|  |
| --- |
|  |
| PRÁCTICA 1: DETECCIÓN DE OBJETOS |
| Visión artificial. Grado en Ingeniería Informática |
|  |
| **Jorge Aranda García**  **Jose Vicente Bañuls García** |
| **31/03/2018** |

|  |
| --- |
|  |

ÍNDICE

[Ejercicio 1: Detección de coches mediante punto de interés 3](#_Toc510171865)

[Ejercicio 1.1: Estadísticas 5](#_Toc510171866)

[Ejercicio 2: Detección de coches usando cv2.CascadeClassifier 6](#_Toc510171867)

[Ejercicio 2.1: Estadísticas 6](#_Toc510171868)

[Ejercicio 3: Detección del coche en secuencia del vídeo 8](#_Toc510171869)

# Ejercicio 1: Detección de coches mediante punto de interés

El objetivo de este ejercicio es detectar coches en fotografías tomadas en distintas situaciones, posiciones y calidad, a partir de unas fotografías de coches tomadas de frente y con un cierto nivel de calidad. Para ello hemos seguido un camino bastante progresivo. Primero comenzamos leyendo las imágenes de entrenamiento del algoritmo y reconociendo los puntos de interés y los descriptores de éstos, comprobando que este procedimiento se realizaba correctamente. Para ello se usan las funciones *"detect"* y *"compute"* de la clase ORB.



*Ejemplos simples de detección de puntos de interés. 3 puntos de interés con un 1 nivel de profundidad de la pirámide*

**

*Ejemplos más complejos de detección de punto de interés. En concreto 500 puntos de interés con 6 niveles de profundidad de la pirámide*

A continuación, estos puntos de interés se almacenan en una estructura de datos "*FLANN*", que es una librería pensada para realizar búsquedas aproximadas del vecino más cercano de forma rápida en grandes espacios.

Tras haber almacenado los puntos de interés de las imágenes de entrenamiento, se empieza a tratar las imágenes de test. Además de encontrar los puntos de interés y descriptores de estas imágenes hay que hacer algunas cosas más. Una vez encontrados los puntos de interés hay que buscar para cada uno de ellos, los vecinos más parecidos de entre los encontrados en las imágenes de entrenamiento (Al principio fueron 2 y al final 6. **lp = flann.knnMatch(d, k=6)**, donde d es la lista de descriptores de la imagen de test y k es el número de vecinos a explorar, devuelve la lista de los descriptores parecidos). Una vez comparados los puntos, hay que crear una matriz de votación del tamaño que la imagen de test, para decidir dónde está el coche. Para conseguir que la acumulación sea evidente se ha decidido dividir las dimensiones de esta matriz por 10, para que varios puntos distintos voten en la misma posición. Por lo tanto, en la imagen real, el centro del coche no estará descrito por un único píxel, sino por una región de píxeles. Para realizar las votaciones, se coge el vector que va del punto de interés de la imagen de entrenamiento al centro de esa imagen y se suma al punto de interés. En la posición resultante se suma un voto. Tras haber realizado el mismo proceso con todos los puntos de interés, la posición de la matriz con mayor número (se saca con la función "*cv2.unravel\_index*") de votos sería el supuesto centro del coche. A pesar de que esta primera aproximación no es realista, ya empezamos a obtener los primeros resultados. Como se ha comentado anteriormente, esta aproximación no es realista puesto que para el cálculo de la matriz de votaciones no se ha tenido en cuenta ni el ángulo ni la escala de los puntos de interés. Por lo tanto, ahora, al sumar el vector hay que escalar éste de tal forma que se corresponda con la escala del punto de interés de la imagen de test. Además, los puntos de interés tienen un ángulo que no tiene por qué coincidir, por lo que hay que orientar el vector con relación al ángulo del descriptor de la imagen de test. En este punto los resultados sí que son realistas y ya podemos pasar a dibujar la situación que nuestro algoritmo devuelve como centro del coche.

*A continuación, se muestran algunos ejemplos del funcionamiento de nuestro algoritmo, en los que el centro del coche está representado por el cuadrado resultante de la intersección de las rectas.*

## Ejercicio 1.1: Estadísticas

En este apartado se tratará de mostrar los resultados y dar algunas explicaciones del por qué. Para empezar, vamos a mostrar un gráfico con los resultados, para un total de 33 imágenes de test:

Como se puede observar, los resultados son bastante buenos. Se han catalogado como aceptables los aciertos y aquellos que no se sitúan exactamente en el centro del coche pero que se han desviado muy ligeramente y que en definitiva reconocen el coche. Pero, sin embargo, a pesar de haber seguido los pasos correctos, aún hay algún fallo (aunque el número es casi despreciable). En concreto los dos casos que fallan son estos:

En la primera imagen, creemos que debido al ruido (en el sentido de que se ven varios coches), el algoritmo no es capaz de reconocer el coche objetivo, pero increíblemente, no sabemos si por suerte, ¡RECONOCE EL COCHE DE DETRÁS! A pesar de todo, al no ser el coche objetivo lo consideramos un fallo.

En la segunda imagen, es posible que la causa de que falle sea la mala calidad de la imagen. Además, al fondo, hay varias cosas prácticamente imperceptibles, lo cual puede hacer que se reconozcan puntos de interés que no son tal.

# Ejercicio 2: Detección de coches usando cv2.CascadeClassifier

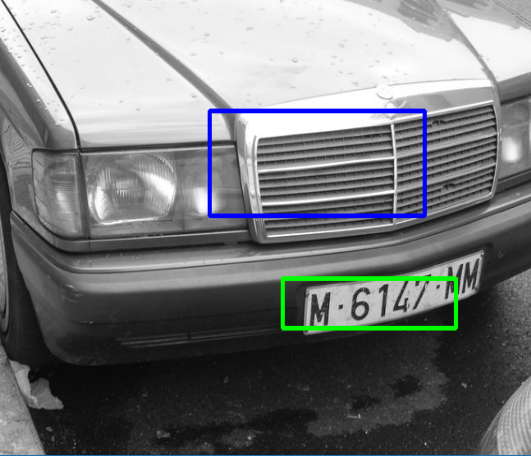
El objetivo de este ejercicio es desarrollar un algoritmo basado en el detector de P.Viola y M.Jones que detecte la posición de un coche en una imagen. Para ello se han proporcionado dos ficheros, coches.xml y matriculas.xml, que se corresponden con un clasificador de coches y de matrículas respectivamente. Una vez almacenados estos clasificadores, se utiliza la función *detectMultiScale* para detectar objetos de diferentes tamaños en cada una de las imágenes de test. Los parámetros de esta función son: **image**: la imagen en la que hay que detectar los objetos; **scaleFactor**: parámetro que especifica el cuánto disminuye el tamaño de la imagen en cada escala. Este parámetro cuanto más pequeño es, más preciso es el algoritmo, pero sin embargo se pierde algo de rendimiento (influye más en el siguiente ejercicio); **minNeighbors**: este parámetro es el mínimo número de vecinos que debe contener cada rectángulo. Además, en la detección de los frontales de coches, hemos utilizado un parámetro más para que nos sirva un poco de filtro, que es minSize: especifica el tamaño mínimo de los rectángulos. Nos sirve para descartar rectángulos pequeños que contengan varios descriptores. Una vez clasificada cada imagen dibujamos los rectángulos en éstas para ver qué se ha reconocido.

## Ejercicio 2.1: Estadísticas

En este apartado se exponen los resultados de la ejecución, se han clasificado las imágenes resultantes como aciertos: se detectan tanto el frontal como la matrícula; solo matrícula: aquellas imágenes en las que solo se detectan las matrículas; y solo frontales: aquellas imágenes en las que solo se detectan los frontales de los coches.

Aquí se expone un gráfico con los resultados:

Como se puede observar, los resultados son bastante buenos. Hay 27 aciertos, 3 casos en los que solo se detecta la matrícula y otros 3 casos en los que solo detecta el frontal.

Aquí se muestran las imágenes que fallan:

Donde se puede observar que en ninguno de los tres casos no se han detectado los frontales de los coches.

Donde se puede observar que no se reconocen las matrículas.

También se exponen algunos ejemplos del funcionamiento del algoritmo:

# Ejercicio 3: Detección del coche en secuencia del vídeo

El objetivo de este apartado es la detección del coche y de la matrícula del coche en dos secuencias de vídeo proporcionados. Para ello se sigue el mismo procedimiento que en el apartado anterior con los clasificadores proporcionados. Hay que conseguir dividir el vídeo en imágenes para poder tratar cada una de ellas por separado. Para ello se utiliza la función *videoCapture*, que divide el vídeo en cada uno de sus frames. Con los frames por separado se pueden reconocer tanto el frontal como la matrícula como en el caso anterior. Cabe destacar que el valor de **scaleFactor** de la función *detectMultiscale*  para el reconocimiento de matrículas tiene un valor bastante más alto que en el apartado anterior. Tiene una explicación: el rendimiento. En el primer vídeo no éramos capaces de reconocer la matrícula de ninguna manera, creemos que debido a la calidad del vídeo y a que ninguna de las tomas está lo suficientemente cerca como para que pueda reconocer los puntos de interés, y además la ejecución era muy lenta. Por lo tanto, decidimos subir el valor de este parámetro ya que para la segunda secuencia de vídeo funciona correctamente.

A continuación, se muestran algunos de los frames de los vídeos para comprobar el correcto funcionamiento.

