|  |
| --- |
| PRÁCTICA 4 |
| SEGMENTACIÓN Y RECONOCIMIENTO |
| VISIÓN ARTIFICIAL |

|  |
| --- |
| Azael Escudero López, Sergio Ferrer Gimeno (GII+GIS)  15-5-2014 |

Contenido

[1. Segmentación de los dígitos de la matrícula 2](#_Toc387920566)

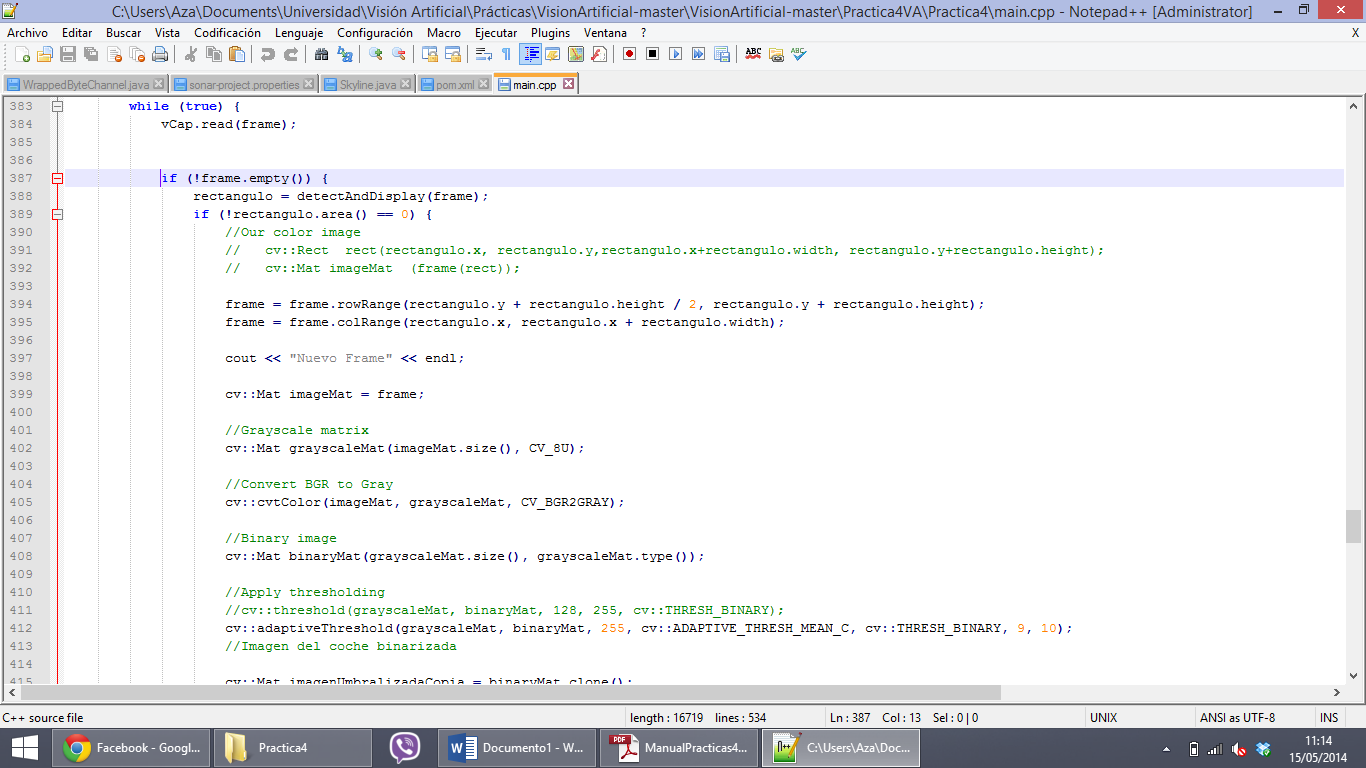
[2. Reconocimiento de los dígitos segmentados 3](#_Toc387920567)

[2.1. Fase de aprendizaje 3](#_Toc387920568)

# Segmentación de los dígitos de la matrícula

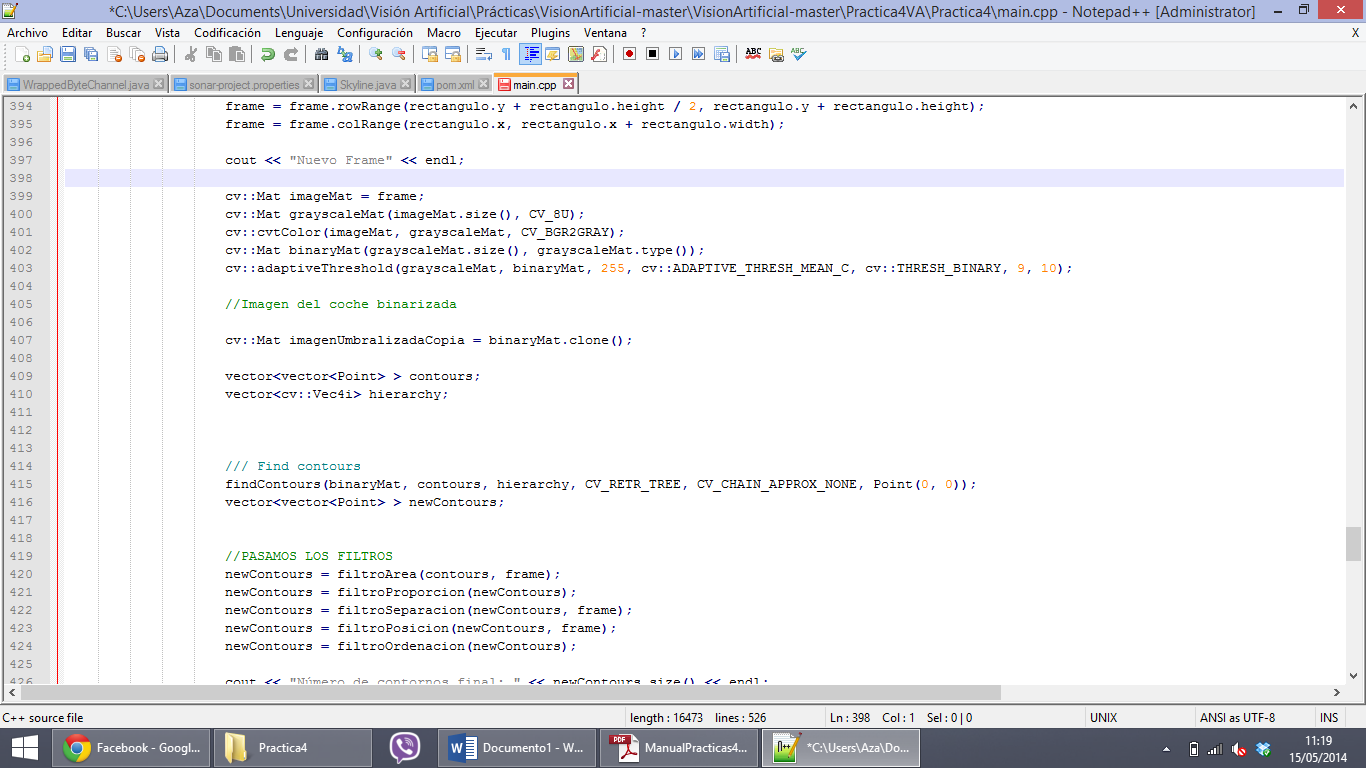
Para construir el sistema de segmentación de los dígitos, nos basamos en la binarización de la imagen y posterior detección de objetos conexos.

Antes de binarizar, para ahorrar recursos decidimos que era más eficiente reducir nuestro espacio de trabajo a la mitad inferior de de la región en la que se encuentra el coche, ya que ahí es donde supuestamente debe estar la matrícula. Para hallar el coche utilizamos el sistema implementado en la anterior práctica.



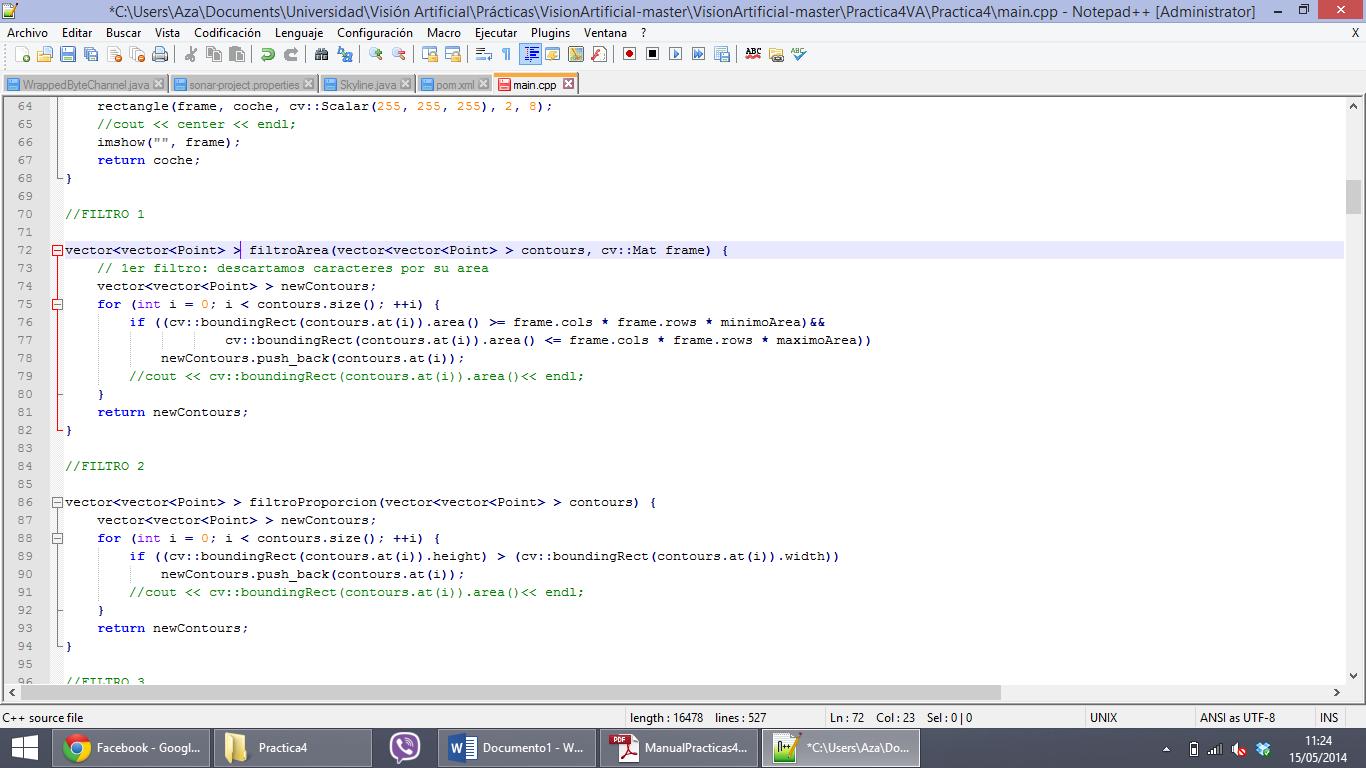
**frame** es la imagen de testing y **rectángulo** la región en la que se encuentra el coche

A continuación pasamos la imagen a escala de grises, y le aplicamos un umbralizado adaptativo para binarizarla.

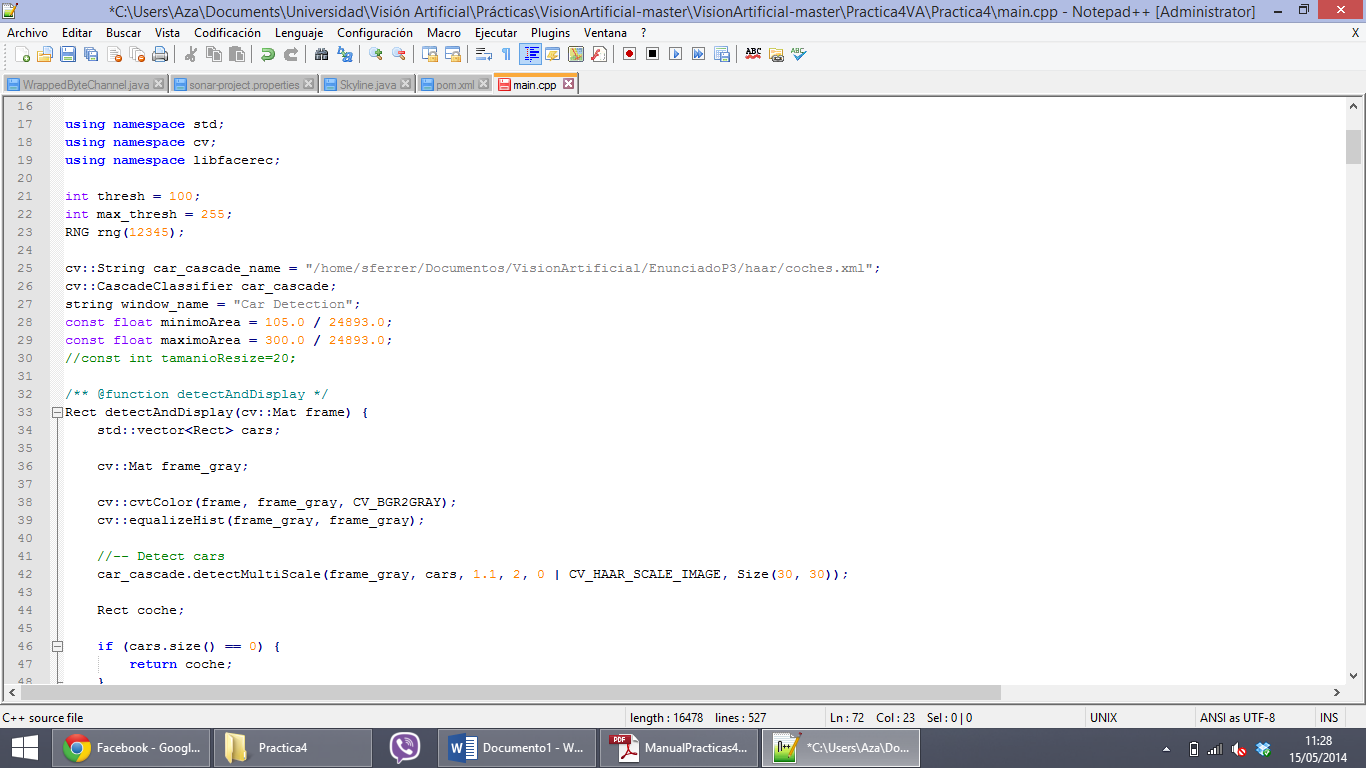


Una vez tenemos la imagen binaria, podemos hallar los contornos que hay en ésta utilizando el método **findContours**. En el vector de vectores de puntos que nos devuelve este método tenemos todos los contornos que se encuentran en la imagen, tanto dígitos de la matrícula como otros posibles contornos que halle el método, por lo que debemos aplicar una serie de “filtros selectivos” para desechar los contornos que no nos interesan.

Primero utilizamos el **filtroArea** en el que descartamos contornos si el área de su **boundingRect** no se encuentra dentro de un determinado rango de valores.



Las variables *minimoArea* y *maximoArea* tienen valores que calculamos mediante ensayo-error utilizando una imagen de test cuya área era 24893:



A continuación pasamos el **filtroProporcion**, que descarta los contornos cuyo **boundingRect** es más ancho que alto; el **filtroSeparacion**, que solo se queda con los contornos que tienen otro contorno a diez (o menos) píxeles de distancia, y el **filtroPosicion** que descarta los contornos que no se encuentren en la zona central de la imagen. Por último, utilizamos el **filtroOrdenacion** para ordenar los supuestos dígitos que se encuentren de izquierda a derecha.

No obligamos al programa a que encuentre 7 caracteres exactos, porque en algunas imágenes de test, las matrículas tienen más de 7 (contando el “ESP”).

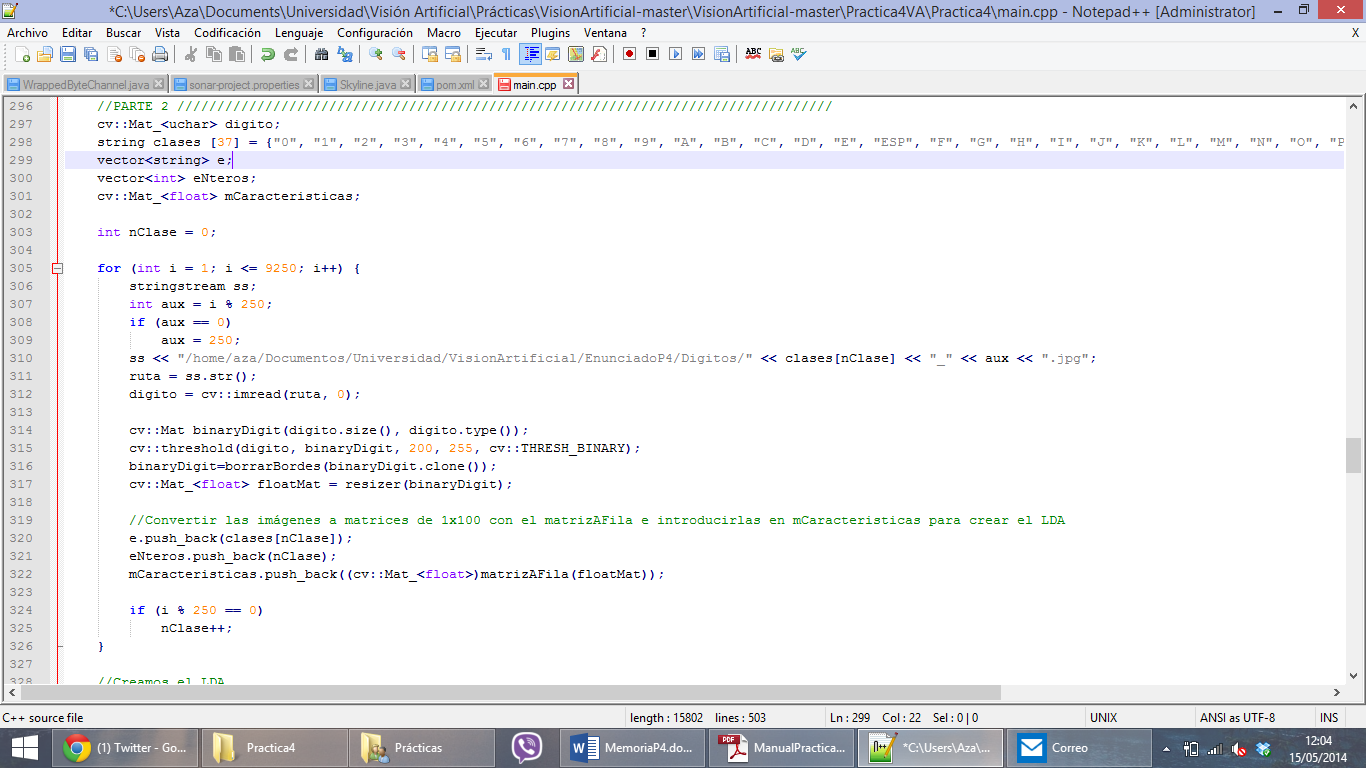
De esta manera, ahora podemos conseguir una lista con todas las regiones (utilizando **boundingRect**) en las que se encuentran los dígitos de la matrícula. Así, “recortamos” cada dígito (utilizando **rowRange** y **colRange** como hicimos anteriormente) y finalmente lo redimensionamos al mismo tamaño que tendrán los caracteres en la fase de Learning.

# Reconocimiento de los dígitos segmentados

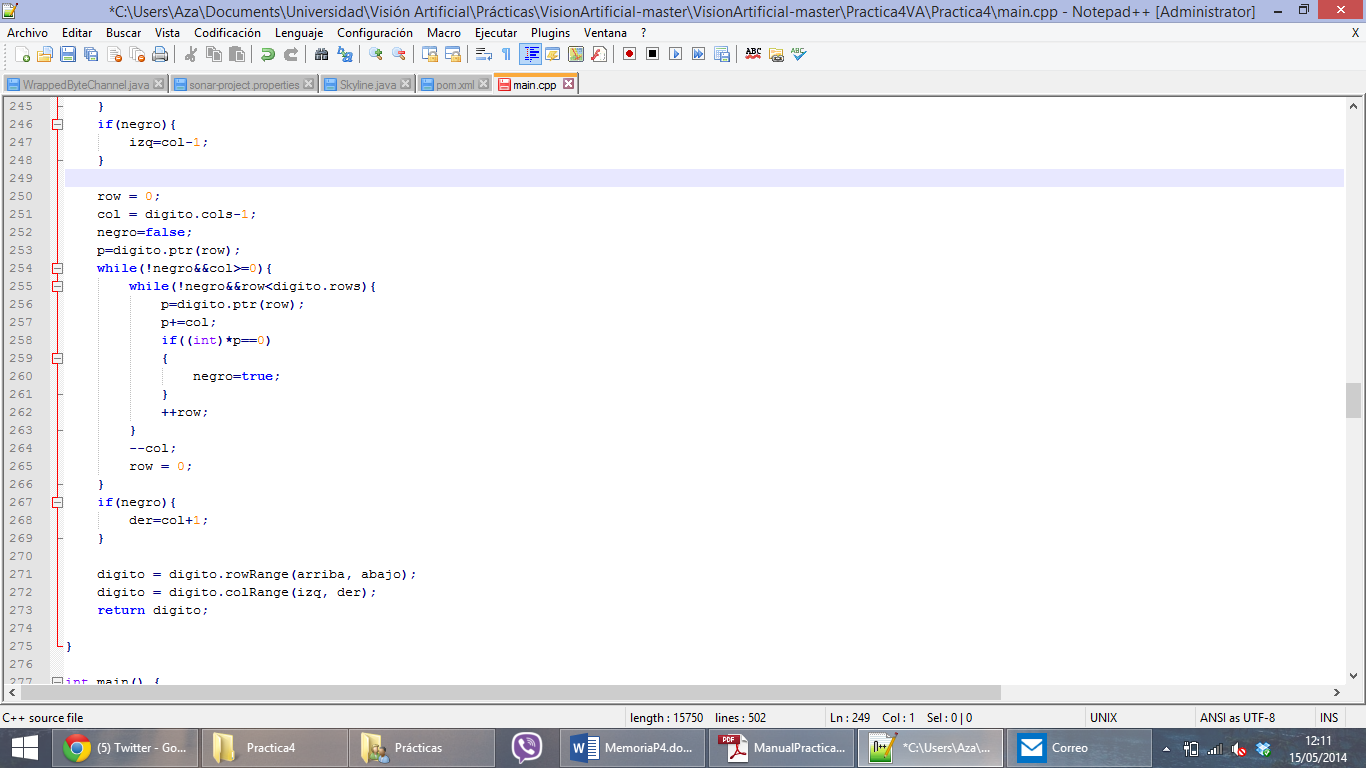
## Fase de aprendizaje

### Preparando los dígitos

Se nos proporciona un conjunto de caracteres de entrenamiento que se corresponden a los caracteres que pueden aparecer en las matrículas. Para construir el sistema de clasificación debemos extraer y guardar el vector de características de cada uno de ellos, que será lo que utilizaremos para comparar con los caracteres encontrados en el apartado de segmentación.

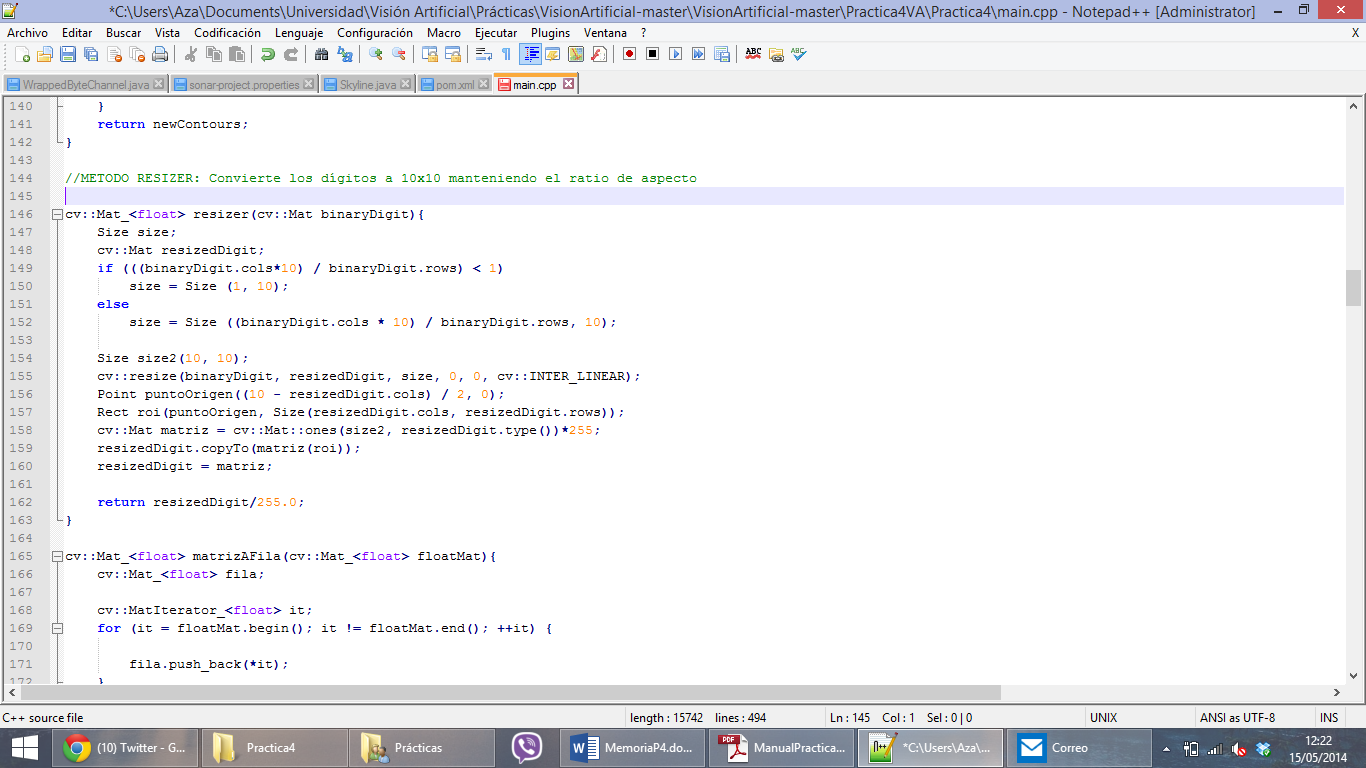


Para ello, cogemos cada imagen (correspondiente a un dígito de aprendizaje), le pasamos un umbralizado (en este caso no es adaptativo, ya que comprobamos que daba mejores resultados) y le aplicamos las funciones **borrarBordes** y **resizer**.

 La función **borrarBordes** sirve para “centrar” el dígito en la imagen, o dicho de otra manera, para ajustar los bordes de la imagen al dígito. Esto lo utilizamos porque los dígitos encontrados en la segmentación también están así. Lo que hace exactamente es recorrer la matriz correspondiente al dígito de arriba abajo, de abajo a arriba, de izquierda a derecha y de derecha a izquierda. En cada una de las cuatro fases encuentra la primera fila (o columna) en la que hay un píxel negro y ajusta (con **rowRange** y **colRange**) la imagen en base a su coordenada.

En este fragmento de código podemos ver el último bucle de esta función, que es el que recorre la matriz digito de derecha a izquierda. Después cuenta las columnas blancas que ha recorrido y se reajusta la matriz con todas las variables obtenidas anteriormente.

Por otro lado, la función **resizer** redimensiona el dígito convirtiéndolo en una imagen de 10x10 (como se pedía en el enunciado). Para ello no se limita a utilizar simplemente el método **cv::resize()** (con interpolación lineal), sino que tiene en cuenta el *aspect ratio* de la imagen para no deformar el dígito al redimensionarlo “pegándolo” redimensionado con alto 10 y ancho *x* en el centro de una imagen en blanco de 10x10. Además deja los valores de los píxeles de la imagen entre 0 y 1, no entre 0 y 255, para que sea más eficaz el sistema a la hora de clasificar los dígitos.

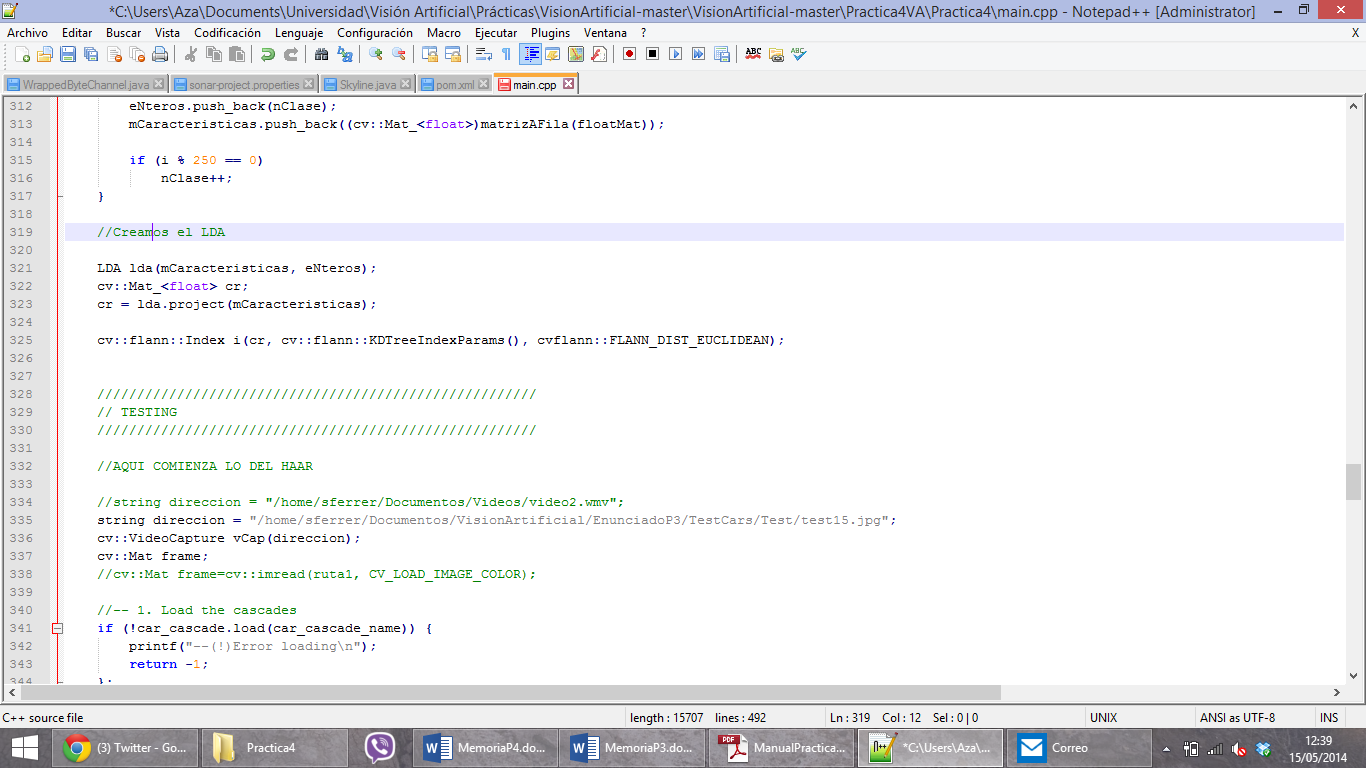


Captura del método **resizer()**

Una vez preparado el dígito, lo convertimos en una matriz de 1x100 con el método **matrizAFila()** para poder usarla como un vector de características. Finalmente almacenamos los más de 2000 vectores de características en *mCaracteristicas*.

### Vectores de características y LDA

Ya tenemos los vectores de características, pero tienen una dimensión demasiado grande. Utilizaremos LDA para reducirla.



En el vector *eNteros* tenemos almacenadas las labels (o clases) correspondientes a cada dígito que puede aparecer en la matrícula (lo fuimos construyendo en el apartado 2.1.1.). Así, construimos un objeto LDA y con su método **project** proyectamos la matriz *cr*, una matriz de características reducidas. Hay que tener en cuenta que la clase correspondiente al dígito de la fila f de *cr*, y la clase que indica la posición f en *eNteros* serán la misma.

# Conclusiones

La realización de esta práctica ha sido, quizás, más contundente que la de la práctica anterior, pero a la vez más interesante puesto que como ya controlábamos mucho más tanto C++ como OPENCV nos hemos podido centrar más en los asuntos conceptuales y del algoritmo y no tanto de la implementación en sí del código.

También comprobar que los algoritmos implementados en la anterior práctica (Detección de coches con HAAR , procesamiento de secuencias de vídeo…) han sido útiles y han servido de apoyo para poder afrontar esta práctica ha sido muy satisfactorio.

Por otro lado la a veces poca precisión del sistema implementado ha podido ser en determinados momentos un poco frustrante aunque somos conscientes de que el sistema funciona así en pos de una mayor eficiencia y simplicidad a la hora de hacer la práctica. Aun así ha sido una muy buena aproximación a un algoritmo de detección de matrículas real que nos ha permitido comprender como funcionan muchos de los algoritmos de visión artificial como los de detección de bordes, contornos, umbralización, reescalado de imágenes y sus interpolaciones, etc.