2013

Escuela Técnica Superior de Ingeniería de Telecomunicación

Proyecto Final de Carrera

Sistema Automatizado de Lectura Visual de Datos de Medida

Autor: ESTEBAN AVIVAR, Ángel Eugenio

Tutores: **RODRIGO PEÑARROCHA, Vicent Miquel**

NARANJO ORNEDO, Valery

ÍNDICE

1. INTRODUCCIÓN

La finalidad del presente proyecto es diseñar un sistema que pueda almacenar automáticamente los valores mostrados por un instrumento de medida que se encuentre en un laboratorio científico. Pues hoy en día siguen existiendo aparatos de medición que no incorporan ningún sistema de adquisición de datos de las medidas que efectúan. Lo único que muestran es, a través de un visualizador, el valor que están midiendo o generando.

Este valor solamente puede ser recogido por el usuario de forma manual, lo que conlleva el empleo de un tiempo innecesario y de una rutina no sistematizada que tiene que realizar el propio usuario.

Un ejemplo es el generador de tensión que se muestra en la imagen siguiente que consta de visualizadores de siete segmentos. Si se desea ir guardando la tensión que proporciona mientras un usuario la va modificando, la única forma que existe en la actualidad para llevarlo a cabo es realizarlo manualmente.

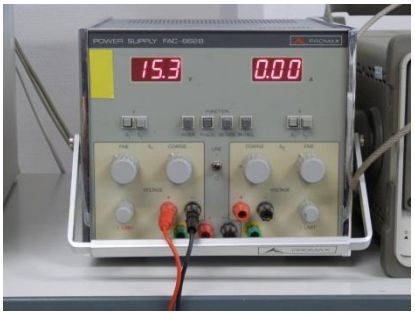


Imagen 1: Generador de tensión con visualizadores de siete segmentos

Otro ejemplo, que es objeto de este proyecto, es el de un multímetro o polímetro. Generalmente tienen un visualizador LCD que muestra el valor medido de forma digital, pero, aun teniendo un convertidor a digital, carece de un sistema de recogida o adquisición de la medida efectuada.

También existen aparatos cuya lectura es analógica, una aguja que rota sobre uno de sus extremos superpuesta a un fondo con líneas escaladas. Pero esto puede ser objetivo de un proyecto futuro.

De esta forma ha surgido la necesidad de crear un sistema que posibilite adquirir de forma automática las medidas para después procesarlas según el interés del usuario.

Ya que no se puede abrir la carcasa del multímetro y trabajar sobre su circuitería, se ha pensado en detectar cada uno de los dígitos usando imágenes y una vez hecho esto ir guardándolos de forma automática. La forma más sencilla y económica es usando una webcam para capturar imágenes que a continuación serán procesadas por un programa que devolverá el valor tal y como es dado por el multímetro.

Imagen 2: Disposición del sistema

Una vez solucionado el problema de la captura de la imagen, el siguiente requerimiento del sistema es si el procesado de las imágenes debe realizarse en tiempo real o a posteriori. Como el planteamiento inicial fue el de facilitar el tratamiento de los datos en el momento mismo de su toma, se ha decidido que este proyecto tiene que dar resultados en tiempo real.

Es por esto por lo que se ha llegado a la conclusión de desarrollar un software para tal fin desarrollado en el entorno Microsoft Visual Studio (versión 2010) empleando como herramienta la librería OpenCV (versión 2.3) y usando el lenguaje C++. Aunque más adelante se hablará de las propiedades de esta plataforma, sí cabe decir ahora que nos proporciona muchas funciones de tratamiento de imágenes ya optimizadas computacionalmente, lo que nos ayuda a centrarnos en qué queremos hacer o cuál es el siguiente paso a realizar, en vez de estar pensando cómo podemos implementar cierto método.

Además de todo esto, no podemos aventurarnos aún a comenzar a realizar el proyecto sin antes habernos planteado posibles problemas o contiendas que se tendrán que solventar:

a) El primer problema que surge en el planteamiento del proyecto es que la generalización del sistema sea válida con la diferente variedad de multímetros existentes, ya que los displays y la forma de los dígitos que se muestran son distintos para cada modelo (altura del dígito, anchura, grosor de la línea, color de la línea, color del display, forma del display,…). En la figura 3 se muestran diferentes multímetros con disponibilidad para la realización de este proyecto final de carrera. Una generalización más amplia se plantea como trabajo para futuros proyectos.

Figura 3: Multímetros con los que se ha trabajado

b) Otro problema que surge es que no se puede situar la webcam a cualquier distancia, es decir, existe una distancia máxima a la que se tiene que colocar debido, simplemente, a la limitación provocada por la resolución de las imágenes que no permite distinción entre píxeles.

c) Por otro lado la iluminación va a jugar un papel fundamental, desde la incidencia de la luz sobre el multímetro hasta su intensidad. Teniendo en cuenta esta problemática, los filtros y sistemas implementados en el proyecto trabajan de una forma adaptativa al tipo de iluminación que se encuentre, incluso llegando a modificarla para un mejor tratamiento de las imágenes.

d) Otro posible problema es la aparición vibraciones o movimientos no deseados de la webcam o del multímetro durante el proceso de captura de las imágenes. Considerando este inconveniente, se procede a realizar un seguimiento frame a frame de aquella región que delimita el visualizador del multímetro.

e) Además, el sistema deberá ser capaz de reconocer los dígitos con cierta inclinación debido a que, dependiendo de la ubicación y del espacio disponible, puede que tanto la webcam como el multímetro no estén alineados.

f) Algo que agradecería el usuario para un posterior tratamiento de los datos sería encontrar o ubicar el punto decimal y fijarlo en la adquisición.

Teniendo en cuenta todo esto se fijan unos objetivos concretos:

1. Crear un programa que en tiempo real sea capaz de leer imágenes de una webcam, procesarlas, segmentarlas para poder extraer los dígitos y reconocer qué dígitos son.
2. Obtener como salida un fichero con los datos adquiridos que resulte de fácil manejo al usuario y pueda ser exportado a cualquier otro programa (Excel, MATLAB,…).
3. Mostrar por pantalla los valores que se detectan en cada una de las imágenes y posibilitar al usuario que inicialice la adquisición cuando él lo crea oportuno.
4. Simplificar el sistema para facilitar futuras mejoras y/o ampliaciones, tales como posibles llamadas al sistema (para realizar accesos remotos a la ejecución del software) o lectura de ficheros.

Una vez establecidas las bases y los objetivos del proyecto ya se puede comenzar a realizar su diseño e implementación, sin olvidar que debe ser lo más transparente posible al usuario.

No obstante, las condiciones del entorno de trabajo necesarias se encuentran en un laboratorio donde las mediciones se realizan con un multímetro. Estas mediciones son capturadas por una webcam conectada a un ordenador personal en el que se ha instalado el software necesario y se desarrolla todo el código del presente proyecto. Una imagen de la ubicación de este entorno y toma de medidas es la siguiente:

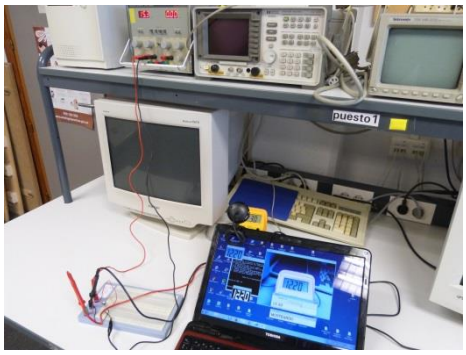


Imagen 4: Imagen del entorno de trabajo

2. ENTORNO DE TRABAJO

2.1. INTRODUCCIÓN

En este apartado se realiza una explicación de todos los medios al alcance para la consecución del proyecto. Estos medios se clasifican en 2 grupos:

- Plataforma software
- Medios físicos

El software utiliza como base el entorno de desarrollo integrado de *Microsoft Visual Studio 2010 Ultimate*. Sobre este entorno se trabaja con una librería que contiene funciones optimizadas para el manejo de imágenes llamada *OpenCV* (versión 2.3). El uso de esta herramienta obliga a manejar el lenguaje *C++* para la programación, que a su vez es soportado por Microsoft Visual Studio 2010 Ultimate.

Debido a la gran cantidad de información de libre consulta de todo este software no se detallarán todas sus características, tan sólo aquellas que se consideran de relevancia, y se relega a consultar la bibliografía propuesta en el último apartado si se desea profundizar en alguna cuestión en concreto.

En cuanto a los medios físicos, además de usar un *ordenador personal* para trabajar sobre el software, es necesaria una *webcam* para poder adquirir las imágenes a procesar. Y dichas imágenes son las propias de un *multímetro* que realiza medidas en el ambiente de trabajo de un *laboratorio*.

2.2. MICROSOFT VISUAL STUDIO 2010 ULTIMATE

Microsoft Visual Studio es un entorno de desarrollo integrado (IDE) para sistemas operativos Windows. Soporta varios lenguajes de programación y entornos de desarrollo web tales como:

- Visual C++
- Visual C#
- Visual J#

- Visual Basic
- .NET
- ASP.NET
- …

Uno de los logros de la versión de Visual Studio 2010 es la de incluir herramientas para desarrollar aplicaciones para Windows7.

Una de las características más destacables de esta versión es la de poder utilizar múltiples monitores y de desacoplar las ventanas de su sitio de trabajo y acoplarlas en otros sitios de la interfaz de trabajo.

Además, teniendo en cuenta la proliferación de las pantallas táctiles, también se pueden desarrollar aplicaciones para estos dispositivos.

A continuación se muestra una tabla comparativa con las distintas ediciones de la versión 2010, recordando que la edición que se usa en este proyecto es *Ultimate*:

Product	Extensions	Projects templates	MSDN integration	Debugging	Profiling	Static analysis	IntelliTrace	Unit test	Code coverage	Coded UI test	Test impact analysis	Load testing	Lab management	Architecture and modelling	Windows Phone development
Express	Yes	Limited	Essential	Yes	No	No	No	No	No	No	No	No	No	No	Yes
Professional	Yes	Yes	Full or Essentials	Yes	No	No	No	Yes	No	No	No	No	No	No	Yes
Premium	Yes	Yes	Full	Yes	Yes	Yes	No	Yes	Yes	Yes	Yes	No	No	Read-only	Yes
Ultimate	Yes	Yes	Full	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Test Professional	No	No	Full	No	No	No	No	No	No	No	Yes	No	Yes	No	No

Referencia: http://en.wikipedia.org/wiki/Microsoft_Visual_Studio

2.3. OPENCV

2.3.1. GENERALIDADES

OpenCV es una librería de uso libre originalmente desarrollada por *"Intel"*. Sus siglas provienen de los términos "Open Source Computer Vision Library", es decir, es una librería de tratamiento de imágenes destinada esencialmente a aplicaciones de visión por computador en tiempo real.

Esta librería se ha utilizado en infinidad de aplicaciones:

- Sistemas de reconocimiento facial
- Estimaciones de movimiento
- Reconocimiento de gestos
- Interacción persona-ordenador (HCI, Human-Computer Interaction)
- Identificación de objetos

- Segmentación y reconocimiento
- Estereopsis, visión estéreo: percepción de profundidad a partir de dos cámaras
- Estructura del movimiento (SFM, Structure from motion)
- Seguimiento del movimiento
- Realidad aumentada
- …

Esta disparidad de aplicaciones se debe a que su publicación se da bajo licencia BSD (Berkeley Software Distribution), que permite que sea usada libremente para propósitos comerciales y de investigación con las condiciones en ella expresadas.

Las ediciones más actuales de Open CV son multiplataforma, existiendo versiones para GNU/Linux, Mac OS X y Windows.

El desarrollo de esta librería pretendía proporcionar un entorno fácil de usar y de un alto grado de eficiencia. Esto se logró empleando su programación los lenguajes C y C++ optimizados, aprovechando las capacidades que proporcionan los procesadores multinúcleo. Además, OpenCV puede usar un conjunto de rutinas de bajo nivel determinadas para procesadores Intel.

2.3.2. TIPOS DE DATOS

Es fundamental conocer los distintos tipos de datos que nos proporciona OpenCV ya que son los elementos básicos sobre los que se trabaja y son parámetros de entrada a las funciones. Es por esto, por lo que es muy importante fijarse qué tipo de datos acepta la función a emplear y usarlos de forma adecuada.

Los distintos tipos de datos son:

1. CvArr

Este tipo de dato es un “metatype”, es decir, un tipo de dato ficticio que se utiliza de forma genérica a la hora de describir los parámetros de las funciones.

2. IplImage

Es el tipo de datos básico en OpenCV. Con este tipo de datos se representan las imágenes sean del tipo que sean: BGR, intensidad...

La ordenación y los campos de esta estructura es la siguiente:

```
typedef struct _IplImage {
                    int nSize;
                    int ID;
                    int nChannels;
                    int alphaChannel;
                    int depth;
                    char colorModel[4];
                    char channelSeq[4];
                    int dataOrder;
                    int origin;
                    int align;
                    int width;
                    int height;
                    struct _IplROI *roi;
                    struct _IplImage *maskROI;
                    void *imageId;
                    struct _IplTileInfo *tileInfo;
                    int imageSize;
                    char *imageData;
                    int widthStep;
                    int BorderMode[4];
                    int BorderConst[4];
                    char *imageDataOrigin;
} IplImage;
```

A continuación se describen los campos que han sido más usados:

- *width* y *height*: la anchura y la altura de la imagen expresadas en píxeles.
- *depth* contiene información sobre el tipo de valor de los píxeles. Los posibles valores del campo depth son los siguientes:

  IPL_DEPTH_8U: Enteros sin signo de 8 bits (unsigned char)
  IPL_DEPTH_8S: Enteros con signo de 8 bits (signed char o simplemente char)
  IPL_DEPTH_16S: Enteros de 16 bits con signo (short int)
  IPL_DEPTH_32S: Enteros con signo de 32 bits (int)
  IPL_DEPTH_32F: Números en coma flotante con precisión simple de 32 bits (float)

- *nChannels*: es el número de canales de la imagen. Las imágenes en escala de grises tienen un sólo canal, mientras que las de color tienen 3 o 4 canales.

3. CvMat

Es una de las estructuras más usadas para trabajar con imágenes.

La ordenación y los campos de esta estructura es la siguiente:

```
typedef struct CvMat{
                    int rows;
                    int cols;
                    CvMatType type;
```

```
                int step;
                union
                {
                        float* fl;
                        double* db;
                }data;
}CvMat
```

Además de almacenar los elementos como otra matriz cualquiera, posibilita el acceso a información adicional que puede resultar útil.

4. CvScalar

Es un vector de cuatro elementos, pero resulta muy útil para acceder a los píxeles de una imagen, sobre todo si es de color.

Su estructura es:

```
CvScalar
                double val[4];
```

Para acceder al campo de esta estructura tan solo es necesario añadir al nombre de la misma ".val". Por ejemplo:

```
CvScalar ejemp;
ejemp.val[1] = 5; // inicialización del segundo elemento de "ejemp" con el valor 2
```

5. CvPoint

CvPoint define las coordenadas de un punto con números enteros.

La estructura de CvPoint es:

```
typedef struct CvPoint{
                int x; /* coordenada x */
                int y; /* coordenada y */
}CvPoint;
```

6. CvSize

Se utiliza para definir las dimensiones de un rectángulo en píxeles:

```
typedef struct CvSize{
                int width; /* anchura del rectángulo (valor en píxeles)*/
                int height; /* altura del rectángulo (valor en píxeles)*/
}CvSize;
```

Se utiliza para definir las dimensiones de una imagen nueva.

Nota importante: *al compilar el código, el compilador acepta que en algunas funciones se introduzcan unos tipos de datos distintos a los establecidos. Pero al ejecutarlo aparecerá una excepción indicando un error en una posición concreta de la memoria sin informarnos en qué función del código resulta este problema. Para evitar este problema, hay que introducir el tipo de dato indicado en la definición de cada función.*

2.3.3. FUNCIONES Y CONSTRUCTORES

A continuación se muestran todas las funciones y constructores propios de OpenCV que se han empleado en la realización del proyecto.

a) `adaptiveThreshold( InputArray src, OutputArray dst, double maxValue, int adaptiveMethod, int thresholdType, int blockSize, double C )`

Esta función aplica umbralización adaptativa a un array.

Parámetros:

- *src*: imagen fuente de un solo canal de 8 bits.
- *dst*: imagen destino del mismo tamaño y tipo que src.
- *maxvalue*: valor distinto de cero asignado a los píxeles que cumple la condición a satisfacer. Ver a continuación para mayor compresión.
- *adaptiveMethod*: algoritmo de umbralización adaptativa a usar, ADAPTIVE_THRESH_MEAN_C o ADAPTIVE_THRESH_GAUSSIAN_C. Consultar los detalles a continuación.
- *thresholdType:* tipo de umbral a usar entre THRESH_BINARY y THRESH_BINARY_INV.
- *blockSize:* tamaño de vecindad entre píxeles que se utiliza para calcular un valor umbral para el píxel. Debe ser un número impar distinto de 1.
- *C:* constante que se resta de la media o de la media ponderada. Ver más detalles a continuación.

La función transforma una imagen en escala de grises en una imagen binaria de acuerdo con las fórmulas:

- THRESH_BINARY

$$dst(x,y) = \begin{cases} maxValue & si\ src(x,y) > T(x,y) \\ 0 & otros \end{cases}$$

- THRESH_BINARY_INV

$$dst(x,y) = \begin{cases} 0 & si\ src(x,y) > T(x,y) \\ maxValue & otros \end{cases}$$

Donde T(x, y) es un umbral calculado individualmente para cada píxel.

- Para el método de ADAPTIVE_THRESH_MEAN_C, el valor umbral *T(x,y)* es una media de los vecinos de *blockSize x blockSize* del píxel *(x, y)* menos *C*.
- Para el método de ADAPTIVE_THRESH_GAUSSIAN_C, el valor umbral *T(x,y)* es una suma ponderada (correlación cruzada con una ventana gaussiana) de los vecinos de *blockSize x blockSize* del píxel *(x, y)* menos *C*.

La función puede procesar la imagen en el lugar.

b) `approxPolyDP( InputArray curve, OutputArray approxCurve, double epsilon, bool closed )`

Aproxima una curva (o curvas) poligonal con la precisión especificada.

Parámetros:

- *curve:* vector de entrada de un punto 2D.
- *approxCurve:* resultado de la aproximación. El tipo debe coincidir con el tipo de la curva de entrada.
- *epsilon:* parámetro que especifica la exactitud aproximación. Indica la distancia máxima entre la curva original y su aproximación.
- *closed*: si es cierto, la curva aproximada está cerrada (el primer y el último vértice están conectados). De lo contrario, no está cerrada.

Realiza la aproximación poligonal mediante el algoritmo de Douglas-Peucker.

c) `Rect boundingRect( InputArray points )`

Calcula y devuelve el rectángulo mínimo para unos puntos dados.

Parámetro:

- *points:* conjunto de puntos 2D de entrada, almacenados en *std::vector* o *Mat*.

d) `(IplImage*) cvCloneImage( const IplImage* image )`

Hace la copia de una imagen, incluyendo la cabecera, los datos y la ROI.

e) `cvCopy( const CvArr* src, CvArr* dst, const CvArr* mask CV_DEFAULT(NULL) )`

Copia un array a otro.

Parámetros:

- *src:* el array fuente.
- *dst:* el array destino.

- *mask:* máscara de la operación, array de un solo canal de 8 bits. Especifica los elementos de la matriz destino que desea cambiar.

La función copia los elementos seleccionados de una matriz de entrada a una matriz de salida:

$$dst(I) = src(I) \quad si\ mask(I) \neq 0$$

f) `cvCreateCameraCapture( int index )`

Inicializa la captura de video desde una cámara.

Parámetro:

- *index*: índice identificador de la cámara a usar.

g) `cvMatchTemplate( const CvArr* image, const CvArr* templ, CvArr* result, int method )`

Compara una imagen como plantilla con respecto a otra imagen.

Parámetros:

- *image:* imagen donde se ejecuta la búsqueda. Debe ser de 8 bits o 32 bits de coma flotante.
- *templ:* plantilla de búsqueda. No debe ser mayor que *image* y tiene que ser del mismo tipo de datos.
- *result:* mapa de los resultados de la comparación. Debe ser de un solo canal de 32 bits de coma flotante. Si *image* es de tamaño (W x H) y *templ* (w x h), *result* será de tamaño (W-w+1) x (H-h+1).
- *method:* parámetro que especifica el método de comparación. Puede emplear 6 métodos distintos: la diferencia en cuadratura, la correlación cruzada, el coeficiente de correlación, y pueden ser normalizados o no.

h) `cvMinMaxLoc( const CvArr* arr, double* min_val, double* max_val, CvPoint* min_loc CV_DEFAULT(NULL), CvPoint* max_loc CV_DEFAULT(NULL), const CvArr* mask CV_DEFAULT(NULL) )`

Encuentra el mínimo y el máximo global de la matriz que se le pasa.

Parámetros:

- *arr:* array de entrada de un solo canal.
- *min_val:* puntero que devuelve el valor mínimo. Se utiliza *NULL* si no es necesario.
- *max_val:* puntero que devuelve el valor máximo. Se utiliza *NULL* si no es necesario.
- *min_loc:* puntero que devuelve la localización del mínimo. Se utiliza *NULL* si no es necesario.

- *max_loc:* puntero que devuelve la localización del máximo. Se utiliza *NULL* si no es necesario.
- *mask:* máscara opcional que se utiliza para seleccionar un sub-array.

i) `cvNamedWindow( const char* name, int flags CV_DEFAULT(CV_WINDOW_AUTOSIZE))`

Sirve para nombrar y etiquetar una ventana sobre la que se mostrará algún tipo de imagen.

Parámetros:

- *name*: nombre de la ventana que se utiliza cómo identificador.
- *flags*: banderas de la ventana. Para esta versión de OpenCV, la única bandera compatible es: CV_WINDOW_AUTOSIZE.

j) `cvNormalize( const CvArr* src, CvArr* dst, double a CV_DEFAULT(1.), double b CV_DEFAULT(0.), int norm_type CV_DEFAULT(CV_L2), const CvArr* mask CV_DEFAULT(NULL) )`

Normaliza la norma o rango de valores de una matriz.

Parámetros:

- *src:* array fuente.
- *dst:* array destino del mismo tamaño que *src*.
- *a:* valor de la norma para normalizar o límite inferior del rango en el caso de la normalización.
- *b:* límite superior del rango en el caso de la normalización.
- *norm_type:* tipo de normalización.
- *mask:* máscara de la operación. Es opcional.

La función *cvNormalize* escala y desplaza los elementos de la matriz de entrada tal que:

$$\|dst\|_{Lp} = a$$

(donde p = Inf, 1 o 2) cuando norm_Type = NORM_INF, NORM_L1, o NORM_L2, respectivamente; o de manera que

$$\min dst(I) = a\,, \max dst(I) = b$$

cuando norm_Type = NORM_MINMAX (sólo para matrices densas).

La máscara opcional especifica una sub-matriz para ser normalizado.

k) `(IplImage*) cvQueryFrame( CvCapture* capture )`

Graba y devuelve un frame de la cámara.

l) `CvRect  cvRect( int x, int y, int width, int height )`

Crea una estructura que guarda las coordenadas de un rectángulo.

Los parámetros que se le pasan son la coordenada *x* e *y* del punto superior izquierdo, la *anchura* y la *altura*.

m) `cvRectangle( CvArr* img, CvPoint pt1, CvPoint pt2, CvScalar color, int thickness CV_DEFAULT(1), int line_type CV_DEFAULT(8), int shift CV_DEFAULT(0))`

Dibuja un rectángulo sobre una imagen.

Parámetros:

- *img*: imagen.
- *pt1*: vértice del rectángulo.
- *pt2*: vértice del rectángulo opuesto al pt1.
- *color*: color del rectángulo o brillo (escala de grises).
- *thickness*: grosor de las líneas que componen el rectángulo. Los valores negativos, como CV_FILLED, indican que la función tiene que dibujar un rectángulo relleno.
- *line_type*: tipo de la línea.
- *shift*: número de bits fraccionarios en los puntos de las coordenadas.

n) `cvResetImageROI( IplImage* image )`

Libera la estructura de la ROI.

o) `cvSetImageROI( IplImage* image, CvRect rect )`

Establece una Región de Interés de una imagen para un rectángulo dado.

p) `cvSetMouseCallback( const char* window_name, CvMouseCallback on_mouse, void* param CV_DEFAULT(NULL))`

Establece el control del ratón para la ventana especificada.

Parámetros:

- *window_name*: nombre de la ventana.
- *on_mouse*: función donde se entrega el control del ratón.
- *param*: parámetro opcional que se pasa a la función indicada por on_mouse.

q) `cvShowImage( const char* name, const CvArr* image )`

Complementaria a cvNamedWindow, muestra la imagen deseada.

Parámetros:

- *name*: nombre de la ventana que se utiliza cómo identificador.
- *image*: imagen que se desea mostrar.

r) `cvtColor( InputArray src, OutputArray dst, int code, int dstCn=0 )`

Esta función convierte el espacio de color de una imagen a otro.

Parámetros:

- *src*: imagen de la entrada.
- *dst*: imagen de salida del mismo tamaño y profundidad que src.
- *code*: código de conversión del espacio de color.
- *dstCn*: número de canales en la imagen de destino. Si el parámetro es 0 el número de canales se deriva automáticamente de src y code.

Esta función puede hacer las siguientes transformaciones:

- RGB ⟷GRAY
- RGB ⟷CIE XYZ
- RGB ⟷YCrCb JPEG (or YCC)
- RGB ⟷HSV
- RGB ⟷HLS
- RGB ⟷CIE L*a*b*
- RGB ⟷CIE L*u*v*
- Bayer ⟶RGB

s) `cvWaitKey(int delay CV_DEFAULT(0))`

Esta función hace que el programa se pare hasta que se pulse una tecla.

Parámetros:

- *delay*: retardo en milisegundos que espera a que se pulse una tecla. Por defecto se toma el valor 0.

t) `equalizeHist( InputArray src, OutputArray dst )`

Ecualiza el histograma de una imagen en escala de grises.

Parámetros:

- *src*: imagen fuente de un solo canal de 8 bits.
- *dst*: imagen destino del mismo tamaño y tipo que *src*.

La función ecualiza el histograma de la imagen de entrada usando el siguiente algoritmo:

1. Calcula el histograma H de *src*.
2. Normaliza el histograma de modo que la suma de las barras del histograma es 255.
3. Calcula la integral del histograma:

$$H'_i = \sum_{0 \le j < i} H(j)$$

4. Transforma la imagen usando H':

$$dst(x,y) = H'(src(x,y))$$

El algoritmo normaliza el brillo y aumenta el contraste de la imagen.

u) `findContours( InputOutputArray image, OutputArrayOfArrays contours, OutputArray hierarchy, int mode, int method, Point offset=Point())`

Encuentra los contornos en una imagen binaria.

Parámetros:

- *image*: imagen fuente de un solo canal de 8 bits.
- *contours*: contornos detectados. Cada contorno se almacena como un vector de puntos.
- *hierarchy*: vector de salida opcional, que contiene información acerca de la topología de la imagen. Se tiene tantos elementos como número de contornos.
- *mode*: modo de recuperación del contorno:

  CV_RETR_EXTERNAL recupera sólo los contornos exteriores. Se establece hierarchy [i] [2] = hierarchy [i] [3] = -1 para todos los contornos.
  CV_RETR_LIST recupera todos los contornos sin establecer relaciones jerárquicas.
  CV_RETR_CCOMP recupera todos los contornos y los organiza en una jerarquía de dos niveles. En el nivel superior, hay límites externos de los componentes. En el segundo nivel los límites de los agujeros. Si hay otro contorno interior en un agujero de un componente conectado se pone en el nivel superior.
  CV_RETR_TREE recupera todos los contornos y reconstruye una jerarquía completa de los contornos anidados.

- *method*: método de aproximación del contorno:

  CV_CHAIN_APPROX_NONE almacena absolutamente todos los puntos del contorno. Es decir, cualquiera de los 2 puntos posteriores (x1, y1) y (x2, y2) del contorno serán vecinos ya sea horizontal, vertical o diagonal, es decir, max (abs (x1-x2), ABS (Y2-Y1)) == 1.
  CV_CHAIN_APPROX_SIMPLE comprime los segmentos horizontales, verticales y diagonales, y deja sólo los puntos finales. Por ejemplo, un contorno rectangular arriba-derecha se codifica con 4 puntos.
  CV_CHAIN_APPROX_TC89_L1, V_CHAIN_APPROX_TC89_KCOS aplica el algoritmo de aproximación cadena de Teh-Chin.

Esta función recupera los contornos de la imagen binaria utilizando el algoritmo desarrollado por S. Suzuki y K. Abe en la publicación "Topological Structural Analysis of Digitized Binary Images by Border Following" en 1985.

v) `imshow( const string& winname, InputArray mat )`

Muestra una imagen en la ventana especificada.

Parámetros:

- *winname:* nombre de la ventana.
- *mat:* imagen que se muestra.

w) `imwrite( const string& filename, InputArray img, const vector <int>&params = vector<int>())`

Guarda una imagen en un archivo especificado.

Parámetros:

- *filename: nombre del archivo.*
- *img:* imagen que se desea guardar.
- *params:* formato específico para guardar los parámetros codificados en forma de pares paramId_1, paramValue_1, paramId_2, paramValue_2, ... Se soportan los siguientes formatos:

  Para **JPEG (CV_IMWRITE_JPEG_QUALITY)** puede usarse una calidad de 0 a 100 (el más alto es el mejor). El valor por defecto es 95.
  Para **PNG (CV_IMWRITE_PNG_COMPRESSION)** el nivel de compresión puede ser de 0 a 9. Un valor más alto significa un tamaño más pequeño y mayor tiempo de compresión. El valor predeterminado es 3.
  Para **PPM, PGM, o PBM (CV_IMWRITE_PXM_BINARY)** pueden ser una bandera de formato binario, 0 o 1. El valor predeterminado es 1.

x)
```
Scalar mean(InputArray src, InputArray mask=noArray())
```

Calcula la media de los elementos de una matriz.

Parámetros:

- *src: matriz de entrada que debe tener de 1 a 4 canales*
- *mask: máscara. Es opcional.*

y)
```
medianBlur( InputArray src, OutputArray dst, int ksize )
```

Dicha función realiza un filtrado de mediana utilizando un tamaño de apertura que se le pasa como parámetro de entrada.

Parámetros:

- *src*: array de entrada con 1,3 o 4 canales.
- *dst*: array destino resultante del mismo tamaño que src.
- *ksize*: tamaño de la apertura. Tiene que ser un valor impar y mayor que 1.

La función suaviza una imagen usando un filtro de mediana con una apertura de tamaño (*ksize* x *ksize*). Cada canal de una imagen multicanal es procesada independientemente.

z)
```
morphologyEx( InputArray src, OutputArray dst, int op, InputArray kernel,
Point anchor=Point(-1,-1), int iterations=1, int borderType =
BORDER_CONSTANT, const Scalar& borderValue=morphologyDefaultBorderValue())
```

Realiza transformaciones morfológicas avanzadas.

Parámetros:

- *src:* Fuente de la imagen. El número de canales puede ser arbitrario. La profundidad debe ser: CV_8U, CV_16U, CV_16S, CV_32F o CV_64F.
- *dst:* imagen destino del mismo tamaño y tipo que el original.
- *op:* tipo de una operación morfológica. Puede ser uno de los siguientes:

  MORPH_OPEN - una operación de apertura

  MORPH_CLOSE - una operación de cierre

  MORPH_GRADIENT - un gradiente morfológico

  MORPH_TOPHAT - "sombrero de copa"

  MORPH_BLACKHAT - "sombrero negro"
- *iteraciones:* número de veces que la erosión y la dilatación se aplican.
- *borderType:* método de extrapolación de los píxeles.
- *borderValue:* valor del borde en caso de un borde permanente.

Esta función puede realizar transformaciones morfológicas avanzadas utilizando la erosión y la dilatación como operaciones básicas:

Apertura

$$dst = open(src, element) = dilate(erode(src, element))$$

Cierre

$$dst = close(src, element) = erode(dilate(src, element))$$

Gradiente morfológico

$$dst = morph_{grad(src,element)} = dilate(src, element) - erode(src, element)$$

"Sombrero de copa"

$$dst = tophat(src, element) = src - open(src, element)$$

"Sombrero negro"

$$dst = blackhat(src, element) = close(src, element) - src$$

En caso de imágenes multicanal, cada canal es procesado independientemente.

```
aa) resize( InputArray src, OutputArray dst, Size dsize, double fx=0, double
    fy=0, int interpolation=INTER_LINEAR )
```

Redimensiona una imagen.

Parámetros:

- *src*: imagen fuente.
- *dst*: imagen destino.
- *dsize:* tamaño de la imagen de salida. Si es igual a *0* se calcula como:

$$dst = Size(round(fx * src.cols), round(fy * src.rows))$$

- *fx:* factor de escala a lo largo del eje horizontal. Cuando es igual a *0* se calcula como:

$$(double)dsize.width/src.cols$$

- *fy:* factor de escala a lo largo del eje vertical. Cuando es igual a *0* se calcula como:
- *interpolation:* método de interpolación. Tipos:

  INTER_NEAREST - interpolación de vecino más cercano.

  INTER_LINEAR - interpolación bilineal (usado por defecto).

  INTER_AREA - remuestreo usando la relación píxel-área. Puede ser un buen método para diezmar la imagen, ya que da resultados moire'-free. Pero cuando la imagen se amplía es similar al método **INTER_NEAREST.**

  INTER_CUBIC - una interpolación bicúbica sobre píxeles vecinos a 4x4.

  INTER_LANCZOS4 - una interpolación Lanczos sobre píxeles vecinos a 8x8.

bb) `VideoCapture(int device)`

Es un constructor de VideoCapture. Establece el dispositivo que realizará la captura de vídeo.

Parámetro:

- *device:* identificación del dispositivo de captura de vídeo abierto (es decir, el índice de la cámara). Si hay una sola cámara conectada, simplemente se pasa *0*.

cc) `int waitKey(int delay=0)`

Realiza lo mismo que *cvWaitKey* pero, a diferencia de éste, devuelve el valor de un entero indicando el valor ASCII correspondiente a la tecla que ha sido pulsada.

2.4. C++

El lenguaje C++ es una evolución de C que empezó a desarrollar en 1980 Bjarne Stroustrup. Esta evolución suponía mantener la sencillez y flexibilidad de C pero incorporando mecanismos para el tratamiento de objetos.

Con posterioridad se añadieron otras facilidades de programación genérica que, junto a la programación estructurada y a la programación orientada a objetos, dio lugar a definir C++ como un lenguaje de programación multiparadigma.

Su versatilidad, su potencia y su capacidad de generalización es lo que ha hecho que este lenguaje sea tan popular entre los programadores profesionales. Es más, cabe recordar que la librería OpenCV fue desarrollada en base a este lenguaje.

Además, es tan generalista este lenguaje que existe infinidad de documentación de libre consulta.

2.5. CONDICIONES FÍSICAS DEL MEDIO

Aunque la esencia del proyecto está orientada al desarrollo software se necesitan de unos utensilios o herramientas físicas cuyas características limitan en una mayor o menor grado el trabajo a realizar.

No hay que olvidar que se parte de la captura de imágenes en un medio físico. Estas imágenes serán capturadas por una *webcam* en el entorno de un *laboratorio*. Dichas imágenes serán las de un *multímetro* que estará tomando medidas en tiempo real sobre algún tipo de circuitería. A continuación, estas imágenes son procesadas por un *ordenador personal* originando el resultado final que es la creación de un fichero donde se almacenan los datos adquiridos por el multímetro.

Las características y las condiciones en las que trabajan estas herramientas son las siguientes:

Ordenador Personal

Se trata de un ordenador portátil cuyo procesador es el siguiente: Intel® Core(TM) i5-2410M CPU @ 2.30 GHz.

El sistema operativo sobre el que se trabaja es Windows 7 (sistema operativo de 64 bits).

Además consta de una RAM de 4 GB.

Sobre este sistema operativo está instalado Microsoft Visual Studio 2010 Ultimate.

Webcam

La webcam con la que se trabaja proporciona unas imágenes con una resolución de 640x480.

Multímetro

El multímetro con el que se han capturado las imágenes es: *Agilent 3801A*

Imagen 5: Multímetro Agilent 3801A

Imagen 6: Multímetro, ordenador y webcam

Laboratorio

En la toma de medidas y demás pruebas, se ha trabajado en uno de los laboratorios de la Escuela Técnico Superior de Ingenieros de Telecomunicación de la Universidad Politécnica de Valencia.

La iluminación de dicho laboratorio es suficiente y válida para capturar las imágenes. También, en él se encuentran otras herramientas para poder tomar medidas como son generadores de tensión y componentes electrónicos sobre los que poder hacer dichas medidas.

3. MÉTODO

3.1. INTRODUCCIÓN

En este apartado se tratará la estructura que se ha llevado a cabo en la programación del sistema así como el porqué de su implementación.

En primer lugar se mostrará el diagrama general realizando una descripción cualitativa de cada uno de los bloques.

A continuación, cada uno de esos bloques será analizado de una forma mucho más detallada, pero de tal forma que cualquier lector que carezca de los conocimientos específicos sobre el tratamiento digital de la imagen pueda comprenderlo.

No obstante, se acompañará de ejemplos que visualicen mejor lo que se está describiendo.

3.2 DIAGRAMA GENERAL Y DESCRIPCIÓN DEL MÉTODO

3.2.1 DIAGRAMA GENERAL

El esquema general, mostrado como diagrama de bloques, al que se ha llegado a la conclusión del proyecto es el siguiente:

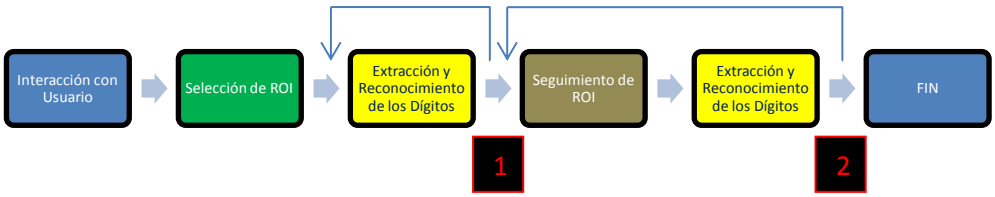


Imagen 7: ESQUEMA GENERAL

3.2.2 INTERACCIÓN CON EL USUARIO

Este primer bloque no tiene ningún uso funcional en el sistema que se ha desarrollado. Su única función es la de indicar al usuario que se va a comenzar a ejecutar la aplicación, así como una pequeña explicación de funcionamiento.

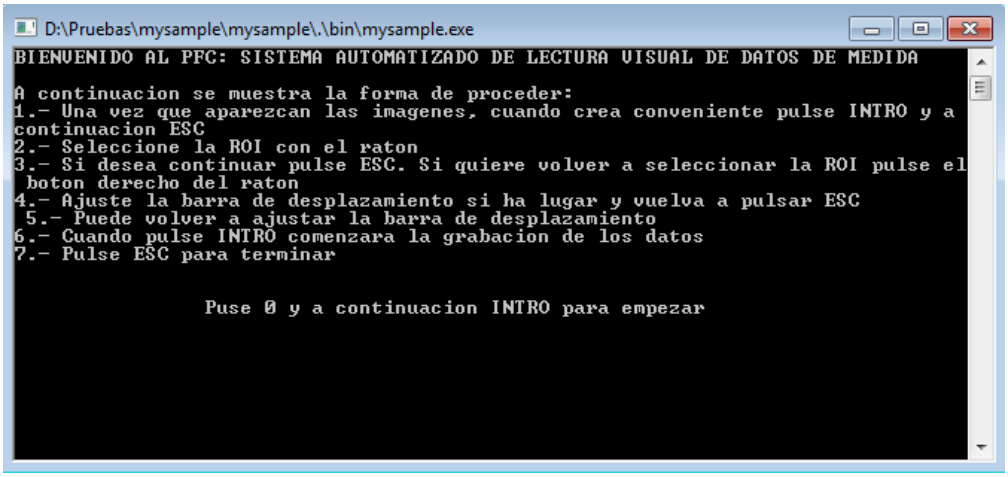


Imagen 8: Comienzo de la ejecución

3.2.3 SELECCIÓN DE ROI

Se trata de un bloque fundamental del sistema, y no sólo por ser del que se parte, sino porque en todo momento se hará referencia a lo establecido aquí, tanto en el seguimiento frame a frame como en el tratamiento digital de la imagen.

Antes que nada hay que comentar y explicar a qué nos referimos cuando decimos ROI. ROI, acrónimo de Region Of Interest (región de interés), es la parte de la imagen en la que nos centraremos en el tratamiento y haremos un seguimiento en cada una de las capturas.

Es importante decir que OpenCV considera la ROI como un rectángulo cuya base es paralela a la horizontal, y no como un polígono arbitrario de cuatro lados. Esto es importante resaltarlo porque no se puede elegir la ROI haciendo clic en los puntos que queremos que sean los vértices o elegirla de una forma similar, ya que es prácticamente imposible cumplir con la condición antes comentada.

Para seleccionar la ROI deseada se usa el ratón. Se hace clic con el botón izquierdo en un punto que se considera como vértice superior izquierdo y, sin soltarlo, arrastraremos hasta el punto que deseamos como vértice inferior derecho.

La ROI debe contener, al menos, los dígitos que se quieren reconocer. Además, cuanto más ajustada a los dígitos esté la ROI más eficaz será el tratamiento de la imagen; es decir, mejor seguimiento en cada frame y mejor tratamiento de la imagen.

No obstante, si una vez elegida la ROI no nos satisface, podemos volver a seleccionarla con un simple clic con el botón derecho del ratón.

Imagen 9: Selección de la ROI

Imagen 10: ROI escogida

3.2.4 EXTRACCIÓN Y RECONOCIMIENTO DE LOS DÍGITOS

Es importante recordar que aquí sólo se trabaja con la ROI.

Este bloque, a su vez, se puede dividir en otros 3 bloques o secciones secuenciales. Tales son:

1. Preprocesado.
2. Procesado y Segmentación.
3. Clasificador.

Generalmente, el preprocesado es necesario cuando se desea realizar un tratamiento digital de la imagen, es decir, se adecua la imagen para poder trabajar con ella.

En la sección de **preprocesado** se prepara la ROI para que más tarde se puedan extraer los dígitos y procesarlos de forma independiente. En nuestro caso pasamos la ROI de RGB a escala de grises, realizamos un filtrado de media, aumentamos el contraste y realizamos una umbralización adaptativa, donde la intención es que los dígitos sean blancos (valor de 255) y todo lo demás negro (valor de 0).

La segmentación es el proceso en el que se identifican objetos dentro de una imagen. En nuestro caso queremos discriminar los dígitos del resto que aparece en la ROI, aunque aún no podemos hacerlo ya que existe ruido en exceso en la imagen binaria que tenemos.

Antes de la segmentación tenemos que realizar un **procesado** para intentar dejar los dígitos como únicos elementos que aparecen en blanco. Este proceso consiste en realizar "Top-Hat", una apertura y un cierre (como se ha indicado anteriormente, se describirá con más detalle en el apartado 3.3.).

A continuación, se buscan los contornos de los grupos de píxeles contiguos que se encuentran en blanco y se encuadran con un rectángulo. Estos rectángulos son ordenados de izquierda a derecha para poder evaluarlos de forma secuencial.

Una vez realizado esto, sólo se trabajará con los rectángulos que superen un límite, relacionado con un tanto por cien de la altura de la ROI. Este tanto por cien es lo que puede ser ajustado por el usuario descartándose todos aquellos rectángulos menores al tanto por cien indicado por éste. Es decir, es aquí cuando se diferencia y se extraen los dígitos, siendo esto la **segmentación**.

Una vez llegado a este punto ya se puede obtener una característica propia de la imagen que se tiene del dígito y compararla con una batería de muestras predefinidas, obteniendo así el valor del dígito asociado. Esto es función del **clasificador**.

La característica propia de la imagen se consigue dividiendo la imagen binaria de cada rectángulo obtenido en el paso anterior en 9 bloques, es decir, una matriz de 3x3 bloques, y se calcula la media de cada uno de los bloques, obteniéndose 9 valores comprendidos entre 0 y 255.

Estos valores se comparan con los de una matriz de tamaño 10x9 construida con el mismo algoritmo pero teniendo algunas consideraciones:

a. Cada una de las filas corresponde con las medias de los bloques de cada dígito; la fila 1 con el dígito 0, la fila 2 con el dígito 1,… la fila 10 con el dígito 9 (véase en el apartado 3.3.3.3 CLASIFICADOR las matrices globales *mediafinal* y *mediafinal2*).
b. Las imágenes de entrenamiento de las que se ha partido, para construir dicha matriz, corresponden con capturas realizadas por nosotros mismos y clasificadas manualmente. Los dígitos a comparar se parecerán más a los capturados por nosotros que a otros tomados arbitrariamente.
c. Para aumentar la precisión del clasificador y no depender de la habilidad de escoger el dígito correcto para calcular su característica, este proceso se realizó con 10 imágenes de cada dígito y se calculó la media entre todas.

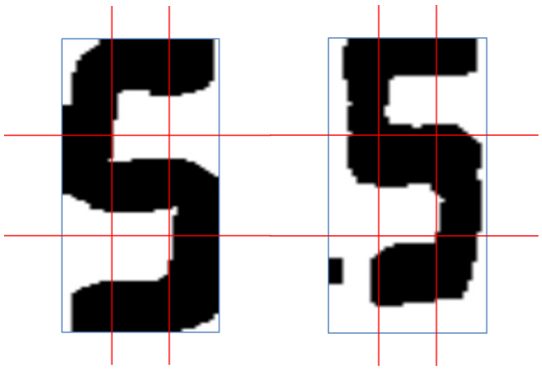


Imagen 11: Imágenes distintas que corresponden al mismo dígito dividido en bloque

Para saber qué número asignamos al dígito de la imagen a tratar, calculamos la distancia euclídea entre la matriz 3x3 obtenida para dicho dígito y cada una de las filas de la matriz predefinida. La fila asociada a la menor distancia corresponderá al resultado devuelto por el clasificador.

Problemas en la clasificación

No obstante, una vez terminada una primera versión del proyecto, se constataron dos problemas que había que solventar:

1. El dígito "1" era muy difícil de analizar debido a que, después de cada uno de los procesados de la imagen que se hacían, aparecía con una forma distinta y se puede decir que hasta amorfa.

2. El segundo dígito empezando por la izquierda que se visualiza en el multímetro daba problemas de correspondencia ya que el punto decimal tendía a fusionarse con él.

El primer problema se ha solventado teniendo en cuenta las dimensiones que tiene la imagen del dígito. Después de haber pasado todos los "filtros" y llegar al clasificador, se considerará como un "1" aquél cuya relación altura/anchura sea mayor que 4. Todos los demás seguirán el proceso explicado anteriormente.

Y para resolver el segundo se ha construido otra matriz 10x9. Esta nueva matriz tiene en cuenta que los bloques situados en la parte derecha de la imagen del dígito tendrán una media distinta. No obstante, para saber qué digito hay que evaluar así, se toma como referencia el vértice inferior derecho del rectángulo asociado y se establece que esté ubicado alrededor de una posición central de la ROI.

Imagen 12: Forma de detectar cuál es el segundo dígito

Imagen 13: Dígito sin punto

Imagen 14: Dígito con punto

Este bloque se repite en dos momentos durante el proceso. La primera vez que aparece es en la fase de calibración del sistema, la cual se lleva a cabo para visualizar los valores obtenidos y comprobar que se corresponden a los esperados, al igual que también sirve para poder aumentar o disminuir el umbral y descartar los rectángulos que no contienen los dígitos, es decir, en la fase en que el usuario establece los diferentes parámetros.

La segunda vez es para ya mostrar los datos y poder adquirirlos en tiempo real. En esta fase, ya la de procesado, también se podrá modificar el valor umbral del que se ha hablado anteriormente mediante una barra de desplazamiento, como ya ocurriera en la fase de validación.

3.2.5 SEGUIMIENTO DE LA ROI

La importancia de este bloque reside en que se debe saber en todo momento la ubicación de la ROI. No se pueden evaluar las imágenes de los dígitos si no se encuentran dónde se esperan obtener.

Es por ello por lo que, frame a frame, hay que determinar dónde se encuentra la ROI, ya que ha podido cambiar de posición relativa con respecto a la imagen capturada por la webcam debido a vibraciones o movimientos, tanto de la webcam como del multímetro.

Para poder realizarlo, se usa una función propia de OpenCV, *cvMatchTemplate*, que, tomando como parámetros de entrada la imagen de la captura actual de la webcam y la ROI inicial, obtiene la diferencia en cuadratura. Así pues, considerando las siguientes condiciones:

- I: es la imagen actual de tamaño $W \times H$

- *T*: es la ROI inicial de tamaño *w x h*

Entonces, la diferencia en cuadratura dada es:

- $R(x,y) = \sum_{x',y'}\left(T(x',y') - I(x+x',y+y')\right)^2$
  donde $x' = 0...w-1, y' = 0... h-1$

Siendo *R* del tamaño $(W - w + 1) \times (H - h + 1)$.

A continuación, obteniendo sus mínimos y máximos globales se puede obtener el rectángulo en la ubicación de la nueva ROI.

Imagen 15: Captura actual

Imagen 16: ROI

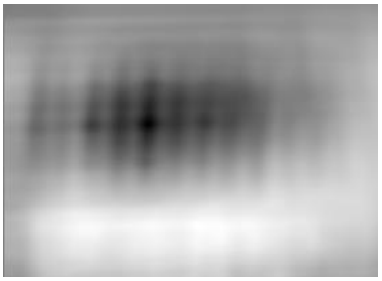


Imagen 17: Diferencia en cuadratura

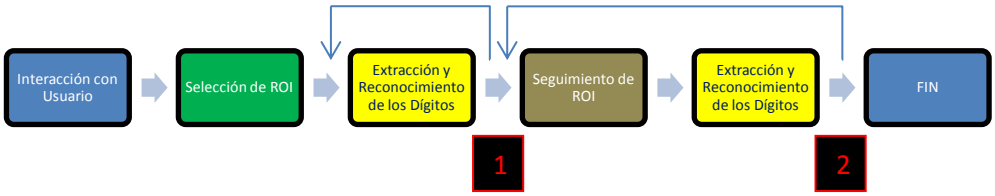
Imagen 18: Imagen actual con ROI actual

Imagen 19: ROI actual

De esta forma, tan sólo hay que preocuparse del procesado digital de esta parte de la imagen capturada, simplificando el procedimiento y los costes computacionales.

3.2.6 PUNTOS DE CONTROL

Retomando el diagrama general:



Cabe destacar los puntos 1 y 2 que aparecen en el esquema. Es en estos puntos donde el usuario indica que se debe salir del bucle (representado por la flecha retrospectiva) y continuar con el siguiente bloque.

Además, en el punto 1 el usuario tiene la opción de modificar un umbral para descartar partes de la imagen que se considerarán como ruido.

En el punto 2, al igual que en el 1, dicho umbral se puede modificar y, cuando el usuario lo desee, se puede comenzar a almacenar los valores obtenidos en el proceso en un fichero con extensión *.txt*.

3.3 DESCRIPCIÓN DE LA APLICACIÓN

En este apartado se explicará de una forma más extensa cada uno de los bloques mostrados en el apartado 3.2. Para ello se hará uso del código empleado para la consecución del proyecto.

Además, para cada una de las funciones propias de OpenCV habrá una explicación de su uso aunque exista una referencia al apartado correspondiente del capítulo 2, donde se enuncian todas las características y se detallan sus parámetros de entrada y de salida.

3.3.1 INTERACCIÓN CON EL USUARIO E INICIALIZACIÓN DE LA CAPTURA

Inicialmente se muestra por el monitor del ordenador un mensaje de bienvenida al usuario y una explicación sencilla de cómo ha de actuar para ejecutar el programa. Para ello se usan las premisas básicas de C++ para mostrar por pantalla.

```
int elec=-1;

      while (elec!=0){
            std::cout<<"BIENVENIDO AL PFC: SISTEMA AUTOMATIZADO DE LECTURA
VISUAL DE DATOS DE MEDIDA\n\n";
            std::cout<<"A continuacion se muestra la forma de proceder:\n";
            std::cout<<"1.- Una vez que aparezcan las imagenes, cuando crea
conveniente pulse INTRO y a continuacion ESC\n";
            std::cout<<"2.- Seleccione la ROI con el raton\n";
            std::cout<<"3.- Si desea continuar pulse ESC. Si quiere volver a
seleccionar la ROI pulse el boton derecho del raton\n";
            std::cout<<"4.- Ajuste la barra de desplazamiento si ha lugar y
vuelva a pulsar ESC\n";
            std::cout<<"5.- Puede volver a ajustar la barra de
desplazamiento\n";
            std::cout<<"6.- Cuando pulse INTRO comenzara la grabacion de los
datos\n";
            std::cout<<"7.- Pulse ESC para terminar\n\n\n";

            std::cout<<"\t\tPuse 0 y a continuacion INTRO para empezar";
            std::cin>>elec;
            }
```

Además, en este momento inicial es cuando se inicializa la captura de imágenes de la webcam y se van recogiendo en una variable del tipo `IplImage`. Las funciones que proporciona OpenCV para este menester son:

- `VideoCapture` (Véase 2.3.3.bb.): es un constructor de VideoCapture. Establece el dispositivo que realizará la captura de vídeo.
- `cvCreateCameraCapture` (Véase 2.3.3..): inicializa la captura de video desde una cámara.
- `cvQueryFrame` (Véase 2.3.3.k.): graba y devuelve un frame de la cámara.

Por consiguiente, el código empleado para esa inicialización es el siguiente:

```
cv::VideoCapture cap(1);
CvCapture* capture;
capture=cvCreateCameraCapture(1);
IplImage* bgr_frame=cvQueryFrame(capture);//Leemos el primer frame y se almacena
en el buffer bgr_frame
```

3.3.2 SELECCIÓN DE ROI

Como se ha comentado anteriormente, este bloque se encarga de definir una zona de la imagen capturada para, posteriormente, hacer un seguimiento de la misma y poder tratarla digitalmente. Esta zona contendrá los dígitos del display del multímetro considerándola nuestra Región de Interés (ROI).

Funcionalmente, lo que se pretende es hacer un clic con el botón izquierdo del ratón en la imagen capturada en el píxel que se desea como vértice superior izquierdo de la ROI. A continuación, sin soltar el botón, se arrastra el puntero hasta llegar al punto que se considerará el vértice inferior derecho de la ROI y se suelta.

Si la elección de la ROI satisface las necesidades para continuar con la ejecución, tan sólo será necesario pulsar la tecla *ESC*. Pero si por el contrario se considera que la ROI no ha sido seleccionada, con un simple clic con el botón derecho del ratón se volverá a la situación inicial.

En cuanto a la inicialización de las variables que vamos a emplear cabe resaltar la variable `drag`. Esta variable se considera como un *flag*, indicando si el botón izquierdo está pulsado (1) o no (0).

```
IplImage* img0, * img1;
CvPoint  point;
int  drag = 0;
cv::Mat Frame1;
```

La inicialización de img0 será con la imagen capturada por la webcam:

```
img0=bgr_frame;
```

A continuación se emplearán funciones propias de OpenCV que permiten la visualización de imágenes y el uso de eventos del ratón sobre las mismas:

- `cvNamedWindow` (Véase 2.3.3.i.): sirve para nombrar y etiquetar una ventana sobre la que se mostrará algún tipo de imagen.
- `cvShowImage` (Véase 2.3.3.q.): complementaria a la función anterior, muestra la imagen deseada.
- `cvSetMouseCallback` (Véase 2.3.3.p.): esta función asigna llamadas a eventos del ratón.

  El primer parámetro de entrada es el nombre de la ventada sobre la que se quiere actuar. El segundo es el nombre de la función en la que se encuentra

el código a realizar para dichos eventos. Y el tercero es el parámetro que se desea pasar a tal función.

El código empleado determinar la ROI es el siguiente:

```
…

std::string nombredepantalla("ROI");
cv::namedWindow("ROI",1);
cvSetMouseCallback("ROI", mouseHandler, NULL); //llamada a eventos del ratón
cvShowImage("ROI", img1);
cvNamedWindow("ROI",1);
cvShowImage("ROI",img1);
cvShowImage("ROI", bgr_frame);    //se muestra la ROI seleccionada
cvWaitKey(0);              //hasta que no se pulsa una tecla no se continúa
Frame1=img1;

…
```

Para cada uno de los eventos y situaciones que se pueden dar para la elección de la ROI ejecutamos distintos comandos.

En el primero de los casos, cuando se pulsa el botón izquierdo del ratón y anteriormente no lo estaba, se guarda en la variable *point* la posición de la imagen sobre la que se encuentra el puntero del ratón y se activa el flag *drag*.

Nos encontramos en otro de los casos cuando se mueve el ratón manteniendo pulsado el botón izquierdo del ratón. Sobre una imagen copiada de img0 se va dibujando un rectángulo conforme se va desplazando el ratón.

Y al soltar el botón izquierdo mostramos la región seleccionada haciendo uso de la función `cvSetImageROI`.

Y como último caso que nos podemos encontrar es cuando deseamos reiniciar el proceso de selección de la ROI pulsando, para ello, el botón derecho del ratón.

Para comprender mejor este código hay que saber que se usan dos estructuras básicas de OpenCV como son `cvPoint` y `cvRect`. La primera de ellas, `cvPoint`, es para conocer las coordenadas y la segunda, `cvRect`, para fijar un rectángulo pasándole como parámetros las coordenadas de inicio y final.

Las funciones de OpenCV que se usan para crear esta función son:

- `cvCloneImage` (Véase 2.3.3.d.): hace la copia de una imagen, incluyendo la cabecera, los datos y la ROI.

- `cvSetImageROI` (Véase 2.3.3.o.): establece una Región de Interés de una imagen para un rectángulo dado.
- `cvRectangle` (Véase 2.3.3.m.): dibuja un rectángulo sobre una imagen.

Y la función empleada para el evento del ratón es:

```
        void mouseHandler(int event, int x, int y,int flags, void* param)
        {

            /* usuario pulsa botón izquierdo del ratón */
            if (event == CV_EVENT_LBUTTONDOWN && !drag)      //Pinchamos
            {
                point = cvPoint(x, y);
                drag  = 1;
            }

            /* usuario arrastra el ratón */
            if (event == CV_EVENT_MOUSEMOVE && drag)  //Sin soltar movemos el
ratón
            {
               img1 = cvCloneImage(img0);
              cvRectangle(img1,point,cvPoint(x, y), CV_RGB(255,255, 255),1,8,0);
              cvShowImage("ROI", img1);
            }

            /* usuario deja de pulsar el botón izquierdo del ratón */
            if (event == CV_EVENT_LBUTTONUP && drag)  // Soltamos el botón
            {
               img1 = cvCloneImage(img0);
               region=cvRect(point.x,point.y,x - point.x,y - point.y);
                cvSetImageROI(img1,region);

                cvShowImage("ROI", img1);
                drag = 0;
            }

            /* usuario pulsa botón derecho del ratón: se resetea */
            if (event == CV_EVENT_RBUTTONUP)
            {
                cvShowImage("ROI", img0);
                drag = 0;
            }
        }
```

3.3.3 EXTRACCIÓN Y RECONOCIMIENTO DE LOS DÍGITOS

Como se comentó con anterioridad, el tratamiento digital de la imagen se realiza bajo dos circunstancias distintas a lo largo del proyecto.

La primera de ellas, la fase de validación, se corresponde cuando, tras seleccionar la ROI deseada, se comprueban los valores que serían adquiridos y se calibra, mediante

una barra de desplazamiento, para descartar los segmentos no deseados modificando un límite relacionado con un tanto por cien de la altura de la ROI (*Véase apartado 4.5*).

En la segunda ya se muestran los valores tal y como se van a adquirir teniendo la posibilidad de ajustar el calibrado anterior.

Para determinar los valores de los dígitos mostrados por el multímetro, una vez que ya se ha seleccionado la ROI, se sigue el siguiente esquema:

3.3.3.1 Preprocesado

En esta fase se pretende llevar a cabo una serie de acciones para adecuar la imagen de la ROI antes de procesarla para poder extraer cada uno de los dígitos.

En nuestro caso, como el objetivo de este conjunto de bloques es conseguir que todo píxel que no forme parte del dígito sea de valor 0 (negro) y el propio dígito tenga el valor 255 (blanco), lo primero que hay que hacer es convertir la imagen en color de la ROI (*Imagen 20.a.*) en otra que se encuentre en escala de grises. Para ello usamos la función *cvtColor*:

- `cvtColor` (Véase 2.3.3.r.): con esta función conseguimos convertir una imagen en color a otra en escala de grises.

  Para ello, como tercer parámetro introducimos la constante `CV_BGR2GRAY`, que indica la realización de un paso de tres canales (R, G y B) a un único canal, la luminancia:

$$Y = 0{,}299 * R + 0{,}587 * G + 0{,}11 * B$$

En la *imagen 20.b*. podemos observar el resultado de una imagen después de realizar una conversión a escala de grises:

Y como es normal en un altísimo porcentaje de sistemas de comunicaciones, tratamientos de señales o de imágenes, es deseable realizar un filtrado intentando eliminar todo el ruido posible que haya sido introducido.

En este caso, el filtro que se ha decidido usar es el de mediana, un filtro muy usado en el tratamiento digital de la señal, el cual realiza un filtrado no lineal eliminando ruido, pero respetando más los bordes que un filtro paso-bajo lineal. Es por esto por lo que usamos la función *medianBlur* de OpenCV:

- `medianBlur` (Véase 2.3.3.y.): dicha función realiza un filtrado de mediana utilizando un tamaño de ventana que se le pasa como parámetro de entrada.

No obstante, hay que indicar el tamaño de la ventana a usar. Pues de una manera empírica se llegó a la conclusión que el mejor tamaño a usar es 5, un mayor tamaño realizaría un "emborronado" de la imagen y un menor tamaño apenas eliminaría ruido (*Imagen 20.c.*).

Una vez conseguido esto se desea binarizar para conseguir que los dígitos, con un nivel muy alto de gris, sean blancos y todo lo demás negro. Pero debido a la baja calidad de la imagen condicionada por la resolución de la webcam y por el filtrado anterior se percibe una imagen algo "difuminada". Es por ello por lo que se desea aumentar el contraste de la imagen y así establecer después un umbral mucho más claro.

Un método muy empleado para ese aumento de contraste es la ecualización del histograma. Este método intenta conseguir que el histograma de la imagen de salida sea constante, y esto se consigue cambiando el nivel de gris de cada píxel por el valor del histograma acumulado para el nivel original:

$$H'_i = \sum_{0 \le j < i} H(j)$$

Desde un punto de vista más visual, la ecualización del histograma realiza una expansión del histograma, proporcionando un aumento del margen dinámico de la imagen (*Imagen 20.d.*).

No obstante, OpenCV ya nos proporciona una función que ya implementa la ecualización del histograma:

- `equalizeHist` (Véase 2.3.3.t.): esta función es muy sencilla de aplicar, tan sólo hay que introducir como parámetro de entrada la imagen que deseamos aumentar su contraste.

Una vez llegada esta situación, ya se puede binarizar empleando para ello umbralización adaptativa. Esto consiste, no utilizando un umbral fijo para toda la imagen, sino en utilizar uno adaptativo para contrarrestar el efecto de la iluminación no uniforme que se aprecia en la *imagen 20.d*. Lo importante para poder abordar esto es saber qué método adaptativo y qué tipo de umbralización se va a usar.

El método que se usa es el del cálculo de la media de los píxeles vecinos de un bloque: *`CV_ADAPTIVE_THRESH_MEAN_C`*, pone como umbral la media de los píxeles de un bloque alrededor del que se quiere umbralizar.

Para nuestro caso, también se podría haber usado el del cálculo gaussiano siendo el valor umbral la suma ponderada (es decir, la correlación cruzada con una ventana gaussiana) de los píxeles de un bloque alrededor del que se desea umbralizar, pero no es necesario ya que hemos ecualizado el histograma y el resultado sería muy similar entre ambos.

En cuanto al tipo hay que recordar que se desea que los dígitos se encuentren en blanco cuando hasta ahora son de un nivel bajo de gris, es por ello que se usa: `CV_THRESH_BINARY_INV`. Además, también introducimos el tipo: `CV_THRESH_OTSU`, que proporciona unos resultados muy buenos cuando existe un foco de luz dirigido a la zona de captura o aparecen sombras. Este último tipo obtiene un umbral óptimo teniendo en cuenta la varianza entre píxeles.

El tamaño de bloque óptimo se ha obtenido empíricamente utilizando un conjunto de vídeos de entrenamiento en diferentes situaciones de iluminación, distancia de la cámara, tipo de multímetro, etc., obteniéndose un valor de 11. El resultado tras la realizar esta umbralización se puede ver en la *imagen 20.e*.

Por consiguiente, la función a emplear teniendo en cuenta todo esto es:

- `adaptiveThreshold` (Véase 2.3.3.a.): esta es la función de OpenCV que proporciona la umbralización adaptativa teniendo en cuenta el método y el tipo a emplear.

Así pues, de una forma compacta, el código empleado es el que se muestra a continuación:

```
…

cv::cvtColor(Frame1,Frame1, CV_BGR2GRAY);
cv::Mat Filtr;
medianBlur(Frame1,Filtr,5);
cv::equalizeHist(Filtr,Filtr);
int constantedeadaptive;
constantedeadaptive=2;  // Es la desviacion de la media
cv::adaptiveThreshold(Filtr,Filtr,255,CV_ADAPTIVE_THRESH_MEAN_C,CV_THRESH_BINARY_
INV || CV_THRESH_OTSU ,11,constantedeadaptive);

…
```

Imagen 20.a.: Imagen RGB

Imagen 20.b.: Imagen en Escala de Grises

Imagen 20.c.: Imagen tras Realizar Filtrado de Mediana

Imagen 20.d.: Imagen tras Realizar la Ecualización del Histograma

Imagen 20.e.: Imagen tras Realizar la Umbralización Adaptativa

3.3.3.2 Procesado y Segmentación

PROCESADO

Ahora se tiene una imagen en blanco y negro sobre la que ya se puede trabajar para poder extraer los dígitos, pero se advierte que existen ciertos aspectos no deseables en la imagen que pueden dar algún tipo de problema. Entre ellos se puede destacar que en ocasiones los números aparecen "partidos", que aparecen los bordes del visualizador, que los dígitos aparecen algo "deformados" o que aún aparece ruido o se ha creado ruido nuevo.

Es por ello por lo que hay que tratar esta imagen binaria, intentando paliar todos los problemas comentados, hasta conformarla de forma que la extracción de los dígitos sea eficiente.

En primera instancia se desea realizar la transformada *Top-hat*, pero empleando la función *morphologyEx* de OpenCV directamente se da una situación rígida en la que no se obtiene el resultado deseado y apenas se tiene libertad de para realizar la morfología deseada, debido en gran parte a que sólo se permite el uso de un tamaño de elemento estructurante.

Conceptualmente, *Top-hat* es un tipo de filtro no lineal que se utiliza para obtener de una imagen elementos y detalles de un determinado tamaño que viene dado por el tamaño del elemento estructurante o ventana. La definición de *Top-hat* viene dada por:

$$dst = tophat(src, element) = src - open(src, element)$$

Donde *open* o apertura es un filtro que elimina los objetos claros más pequeño que el tamaño del elemento estructurante y viene dado por:

$$open = dilate(erode(src, element))$$

Siendo *erode* y *dilate*, la erosión y la dilatación de la imagen respectivamente con el elemento estructurante. La erosión consiste en el cálculo del mínimo de los píxeles dentro de una ventana y la dilatación (operación dual) consiste en el cálculo del máximo.

Pero para esta situación se va a operar de forma muy distinta aunque llegando al objetivo del proceso de Top-hat.

Primero se realiza una apertura (*open*) para suavizar y recomponer los dígitos, al igual que se elimina bastante ruido. De forma análoga a cómo se ha realizado anteriormente para determinar los tamaños de bloques, el tamaño apropiado del elemento estructurante se obtiene empíricamente con la base de entrenamiento, resultando de 6x6.

A continuación se realiza una erosión utilizando un elemento de tamaño 50x1 para intentar eliminar todos los píxeles blancos exceptuando los que se corresponden con los bordes horizontales propios del visualizador.

Por último, para conseguir el efecto de Top-hat se realiza una resta entre ambos resultados. Hay que decir que, aunque la matriz que guarda este resultado se ha denominado "`tophat`", por definición, no es el resultado de tal transformada, ya que lo que se ha obtenido es el residuo de una apertura más una erosión con tamaños de elementos estructurantes diferentes.

En cuanto al código empleado se plasma de la siguiente manera:

```
…

cv::Mat opening;
element=cv::getStructuringElement(0,cvSize(6,6),cvPoint(0,0));
cv::morphologyEx(Filtr, opening,2,element,cv::Point(-1,-1),1,0,0);
cv::Mat elemental;
cv::Mat eroding;
```

```
elemental=cv::getStructuringElement(0,cvSize(50,1),cvPoint(0,0));
cv::morphologyEx(bin,eroding,0,elemental,cv::Point(-1,-1),1,0,0);
cv::Mat tophat;
tophat = opening - eroding;

…
```

Imagen 21: Imagen tras Realizar el Efecto Top-Hat

Después de encontrarnos en esta situación, se desea "erosionar", básicamente, los "restos" o líneas verticales que han quedado.

Una vez hecho esto, se desea recomponer todo aquello de los dígitos que se había "destruido" utilizando la función *morphologyEx* con los parámetros adecuados:

- `morphologyEx` (Véase 2.3.3.z.): realiza ciertas transformaciones morfológicas como son la apertura, el cerraje, la erosión, la dilatación,…

De una forma más específica, el código empleado es:

```
…

cv::Mat ero;
cv::Mat elemental2;
elemental2=cv::getStructuringElement(0,cvSize(3,5),cvPoint(0,0));
cv::morphologyEx(tophat,ero,0,elemental2,cv::Point(-1,-1),1,0,0);
       //El parámetro para usar una transformación u otra es el tercero

cv::Mat dil;
cv::Mat elemental3;
elemental3=cv::getStructuringElement(0,cvSize(1,11),cvPoint(0,0));
cv::morphologyEx(ero,dil,1,elemental3,cv::Point(-1,-1),1,0,0);

…
```

Imagen 22: Imagen tras Realizar Erode-Dilate

Como se puede comprobar, tan sólo se está trabajando con dos funciones propias de OpenCV que ha llevado a un resultado más que admisible. Y, aunque parezca trivial, la búsqueda de la combinación entre todos los tipos de operaciones que proporciona morphologyEx es lo que ha llevado, en gran medida, al éxito de este proyecto.

A partir de ahora existen mecanismos y métodos, más o menos visuales, que son los que permiten la consecución del proyecto, pero no se podrían poner en práctica si lo que se ha realizado hasta ahora en el tratamiento digital de la imagen no hubiera dado un resultado más que aceptable.

SEGMENTACIÓN

Una de las herramientas útiles para el análisis de la forma y la detección de objetos y el reconocimiento es la búsqueda de los contornos. Mediante la función *findContours* de OpenCV se podrá obtener un vector de puntos que almacena cada uno de los contornos encontrados:

- `findContours` (Véase 2.3.3.u.): esta función recupera los contornos de la imagen binaria utilizando el algoritmo desarrollado por S. Suzuki y K. Abe en la publicación bibliográfica *VIII*. Cada contorno se almacena como un vector de puntos.

La intención es la de detectar los contornos de los dígitos y poder extraer el área donde se encuentran y así poder evaluarlos de forma independiente.

Pero aún falta realizar un paso intermedio, ya que el paso de un punto a otro del contorno puede llegar a ser muy abrupto aumentando el número de operaciones a realizar y pudiendo proporcionar una mala aproximación.

Para ello se realiza una aproximación poligonal con una precisión especificada a partir de los puntos del contorno. Esto se consigue empleando la función *approxPolyDP*:

- `approxPolyDP` (Véase 2.3.3.b.): realiza la aproximación poligonal mediante el algoritmo de Douglas-Peucker.

Pues bien, como las plantillas empleadas para obtener las muestras predefinidas para el clasificador son imágenes binarias con forma rectangular, se desea buscar el rectángulo mínimo que envuelve cada uno de los polígonos aproximados anteriormente. OpenCV tiene una función específica para este caso y es *boundingRect*:

- `boundingRect` (Véase 2.3.3.c.): calcula y devuelve el rectángulo mínimo para unos puntos dados.

```
…

std::vector<std::vector<cv::Point> > contours;
std::vector<cv::Vec4i> hierarchy;
cv::Mat copia=dil;

cv::findContours(copia,contours,hierarchy,CV_RETR_LIST,
CV_CHAIN_APPROX_SIMPLE,cvPoint(0,0));

//Aproximar los contornos a polígonos y delimitar con rectángulos
std::vector<std::vector<cv::Point> > contours_poly( contours.size() );
std::vector<cv::Rect> boundRect( contours.size() );


for( int i = 0; i < contours.size(); i++ ){
 approxPolyDP( cv::Mat(contours[i]), contours_poly[i], 3, true ); //Aproximación
       boundRect[i] = boundingRect( cv::Mat(contours_poly[i]) ); //Delimitar con
                                                                 // rectángulos
                    }

…
```

No obstante, cuando se vaya a extraer cada uno de los dígitos se querrá hacer de forma ordenada, es decir, se irán extrayendo de izquierda a derecha de forma correlativa. Pero los rectángulos almacenados en el vector *boundRect* no están ordenados, porque la función *boundingRect* comienza a ejecutarse en la esquina superior izquierda y avanza hacia la inferior derecha pudiendo encontrarse unos contornos antes que otros, y esto hay que solventarlo. La manera más sencilla, rápida y con un coste computacional muy bajo es empleando el "método de la burbuja", donde, para cada iteración, se establece que el rectángulo cuya *coordenada x* sea la menor irá ocupando la primera posición:

```
int polo,j;
std::vector<cv::Rect> aux(1);
for(polo=0;polo<contours.size()-1;polo++){
	for(j=polo+1;j<contours.size();j++){
		if(boundRect[polo].x>boundRect[j].x){
			aux[0]=boundRect[polo];
			boundRect[polo]=boundRect[j];
			boundRect[j]=aux[0];
		}
	}
}
```

Una vez llegado a este punto, donde todos los elementos están delimitados por un rectángulo que los "envuelve", tan sólo queda decidir cuáles de esos elementos son los que se corresponden con los dígitos en cuestión.

Es fácil observar que los dígitos son mucho más grandes que el resto de elementos y hay que recordar que se escogió una ROI que se intentaba ajustar al tamaño del visualizador. Pues bien, el criterio que se ha seguido es el de discriminar todo aquel elemento cuyo rectángulo asociado tenga una altura menor a un tanto por cien a la altura de la ROI.

Inicialmente, ese tanto por cien está establecido en el 40%, pero se puede ajustar mediante la barra de desplazamiento que tiene asociada por referencia la variable "umbral"; dicha barra se encuentra en la ventana donde se visualiza el valor a capturar. El uso de esta barra de desplazamiento se desarrollará en el siguiente capítulo: Funcionalidades adicionales.

Después de esta selección, los dígitos son detectados de forma individual y secuencial para poder pasar a la extracción de características y poder clasificarlos.

Esta detección final se realiza con la ayuda de la función *cvSetImageROI* ya explicada con anterioridad.

Imagen 23: Imagen con los segmentos a evaluar

Este discernimiento se expresa de la siguiente manera:

```
…

int umbral=40;
int altura;
altura=Frame1.rows;


for( int i = 0; i< contours.size(); i++ ){
float umb=umbral;
       float porcent;
porcent=umb/100;

if((boundRect[i].height)>(altura*porcent)){


              cv::imwrite("NumeroGuardado.tiff",copia);
              IplImage* in=cvLoadImage("NumeroGuardado.tiff");
              CvPoint puntoini= boundRect[i].tl();
              cvSetImageROI(in,
cvRect(puntoini.x, puntoini.y,boundRect[i].width,boundRect[i].height));
              IplImage *img = cvCreateImage(cvGetSize(in),in->depth,in-
>nChannels);
              cvCopy(in, img, NULL);

              cvResetImageROI(in);

…
```

Como se viene realizando, si existe necesidad de tener más información sobre las funciones empleadas en esta parte del código, se emplaza a leer el apartado correspondiente:

- `imwrite` (Véase 2.3.3.w.): guarda una imagen con el nombre y la extensión deseada.
- `cvSetImageROI` (Véase 2.3.3.o.): establece una Región de Interés de una imagen para un rectángulo dado.
- `cvCopy` (Véase 2.3.3.e.): copia una imagen sobre otra.
- `cvResetImageROI` (Véase 2.3.3.n.): libera la estructura de la ROI.

3.3.3.3 Clasificador

Llegado este momento, en el que ya se ha extraído el dígito, solamente falta saber a qué número corresponde para, a continuación, tratarlo como un simple dato y poder visualizarlo y capturarlo de forma que quede vinculado al valor mostrado por el visualizador.

Lo que se pretende es realizar una extracción de sus características y poder compararlas con una batería de muestras predefinidas. El clasificador que vamos a utilizar es el vecino más próximo.

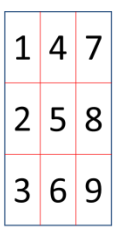
La forma de extraer las características que deseamos es mediante la división de la imagen en 9 bloques (creando una matriz 3x3) y obteniendo la media de los valores de los píxeles par cada uno de los bloques.

Ya que se trata de imágenes binarizadas, en blanco y negro (255 o 0), el valor que se obtendrá al calcular cada una de las medias de los bloques en cuestión estará comprendido entre 0 y 255.

El primer paso es construir una base de datos de entrenamiento obteniendo una batería de imágenes, segmentando los dígitos, clasificándolos manualmente y calculando para cada uno sus características. Con todos los dígitos de entrenamiento se calcula una matriz de referencia necesaria en el proceso de clasificación. Para calcular dicha matriz de referencia (o matriz de entrenamiento del clasificador) se puede utilizar el código explicado en el **Capítulo 4**.

A la hora de calcular esta batería de muestras de entrenamiento se ha tenido en cuenta diversas consideraciones:

- Las imágenes tipo que se han utilizado han sido obtenidas de las imágenes capturadas y segmentadas de este proyecto, ya que así la comparación entre un dígito y esta plantilla es mucho más consistente.
- Para evitar aleatoriedad en los valores de las características de estas imágenes tipo, para cada número se ha realizado la media aritmética tras la realización de este proceso diez veces. Es decir, para el número 0 se ha realizado la extracción de características de diez imágenes que contenían ese número 0 y se ha realizado sus medias aritméticas.
- El orden que se ha seguido en el cálculo de la media de cada uno de los bloques ha sido de arriba abajo empezando por los bloques a la izquierda.



- La variable global donde se guardó estos datos está ordenada de forma que para el número 0 sus valores corresponden a la primera fila, para el número 1 corresponden a la fila 2, y así sucesivamente.

La matriz de tamaño 10x9 donde se encuentran estos valores ha sido nombrada como *mediafinal*:

0	202,242368	219,94945	219,049399	115,389799	12,2017228	171,507809	232,55673	215,492088	189,650095
1	109,926369	146,449954	188,759714	231,351852	235,402778	226,310185	176,096511	126,69131	147,160904
2	117,588932	160,658027	194,11672	143,945198	168,880254	174,379836	234,82102	167,246785	137,802169
3	87,7220266	94,2757426	107,449029	148,948907	175,240401	176,917623	221,72688	217,024315	204,420349
4	221,927458	176,106309	5,23877666	23,0969787	173,346098	22,8689516	205,387191	232,672239	212,048369
5	209,383889	151,20794	116,106153	133,279371	155,276159	166,387826	105,051472	196,405842	202,573925
6	192,443373	205,723374	203,085364	149,123547	169,032207	171,298231	67,7060202	204,134693	213,725454
7	138,149468	0	0	150,161076	39,0086743	82,1954994	245,27765	237,041907	207,235511
8	176,778197	205,58148	207,493522	144,664875	156,747536	165,251694	211,485896	223,095852	185,185081
9	218,211223	165,14199	65,6886906	143,299874	161,650943	147,032494	206,33129	220,77948	202,640645

No obstante, en las primeras ejecuciones se pudo observar que para el dígito que se encuentra a la izquierda del punto decimal que se visualiza no se proporcionaba el resultado obtenido. La causa era que, en ocasiones, dicho punto decimal quedaba adherido al propio dígito después del tratamiento digital.

Es entonces cuando se decidió evaluar tal dígito con otra matriz cuyos valores se verían adecuados a tal situación. Dicha matriz pasó a denominarse *mediafinal2*:

0	202,242368	219,94945	219,049399	115,389799	12,2017228	171,507809	200,55673	200,492088	189,650095
1	109,926369	146,449954	188,759714	231,351852	235,402778	226,310185	176,096511	126,69131	147,160904
2	117,588932	160,658027	194,11672	143,945198	168,880254	174,379836	200,82102	130,246785	137,802169
3	87,7220266	94,2757426	107,449029	148,948907	175,240401	176,917623	200,72688	200,024315	204,420349
4	221,927458	176,106309	5,23877666	23,0969787	173,346098	22,8689516	180,387191	20,6722389	212,048369
5	209,383889	151,20794	116,106153	133,279371	155,276159	166,387826	90,0514717	170,405842	202,573925
6	192,443373	205,723374	203,085364	149,123547	169,032207	171,298231	67,7060202	180,134693	213,725454
7	138,149468	0	0	150,161076	39,0086743	82,1954994	220,27765	215,041907	207,235511
8	176,778197	205,58148	207,493522	144,664875	156,747536	165,251694	200,485896	200,095852	185,185081
9	218,211223	165,14199	65,6886906	143,299874	161,650943	147,032494	190,33129	200,77948	202,640645

Así pues, tan sólo queda diferenciar si hay que evaluar el dígito mediante *mediafinal* o con *mediafinal2*. La forma de saberlo es conociendo el punto final del rectángulo asociado (el vértice inferior derecho).

Dicho punto se puede conocer simplemente accediendo al atributo (.br) del rectángulo. Si la coordenada X de ese punto se encuentra alrededor de la mitad de la anchura de la ROI se evaluará con *mediafinal2* y se añadirá, seguidamente, a la trama de caracteres donde se van almacenando los valores del *frame* actual.

Como criterio para determinar si el punto del que se habla se encuentra en torno de la posición central, se toma como intervalo 1/8 de la anchura de la ROI a ambos lados del punto central de la misma ROI.

El siguiente código se realiza para todos los rectángulos deseados en la ROI actual, donde *comparador* es una función que se ha creado para realizar la clasificación:

```
…

int anchura;
anchura=Frame1.cols;

std::stringstream eco;
eco.clear();
…

…
cv::Point finbound=boundRect[i].br(); //punto final del rect. Esquina sup-izq
int final=finbound.x;                 //coordenada x
bool escentral(false);

if((final>anchura*0.375)&(final<anchura*0.625)){ //Comprobamos si está en la
                                                  // parte central de la ROI
       escentral=true;
}

posi=comparador(rotated,escentral);            //Vamos a la función comparador
eco<<posi;                  //guardamos en la cadena de caracteres

if(escentral==true){        //si es el segundo dígito añadimos el punto decimal
        eco<<".";
}

…
```

La función *comparador* devuelve un entero con la posición de la fila correspondiente al valor del dígito. Los únicos parámetros que necesita son la imagen del dígito extraído (véase apartado 3.4.3.2) y si hay que evaluarlo con *mediafinal* o con *mediafinal2*.

De forma análoga a como se explicó para la batería de muestras, se divide la imagen que se pasa como parámetro en 9 bloques y se va calculando sus medias. Aunque antes hay que redimensionar tal matriz usando *resize*, ya que las plantillas de las que se obtuvieron los valores predefinidos tenían un valor estandarizado por nosotros de 164x308.

- `resize` (Véase 2.3.3.aa.): redimensiona la imagen al tamaño establecido.

Una vez calculadas las medias, se pasa a determinar la distancia euclídea con respecto a cada una de las filas de la matriz predefinida de medias. Y la fila con la distancia menor nos proporcionará el valor que le corresponde al dígito evaluado (clasificación por el vecino más próximo). Para recordar, la distancia euclídea se obtiene así:

$$d = \sum_{k=0}^{n} (med[k] - med'[k])^2$$

- `mean` (Véase 2.3.3.x.): obtiene la media de los puntos de una matriz.

```
int comparador(cv::Mat num, bool central){
	//LO SIGUIENTE SE HACE PARA VER SI ES UN UNO
	double ancho=num.cols;
	double alto=num.rows;
	double relacion=alto/ancho;
	bool esuno(false);
	if(relacion>4){
		esuno=true;
	}

	cv::resize(num,num,cv::Size (164,308),0,0,1);

	int posicion=0;
	int j,r;
	float valant=100000000000000;

	for(int b=0;b<10;b++){
//Se divide num en 9 bloques (convertirlo en una matriz 3x3 bloques)
		cv::Mat bloque;
		float plantt[9];
		cv::Scalar media;
		int columnas=num.cols;
		int filas=num.rows;
		int elem=0;

		for(int y=0;y<3;y++){
			for(int x=0;x<3;x++){

bloque=num(cv::Range((filas/3)*x,(filas/3)*(x+1)),cv::Range((columnas/3)*y,(colum
nas/3)*(y+1)));
				media=cv::mean(bloque);
				plantt[elem]=media.val[0];
				elem++;
			}
		}
//Ahora se obtiene la distancia euclidea con respecto a la de las plantilla

float val=0;
		if (central==false){
			for(r=0;r<9;r++){
val=((plantt[r]-mediafinal[b][r])*(plantt[r]-mediafinal[b][r]))+val;
}
}
```

```
                else{
                        for(r=0;r<9;r++){
val=((plantt[r]-mediafinal2[b][r])*(plantt[r]- mediafinal2[b][r]))+val;
}
                }

if(val<valant){
                        valant=val;
                        posicion=b;
                }
        }

        if (esuno==true){
                posicion=1;
        }

        return posicion;
}
```

Pero evaluando para diversas situaciones, se percibió que, en ocasiones, cuando se deseaba discernir sobre la imagen de un "1" daba unos resultados distintos a los deseados. Esto se debe a que, después de tanto trabajar la imagen digitalmente, el "1" aparece con una forma distorsionada, aunque las dimensiones seguían siendo similares.

Es entonces cuando, sabiendo que el "1" es mucho más estrecho que los otros números, se tomó como criterio que aquella imagen del dígito extraído de la ROI que tuviese una relación altura/anchura mayor que 4 se le proporcionaría el valor de 1, independientemente del cálculo de la distancia con los valores preestablecidos.

3.3.4 SEGUIMIENTO DE LA ROI

Una de las cosas que hay que tener en cuenta es que la ROI seleccionada al principio puede variar su posición con respecto a la imagen capturada por la webcam conforme avanza el tiempo.

Esto conlleva a una situación crítica, ya que no podemos fijar de forma estática la posición relativa de la ROI con respecto a la imagen que se captura. Es decir, a partir de la primera ROI validada por el usuario se debe poder establecer su posición real, frame a frame, de una manera dinámica.

Para poder determinar esa posición se hará uso de la función *cvMatchTemplate* proporcionada por OpenCV. Como parámetros de entrada se proporciona la imagen capturada actual y la imagen de la ROI seleccionada por el usuario al comienzo de la ejecución que servirá como plantilla durante toda la ejecución. Y se guardará en otra

imagen la diferencia en cuadratura de ambas. A continuación, este resultado se normaliza empleando la función *cvNormalize* (también de OpenCV).

- `cvMatchTemplate` (Véase 2.3.3.g.): compara una imagen como plantilla con respecto a otra imagen. Puede emplear 6 métodos distintos: la diferencia en cuadratura, la correlación cruzada, el coeficiente de correlación, y pueden ser normalizados o no.
- `cvNormalize` (Véase 2.3.3.j.): normaliza una imagen respecto a sus máximos y sus mínimos.

```
…

IplImage *src,*templ,*ftmp; // ftmp will hold results

templ=cvLoadImage("PrimerROI.tiff");

int iwidth = bgr_frame->width - templ->width + 1;
int iheight = bgr_frame->height - templ->height + 1;
ftmp = cvCreateImage( cvSize( iwidth, iheight ),32, 1);

src=cvQueryFrame(capture);
cvMatchTemplate( src, templ, ftmp, 0);

cvNormalize( ftmp, ftmp, 1, 0, CV_MINMAX );

…
```

No obstante, esto no es suficiente y hay que interpretar de alguna forma esta información. Como lo que se está calculando es la diferencia en cuadratura y, según la bibliografía sobre OpenCV, las mejores coincidencias se encuentran como mínimos locales, es por ello que se procederá para obtener el mínimo local.

Para solucionar esto, OpenCV ya incluye una función que lo calcula y es *cvMinMaxLoc*:

- `cvMinMaxLoc` (Véase 2.3.3.h.): encuentra el mínimo y el máximo global de la matriz que se le pasa. Devuelve el mínimo y el mayor valor y su localización.

```
…

double min_val=0, max_val=0;
CvPoint min_loc, max_loc;
cvMinMaxLoc(ftmp, &min_val, &max_val, &min_loc, &max_loc);

…
```

Una vez conocido ese valor mínimo y su localización, tan sólo hay que emplear las funciones que ya conocemos para obtener la ubicación y generación de la nueva ROI. Aunque, para que el usuario pueda observar cómo varía la posición de la ROI, se ha dibujado también un rectángulo en su ubicación en la imagen que se captura.

- `cvCloneImage` (Véase 2.3.3.d.): hace la copia de una imagen, incluyendo la cabecera, los datos y la ROI.
- `cvSetImageROI` (Véase 2.3.3.o.): establece una Región de Interés de una imagen para un rectángulo dado.
- `cvRectangle` (Véase 2.3.3.m.): dibuja un rectángulo sobre una imagen.
- `cvRect` (Véase 2.3.3.l.): crea una estructura que guarda las coordenadas de un rectángulo. Los parámetros que se le pasan son la coordenada x e y del vector superior izquierdo, la anchura y la altura.

```
…

cvRectangle(src, min_loc, cvPoint(min_loc.x+templ->width, min_loc.y+templ-
>height), cvScalar(0), 1);

region2=cvRect(min_loc.x,min_loc.y,templ->width,templ->height);
imgroivar = cvCloneImage(src);

cvSetImageROI(imgroivar, region2);

…
```

Cabe decir que esta forma de realizar el seguimiento de la ROI es francamente buena. Ante vibraciones, cambios bruscos en la imagen, desaparición por un tiempo del objeto que se tenía por ROI, cambios drásticos de luz, siempre que se vuelva a una situación de luminosidad y de entorno similar se encontrará la nueva posición de la ROI.

También hay que denotar que no sólo se ha probado esta parte con el multímetro, sino también con cualquier otro tipo de objeto, incluso se ha realizado el seguimiento de caras, de ojos, de cualquier otro rasgo facial, de manos, de dedos o cualquier otra cosa con ostensible movimiento.

4. FUNCIONALIDADES ADICIONALES

4.1. INTRODUCCIÓN

En el capítulo anterior se trató y se describió la metodología empleada para la consecución del proyecto, así como las funciones, código, estructuras y conceptos del tratamiento digital de la imagen fundamentales empleados.

En este capítulo se aborda todo ese código complementario y necesario para poder constituir el proyecto, así como el desarrollo de alguna función y la implementación de una interfaz gráfica que ayuda al usuario a su ejecución.

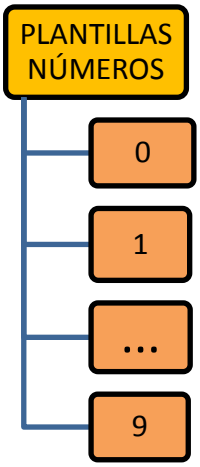
4.2. CÓDIGO PARA CALCULAR LA MEDIAS

La función que se encuentra a continuación es la que nos proporciona los valores de las matrices globales *mediafinal* y *mediafinal2* empleadas por el *clasificador*:

```
void obtenermediafinal(){

	int posicion=0;
	int j,r;
	std::string vv("PLANTILLAS NUMEROS/");
	std::string bb;

	cv::vector<cv::Mat> vector;
	float mediasdedigito[10][9];
	float columnasmedias[9];
	float mediasfinal[10][9];
		for(j=0;j<10;j++){ //Nos deplazamos por todas las subcarpeta

			std::stringstream ss;
			ss<<j;
			std::string hh(vv+ss.str());
			std::string cc(hh+"/");
			for(r=1;r<11;r++){//Nos deplazamos por todas las plantillas
de cada subcarpeta
				std::stringstream zz;
				zz<<r;
				std::string ee(cc+zz.str());
				cv::Mat
num2=cv::imread(ee+".jpg",CV_LOAD_IMAGE_UNCHANGED);
				imshow("PlantillaCompa", num2);
```

```
	//AQUÍ se divide num2 en 9 bloques (convertirlo en una matriz 3x3 bloques)
				cv::Mat bloque;
				float plantt[9];
				cv::Scalar media1, media2;
				int columnas=num2.cols;
				int filas=num2.rows;
				int elem=0;

				for(int x=0;x<3;x++){
					for(int y=0;y<3;y++){

	bloque=num2(cv::Range((filas/3)*x,(filas/3)*(x+1)),cv::Range((columnas/3)*
y,(columnas/3)*(y+1)));
							media2=cv::mean(bloque);
							plantt[elem]=media2.val[0];
							elem++;

					}
				}


				for(int u=0;u<9;u++){//Se almacena todas las medias de
los bloques para cada dígito
					mediasdedigito[r-1][u]=plantt[u];
				}


			}
		//Ahora se calcula la media para cada bloque común de cada digito

			for(int l=0;l<9;l++){
				float acum=0;
				for(int f=0;f<10;f++){
					acum=mediasdedigito[f][l]+acum;
				}
				columnasmedias[l]=acum/10;
			}

			for(int b=0;b<9;b++){
				mediasfinal[j][b]=columnasmedias[b];
			}
		}

		//Se muestran por pantalla todos los valores calculados
		for(int a=0;a<10;a++){
			for(int s=0;s<9;s++){
				std::cout<<" "<<mediasfinal[a][s];
			}
			std::cout<<"\n";
		}
}
```

Antes de explicar la forma de proceder en su funcionamiento hay que decir que las plantillas están guardadas en carpetas estructuradas de la siguiente forma:



De tal forma que en cada carpeta existen diez plantillas del número correspondiente a como se ha nombrado. Es decir, en la carpeta "0" hay diez plantillas del dígito "0", en la carpeta "1" hay diez plantillas del dígito "1",…

Cada vez que se accede a una plantilla, ésta se divide en 9 bloques (3x3) y se calcula la media de cada uno de los bloques, guardando estos valores en la matriz `mediasdedigito`. Una vez que se ha accedido a todas las plantillas de una carpeta, se calcula la media de cada columna de `mediasdedigito`, donde cada columna corresponde a un mismo bloque.

Esta última operación se almacena en la matriz `mediasfinal` cada vez que se termina de acceder a cada carpeta.

4.3. MOSTRANDO/GRABANDO.

Una vez que el usuario ha seleccionado una ROI válida puede determinar en qué momento comienzan a guardarse los datos en un fichero. Cuando se desee guardar dichos datos habrá que pulsar la tecla INTRO. Para ello se establecido un *flag* booleano llamado `grabar`.

Pero para que el usuario pueda interpretar qué se está haciendo en cada momento, se ha creado una ventana de texto que le informa si solamente se están mostrando por pantalla los datos adquiridos o si además se están grabando.

Para poder mostrar esta información se usa la función *putText*:

- `putText( Mat& img, const string& text, Point org, int fontFace, double fontScale, Scalar color, int thickness=1, int linetype=8, bool bottomLeftOrigin=false )`

  Muestra una cadena de texto.

  Parámetros:

  - *img:* la imagen.
  - *text:* el texto a mostrar.
  - *org:* La esquina inferior izquierda de la imagen donde comenzará el texto.
  - *fontFace:* el tipo de fuente. Puede ser: FONT_HERSHEY_SIMPLEX, FONT_HERSHEY_PLAIN, FONT_HERSHEY_DUPLEX, FONT_HERSHEY_COMPLEX, FONT_HERSHEY_TRIPLEX, FONT_HERSHEY_COMPLEX_SMALL, FONT_HERSHEY_SCRIPT_SIMPLEX o FONT_HERSHEY_SCRIPT_COMPLEX
  - *fontScale:* factor de escala de la fuente.
  - *color:* color del texto.
  - *thickness:* grosor de las líneas para dibujar el texto.
  - *linetype:* tipo de línea.
  - *bottomLeftOrigin:* Cuando es verdadero, el origen de datos de la imagen se encuentra en la esquina inferior izquierda, si no es en la esquina superior izquierda.

El código donde se ve reflejado lo explicado es el siguiente:

```
…

double fontScale = 1;
int thickness = 2;
cv::Point textOrg(10, 30);

…

bool grabar(false);
std:: string funcionando("MOSTRANDO");
std::string grab("GRABANDO");
cv::Mat ejec(45, 400, cv::DataType<float>::type);
ejec=255;
cv::putText(ejec, funcionando, textOrg, 3, fontScale, cv::Scalar::all(0),
thickness,8);
imshow("EJECUCION",ejec);
…

wk=cv::waitKey(1);

if(wk==13){
```

```
        grabar=true;
        ejec=255;
        }


…


if (grabar==true){
        …
        cv::putText(ejec, grab, textOrg, 3, fontScale, cv::Scalar::all(0),
thickness,8);
        imshow("EJECUCION",ejec);
  }
…
```

A continuación se muestra un ejemplo de cómo se visualiza en el monitor esta interfaz.

Imagen 24: Imagen de la interfaz si está grabando o no

4.4. LAS VARIABLES "POSI" Y "ECO"

La variable *posi* es donde se va almacenando cada uno de los dígitos determinados por el clasificador.

Y la variable *eco* es la cadena de caracteres donde se guardan correlativamente los dígitos en cada una de las capturas de la webcam.

```
…

int posi;
std::stringstream eco;

…

posi=comparador(rotated,escentral);
eco<<posi;

…
```

4.5. CONTROL DE LOS DÍGITOS Y MOSTRAR RESULTADO

Como se indicó en el punto 3.3.3.2 (SEGMENTACIÓN) para poder controlar y discernir si el contorno que se está evaluando pertenece a un dígito o no se utiliza una barra de desplazamiento con la que el usuario podrá discriminarlos.

Además, para un manejo más sencillo, se puede ver en la misma ventana el valor que se obtendría para cada posición de la barra de desplazamiento.

La forma de visualizar los valores se realiza usando la función *putText* explicado en un apartado anterior. Y la función que nos proporciona la barra de desplazamiento es *createTrackbar*:

- `createTrackbar( const string& trackbarname, const string& winname, int* value, int count, TrackbarCallback onChange=0, void* userdata=0)`

  Crea una barra de desplazamiento y la adjunta a la venta especificada.

  Parámetros:

  - *trackbarname:* nombre que se le da a la barra de desplazamiento.
  - *winname:* nombre de la ventana que se utiliza.
  - *value:* puntero opcional a una variable de tipo entero cuyo valor refleja la posición de la barra. Tras su creación, la posición del control deslizante está definido por esta variable.
  - *count:* la posición máxima de la barra. La posición mínima es siempre 0.
  - *onChange:* puntero a la función que se llamará cada vez que el cursor cambia de posición.
  - *userdata:* los datos de usuario que se pasan como es la devolución de la llamada.

El entorno del código donde se usan se puede consultar a continuación:

```
…

int umbral=40;
…

cv::createTrackbar("ValUmbr","TEXTO",&umbral,100);

…
saliendo=eco.str();
cv::putText(fofo, saliendo, textOrg, 3, fontScale, cv::Scalar::all(0),
thickness,8);
imshow("TEXTO",fofo);
```

La forma en la que aparecen y se muestran al usuario es la siguiente:

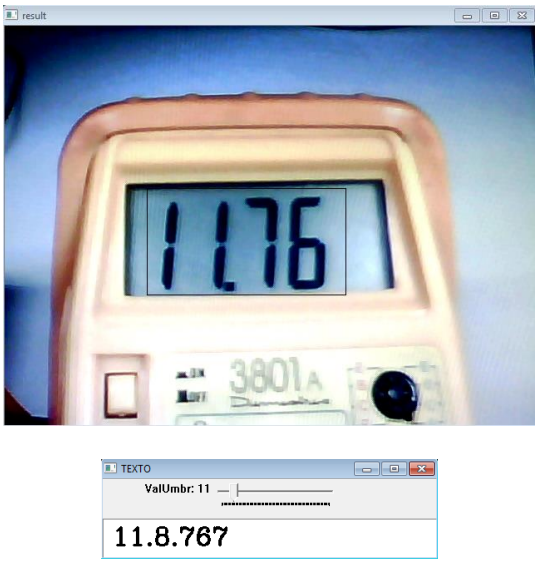


Imagen 25: Mostrar valores y elección mala del valor umbral

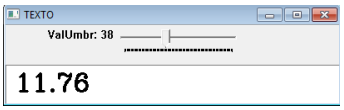


Imagen 26: Mostrar valores y elección buena del valor umbral

4.6. CREACIÓN DEL FICHERO.

El resultado final de la detección y clasificación de los dígitos se almacena en un fichero con extensión *.txt* con el nombre *Datos*.

Además de esta información, para completarla, se añade la fecha y la hora en la que han sido almacenados.

Se ha determinado que, para una sencilla importación de este fichero a otro programa para el tratamiento de los datos, la separación entre el valor obtenido y la fecha y hora sea mediante un espaciado tabulado.

```
…

time_t rawtime;
std::ofstream mifichero;
mifichero.open ("Datos.txt");
…

mifichero<<posi;
…
```

```
time ( &rawtime );
mifichero <<"\t"<< ctime (&rawtime);
…

mifichero.close();
…
```

Seguidamente se muestra la forma con la que el fichero Datos.txt muestra la información:

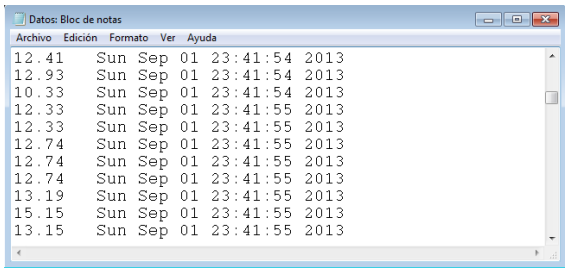


Imagen 27: Datos.txt

5. RESULTADOS

Definidos y explicados todos los elementos del sistema es necesario conocer la bondad y fiabilidad del proyecto en todo su conjunto e interpretar los datos que proporciona su ejecución.

Para realizar la validación del sistema se plantea la medición de la tensión de carga de un condensador de tal forma que se obtenga la mayor variedad de adquisiciones y forzar todas las posibles anomalías que pudieran surgir.

Cabe recordar que la tensión de carga de un condensador viene dada por la fórmula:

$$Vc(t) = \mathcal{E}(1 - \exp(-t/RC))$$

Donde:

- $V_C(t)$ es la tensión en el condensador.
- $\mathcal{E}$ es la tensión de la fuente.
- t es el tiempo.
- R es la resistencia del circuito.
- C es la capacidad del condensador.
- τ es la constante de tiempo.
- El criterio para considerar que el condensador está cargado es: $t = 5\tau$.

Así pues, el circuito sobre el que vamos a medir la tensión del condensador con el multímetro es el siguiente:

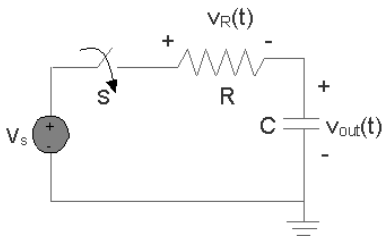


Imagen 28: Circuito de carga de un condensador

A continuación se establece como criterio para establecer los valores de R y C que **t ≃ 60 segundos**, tiempo necesario para capturar las suficientes imágenes con sus respectivos procesados y poder realizar un estudio de los valores adquiridos.

Ya que en el laboratorio el número de componentes es limitado, se escogen una resistencia y un condensador que nos proporcionen un resultado similar al criterio establecido: R = 100 kΩ y C = 100 μF. Pero el valor real de la resistencia es R = 99,1 kΩ. Por tanto:

R = 99,1 kΩ, C = 100 μF → τ = 9,91 segundos

Siendo **t** = 5 τ = **49,55 segundos**

Así pues, hay que realizar la medida de tensión del condensado durante 50 segundos como mínimo.

El montaje del circuito queda así:

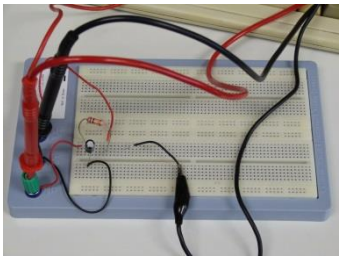


Imagen 29: Circuito con las sondas del multímetro para medir

Además, como último dato falta conocer la tensión que proporciona el generador, siendo ésta: **$\mathcal{E}$ = 15.35 V.**

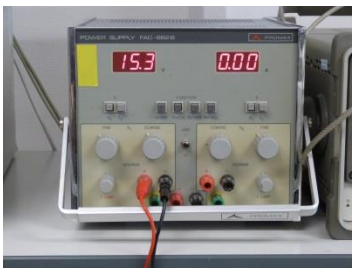


Imagen 30: Generador de tensión

Una vez dispuesto el circuito, el multímetro y la webcam conectada al ordenador se procede a la ejecución del proyecto de la misma manera que se ha ido explicando a lo largo de toda la memoria. En la imagen siguiente se puede ver el montaje completo en el laboratorio:

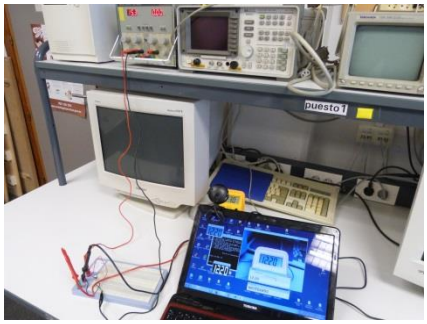


Imagen 31: Montaje para realizar las medidas

No obstante, a continuación se enumeran los pasos a seguir para que no haya lugar a la duda:

1. Se da comienzo para ejecutar el programa.
2. Aparece la pantalla de inicio, se pulsa 0 e INTRO.
3. Se enfoca el multímetro con la webcam si no lo estaba antes.
4. Se pulsa INTRO y seguidamente ESC para que aparezca la imagen donde se selecciona la ROI.
5. Se selecciona la ROI con el ratón. Si no se ha escogido correctamente puede volver a repetirse este paso tantas veces como se desee, simplemente pulsando el botón derecho del ratón.
6. Se ajusta el valor de los contornos para discriminar el ruido mediante la barra de desplazamiento si procede y se pulsa ESC.
7. En cada captura se muestran los valores que se obtienen después de todo el proceso. Se puede volver a ajustar la barra de desplazamiento.
8. Cuando se desee empezar a guardar los valores se pulsa INTRO.
9. Se pulsa ESC para finalizar.

Una muestra de lo que se puede ir visualizando en el monitor del ordenador mientras se está ejecutando es la siguiente:

Imagen 32: Visualización del ordenador mientras se toman medidas

Una vez realizado estos pasos se ha creado el fichero *Datos.txt* con todos los valores junto a la fecha y hora a la que se han adquirido.

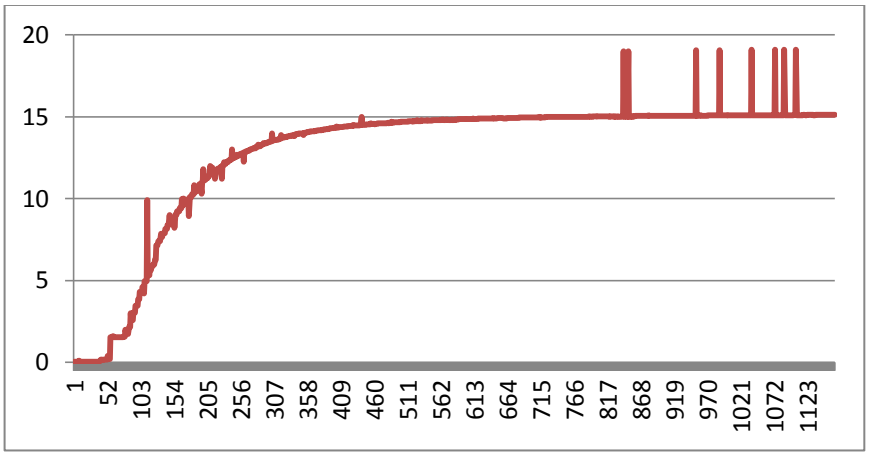
Siguiendo nuestro caso para los valores de R y C anteriores, se abre tal fichero y lo primero que se comprueba es el primer y último dato para ver qué diferencia de tiempo existe entre ellos:

```
0.05    Tue Sep 03 10:26:26 2013
…
15.10   Tue Sep 03 10:28:34 2013
```

Como se puede observar, la diferencia de tiempos es de **128 segundos**, superando con creces el criterio establecido.

Esta medición tan larga se ha realizado a conciencia con el propósito de comprobar cómo se comporta la ejecución del proyecto con valores estáticos.

Así pues, la gráfica resultante con todos los datos es la siguiente:

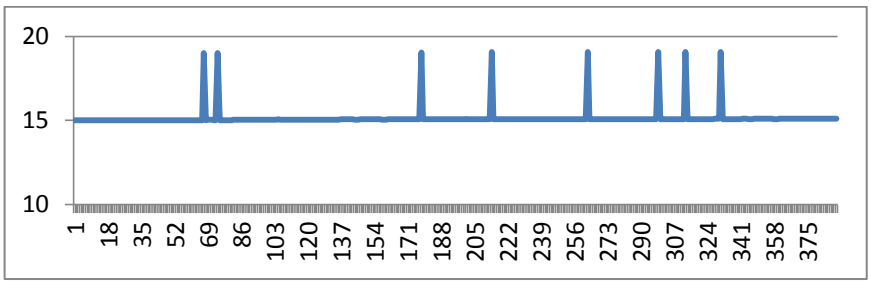


Como se puede comprobar, la gráfica tiene la forma de la evolución de la carga de tensión del condensador como cabía esperar, aunque se pueden ver algunos valores que se comportan de forma errática.

En primera instancia podría pensarse que todos estos valores erróneos se deben a la misma causa. Pues bien, esto no es así y hay que diferenciarlos en dos grupos: los que ocurren cuando se ha estabilizado el valor de la tensión en torno a los 15V y los que aparecen mientras se está cargando el condensador.

En cuanto a los primeros, se puede decir que se dan principalmente por variaciones en el ambiente de trabajo. Cabe recordar que las mediciones, aunque se realizan en un laboratorio, existen factores que no podemos controlar como son posibles reflejos, sombras,… que pueden alterar el estado del visualizador del multímetro, provocados incluso por el usuario que está realizando las medidas.

Aun así, como podemos ver en la gráfica, desde que se estabilizan los valores por encima de 15V tan sólo aparecen 8 valores erróneos frente un número de muestras de 389. Es decir, el número de muestras erróneas en este intervalo es del **2.3%**, pudiendo ser descartadas con facilidad.



Ahora bien, a los otros errores no solamente les afecta este condicionante, están influidos además por los cambios de un dígito a otro. Es decir, cuando se cambia de un valor a otro, hay un instante en el que se superponen los dos valores y se visualizan a la vez. Por esto, si coincide en ese mismo momento que se captura una imagen, no se podrá distinguir el dígito correctamente.

Se deja para trabajo futuro la reducción de estos errores.

A continuación se muestra una imagen capturada en el que aparecen dos valores:

Imagen 33: Instante del multímetro cambiando de un valor a otro (14.29 y 19.68)

Pero de los datos aún podemos obtener otra conclusión. Si los datos han estado almacenándose durante 128 segundos y tenemos 1165 valores guardados, por consiguiente, tenemos una media de **9.1 capturas procesadas por segundo**, o lo que es lo mismo, **una captura cada 0.11 segundos**.

Esta velocidad de captura, procesado software y almacenamiento nos indica que se cumple el objetivo de proporcionar resultados en tiempo real.

Pero hay que tener en cuenta que esta velocidad no es siempre la misma. Dependerá del ordenador desde donde se esté ejecutando y de los recursos que esté empleando durante la ejecución. Para este caso se trabaja con un ordenador con las siguientes características:

- Procesador Intel® Core(TM) i5-2410M CPU @ 2.30 GHz.
- El sistema operativo sobre el que se trabaja es Windows 7 (sistema operativo de 64 bits).
- RAM de 4 GB.

Aunque también hay que decir que esta velocidad de la que se habla viene dada, en gran parte, a que la librería OpenCV fue desarrollada con miras de aprovechar los recursos proporcionados por procesadores multi-core.

6. CONCLUSIONES Y LÍNEAS FUTURAS

CONCLUSIONES

En esta memoria se ha pretendido explicar las distintas etapas necesarias para el diseño completo de un sistema que posibilitara adquirir de forma automática los valores mostrados por un aparato de medida sin capacidad de almacenaje. Este aparato de medida se trata de un multímetro específico del que se obtienen imágenes a través de una webcam, un elemento de captura de imágenes sencillo, barato y, además, limitado.

Para que el usuario pudiera disponer fácilmente de los datos para su tratamiento, se ha determinado que el resultado final sea almacenado en un fichero de texto, de tal forma que pueda ser importado por cualquier programa de tratamiento de datos u hojas de cálculo.

Además, se ha conseguido que todo el proceso se ejecute en tiempo real para que el usuario pueda acceder al instante a los datos. Para ello se ha intentado evitar utilizar código que pudiera aumentar los costes de carga del sistema.

Otro objetivo que se planteó al inicio del desarrollo de este proyecto fue que el sistema tuviera cierta inmunidad a movimientos y vibraciones de la imagen. Pues esto se ha conseguido, y con creces, realizando el seguimiento de la ROI frame a frame. Incluso, si en algún momento desaparece el multímetro de la imagen captada por la webcam, con posterioridad, sin necesidad de parar la ejecución, puede volver a reconocerlo.

A la hora de tener en cuenta la luminosidad de la estancia que afecta a la captura de la imagen, se ha conseguido que la imagen se adapte a la procedencia de los distintos focos de luz que pudieran haber, exceptuando cuando un foco de luz es reflejado por el visualizar del multímetro deteriorando la imagen captada de los dígitos.

Otro propósito que se planteó fue el reconocimiento de los dígitos con algún grado de inclinación. Pues bien, empleando el entrenamiento que se ha desarrollado se consigue que no sea necesario que los dígitos estén perfectamente alineados, pero una inclinación notoria hace que el planteamiento falle.

Y por último, y no menos importante, para facilitar al usuario el tratamiento de los datos que se guardan, se ha conseguido determinar la posición del punto decimal de la medida en cuestión sin que tenga que determinarlo él mismo.

Por consiguiente, aunque el sistema ha sido diseñado para un aparato de medida específico, proporciona una sencillez tanto en ejecución como en resolución, al igual que una robustez y una estabilidad, que posibilita una amplia gama de futuros desarrollos y aplicaciones y de mejoras posteriores.

LÍNEAS FUTURAS

Como se ha comentado en el apartado de conclusiones, la finalización de este proyecto proporciona una buena base donde asentarse el inicio de otros tantos.

No obstante, es preferible hablar de posibles líneas inmediatas y factibles que aventurarse en acometidas ideales sin tener una estructura donde originarse.

Siguiendo esta pauta, sería de agradecer por parte del usuario final que existiera la posibilidad de la detección automática de la ROI. Para esto, seguramente, no existe un camino único y válido y dependerá de cómo se afronte este problema. Podría hacerse detectando formas, o etiquetando grupos de píxeles similares, o detectando una marca puesta en el multímetro,…

Como se ha dicho anteriormente, este proyecto se ha realizado para un multímetro en concreto, pero no resultaría difícil generalizarlo para otros modelos. Incluso se podría generalizar a otros aparatos de medida muy distintos, como podría ser un termómetro eléctrico en una cámara blanca o una báscula electrónica.

Además, sería interesante procesar los datos convenientemente para sustraer los valores erróneos y dar mucha más fiabilidad al sistema, al igual que añadir algún otro dato de interés en el fichero. Una solución para reducir el número de errores es la de no almacenar un dígito hasta que no se estabilice, es decir, si en N capturas consecutivas el dígito es el mismo se almacena y, si no, no se almacena.

Y aunque la interfaz gráfica proporcionada por la plataforma software que se ha utilizado es suficiente para el correcto funcionamiento, no está de más poder mejorarla teniendo en cuenta la aplicación y el uso que se le vaya a dar.

Por último, se plantea que también pueda realizarse el visualizado y detección de aparatos de medida analógicos, aquellos que la medida la proporcionan mediante una aguja sobre unas líneas de medición. Pero para este tipo de útiles de medir el procedimiento a seguir es muy distinto al expuesto en este proyecto. En estos casos estaríamos hablando de seguimiento y estimación de movimiento.

BIBLIOGRAFÍA

I. Learning OpenCV. Computer Vision with the OpenCV Library. Ed. O'REILLY

II. Tutorial de OpenCV. Computer Vision Lab. Autores: Raúl Igual, Carlos Medrano

III. Transparencias y Notas de la Asignatura: Tratamiento Digital de la Imagen, ETSIT. Valery Naranjo Ornedo

IV. Digital Image Processing, W.K. Pratt Ed. Wiley

V. Digital Image Processing, González Addison-Wesley

VI. Digital Video Processing, Murat Tekalp Ed. Prentice Hall

VII. The Image Processing Handbook, John Russ. Ed. C.R.C. Press

VIII. Topological structural analysis of digitized binary images by border following, Satoshi Suzuki, and Keiichi Abe. *Computer Vision, Graphics, and Image Processing 30(1):32-46* (*1985*)

IX. C++ Language Tutorial Autor: Juan Soulié

X. http://opencv.willowgarage.com/wiki/

XI. http://docs.opencv.org/

XII. http://es.wikipedia.org

XIII. http://en.wikipedia.org

XIV. http://stackoverflow.com/