Proyecto Fin de Máster
Máster en Ingeniería Electrónica, Robótica y Automática

Desarrollo de algoritmos de visión por computador para la detección, localización y orientación de un vehículo no tripulado.

Autor: Víctor Ramos Amo
Tutores: Ignacio Alvarado Aldea
Mario Pereira Martín

Dpto. Ingeniería de Sistemas y Automática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2020

Proyecto Fin de Máster
Máster en Ingeniería Electrónica, Robótica y Automática

Desarrollo de algoritmos de visión por computador para la detección, localización y orientación de un vehículo no tripulado.

Autor:

Víctor Ramos Amo

Tutores:

Ignacio Alvarado Aldea

Profesor titular

Mario Pereira Martín

Dpto. Ingeniería de Sistemas y Automática

Escuela Técnica Superior de Ingeniería

Universidad de Sevilla

Sevilla, 2020

Proyecto Fin de Máster: Desarrollo de algoritmos de visión por computador para la detección, localización y orientación de un vehículo no tripulado.

Autor: Víctor Ramos Amo

Tutores: Ignacio Alvarado Aldea

Mario Pereira Martín

El tribunal nombrado para juzgar el Proyecto arriba indicado, compuesto por los siguientes miembros:

Presidente:

Vocal/es:

Secretario:

acuerdan otorgarle la calificación de:

El Secretario del Tribunal

Sevilla, 2020

A mis abuelos

A mis padres

A mis hermanas

A mis tíos

A todas las personas que me acompañaron en este camino

Agradecimientos

En primer lugar, a Mario y a Ignacio, por su inestimable dedicación, disponibilidad, ayuda y predisposición, siempre buscando lo mejor para el trabajo con la máxima exigencia, que me ha mantenido motivado en todo momento.

A mis profesores, por tener siempre la mano tendida durante mi trayectoria académica, tanto en Badajoz como en Sevilla, que hicieron todo cuanto estaba a su alcance por hacer que el camino más llevadero y supieron transmitir sus conocimientos, sembrando en mí una inquietud por la tecnología que hoy en día sigue creciendo a un ritmo frenético.

En el aspecto personal, a mis padres, hermanas y a toda mi familia, por el apoyo y la entrega total durante los años de estudio, que me han guiado hasta aquí. Gracias a ellos y a ellas hoy soy quien soy y puedo estar donde estoy. Mención especial, a toda mi familia scout, que también me ha visto crecer y a la que le debo parte de mi faceta más solidaria, que espero que a través de la tecnología pueda servir para ayudar a toda aquella persona que lo necesite.

Víctor Ramos Amo

Máster en Ingeniería Electrónica, Robótica y Automática.

Sevilla, 2020

Resumen

Se ha instalado en la Raspberry Pi 3 B+ el sistema operativo Raspbian con el que se han desarrollado los algoritmos en Python sobre Spyder3.

Se han instalado en Matlab las librerías Computer Vision y la app Image Labeler, con las que se trabajará.

Se han desarrollado algoritmos de almacenamiento y creación de datasets con imágenes para el entrenamiento de los detectores usados en Python.

Se han creado mediante la app Image Labeler tablas que contienen las bounding boxes con la información necesaria para crear un clasificador mediante un entrenamiento automático en cascada.

Se han creado detectores mediante entrenamiento en cascada para Python mediante la app Cascade Trainer GUI, con la que posteriormente se han testeado estos clasificadores.

Se han comparado las detecciones realizadas con los algoritmos en Python y en Matlab.

Se han desarrollado algoritmos de análisis de imágenes, así como de aplicación de detectores sobre las mismas.

Se han desarrollado algoritmos de análisis de archivos de vídeo y de vídeos en streaming para la detección de objetos mediante los detectores entrenados.

Se han desarrollado algoritmos de detección de umbral de color para hacer más robustas las detecciones de los clasificadores.

Se han aplicado, sobre los algoritmos de detección, una serie de operaciones que permiten entregar al usuario final la posición y el ángulo del Segway detectado.

Abstract

The operative system Raspbian has been installed on the Raspberry Pi 3 B+, where most of the algorithms have been developed on Python, helped by Spyder3.

Computer Vision packages and Image Labeler app have been installed on Matlab, both of which will be used later on.

Images dataset's storaging and creation algorithms have been developed to train the classifiers used on Python.

Tables containing bounding boxes information have been created through Image Labeler to create a classifier through an automatic cascade training.

Classifiers for Python have been created through cascade training using Cascade Trainer GUI, where later on, Those classifiers will be tested.

Detections made by Matlab's and Python's algorithms have been compared.

Image analysis algorithms have been developed, as well as algorithms that apply this classifiers on those images.

Video and streaming video analysis algorithms have been developed in order to detect ítems through the trained classifiers.

Color threshold detection algorithms have been developed to make detections even stronger.

Operations have been applied over detection algorithms that allows them to deliver to the user information about the position and the angle of the detected Segway.

Índice

ÍNDICE DE TABLAS

ÍNDICE DE FIGURAS

1 INTRODUCCIÓN

La visión por computador es una rama de la tecnología que engloba el proceso de adquisición, procesamiento, análisis y comprensión de imágenes tomadas del mundo exterior. Su aplicación es vital para el correcto funcionamiento de gran variedad de sistemas que están presentes en el día a día actualmente, como, por ejemplo, en el control de calidad de grandes cadenas de fabricación, en la automoción moderna o incluso en la videovigilancia.

Es importante destacar que las imágenes con las que se trabaja poseen una cantidad de información muy elevada, por lo que el correcto desarrollo de los sistemas de visión por computador cobra una gran importancia. Dentro de esta cantidad de información se encuentran las características que definen la imagen, como la textura o el color. Como se describirá más adelante, el color cobra una importancia vital en ciertos tratamientos de la imagen para facilitar la detección de elementos.

Por otro lado, se trabajará a lo largo del presente proyecto con un vehículo terrestre no tripulado controlado de forma remota, definido como Segway durante la redacción. Este vehículo se creó en un proyecto anterior del propio Departamento [1] con el objetivo de realizar diferentes tareas, tanto como elemento de investigación y desarrollo de tecnologías y aplicaciones nuevas de gran interés como de, en este caso, objeto a detectar con diferentes algoritmos.

Cabe destacar que son varios los vehículos que se desean identificar, pero dado que el método de detección es válido para un único vehículo, se asume que es aplicable para cualquier cantidad de ellos que se quiera utilizar.

El objetivo del presente proyecto será entregar información al vehículo que él mismo no puede adquirir de una forma del todo precisa, por esto cobra importancia la visión por computador. La información será recogida mediante una cámara en posición cenital que constantemente tendrá al vehículo dentro de su campo de visión. Esta información a la que se hace referencia es, más concretamente, la posición y el ángulo instantáneo de cada Segway. Estos datos son difíciles de conocer, pues el Segway está en constante movimiento y puede sufrir algún tipo de alteración en su posición debido a la interacción del ser humano, o por algún elemento que se encuentre en su entorno que limite su movimiento. En la Figura 1 se muestra el esquema general con la ubicación de la Raspberry, en el techo de la habitación y el Segway en movimiento por la superficie.

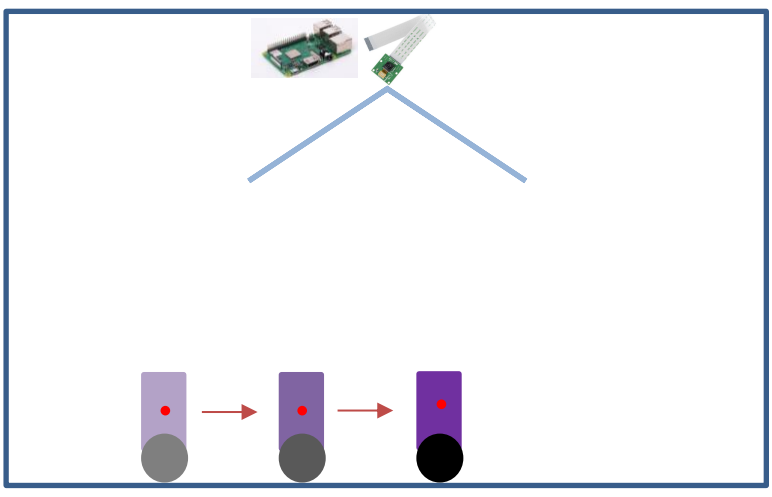


Figura 1. Corte transversal de la habitación con Raspberry y el Segway ubicados.

Mediante el uso de la visión por computador se buscará dotar de esa información al Segway y al usuario, de tal modo que se pueda emplear para realizar movimientos más precisos o para definir trayectorias conjuntas cuando entren en juego varios vehículos.

En este primer capítulo se sentarán las bases teóricas sobre las que se trabajará durante el proyecto, definiendo conceptos básicos del campo de la visión por computador y todos aquellos elementos necesarios para la comprensión del trabajo.

1.1. Pérdida de ubicación del Segway

El vehículo está compuesto por una Raspberry Pi 3B+ [2] que gestiona por completo su funcionamiento, así como de un módulo Bluetooth para controla su movimiento, un giróscopo que entrega información al procesador sobre la inclinación del robot, haciendo que se mantenga equilibrado y en posición vertical en todo momento mediante dos servomotores que ejercen la fuerza motriz necesaria para desplazar al Segway completo y corrigen cualquier fallo en esta inclinación.

En la Figura 2 se puede ver una imagen descriptiva del robot con todos los elementos mencionados con anterioridad.

Figura 2. Segway objeto del presente trabajo.

Como solución a este problema se plantea la detección por varios métodos basado en visión por computador del Segway de tal modo que se pueda acceder a la posición instantánea en la que se encuentra externamente, sin interactuar con él.

1.2. Aplicación de la visión por computador en el proyecto

La visión por computador es una herramienta de gran potencia y capacidad de aplicación que cada vez se emplea más en el entorno del transporte por tierra, tanto de personas como de mercancías, además de en entornos tanto industriales como no industriales. En este proyecto se utiliza esta tecnología para maximizar la precisión en la localización del Segway de forma instantánea. Con esto se conseguirá evitar el uso de tecnologías de mayor coste como módulos GPS que podrían no ser del todo útiles puesto que podrían tener errores en la medida al estar ubicados los robots en zonas interiores.

Para la programación de los diferentes algoritmos se ha empleado una Raspberry Pi 3B+, un ordenador de placa reducida de bajo coste muy versátil, pensada para casi cualquier tipo de aplicación informática ya que, gracias a su versatilidad, su software de código abierto y la posibilidad de instalarle un amplio abanico de sistemas operativos facilita el acceso al aprendizaje

de diferentes lenguajes de programación al mundo. Se muestra en la Figura 3 una imagen de la placa.

Figura 3. Raspberry Pi 3B+.

En este caso, el software empleado para el desarrollo de los algoritmos ha sido Spyder 3, instalado en la propia Raspberry y Matlab, instalado en otro dispositivo. Spyder [3] es un entorno de desarrollo interactivo para Python, lenguaje empleado en la mayor parte del proyecto, aprovechando la consola de depuración integrada y la comodidad que aporta. Matlab [4], por otro lado, es una aplicación de carácter académica que presenta numerosas ventajas, principalmente debido a su fácil portabilidad y la posibilidad de implementación del algoritmo creado en cualquier sistema, pues existen una gran cantidad de rutinas ya programadas y métodos de conversión del código creado a otros lenguajes diferentes. Además, dispone de un gran número de herramientas de visualización y de tratamiento de imágenes en las cuales se sustenta el presente proyecto.

Si se desean utilizar técnicas de visión por computador, con independencia del software con el que se esté trabajando, se deben tener en cuenta algunos aspectos fundamentales, como el proceso de adquisición de imágenes y el planteamiento utilizado para la resolución deseada del problema que se aborda. La solución a la primera cuestión planteada es simple: para la obtención de las imágenes con las que trabajar se ha utilizado el módulo de la cámara (versión 2) de 8 megapíxeles [5] mostrado en la Figura 4, haciendo uso de algoritmos desarrollados también en Spyder 3. La cámara está situada en posición cenital respecto al robot, con lo que este es visible en todo momento. De este modo se consigue también aumentar la estabilidad de la imagen, buscando adquirir la mejor imagen posible.

Figura 4. Módulo PiCamera V2.

Respecto a la resolución del problema se propone el siguiente método, mediante el cual, a partir de imágenes del Segway, se crea y entrena un detector en función de una serie de parámetros, que será el encargado de realizar la detección. Finalizado este proceso se sintonizará buscando la mejor de las detecciones posibles. Por último, se compararán los valores que se desprendan de estas evaluaciones para conocer cuál de los dos métodos es más indicado para la detección del Segway en tiempo real.

Para el desarrollo del presente proyecto se utilizarán varias librerías de Python, como *Numpy* o *Math*, para la realización de operaciones matemáticas, *argparse*, *imutils* o *collections* para el tratamiento de ciertas variables, pero la más importante y en la que se basará la práctica totalidad del proyecto será OpenCV [6], una librería de tratamiento de imagen muy extendida y de gran potencia. Además, en Matlab, se empleará su librería "Computer Vision" y también se recurrirá a la aplicación "Image Labeler". A continuación, se muestra en la Figura 5 el esquema general del trabajo.

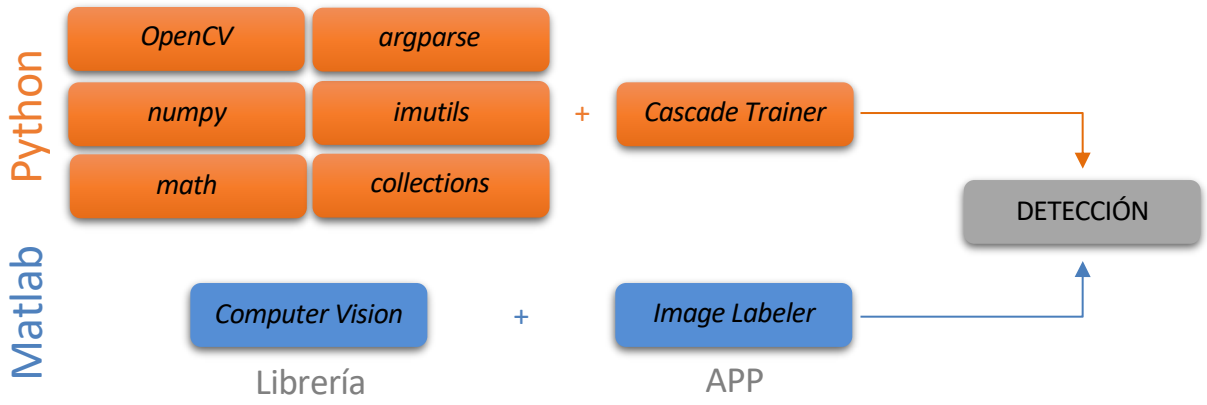


Figura 5. Esquema general de trabajo.

1.3. Objetivos

Tal y como se ha comentado al inicio de la presente sección, el objetivo del trabajo es la obtención de los puntos en los que se ubica el Segway.

Concretando lo anterior, se definen los siguientes objetivos específicos:

- Aplicar varios procedimientos de modelado y tratamiento de imágenes ampliamente extendidos como son Haar y HOG (Histograma de Gradientes Orientados).
- Aplicar sistemas de aprendizaje supervisado para realizar la fase de detección como las SVM (Máquinas de Vectores de Soporte).
- Desarrollar algoritmos de tratamiento de imágenes basados en la librería OpenCV de Python para una presentación de los resultados de fácil comprensión.
- Comparar los resultados de los procedimientos realizados con Python y Matlab.

1.4. Visión por computador

La visión por computador es una rama de la tecnología cuyo objetivo es adquirir, procesar, analizar y comprender imágenes tomadas mediante una cámara de tal forma que se pueda trabajar con los datos obtenidos en un ordenador. Es una disciplina que ha experimentado un gran auge en las últimas dos décadas.

A pesar de que nació a finales de los años sesenta, su punto de máxima evolución vino acompañado por el gran impulso vivido en el campo de la informática durante las últimas décadas del siglo XX, a lo largo del cual el desarrollo de los computadores sufrió un crecimiento exponencial, pasando de ser un artículo de lujo que solo podía ser adquirido por aquellas empresas con mayor poder económico y mejor posición dentro del mercado mundial, a ser un producto asequible que está presente en la mayoría de los hogares a día de hoy. Actualmente sigue siendo una de las áreas tecnológicas con mayor evolución y aplicación, como por ejemplo, en los actuales coches autónomos. Este hecho ha facilitado y propiciado en gran medida la creación de unas aplicaciones que anteriormente no eran más que ideas, llegando al punto de buscar imitar el órgano de la visión humana, tal como se muestra en la Figura 6.

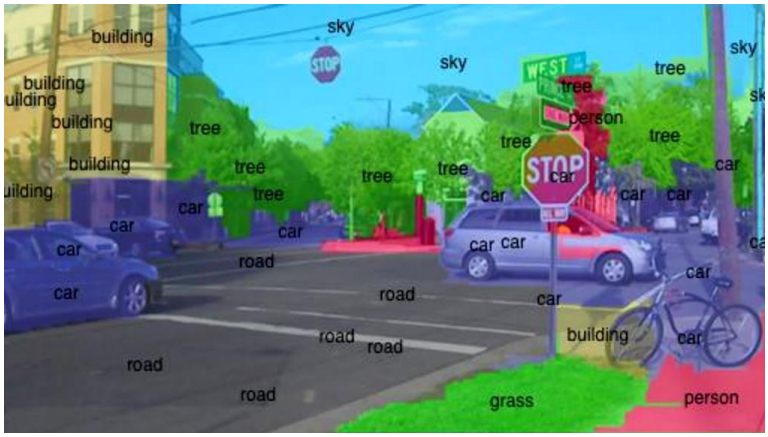


Figura 6. Ejemplo de una imagen analizada mediante visión por computador.

La posibilidad de reconocer y localizar los elementos presentes en una imagen ha sido un campo dentro de la inteligencia artificial que ha interesado al ser humano desde hace mucho tiempo. Es por ello que poco a poco se han desarrollado diversas técnicas y procedimientos con el objetivo de lograr reconocer estos objetos o incluso personas en una imagen del mejor modo posible, pero ¿De dónde viene la técnica del reconocimiento de objetos?

En sus inicios, la visión por computador se basaba en imágenes binarias, compuestas solo por blancos y negros, que eran procesadas normalmente por píxeles, aunque también estos se podían llegar a tratar por agrupaciones, en bloques o en ventanas [7].

Poco a poco fue desarrollándose esta tecnología, hasta lograrse localizar el objeto en la imagen y registrar su posición. A pesar de esto, aún se precisaba de una mayor evolución, ya que existían problemas cuando la intensidad de la luz se reflejaba en la imagen, generando regiones de píxeles con niveles de grises diferentes a los reales.

Para esto se introdujeron sistemas que fuesen capaces de medir estas intensidades de grises, con lo que se proporcionaría a cada píxel un valor proporcional al nivel de gris que poseía. Gracias a esto se consiguió que los sistemas de detección funcionasen con cualquier nivel de gris, ya que los bordes de los objetos eran detectados al localizar variaciones en los niveles de intensidad de los

píxeles.

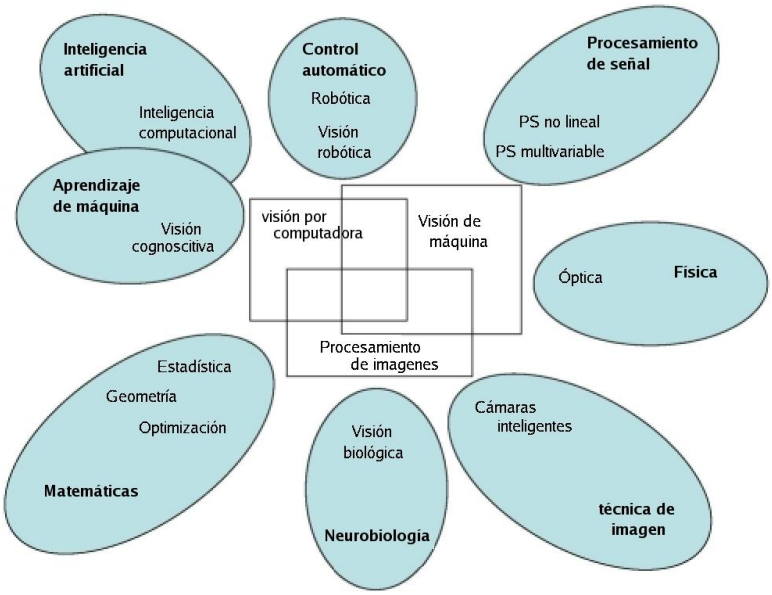


Figura 7. Ramas de la visión artificial.

En la actualidad, estos sistemas requieren de computadores de gran potencia que sean capaces de procesar una gran cantidad de datos a alta velocidad, de tal modo que se realicen aproximadamente mil millones de cálculos por segundo sobre la imagen. Los sistemas de visión por computador hacen uso de técnicas de reconocimiento de imágenes de tal modo que se compara la misma con una base de datos en busca del objeto que se desea localizar en la misma. Esto hace mucho más accesible el uso de las técnicas nombradas hasta el punto de poder ser aplicada en los campos mostrados en la Figura 7.

El principio de funcionamiento al que se reduce la visión por computador es similar al de la visión humana, sustituyendo los órganos encargados de la misma por una cámara, cuya tarea será tomar las imágenes y un ordenador que se encargará de procesar la intensidad de la luz presente en las mismas. De este modo se logrará rescatar información muy valiosa con la que se podrá trabajar en función de lo que se detecte.

Dentro de la visión por computador se pueden diferenciar con claridad varias fases por las que es necesario pasar para obtener la mejor detección de la imagen posible y con ello la mejor respuesta. En la Figura 8 se presenta un esquema que enumera y enlaza las etapas seguidas en el procesamiento de una imagen utilizando la visión por computador.

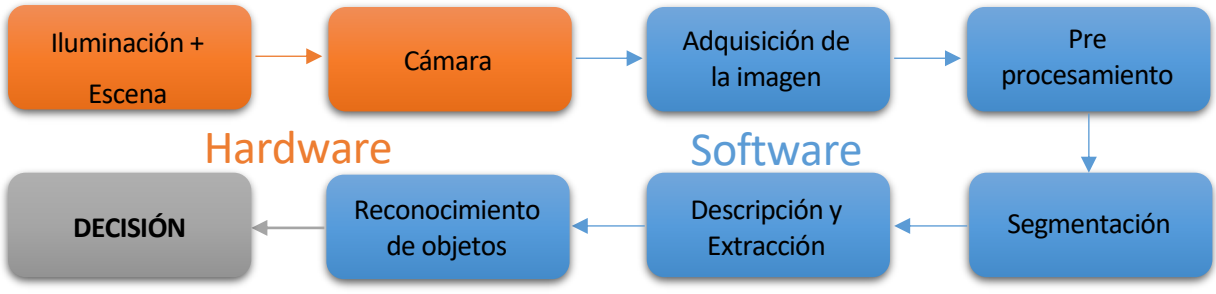


Figura 8. Esquema de funcionamiento de la visión por computador.

Para empezar, es necesario **adquirir la imagen**, resultado de la obtención a través de la cámara de las intensidades de luces y reflejos comentados con anterioridad. Después se debe **refinar** la misma con un proceso que se conoce como preprocesamiento, mediante el cual se mejora la imagen y con ella las fases posteriores por las que debe pasar.

Para finalizar este primer grupo de etapas se **extraen** de la imagen las **características singulares**, es decir, aquellas cuyo estudio puede resultar interesante para una fase posterior del proceso de detección. Estas características van desde la detección de bordes hasta la detección de movimiento, pasando por la detección de texturas o la detección de esquinas o puntos angulosos.

En el segundo conjunto de etapas se agrupan aquellas que poseen un mayor grado de complejidad. El primer paso será la **segmentación**, con la que se conseguirá separar los elementos de la imagen para diferenciarlos.

Posteriormente se **eliminará el ruido** de la imagen, producido en la fase de segmentación y se pasará a la fase de **reconocimiento**, en la que se buscarán los objetos conocidos presentes en la escena, para pasar a su **localización**, es decir, dónde y cómo se muestran estos objetos. La etapa final y que encierra una mayor complejidad es la de la **interpretación de la escena**, en la cual se define el conjunto completo de la imagen tomada y las detecciones señaladas en ellas.

El presente proyecto evaluará el funcionamiento de dos métodos de detección en imágenes utilizando el reconocimiento de formas como principal característica a detectar, aunque como se comentará con posterioridad, las posibilidades en cuanto a la selección del método de detección son muy elevadas.

Si a la técnica de identificación elegida se le añade una red neuronal potente, se podría llegar a obtener una máquina capaz de reconocer objetos y/o rasgos faciales mejor que cualquier persona.

1.5. Conceptos importantes

A continuación, se describirán una serie de conceptos que se utilizarán a lo largo del presente documento y en los que se irá profundizando [8] [9].

- **Característica:** es aquel elemento singular de la imagen de entrada que puede ser detectado. La visualización y detección de este elemento se podría mejorar, por ejemplo, optimizando la región de entrada para que el elemento singular sea lo más parecido posible al que se busca detectar o entrenando a la máquina con un conjunto de imágenes que contengan dicho elemento singular con una variación mínima.

- **Region Of Interest (ROI):** sector de la imagen que contiene información relevante a cerca del elemento a detectar. A la hora de realizar la búsqueda de coincidencias, serán éstas las regiones que se buscarán.

- **Candidato:** se define como candidato aquella característica de la imagen que ha sido detectada en la imagen de entrada. Estos quedan recogidos dentro de las **bounding box**.

- **Ventana:** esta es la región de la imagen de entrada que contiene el elemento singular a detectar. Para realizar la detección y la posterior selección de uno o varios candidatos, se desliza una ventana de tamaño canónico por toda la imagen de entrada de arriba hacia abajo y de izquierda a derecha. Una vez coincida la característica buscada, extraída de las ROIs, con una región de la imagen, se generarán ventanas que encierren a uno o varios

candidatos, así como nuevas regiones de interés que posteriormente serán procesadas. Aquellas ventanas que contengan candidatos se denominan **bounding box**.

- **Pirámide:** conjunto que contiene la imagen de entrada reescalada y suavizada en diferentes tamaños sobre los cuales deslizará la ventana canónica buscando las ROIs predefinidas manualmente. El tamaño mínimo de la imagen reescalada debe ser superior al de la ventana canónica.

- **Detector o clasificador:** es el elemento encargado de la detección de las características en la imagen de entrada.

- **Ground Truth:** concepto que se refiere al resultado correcto que debería dar un clasificador. Cruzando estos con los resultados de la clasificación se obtiene una **matriz de confusión**, en la que se agrupan cuatro tipos de elementos:

  - Reales positivos: son aquellas detecciones correctas, las que se buscan.

  - Falsos negativos: aquí se engloban los elementos correctos no detectados que han sido descartados por el detector.

  - Falsos positivos: estos serán los elementos detectados de forma incorrecta.

  - Reales negativos: son los elementos incorrectos no detectados y descartados.

  A modo de ejemplo, se muestra la Figura 9, en la que se define gráficamente la matriz de confusión obtenida al tratar de detectar personas en la imagen de la derecha.

Figura 9. Matriz de confusión.

De la matriz de confusión se desprenden los conceptos de **exactitud**, **precisión**, **sensibilidad** y **especificidad:**

- **Exactitud:** se define como la suma del número de reales positivos más los reales negativos dividido entre las predicciones totales, obteniendo un valor de la proximidad entre el resultado y la clasificación exacta que se busca.

- **Precisión:** hace referencia a la calidad de la respuesta del clasificador y se define como el número de reales positivos dividido entre la suma de los reales positivos y los falsos positivos.

- **Sensibilidad:** se refiere a la eficiencia de la clasificación de los elementos a buscar y se calcula como el número de reales positivos dividido entre la suma de los reales positivos y los falsos negativos.

- **Especificidad:** por último, la especificidad indica la eficiencia en la clasificación de los elementos que no se buscan, es decir, el número de reales negativos dividido entre la suma de los reales negativos y los falsos positivos.

Todas estas variables toman valores entre el cero y el uno, en tanto por uno, siendo un mejor clasificador aquel con mayores valores de ellas.

Existen diversos modos de adquisición, procesamiento y análisis de las imágenes. A continuación, se definirán los utilizados en este trabajo y se explicará el porqué de su elección frente a los otros.

1.6. Histograma de gradientes orientados (HOG)

Se define detección de un objeto como el procedimiento mediante el cual, a través de una serie de instrucciones previamente entregadas a la máquina, esta localiza dentro de cada una de las imágenes la o las características iguales a las predefinidas. Dentro de este proceso podemos diferenciar dos partes, una primera, de **extracción de características** y posteriormente, la **búsqueda de objetos con esas características**.

Como se ha presentado en la sección 1.4, el proceso de detección está dividido en varias etapas que deben seguirse en el riguroso orden especificado. En este apartado se procede a definir más puntualmente los pasos a seguir durante este proceso.

1.6.1 Extracción de características

La **extracción de características** es el proceso mediante el cual se obtiene un conjunto de datos matemáticos, estructurados de tal manera que condensen el contenido de la imagen de entrada leída para que el proceso de búsqueda de estas características sea lo más rápido y eficaz posible. Estos conjuntos se definen como **descriptores**.

Estos descriptores pueden variar en función del tipo de reconocimiento y del momento de la detección. Los descriptores que serán utilizados a lo largo del presente proyecto serán de tipo HOG (Histogram of Oriented Gradients), gracias a los cuales se puede diferenciar una variación de textura o color en la imagen como puede verse en la Figura 10 mostrada a continuación.

Los descriptores HOG, para aumentar la precisión de los datos tomados de una imagen y condensar la información extraída de la misma, dividen primero la imagen en cuadrículas del mismo tamaño en las que se representan los datos más relevantes del sector analizado una serie de vectores caracterizados por su magnitud y dirección.

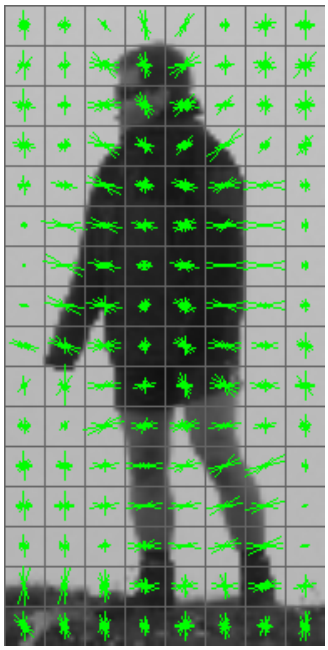


Figura 10. Matrices de gradientes HOG obtenidas de la imagen.

Estos vectores son conocidos como **gradientes**, los cuales, en función de su orientación, quedan divididos en conjuntos o intervalos de 30˚ aproximadamente. Existen casos en los que estos intervalos recogen los gradientes a lo largo de los 360˚posibles y otros que los aúnan en 180˚, en función de si se toma en consideración el sentido del vector o no. El uso del valor de 30˚ viene de ser aquel valor de orientación del gradiente que ha aportado mejores resultados en experimentaciones anteriores al desarrollo de este trabajo. Cada uno de los gradientes posee unos valores de magnitud y orientación y en función de ellos, un peso específico en estas agrupaciones, pudiendo aportar información a dos intervalos a la vez. Esto queda reflejado en la Figura 11.

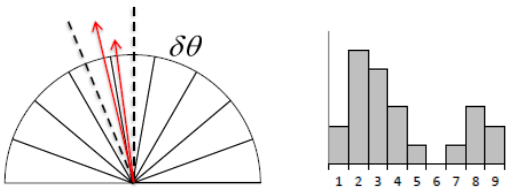


Figura 11. (a) Representación gráfica de los gradientes. (b) Ejemplo de histograma obtenido.

Cabe destacar que los gradientes con sentidos opuestos se agrupan en los mismos intervalos, dado que su orientación es igual. A pesar de contar con nueve divisiones en el sector de 0 a 180º, un gradiente perteneciente a uno de los intervalos puede contribuir a los intervalos vecinos en función de su valor. Se definen las ecuaciones (1) a las que se recurre para obtener dicho valor.

$$\omega_k(x,y) = \max\left(0, 1 - \frac{\theta(x,y) - \theta_k}{\delta\theta}\right) \qquad h(k) = \sum_{(x,y)\in C} \omega_k(x,y) g(x,y) \qquad (1)$$

Siendo $\omega_k(x, y)$ el valor del gradiente recogido dentro del intervalo *k*, de tal modo que se toma el valor de 0 si la posición del gradiente se encuentra muy alejado del intervalo vecino y un valor ponderado del gradiente en función de la distancia a la que se encuentre del mismo.

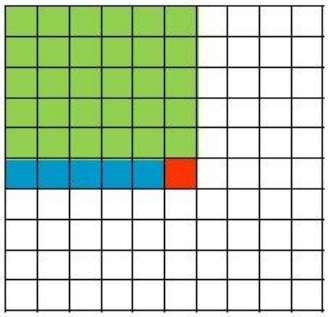
De este modo se pasa de tener una imagen a un conjunto de datos agrupados que proporcionan la misma información, con la salvedad de que con la imagen sin tratar no se puede trabajar. Estas matrices, cuyos elementos son histogramas, pasan a ser tratadas por separado en función del número de intervalos elegidos.

Para la creación de un detector robusto, se reúnen las celdas en bloques, definidos por la concatenación de los histogramas perteneciente a esas celdas en forma de vector (v), de tal modo que un histograma aporte información a varios bloques. Además, antes de la creación del detector se normaliza el bloque con el objetivo de que el detector final no se vea afectado por variaciones en la intensidad o color de la imagen de entrada. Para el proceso de normalización se aplica la ecuación (2):

$$\boldsymbol{v}' = \frac{\boldsymbol{v}}{\sqrt{\|\boldsymbol{v}\|^2 + \boldsymbol{\varepsilon}}} \qquad (2)$$

Con este procedimiento se independiza el valor de los bloques de esas variaciones en la iluminación de la imagen de entrada recogidas en los histogramas.

Otro método de definición del descriptor HOG sería la utilización de la imagen integral, en la que cada píxel de la imagen pasará a ser definido como suma del valor de todos los píxeles situados a su izquierda y por encima de él, de tal modo que el valor de cada píxel queda definido según se ilustra en la Figura 12:



$$II(x, y) = II(x, y-1) + s(x, y)$$

$$s(x, y) = \sum_{x'<=x} I(x', y) = s(x-1, y) + I(x, y)$$

Figura 12. Definición de Imagen Integral.

En este caso el píxel de color rojo toma el valor de *II(x,y)*, en el que aparecen sumados los términos *II(x,y-1)*, que indica la suma de los valores de los píxeles de color verde y *s(x,y)*, que define la suma de los valores de los píxeles en color verde, más el píxel en cuestión, coloreado de rojo.

Con esto se obtiene un nuevo valor específico para cada uno de los píxeles de la imagen en función de su valor del gradiente, así como de una región de la misma. Este dato en cuestión es utilizado para obtener coincidencias con la ventana deslizante seleccionada. Gracias a la imagen integral es posible definir el valor de una región de la imagen según se muestra en la Figura 13

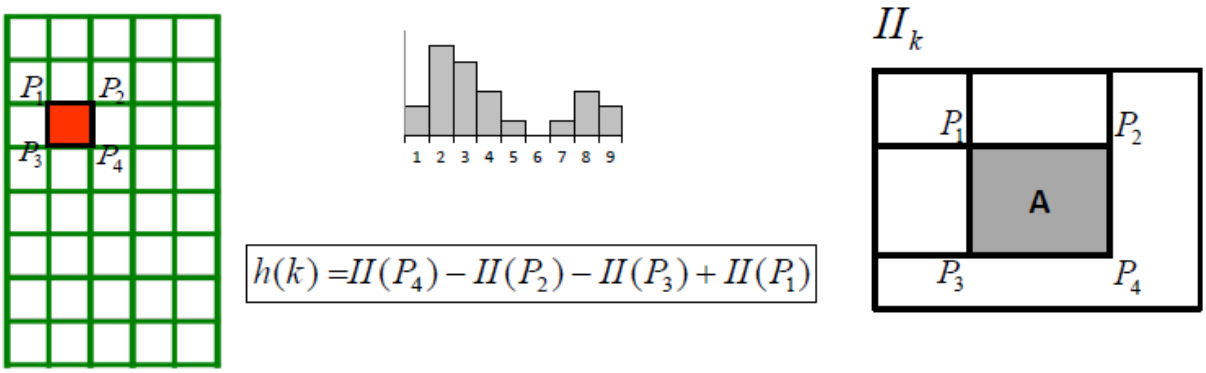


Figura 13. Valor de una región de celdas A definida por P_1, P_2, P_3 y P_4.

Según se aprecia en la figura anterior, *h(k)* indica el valor de la región A definida por la suma de los píxeles, en este caso histogramas, que quedan contenidos dentro de ella. Este valor es la suma de los valores situados a la izquierda y por encima del píxel P_4, al cual se le deben restar las zonas blancas definidas a la izquierda y por encima de los píxeles P_3 y P_2 respectivamente. La variable *k* indica el intervalo con el que se está trabajando, obteniendo un valor diferente para cada uno de los definidos, por lo que, al haber nueve intervalos, existirían nueve imágenes integrales.

Una vez se han obtenido los valores de la región definida para cada uno de los intervalos, se procederá a desplazar la ventana canónica por la pirámide de imágenes, en busca de alguna coincidencia.

1.6.2 Búsqueda de coincidencias

La **búsqueda de objetos con las características deseadas** se realiza mediante unos clasificadores o detectores. Una vez obtenidos los histogramas que definen la imagen, estos clasificadores se encargarán de diferenciar unas imágenes de otras mediante un proceso por el cual cada elemento a detectar se asemejará a un punto y cada elemento a descartar por otro, formando toda la imagen una nube de puntos.

Para seleccionar los elementos que realmente se buscan, se utilizan los detectores mencionados anteriormente. Estos se pueden asemejar a una división, llamada hiperplano, de la nube de puntos, con mayor o menor grado de exactitud en función del método utilizado para su obtención, que separa los elementos a detectar de los que se desean descartar, definidos como vectores de clase.

Estos clasificadores, pueden tener una cierta holgura, es decir, la división que realicen puede no ser perfecta. No obstante, la definición de los hiperplanos se realiza de forma que para su creación se apoya en los llamados vectores de soporte, que son aquellas detecciones, tanto aceptadas como descartadas, que quedan más próximas entre sí. Para entender mejor esto solo hace falta fijarse en los puntos azules y naranjas de la Figura 14.

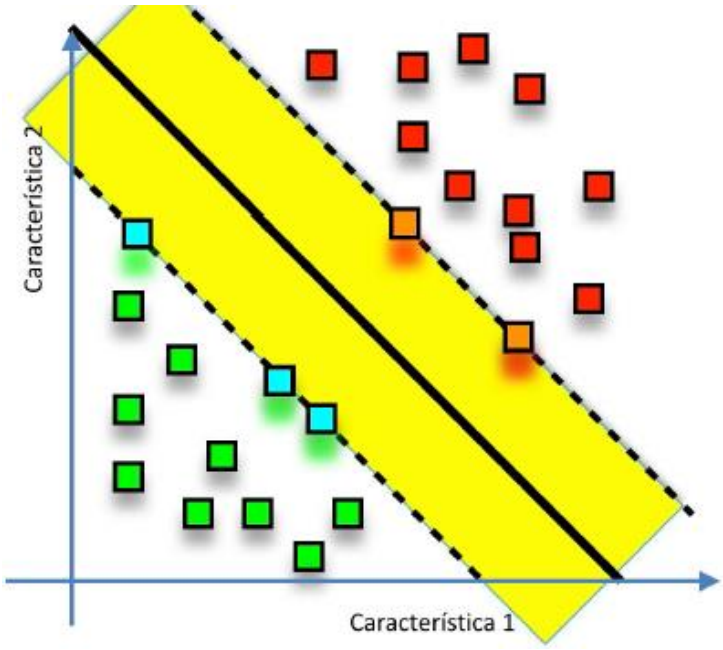


Figura 14. Ejemplo de un clasificador Support Vector Machine (SVM) y su frontera (hiperplano).

Este clasificador no se vería afectado si la posición de los vectores de clase cambia, sin embargo, sí que se modificaría si la que cambia es la posición de los vectores de soporte.

Para el cálculo de estos hiperplanos se utilizará una formulación matemática específica que resuelve esta cuestión como si de un problema de optimización se tratase.

Posteriormente, dada una detección, se va a comparar el solapamiento de la misma con el *ground truth* y se validarán solo aquellas detecciones cuyo grado de solapamiento supere un cierto umbral, cuyo valor común es 50%. El valor de este grado de solapamiento se obtiene de la división del área de la intersección entre el área de la unión de la detección con el *ground truth*.

Algunas medidas que proporcionan un nivel de calidad del detector son:

- **Tasa de detección:** valor obtenido de la división del número de reales positivos entre objetos en el *ground truth* que da una idea de la calidad del detector.
- **Tasa de error:** complementario de la tasa de detección.
- **Falsos positivos por imagen:** se define como el número de detecciones incorrectas por imagen y su valor viene dado por el número de falsos positivos entre el número de imágenes en el *ground truth.*

A pesar de todo este proceso, es posible que persistan los problemas, puesto que una imagen puede ser detectada de una manera muy diferente con un simple cambio en la intensidad de la luz o de posición.

1.7. Haar-features

Además del clasificador HOG se va a utilizar también uno tipo Haar, que también sigue los mismos procedimientos explicados al inicio del epígrafe 1.6: **extracción de características** y **búsqueda de coincidencias** los cuales se explicarán ahora con más detalle.

1.7.1 Extracción de características

Así como el clasificador HOG utiliza los gradientes para la creación del detector, el clasificador Haar utiliza unas características singulares, mostradas en la Figura 15. Gracias a ellas, mediante unas sumas y restas de los píxeles bajo las regiones negras/blancas, se crea el clasificador Haar.

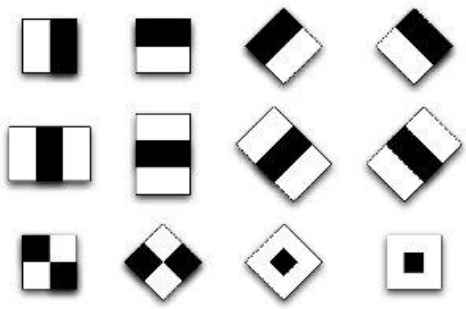


Figura 15. Descriptores Haar para la detección.

De este modo, de arriba hacia abajo, pueden diferenciarse tres tipos de descriptores:

- En la línea superior se encuentran los descriptores de **bordes**. Se caracterizan por tener únicamente dos regiones, buscando un gradiente completo en la imagen. Los dos primeros son para bordes verticales y horizontales y los dos siguientes son para bordes girados, de tal modo que todos quedan descritos y clasificados correctamente.
- La línea intermedia la ocupan los descriptores de **líneas**. Destacan por contener tres franjas, siendo iguales las de los extremos y la central diferente. Igual que en el caso anterior, las verticales y horizontales van acompañadas de las giradas.
- Por último, se encuentran los de **diagonales**. Éstos están formados por cuatro regiones en las que las diagonales poseen el mismo color, con lo que la diferencia se calcula entre los pares diagonales.

Para la clasificación se desliza cada uno de los descriptores por las imágenes buscando coincidencias en forma de cascada, una tras otra, obteniendo detecciones como las de la Figura 16.

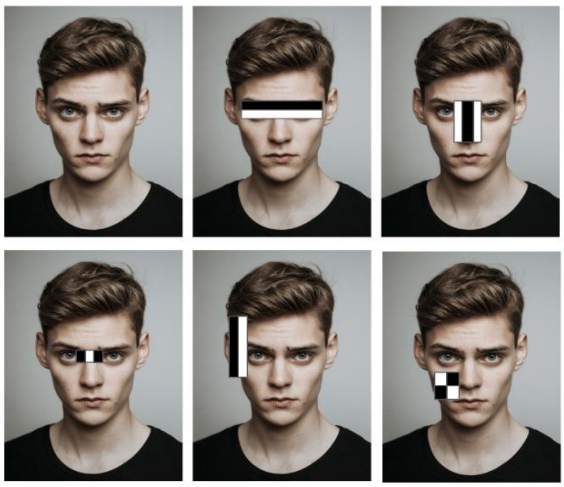


Figura 16. Ejemplos de detecciones Haar.

Para poder trabajar con tal cantidad de detecciones se hace necesario un método que simplifique el proceso. Es por esto que en la detección Haar también se hace uso de las **imágenes integrales**, explicadas al final del epígrafe 1.6.1. De este modo, de cada imagen en escala de grises se obtiene un valor ponderado por cada región, obtenido según el procedimiento descrito en el epígrafe mencionado.

1.7.2 Búsqueda de coincidencias

Para el proceso de búsqueda de características el detector Haar hace uso de una herramienta que permita gestionar, operar y comparar los resultados obtenidos gracias a la imagen integral. Esta herramienta es conocida como **Adaboost** [10] [11].

El procedimiento a seguir es similar al caso de los gradientes orientados. Se obtienen valores de las diferentes regiones de todas las imágenes de entrenamiento y se trata de buscar coincidencias o zonas de interés en cascada. En función de estas coincidencias se clasificarán las detecciones como positivas o negativas, con lo que aparecerán errores en las clasificaciones, por lo que se seleccionan las características con tasas de errores mínimas, que serán las que mejor definan la característica a buscar.

Al principio del entrenamiento todas las imágenes tienen el mismo peso, pero este va variando según las fases de clasificación por las que pasen, en función del número de clasificaciones correctas y las tasas de error. Estas dos variables son calculadas después de cada fase de forma repetitiva hasta que se alcanza el valor mínimo buscado, definido anteriormente como tasa de falsos positivos.

Aquí es donde Adaboost entra en juego, tomando los valores entregados por estos clasificadores débiles para tomar una decisión final sobre el conjunto de la clasificación. Los clasificadores débiles se diseñan de tal modo que sean capaces de seleccionar una característica concreta de cada imagen, tomando decisiones simples. Al formar una cascada, las decisiones del clasificador anterior son tenidas en cuenta por el clasificador posterior, de tal modo que se va reduciendo poco a poco el error. Esta representación gráfica se puede ver en la Figura 17.

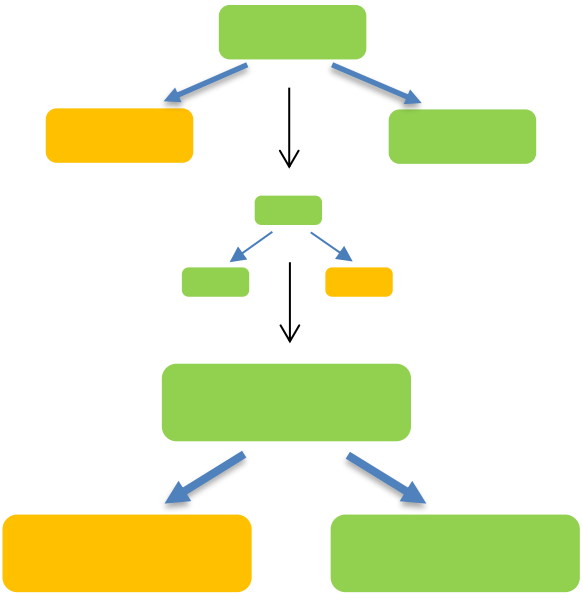


Figura 17. Representación gráfica del entrenamiento en cascada (Adaboost)

Cada vez que se finaliza una fase de clasificación se ajustan los pesos de cada clasificador, para mejorar la respuesta que genera, con lo que se consigue, poco a poco, mejorar tanto el clasificador como su respuesta. De este modo, la respuesta general del clasificador sería mejorada poco a poco hasta alcanzar el objetivo deseado y definido por el usuario.

Descrito matemáticamente, para empezar, en la ecuación (3) se define el conjunto de entrada, donde ***n*** es el número de atributos en el conjunto, en este caso uno; ***x*** es el conjunto de entrada; ***y*** es la variable objetivo que puede ser -1 o 1 al tener un caso de clasificación binaria.

$$\boldsymbol{x_i} \in \mathbb{R}^{\boldsymbol{n}}, \boldsymbol{y_i} \in \{-\mathbf{1}, \mathbf{1}\} \tag{3}$$

A continuación, se consideraría cada imagen con su peso (ω) correspondiente, definido en función del número de imágenes totales (N) en la ecuación (4):

$$\boldsymbol{\omega} = \frac{\mathbf{1}}{\boldsymbol{N}} \in [\mathbf{0}, \mathbf{1}] \tag{4}$$

Es evidente que la suma de los pesos de todas las imágenes es la unidad, por lo que los valores de estos deben estar siempre entre 0 y 1. Pero este peso debe ser actualizado periódicamente como se describirá más adelante.

Otra variable interesante es α, pues indica la influencia de cada clasificador en la clasificación final, y se define como (5):

$$\boldsymbol{\alpha_t} = \frac{\mathbf{1}}{\mathbf{2}} \frac{\boldsymbol{ln}(\mathbf{1}-\boldsymbol{ErrorTotal})}{\boldsymbol{ErrorTotal}} \tag{5}$$

El *ErrorTotal* es la suma del total de detecciones incorrectas divididas entre el tamaño del conjunto. Si se muestra el valor de α frente a la tasa de error (que toma siempre un valor comprendido entre 0 y 1) se obtiene una gráfica similar a la mostrada en la Figura 18:

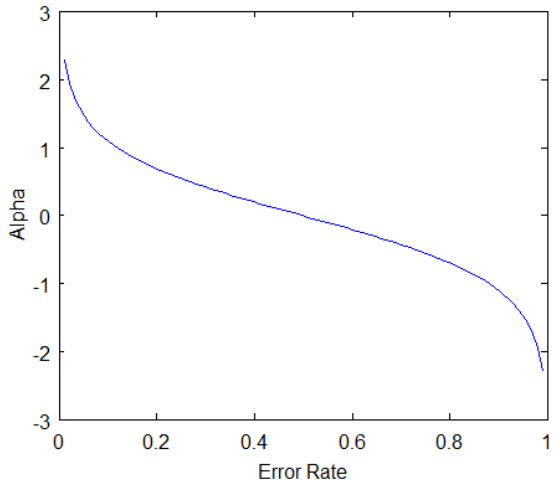


Figura 18. α frente a la tasa de error.

Se puede apreciar cómo, si destaca un número de clasificaciones erróneas bajo, o cercano a 0, el valor de α sería positivo y más alto cuanto más cercana a 0 sea la tasa de error del clasificador. Evidentemente, si la tasa de detecciones erróneas es muy alta, el valor de α será negativo, con lo que contribuirá negativamente al peso de ese clasificador débil.

Los pesos de los clasificadores, como se ha comentado anteriormente, se actualizan periódicamente. Esa actualización viene definida por el peso anterior del clasificador y por el valor de α que se acaba de explicar y se muestra en la ecuación (6):

$$\boldsymbol{\omega_i = \omega_{i-1} * e^{\pm\alpha}} \quad (6)$$

Es decir, un valor de alto en la tasa de error provoca que el peso del clasificador débil se reduzca, haciendo que su influencia, en este supuesto, errónea, sobre el clasificador total disminuya. Por lo tanto, se estaría mejorando su respuesta.

A continuación, se pasará a describir el proyecto en mayor profundidad y detalle, comenzando por breve capítulo en el cual se definirán los procesos seguidos hasta la creación del vehículo.

2 DESCRIPCIÓN DEL SEGWAY

El Segway objeto del presente trabajo fue desarrollado durante la consecución del proyecto de fin de grado del alumno Nicolás Cortés Fernández, titulado "Diseño, fabricación, montaje, estudio dinámico, control y teleoperación de un vehículo tipo péndulo invertido sobre dos ruedas".

En este capítulo se definirá de una forma breve el trabajo llevado a cabo por Nicolás, que ha servido como base para el desarrollo del presente documento.

2.1. Modelado dinámico

Como primer paso previo a la creación, diseño y fabricación del vehículo se realizó un estudio teórico del modelo en cuestión, basándose en el Teorema de Cantidad de Movimiento y en el Teorema del Momento Cinético, los cuales enuncian lo siguiente:

- **Teorema de Cantidad de Movimiento:**

"La derivada respecto al tiempo de la cantidad de movimiento de un sistema de masas es igual al sumatorio de las fuerzas externas actuando sobre él. En otras palabras, si sobre un sistema de partículas no actúa ninguna fuerza externa, su cantidad de movimiento permanece constante."

Descrito matemáticamente:

$$\frac{d\vec{C}}{dt} = \frac{d(\iiint \rho \vec{v} dV)}{dt} = \frac{d(M_2 \overrightarrow{v_{21}^G})}{dt} = \sum_{i=1}^{n} \overrightarrow{F_{l_{ext}}} \quad (7)$$

Donde $\vec{C}$ es la cantidad de movimiento, ρ la densidad de la masa, dV el diferencial del volumen, M_2 la masa total del sólido, $\overrightarrow{v_{21}^G}$ la velocidad del centro de masas del sólido 2 respecto al origen 1 y $\sum_{i=1}^{n} \overrightarrow{F_{l_{ext}}}$ el sumatorio de fuerzas externas que actúan sobre el sólido 2.

- **Teorema del Momento Cinético:**

"La derivada con respecto al tiempo del momento cinético en un sistema de masas respecto de un punto geométrico A, es igual al sumatorio de los momentos provocados por las fuerzas externas al sistema respecto de A, más el producto vectorial de la cantidad de movimiento y la velocidad del punto sobre el que se toman los momentos A."

Descrito matemáticamente:

$$\frac{d\overrightarrow{L_A}}{dt} = \sum_{i=1}^{n} \overrightarrow{M_{Al}^{ext}} + \vec{C} x \dot{\overrightarrow{O_1 A}} \quad (8)$$

$$\overrightarrow{L_A} = \overleftrightarrow{I_A} \overrightarrow{\omega_{21}} + M_2 \overrightarrow{AG_2} x \overrightarrow{v_{21}^A} \quad (9)$$

Donde $\overrightarrow{L_A}$ es el momento cinético respecto del punto A, $\overrightarrow{O_1\dot{A}}$ la velocidad de ese punto respecto al origen 1, $\sum_{i=1}^{n} \overrightarrow{M_{Al}^{ext}}$ el sumatorio de momentos externos respecto del punto A actuando sobre el sistema y $\overleftrightarrow{I_A}$ la matriz de inercia calculada en el punto A.

Tomando como base estos teoremas y aplicándolos sobre el caso del vehículo en cuestión, tratándolo como un péndulo invertido sobre dos ruedas, se desarrolló el marco teórico que sustentó el resto de las tareas para la correcta consecución del desarrollo del vehículo.

Como punto final del marco teórico se calcularon de forma precisa con ayuda de Matlab las variables restantes, quedando perfectamente definidos los parámetros y variables a utilizar.

2.2. Diseño electrónico y estructural

La segunda parte del trabajo se centró en el diseño y montaje de la electrónica del Segway, así como de su apariencia física y su diseño estructural.

El sistema se diseñó para estar formado por diferentes elementos, tales como un Arduino UNO, (que en un futuro se sustituyó por una Raspberry Pi 3b+) para la monitorización, gestión y control del resto de componentes mediante de diferentes funciones desarrolladas en el trabajo, dos motores paso a paso para el desplazamiento, un módulo inercial para controlar en todo momento el equilibrio, una batería para alimentar el sistema. Además, se utilizaron también unos drivers para los motores y un divisor de tensión para medir la carga de la batería.

El resultado, después de crear el chasis con una cortadora láser se muestra en la Figura 19:

Figura 19. Vista del prototipo del vehículo.

El siguiente paso fue implementar dos controladores PID, uno para el ángulo respecto a la vertical de vehículo y otro para la velocidad, de tal modo que se pueda actuar sobre la velocidad de las ruedas para corregir uno de estos dos parámetros y conseguir que el vehículo mantenga la estabilidad.

Finalmente se abordó el control mediante bluetooth a través del joystick de un un mando tipo Nunchuck de la consola Nintendo Wii como el que se muestra en la Figura 20.

Figura 20. Mando empleado en el control del Segway.

Cabe destacar que la apariencia del vehículo cambió posteriormente, de tal modo que el chasis actual está impreso mediante una impresora 3D con un filamento en color morado, por lo que ese es el color que define al Segway en la actualidad. Aparece el diseño final en la

Figura **21.**

Este dato cobra mucha importancia puesto que algunos de los algoritmos desarrollados se apoyarán en el color del mismo para su detección, como son los encontrados en los apartados h) e i) del capítulo Anexos. Este punto puede adquirir gran importancia si se desea, pues, como se podrá leer en el epígrafe 6.1, una de las propuestas de mejora del proyecto es la detección de vehículos de varios colores, por ello, para el correcto funcionamiento de los algoritmos se deben crear unas variables que guarden un umbral determinado para el color, adaptado a cada uno de ellos.

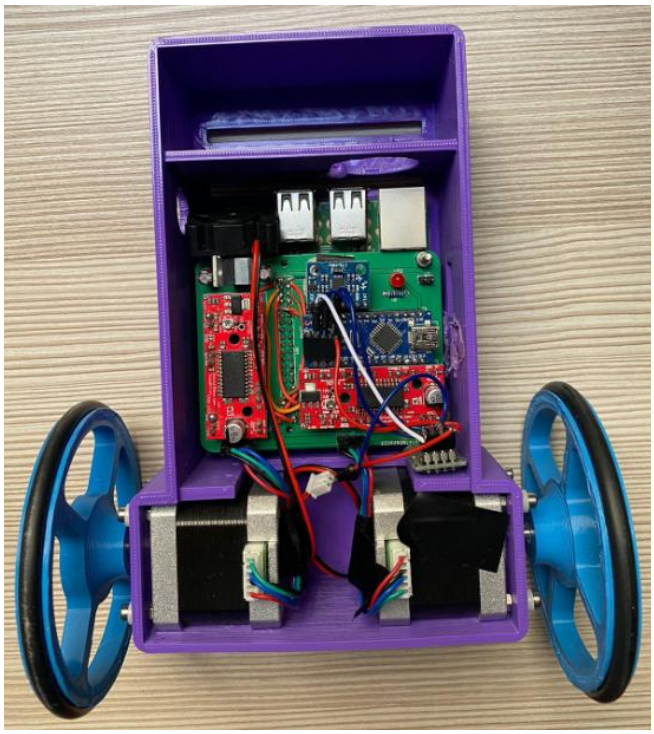


Figura 21. Diseño actual del segway.

3 ALGORITMOS DE ENTRENAMIENTO

En este capítulo se comentarán los diferentes algoritmos y aplicaciones empleadas en la creación y entrenamiento de los detectores que se utilizarán para la identificación de los elementos que se quieren detectar [12].

Primero se definirá el proceso de adquisición de imagen y posteriormente se explicará detalladamente la creación del detector por diversos métodos.

3.1. Adquisición de imágenes

Para la adquisición de imágenes se emplearán dos algoritmos diferentes, desarrollados y ejecutados en la Raspberry, uno basado en la adquisición de imágenes con la PiCamera [13], creando un dataset propio y único y el otro centrado en el almacenamiento de imágenes temáticas ubicadas en internet. Ambos son códigos sencillos, con poca carga de trabajo para el procesador de la placa y, por lo tanto, de rápida ejecución. Cabe destacar que los datasets creados con estos algoritmos tienen un peso muy bajo puesto que la resolución de las imágenes está controlada para facilitar su portabilidad y que además gracias a ellos se puede formar tanto el conjunto positivo de entrenamiento como el negativo, indistintamente. Se muestra un diagrama en la Figura 22.

Se emplearán en situaciones bien diferenciadas:

- El primero se utilizará cuando el elemento a detectar se pueda encontrar con facilidad, como por ejemplo un cubierto, un componente electrónico concreto o en este caso, una pinza de la ropa.
- El segundo será útil cuando el elemento a detectar no pueda ser encontrado o fotografiado

con facilidad o no se tenga acceso a él, como una casa, un avión o en este caso, un Segway.

Ambos algoritmos se muestran completos en el capítulo Anexos al final del presente documento, pero en este epígrafe se detallarán las variables, funciones y librerías utilizadas.

Cabe destacar que a lo largo del presente capítulo se definirán los procesos seguidos para lograr la detección del vehículo no tripulado, robot o Segway, pero el mismo procedimiento ha sido empleado para la detección de **pinzas** y **caras**.

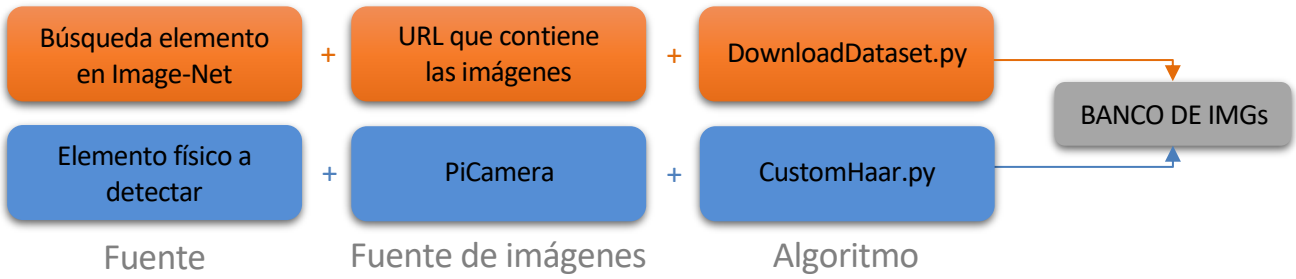


Figura 22. Esquema de creación del banco de imágenes.

3.1.1 Creación del dataset con la PiCamera

Corresponde con el epígrafe a) del capítulo Anexos y se basa en la utilización de la PiCamera para que, de manera automática, una vez se encuentre por debajo de un cierto umbral de ruido en la imagen, realice una captura de lo que actualmente esté viendo, cada diez fotogramas.

A continuación, se muestra el algoritmo en cuestión:

```
# Importación de librerías
import cv2
import os
import time

########################################################
# Inicializaciones

myPath = '/home/pi/Desktop/data/images'
cameraNo = 0       # Selección de la cámara
cameraBrightness = 180 # Selección del brillo
moduleVal = 10     # Guarda un frame daca 'moduleVal' para evitar repeticiones
minBlur = 100      # Valor que evita guardar imágenes borrosas
grayImage = False # Flag para el guardado de imágenes en escalade grises
saveData = True    # Flag para el guardado de imágenes
showImage = True   # Flag para mostrar la preview
imgWidth = 180     # Definición del ancho y el alto de la imagen
imgHeight = 120
count = 0
countSave = 0
```

En primer lugar, se importan las librerías necesarias, que serán las OpenCV (*cv*) para el tratamiento de imágenes, *os* para el tratamiento de las rutas y ubicaciones de archivos en la

Raspberry y *time* para hacer operaciones a tiempo real. También se inicializarán todas las variables que se utilizarán durante la ejecución del algoritmo.

El siguiente paso es la declaración de la variable que irá dando nombre a las carpetas en las que se almacenen las imágenes y la inicialización de la cámara con `cv2.VideoCapture` indicando qué cámara se va a utilizar (`cameraNo`). También se configurará la resolución de la cámara y la orientación de la grabación gracias a `cap.set`, en la cual el primer elemento indica la variable de la cámara a configurar y el segundo el valor que recibe. En este caso se trabajará con una resolución de 640x480 píxeles y con la imagen girada 180˚ para que se vea correctamente.

```
#########################################################

# Inicio de la cámara y setup
global countFolder
cap = cv2.VideoCapture(cameraNo)
cap.set(3, 640)
cap.set(4, 480)
cap.set(35,180) # Giro 180º la imagen de la cámara
```

Posteriormente se definirá la función que va asignando el nombre a cada una de las carpetas de imágenes capturadas, evitando que se sobreescriba la anterior al ejecutar el código en sucesivas ocasiones. Con esto se busca poder crear varios datasets sin tener la necesidad de crear manualmente las carpetas además de aumentar notablemente la comodidad.

No se asigna en este punto el nombre “p” o “n” que se emplearán en el entrenamiento del detector porque las imágenes deben ser revisadas para verificar que todas y cada una de las imágenes contienen información relevante, si son positivas, o no contienen el elemento a detectar, si son negativas, aumentando de este modo la calidad del mismo.

```
# Creación de una función que irá nombrando cada carpeta por orden
def saveDataFunc():
    global countFolder
    countFolder = 0
    while os.path.exists(myPath+ str(countFolder)):
        countFolder += 1
    os.makedirs(myPath + str(countFolder))


if saveData:saveDataFunc()
```

Después se ejecutará el bucle infinito mediante el cual se irán capturando las imágenes. En primer lugar, se almacenará en la variable `img` toda la información capturada por la cámara en tiempo real para poder tratarla con posterioridad. Luego se reescalará en función de las variables previamente definidas y, si se ha elegido la opción de almacenar las capturas en escala de gris, se reescribirá la variable `img` para que guarde todas las imágenes en esta escala de color.

El paso posterior, ya se haya guardado en escala de grises o no, será comparar los niveles de ruido de la imagen mediante el uso del operador laplaciano, que permitirá filtrarla de tal modo que los bordes queden resaltados como se muestra en la Figura 23.

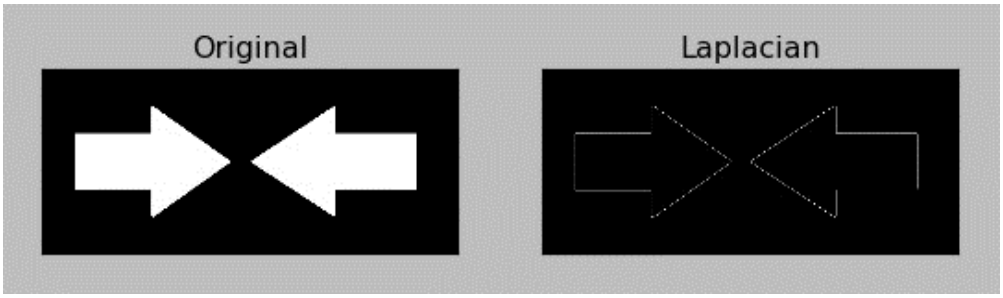


Figura 23. Resultado del operador laplaciano.

Esto facilitará la tarea de buscar ruido en la imagen, pues resalta únicamente los bordes de cualquier elemento que aparezca en ella, de tal modo que solo hay que comparar el nivel de ruido de la imagen capturada con el definido inicialmente mediante un simple operador *if*. El objetivo de emplear este operador es buscar una imagen muy correcta que aporte información valiosa para el entrenamiento del detector y eso se consigue con una imagen bien enfocada y capturada.

Después simplemente se almacena el valor de tiempo actual en una variable llamada `nowTime` y se asigna el nombre a la imagen capturada en función de la cuenta de imágenes capturadas en esa sesión, el nivel de ruido de la imagen y la hora actual, sin olvidar el formato de la imagen, en este caso, .png por tener una mejor escalabilidad al no contar con compresión.

Para finalizar esta sección de código, por comodidad para el usuario, se muestra por pantalla el momento en el que se captura la imagen, así el usuario tendrá información sobre si realmente se están capturando imágenes, o si por el contrario la calidad que está midiendo la cámara no es la esperada. Como último paso, se actualizan los valores de los contadores.

```
while True:

    success, img = cap.read()   # Se lee la imagen de la cámara
    img = cv2.resize(img, (imgWidth, imgHeight)) # Se reescala
    if grayImage:img = cv2.cvtColor(img,cv2.COLOR_BGR2GRAY) # Guarda escala gris
    # Guarda la imagen si no tiene excesivo ruido, indicando la hora y el ruido
    if saveData:
        blur = cv2.Laplacian(img, cv2.CV_64F).var()
        if count % moduleVal ==0 and blur > minBlur:
            nowTime = time.time()
            cv2.imwrite(myPath + str(countFolder) +
                    '/' + str(countSave)+"_"+ str(int(blur))+"_"+str(nowTime)+".
png", img)
            print("Photo") # Indica cuando se ha capturado una imagen
            countSave+=1
        count += 1
```

Esta última sección del algoritmo está implementada para facilitar el trabajo al usuario, mostrando por pantalla la captura de la cámara a tiempo real en una ventana nueva, escalada a la resolución a la que se capturarán las imágenes, gracias al comando `cv2.imshow`.

Como paso final, si se pulsa en cualquier momento de la ejecución la tecla Q del teclado, se detendrá la ejecución del algoritmo y se cerrará la ventana creada.

```
    # Muestra un vídeo para guiar al usuario mientras captura las imágenes
    if showImage:
        cv2.imshow("Image", img)

    # Al pulsar la tecla Q cierra la ventana y detiene la ejecución
    if cv2.waitKey(1) & 0xFF == ord('q'):
        break

cap.release()
cv2.destroyAllWindows()
```

Esto resultará en una carpeta de imágenes con una calidad razonable para en entrenamiento del detector

3.1.2 Creación del dataset desde una url

En este epígrafe se detallarán las librerías, variables y comandos empleados para la captura de imágenes a partir de sitios web que posean enlaces con imágenes.

La creación del dataset mediante este método es más sencilla, pues simplemente hay que entregarle al algoritmo la url que contiene el conjunto de enlaces de las imágenes que se desean descargar. Para ello se hará uso de la web http://www.image-net.org/index [14], que es una base de datos de imágenes previamente organizadas en función del contenido de la misma. En la página se encontrará un índice en el cual aparecen todas las imágenes que se pueden encontrar. En este caso, dado que se probarán varios clasificadores, se hará uso de imágenes de casas para formar el conjunto de imágenes negativas.

Este proceso, de no llevarse a cabo mediante el método explicado podría llevar demasiado tiempo, pues la forma más directa, que no la más rápida, sería ir a Google Imágenes y descargar una por una todas las imágenes que fuesen necesarias.

Ahora se pasará a explicar el contenido del algoritmo:

En primer lugar, como en el caso anterior, se importarán las librerías utilizadas, que en este caso son *urllib.request* para el tratamiento de los enlaces, *cv2* para el tratamiento de las imágenes, *numpy* para el trabajo con variables numéricas y *os* para la gestión de archivos y su ubicación.

```
# Importación de librerías
import urllib.request
import cv2
import numpy as np
import os
```

El siguiente paso será la inicialización de las variables a utilizar. La primera (*neg_images_link*) se utilizará para guardar el enlace que contiene el resto de links de las que se extraerán las imágenes. La siguiente variable a utilizar será *neg_images_urls*, que abrirá el enlace almacenado en la variable anterior y leerá todas las urls como paso previo a la descarga de las imágenes. Por último, se utilizará una variable numérica que irá nombrando las imágenes de forma ascendente según se vayan descargando. Esto acelera aún más el proceso, pues de no hacerlo habría que ir nombrando las imágenes una a una.

```
# Lectura del link que contiene los urls de las imágenes
neg_images_link = 'http://www.image-
net.org/api/text/imagenet.synset.geturls?wnid=n09618957'
# Lectura de las urls de las imágenes
neg_image_urls = urllib.request.urlopen(neg_images_link).read().decode()
pic_num = 1
```

A continuación, se comprobará si existe una carpeta para almacenar las imágenes negativas y en caso negativo, se creará para comenzar con la descarga.

```
# Si no existe una carpeta llamada ‘neg’ la crea para las imágenes descargadas
if not os.path.exists('neg'):
    os.makedirs('neg')
```

El siguiente paso será la ejecución del bucle en el que se descargarán las imágenes. El proceso se repite por cada línea de texto en la web almacenada al inicio, debido al uso del método `.split`, que divide el texto por cada salto de línea (\n) encontrado.

Posteriormente se mostrará por pantalla el link de la imagen, se almacenará en la carpeta creada mediante el método *.request.urlretrieve* con el nombre correspondiente.

Finalmente se leerá imagen guardada y se pasará a escala de grises para facilitar el entrenamiento posterior. Esto se conseguirá mediante el uso de la librería *cv2* y los métodos `.imread` para leer la imagen descargada y `.imwrite` para sobreescribir esa misma imagen en escala de grises.

```
# Bucle para descargar las imágenes, numerarlas y almacenarlas en la carpeta
for i in neg_image_urls.split('\n'):
    try:
        print(i)
        urllib.request.urlretrieve(i, "neg/"+str(pic_num)+".jpg")
        img = cv2.imread("neg/"+str(pic_num)+".jpg", cv2.IMREAD_GRAYSCALE)
        cv2.imwrite("neg/"+str(pic_num)+".jpg", img)
        pic_num += 1
```

Como paso final del algoritmo, se creará una excepción en el caso de que la url no contenga una imagen y se mostrará por pantalla un aviso.

```
# Si la url no contiene una imagen, lo indica y pasa a la siguiente
    except Exception as e:
```

```
        print(str(e))
```

Al finalizar la ejecución del código completo se obtendrá una carpeta con el número de imágenes que se consideren suficientes, pues el bucle de descarga finalizará en el momento en el que el usuario decida detener la ejecución.

3.2. Proceso de creación del detector

Una vez se cuenta ya con los bancos de imágenes preparados, se pasará a la creación del detector. Este es un proceso complejo, abordado en este proyecto de varios modos para encontrar el mejor posible, buscando la obtención de detecciones correctas en todo momento y situación.

Esta tarea está compuesta por varias fases o acciones sobre dos conjuntos de datos, en este caso imágenes, con los que se va a trabajar: **conjunto de entrenamiento** y **conjunto de validación**.

- El conjunto de **entrenamiento (CE)** es aquel grupo de imágenes, tanto positivas como negativas, empleado para la definición del detector. Se consideran imágenes positivas aquellas que contienen el elemento a detectar y las negativas el caso contrario. Estos conjuntos se emplean para que durante las fases del entrenamiento se busquen elementos comunes en las imágenes positivas, caracterizando el objeto que se desea identificar de manera inequívoca. Las imágenes negativas aportan información sobre cuáles son los elementos que no se desean detectar, pero que la cámara podría encontrar en la posición en la que se va a plantear que trabaje.

  Puede parecer una selección banal, pero es la base sobre la que se sustenta el clasificador: una mala selección de imágenes para los bancos positivos y negativos puede provocar que el clasificador funcione de una forma inesperada no detectando el objeto.

- El conjunto de **validación (CV)** es un conjunto de imágenes diferentes que se emplea para probar el clasificador una vez creado, este paso también influye en el clasificador dado que se puede comprobar a simple vista si las detecciones realizadas son correctas. En caso de no ser así, se puede añadir imágenes concretas al conjunto de entrenamiento para corregir el funcionamiento del detector.

Durante el desarrollo del trabajo se mostrarán tablas con los resultados obtenidos al emplear los diferentes clasificadores sobre los conjuntos de validación seleccionados.

Para la creación de los detectores que se emplearán y explicarán a lo largo del presente documento se han utilizado dos aplicaciones bien diferenciadas.

3.2.1 Cascade Trainer GUI

La primera y la que mejor resultados ha aportado es "***Cascade Trainer GUI***" [15], un software creado específicamente para entrenar, probar y mejorar un modelo de clasificador en cascada. Está basado en OpenCV y emplea la técnica de entrenamiento en cascada descrita en el epígrafe anterior.

Esta aplicación ha sido instalada en Windows 10 para evitar cualquier posible problema de incompatibilidad.

Para utilizarla simplemente hace falta crear, con el dataset de imágenes, dos carpetas, una de imágenes positivas (p) y otra de imágenes negativas (n) del modo en que se muestra en la Figura

24.

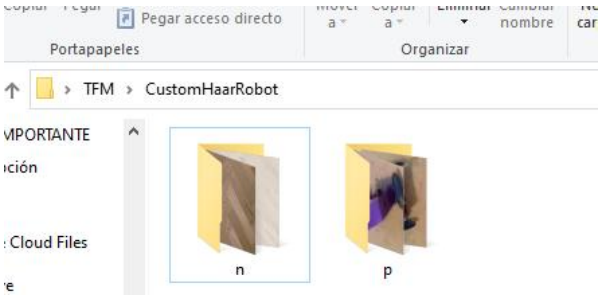


Figura 24. Organización del dataset.

Una vez hecho esto, se ejecuta la aplicación, cuya interfaz es muy simple e intuitiva. A continuación, se detallarán los parámetros a modificar para conseguir el mejor resultado posible, el resto de los parámetros se pueden dejar en su configuración por defecto, puesto que es correcta.

Nada más abrir la aplicación, se encuentra la pestaña de configuración del entrenamiento (Figura 25) en la cual se debe seleccionar la ruta en la que se encuentran las carpetas **p** y **n** creadas anteriormente y se indica el número exacto de imágenes negativas localizadas en la carpeta n. Este último paso es vital, pues condiciona todo el entrenamiento.

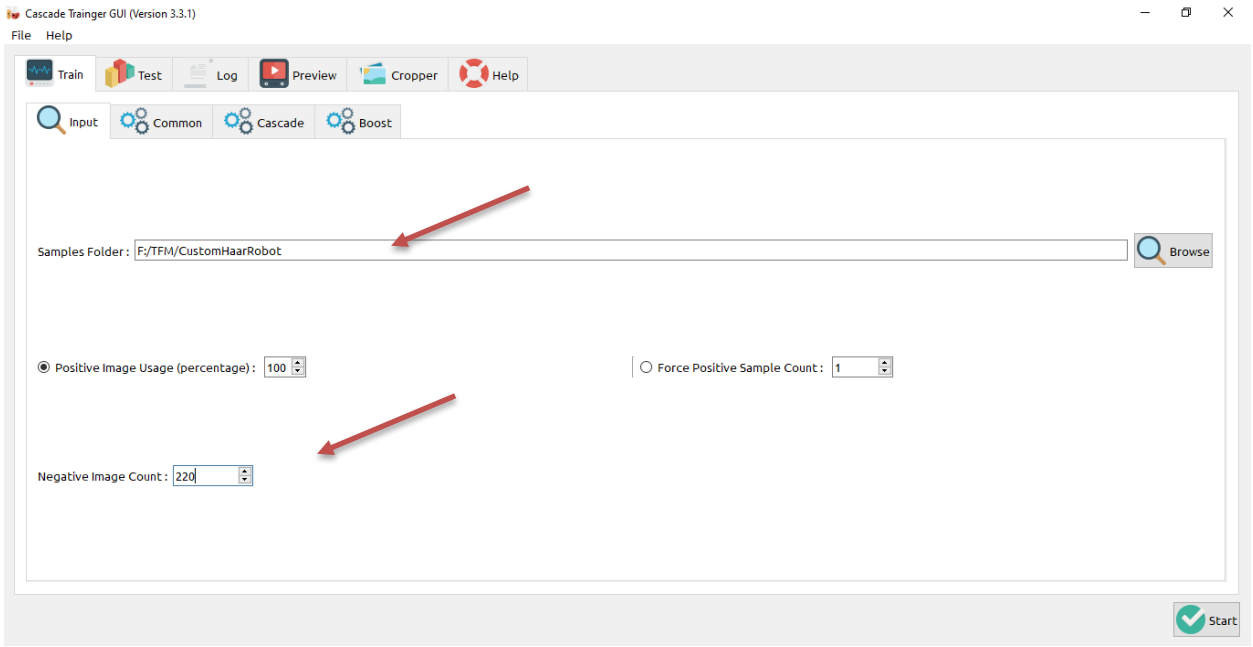


Figura 25. Ventana inicial de *Cascade Trainer Gui*.

En la siguiente pestaña (Figura 26) se seleccionará el número de fases de entrenamiento deseadas. Se ha detectado que el mejor resultado se consigue entre 15 y 20 fases.

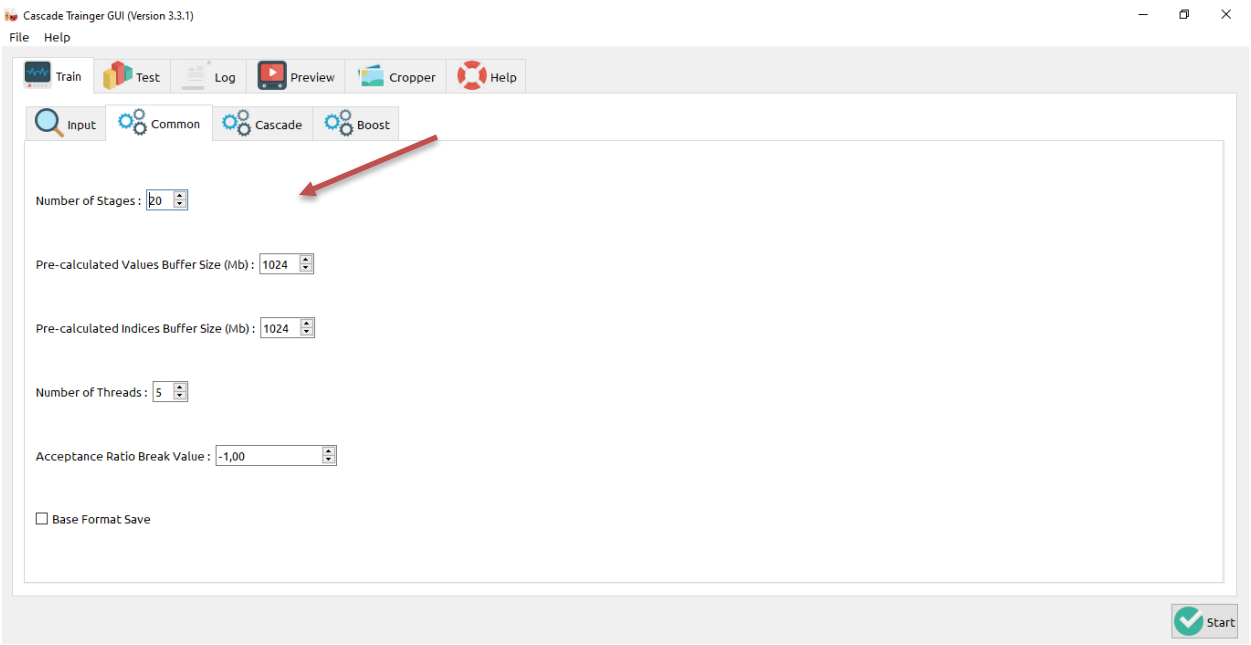


Figura 26. Parámetros dentro de la pestaña "common".

Finalmente, en la pestaña "Cascade" se selecciona la anchura y altura de las imágenes del dataset y se selecciona el tipo de clasificador. En este caso se trabajará con el detector tipo Haar, pues ha sido el que mejores resultados ha aportado como se verá más adelante.

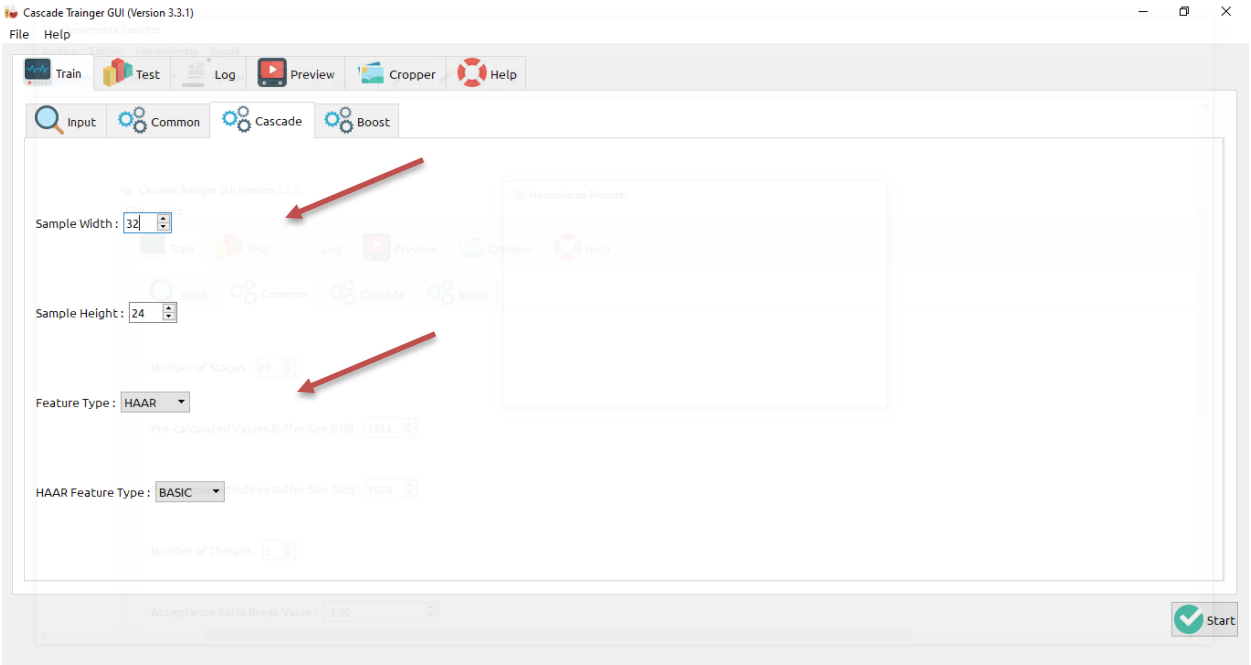


Figura 27. Parámetros dentro de la pestaña "cascade".

En este caso se seleccionarán imágenes con una resolución variable, pero proporcional a la indicada en la Figura 27.

Al ejecutar el entrenador se obtiene, en la misma carpeta en la que se alojan tanto la carpeta positiva como la negativa, una nueva carpeta llamada "classifier" que en su interior contiene los resultados del entrenamiento. Estos son un archivo llamado "cascade.xml" que es el clasificador final resultado del entrenamiento completo, un archivo "stage.xml" extra por cada fase del

entrenamiento (si se deciden emplear 20 fases se generarían ese mismo número de archivos "stage.xml") y un archivo de texto que contiene el registro del entrenamiento, por si fuese necesario consultarlo.

3.2.2 Matlab

Como segunda opción para la creación del clasificador se ha empleado **Matlab**. El empleo de esta aplicación viene dado porque quiere tratarse de implementar algoritmos en diferentes plataformas con el fin de comprobar los resultados y elegir en función de la eficacia de cada una de ellas.

De manera independiente a la obtención de las imágenes se puede trabajar en otra etapa vital para la obtención del detector, que es la selección de las regiones de interés (ROI) de las imágenes de entrada.

Para la creación del clasificador en Matlab se hará uso de la app llamada "Image Labeler", que será empleada para la selección manual de las regiones de interés de cada una de las imágenes positivas.

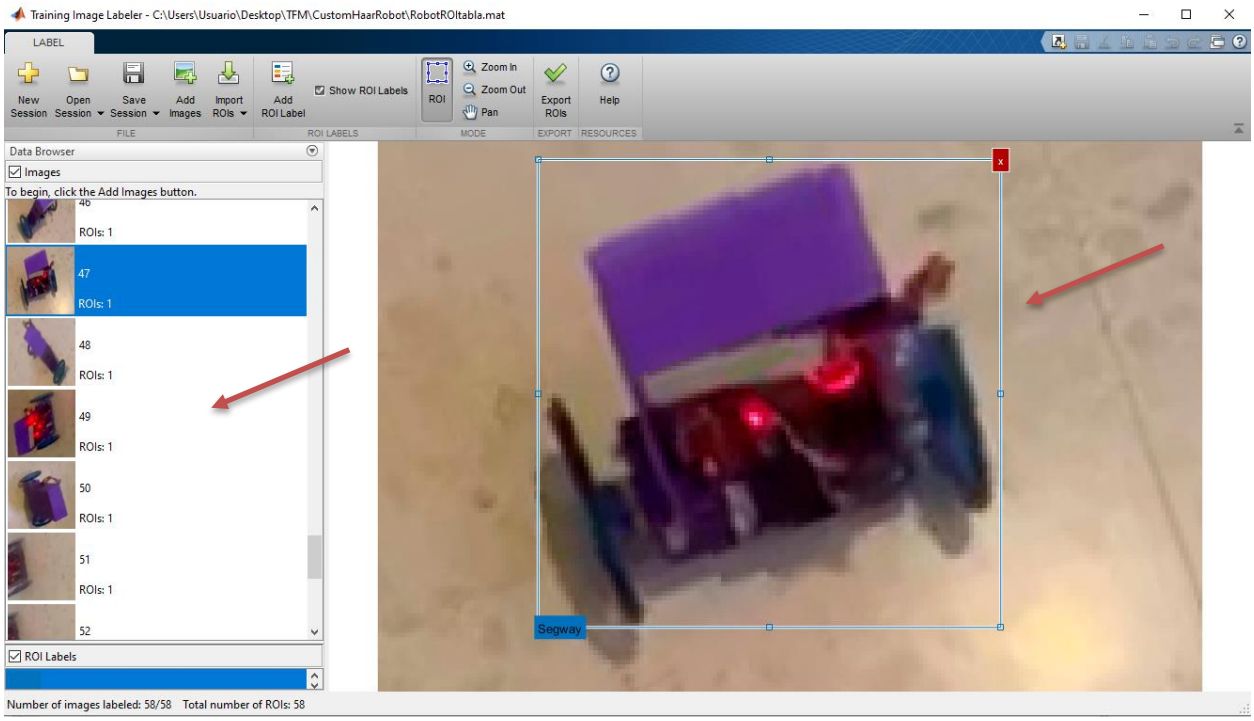


Figura 28. Visión general de la app "Image Labeler".

La vista general, una vez se ejecuta la app, se muestra en la Figura 28. En la banda superior se encuentran herramientas como el inicio de una nueva sesión de etiquetado, el guardado o la carga de una sesión o la exportación de las ROIs ya seleccionadas.

En la región izquierda se muestran las imágenes con las que se está trabajando, así como el número de ROIs seleccionadas en cada una de ellas. Debajo de esto se encuentran las etiquetas. En este caso solo se empleará una etiqueta por sesión de entrenamiento en función del objeto a detectar.

En la zona central aparece la imagen seleccionada de la lista izquierda, permitiendo la selección de las ROIs y las etiquetas correspondientes a ellas. Es importante destacar que las ROIs deben ser seleccionadas con cuidado y atención, pues una selección incorrecta puede provocar que las detecciones sean incorrectas o simplemente, que no existan.

Una vez la selección de las ROIs sea la deseada, se exportarán mediante el botón de la banda de herramientas superior y se creará una variable tipo tabla en el espacio de trabajo, que en este caso se llamará "RobotROItabla.mat" que será guardada en la carpeta indicada, en función del elemento a detectar, para poder ser cargada posteriormente cuando se vaya a llevar a cabo el entrenamiento del clasificador. Estas ROIs serán empleadas en la fase de entrenamiento [16] del clasificador creado y entrenado mediante el algoritmo que se muestra en el Anexos c), no obstante, se procederá a comentar a continuación de manera exhaustiva para aclarar su funcionamiento y justificarlo.

```
clc
clear all

% Inicializaciones
tabla='RobotROItabla.mat';
haar='CustomHaarRobot';
detectorname='RobotDetector_';
```

El primer paso que se realiza es la limpieza tanto del espacio de trabajo como de las variables anteriormente utilizadas para evitar posibles errores generados de la sobreescritura de variables. Posteriormente se pasa a la inicialización de unas variables tipo carácter que se modificarán durante la ejecución del código:

- `tabla` hace referencia al archivo en el que se almacenan las regiones de interés seleccionadas a través de la app Image Labeler. La variable es una tabla de Nx2 siendo N el número de imágenes utilizadas para las selecciones de las ROIs. La primera columna, en todos los casos, almacenará la ruta de la ubicación de la imagen en el sistema mientras que la segunda estará compuesta por cuatro elementos que corresponderán a los vértices de las ROIs seleccionadas.

  Este archivo será empleado en el entrenamiento para localizar la región de la imagen en la que se encuentra el elemento positivo que se desea localizar.

- `haar` almacena el nombre de la carpeta en la que se trabajará. Dentro de ella se encuentran tanto la carpeta de imágenes positivas como de las negativas, además de los archivos .xml que contienen los clasificadores creados con matlab o con la app Cascade Trainer GUI y el archivo .mat descrito en el punto anterior.

- `detectorname` es una cadena de caracteres empleada para la creación automática de los clasificadores entrenados en Matlab. Esta variable será modificada más adelante en función del elemento que se quiera clasificar como se describirá más adelante.

En este siguiente paso, lo que se hará simplemente será adaptar las variables inicializadas anteriormente al elemento a detectar, haciendo que coincida el nombre del elemento con las variables, de tal forma que todo el trabajo realizado con las variables sea correcto, empleando la carpeta, el clasificador y las regiones de interés indicadas.

Este trabajo se realizará mediante la solicitud al usuario de introducción del elemento que se desea clasificar. Es decir, si se desea clasificar el Segway o robot se entregará al algoritmo desde el teclado una "R", si se desea trabajar con pinzas se empleará la "P", y del mismo modo con las caras, se entregará la "C". Por defecto, el algoritmo está preparado para trabajar con el robot.

En caso de que no se introduzca la inicial del objeto de forma correcta, se entregará un mensaje de error.

```
% Creación de las variables

% Se solicita la introducción del objeto que se desea clasificar y renombra
las variables inicializadas anteriormente
prompt='Selecciona un objeto ([R]obot, [P]inza, [C]aras) [R]:\n';
name=input(prompt,'s');
if (~isempty(name) && (name=='P'))==1
    name='Pinza';
elseif (~isempty(name) && (name=='R'))==1
    name='Robot';
elseif (~isempty(name) && (name=='C'))==1
    name='Caras';
else
    disp('ERROR: Nombre no encontrado.');
end
tabla(1:5)  = name;
haar(11:15) = name;
detectorname(1:5) = name;
```

A continuación, se muestra la Figura 29, en la que se aprecia la solicitud recientemente explicada:

```
%==========================================
%==========CREACIÓN DEL DETECTOR===========
%==========================================

% Inicializaciones
tabla='RobotROItabla.mat';
haar='CustomHaarRobot';
detectorname='RobotDetector_';


% Creación de las variables
prompt='Selecciona un objeto ([R]obot, [P]inza, [C]aras) [R]:\n';
name=input(prompt,'s');
if (~isempty(name) && (name=='P'))==1
    name='Pinza';
elseif (~isempty(name) && (name=='R'))==1
    name='Robot';
elseif (~isempty(name) && (name=='C'))==1
    name='Caras';
else
    disp('ERROR: Nombre no encontrado.');
end
tabla(1:5)  = name;
haar(11:15) = name;
detectorname(1:5) = name;


% Tabla de detecciones
load(tabla);
```

```
Command Window
  Selecciona un objeto ([R]obot, [P]inza, [C]aras) [R]:
fx
```

Figura 29. Solicitud al usuario de la selección del elemento a clasificar.

El siguiente paso en la creación del clasificador es la carga de las ROIs seleccionadas, almacenadas en la variable `tabla` como se ha explicado antes. Esta operación se realiza para que en espacio de trabajo se genere la variable que contiene la matriz en la que se encuentran las direcciones de las imágenes, así como la información de ubicación de las regiones de interés de cada una de ellas.

Esta tabla se almacenará en la variable `deteccionesPositivas` para homogeneizar el proceso de entrenamiento del detector posterior.

```
% Tabla de detecciones
% Se carga la tabla que contiene las ROIs del elemento seleccionado
load(tabla);
if name=='Robot'
    deteccionesPositivas = RobotROItabla(:,1:2);
elseif name=='Pinza'
    deteccionesPositivas = PinzaROItabla(:,1:2);
```

```
elseif name=='Caras'
    deteccionesPositivas = CarasROItabla(:,1:2);
end
```

A continuación, se añadirán a las direcciones "activas" en Matlab aquellas que contengan las imágenes negativas y positivas del elemento seleccionado, la carpeta con imágenes en las que aparece el elemento a detectar ("p") se almacenará en la variable `carpetaPositivas` y la carpeta en la que no aparezca ese elemento ("n") se guardará en la variable `carpetaNegativas`.

Este trabajo se llevará a cabo gracias al comando `fullfile` con el que se creará la dirección completa a la que apuntar y el comando `addpath` que incluirá esa dirección en la lista de direcciones "activas" de la sesión actual de Matlab.

```
% Paths

% Ubicación de los CE, positivos y negativos
carpetaPositivas =
fullfile('C:\','Users','Usuario','Desktop','TFM',haar,'p');
addpath(carpetaPositivas);
carpetaNegativas =
fullfile('C:\','Users','Usuario','Desktop','TFM',haar,'n');
deteccionesNegativas = imageDatastore(carpetaNegativas);
```

Finalmente se entrenará el detector con toda la información entregada previamente a Matlab. El primer paso en la detección será la selección por parte del usuario, mediante el comando `input`, de los valores de dos parámetros claves para el entrenamiento:

- **`FalseAlarmRate`***:* que indica la tasa de falsos positivos que se desea.
- **`NumCascadeStages`***:* que define el número de fases del entrenamiento.

Para garantizar el correcto funcionamiento de las etapas, debe existir un compromiso entre el valor de estos parámetros. Esto se consigue seleccionando un valor bajo de **`FalseAlarmRate`**, del orden de décimas o centésimas, haciendo que en cada etapa aparezcan muchas detecciones descartadas y muy poco error en las detecciones positivas obtenidas. Las detecciones negativas no pueden corregirse en las etapas posteriores, pero sí las falsas positivas, ya que el resultado de etapas anteriores se vuelve a evaluar en las etapas sucesivas, de tal modo que se mejora el rendimiento del futuro detector a un valor muy elevado.

Se debe prestar atención también al valor de la variable **`NumCascadeStages`**, que indica el número de fases del entrenamiento, dato del que depende tanto el tiempo de ejecución del algoritmo, como la calidad del detector. Teniendo en cuenta que cada fase del entrenamiento elimina y aprueba varias detecciones de forma errónea, es posible afirmar que un mayor número de etapas reducirá el error del futuro detector a un valor muy bajo

El número de muestras positivas que se recibirán será el resultado de la siguiente formulación matemática:

$$Nº\ muestras\ positivas = \frac{Total\ de\ muestras\ positivas}{1 + (Nº\ fases\ entrenamiento - 1) * (1 - Ratio\ reales\ positivos)}$$

Estos parámetros serán almacenados en las variables `far` y `ncs` respectivamente. Si alguno de los dos está vacío, o ambos, recibirán los valores 0.15 y 15 como opción por defecto. El valor de estos parámetros ha sido seleccionado después de la repetida ejecución del algoritmo completo con diferentes valores de estos, obteniendo para un valor muy elevado de `NumCascadeStages` un tiempo de ejecución extremadamente alto para la mejoría en los resultados y una variabilidad muy elevada en los valores de `FalseAlarmRate` que será explicada detenidamente más adelante.

```
%======Entrenamiento del detector======

% Elección del valor de los parámetros de entrenamiento
% FalseAlarmRate:   Tasa de falsos positivos deseada.
% NumCascadeStages: Número de fases del entrenamiento.
prompt='Selecciona un valor para FalseAlarmRate (Tasa de falsos positivos
deseada) (0-1) [0.5]:\n';
far=input(prompt);
prompt='Selecciona un valor para NumCascadeStages (Número de fases del
entrenamiento) (1-20) [15]:\n';
ncs=input(prompt);
if isempty(far)
    far=0.5;
end
if isempty(ncs)
    ncs=15;
end
```

Como paso final, se adaptará el nombre del clasificador creado al elemento a detectar, al valor de `FalseAlarmRate` y al valor de `NumCascadeStages`, componiendo el nombre del clasificado del siguiente modo:

{Elemento a detectar}Detector_{far}_{ncs}.xml

Y finalmente se ejecutará el entrenamiento del clasificador mediante el comando `trainCascadeObjectDetector`, al que se le entregarán varias variables:

- `DetectorName`: Nombre del detector.
- `deteccionesPositivas`: Ruta a la carpeta con imágenes positivas.
- `carpetaNegativas`: Ruta a la carpeta con imágenes negativas.
- Valores de `FalseAlarmRate` y de `NumCascadeStages`.

```
% Nombre del detector en función del valor de las variables
if (length(num2str(far,3))<=3)
    aux=strcat(num2str(far,3),'0_',num2str(ncs));
else
aux=strcat(num2str(far,3),'_',num2str(ncs));
end
DetectorName=strcat(detectorname,num2str(aux),'.xml');
```

```
% Inicio del entrenamiento en cascada
trainCascadeObjectDetector(DetectorName,deteccionesPositivas,carpetaNegativas
,'FalseAlarmRate',far,'NumCascadeStages',ncs);
detector=vision.CascadeObjectDetector(DetectorName);
```

Como paso final, se almacenará el detector en la variable `detector` para trabar con ella más adelante en una sección diferente de este mismo algoritmo.

En la Figura 30 se aprecia el resultado del proceso de entrenamiento a través del espacio de trabajo:

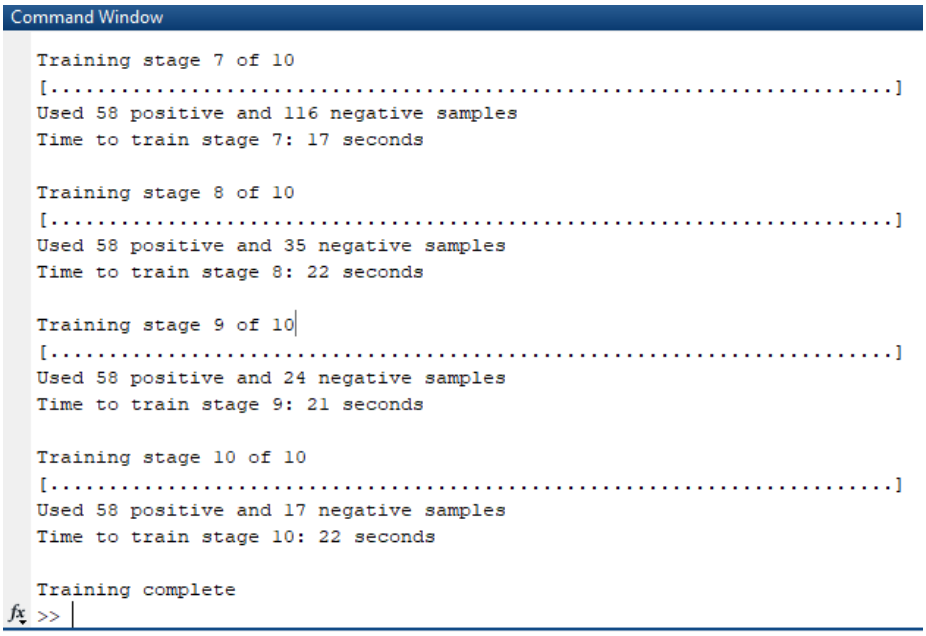


Figura 30. Resultado del entrenamiento empleado el algoritmo detector.m en Matlab.

En la imagen anterior puede apreciarse cómo a medida que avanzan las fases del entrenamiento el tiempo de cada fase va aumentando y cómo, a su vez, el número de imágenes negativas se ve reducido cada vez más. Este ejemplo ha sido realizado con una tasa de falsos positivos de 0.15 y 10 fases de entrenamiento.

A continuación, se describirán detalladamente los algoritmos encargados de la identificación de los diferentes objetos a partir de los clasificadores creados en este capítulo.

4 ALGORITMOS DE DETECCIÓN

En este capítulo se procederá a definir el proceso que se ha seguido hasta la consecución del objetivo del proyecto: la correcta detección del Segway y la obtención precisa de la localización instantánea del mismo.

Debido a que se han empleado algoritmos anidados o con modificaciones leves respecto a alguno que actúa como base, algunas líneas de código no se explicarán con tanta profundidad como otras, buscando evitar la repetición en las explicaciones y hacer la lectura más cómoda.

4.1. Algoritmo de detección mediante un detector HOG (detecciónObj.m)

Para comenzar con las detecciones y aprovechando que el último algoritmo descrito es el que se encarga de realizar el entrenamiento del clasificador en Matlab, se explicará el procedimiento de detección aplicando este mismo software, desarrollado como un código independiente pero que debe ser ejecutado con posterioridad al encontrado en el epígrafe d) del capítulo Anexos dado que se emplea una variable clave que en él se genera. Esta variable es `detector` y contiene al clasificador.

Cabe destacar que en este caso se ha trabajado con un vídeo pregrabado pues después de varios intentos de trabajar con vídeos en streaming con Matlab, ya sea mediante conexión con la Raspberry o con otro dispositivo o cámara se ha descartado pues generaba diversos problemas de software y conectividad.

En primer lugar, en esta sección de código se carga el archivo de vídeo que contiene el elemento a detectar. Este proceso se realiza gracias al comando `VideoReader` que permite leer un archivo de vídeo y almacenarlo en una variable con la que posteriormente se operará.

Un punto importante, en caso de que se quiera trabajar con el detector creado mediante la app Cascade Trainer GUI, es que se debe descomentar la línea de código a la que se hace referencia en el propio algoritmo para sobreescribir la variable que contenía al detector creado con Matlab con la desarrollada en la nueva app.

```
% Vídeo en el que aparece el objeto a detectar
vid = VideoReader('videoSegway.mp4');

% En caso de querer comprobar el detector creado en la otra aplicación,
descomentar la siguiente línea
%detector = vision.CascadeObjectDetector('cascadeR.xml');
```

A continuación, se pasa a trabajar con el vídeo mediante un bucle que continuará siempre y cuando la variable en la que se guardó el vídeo siga contando con información, en este caso frames o imágenes dentro del propio vídeo. Este proceso se realiza gracias a `hasFrame`, que comprueba si la variable contiene información legible con formato de vídeo, en caso afirmativo entrega un `1` manteniendo el bucle y en caso contrario un `0` provocando el abandono del bucle.

```
% Bucle infinito para la detección
while hasFrame(vid)
    vf = readFrame(vid);
    bbox = step(detector,vf);
    detectedImg =
insertObjectAnnotation(vf,'rectangle',bbox,'Segway','LineWidth',2);
    imshow(detectedImg);
    pause(1/vid.FrameRate);
end
```

Dentro del bucle el trabajo que se realiza es simple. El primer paso es almacenar cada frame del vídeo en la variable `vf` mediante la función `readFrame` que lee cada una de las imágenes del vídeo que se le entregue.

Después se ejecuta un `step` sobre cada uno de los frames empleando el detector almacenado anteriormente. Este paso es clave, pues mediante él, se buscan detecciones correctas en el vídeo, así se analizará el vídeo completo, evitando cualquier posible error en el tratamiento del mismo, por ejemplo, debido al ruido. En caso de que existan detecciones positivas, estas se almacenarán en la variable `bbox` que tendrá tantas filas como detecciones correctas localice y 4 columnas que indicarán la posición de la bounding box.

El siguiente paso es mostrar sobre cada frame las detecciones localizadas. Esto se consigue mediante el comando `insertObjectAnnotation` al que se le debe entregar cada frame (`vf`) así como el tipo de bounding box a utilizar, en este caso un rectángulo, y su posición, almacenada en la variable `bbox`. También se puede etiquetar la detección, en este caso se opta por etiquetar la detección como `'Segway'` ya que es el objeto que se está buscando, y finalmente se puede seleccionar una anchura para la bounding box.

```
>> vid = VideoReader('videosegway.mp4');
>> fps=vid.FrameRate

fps =

   29.9939

>> dur=vid.Duration

dur =

  114.4233

>> frames=vid.NumberOfFrames

frames =

        3432

>> frames/dur

ans =

   29.9939
```

Figura 31. Frames por segundo, duración completa y frames del vídeo.

Una vez están las detecciones mostradas sobre cada frame, se mostrará el vídeo completo, imagen a imagen mediante `imshow` y el comando `pause`, que se encargará de que el vídeo se muestre a una velocidad correcta. Esto se conseguirá deteniendo cada una de las imágenes una fracción de tiempo correspondiente al número de imágenes por segundo del vídeo original, es decir, 1/FramesPorSegundo del vídeo.

En la Figura 31 se pueden ver las características del vídeo con el que se va a trabajar y cómo el número de imágenes por segundo coincide a la perfección con la operación realizada, en la que se divide el número total de frames del vídeo entre la duración total del mismo. De este modo se puede apreciar cómo el vídeo posee 29.9939 ~ 30 imágenes por segundo, por tanto, mediante el comando `pause` explicado en el párrafo anterior, se puede conseguir esa frecuencia de imágenes en el vídeo con las bounding boxes. Como ejemplo en la Figura 32 se puede ver un frame del archivo ya tratado, en el que se muestra la identificación del Segway llevada a cabo por el detector.

Figura 32. Frame del vídeo trabajado con el algoritmo *detector.m* en Matlab.

En el capítulo posterior se estudiarán los resultados obtenidos en mayor profundidad.

4.2. Algoritmo de detección en una imagen (PhotoDetector.py):

En este epígrafe se comentará el primero de los algoritmos de detección creado enteramente en Python. El objetivo de este algoritmo es la detección del vehículo no tripulado en una única imagen, para comprobar el correcto funcionamiento del código. Éste actuará como base sobre la que se desarrollarán el resto de los algoritmos de detección que se explicarán en los próximos epígrafes.

El primer paso es la importación de librerías y la declaración de las variables que se utilizarán.

```
# Importación de librerías
import cv2
import numpy as np

# Carga del clasificador a utilizar
faceCascade = cv2.CascadeClassifier("/home/pi/Desktop/TFM/CustomHaarRobot/classi
fier/cascadeR.xml")
# Carga de la imagen a detectar
img = cv2.imread("/home/pi/Desktop/TFM/scene3Segways.png")
imgGray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY) # Paso a escala de grises
```

En este caso solo se trabajarán con las librerías OpenCV (*cv2*) y Numpy (*np*). La primera para el tratamiento de imágenes y la segunda para operaciones numéricas.

Respecto a las inicializaciones, se contará con `faceCascade` que es la variable en la que se almacenará el clasificador creado con la app Cascade Trainer GUI, `img` que hace referencia a la imagen a analizar y `imgGray` que es la imagen anterior, pero en escala de grises.

Cabe destacar en este punto que la imagen a analizar puede contener uno o varios elementos a identificar, puesto que el detector no está limitado a la detección de un solo elemento. No obstante, se ha realizado una prueba con una imagen con tres Segways, que será la que se muestre al final de este epígrafe.

El siguiente paso es la realización de la detección, que se ejecuta gracias al método `.detectMultiScale` aplicado junto con la variable que almacenaba el clasificador sobre la imagen almacenada, de modo que se almacena en la variable `Segway` la información de la posición de los vehículos detectados. Los parámetros que se aplican junto con la imagen corresponden con el factor de escala (1.2) y el mínimo de vecindades de la detección (4), esto es, la relación entre las detecciones que se encuentren cercanas entre sí. Es una idea similar si se habla en términos de píxeles, donde el concepto de la vecindad hace referencia a aquellos píxeles que se encuentran adyacentes al estudiado.

```
# Se realiza la detección con el clasificador
Segway = faceCascade.detectMultiScale(imgGray, 1.2, 4)
```

El formato en el que se almacenan los datos en la variable `Segway` es similar a como lo hacen en la variable `bbox` en el capítulo anterior: una matriz de Nx4, en la que el número de filas depende de las detecciones instantáneas y las columnas hacen referencia a su posición, en las dos primeras columnas, y a su anchura y altura en las dos segundas.

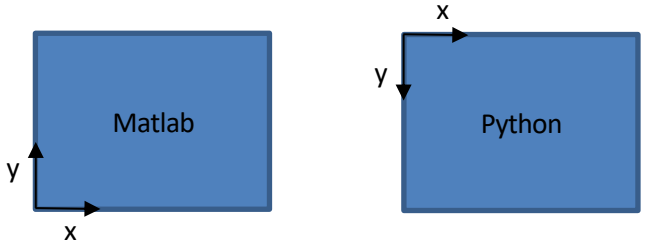


Figura 33. Origen de una imagen en Matlab y Python.

Como se puede ver en la Figura 33, en el caso de Python es muy importante recalcar que el origen de la imagen no es el punto localizado en la parte inferior izquierda, sino que está ubicado en la parte superior izquierda, teniendo su sentido positivo hacia abajo. Esto es vital, pues todas las operaciones relacionadas con posiciones dentro de una imagen en Python se ven modificadas debido a la diferente referencia.

A continuación, se creará un bucle *for* de tal modo que por cada fila que se encuentre en la matriz `Segway` se buscará la posición correspondiente y se colocará un rectángulo gracias a `cv2.rectangle` con las dimensiones de la detección en cuestión, así como una línea de texto mediante `cv2.putText` que indique el elemento que se ha detectado. Para cerrar el bucle se colocará un punto en el centro del rectángulo con el método `cv2.circle` para indicar la posición real del Segway.

La configuración de los elementos explicados en el párrafo anterior se realiza en la misma línea de código, definiendo en todos los casos el código de color a emplear, que consta de tres dígitos de 0 a 255, pero que, al contrario que como se pudiese pensar, no mantienen el formato RGB. En este caso se trabaja con el formato BGR, por lo tanto, la primera cifra hará referencia al color azul, la segunda al verde, y la tercera al rojo. Finalmente se define el grosor de la línea de cada uno de los elementos.

```
# Bucle para crear todas las bounding boxes y localizar su centro
for (x, y, w, h) in Segway:
    a = x+w//2
    b = y+h//2
    cv2.rectangle(img, (x, y), (x+w, y+h), (0,255,255), 2)
    cv2.putText(img, 'Segway', (x, y-5), cv2.FONT_HERSHEY_COMPLEX_SMALL, 1,
(0,255,255), 2)
    roi_color = img[y:y+h, x:x+w]
    cv2.circle(img, (a, b), 1, (255,255,255), 2)
```

Las variables `a` y `b` servirán de apoyo para localizar el punto central de la detección y, como se explicará en algoritmos sucesivos, serán las variables que se entreguen de forma final al usuario junto con el ángulo instantáneo.

Para finalizar, se almacena la imagen final gracias a `cv2.imwrite`, con las detecciones sobre ella, en formato .png y se muestra en una ventana emergente mediante `cv2.imshow`. por último, se espera el accionamiento de una tecla para detener la ejecución del código.

```
cv2.imwrite("Detección.png", img) # Se guarda la imagen con las bounding boxes
cv2.imshow("Segway", img) # Se muestra la imagen
cv2.waitKey(1)
```

El resultado de la ejecución del presente algoritmo se puede ver en la Figura 34, mostrada a continuación.

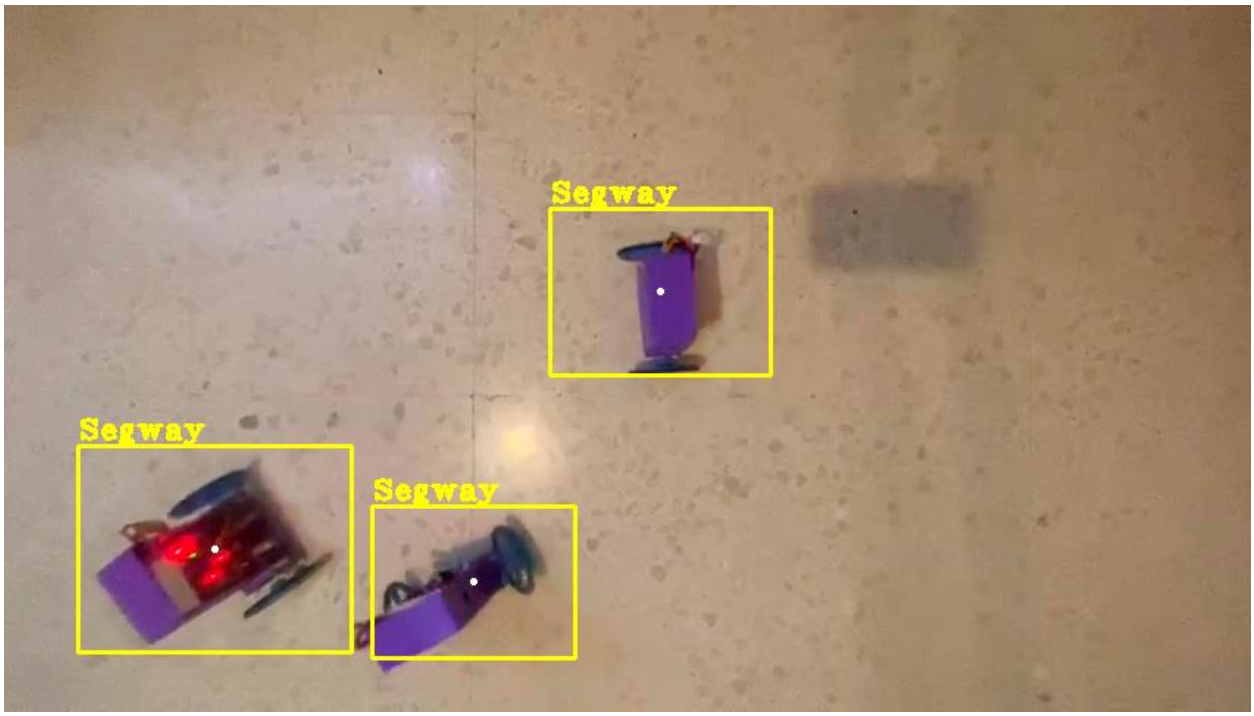


Figura 34. Detección de múltiples elementos en una misma imagen.

4.3. Algoritmo de detección de objetos en streaming (ObjDetection.py)

En este caso se abordará el problema de la detección del Segway en un vídeo en streaming capturado por la cámara de la Raspberry. En este caso la detección se realiza en tiempo real, con el objeto que se desea analizar desplazándose mientras sea analizan todas las imágenes que captura la PiCamera de forma continua. Este algorimo está reflejado en el capítulo Anexos f) al final de este documento.

Este algoritmo mantiene una serie de similitudes con el del epígrafe anterior, pues en vez de analizar una imagen estática se analiza un vídeo, pero se le añaden unas características nuevas que hacen mucho más cómodo e interesante el análisis. Al igual que en el caso anterior, este algoritmo sirve de base para el desarrollo de los posteriores pues se añaden procesos nuevos que se mantendrán en los sucesivos.

Como siempre, se comenzará importando las librerías y declarando las variables que se emplearán a lo largo del algoritmo. En esta ocasión solo se importarán las OpenCV (*cv*) pues no se realizarán operaciones matemáticas complejas como en el caso anterior.

```
# Importación de librerías
import cv2

#####################################################################
# Inicializaciones

# Ubicación del clasificador
path = '/home/pi/Desktop/TFM/CustomHaarRobot/classifier/cascade0.xml'
```

```
cameraNo = 0               # ID Cámara
objectName = 'Segway'      # Nombre a mostrar
frameWidth= 640            # Ancho preview
frameHeight = 480          # Alto Preview
color = (255,0,255)        # Color de la detección
##################################################################

# Setup de la cámara
cap = cv2.VideoCapture(cameraNo)
cap.set(3, frameWidth)
cap.set(4, frameHeight)
cap.set(35, 180)                    # Giro de 180º de la imagen
```

En este caso, se deben inicializar más elementos, pues la configuración tanto de la cámara como del vídeo es mayor al tener que ir grabando mientras se analiza lo grabado. En primer lugar, se define la dirección (`path`) en la que se encuentra el detector. Debido a problemas de incompatibilidad entre los clasificadores creados en Matlab y Python, todos los algoritmos desarrollados en Python trabajarán con el detector entrenado y generado por la app Cascade Trainer GUI.

Posteriormente se definen la cámara a utilizar (`cameraNo`), que en este caso se define así por ser la cámara conectada directamente en la ranura específica de la PiCamera, el nombre del elemento a identificar (`objectName`) así como el color en el que se identificará en el vídeo (`color`) las detecciones realizadas. Finalmente se seleccionará la resolución de la cámara mediante `cap.set` así como el giro de la imagen de la misma.

Con estas definiciones estaría el algoritmo listo para empezar a capturar, pero se ha optado por implementar una serie de mejoras basadas en el uso de las barras de acción mediante las cuales se pueden introducir modificaciones sobre unos parámetros asignados a cada una en tiempo real, durante la propia ejecución del algoritmo. Con esto se consigue un ajuste mayor de la imagen y del clasificador, por tanto, se hace notable una mejoría en las detecciones.

```
def empty(a):
    pass

# Definición de trackbars para ajustar la imagen y el detector a tiempo real
cv2.namedWindow("Result")
cv2.resizeWindow("Result", frameWidth, frameHeight+100)
cv2.createTrackbar("Scale", "Result", 300, 1000, empty)
cv2.createTrackbar("Neig", "Result", 6, 50, empty)
cv2.createTrackbar("Min Area", "Result", 50, 100000, empty)
cv2.createTrackbar("Brightness", "Result", 60, 255, empty)
```

La para la creación de estas barras, lo primero que hay que crear es la ventana sobre la que se mostrará, por tanto, la primera parte de esta sección de código emplea los métodos `cv2.namedWindow` y `cv2.resizeWindow` a través de los cuales se creará esa ventana y se escalará al tamaño correspondiente a la resolución de la cámara, más un espacio extra en el que se ubicarán las barras de acción recientemente creadas.

La vista de la ventana se muestra en la Figura 35.

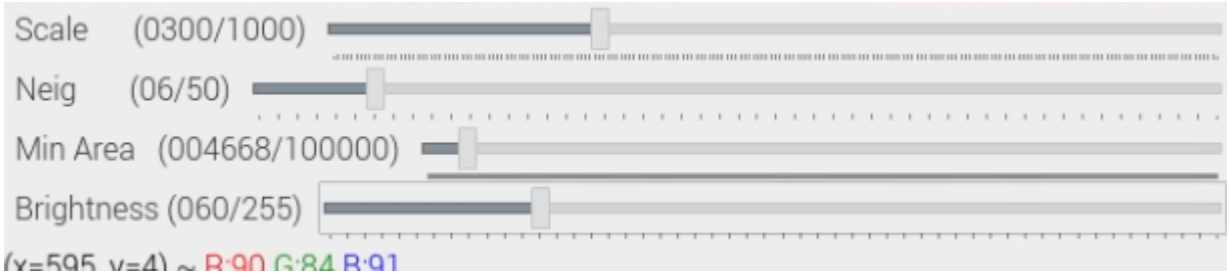


Figura 35. Vista de las Trackbars implementadas.

Estas barras de acción interactivas facilitan la captura en condiciones óptimas de la imagen de la cámara, gracias a la que controla la iluminación (`Brightness`) haciendo posible la corrección de esta característica desde una oscuridad muy elevada hasta un nivel de brillo máximo.

Las tres primeras no influyen sobre la imagen en sí, sino que controlan las características del clasificador en tiempo real, de tal modo que actúan sobre la escala (`Scale`) de las detecciones presentes en la imagen limitando su tamaño con un valor mínimo, el número de vecindades (`Neig`) así como el área mínima que deben tener para ser tenidas en consideración (`Min Area`).

Todas estas barras adquieren el valor inicial seleccionado en el momento en el que se definen mediante `cv2.createTrackbar`, pero para ser utilizadas, deben ir leyéndose esos valores de forma constante en el tiempo, es por eso que, como se explicará en la siguiente sección de código, están incluidas en el bucle infinito gracias al cual funcionará el algoritmo.

Antes del bucle se debe almacenar en una variable el clasificador para su posterior ejecución, por lo que es lo primero que se realiza, antes de la creación del propio bucle.

```
# Se carga el clasificador
cascade = cv2.CascadeClassifier(path)

while True:
    # Modificación del brillo de la cámara
    cameraBrightness = cv2.getTrackbarPos("Brightness", "Result")
    cap.set(10, cameraBrightness)
    # Conversión a escala de grises
    success, img = cap.read()
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    # Selección de escala y número de vecindades necesarios para la detección
    scaleVal = 1 + (cv2.getTrackbarPos("Scale", "Result") /1000)
    neig = cv2.getTrackbarPos("Neig", "Result")
    # Detección del objeto mediante el clasificador
    objects = cascade.detectMultiScale(gray, scaleVal, neig)
    # Localización del objeto y muestra de las bounding boxes con la detección
    for (x, y, w, h) in objects:
        area = w*h
        minArea = cv2.getTrackbarPos("Min Area", "Result")
        if area > minArea:
            cv2.rectangle(img, (x, y), (x+w, y+h), color, 3)
            cv2.putText(img, objectName, (x, y-5),
cv2.FONT_HERSHEY_COMPLEX_SMALL, 1, color, 2)
            roi_color = img[y:y+h, x:x+w]
```

Dentro del bucle se realizan una serie de tareas, muy similares a las del epígrafe anterior:

En primer lugar, se modificará el brillo de la cámara en función del valor de su barra de acción correspondiente, gracias a la combinación de los métodos `cv2.getTrackbarPos` y `cap.set`, que en este caso apunta al parámetro brillo y le asigna el valor leído de la trackbar. El siguiente paso es la lectura de la imagen de la PiCamera mediante `cap.read()` para poder realizar la conversión de esa imagen a escala de grises empleando `cv2.cvtColor` para la modificación del color y `cv2.COLOR_BGR2GRAY` para la selección del cambio de BGR a escala de grises.

El siguiente paso es la lectura de los parámetros de escala y vecindad, del mismo modo que se ha explicado para el brillo, con la salvedad de que, en este caso, por ahora, se almacenan en una variable. Ambos parámetros, junto con la imagen en escala de grises capturada con el brillo seleccionado, configurarán la detección del clasificador, ejecutado con el método `cascade.detectMultiScale` que almacenará las detecciones en la matriz Nx4 explicada en el epígrafe anterior.

El siguiente paso es igual que el que ya se comentó para el caso de la detección sobre una imagen estática preguardada: se coloca un recuadro de las dimensiones almacenadas en la variable `objects` y se colocará el texto elegido en el apartado de definiciones al inicio del algoritmo. El cambio, en este caso, radica en que también se tiene en cuenta el parámetro `Min Area` extraído de su barra correspondiente y comparado con el área de las detecciones hechas por el clasificador, limitando de esta forma el tamaño mínimo de las detecciones realizadas. Esto, en gran medida, ayuda a eliminar errores presentes como pequeños falsos positivos en la imagen y, por tanto, mejora la calidad del clasificador.

```
    # Muestra la vista de la PiCamera para la detección
    cv2.imshow("Result", img)

    # Al pulsar la tecla Q cierra la ventana y detiene la ejecución
    if cv2.waitKey(1) & 0xFF == ord('q'):
        break
```

Finalmente, como en casos anteriores, se muestra la imagen de salida mediante `cv2.imshow` y se espera el accionamiento de la tecla “Q” en el teclado para detener la ejecución del algoritmo.

4.4. Detección de objetos en un vídeo pregrabado (DetecSegway.py):

En este epígrafe se pasará a explicar de forma detallada el algoritmo empleado para la detección e identificación de objetos en un vídeo que se encuentre almacenado en la propia Raspberry. Este algoritmo puede encontrarse en el capítulo Anexos g) al final del presente documento.

En este caso el trabajo será muy similar al caso abordado anteriormente por ello no se explicará de forma tan exhaustiva esas secciones del código ya analizadas. La diferencia en este algoritmo radica en que simplemente se suprime el uso de las barras de acción dado que no hay variabilidad en el brillo de la imagen ni en las detecciones, pues siempre se contará con la misma imagen en movimiento.

Las inicializaciones en este caso son las mismas que en el anterior, salvo que, al capturar la imagen, no se realiza mediante la variable que identifica la cámara (`cameraNo`), sino que se añade al método de lectura `cv2.VideoCapture` la ruta en la que se ubica el vídeo a analizar, el resto permanece

invariable.

```
# Importación de librerías
import cv2
import numpy as np

#####################################################################
# Inicializaciones

# Ubicación del clasificador Segway
path = '/home/pi/Desktop/TFM/CustomHaarRobot/classifier/cascadeRRR.xml'
cameraNo = 0                # ID Cámara
objectName = 'Segway'       # Nombre a mostrar
frameWidth= 640             # Ancho preview
frameHeight = 480           # Alto Preview
color = (0,255,255)         # Color de la detección
#####################################################################

# Lectura del vídeo
cap = cv2.VideoCapture("/home/pi/Desktop/TFM/CustomHaarRobot/videoSegway0.mp4")
```

A continuación, se crea y se reescala la ventana en la que se mostrará el vídeo y se carga el clasificador, igual que se hizo anteriormente, para pasar a iniciar el bucle infinito.

```
def empty(a):
    pass

# Definición de la ventana en la que se mostrará el vídeo
cv2.namedWindow("Result")
cv2.resizeWindow("Result", frameWidth, frameHeight+100)

# Se carga el clasificador
cascade = cv2.CascadeClassifier(path)
```

Ya dentro del bucle, el primer paso es la lectura del vídeo completo, frame a frame. Posteriormente se transforma el vídeo a escala de grises para su análisis y se realizan las detecciones mediante `cascade.detectMultiScale`.

```
while True:

    # Lectura de cada frame del vídeo
    success, img = cap.read()
    # Tratamiento de los frames para su correcta lectura y clasificación
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

    # Detección del objeto mediante el clasificador
    objects = cascade.detectMultiScale(gray)
```

Del mismo modo que se hizo en casos anteriores, se identifica cada una de las detecciones mediante la colocación de un rectángulo del tamaño correspondiente sobre el frame en cuestión siempre y cuando el área detectada supere un umbral que en este caso es fijo y viene definido directamente como `minArea = 1000`. Se ha optado por este valor pues es el que elimina esas pequeñas detecciones consideradas falsos positivos.

```
    # Localización del objeto
    for (x, y, w, h) in objects:
        area = w*h
        a = x+w//2
        b = y+h//2
        minArea = 1000
        if area > minArea:
            cv2.rectangle(img, (x, y), (x+w, y+h), color, 3)
            cv2.putText(img, objectName, (x, y-5),
cv2.FONT_HERSHEY_COMPLEX_SMALL, 1, color, 2)
            roi_color = img[y:y+h, x:x+w]

            # Centro de la detección
            cv2.circle(img, (a, b), 1, (255, 255, 255), 2)
```

Finalmente se muestra la imagen resultante y se comprueba la pulsación de la tecla “Q” en el teclado para detener la ejecución del algoritmo y cerrar las ventanas generadas.

```
    # Muestra del resultado
    cv2.imshow("Result", img)

    # Al pulsar la tecla Q cierra la ventana y detiene la ejecución
    key = cv2.waitKey(1) & 0xFF
    if key == ord('q'):
         break

cv2.destroyAllWindows()
```

El resultado de la ejecución del algoritmo sobre un vídeo en el que aparece el Segway es el que se aprecia en la Figura 36. Es importante destacar que del vídeo estudiado no se han extraído capturas empleadas a su vez en el entrenamiento del detector, pues no tendría sentido alguno dado que se le estaría indicando, dentro del propio vídeo, la localización del Segway en un instante determinado. Esto podría llegar a provocar que el detector no fuese capaz de generalizar y se limitase a detectar aquello que encuentra únicamente en las imágenes del conjunto positivo de entrenamiento.

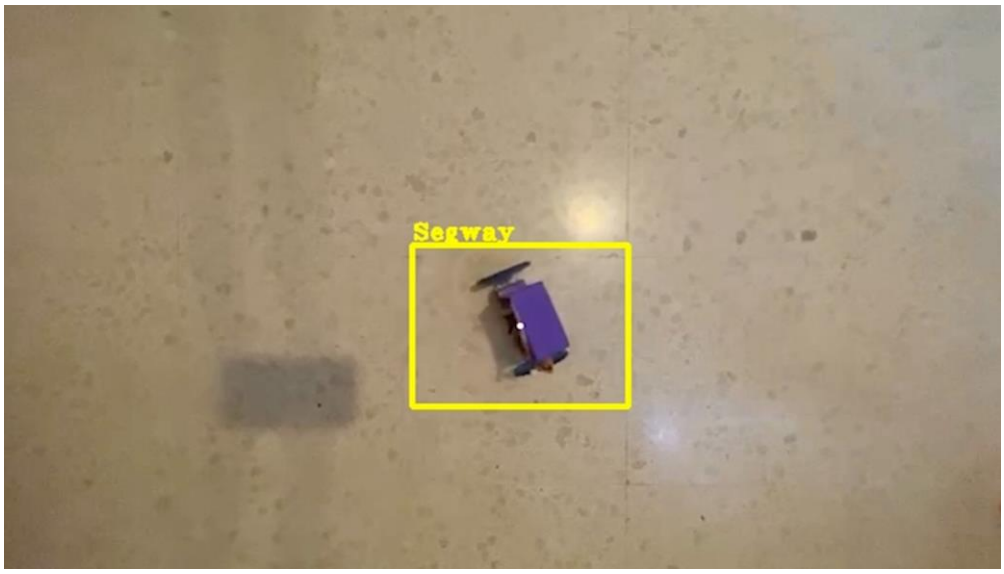


Figura 36. Detección obtenida mediante el algoritmo DetecSegway.py.

4.5. Localización de umbral de color y obtención de su posición y su ángulo instantáneo (DetecFiltradaXYt.py)

Este algoritmo se ha desarrollado en base a los anteriores, pero en este caso sí que se incluyen modificaciones llamativas, pues en este caso se quiere obtener la posición y el ángulo del propio Segway en tiempo real. El algoritmo puede encontrarse en el capítulo Anexos h). En este caso no se realiza ninguna detección, pues se ha optado por probar un método diferente de identificación del Segway. En este caso se trabajará con el umbral de color que segmenta el Segway respecto del fondo, es decir, en este caso se buscarán combinaciones del espacio BGR que resulten en tonos morados o similares.

Para la obtención de este umbral se ha estudiado el vídeo sobre el que se va a trabajar, observando los colores identificados en el Segway cuando se encuentra en diferentes posiciones. Esta operación se ha realizado analizando las imágenes que forman el conjunto positivo de imágenes de entrenamiento, dado que presentan una amplia variedad de posiciones, así como de niveles de iluminación sobre el propio vehículo.

De este trabajo se desprenden los valores definidos junto con el resto de las inicializaciones del algoritmo, es decir, que el color morado se ubica, en esta situación, entre los valores `(50, 0, 60)` y `(170, 170, 170)`. Teniendo esto en cuenta, se procede a analizar el código de manera más exhaustiva.

Lo primero que resalta es que las importaciones de librerías en este caso son mayores, pues el trabajo que se desea realizar requiere de operaciones más complejas que los llevados a cabo hasta este punto. En esta ocasión se emplearán las OpenCV (*cv*) como en los algoritmos explicados anteriormente, para el tratamiento de imágenes, así como se han utilizado las Numpy (*np*) para operaciones matemáticas simples. Además de las ya comentadas, se necesita una serie de librerías más extensa:

- *imutils*: para operaciones simples con imágenes, apoyando a las que ya se pueden realizar con las OpenCV.

- *argparse*: empleada para el procesamiento de elementos conocidos como "argumentos" y para analizar líneas de comandos. En este caso se empleará junto con *deque* para definir un argumento que se utilizará para almacenar las posiciones anteriores del Segway para poder crear un registro con el que ir comparando la posición actual y así obtener el ángulo instantáneo.
- *math*: ha sido utilizada para realizar operaciones matemáticas más complejas para las que *numpy* no tiene alcance suficiente.
- *collections*: más concretamente la sección *deque* que permite la creación de colas doblemente terminadas, es decir, un elemento que permite un abanico de operaciones más amplio que uno tipo *list*, pues las *deque* permiten añadir elementos en cualquier posición, así como hacer crecer o disminuir la cola en cualquiera de los dos extremos. En este caso se ha utilizado apoyado en *argparse* para almacenar las N posiciones anteriores del Segway, con el objetivo de obtener el ángulo instantáneo del Segway.

```
# Importación de librerías
import cv2
import imutils
import argparse
import math
import numpy as np
from collections import deque
```

Una vez explicado el uso de las librerías para este algoritmo, la inicialización y definición de parte de las variables ya se ha visto en epígrafes anteriores, por lo que en esta ocasión la explicación se centrará en las nuevas variables.

Como se ha comentado en la explicación de las librerías, este algoritmo se ha desarrollado en base a la combinación del uso de dos librerías que permiten crear un buffer que almacenará las últimas posiciones del robot. Las variables que hacen referencia a esto son `ap`, que se utilizará para crear el argumento que será un carácter entero cuyo valor por defecto será 32, `args` variable tipo diccionario que almacenará el argumento junto con su valor y `pts` que será una variable tipo cola doblemente terminada en la que se almacenarán la posición del Segway en los últimos 32 frames del vídeo.

También se inicializa un contador (`counter`) que será utilizado para contar el número de ejecuciones completas del bucle realizadas, las posiciones instantáneas en X e Y del propio Segway (`dX`, `dY`) así como el ángulo instantáneo (`t`) además de una variable que definirá la hipotenusa del triángulo rectángulo formado por los valores de `dX` y `dY` y así obtener el valor del ángulo `t`. Además, como se ha explicado al inicio de este epígrafe, se define también el umbral superior e inferior (`purplemin` y `purplemax`) del color a buscar y la ruta en la que se encuentra el vídeo a analizar (`cap`).

```
#################################################################
# Inicializaciones

cameraNo = 0                # ID Cámara
objectName = 'Segway'       # Nombre a mostrar SEGWAY
frameWidth= 640             # Ancho preview
```

```
frameHeight = 480          # Alto Preview

# Definición del buffer que almacenará los puntos en los que ha estado el Segway
ap = argparse.ArgumentParser()
ap.add_argument("-b", "--buffer", type=int, default=32, help="max buffer size")
args = vars(ap.parse_args())
pts = deque(maxlen=args["buffer"])
counter = 0

(dX, dY) = (0, 0)
t = 0
dH = 0

# Niveles de color a buscar en la imagen
purplemin = (50,0,60)
purplemax = (170,170,170)
######################################################################

# Lectura del vídeo
cap = cv2.VideoCapture("/home/pi/Desktop/TFM/CustomHaarRobot/videoSegway0.mp4")
```

A continuación, como se ha hecho en ocasiones anteriores, se crea y se reescala la ventana en la que se mostrará el vídeo y comenzará la ejecución del bucle infinito.

```
def empty(a):
    pass

# Definición de la ventana en la que se mostrará el vídeo
cv2.namedWindow("Result")
cv2.resizeWindow("Result", frameWidth, frameHeight+100)
```

La primera acción a realizar dentro del bucle es la lectura del vídeo frame a frame. Posteriormente estos frames serán tratados mediante, primero un filtro gaussiano y posteriormente convertido de BGR a HSV para obtener una imagen con la que se pueda trabajar sin problemas. El empleo del filtro gaussiano está justificado por la búsqueda de reducción de ruido de alta frecuencia presente en la imagen y la conversión al espacio HSV porque se hacen accesibles elementos como el tono, la saturación y el valor del color. Este espacio de color se representa en la Figura 37.

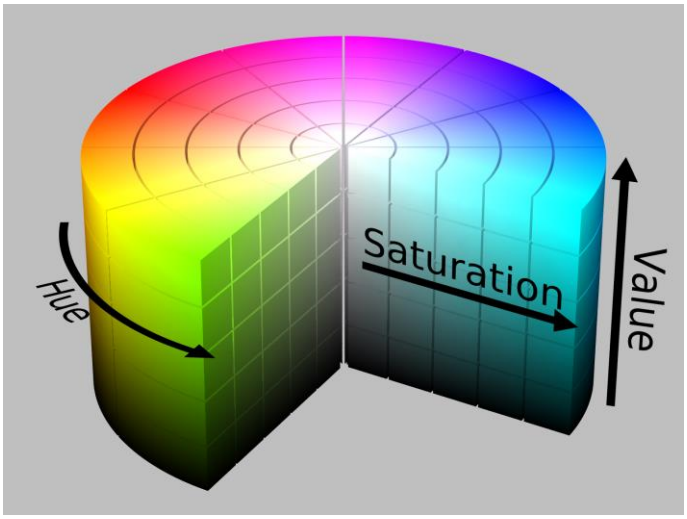


Figura 37. Espacio de color HSV.

Posteriormente se pasa a buscar los píxeles de la imagen que se encuentran dentro del rango de valores definidos como umbrales para el color morado (`mask`), encontrando los que se muestran en la Figura 38.

Figura 38. Imagen filtrada con el umbral de color morado.

Los píxeles presentes en esta imagen serán, en primero lugar erosionados gracias al método `cv2.erode`, en poca medida (Figura 39), para pasar a ser dilatados en gran medida (Figura 40). Esto facilita la localización de contornos en la imagen, que es el siguiente paso del algoritmo.

Figura 39. Píxeles del umbral de color morado erosionados.

La erosión es el proceso mediante el cual el conjunto de píxeles ve reducido su tamaño erosionando en función del parámetro `iterations`, literalmente, los píxeles de las regiones más externas de aquellas zonas coloreadas de blanco en la imagen superior. Cuanto mayor sea el valor de `iterations`, mayor será la cantidad de píxeles que desaparezcan de la imagen original. La región roja de la imagen indica de forma aproximada una de las zonas en las que, al erosionar la imagen inicial, hay píxeles del umbral de color que desaparecen.

Figura 40. Píxeles del umbral de color morado dilatados después de haber sido erosionados.

En el caso de la dilatación (`cv2.dilate`), el proceso es opuesto al recién explicado. Ahora los píxeles pasarán a "crecer" en función del parámetro `iterations`, que en este caso posee un valor más elevado. Este crecimiento de los píxeles hace que la región blanca de la imagen crezca y pase a

estar unida, de modo que la detección de los contornos es mucho más simple.

El proceso de detección de los contornos se realiza con el método `cv2.findContours` al cual se le entrega una copia de la máscara que se ha erosionado y dilatado (Figura 40), además de la elección del contorno externo (`cv2.RETR_EXTERNAL`) y el método de búsqueda de los mismos (`CHAIN_APPROX_SIMPLE`).

Se ha elegido la opción del método simple de búsqueda de contornos pues no se necesitan más que los puntos más externos del contorno para poder crear un rectángulo utilizando esos puntos como base. Los valores del contorno quedan almacenados en la variable `cotr` que luego será estudiada para seleccionar el contorno mayor y así poder localizar el Segway.

```
while True:

    # Lectura de cada frame del vídeo
    success, img = cap.read()

    # Tratamiento de los frames para su correcta lectura y localización
    blurred = cv2.GaussianBlur(img, (9,9), 0)
    hsv = cv2.cvtColor(blurred, cv2.COLOR_BGR2HSV)

    # Tratamiento de los píxeles en el rango de color definido para generar un
único contorno detectable
    mask = cv2.inRange(hsv, purplemin, purplemax)
    mask = cv2.erode(mask, None, iterations=2)
    mask = cv2.dilate(mask, None, iterations=20)
    cotr = cv2.findContours(mask.copy(), cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIM
PLE)
    cotr = imutils.grab_contours(cotr)
    center = None
```

Una vez se cuenta con el contorno localizado, el proceso es similar al de la identificación mediante un rectángulo del Segway, como se ha explicado en algoritmos anteriores. En este caso se aplica el mismo procedimiento con la salvedad de que la posición, anchura y altura (`x, y, w, h`) del rectángulo mayor se obtiene mediante `cv2.boundingRect`. Obtenidos estos valores se calcula la posición del centro de ese rectángulo (`a, b`) pues será la posición instantánea del Segway.

En esta ocasión se utilizarán los momentos de las OpenCV (`cv2.moments`), pues ayudarán en el proceso del seguimiento del robot. Para entender qué es el concepto de momento de una imagen, primero se debe comprender el concepto de centroide [17].

El centroide de una figura es la media aritmética (o promedio) de todos los puntos que forman esa figura. Suponiendo que el conjunto venda dado por n puntos x_1 … x_n, el centroide corresponde con:

$$c = \frac{1}{n} \sum_{i=1}^{n} x_i \qquad (10)$$

En el caso de una imagen, la figura corresponderá con un conjunto de píxeles y el centroide con la media ponderada de todos esos píxeles.

Los momentos de una imagen son una medida particular de la media ponderada de la intensidad de los píxeles de la imagen, que pueden llevar a localizar propiedades de la imagen como el radio, el área o el centroide. Para localizar el centroide de una imagen, el primer paso es binarizarla para posteriormente aplicar las siguientes ecuaciones:

$$C_x = \frac{M_{10}}{M_{00}} \qquad C_y = \frac{M_{01}}{M_{00}} \tag{11}$$

Donde C_x es la coordenada X y C_y la coordenada Y del centroide, mientras que M denota el momento.

Una vez entendido esto, se aplica lo explicado y se calculan tanto los momentos (`M`) como el centroide (`center`) de la imagen.

```
    if len(cotr) > 0:

        # Se busca el contorno mayor
        c = max(cotr, key=cv2.contourArea)
        (x, y, w, h) = cv2.boundingRect(c)
        a = x+w//2
        b = y+h//2
        M = cv2.moments(c)
        center = (int(M["m10"] / M["m00"]), int(M["m01"] / M["m00"]))
```

El siguiente paso que se debe realizar es la colocación del rectángulo (`cv2.rectangle`) que englobe todos los píxeles correspondientes a la región del umbral morado centrado en el centroide, de tal modo que en todo momento se identifique inequívocamente el Segway en toda su trayectoria. Además del rectángulo también se identificará el centroide con un círculo (`cv2.circle`) y se acumulará la nueva posición del Segway, correspondiente con el centroide, en el buffer de posiciones (`pts`).

```
        if cv2.contourArea(c) > 100:
            cv2.rectangle(img, (x, y), (x+w, y+h), (0,255,0), 2)
            # Centro del cuadrado (posición instantánea)
            cv2.circle(img, (a, b), 1, (255,255,255), 2)
            pts.appendleft(center)
```

A continuación, se ejecuta un pequeño bucle sobre los puntos ya almacenados en el buffer con el objetivo de comprobar, por parejas, si alguno de ellos no tiene ningún valor (`None`) en ese caso, el bucle continúa.

En caso de que existan suficientes puntos con valor definido en él, se puede calcular el desplazamiento del Segway comparando el valor de la posición actual (`x=pts[i][0]`, `y=pts[i][1]`) respecto a la última posición almacenada en el buffer (`xf=pts[-10][0]`, `yf= pts[-10][1]`). Este movimiento quedaría almacenado en las variables `dX` para el desplazamiento en X y `dY` para el desplazamiento en Y. Además, para corregir el cambio del origen de coordenadas y poder trabajar con el origen más común, situado en el extremo inferior izquierdo de la imagen, se sobreescriben los desplazamientos como se muestra en la sección de código inferior.

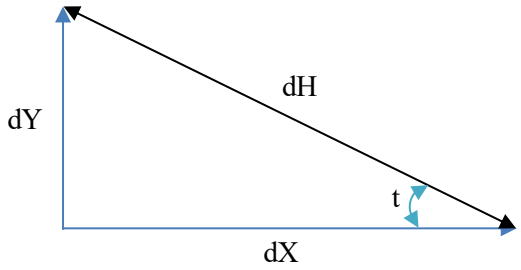
Es muy importante hacer una apreciación en este punto, pues lo más rápido sería pensar en el desplazamiento de un objeto como el desplazamiento entre un frame y el anterior. Ese sería un pensamiento lógico en caso de contar con un objeto que se mueve a una velocidad muy alta, es decir, presenta una gran variación en las posiciones de dos frames consecutivos, no obstante, ese no es el caso que se está estudiando. En caso de presentar variaciones pequeñas en esas posiciones estudiadas, el resultado presentaría un nivel de ruido muy alto, así como una variabilidad incontrolable. Es por eso que se propone emplear un buffer con capacidad de almacenar múltiples posiciones, se busca ser capaz de computar variaciones en la posición tan pequeñas que el propio ojo humano no sería capaz de detectar.

```
    # Se van almacenando en el buffer las posiciones anteriores del Segway
    for i in np.arange(1, len(pts)):
        if pts[i - 1] is None or pts[i] is None:
            continue

        if counter >= 10 and i == 1 and pts[-10] is not None:

            dX = pts[-10][0] - pts[i][0]
            dY = pts[-10][1] - pts[i][1]
            dX = float(-dX)
            dY = float(-dY)
```

La próxima sección de código define, superado un número mínimo de posiciones almacenadas en el buffer, el valor del ángulo instantáneo `t`, medido en sentido positivo del eje X y vertical ascendente del eje Y. El algoritmo está basado en la siguiente situación:



Por lo que la respuesta al valor de `t` se haya mediante una simple formulación matemática:

$$dH = \sqrt{dX^2 + dY^2} \tag{12}$$

$$\cos t = \frac{dX}{dH}\ [rad]; \qquad t = \frac{\cos^{-1}\left(\frac{dX}{dH}\right) 360}{2\pi}\ [°] \tag{13}$$

De este modo se obtendría el valor del ángulo buscado. Para corregir la variación del origen de coordenadas se hace necesaria la inclusión de la comprobación del signo de dY, puesto que, si este es positivo, el ángulo debe ser el correspondiente al sentido opuesto al calculado.

```
# Se calcula el ángulo que posee el Segway
if counter >= 11:
    dH = math.sqrt(dX**2 + dY**2)
    t = (((math.acos(dX / float(dH))) * 360) / (2 * math.pi))
    if np.sign(dY) == 1:
        t = -t
```

Para finalizar los cálculos de este algoritmo, se dibujará una estela detrás del Segway, para mostrar de una forma más visual aquellos puntos por los que ha pasado. Este proceso se realizará en dos fases:

- En la primera se definirá la anchura (`thickness`) de la estela en función de la cercanía al Segway, siendo más ancha en este caso y más estrecha cuanto más lejos se encuentre. Esto se logra mediante la siguiente formulación:

$$thickness = \sqrt{\frac{elementos\ en\ el\ buffer}{posición\ actual}} * 2.5 \tag{14}$$

  Por lo tanto, teniendo en cuenta que el buffer puede almacenar hasta 32 elementos, la anchura de la estela irá desde aproximadamente 14 en el centro del Segway hasta 2.5 al final de la estela.

- Por otro lado, se definirá la línea (`cv2.line`) de la estela sobre cada uno de los frames, mostrando en cada uno la posición anterior y la actual, con el color y el ancho definidos.

```
# Creación de la estela que ha dejado el Segway
thickness = int(np.sqrt(args["buffer"] / float(i + 1)) * 2.5)
cv2.line(img, pts[i - 1], pts[i], (0,0,255), thickness)
```

Finalmente se mostrará sobre cada uno de los frames del vídeo la posición en tiempo real en la zona inferior izquierda de la imagen, así como el ángulo instantáneo del Segway justo debajo. Estas operaciones serán posibles gracias a `cv2.putText`, donde se definirá la imagen sobre la que se desea escribir, el texto junto con las variables a mostrar, la posición la fuente, la escala, el color y el grosor.

Además, se mostrarán cada uno de los frames en dos ventanas diferentes, en una () se mostrará la imagen binarizada con la que se localiza el centroide del objeto gracias a los momentos, y en la otra la imagen el vídeo del Segway junto con las anotaciones ya comentadas.

Como paso final, se incrementa el contador y, al igual que en los casos anteriores, se espera un accionamiento de la tecla "Q" del teclado para detener la ejecución y cerrar todas las ventanas creadas.

```
    # Se muestra sobre el vídeo tanto la posición como el ángulo instantáneo
    cv2.putText(img, "Posicion X: {}, Y: {}".format(a, b), (10, img.shape[0] -
 20), cv2.FONT_HERSHEY_SIMPLEX, 0.35, (255,255,255), 1)
    cv2.putText(img, "Angulo: {:.2f} grados".format(t), (10, img.shape[0] -
 10), cv2.FONT_HERSHEY_SIMPLEX, 0.35, (255,255,255), 1)

    # Se muestran tanto la máscara de píxeles de color como la detección
    cv2.imshow("Máscara", mask)
    cv2.imshow("Result", img)

    # Al pulsar la tecla Q cierra la ventana y detiene la ejecución
    key = cv2.waitKey(1) & 0xFF
    counter += 1
    if key == ord('q'):
        break
cv2.destroyAllWindows()
```

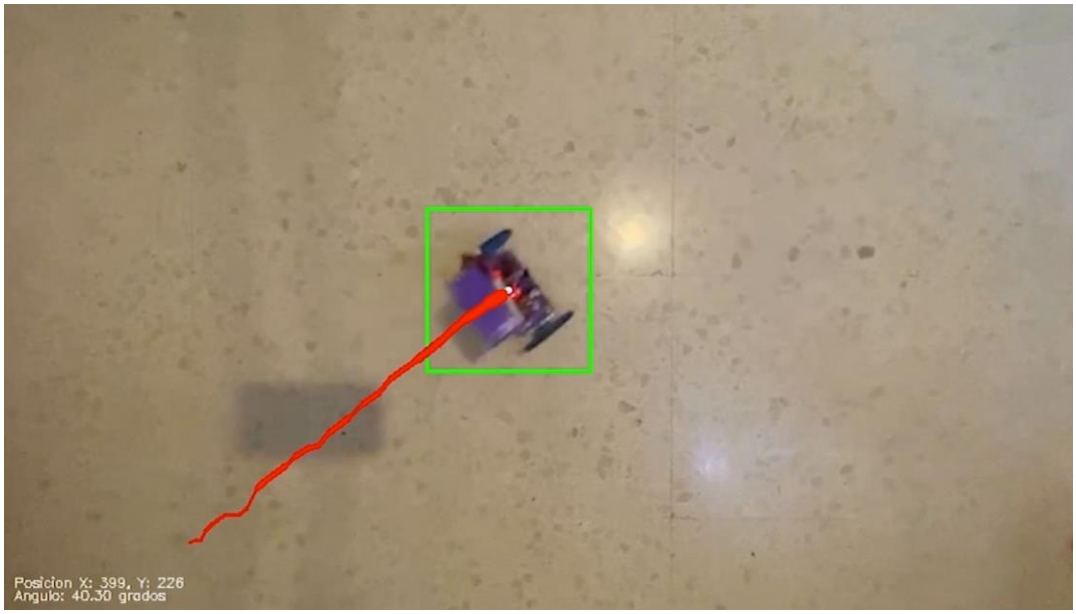


Figura 41. Resultado final de la ejecución del algoritmo DetecFiltradaXYt.py.

El resultado final del algoritmo se muestra en la Figura 41.

4.6. Localización de umbral de color y obtención y entre al usuario de su posición instantánea (DetecFiltradaSOLOXY.py)

Este algoritmo es exactamente igual que el anterior con la salvedad de que en este caso se entregan al usuario la posición y el ángulo instantáneos, en lugar de mostrarlos sobreimpresos en el vídeo en una ventana emergente. Puede encontrarse completo en el capítulo Anexos i).

Tal y como se destacó al inicio del presente documento, el objetivo de este trabajo era ser capaz de dotar al usuario de la posición y el ángulo que poseía el Segway en todo momento. Esto se consigue gracias a la ejecución de este algoritmo completo, que puede realizarse desde una IDE de desarrollo de código como Spyder3, o desde la propia ventana de comandos de la propia Raspberry. Este segundo punto es bastante más interesante, ya que se podría conectar la placa mediante VNC o un cable ethernet a otro ordenador, ubicado en la oficina o en el despacho, desde el que observar los resultados además de controlar la Raspberry si se hiciese necesario. Además, contando ya con los datos extraídos, se podría trabajar con ellos, o entregarlos a otro elemento, como podría ser, por ejemplo, otro Segway, para evitar colisiones o para diseñar trayectorias.

Es importante destacar, que en esta ocasión se ha trabajado también con un archivo de vídeo, pero se ha declarado la variable `cameraNo` que hace referencia a la cámara conectada directamente sobre la Raspberry, simplemente habría que modificar el método que selecciona el vídeo a estudiar (`cap = cv2.VideoCapture`), cambiando la dirección en la que está almacenada el vídeo por la variable definida anteriormente. Como se explicó al inicio de este documento, el proyecto está planteado para que la placa se encuentre en el techo con la cámara enfocando el suelo, para obtener una vista cenital del o de los Segways que se vayan a utilizar, similar a la vista en la Figura 41.

Profundizando en el código, destaca que en esta ocasión se empleará la librería `time` para controlar el tiempo de ejecución del algoritmo y más concretamente, la diferencia de tiempo con la que se entrega una posición y ángulo nuevos. Dentro de las inicializaciones, resaltar que solamente se declara la variable `start_time`, en la que se almacenará el instante temporal inicial en el que se ejecuta el código, para compararlo con el instante en el que se entreguen los valores de la posición y el ángulo y así obtener el segundo exacto en el que se ha entregado ese par de datos concreto.

```
# Importación de librerías
import cv2
import imutils
import argparse
import math
import numpy as np
from collections import deque
import time

######################################################################
# Inicializaciones

cameraNo = 0                # ID Cámara

# Definición del buffer que almacenará los puntos en los que ha estado el Segway
ap = argparse.ArgumentParser()
```

```
ap.add_argument("-b", "--buffer", type=int, default=32, help="max buffer size")
args = vars(ap.parse_args())
pts = deque(maxlen=args["buffer"])
counter = 0
(dX, dY) = (0, 0)
t = 0
dH = 0
start_time = time.time()

# Niveles de color a buscar en la imagen
purplemin = (50,0,60)
purplemax = (170,170,170)
######################################################################

# Lectura del vídeo
cap = cv2.VideoCapture("/home/pi/Desktop/TFM/CustomHaarRobot/videoSegway0.mp4")

def empty(a):
    pass

# Definición de la ventana
cv2.namedWindow("Result")
```

Una vez declaradas todas las variables e inicializaciones, se pasa al bucle infinito, que funciona exactamente igual que el algoritmo explicado en el epígrafe anterior, salvo que en esta ocasión no se colocan ni los rectángulos ni el círculo central puesto que no se muestra el vídeo y se estaría perdiendo tiempo en la ejecución de este algoritmo. El resto del bucle permanece inalterado, pues todo el código es imprescindible para el cálculo de la posición y el ángulo del Segway.

```
while True:

    # Lectura de cada frame del vídeo
    success, img = cap.read()

    # Tratamiento de los frames para su correcta lectura y localización
    blurred = cv2.GaussianBlur(img, (9,9), 0)
    hsv = cv2.cvtColor(blurred, cv2.COLOR_BGR2HSV)
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

    # Tratamiento de los píxeles en el rango de color definido para generar un
único contorno detectable
    mask = cv2.inRange(hsv, purplemin, purplemax)
    mask = cv2.erode(mask, None, iterations=2)
    mask = cv2.dilate(mask, None, iterations=20)
    cotr = cv2.findContours(mask.copy(), cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIM
PLE)
    cotr = imutils.grab_contours(cotr)
    center = None
```

```
    if len(cotr) > 0:
        # Se busca el contorno mayor
        c = max(cotr, key=cv2.contourArea)
        (x, y, w, h) = cv2.boundingRect(c)
        a = x+w//2
        b = y+h//2
        M = cv2.moments(c)
        center = (int(M["m10"] / M["m00"]), int(M["m01"] / M["m00"]))

        if cv2.contourArea(c) > 100:
            # Posición instantánea del Segway almacenada en el buffer
            pts.appendleft(center)

    # Se van almacenando en el buffer las posiciones anteriores del Segway
    for i in np.arange(1, len(pts)):
        if pts[i - 1] is None or pts[i] is None:
            continue

        if counter >= 10 and i == 1 and pts[-10] is not None:

            dX = pts[-10][0] - pts[i][0]
            dY = pts[-10][1] - pts[i][1]
            dX = float(-dX)
            dY = float(-dY)

            # Se calcula el ángulo que posee el Segway
            if counter >= 11:
                dH = math.sqrt(dX**2 + dY**2)
                t = (((math.acos(dX / float(dH))) * 360) / (2 * math.pi))
                if np.sign(dY) == 1:
                    t = -t
```

La sección de código mostrada a continuación es la única que se ha incluido en el bucle original, pues es la que entrega al usuario los valores buscados. En el caso mostrado debajo, se define una sentencia `if` para reducir la velocidad a la que se muestra la información, pero puede ser suprimida para que la información se muestre de forma más rápida, haciendo que se tenga un control más preciso de ambos parámetros.

En la Figura 42 y en la Figura 43 se muestra la diferencia de tiempos de entrega de datos manteniendo el algoritmo como se muestra debajo, o suprimiendo la sentencia `if`.

Figura 42. Entrega de datos con la sentencia if.

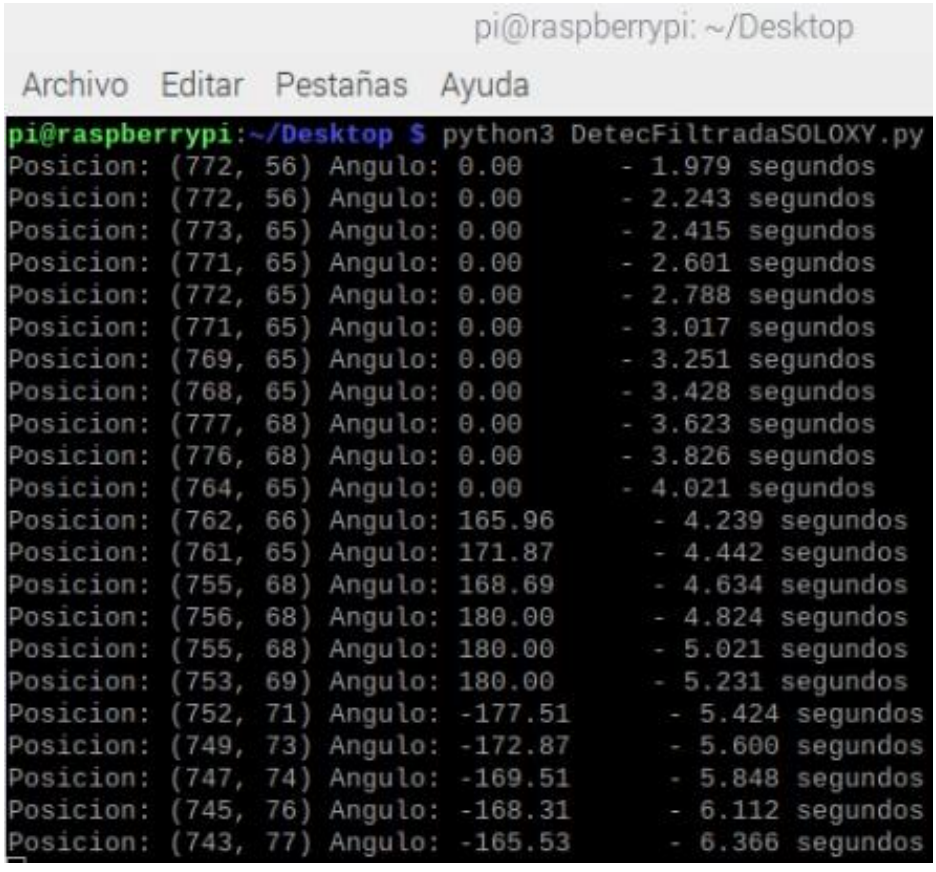


Figura 43. Entrega de datos sin la sentencia if.

Es fácil calcular el tiempo de respuesta de cada uno de ellos mediante cálculos simples, de modo que se obtiene que en el primer caso se contaría con un tiempo de respuesta que se puede ver en la Tabla 1.

Tabla 1. Tiempo de respuesta con la sentencia if.

Marca temporal	Diferencia
1,975 s	-
3,957 s	1,982 s
5,858 s	1,901 s
7,947 s	2,089 s
9,891 s	1,944 s
11,896 s	2,005 s
13,880 s	1,984 s
15,927 s	2,047 s
17,964 s	2,037 s
19,999 s	2,035 s
21,932 s	1,933 s
23,715 s	1,783 s
25,839 s	2,124 s

De aquí, haciendo la media aritmética de las diferencias entre las marcas temporales, se obtiene que el tiempo de respuesta aproximado del algoritmo con la sentencia if incluida es de 1,988 segundos.

En la Tabla 2 se aprecia la gran diferencia que existe en el tiempo de ejecución, pues la sentencia ya comentada en este caso no actúa entregando un dato cada diez vueltas completas del bucle, considerando el valor del contador, sino que proporciona un dato en cada ejecución del mismo. A continuación, se puede ver la gran variación temporal respecto al primer caso.

En el segundo caso, realizando los mismos cálculos, se obtiene un tiempo de respuesta aproximado de unos 0,20 segundos, que se asemeja a lo previamente medido, dado que en este caso se tarda en obtener un nuevo dato una décima parte del tiempo calculado en el caso anterior.

Tabla 2. Tiempo de respuesta sin la sentencia if.

Marca temporal	Diferencia
1,979 s	-
2,243 s	0,264 s
2,415 s	0,172 s
2,601 s	0,186 s
2,788 s	0,187 s
3,017 s	0,229 s
3,251 s	0,234 s
3,428 s	0,177 s
3,623 s	0,195 s
3,826 s	0,203 s
4,021 s	0,195 s
4,239 s	0,218 s
4,442 s	0,203 s
4,634 s	0,192 s
4,824 s	0,190 s
5,021 s	0,197 s
5,231 s	0,210 s
5,424 s	0,193 s
5,600 s	0,176 s
5,848 s	0,248 s
6,112 s	0,264 s
6,366 s	0,254 s

```
    # Cada diez posiciones almacenadas se muestra la posición por la terminal
    if (counter % 2) == 0:
        print("Posicion:", center, "Angulo: {:.2f}".format(t), "     -{:.3f}
segundos".format(time.time() - start.time))
```

Finalmente, igual que en todos los algoritmos ya comentados, se incrementa el contador y se espera la pulsación de la tecla "Q" en el teclado para detener la ejecución y cerrar las ventanas.

```
    # Al pulsar la tecla Q cierra la ventana y detiene la ejecución
    key = cv2.waitKey(1) & 0xFF
    counter += 1
    if key == ord('q'):
         break

cv2.destroyAllWindows()
```

4.7. Algoritmo auxiliar de grabación de los vídeos mostrados (SaveVideo.py)

En esta ocasión se explicará, no un algoritmo ejecutable como tal, sino uno empleado como apoyo para los que ya se han comentado. El objetivo de este algoritmo es la grabación de los vídeos mostrados en las ventanas emergentes generadas en al ejecutar los códigos anteriores. Se puede ver el algoritmo completo en el capítulo Anexos, epígrafe j).

A continuación, se pasará al análisis exhaustivo del algoritmo, en el que se encuentra, como siempre y en primer lugar, la importación de librerías y las inicializaciones. Para este algoritmo será necesario contar con las OpenCV para el tratamiento de imágenes y las numpy para operaciones matemáticas simples.

Respecto a las inicializaciones, en este caso será necesario definir el códec del vídeo (`fourcc`) mediante `cv2.VideoWriter_fourcc` para poder realizar la grabación en formato .avi, así como la variable que almacenará el vídeo (`out`). Para la grabación de vídeo en otros formatos será necesario crear un códec diferente, pues uno debe existir un compromiso entre ellos. En este caso se declara la variable `cap` que mostrará la vista de la cámara de la Raspberry para tener un elemento que grabar, pero al aplicar este algoritmo auxiliar a uno de los anteriormente explicados existirá una variable diferente encargada de proporcionar el vídeo que se desea guardar, por lo que se debe tener identificada.

```
# Importación de librerías
import numpy as np
import cv2

# Lectura del vídeo
cap = cv2.VideoCapture(0)

# Definición del códec y creación del objeto de grabación del vídeo
fourcc = cv2.VideoWriter_fourcc(*'XVID')
out = cv2.VideoWriter('output.avi', fourcc, 10.0, (640, 480))
```

El siguiente paso será la declaración del bucle infinito, que simplemente mostrará la vista de la cámara y, si la lectura del frame es correcta (`if ret==True:`), habilitará la grabación del vídeo

mediante `out.write(frame)`, de este modo, quedará el vídeo almacenado en la variable inicializada anteriormente. En este caso, esa variable creará un archivo `'output.avi'` en el escritorio de la Raspberry e irá escribiendo en él cada uno de los frames cuya lectura haya sido correcta con el formato definido en la variable `out`, que sería el nombre del archivo, el códec de vídeo, el número de frames por segundo que se reproducirán y la resolución del vídeo.

Para finalizar el bucle y como apoyo en este caso, se muestra la vista de la PiCamera facilitando el trabajo. No obstante, igual que ya se ha comentado con otras, esta línea de código tampoco será necesaria pues los algoritmos anteriores ya cuentan con ella. También se comprobará la pulsación de la tecla "Q" en el teclado para detener la ejecución del algoritmo, cerrar las ventanas y limpiar las variables utilizadas.

```
# Bucle infinito de grabación
while (cap.isOpened()):
    ret, frame = cap.read()
    if ret==True:
        out.write(frame)

        # Muestra en pantalla la vista de la PiCamera
        cv2.imshow('frame', frame)

        # Al pulsar la tecla Q cierra la ventana y detiene la ejecución
        if cv2.waitKey(1) & 0xFF == ord('q'):
            break
    else:
        break

cap.release()
out.release()
cv2.destroyAllWindows()
```

Por lo tanto, las líneas con mayor interés de este algoritmo serían la definición del códec y la variable que almacenará el vídeo (`out`) y el método encargado de escribir frame a frame en esta última (`out.write`), que siempre debe encontrarse dentro del bucle.

5 RESULTADOS

Este último capítulo se centrará en el estudio de los resultados obtenidos con los diferentes algoritmos y, sobre todo, con los métodos de entrenamiento de los detectores con el fin de conocer el mejor modo de entrenamiento y clasificador generado.

Este trabajo se realizará empleando los diferentes clasificadores sobre bancos de imágenes previamente creados con ayuda de los algoritmos encontrados en los epígrafes a) y b) del capítulo Anexos. Los entornos de trabajo serán la propia aplicación Cascade Trainer GUI ya que proporciona una herramienta veloz y fiable y Matlab, dado que los clasificadores creados en este software no son reconocidos por la primera aplicación propuesta. Se testeará cada clasificador en el software en el que han sido creados para evitar cualquier tipo de problema de incompatibilidades, ya que ha sido imposible adaptar los archivos .xml de un software al otro. Se muestra en la Figura 44 la vista general de la app utilizada para los clasificadores empleados en Python.

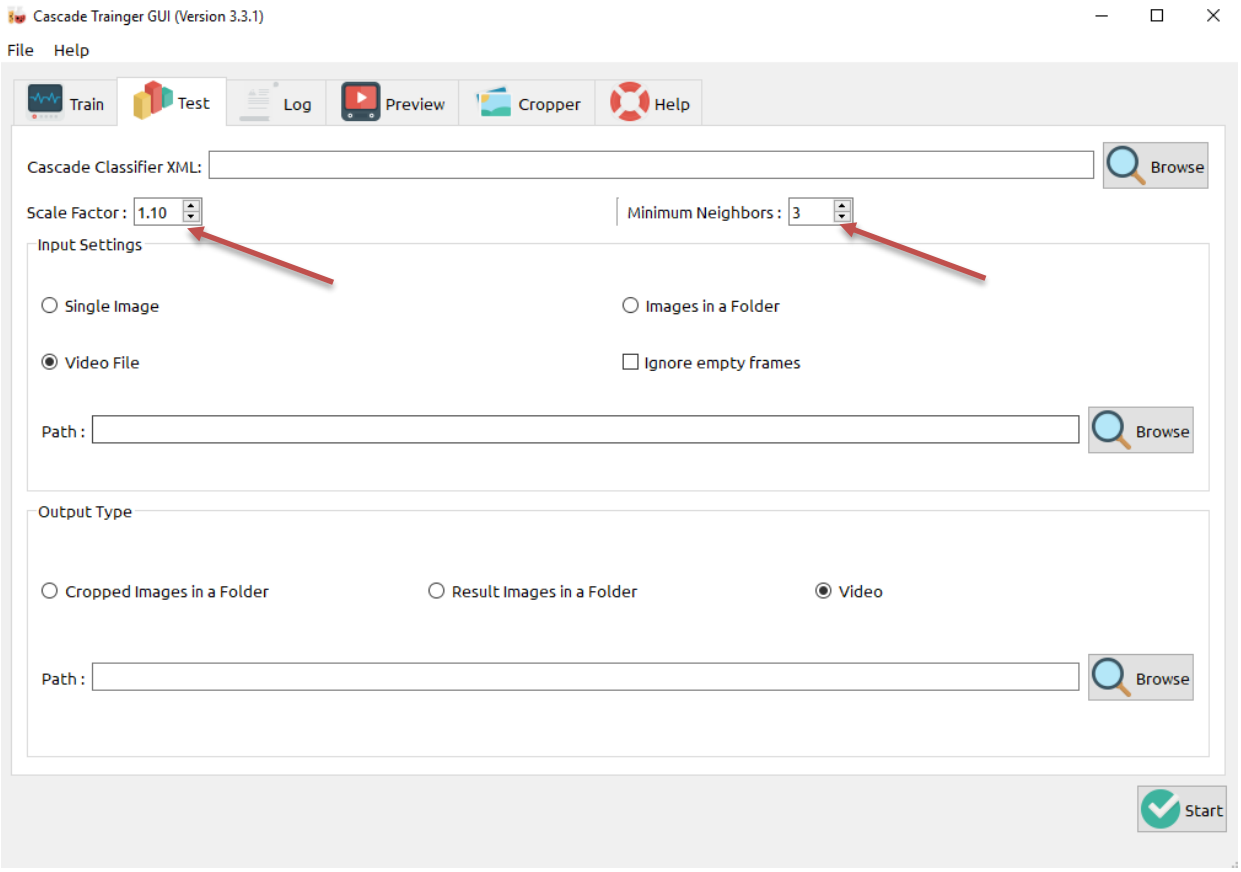


Figura 44. Vista general de la aplicación para el testeo de los clasificadores.

Se puede comprobar el acceso a las variables modificadas y comentadas en el epígrafe 4.3 como son el factor de escala y las vecindades. En el epígrafe en cuestión se accede a ellas mediante unas barras interactivas al usuario para aumentar la comodidad y el control sobre el valor de las variables en cuestión, pero dado que en este caso se desea testear el clasificador sobre un conjunto de prueba de manera rápida y provechando esta opción que brinda la app, se ha decidido utilizar esta por encima de un algoritmo.

En el caso de Matlab, no existe una app con características similares a la explicada, por lo que se ha desarrollado un algoritmo capaz de aplicar de forma autónoma el clasificador cargado sobre el conjunto de prueba completo y guardar los resultados en una carpeta nueva.

El algoritmo en cuestión se apoya en el explicado en el epígrafe 3.2.2, donde se creaba el archivo .xml que contenía al clasificador. En esta ocasión se cargará el clasificador creado mediante ese código y se probará su rendimiento sobre el conjunto de imágenes previamente almacenado con conforma el conjunto de prueba. El algoritmo completo se puede encontrar en el epígrafe Anexos k).

Para comenzar se almacenan en sendas variables tanto la ubicación de las imágenes a analizar como los nombres que se desean entregar a las resultantes. Además, se actualizará la ubicación anteriormente comentada en función del elemento a detectar.

```
%Se cargan los paths de las imágenes a testear y a almacenar.
nameimg='C:\Users\Usuario\Desktop\TFM\CustomHaarRobot\TESTEAR\01.jpg';
Img='test01.png';

%Adaptación del path en función del elemento a detectar
nameimg(40:44)=name;
```

Inicio del bucle infinito, en el que se actualizará el nombre de la imagen a analizar. Al ser un elemento numérico y ascendente, simplemente se le asigna el valor del contador del bucle. Este proceso se repetirá al final del bucle para preparar la imagen resultante para su almacenaje.

Posteriormente se ejecutará el detector sobre cada una de las imágenes, obteniendo como resultado la misma imagen con una detección en forma de rectángulo etiquetado en función del elemento superpuesta.

Al final del bucle simplemente se almacenará la imagen en la ubicación seleccionada como carpeta actual de trabajo en Matlab.

```
%Bucle infinito de testeo
for i=1:30

    %Actualización de la imagen a analizar
    if i<10
        nameimg(55)=num2str(i);
    else
        nameimg(54:55)=num2str(i);
    end
    img=imread(nameimg);

    %Ejecución del detector sobre las imágenes
    bbox=step(detector,img);
    imgDetectada=insertObjectAnnotation(img,'rectangle',bbox,name);

    %Actualización del nombre de la imagen resultante
    I=figure; imshow(imgDetectada);
    if i<10
        Img(6)=num2str(i);
    else
        Img(5:6)=num2str(i);
```

```
    end

    %Almacenamiento de la imagen resultante
    saveas(I,Img)
end
```

El procedimiento a seguir para ambos casos será simple:

- Se aplicará el clasificador sobre el conjunto de prueba de cada elemento a detectar, ya sea con la app o con el algoritmo, dependiendo del caso. Se buscará la mejor combinación de variables posible, esto es, el punto en el que el número de reales positivos sea mayor mientras que el de falsos positivos sea menor.
- Se variará tanto el factor de escala como el número de vecindades, comprobando la influencia de cada uno de los parámetros en la detección final.
- Se compararán los resultados obtenidos mediante el clasificador desarrollado en la app Cascade Trainer GUI y empleado en Python (Raspberry) con los que se obtendrán empleando el clasificador desarrollado en Matlab en ese mismo entorno mediante un algoritmo desarrollado específicamente para ese fin. Para realizar una comparación precisa entre los diferentes clasificadores y combinación de parámetros se calculará el valor de la **precisión** de cada uno de ellos como se describió en el epígrafe 1.5.

Dado que el procedimiento general de ambos casos ha quedado explicado y bien definido, a continuación se pasarán a mostrar los resultados obtenidos en los diferentes estudios.

5.1. Detección de pinzas

Primero se estudiará el caso de la **pinza**, como elemento común y de uso diario. Este no será objeto de estudio como tal, puesto que el caso importante que se viene definiendo desde el inicio del algoritmo es el del Segway, pero se deseaba tener un elemento de fácil “acceso” y más común que el vehículo no tripulado, que es único y característico.

En primer lugar, se mostrarán los resultados obtenidos con Matlab.

5.1.1 Detección de pinza mediante Matlab

Como ya se ha explicado, se empleará el algoritmo comentado en el epígrafe anterior, variando el la tasa de falsos positivos y el número de fases de entrenamiento buscada, con el objetivo de obtener la mejor detección posible. Se computarán los datos de todas las imágenes estudiadas, comprobando el número de detecciones correctas que se han realizado y se acumularán en una tabla que recoja toda la información que lleve a la mejor configuración del clasificador.

El proceso a seguir será analizar cada una de las imágenes comparando el resultado de la clasificación obtenido con la clasificación perfecta, de modo que las detecciones que se superpongan al menos un cincuenta porciento con la detección “perfecta” se clasificará como real positivo y aquellas que no superen ese umbral se considerarán falsos positivos.

Tabla 3. Comparativa de detecciones del clasificador de pinzas en Matlab.

FAR-NCS	Real Positivo	Falso Positivo	Falso negativo	Precisión
0,75-25	56	93	29	37.58%
0,5-15	66	63	19	51.16%
0,5-20	67	154	18	30.32%
0,25-15	63	145	22	31.29%

En la Tabla 3 se puede ver claramente el número de reales positivos, falsos positivos y falsos negativos entregados por el clasificador al analizar el conjunto de prueba seleccionado para las pinzas. En este caso, el número total de detecciones es de 85.

Se aprecia cómo existe una variabilidad muy alta en las detecciones, pero destaca el hecho de que cuanto mayor es el parámetro *FalseAlarmRate* (FAR), mayor es el número de falsos negativos, es decir, el número de elementos correctos en la imagen que quedan sin detectar. También es cierto que a mayor valor de *NumCascadeStages* (NCS) menor número de detecciones incorrectas, es por eso que se debe buscar un compromiso entre ambas partes. En esta ocasión, se destaca como el mejor clasificador el resultante de emplear un **FAR de 0.5** y un **NCS de 15**, puesto que, además de ser el que mayor precisión entrega, al aumentar el número de fases de entrenamiento por encima de cierto umbral, aumentan los reales positivos "pagando" un valor de falsos positivos muy elevado, lo cual no es bueno. Se muestra como ejemplo la Figura 45 en la que se aprecia cómo desaparecen falsos positivos y aumenta el número de reales positivos.

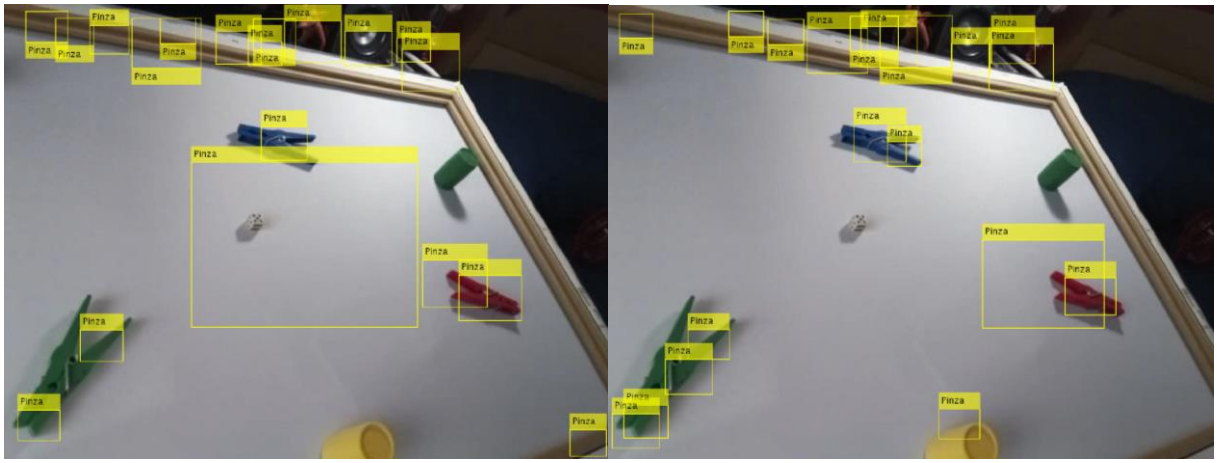


Figura 45. Corrección de las detecciones aumentando el número de fases de entrenamiento.

5.1.2 Detección de pinza mediante la app Cascade Trainer GUI

En este segundo caso se repetirá el proceso anterior con las detecciones realizadas por la app Cascade Trainer GUI. La única modificación que se va a realizar en referente a los parámetros a editar, pues ahora se variará el factor de escala de las detecciones y el número de vecindades que se debe tener la detección para considerarse positiva.

Tabla 4. Comparativa de detecciones del clasificador de pinzas mediante Cascade Trainer GUI.

ESC-VEC	Real Positivo	Falso Positivo	Falso negativo	Precisión
1,05-2	28	11	57	71.79%
1,3-3	4	-	81	-
1,5-3	5	-	80	-

Se muestran en la Tabla 4 los resultados obtenidos de este estudio. En esta ocasión llama la atención la caída del número de falsos positivos. Esto se da cuando el clasificador busca un nivel de confianza muy elevado en sus detecciones y, por tanto, no clasifica un elemento hasta que no presenta unas características muy similares a las entregadas como positivas durante el entrenamiento.

Esto es positivo, pues se garantiza que las detecciones que se realicen, en gran medida, van a ser el elemento que se busca, pero, como punto negativo, provoca que se pierda un número elevado de elementos que sí deberían haber sido detectados.

La influencia de los parámetros es clara, pues, a mayor valor de las vecindades, menor es el número de detecciones, en general, puesto que no se dan tantas detecciones próximas como para obtener una detección muy robusta que represente la información común a esas clasificaciones vecinas. Por otro lado, cuanto mayor es el valor de la escala, de mayor tamaño deben ser las clasificaciones para ser tenidas en cuenta.

Teniendo en cuenta los datos mostrados anteriormente es fácil identificar la mejor configuración para el clasificador: **factor de escala de 1,05 y 2 vecindades**.

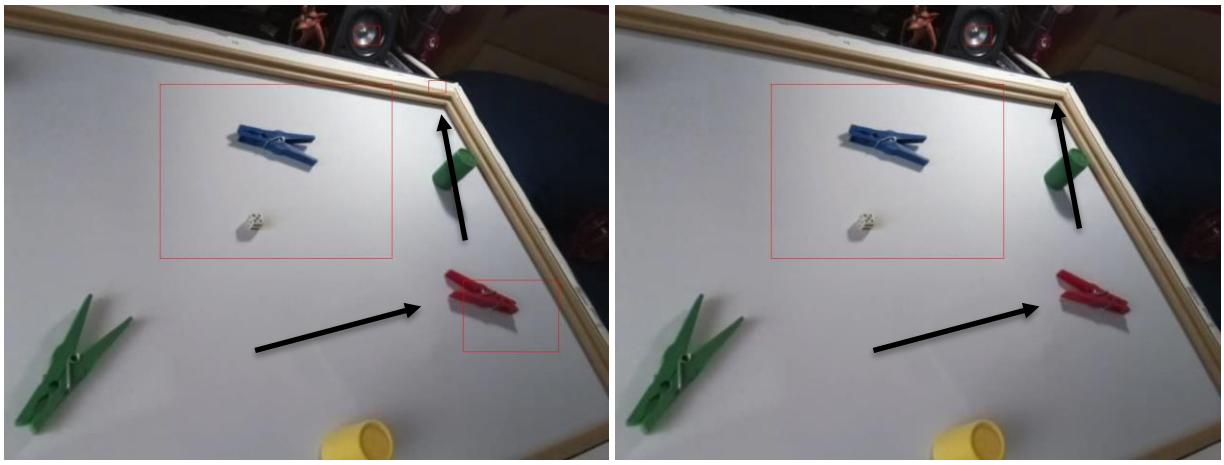


Figura 46. Variación en las detecciones mediante el aumento del número de vecindades.

5.2. Detección de caras

En este epígrafe, para tener aún más datos en las comparativas, se ha estudiado el resultado aportado por diferentes clasificadores de caras. En este caso en lugar de contar con dos clasificadores se contará con tres: uno creado en Matlab, uno creado con la app Cascade Trainer

GUI y un último que es el que proporcionan por defecto las OpenCV al instalarlas en la Raspberry.

Igual que en el caso anterior, se cuenta con un banco de imágenes positivas con elementos a identificar, ya sea de forma única en la imagen, o múltiple. Se seguirá el mismo procedimiento explicado al inicio de este mismo capítulo.

5.2.1 Detección de caras mediante Matlab

Al igual que en el epígrafe 5.1.1 se aplicará el algoritmo desarrollado y se compararán las detecciones con la detección perfecta, obteniendo así las tasas de reales positivos, falsos positivos y falsos negativos.

Tabla 5. Comparativa de detecciones del clasificador de caras en Matlab.

FAR-NCS	Real Positivo	Falso Positivo	Falso negativo	Precisión
0,5-20	26	62	16	29.54%
0,4-17	24	14	18	63.16%
0,3-20	25	8	17	75.76%
0,25-20	15	3	26	83.33%

En esta ocasión, el número total de elementos a identificar es 42. Observando la Tabla 5 llama la atención la gran variación en la tasa de falsos positivos. La explicación para esto es sencilla, a menor nivel de *FalseAlarmRate*, mayor es la cantidad de detecciones que se muestran, por eso el valor de *NumCascadeStages* es tan elevado, porque de no serlo, el número de falsos positivos se dispararía por encima de niveles aceptables.

Para esta situación se alzaría como el mejor clasificador el obtenido al emplear una **tasa de falsos positivos de 0.3 y 20 fases de entrenamiento** puesto que, además de ser el que el segundo que mayor precisión aporta, es uno de los que menor tasa de falsos positivos muestra. Esto indica que las detecciones que realiza son en una muy amplia parte de los casos correctas, sacrificando otras tantas que no realiza, lo cual no llega a ser bueno, pero en función de la situación en la que se vaya a aplicar el clasificador puede merecer la pena. Se descarta la opción de la máxima precisión puesto que descarta un número muy elevado de elementos que debería detectar.

Se muestra como ejemplo ilustrativo la Figura 47, en la que se puede apreciar cómo el número de detecciones se ve afectado por los parámetros, al pasar de 0.25 (FAR) a 0.3 (FAR). El número de detecciones se reduce puesto que se acepta un menor número de detecciones por cada fase del entrenamiento, haciendo que se descarten muchas de ellas, sean posibles detecciones positivas o no.

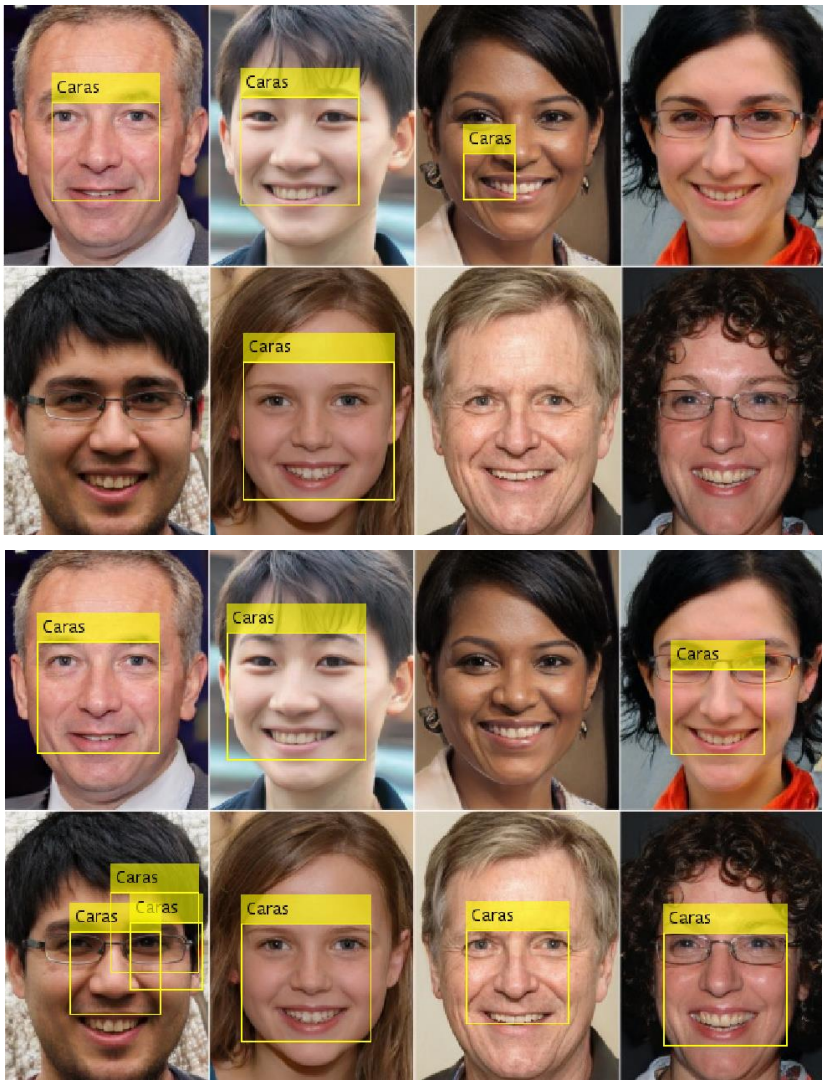


Figura 47. Corrección de las detecciones mediante la variación de la tasa de falsos positivos.

5.2.2 Detección de caras mediante la app Cascade Trainer GUI

Realizando el estudio de las detecciones entregadas por el clasificador se obtienen los siguientes resultados.

Tabla 6. Comparativa de detecciones del clasificador de caras mediante Cascade Trainer GUI.

ESC-VEC	Real Positivo	Falso Positivo	Falso negativo	Precisión
1,1-15	32	75	10	29,91%
1,1-20	30	54	12	35,71%
1,3-5	33	50	8	39,76%
1,5-5	21	41	20	33,87%

Ante los resultados obtenidos, lo primero que llama la atención es la escasa precisión que aporta el

clasificador. A pesar de ello también destaca como dato muy positivo la bajada de los falsos negativos, que en el caso de Matlab llegaban a superar a los reales positivos, lo cual no hablaba muy bien del clasificador.

En este caso, el clasificador que aporta unos mejores resultados es el obtenido al emplear un factor de escala igual a 1,3 y 5 vecindades, porque, aunque por poco, es el que mayor precisión presenta y además, menor número de falsos positivos.

5.2.3 Detección de caras empleando el clasificador por defecto de las OpenCV

En esta ocasión se va a comprobar el funcionamiento del clasificador que proporciona por defecto la librería OpenCV para Python.

Las pruebas se realizarán, al igual que antes, con el banco de imágenes de prueba positivas, pero en esta ocasión se trabajará solamente con los parámetros por defecto, es decir, un factor de escala de 1,1 y 3 vecindades. Se ha optado por hacerlo así para limitar la influencia de los parámetros en las detecciones de modo que se pruebe el rendimiento del clasificador en sí dado que el propio clasificador ya está perfectamente sintonizado para la detección e identificación de rasgos faciales.

Tabla 7. Estudio de detecciones del clasificador de caras por defecto de OpenCV mediante la app.

ESC-VEC	Real Positivo	Falso Positivo	Falso negativo	Precisión
1,1-3	41	8	1	83,67%

Tal y como se muestra en la Tabla 7, el clasificador aporta unos datos muy cercanos a la perfección. Esto se da porque el clasificador emplea un banco de imágenes y un número de árboles de decisión de entrenamiento muchísimo mayor, recogiendo información de rasgos como barba, gafas, y expresiones faciales, que son las características que más han afectado en el desarrollo de los clasificadores creados en el desarrollo del presente proyecto.

En vista de los resultados se puede afirmar que todos aquellos elementos que este clasificador identifique como cara, lo serán, puesto que presenta más de un 83 % de precisión, y solo ha aportado 8 pequeños falsos positivos en la prueba, sin modificar los parámetros de trabajo.

Para llegar a alcanzar, o al menos aproximarse a estos resultados se debería aumentar notablemente el banco de imágenes y conseguir que el conjunto positivo de entrenamiento recoja mayor cantidad de información y muchas más situaciones que las que se han llegado a aportar en este trabajo.

5.3. Detección del Segway

Como punto final en el estudio de los resultados de este trabajo, se comprobarán los resultados aportados por los clasificadores para el robot desarrollados tanto en Matlab como en Python. En esta ocasión, el objetivo es trabajar sobre un archivo de vídeo, ya sea almacenado o en tiempo real. No obstante, al analizar un vídeo con los algoritmos explicados en el capítulo 4, no se analiza el vídeo como archivo, sino que se lee al completo y se va extrayendo frame a frame para analizarlo y ejecutar el detector sobre ellos, del mismo modo que se está haciendo ahora con el banco de

imágenes positivo.

5.3.1 Detección del Segway mediante Matlab

En primer lugar, se analizarán los resultados entregados por Matlab, ya que se ha buscado el mejor resultado mediante la variación de los parámetros de entrenamiento. Esto se hace porque a la hora de ejecutar el detector solo se tienen en cuenta los parámetros de entrenamiento, no permite su modificación durante la ejecución como en el caso de Python.

A continuación, se muestra la Tabla 8 en la que se estudian exhaustivamente los resultados obtenidos.

Tabla 8. Comparativa de detecciones del clasificador del Segway en Matlab.

FAR-NCS	Real Positivo	Falso Positivo	Falso negativo	Precisión
0,75-15	29	250	1	10,39%
0,5-15	19	29	9	39,58%
0,2-15	18	27	12	40,00%
0,15-15	12	9	18	57,14%

Observando la tabla anterior es fácilmente asimilable la influencia del parámetro *FalseAlarmRate*, pues a medida que éste disminuye, el número de detecciones cae con él. Al comienzo, con un valor del parámetro 0,75, se contaba con 280 detecciones positivas, ya fuesen reales o falsas y al final, con un valor de 0,15, se obtiene un total de 21 detecciones.

Esto guía, en vista del número de falsos negativos de cada una de las comprobaciones, a asegurar que la mejor clasificación se obtendría con una combinación de *FalseAlarmRate*=0.5 y *NumCascadeStages*=15, pues a pesar de que no se obtiene la máxima precisión, la diferencia con respecto a ese valor tope se ve compensada con una menor tasa de falsos negativos, los cuales podrían provocar, en un momento crítico, que fallase el sistema al no ser capaz de localizar al Segway.

Además, este estudio sirve para identificar, tanto la zona más luminosa, como la sombra que aparece en las imágenes, como las regiones más conflictivas para la detección. Se reconoce puesto que cada vez que el Segway se acerca a una de ellas, las detecciones no son todo lo correctas que podrían ser. Además, se muestra la Figura 48 como apoyo a esta cuestión. Se hace notable hasta el punto de que la mayor parte de los falsos positivos, coinciden con la región más iluminada. Este hecho se podría corregir aumentando el número de imágenes del conjunto de entrenamiento negativo que contengan esas zonas a la hora de realizar el entrenamiento.

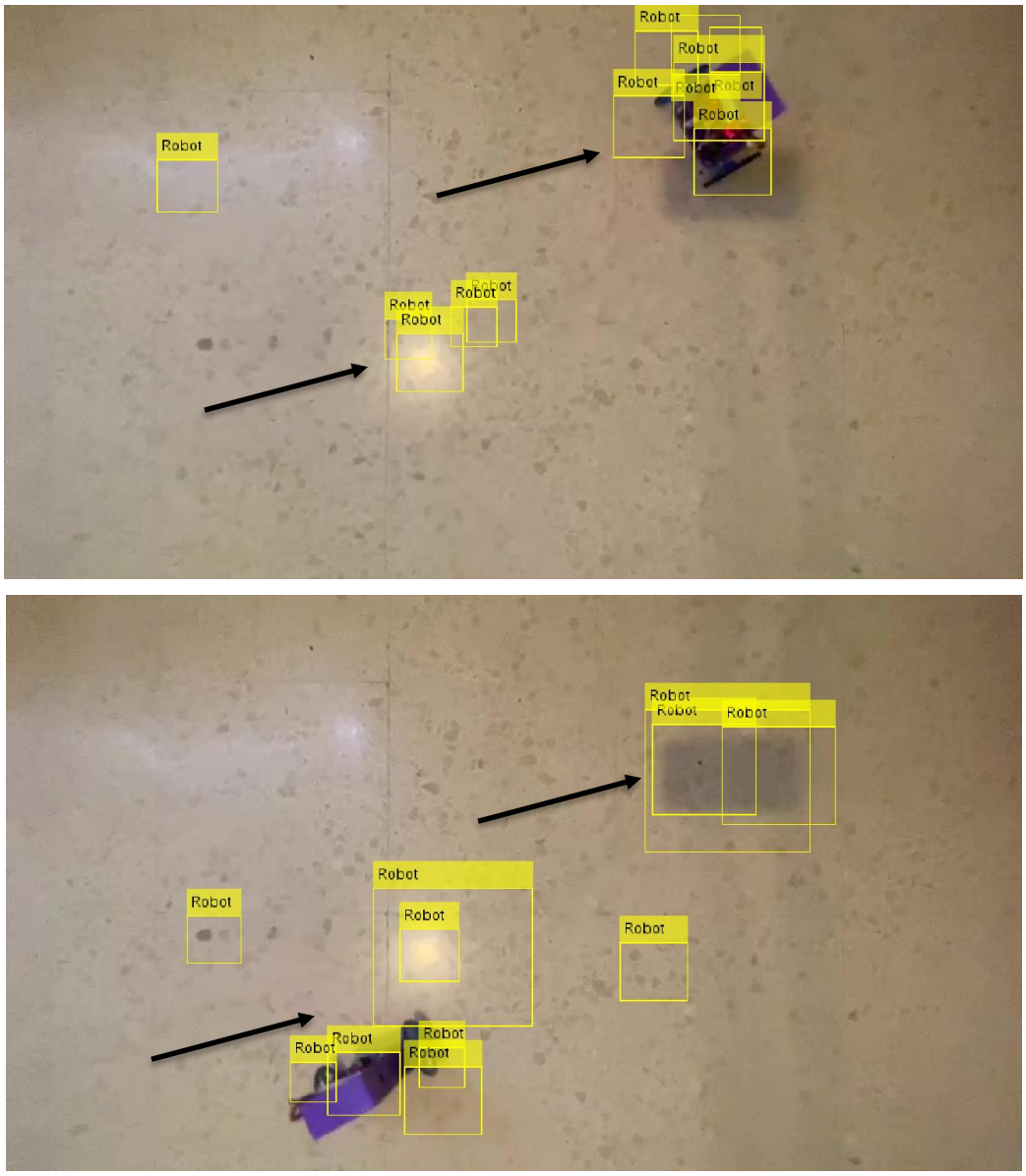


Figura 48. Influencia de las zonas conflictivas en la identificación.

5.3.2 Detección del Segway mediante la app Cascade Trainer GUI

Este es uno de los puntos más importantes del documento, pues es el que aporta información sobre las detecciones que realmente se darán en la identificación. En todos los algoritmos definidos en el capítulo anterior se empleó la configuración por defecto del clasificador, al igual que se ha hecho con el clasificador de caras contenido en las OpenCV, es por eso que en este caso tampoco se modificarán los parámetros de testeo, puesto que no corresponderían con los que se obtendrían al aplicar los algoritmos ya comentados.

A continuación, se muestran los resultados del testeo del clasificador sobre el conjunto de imágenes que se ha trabajado en el epígrafe anterior.

Tabla 9. Estudio de detecciones del clasificador del Segway mediante la app.

ESC-VEC	Real Positivo	Falso Positivo	Falso negativo	Precisión
1,1-3	29	-	1	100,0%

Estos datos reflejan la calidad del clasificador creado. Es fácil comprender que este clasificador sería ideal para la aplicación para la que se quiere destinar, puesto que en todos los casos identifica la posición del Segway de manera correcta. De este modo se podrían obtener la posición y la orientación en tiempo real del vehículo, con lo que se podrían entregar estos datos al usuario para que pudiese trabajar con ellos. Otra opción es entregar al propio Segway esta información, con el fin de poder actuar en consecuencia, llegando hasta el punto de poder dibujar cualquier posible trayectoria.

Ante la imposibilidad de trabajar en este documento con un archivo de vídeo se mostrarán todos los resultados en la presentación del propio trabajo de fin de estudios, a fin de mostrar fielmente los resultados obtenidos.

Este clasificador, no obstante, sigue manteniendo la influencia de las regiones comentadas al final del epígrafe anterior, pero en mucha menor medida. A continuación, se puede apreciar lo explicado, viendo la Figura 49.

Figura 49. Influencia de las zonas oscuras y luminosas en la detección.

6 CONCLUSIONES

En este capítulo final se analizará la consecución de los objetivos marcados al inicio del presente documento y se tratarán de aportar ciertas propuestas de mejora o de continuación del trabajo aquí desarrollado.

De este modo, y rescatando los objetivos descritos en el capítulo 1.3 de este proyecto, se puede afirmar que todos y cada uno de ellos han sido logrados:

El primero de ha sido la aplicación de procedimientos HOG, para la detección de elementos en imágenes mediante algoritmos desarrollados en Matlab, abarcando desde la creación de los detectores hasta la puesta en funcionamiento sobre un archivo de vídeo, y Haar, empleado para el trabajo en Python y el entrenamiento de algoritmos mediante la aplicación Cascade Trainer GUI, que ha sido de gran ayuda. Este punto incluye, además, el uso de máquinas de vectores de soporte o SVM que entraban en juego en el proceso de detección.

Apoyados en los procedimientos HOG y Haar comentados, se han desarrollado múltiples algoritmos de detección, gestión de archivos y tratamiento de imágenes que no podrían haber sido creados sin OpenCV. Durante el presente proyecto se han empleado una cantidad muy elevada de métodos que requerían de ellas, en concreto, para ser empleados en Python sobre la Raspberry. La posibilidad de crear estos algoritmos en una placa con tanta potencia en relación con el tamaño y precio que tiene hace que sea muy fácil crear aplicaciones como ésta y aplicarlas al ámbito que en cada caso se desee, puesto que para ponerla en funcionamiento no se requiere más que de una fuente de alimentación.

Finalmente, en el último capítulo de este documento, se compararon los resultados obtenidos al aplicar los algoritmos tanto en Matlab como en Python, buscando los mejores resultados en la detección de tres elementos bien diferenciados.

6.1. Propuestas de mejora y continuación del trabajo realizado

Como punto final del proyecto se desean proponer una serie de posibles mejoras, así como de continuación del trabajo, puesto que lo que se ha conseguido aquí puede ser un punto de partida muy interesante si se desea seguir explotando.

La primera mejora que se podría realizar sería la actualización de la placa con la que se ha trabajado al modelo más nuevo, puesto que facilitaría el trabajo abriendo camino a la realización de un mayor número de operaciones simultáneas, además de una mayor rapidez en la ejecución de los algoritmos desarrollados.

Otra mejora podría ser la conexión de esta Raspberry con otras, para así aumentar el campo de trabajo, posibilitando a su vez la puesta en marcha de un número más elevado de Segways. Esto abriría la puerta a un amplio campo de posibilidades, pues se podrían crear trazados por la habitación, planta o incluso edificio en el que se desee trabajar.

Como continuación a este trabajo, contando con los datos de posición y orientación del Segway, se podría, somo se ha comentado varias veces en este documento, coordinar junto con otros vehículos para la realización de dibujos, movimientos coordinados, o incluso tareas.

Estas tareas podrían ir desde la identificación de elementos en el suelo, como materiales o herramientas que se hayan caído de algún puesto de trabajo hasta, si se incorpora de algún modo

un módulo de manipulación al vehículo, la recogida de estos elementos, o el transporte de un punto a otro.

También, aplicado al campo de la asistencia, se podría emplear este trabajo para el guiado a personas de avanzada edad con problemas, por ejemplo, de visión, si se incluye algún tipo de elemento sonoro al vehículo. Asimismo, podría trabajar como guía en un edificio, planta o sala.

Aquí concluye el capítulo Conclusiones.

a) Creación del dataset con la PiCamera (customhaar.py):

```
# Importación de librerías
import cv2
import os
import time


########################################################
# Inicializaciones

myPath = '/home/pi/Desktop/data/images'
cameraNo = 0      # Selección de la cámara
cameraBrightness = 180 # Selección del brillo
moduleVal = 10    # Guarda un frame daca 'moduleVal' para evitar repeticiones
minBlur = 100     # Valor que evita guardar imágenes borrosas
grayImage = False # Flag para el guardado de imágenes en escalade grises
saveData = True   # Flag para el guardado de imágenes
showImage = True  # Flag para mostrar la preview
imgWidth = 180    # Definición del ancho y el alto de la imagen
imgHeight = 120
count = 0
countSave = 0

########################################################

# Inicio de la cámara y setup
global countFolder
cap = cv2.VideoCapture(cameraNo)
cap.set(3, 640)
cap.set(4, 480)
cap.set(35,180) # Giro 180º la imagen de la cámara

# Creación de una función que irá nombrando cada carpeta por orden
def saveDataFunc():
    global countFolder
    countFolder = 0
    while os.path.exists(myPath+ str(countFolder)):
        countFolder += 1
    os.makedirs(myPath + str(countFolder))


if saveData:saveDataFunc()


while True:
```

```
    success, img = cap.read()   # Se lee la imagen de la cámara
    img = cv2.resize(img,(imgWidth,imgHeight)) # Se reescala
    if grayImage:img = cv2.cvtColor(img,cv2.COLOR_BGR2GRAY) # Guarda escala gris
    # Guarda la imagen si no tiene excesivo ruido, indicando la hora y el ruido
    if saveData:
        blur = cv2.Laplacian(img, cv2.CV_64F).var()
        if count % moduleVal ==0 and blur > minBlur:
            nowTime = time.time()
            cv2.imwrite(myPath + str(countFolder) +
                    '/' + str(countSave)+"_"+ str(int(blur))+"_"+str(nowTime)+".
png", img)
            print("Photo") # Indica cuando se ha capturado una imagen
            countSave+=1
        count += 1

    # Muestra un vídeo para guiar al usuario mientras captura las imágenes
    if showImage:
        cv2.imshow("Image", img)

    # Al pulsar la tecla Q cierra la ventana y detiene la ejecución
    if cv2.waitKey(1) & 0xFF == ord('q'):
        break

cap.release()
cv2.destroyAllWindows()
```

b) Creación del dataset desde una url (DownloadDataset.py):

```
# Importación de librerías
import urllib.request
import cv2
import numpy as np
import os


# Lectura del link que contiene los urls de las imágenes
neg_images_link = 'http://www.image-
net.org/api/text/imagenet.synset.geturls?wnid=n09618957'
# Lectura de las urls de las imágenes
neg_image_urls = urllib.request.urlopen(neg_images_link).read().decode()
pic_num = 1

# Si no existe una carpeta llamada ‘neg’ la crea para las imágenes descargadas
if not os.path.exists('neg'):
    os.makedirs('neg')

# Bucle para descargar las imágenes, numerarlas y almacenarlas en la carpeta
for i in neg_image_urls.split('\n'):
    try:
        print(i)
        urllib.request.urlretrieve(i, "neg/"+str(pic_num)+".jpg")
        img = cv2.imread("neg/"+str(pic_num)+".jpg", cv2.IMREAD_GRAYSCALE)
        cv2.imwrite("neg/"+str(pic_num)+".jpg", img)
        pic_num += 1

# Si la url no contiene una imagen, lo indica y pasa a la siguiente
    except Exception as e:
        print(str(e))
```

c) Creación y entrenamiento del detector (detector.m):

```
%==========================================
%===========CREACIÓN DEL DETECTOR===========
%==========================================

% Inicializaciones
tabla='RobotROItabla.mat';
haar='CustomHaarRobot';
detectorname='RobotDetector_';


% Creación de las variables

% Se solicita la introducción del objeto que se desea clasificar y renombra
las variables inicializadas anteriormente
prompt='Selecciona un objeto ([R]obot, [P]inza, [C]aras) [R]:\n';
name=input(prompt,'s');
if (~isempty(name) && (name=='P'))==1
    name='Pinza';
elseif (~isempty(name) && (name=='R'))==1
    name='Robot';
elseif (~isempty(name) && (name=='C'))==1
    name='Caras';
else
    disp('ERROR: Nombre no encontrado.');
end
tabla(1:5)  = name;
haar(11:15) = name;
detectorname(1:5) = name;


% Tabla de detecciones

% Se carga la tabla que contiene las ROIs del elemento seleccionado
load(tabla);
if name=='Robot'
    deteccionesPositivas = RobotROItabla(:,1:2);
elseif name=='Pinza'
    deteccionesPositivas = PinzaROItabla(:,1:2);
elseif name=='Caras'
    deteccionesPositivas = CarasROItabla(:,1:2);
end


% Paths

% Ubicación de los CE, positivos y negativos
carpetaPositivas =
fullfile('C:\','Users','Usuario','Desktop','TFM',haar,'p');
addpath(carpetaPositivas);
carpetaNegativas =
fullfile('C:\','Users','Usuario','Desktop','TFM',haar,'n');
deteccionesNegativas = imageDatastore(carpetaNegativas);


%======Entrenamiento del detector======

% Elección del valor de los parámetros de entrenamiento
```

```
% FalseAlarmRate:   Tasa de falsos positivos deseada.
% NumCascadeStages: Número de fases del entrenamiento.
prompt='Selecciona un valor para FalseAlarmRate (Tasa de falsos positivos
deseada) (0-1) [0.5]:\n';
far=input(prompt);
prompt='Seleccioan un valor para NumCascadeStages (Número de fases del
entrenamiento) (1-20) [15]:\n';
ncs=input(prompt);
if isempty(far)
    far=0.5;
end
if isempty(ncs)
    ncs=15;
end


% Nombre del detector en función del valor de las variables
if (length(num2str(far,3))<=3)
    aux=strcat(num2str(far,3),'0_',num2str(ncs));
else
aux=strcat(num2str(far,3),'_',num2str(ncs));
end
DetectorName=strcat(detectorname,num2str(aux),'.xml');


% Inicio del entrenamiento en cascada
trainCascadeObjectDetector(DetectorName,deteccionesPositivas,carpetaNegativas
,'FalseAlarmRate',far,'NumCascadeStages',ncs);
detector=vision.CascadeObjectDetector(DetectorName);
```

d) Detección del objeto sobre un vídeo (detecciónObj.m):

```
%====================================
%========DETECCIÓN DE OBJETOS=========
%====================================

% Vídeo en el que aparece el objeto a detectar
vid = VideoReader('videoSegway.mp4');

% En caso de querer comprobar el detector creado en la otra aplicación,
descomentar la siguiente línea
%detector = vision.CascadeObjectDetector('cascadeR.xml');

% Bucle infinito para la detección
while hasFrame(vid)
    vf = readFrame(vid);
    bbox = step(detector,vf);
    detectedImg =
insertObjectAnnotation(vf,'rectangle',bbox,'Segway','LineWidth',2);
    imshow(detectedImg);
    pause(1/vid.FrameRate);
end
```

e) Detección de objetos en una imagen (PhotoDetector.py):

```
# Importación de librerías
import cv2
import numpy as np

# Carga del clasificador a utilizar
faceCascade = cv2.CascadeClassifier("/home/pi/Desktop/TFM/CustomHaarRobot/classi
fier/cascadeR.xml")
# Carga de la imagen a detectar
img = cv2.imread("/home/pi/Desktop/TFM/scene3Segways.png")
imgGray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY) # Paso a escala de grises

# Se realiza la detección con el clasificador
Segway = faceCascade.detectMultiScale(imgGray, 1.2, 4)

# Bucle para crear todas las bounding boxes y localizar su centro
for (x, y, w, h) in Segway:
    a = x+w//2
    b = y+h//2
    cv2.rectangle(img, (x, y), (x+w, y+h), (0,255,255), 2)
    cv2.putText(img, 'Segway', (x, y-5), cv2.FONT_HERSHEY_COMPLEX_SMALL, 1,
(0,255,255), 2)
    roi_color = img[y:y+h, x:x+w]
    cv2.circle(img, (a, b), 1, (255,255,255), 2)

cv2.imwrite("Detección.png", img) # Se guarda la imagen con las bounding boxes
cv2.imshow("Segway", img) # Se muestra la imagen
cv2.waitKey(1)
```

f) Detección de objetos en streaming con la PiCamera (ObjDetection.py):

```
# Importación de librerías
import cv2

#####################################################################
# Inicializaciones

# Ubicación del clasificador
path = '/home/pi/Desktop/TFM/CustomHaarRobot/classifier/cascade0.xml'
cameraNo = 0              # ID Cámara
objectName = 'Segway'     # Nombre a mostrar
frameWidth= 640           # Ancho preview
frameHeight = 480         # Alto Preview
color = (255,0,255)       # Color de la detección
#####################################################################

# Setup de la cámara
cap = cv2.VideoCapture(cameraNo)
cap.set(3, frameWidth)
cap.set(4, frameHeight)
cap.set(35, 180)                    # Giro de 180º de la imagen

def empty(a):
    pass

# Definición de trackbars para ajustar la imagen y el detector a tiempo real
cv2.namedWindow("Result")
cv2.resizeWindow("Result", frameWidth, frameHeight+100)
cv2.createTrackbar("Scale", "Result", 300, 1000, empty)
cv2.createTrackbar("Neig", "Result", 6, 50, empty)
cv2.createTrackbar("Min Area", "Result", 50, 100000, empty)
cv2.createTrackbar("Brightness", "Result", 60, 255, empty)

# Se carga el clasificador
cascade = cv2.CascadeClassifier(path)

while True:
    # Modificación del brillo de la cámara
    cameraBrightness = cv2.getTrackbarPos("Brightness", "Result")
    cap.set(10, cameraBrightness)
    # Conversión a escala de grises
    success, img = cap.read()
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    # Selección de escala y número de vecindades necesarios para la detección
    scaleVal = 1 + (cv2.getTrackbarPos("Scale", "Result") /1000)
    neig = cv2.getTrackbarPos("Neig", "Result")
    # Detección del objeto mediante el clasificador
    objects = cascade.detectMultiScale(gray, scaleVal, neig)
```

```
    # Localización del objeto y muestra de las bounding boxes con la detección
    for (x, y, w, h) in objects:
        area = w*h
        minArea = cv2.getTrackbarPos("Min Area", "Result")
        if area > minArea:
            cv2.rectangle(img, (x, y), (x+w, y+h), color, 3)
            cv2.putText(img, objectName, (x, y-5),
cv2.FONT_HERSHEY_COMPLEX_SMALL, 1, color, 2)
            roi_color = img[y:y+h, x:x+w]

    # Muestra la vista de la PiCamera para la detección
    cv2.imshow("Result", img)

    # Al pulsar la tecla Q cierra la ventana y detiene la ejecución
    if cv2.waitKey(1) & 0xFF == ord('q'):
         break
```

g) Detección de objetos en un .mp4 (DetecSegway.py):

```
# Importación de librerías
import cv2
import numpy as np

######################################################################
# Inicializaciones

# Ubicación del clasificador Segway
path = '/home/pi/Desktop/TFM/CustomHaarRobot/classifier/cascadeRRR.xml'
cameraNo = 0                # ID Cámara
objectName = 'Segway'       # Nombre a mostrar
frameWidth= 640             # Ancho preview
frameHeight = 480           # Alto Preview
color = (0,255,255)         # Color de la detección
######################################################################

# Lectura del vídeo
cap = cv2.VideoCapture("/home/pi/Desktop/TFM/CustomHaarRobot/videoSegway0.mp4")

def empty(a):
    pass

# Definición de la ventana en la que se mostrará el vídeo
cv2.namedWindow("Result")
cv2.resizeWindow("Result", frameWidth, frameHeight+100)

# Se carga el clasificador
cascade = cv2.CascadeClassifier(path)

while True:

    # Lectura de cada frame del vídeo
    success, img = cap.read()

    # Tratamiento de los frames para su correcta lectura y clasificación
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

    # Detección del objeto mediante el clasificador
    objects = cascade.detectMultiScale(gray)

    # Localización del objeto
    for (x, y, w, h) in objects:
        area = w*h
        a = x+w//2
        b = y+h//2
        minArea = 1000
        if area > minArea:
```

```
            cv2.rectangle(img, (x, y), (x+w, y+h), color, 3)
            cv2.putText(img, objectName, (x, y-5),
cv2.FONT_HERSHEY_COMPLEX_SMALL, 1, color, 2)
            roi_color = img[y:y+h, x:x+w]

            # Centro de la detección
            cv2.circle(img, (a, b), 1, (255,255,255), 2)

    # Muestra del resultado
    cv2.imshow("Result", img)

    # Al pulsar la tecla Q cierra la ventana y detiene la ejecución
    key = cv2.waitKey(1) & 0xFF
    if key == ord('q'):
         break

cv2.destroyAllWindows()
```

h) Localización de umbral de color en un .mp4 y muestra de su posición y ángulo (DetecFiltradaXYt.py):

```
# Importación de librerías
import cv2
import imutils
import argparse
import math
import numpy as np
from collections import deque

#################################################################
# Inicializaciones

cameraNo = 0                # ID Cámara
objectName = 'Segway'       # Nombre a mostrar SEGWAY
frameWidth= 640             # Ancho preview
frameHeight = 480           # Alto Preview

# Definición del buffer que almacenará los puntos en los que ha estado el Segway
ap = argparse.ArgumentParser()
ap.add_argument("-b", "--buffer", type=int, default=32, help="max buffer size")
args = vars(ap.parse_args())
pts = deque(maxlen=args["buffer"])
counter = 0

(dX, dY) = (0, 0)
t = 0
dH = 0

# Niveles de color a buscar en la imagen
purplemin = (50,0,60)
purplemax = (170,170,170)
#################################################################

# Lectura del vídeo
cap = cv2.VideoCapture("/home/pi/Desktop/TFM/CustomHaarRobot/videoSegway0.mp4")

def empty(a):
    pass

# Definición de la ventana en la que se mostrará el vídeo
cv2.namedWindow("Result")
cv2.resizeWindow("Result", frameWidth, frameHeight+100)


while True:

    # Lectura de cada frame del vídeo
```

```
    success, img = cap.read()

    # Tratamiento de los frames para su correcta lectura y localización
    blurred = cv2.GaussianBlur(img, (9,9), 0)
    hsv = cv2.cvtColor(blurred, cv2.COLOR_BGR2HSV)

    # Tratamiento de los píxeles en el rango de color definido para generar un
único contorno detectable
    mask = cv2.inRange(hsv, purplemin, purplemax)
    mask = cv2.erode(mask, None, iterations=2)
    mask = cv2.dilate(mask, None, iterations=20)
    cotr = cv2.findContours(mask.copy(), cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIM
PLE)
    cotr = imutils.grab_contours(cotr)
    center = None

    if len(cotr) > 0:

        # Se busca el contorno mayor
        c = max(cotr, key=cv2.contourArea)
        (x, y, w, h) = cv2.boundingRect(c)
        a = x+w//2
        b = y+h//2
        M = cv2.moments(c)
        center = (int(M["m10"] / M["m00"]), int(M["m01"] / M["m00"]))

        if cv2.contourArea(c) > 100:
            cv2.rectangle(img, (x, y), (x+w, y+h), (0,255,0), 2)
            # Centro del cuadrado (posición instantánea)
            cv2.circle(img, (a, b), 1, (255,255,255), 2)
            pts.appendleft(center)

    # Se van almacenando en el buffer las posiciones anteriores del Segway
    for i in np.arange(1, len(pts)):
        if pts[i - 1] is None or pts[i] is None:
            continue

        if counter >= 10 and i == 1 and pts[-10] is not None:

            dX = pts[-10][0] - pts[i][0]
            dY = pts[-10][1] - pts[i][1]
            dX = float(-dX)
            dY = float(-dY)

            # Se calcula el ángulo que posee el Segway
            if counter >= 11:
                dH = math.sqrt(dX**2 + dY**2)
                t = (((math.acos(dX / float(dH))) * 360) / (2 * math.pi))
                if np.sign(dY) == 1:
                    t = -t
```

```
            # Creación de la estela que ha dejado el Segway
            thickness = int(np.sqrt(args["buffer"] / float(i + 1)) * 2.5)
            cv2.line(img, pts[i - 1], pts[i], (0,0,255), thickness)

        # Se muestra sobre el vídeo tanto la posición como el ángulo instantáneo
        cv2.putText(img, "Posicion X: {}, Y: {}".format(a, b), (10, img.shape[0] -
    20), cv2.FONT_HERSHEY_SIMPLEX, 0.35, (255, 255, 255), 1)
        cv2.putText(img, "Angulo: {:.2f} grados".format(t), (10, img.shape[0] -
    10), cv2.FONT_HERSHEY_SIMPLEX, 0.35, (255, 255, 255), 1)

        # Se muestran tanto la máscara de píxeles de color como la detección
        cv2.imshow("Máscara", mask)
        cv2.imshow("Result", img)

        # Al pulsar la tecla Q cierra la ventana y detiene la ejecución
        key = cv2.waitKey(1) & 0xFF
        counter += 1
        if key == ord('q'):
            break
    cv2.destroyAllWindows()
```

i) Localización de umbral de color en un .mp4 y entrega de su posición instantánea (DetecFiltradaSOLOXY.py):

```
# Importación de librerías
import cv2
import imutils
import argparse
import math
import numpy as np
from collections import deque
import time

######################################################################
# Inicializaciones

cameraNo = 0                # ID Cámara

# Definición del buffer que almacenará los puntos en los que ha estado el Segway
ap = argparse.ArgumentParser()
ap.add_argument("-b", "--buffer", type=int, default=32, help="max buffer size")
args = vars(ap.parse_args())
pts = deque(maxlen=args["buffer"])
counter = 0
(dX, dY) = (0, 0)
t = 0
dH = 0
start_time = time.time()

# Niveles de color a buscar en la imagen
purplemin = (50,0,60)
purplemax = (170,170,170)
######################################################################

# Lectura del vídeo
cap = cv2.VideoCapture("/home/pi/Desktop/TFM/CustomHaarRobot/videoSegway0.mp4")

def empty(a):
    pass

# Definición de la ventana
cv2.namedWindow("Result")

while True:

    # Lectura de cada frame del vídeo
    success, img = cap.read()

    # Tratamiento de los frames para su correcta lectura y localización
    blurred = cv2.GaussianBlur(img, (9,9), 0)
    hsv = cv2.cvtColor(blurred, cv2.COLOR_BGR2HSV)
```

```
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

    # Tratamiento de los píxeles en el rango de color definido para generar un
único contorno detectable
    mask = cv2.inRange(hsv, purplemin, purplemax)
    mask = cv2.erode(mask, None, iterations=2)
    mask = cv2.dilate(mask, None, iterations=20)
    cotr = cv2.findContours(mask.copy(), cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIM
PLE)
    cotr = imutils.grab_contours(cotr)
    center = None

    if len(cotr) > 0:
        # Se busca el contorno mayor
        c = max(cotr, key=cv2.contourArea)
        (x, y, w, h) = cv2.boundingRect(c)
        a = x+w//2
        b = y+h//2
        M = cv2.moments(c)
        center = (int(M["m10"] / M["m00"]), int(M["m01"] / M["m00"]))

        if cv2.contourArea(c) > 100:
            # Posición instantánea del Segway almacenada en el buffer
            pts.appendleft(center)

    # Se van almacenando en el buffer las posiciones anteriores del Segway
    for i in np.arange(1, len(pts)):
        if pts[i - 1] is None or pts[i] is None:
            continue

        if counter >= 10 and i == 1 and pts[-10] is not None:

            dX = pts[-10][0] - pts[i][0]
            dY = pts[-10][1] - pts[i][1]
            dX = float(-dX)
            dY = float(-dY)

            # Se calcula el ángulo que posee el Segway
            if counter >= 11:
                dH = math.sqrt(dX**2 + dY**2)
                t = (((math.acos(dX / float(dH))) * 360) / (2 * math.pi))
                if np.sign(dY) == 1:
                    t = -t

    # Cada diez posiciones almacenadas se muestra la posición por la terminal
    if (counter % 2) == 0:
        print("Posicion:", center, "Angulo: {:.2f}".format(t), "     -{:.3f}
segundos".format(time.time() - start.time))
```

```
    # Al pulsar la tecla Q cierra la ventana y detiene la ejecución
    key = cv2.waitKey(1) & 0xFF
    counter += 1
    if key == ord('q'):
         break

cv2.destroyAllWindows()
```

j) Algoritmo auxiliar para guardar los vídeos mostrados (SaveVideo.py):

```
# Importación de librerías
import numpy as np
import cv2

# Lectura del vídeo
cap = cv2.VideoCapture(0)

# Definición del códec y creación del objeto de grabación del vídeo
fourcc = cv2.VideoWriter_fourcc(*'XVID')
out = cv2.VideoWriter('output.avi', fourcc, 10.0, (640, 480))

# Bucle infinito de grabación
while (cap.isOpened()):
    ret, frame = cap.read()
    if ret==True:
        out.write(frame)

        # Muestra en pantalla la vista de la PiCamera
        cv2.imshow('frame', frame)

        # Al pulsar la tecla Q cierra la ventana y detiene la ejecución
        if cv2.waitKey(1) & 0xFF == ord('q'):
            break
    else:
        break

cap.release()
out.release()
cv2.destroyAllWindows()
```

k) Algoritmo de testeo del clasificador desarrollado en Matlab (testeo.m):

```
%Se cargan los paths de las imágenes a testear y a almacenar.
nameimg='C:\Users\Usuario\Desktop\TFM\CustomHaarRobot\TESTEAR\01.jpg';
Img='test01.png';

%Bucle infinito de testeo
for i=1:30
    %Adaptación del path en función del elemento a detectar
    nameimg(40:44)=name;

    %Actualización de la imagen a analizar
    if i<10
        nameimg(55)=num2str(i);
    else
        nameimg(54:55)=num2str(i);
    end
    img=imread(nameimg);

    %Ejecución del detector sobre las imágenes
    bbox=step(detector,img);
    imgDetectada=insertObjectAnnotation(img,'rectangle',bbox,name);

    %Actualización del nombre de la imagen resultante
    I=figure; imshow(imgDetectada);
    if i<10
        Img(6)=num2str(i);
    else
        Img(5:6)=num2str(i);
    end

    %Almacenamiento de la imagen resultante
    saveas(I,Img)
end
```

REFERENCIAS

[1] Nicolás Cortés Fernández, “Diseño, fabricación, montaje, estudio dinámico, control y teleoperación de un vehículo tipo péndulo invertido sobre dos ruedas”, Trabajo de fin de Grado año 2019.

[2] Raspberry Pi.
https://www.raspberrypi.org/

[3] Spyder – The Scientific Python Development Environment.
https://www.spyder-ide.org/

[4] Matlab – Mathworks.
https://es.mathworks.com/products/matlab.html

[5] Raspberry Pi – Camera Module V2.
https://www.raspberrypi.org/products/camera-module-v2/

[6] OpenCV.
https://opencv.org

[7] Manuel Ruiz Arahal, “Apuntes de la asignatura de Percepción en Automática y Robótica”, curso 2018-2019.

[8] A. de la Escalera Hueso, “Visión por computador: fundamentos y métodos”, Editorial Prentice Hall, 2001.

[9] Universidad Autónoma de Barcelona, “Detección de objetos”, Departamento de ciencias de la computación, 2015.

[10] Adaboost Classifier.
https://medium.com/machine-learning-101/https-medium-com-savanpatel-chapter-6-adaboost-classifier-b945f330af06.

[11] A Guide to AdaBoost: Boosting To Save The Day.
https://blog.paperspace.com/adaboost-optimizer/

[12] Sergio Luis Toral Martín, “Apuntes de la asignatura de Sistemas Digitales Avanzados y Aplicaciones”, curso 2018-2019.

[13] OpenCV Documentation
https://docs.opencv.org/2.4/index.html

[14] ImageNet Database.
http://www.image-net.org/index

[15] Cascade Trainer GUI
https://amin-ahmadi.com/cascade-trainer-gui/

[16] Train Cascade Object Detector – MathWorks.
https://es.mathworks.com/help/vision/ref/traincascadeobjectdetector.html?s_tid=doc_ta

[17] Find the Center of a Blob (Centroid) using OpenCV (C++/Python).
https://www.learnopencv.com/find-center-of-blob-centroid-using-opencv-cpp-python/