características tales como el área, centroide y otras de objetos previamente etiquetados o clasificados por la función bwlabel con el objetivo de identificar su posición en la imagen. Matlab dispone de la función imfeature para encontrar tales características. El formato de esta función es

```
Stats=imfeature(L,medición);
```

Donde Stats es un tipo de dato compuesto que contiene todas la medición indicada en la cadena de texto **medicion**, **medicion** es una cadena de texto que indica a la función cuales mediciones realizar sobre los objetos contenidos en la imagen el conjunto de mediciones posibles vienen sumarizados en la tabla 3.1.

Es importante notar que Stats es un tipo de dato que contiene la medición de todos y cada uno de los objetos contenidos en la imagen binaria, lo cual hace

Figura 3.16: Imagen utilizada para ejemplificar el uso de la función `imfeature.`

necesario que el tipo de dato sea indexado; es decir podemos controlar mediante un indice la medicion del objeto que deseamos. Por ejemplo si la imagen contiene 4 elementos, podemos acceder a la medición de cada uno de los diferentes objetos escribiendo:

```
stats(1).Medicion

stats(2).Medicion

stats(3).Medicion

stats(4).Medicion
```

Consideremos que tenemos la imagen previamente binarizada (contenida en la variable imagebinary) representada en la figura 3.16 y se intenta encontrar el centroide de ambas figuras contenidas en tal imagen.

Como primer paso utilizamos la función *bwlabel* y verificamos el numero de objetos:

```
>>imageR=bwlabel(imagebinary,8);
>>max(max(imageR))
ans=
2
```

Ahora se utiliza la funcion *imfeature* para encontrar el centroide de ambas figuras:

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

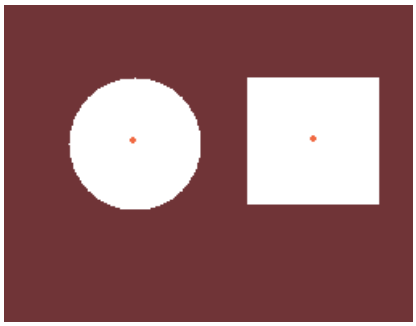


Figura 3.17: Identificación de los centroides.

```
>>s(2).Centroid
ans =
192.5000 85.0000
```

3.8. Funciones para la conversión de imágenes y formatos de color

El formato de representación de color ofrecido por las imágenes RGB resulta no apropiado para aplicaciones en las cuales el cambio de iluminación es problema. Otro tipo de formatos de color menos sensibles al cambio de iluminación han sido propuestos, tales como el modelo HSV. Matlab dispone de funciones especiales para realizar cambios entre modelos de color y para convertir imágenes de color a escala de grises; algunas de esas funciones seran tratadas en este apartado.

La función `rgb2gray` cambia una imagen en formato RGB a escala de grises, el formato de dicha función es:

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

Por su parte la función `rgb2hsv` cambia del modelo de color RGB al modelo HSV, esta función toma como entrada una imagen RGB compuesta de tres planos y devuelve la imagen convertida al modelo HSV compuesta a su vez de tres planos correspondientes al H, S y V. El formato de esta función es:

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

La conversión contraria la realiza la función `hsv2rgb`.

Para ejemplificar el uso de estas funciones considérese la imagen representada en la figura 3.18. La idea es cambiar dicha Imagen a un formato menos sensible

Figura 3.18: Imagen RGB.

Figura 3.19: Imagen HSV.

a los cambios de contendido de color como lo es el HSV y hacer evidente a la pelota de tal forma que pueda ser segmentada.

Escribiendo en línea de comandos obtenemos como resultado la imagen HSV que si la representamos considerando el modelo de planos RGB tendría el aspecto de la figura 3.19.

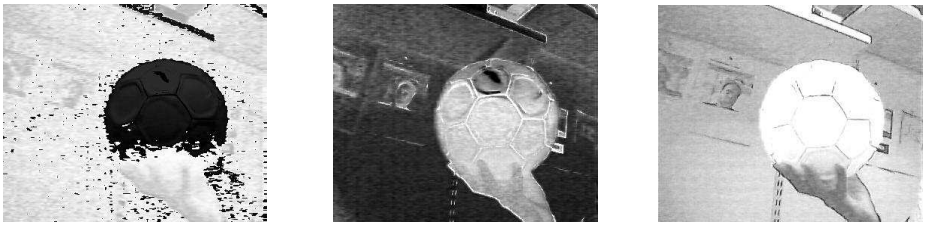
```
>>imageHSV=rgb2hsv(imageRGB);
```

Si dividimos la imagen en sus respectivos planos HSV, escribiendo en línea de comandos:

```
>>H=imageHSV(:,:,1);

>>S=imageHSV(:,:,2);

>>V=imageHSV(:,:,3);
```



(a) (b) (c)

Figura 3.20: Planos de la imagen a) H, b) S y c) V.

obtenemos como resultado las figuras 3.20 (a), 3.20 (b) y 3.20 (c).

De las anteriores imágenes resulta evidente que puede utilizarse la del plano S (saturación) para poder desempeñar una posible segmentación del objeto.

Capítulo 4

La herramienta vfm

Sin embargo un problema importante en las versiones anteriores a la Release 14 (en esta, superado con la incorporación del *toolbox de adquisición de imagenes*) es la falta de conexión entre una imagen tomada por un adquisidor de imagen (tal como una tarjeta adquisidora o bien simplemente una Web-cam) y el procesamiento realizado por matlab. Tener esta ventaja permite implementar los algoritmos de visión con las características reales de cámara o captador sin ningún esfuerzo adicional (como lo seria implementar un programa que grabe la imagen captada en un archivo con un formato especifico, para después utilizar los comandos normales del toolbox para abrir el archivo).

La herramienta *vfm* permite resolver este problema, *vfm* es un conjunto de librerías dinámicas que permiten acceder directamente a los controladores del dispositivo registrados por windows. De esta manera USB Web-cams de bajo costo pueden ser utilizadas así como otros dispositivos para captar una imagen y poderla utilizar en línea de comandos por matlab y sus toolbox.

La herramienta una vez instalada permite ser utilizada de una manera muy simple, solo hay que poner en línea de comandos :

```
>>vfm
```

A continuación aparecerá una ventana como la mostrada en la figura 4.1.

La herramienta cuenta con tres menús que controlan la captura. En el menú **Driver** se permite seleccionar una de varias fuentes de captación de video instaladas, en el menú **Configure** se puede configurar el formato y características propias del driver previamente seleccionado, por ultimo en el menú **Window** se permite controlar el flujo de la presentación de la imagen, normalmente se prefiere el modo de Preview para que la imagen se este mostrando en todo momento en la ventana.

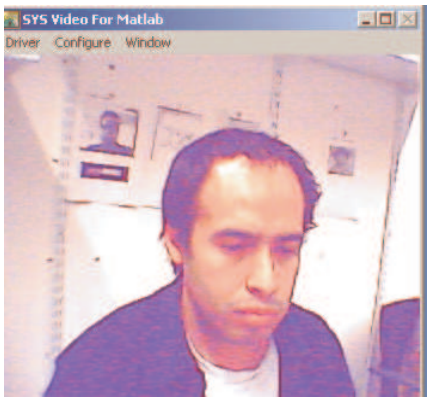


Figura 4.1: Ventana de la herramienta `vfm`.

4.1. Captura de la imagen en matlab

Las imágenes capturadas utilizando la herramienta vfm son entregadas al espacio de trabajo en formato RGB.

Para capturar las imágenes entregadas por los drivers en el espacio de trabajo de matlab se utiliza la función vfm que tiene el siguiente formato

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

Donde *ImageCaptured* es la variable que contiene la imagen RGB, y n es el numero de 'frames' a capturar, en dado caso de que sea mas de uno la variable de la imagen capturada se indexa para recuperar el 'frame' necesario.

De tal forma que si se quisiera capturar el 'frame' actual visto por la cámara se procedería a escribir en línea de comandos:

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

Obteniéndose como resultado el mostrado en la figura 4.2.

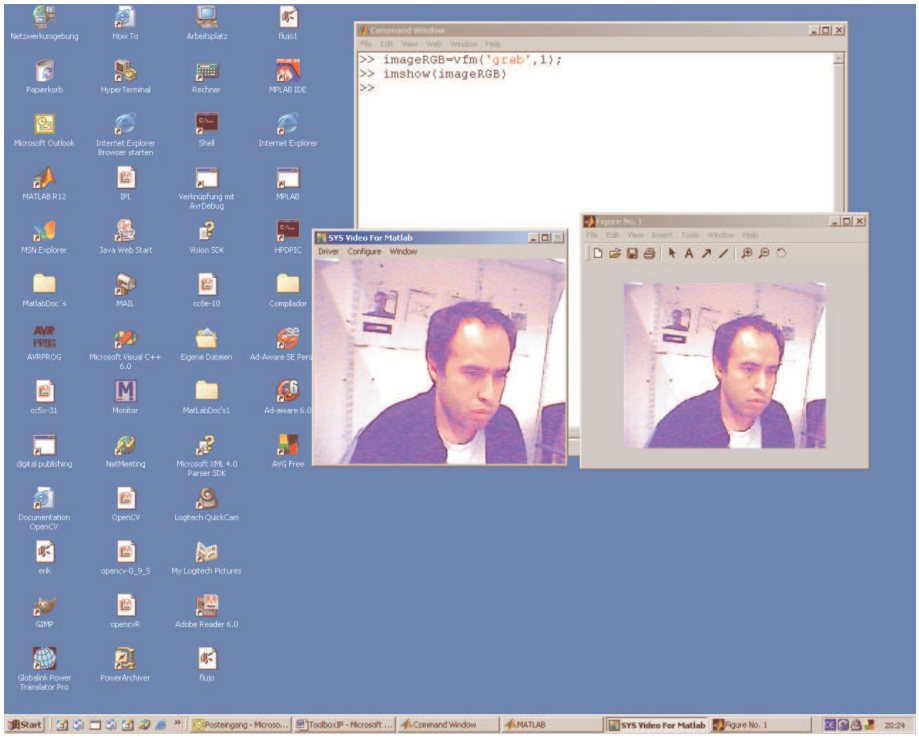


Figura 4.2: Proceso de captura de la herramienta `vfm`.