Trabajo Fin de Grado
Grado de Ingeniería Electrónica, Robótica y Mecatrónica,
Intensificación en Automática y Robótica

Detección de objetos en ROS para aplicaciones en almacenes

Autor: Felipe Pérez Molina
Tutor: Miguel Ángel Ridao Carlini

Dep. de ingeniería de sistemas y automática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla
Sevilla, 2017

Trabajo Fin de Grado

Grado de ingeniería electrónica, robótica y mectrónica

Intensificación en automática y robótica

Detección de objetos en ROS para aplicaciones en almacenes

Autor:

Felipe Pérez Molina

Tutor:

Miguel Ángel Ridao Carlini

Profesor Titular de Universidad

Departamento de ingeniería de Sistemas y Automática

Escuela Técnica Superior de ingeniería

Universidad de Sevilla

Sevilla, 2017

Trabajo Fin de Grado:

Detección de objetos en ROS con HOG,SVM y features2d de OpenCV

Autor: Felipe Pérez Molina

Tutor: Miguel Ángel Ridao Carlini

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

Presidente:

Vocales:

Secretario:

Acuerdan otorgarle la calificación de:

Sevilla, 2017

El secretario del Tribunal

Índice

Índice de ilustraciones

Índice de tablas

1. Introducción

Acciones tan triviales para un ser humano como distinguir un objeto del fondo, o reconocer intuitivamente la posición de éste suponen un gran reto para la computación. A lo largo de su evolución se han desarrollado distintos algoritmos cada vez más complejos y funcionales de acorde con la potencia computacional del momento. La disciplina encargada de este estudio recibe el nombre de **visión artificial o visión por computador**.

La visión artificial o visión por computador se encarga de la obtención de datos de una cámara, su procesamiento y análisis con el fin de producir información que pueda ser tratada por un computador.

Ésta se encuentra en numerosos ámbito desde la medicina, la industria hasta en tratamiento de imágenes en un evento deportivo. En este caso el ámbito que nos incumbe es el de la robótica.

La visión juega un papel esencial en el campo de la robótica ya que un robot interactúa con el entorno y por lo tanto, requiere de información sobre éste. No tan sólo para evitar un posible obstáculo si no para poder detectar ciertos objetos deseados por ejemplo.

Este Trabajo Fin de Grado tratará sobre esto último: **detección de objetos** para lugares como un almacén, un supermercado.... Presentaré una serie de técnicas para poder detectar unos objetos determinados aunque se podría extender a otros objetos.

1.1 Motivación

Siendo un alumno del Grado de Ingeniería Electrónica, Robótica y Mecatrónica cursando la intensificación de Robótica y Automática quería realizar un TFG que estuviese relacionado con el campo de la robótica y que fuese una aplicación real. Habiendo cursado la asignatura de Sistemas de Percepción quería ampliar mis conocimientos en el terreno de la visión por computador.

Además quería conocer cómo funciona el framework de ROS, ya que es un meta sistema operativo de código abierto que se encuentra ampliamente adoptado por la investigación, centros educativos y por la industria y conocer la programación orientada a objetos tan ampliamente usada.

1.2 Objetivos

Los objetivos principales de este Trabajo Fin de Grado es el desarrollo e implementación de algoritmos de visión para la detección de objetos.

Además, los objetivos secundarios que se han establecido son los siguientes:

- Implementación de los algoritmos en el *framework* ROS.
- Comparación entre distintos algoritmos.
- Utilización de técnicas que requieran la menor intervención humana.

Para poder abordar estos objetivos requerimos de:

- Un ordenador
- Cámara: en este caso Kinect v2.

1.3 Organización de la memoria

La memoria se encuentra estructurada de la siguiente forma:

1. Estado del arte: En esta sección expondré un breve descripción de las técnicas usadas a lo largo de este Trabajo Fin de Grado que en el siguiente apartado se desarrollaran sus aspectos más técnicos.

2. Conceptos teóricos: en esta sección se explicará todos los conceptos utilizados para poder lograr entender la parte de la implementación

3. Implementación: en esta sección se describirá en detalle cómo se ha implementado los conceptos explicados,así como su diseño y herramientas utilizadas.

4. Pruebas y resultados: En este apartado se expondrá todas las pruebas a las que se ha sometido los distintos algoritmos y sus resultados.

5. Conclusiones: Por último, se expondrá las conclusiones sacadas de los experimentos realizados y una valoración acerca de la realización de este trabajo fin de grado.

6. Especificaciones técnicas: lista del hardware utilizado con sus características.

7. Manual usuario: se explicará qué paquetes son necesarios y cómo instalarlos.

8. Referencias: lista completa de todas las referencias que he utilizado para realizar el Trabajo Fin de Grado, desde los documentos que he utilizado para desarrollar los conceptos teóricos hasta los repositorios de los códigos utilizados.

2. Estado del Arte

En la actualidad, existen grandes compañías que apuestan por los sistemas automatizados para la carga/descarga de sus almacenes, empresas tales como Amazon ha inaugurado en España [26] instalaciones con una superficie de más de 17,148 metros cuadrados en Castellbisbal. En esta área se encuentran trabajando unos 350 robots móviles moviendo los productos.

En este almacén el robot, bautizado como Drive, se encarga de desplazar los productos en unas estanterías especiales denominadas Pods con una carga máxima de hasta 1300 Kg. El sistema le manda el mensaje de que tiene que mover que una estantería de un lado para otro y a través de sus sensores evita obstáculos y planifica su trayectoria.

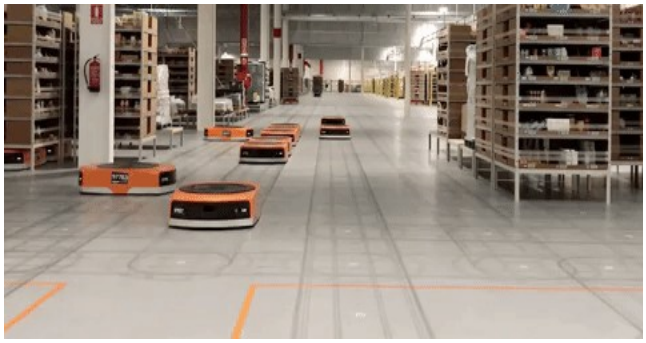


Ilustración 1: Robot Drive de Amazon

Ilustración 2: Estantería especial, Pod

Este Trabajo Fin de Grado difiere de estos robots móviles en el sentido de que mi aplicación se utilizaría en robots móviles con realimentación visual a través de una cámara. Como operario deseas que el robot tenga en su sistema un mapa del lugar, y mandarle que coja cierto objeto y que vuelva con él.

Para ello debe tener una cámara para detectarlo, un sensor de profundidad para saber dónde se encuentra y un brazo manipulador para cogerlo. Sin embargo, no sólo hace falta una serie de componentes, se requiere de un sistema para poder controlar el robot y poder comunicarse tanto con otros robots como con los operarios.

La estrategia que más se adecua a este tipo de sistemas son aquellos que se encuentran basados en **sistemas distribuidos** debido a las numerosas ventajas que posee con respecto a los **sistemas centralizados** que requieren de una súper máquina que realice todos los cálculos.

Si bien las ventajas que ofrece el empleo de los sistemas distribuidos son notables con respecto a la mayoría de los parámetros, exceptuando el costo económico y la seguridad, estos también requieren de un mecanismo que permita desarrollar y ejecutar aplicaciones de forma distribuida y, al mimo tiempo, debe ser capaz de manejar la heterogeneidad de hardware presente en dichos sistemas y proveer un conjunto común de servicios específicos. Tal mecanismo es conocido como **middleware**.

Los middlewares son el corazón de los sistemas distribuidos y, por tal motivo, son requeridos cada vez que se desee desarrollar sistemas de este tipo. Este provee un conjunto de componentes básicos de alto nivel que simplifican la construcción del sistema y permiten, a desarrolladores e investigadores, abstraerse de los detalles de la implementación a bajo nivel, tales como control de concurrencia, gestión de transacciones y la comunicación a través de la red, entre otros.

Existen una gran variedad de middlewares en la actualidad lo cual se debe, entre otras razones, a la diversidad de temas en los que pueden ser aplicados. En este proyecto se ha decidido utilizar el middleware llamado “Robotic Operating System” (**ROS**) debido a sus numeras ventajas que se expondrá a continuación.

2.1 ¿Por qué ROS?

ROS ofrece una serie de características que lo hace único, las cuales lo hacen perfecto para este tipo de aplicaciones que se pueden resumir de la siguiente forma:

1. Topología de comunicación punto a punto

   Básicamente consiste en un conjunto de procesos que se ejecutan en diferentes computadoras, las cuales se interconectan en tiempo real siguiendo una topología de comunicación punto a punto.

   La idea de esta filosofía es que si el robot debe realizar una tarea específica entonces sólo active y comunique aquellos procesos que son necesarios para lograr dicho objetivo no saturando el ancho de banda de la red.

2. Basado en herramientas

   Para gestionar la complejidad de ROS sus desarrolladores implementaron una gran cantidad de pequeñas herramientas que permiten compilar y ejecutar varios componentes de ROS.

3. Multi-lenguaje

   Los desarrolladores pueden trabajar con ROS independientemente del lenguaje de programación: c++, Python,Octave y LISP.

4. Ligero

   Ros te ofrece la posibilidad de reutilizar los códigos y controladores generados durante el desarrollo de sus proyectos. Para poder lograr esto el desarrollo de algoritmos y controladores ocurre dentro de bibliotecas independientes que no contienen ninguna dependencia en ROS, lográndose con esto ubicar de forma virtual toda la complejidad.

5. Gratis y de código abierto

   Todo el código fuente de ROS está publico y disponible para cualquier usuario.

Todas estas ventajas sumadas a que es una buena experiencia para poder aprender cómo funciona me han llevado a tomar la decisión de utilizarlo como base en este proyecto.

2.2 Técnicas de detección

Como ya he mencionado mi sistema se encuentra basado en la información visual para detectar los objetos. Existen numerosas técnicas y algoritmos para detectar ciertos patrones en una imagen e identificarlos como una clase conocida. Ahora bien, en este proyecto se han abordado dos modalidades para enfrentarnos a este reto: un método basado en aprendizaje y métodos basados en la extracción de características.

2.2.1 Métodos basados en aprendizaje

Mediante este tipo de métodos nos encontramos con un problema de clasificación de dos clases (objeto, no-objeto). La primera etapa que se realiza en estos métodos, es una etapa de aprendizaje: a partir de un conjunto de imágenes se realiza el entrenamiento de un clasificador mediante el uso de las características extraídas previamente de dicha imagen. Es importante que el conjunto de entrenamiento tenga una alta variabilidad para conseguir un clasificador robusto.

Para determinar que secciones de las imágenes se encuentra el objeto , se extrae un vector de características y se procede a su clasificación.

Para extraer las características he usado el **descriptor HOG** ya que funciona muy bien en condiciones de ruido y de cambios en la iluminación. Por otro lado, el clasificador elegido ha sido el **clasificador SVM** desarrollados por Vladimir Vapnik debido a sus buenos resultados. Ambos conceptos se explicarán más adelante.

2.2.2 Métodos basados en la extracción de características

Éstos métodos se basan en el uso de detectores para obtener los posibles puntos de interés de un objeto, tales como esquinas, se selecciona aquellos puntos de interés importantes mediante extractores de puntos de interés en una imagen que se usaremos como plantilla y en la imagen en la que deseamos encontrar el objeto.

Mediante **algoritmos del vecino más cercano** se detecta el objeto en la imagen seleccionando aquellos puntos de interés que se corresponden con los de la imagen plantilla.

Los detectores/extractores que he usado son aquellos presentes en **features2d** perteneciente a la librería libre de visión por computador **OpenCV**: SIFT, SURF, ORB, MSER, STAR, BRISK...**.** Además para encontrar la similitud entre los puntos de interés de ambas imágenes he usado la librería **FLANN** implementado también por OpenCV.

3. Conceptos teóricos

Una vez mencionados en el apartado anterior por qué he decidido usar ROS y una breve descripción de las técnicas usadas, en este apartado desarrollaré todos los aspectos matemáticos y explicaré qué es lo que hace todas las técnicas mencionas, comenzando con HOG y el SVM, y luego los algoritmos implementados en la librería features2d de OpenCV.

Además explicaré una serie de conceptos que mencionaré a lo largo del documento relacionados con el *framework* ROS.

3.1 ROS

Para poder entender este proyecto debemos primero conocer algunos conceptos clave en los que se cimienta ROS, [0]:

- **Master**: El ROS Master proporciona el registro de nombre y consulta el resto del grafo de computación. Sin el maestro, los nodos no serían capaces de encontrar al resto de nodos, intercambiar mensajes o invocar servicios.
- **Nodos**: Los nodos son procesos que realizan la computación. ROS está diseñado para ser modular en una escala de grano fino; un sistema de control de robots comprende por lo general muchos nodos. Por ejemplo, un nodo controla un láser, un nodo controla los motores de las ruedas, un nodo lleva a cabo la localización, un nodo realiza planificación de trayectoria, un nodo proporciona una vista gráfica del sistema, y así sucesivamente. Un nodo de ROS se escribe con el uso de las librerías cliente de ROS, roscpp y rospy.
- **Servidor de parámetros**: El servidor de parámetros permite almacenar datos mediante un clave en una localización central, actualmente es parte de Master.
- **Mensajes**: Los nodos se comunican mediante el paso de mensajes. Un mensaje es una estructura de datos compuestos. Un mensaje comprende una combinación de tipos primitivos y mensajes. Los tipos primitivos de datos (integer, floating point, boolean, etc.) están soportados, como los arrays de tipo primitivo. Los mensajes pueden incluir estructuras y arrays (como las estructuras de C).
- **Tópicos**:Los mensajes son enrutados por un sistema de transporte que utiliza semántica publicador/subscriptor. Un nodo envía mensajes publicando en un tópico. El tópico es un nombre que se usa para identificar el contenido del mensajes. Un nodo que esta interesado en cierto tipo de datos se subscribirá al tópico apropiado. Pueden existir muchos publicadores concurrentemente para un solo tópico y un nodo puede publicar y/o suscribirse a múltiples tópicos. En general, los publicadores y subscriptores no son conscientes de la existencia de los otros. La idea es desacoplar la producción de información de su consumición. Lógicamente, uno puede pensar que un tópico es mensaje fuertemente tipado

en un bus. Ese bus tiene un nombre y nadie se puede conectar al bus para enviar o recibir mensajes si no están correctamente tipados.

- **Servicios**: El modelo de publicar/subscriptor es un paradigma de comunicación muy flexible, pero en muchos casos el transporte en un único sentido no es suficiente para las interacciones de petición y respuesta que a menudo se requieren en un sistema distribuido. La petición y respuesta se realiza a través de los servicios, que se definen a partir de una estructura de mensajes: una para la petición y otra para la respuesta. Un nodo proporciona un servicio con un nombre y un cliente utiliza dicho servicio mediante el envío del mensaje de petición y espera a la respuesta. La librería cliente de ROS generalmente presenta esta interacción como si fuera una llamada a procedimiento remoto.

3.2 HOG

3.2.1 Conceptos básicos

El **gradiente** se define como un cambio de intensidad de la imagen en una cierta dirección, aquella dirección en la que el cambio de intensidad es máximo. Se calcula para cada uno de los píxeles de la imagen y queda definido para cada píxel los dos valores, la dirección donde el cambio de intensidad es máximo y la magnitud del cambio en esa dirección.

3.2.1 Cálculo del gradiente

El gradiente se puede calcular de varias formas diferentes. En el contexto del descriptor HOG este cálculo se realiza a partir de la diferencia de intensidad de los píxeles vecinos en dirección tanto horizontal como vertical tal y como se explica en [1].

$$dx = I(x, y) - I(x-1, y) \qquad dy = I(x, y) - I(x, y-1)$$

Siendo I la imagen, dx diferencia en la dirección horizontal y dy: diferencia en la dirección vertical.

A partir de la diferencias en vertical y en horizontal se puede calcular la orientación y la magnitud del Gradiente como:

Orientación $\theta = arctg(dy/dx)$ $Magnitud = \sqrt{(dx^2 + dy^2)}$

3.2.2 Cálculo de los histogramas

Tal y como está definido el gradiente los valores altos de magnitud se corresponden con contornos o con la silueta de objetos con el fondo. Mientras que la orientación nos aporta la información de cómo es ese contorno.

El problema con esta representación a nivel de píxel del gradiente es que difícilmente se puede utilizar junto con un clasificador en un sistema de detección de objetos, ya que un clasificador normalmente requiere como entrada una representación global de toda la imagen de una dimensión fija y además esta información a nivel de píxel puede ser muy sensible a pequeñas variaciones tanto en la forma como en la localización del objeto en la imagen.

Por ello debemos extender esta información local de un píxel a una representación global de la imagen. El descriptor HOG(Histogram of Oriented Gradients) actúa en dos pasos para conseguir esta representación, tal y como se encuentra explicado en [2]:

En primer lugar, divide la imagen en celdas de tamaño fijo y para cada uno de esas celdas calcula histograma de las orientaciones de los gradientes en esa celda.

A continuación, se calcula el histograma para para todas las celdas y todos esos histogramas se combinan para dar lugar la representación global de la imagen en forma de vector de características.

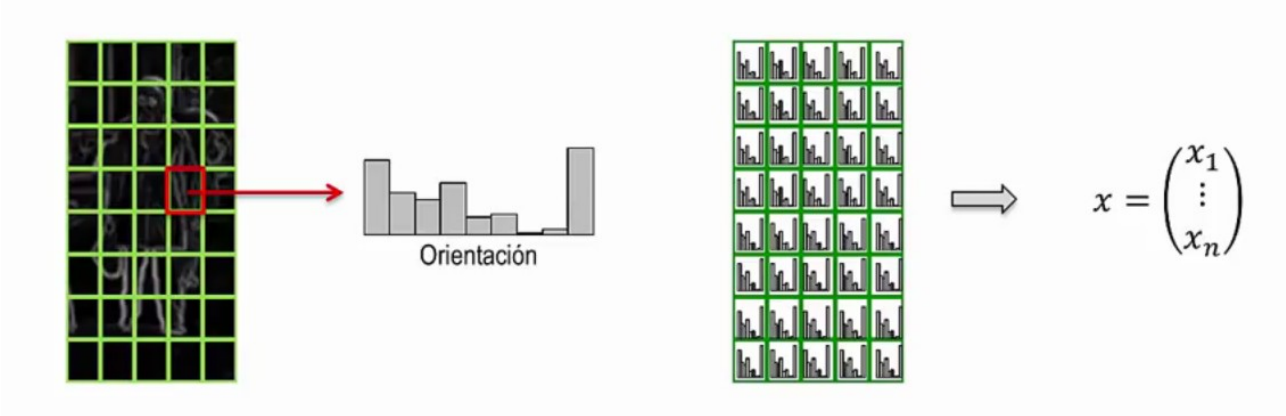


Ilustración 3: Se divide la imagen en celdas, se calcula los histogramas y se concatena

Uno de los parámetros que debemos considerar es entonces el tamaño de la celdas, valores entre 6 y 8 píxeles de ancho por alto suelen ser los habituales. Otro de los parámetros que debemos fijar es la división del rango de orientaciones en un número de intervalos fijos.

Una vez fijados estos parámetros calcularemos los gradientes de cada píxel dentro de la celda y los clasificaremos en el rango de orientaciones establecido.

El principal problema de este método es que es sensible a pequeños variaciones del gradiente provocando que se clasifique de forma errónea en un rango u otro.
Para solucionar este problema debemos asignar cada gradiente a los dos intervalos más cercanos con un peso proporcional a la distancia de la orientación al centro de cada intervalo.

3.2.3 Cálculo del descriptor

Uno de los objetivos de cualquier descriptor debe ser conseguir la máxima invarianza posible a todas aquellas variaciones que se pueden producir en la imagen de entrada, ya sean de iluminación,posición, escala, aspecto, etcétera.

Cuando tenemos cambios de iluminación la intensidad del gradiente va a cambiar. Por lo tanto, estos cambios en la intensidad del gradiente también se van a reflejar en los valores del histograma.

Para minimizar estas diferencias en la descripción de las imágenes, va a ser conveniente normalizar los valores de los histogramas, con el objetivo de conseguir que la magnitud global del gradiente sea similar creando un vector de características tal y como se explica en [3].

Debemos tener en cuenta que los cambios de iluminación puede que no sean constantes a lo largo de toda la imagen, no va a ser suficiente aplicar una única normalización uniforme en toda la imagen sino que va a ser preferible una normalización local adaptada a cada una de las zonas de la imagen.

Así pues debemos agrupar las celdas para con el objetivo de normalizarlas, en lo que denominaremos **bloque**. Una configuración bastante habitual en el descriptor HOG es utilizar bloques de dos celdas en horizontal y dos celdas en vertical. Así para cada bloque vamos a coger los histogramas de cada una de las celdas y los vamos a concatenar para obtener el vector con la representación del bloque.

La normalización de los histogramas se va a realizar a nivel de bloque, es decir, vamos a normalizar este vector resultado de concatenar todas las celdas. La normalización se obtiene dividiendo cada uno de los componentes del vector por su norma L2.

$$v=(x_1,\dots,x_n) \quad \rightarrow \quad v'=\frac{v}{\sqrt{||v||_2^2+\varepsilon}}$$

ε es una constante con un valor muy pequeño para evitar las divisiones por 0 en los bloque en los que la intensidad sea muy similar.

En la práctica los bloques se definen de forma que tengan un cierto solapamiento entre ellos. La redundancia que se va a obtener con este solapamiento, va a ayudar a conseguir un descriptor más robusto ante deformaciones y variaciones en la forma del objeto.

Habitualmente, los bloques se colocan con una separación de una sola celda entre ellos tanto en horizontal como en vertical, y la representación final del descriptor HOG se obtiene simplemente concatenando la representación normalizada de todos estos bloques solapados.

3.3 SVM

3.3.0 Definiciones

Una vez extraída la información de una imagen debemos distinguir entre el objeto y el fondo. El encargado de discernir qué es un objeto y qué no recibe el nombre de **clasificador**. Éste hace referencia al algoritmo utilizado para asignar un elemento entrante no etiquetado en una categoría conocida.

3.3.1 Conceptos básicos

Como bien sabemos existen infinitas formas de trazar líneas que particionen el espacio de características para el problema de dos clases(el objeto y el fondo). Para obtener esta línea podemos utilizar 2 métodos bien diferenciados:

Por un lado tenemos lo que se conoce como **modelos generativos** que tratan de construir estas fronteras de separación entre clases a partir de estimar la función de densidad de probabilidad que podemos asociar a cada una de las clases.

Por otro lado tenemos lo que conocemos como **modelos discriminativos** que tratan de clasificar las muestras sin tener que generar estas funciones de densidad de probabilidad de las clases, y por lo tanto encuentra la frontera a partir de un conjunto de entrenamiento que sea suficientemente representativo siendo el caso de SVM[4].

3.3.2 Desarrollo matemático

La finalidad de SVM es el de optimizar este margen a partir de los vectores de soporte(aquellas muestras cercanas al margen) para diferenciar las dos clases. Para poder desarrollar matemáticamente el SVM me he ayudado de [5] como referencia.

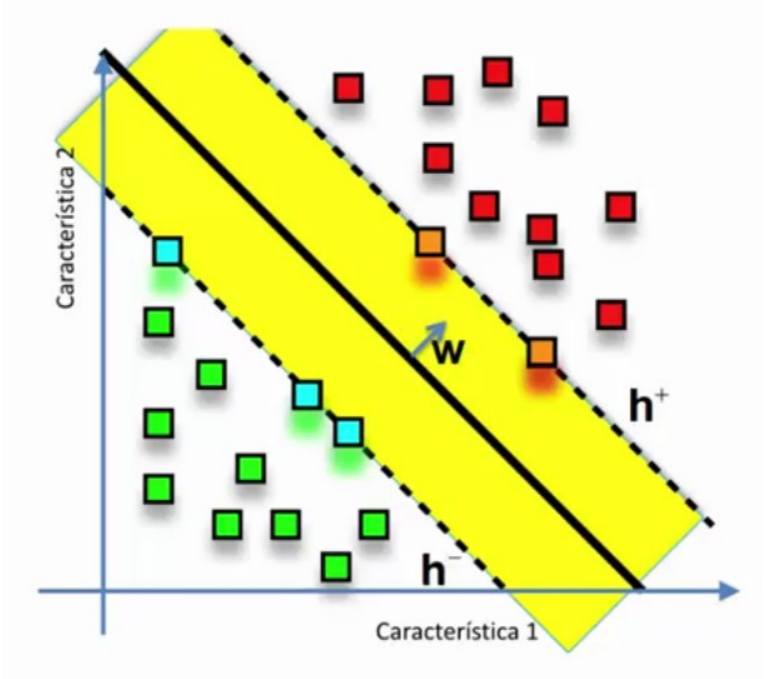


Ilustración 4: Representación del de características, en las que las clases naranja y azul representa los vectores de soporte

Para poder diferenciar entre las dos clases deberemos encontrar el la región más amplia del espacio de características que nos permite separar ambas clases y que está limpia de muestras. Cualquier otra solución alternativa a este hiperplano nos va a dar como resultado un margen más estrecho.

Al ser el Support Vector Machines un clasificador lineal implica que existe un hiperplano en solución, cuya solución se puede expresar como:

$w^T x_i + b = 0$ Siendo w un vector ortogonal al hiperplano solución y b un coeficiente de intersección.

Este hiperplano en solución se obtiene como el hiperplano medio entre otros dos hiperplanos el h+ y h- que son los que contienen los vectores de soporte. Siendo la expresión $w^T x_i + b = +1$ para definir el hiperplano solución perteneciente a la clase de imágenes positivas(aquellas imágenes en las que se encuentra el objeto) y $w^T x_i + b = -1$ para las imágenes negativas(fondo sin el objeto).

Estas dos condiciones se pueden agrupar en la siguiente expresión:

$y_i(w^T x_i + b) \geq 1$ representando la "y" a la clase que pertenece: $y_1 = +1$ e $y_2 = -1$

Para calcular el margen deberemos hallar la distancia media entre h+ y h- aproximándose a:

$$margen = \frac{2}{||w||}$$

Maximizar el margen puede equivaler a minimizar el inverso. Esto se realiza para facilitar los cálculos quedándonos un problema de **optimización cuadrática:**

$$\left.\begin{array}{l} \Theta(w) = \frac{1}{2} w^T w \\ y_i(w^T x_i + b) \geq 1 \end{array}\right\}$$

Para poder resolver este problema nos ayudamos con la herramienta matemática del Lagrangiano L:

$$L(X, \alpha) = f(x) + \sum_i \alpha g_i(x) \qquad \forall \alpha_i \geq 0$$

Siendo f(x) → `Θ(w)` `gi(x)` → $y_i(w^T x_i + b) \geq 1$ α → Multiplicadores de Lagrange

Quedándonos como resultado:

$$o(\alpha)=\frac{-1}{2}\sum_i \sum_j \alpha_i \alpha_j y_i y_j x_i^T x_j + \sum_i \alpha$$

$$\sum_i \alpha_i y_i = 0 \quad \text{si} \quad \alpha_i \geq 0$$

Denominado **SVM DUAL**

Finalmente, para poder resolver el problema de SVM DUAL haremos uso las funciones Kernel (las cuáles no voy a entrar en detalle) introduciéndonos un serie de parámetros que deben ser ajustados.

3.4 Features2d

3.4.1 Detección de Características

En la visión por computador y en el procesamiento de imágenes el concepto de Detección de características se refiere a los métodos que poseen un nivel de abstracción que le permite tomar la decisión en cada punto de la imagen si es una característica o no.

Las características resultantes serán subconjuntos de la imagen, a menudo en forma de puntos aislados, curvas continuas o regiones conectadas.

Los métodos de detección de características que he usado en este TFG son los siguientes:

SIFT

El detector SIFT(Scale-Invarient Feature Transform) es un algoritmo muy robusto, invariante a la escala,[6], basado en los siguientes pasos:

1. En primer lugar, SIFT crea un espacio escalado dividido en octavas y genera imágenes borrosas progresivamente. Toma la imagen original, la reescala a la mitad y la emborrona (octava). La cantidad de octavas dependerá del tamaño de la imagen.

   Matemáticamente, “emborronar” se refiere a la convolución del operador Gaussiano y la imagen. El desenfoque Gaussiano tiene una expresión o operador que es aplicado a cada píxel de la imagen.

   $$L(x,y,\sigma)=G(x,y,\sigma)*I(x,y)$$

   Siendo L la imagen desenfocada, G es el operador Gaussiano, “x” e “y” las coordenadas de la imagen y σ el parámetro de desenfoque.

   $$G(x,y,\sigma)=\frac{1}{2\pi\sigma^2}e^{-(x^2+y^2)/(2\sigma^2)}$$

2. Generamos un nuevo conjunto de imágenes a partir de las imágenes emborronadas que habíamos creado denominado Diferencia de Gaussianas (DoGs) que serán vitales para encontrar los puntos de interés en la imagen.

   Para ello, se seleccionan dos imágenes consecutivas de una octava y se restan, luego a continuación se toma el siguiente par consecutivo y se repite la resta. Repitiendo para todas las octavas se obtendrá el conjunto mencionado.

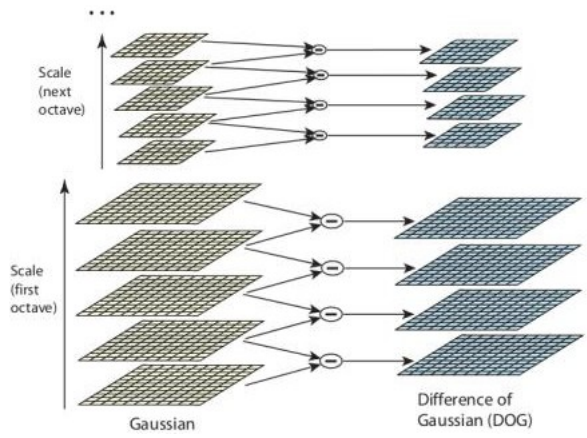


Ilustración 5: Aproximación de la Laplaciana Gaussiana

3. Localizamos máximos y mínimos en las imágenes DoG generadas en el paso anterior. Para ello iteramos para cada píxel y comprobamos todos sus vecinos tanto de su escala como una superior e inferior.

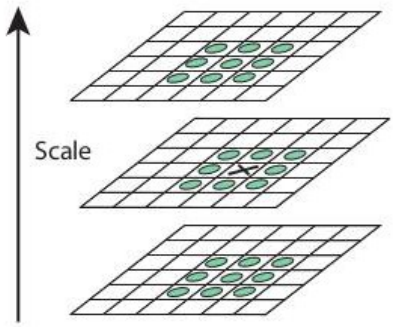


Ilustración 6: Caso de máx/min

La "X" marca el píxel actual, éste será un posible **punto de interés** si es el mayor de los 26 vecinos (9 vecinos arriba, 9 vecinos abajo, 8 vecinos de su escala actual). Una vez verificados los máximos y mínimos aproximados se busca la posición exacta de éstos con la expansión de de la series Taylor igual a 0.

$$D(x) = D + \frac{\partial D^T}{\partial x} + \frac{1}{2} x^T \frac{\partial^2 D}{\partial x^2} x$$

4. Rechazamos los puntos clave que posean un bajo contraste o si se encuentran en un borde.
5. Calculamos la orientación y la magnitud del Gradiente.

SURF

SURF (Speed Up Robust Features) está basado en los mismos principios que SIFT, pero utiliza un esquema diferente otorgándole una velocidad mayor. Para explicar este algoritmo me he documentado en [7].

El algoritmo se encuentra basado en la **matriz Hessiana** debido a su buen funcionamiento en tiempos de cálculo y precisión. Sin embargo, en lugar de utilizar la misma medida que el detector Hessian-Laplace para seleccionar la ubicación y la escala, éste se basa en el determinante de la matriz para ambos. Dado un punto X = (x,y) en la imagen I, la matriz Hessiana H(x,σ) se define como:

$$H(X,\sigma)=\begin{bmatrix} L_{xx}(X,\sigma) & L_{xy}(X,\sigma) \\ L_{xy}(X,\sigma) & L_{yy}(X,\sigma) \end{bmatrix}$$

Donde $L_{xx}(X,\sigma)$ es la convolución de la derivada de segundo orden Gaussiana con la imagen I en el punto X, y similar para $L_{xy}(X,\sigma)$ y $L_{yy}(X,\sigma)$.

Como los filtros Gaussianos no son ideales, y dado el éxito de SIFT con las aproximaciones LoG, SURF va más allá y los aproxima con "box filters". Éstos aproximan a las derivadas de segundo orden Gaussianas con la peculiaridad que pueden ser evaluados muy rápidos. Se denominaran estas aproximaciones como Dxx,Dxy y Dyy.

Los 9x9 filtros que muestra la ilustración son aproximaciones de la derivada de segundo orden Gaussiana con σ=1.2.

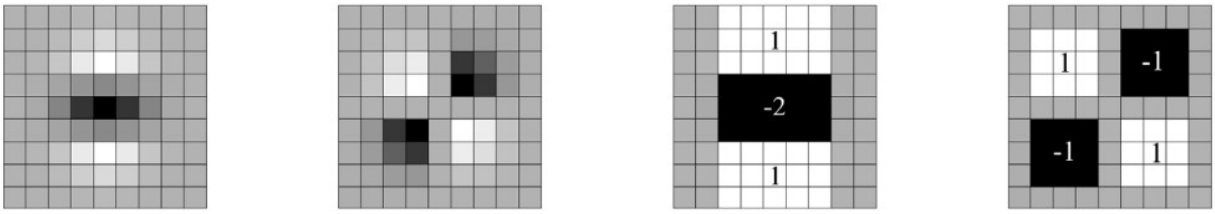


Ilustración 7: La derivada segunda Gaussiana y su aproximación con box filters

El peso aplicado en las regiones rectangulares se mantiene constante para la eficiencia computacional, pero necesitamos equilibrar aun más los pesos en el determinante de la Hessiana con $\frac{|L_{xy}(1.2)|_F|D_{xx}(9)|_F}{|L_{xx}(1.2)|_F|D_{xy}(9)|_F}=0{,}912..\simeq 0.9$, donde $|x|_F$ es la norma Frobenius. Esto implica:

$$det(H_{approximada})=D_{xx}D_{yy}-(0.9\,D_{xy})^2$$

Además, la respuesta del filtro está normalizado con respecto al tamaño de la máscara utilizada, garantizando que la norma de Frobenius se mantenga constante.

El **espacio escalado** se implementa como imágenes piramidal. Las imágenes son repetidamente suavizadas con un Gaussiano y posteriormente sub-muestreadas para lograr un alto nivel de la pirámide. Debido al uso de box filters y de imágenes integrales, no tiene por qué aplicar el mismo filtro a la salida de un capa previamente filtrada, sino que puede aplicar los filtros de cualquier tamaño exactamente a la misma velocidad directamente sobre la imagen original, e incluso en paralelo. Por lo tanto, el espacio escalado se analiza escalando el tamaño del filtro en lugar de reducir el tamaño de la imagen original.

La salida del filtro 9x9 anterior se considera como la capa inicial, con escala s=1,2 (correspondiente con el Gaussiano σ=1.2). Las capas siguientes se obtienen filtrando la imagen con filtros cada vez mas grandes: 9x9, 15x15,21x21,27x27,etc. Para escalas mayores de la pirámide, el paso entre filtros también debe escalarse en consecuencia: para cada nueva octava el filtro se duplica de 6 a 12 a 24. Simultáneamente, los intervalos de muestreo para la obtención de puntos de interés también se deben duplicar.

Con el fin de localizar los puntos de interés en la imagen y sobre las escalas, se aplica la supresión no-máxima en un vecindario 3x3x3. Los máximos del determinante de la matriz Hessiana son luego interpolados en escala y espacio en la imagen.

FAST

El detector FAST [8], usa un círculo de 16 píxeles (un círculo Bresenham de radio 3) para clasificar si un candidato de punto *p* es una esquina. Cada píxel en el circulo es etiquetado del número entero 1 al 16 en el sentido de las agujas del reloj. Si un conjunto de N píxeles contiguos en el círculo son mas brillantes que la intensidad del candidato *p* (denotado por I_p) más un valor umbral *t* o todo es más oscuro que la intensidad del píxel candidato p menos el valor umbral, el candidato *p* es clasificado como una esquina. Las condiciones pueden ser escritas como:

1. Conjunto de N píxeles contiguos S, $\forall x \in S$, la intensidad de x (I_x) > I_p + umbral *t*.
2. Conjunto de N píxeles contiguos S, $\forall x \in S$, $I_x < I_p - t$.

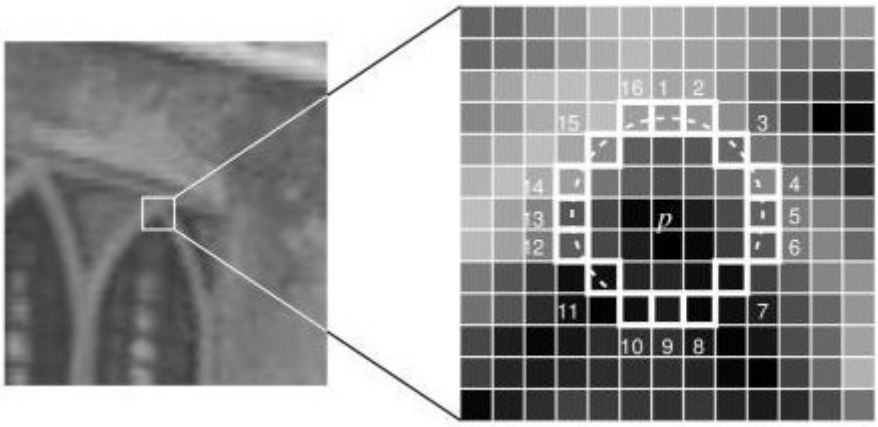


Ilustración 8: Metodología FAST

Hay una compensación de elegir N, el número de píxeles contiguos y el umbral t. Por un lado el número de esquinas detectadas no debe ser demasiado, y por otro lado, el alto rendimiento no se debe lograr sacrificando la eficiencia computacional.

ORB

ORB (Oriented FAST and Rotate BRIEF) utiliza el algoritmo de FAST detector para detectar los puntos de interés pero añadiéndole un cálculo eficiente de la orientación: **Orientación por la intensidad del centroide** tal y como se explica en [9]**.** La intensidad del centroide asume que la intensidad de una esquina está desplazada de su centro, y este vector de desplazamiento se puede utilizar para calcular su orientación.

Recordemos que los momentos de un píxel se definen como:

$$m_{pq}=\sum_{x,\,y} x^p\, y^q I(x,y)$$

y con estos momentos podemos hallar el centroide:

$$C=(\frac{m_{10}}{m_{00}},\frac{m_{01}}{m_{00}})$$

Una vez calculado el centroide podemos construir un vector desde el centro de la esqina, O, hasta el centroide *OC.* La orientación de la esquina sera entonces simplemente:

$$\theta=arctag(\frac{m_{01}}{m_{10}})$$

Para mejorar la invarianza de esta medida, ORB se asegura que los momentos son calculados con x e y permaneciendo dentro de la región circular de radio *r.* Se selecciona r para permanecer dentro del círculo de Bresenham (FAST) , de modo que esa *x* e *y* se ejecuten en [-r,r].

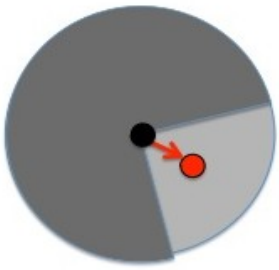


Ilustración 9: Esquina detectada con FAST y orientación centroide

STAR

El detector de puntos de interés STAR fue implementado como parte de una librería de OpenCV. Es derivado del detetector CenSurE (Center Surround Extrema) tal y como se resume en el documento [10].

El detector utiliza una aproximación de dos niveles del filtro de Laplacian de Gaussian (LoG). La forma circular de la máscara se sustituye por una aproximación que preserva la invariancia rotacional y permite el uso de imágenes integrales para un cómputo eficiente.

El algoritmo STAR crea un espacio-escalado sin interpolación, aplicando máscaras de diferentes tamaños.

GFTT

GFTT (Good Features to Track) es una modificación del detector de esquinas Harris que muestra unos mejores resultados. La función principal para detectar las esquinas viene dada por:

$R = \lambda_1 \lambda_2 - k(\lambda_1 + \lambda_2)^2$ Donde λ son los autovalores del tensor de la imagen como suma de diferencias de cuadrado (SDC).

En lugar de esa expresión, GFTT propone:

$R = min(\lambda_1, \lambda_2)$

Si es más grande un cierto valor del umbral es considerado como esquina. Si dibujamos el espacio generado por el Detector de esquinas Harris, obtenemos la siguiente imagen:

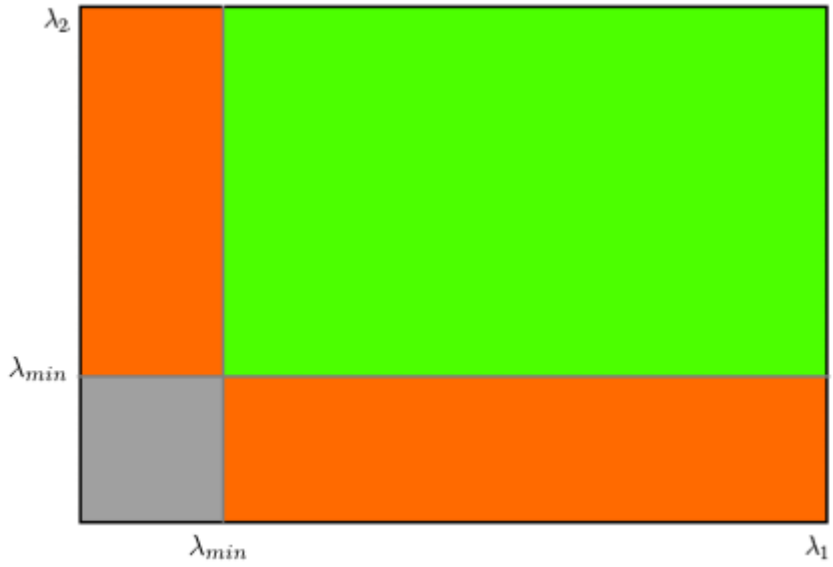


Ilustración 10: Caso de esquina en GFTT

Como podemos ver, solo cuando λ_1 y λ_2 superan un cierto valor λ_{min} es considerado como esquina(región verde). Para explicar este método he utilizado la fuente [11].

MSER

MSER (Maximally Stable Extremal Regions) es un método para la detección de regiones en una imagen. El algoritmo de MSER extrae de la imagen un número de regiones covariantes, llamadas MSERs (una componente estable de algunos conjuntos de niveles de gris de la imagen).

Los MSERs son aquellas regiones que se mantienen casi iguales a través de una amplia gama de regiones:

- Todos los píxeles por debajo de un umbral son blancos y todos los que lo superan o igualan negros.
- Si se nos muestra una secuencia de imágenes I_t con el fotograma *t* correspondiente al umbral t, veremos una imagen en negro y luego manchas blancas correspondientes a los mínimos de intensidad local y luego se harán más grandes.
- Estas manchas blancas eventualmente se fusionaran hasta que la imagen sea completamente blanca.
- El conjunto total de de todos los componentes conectados en la secuencia es el conjunto de todas las regiones extremas.

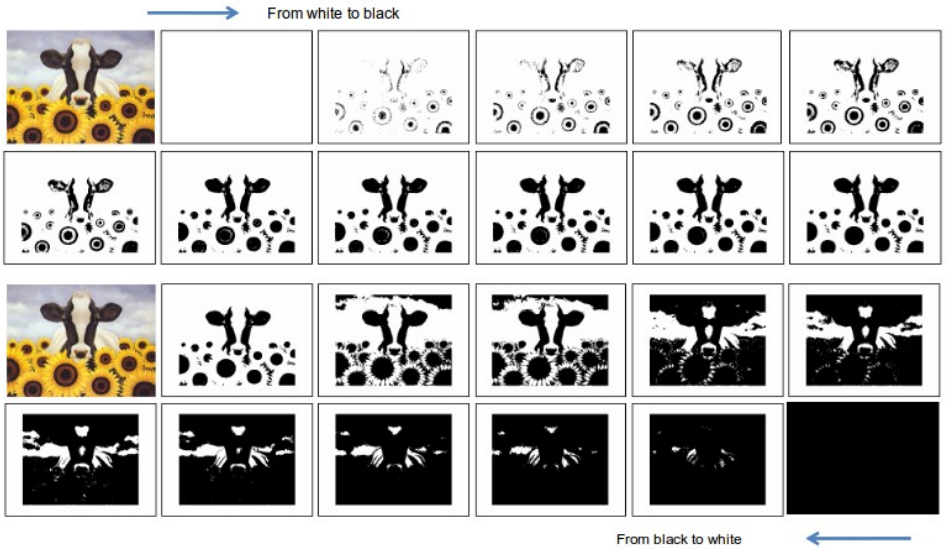


Ilustración 11: Ejemplo de aumento/disminución del umbral

Una vez comprendido que características tiene un MSER, debemos explicar cómo funciona el algoritmo, tal y como se indica en [12]:

1. Barremos el umbral de intensidad de negro a blanco.
2. Extraemos las componentes conectadas(“Extremal Regions”).
3. Encontramos un umbral en el que la región es “máximamente estable”, es decir mínimo local del crecimiento relativo de su cuadrado. Debido a la naturaleza discreta de la imagen, la región coincide con la región real.
4. Aproximar la región con una elipse.
5. Mantener los descriptores de regiones como características.

BRISK

En el caso de BRISK (Binary Robust Invarian Scalable Keypoints) [13] las capas del espacio-escala de la "pirámide" consiste en n octavas ci y n entre-octavas di, desde i= {0,1, . . . ,n−1} tipicamente n = 4. Las octavas son formadas progresivamente muestreando la mitad de datos ("halfsampling")de la imagen original (correspondiendo con c0). Cada entre-octava di es localizada entre las capas ci y ci+1.

Para el caso de la primera entre-octava d0, ésta es obtenida por "downsampling" a la imagen original c0 por un factor de 1.5, mientras que el resto de las capas entre-octavas son derivadas sucesivamente por el "halfsampling". Por lo tanto, si *t* denota la escala entonces: $t(ci) = 2^i$ y $t(di) = 1{,}5*2^i$.

Inicialmente el detector FAST 9-16 es aplicado en cada octava y entre-octava separadamente usando el mimo umbral T para identificar las posibles regiones de interés. A continuación, los puntos pertenecientes a esas regiones se someten a no-máxima supresión en el espacio-escalado: primero, los puntos en cuestión deben cumplir la condición de máximo con respecto a sus 8 vecinos en la misma capa. En segundo lugar el valor de los vecinos de las capas superiores e inferiores de entre-octavas deben ser menores también. Para determinar donde se encuentra el punto de interés en las entre-octavas se interpola tal y como muestra la imagen:

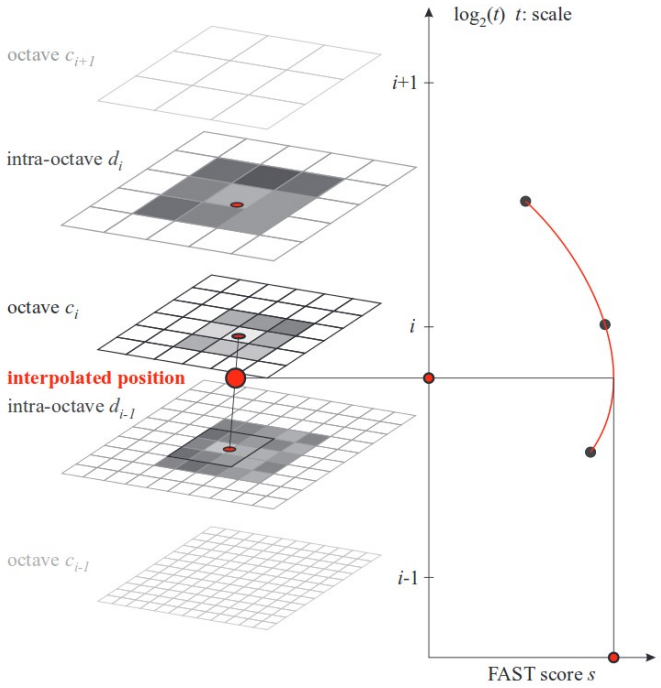


Ilustración 12: Espacio-escalado de BRISK

Para el caso de la primera octava aplicamos la máscara FAST 5-8 para obtener la entre-capa d_{-1}.

Si superan las condiciones dadas se obtiene los subpíxeles de las puntos de interés. Para refinar y obtener la localización del punto en la imagen original primeramente, obtenemos la función de mínimos cuadráticos 2D a cada una de las cada 3 capas, obteniendo como resultado 3 puntos. A continuación se forma una parábola que pase por esos tres valores y su máximo será el valor estimado. Como paso final debemos re-interpolar a la imagen original.

Ilustración 13: Ejemplo de puntos de interés BRISK

3.4.2 Extracción de características

Cuando los datos de entrada a un algoritmo son demasiado grandes para ser procesados y se sospecha que son redundantes, se deben transformar en un conjunto de representación reducido de características (denominado **descriptor**). La transformación de los datos de entrada en el vector de características se denomina **extracción de características**.

Si se elige cuidadosamente las características, se usará la representación reducida en lugar de los datos de entrada con un mayor tamaño.

BRIEF

BRIEF (Binary Robust Independent Elementary Features),[14], toma la imagen, la suaviza y selecciona un conjunto de $n_d(x,y)$ previamente definidos y construye el vector:

$$v(p;x,y)=\begin{cases} 1, & p(x)<p(y) \\ 0, & para\ otros\ casos \end{cases}$$

Donde p(x) es el píxel con la intensidad en la imagen suavizada de p siendo $x = (u,v)^T$.

Esto es aplicado para todos los n_d paraes localizados para obtener una cadena de bits con dimensión n_d.

$$f_{n_d}:=\sum_{1\le i\le n_d} 2^{i-1} v(p;x_i,y_i)$$

ORB

ORB usa para extraer las características el descriptor BRIEF. Pero dado que no utiliza la orientación, ORB lo modificará acorde con la nueva orientación calculada de los puntos clave, tal y como se describe en [9.1]. Para cada conjunto de características de **n** pruebas binarias localizadas en (x_i,y_i), define una matriz de 2xn, ***S*** que contiene las coordenadas de los píxeles. Luego usando la orientación, θ calculada para cada punto de interés se calcula la matriz S_θ, rotada.

ORB discretiza el ángulo en intervalos de 2π/30 (12 grados), y construye una tabla de búsqueda de patrones BRIEF precalculados. Siempre y cuando el punto clave con orientación θ sea consistente entre las vistas, se utilizará el conjunto correcto de puntos S_θ para calcular su descriptor.

BRIEF tiene una importante propiedad y es que para cada característica bit tiene una gran varianza y una media cerca del 0,5. Pero una vez que se orienta a lo largo de la dirección del punto clave, pierde esta propiedad y se vuelve más distribuida. Además La alta varianza hace que una característica sea más discriminativa, ya que responde diferencialmente a las entradas. Otra propiedad interesante es tener las pruebas no correlacionadas.

Por ello ORB ejecuta una búsqueda codiciosa entre todas las posibles pruebas binarias para encontrar los que tienen tanto alta varianza y medios cerca de 0,5, como las que no presentan correlación. El descriptor resultante es denominado **rBrief**.

SIFT

Para cada punto de interés se toma un vecindario de 16 x 16 puntos como se puede comprobar en [6.1]. Este, a su vez, se divide en sub-bloques de tamaño 4x4. A continuación para cada uno se crea un histograma de orientaciones. La concatenación en un vector de los valores de las cajas de cada histograma para los 16 sub-bloques del punto de interés constituye su descriptor.

SURF

El primer paso para obtener el descriptor una vez calculado el escalado es el cálculo de la **orientación** del punto de interés[7.1]. Para obtener un punto invariante a las rotaciones, iluminación y orientación se utiliza el wavelet de Haar sobre las direcciones de x e y en una región circular de radio 6s, siendo *s* es la escala del punto de interés.

Para la extracción del descriptor, el primer paso es la construcción de una región cuadrada alrededor del punto de interés y con la orientación ya calculada.

La región es dividida en 4x4 sub-regiones cuadradas más pequeñas. Para cada sub-región calculamos unas simples características de 5x5 alrededor de nuestro punto, el Wavelet de Haar para x e y. Por motivos de simplicidad llamaré dx a la respuesta de Wavelet en el eje x y dy a la respuesta en el eje y.Hay que tener en cuenta que “horizontal” y “vertical” es definido en relación a la dirección del punto de interés.

Además para una mayor robustez hacia deformaciones geométricas y errores de localización, las respuestas dx y dy se ponderan con un filtro Gaussiano (σ=3.3s) centrado en el keypoint.

A continuación, las respuestas wavelet dx y dy se suman en cada subregión y forman un primer conjunto de entradas en el vector de características. Para obtener información sobre la polaridad de los cambios de intensidad, también extraemos la suma de los valores absolutos de las respuestas, | dx | y | dy |. Por lo tanto, cada sub-región tiene un vector descriptor cuadrimensional, **v** definido como:

$$v = (\sum d_x, \sum d_y, \sum |d_x|, \sum |d_y|)$$

Este resultará en un vecotr para todas las 4x4 sub-regiones de longitud 64. Dado que las respuestas Wavelet son invariantes a la iluminación, no debemos preocuparnos, y para el caso de la invarinza al contraste se logra convirtiendo el descriptor en un vector unitario.

BRISK

Para la formación del descriptor de rotación y escala normalizado [13.1], BRISK aplica un ruido Gaussino (σ) con el fin de evitar los efectos de aliasing, y rota cada punto ángulo α calculado como el arcotangente del gradiente dirección “y” y el gradiente dirección “x”. El vector descriptor binario dk es construido con las comparaciones entre los pares $(p_i^{\alpha}, p_j^{\alpha}) \in S$ en las que para cada bit b corresponde:

$$b = \begin{cases} 1, & I(p_j^{\alpha}, \sigma_j) > I(p_i^{\alpha}, \sigma_i) \\ 0, & para\ otros\ casos \end{cases} \qquad \forall (p_i^{\alpha}, p_j^{\alpha}) \in S$$

FREAK

FREAK (Fast Retina Keypoints) es un algoritmo basado en la composición de la retina para poder detectar los puntos de interés [15], y es que las células ganglion son las artífices en la detección del objeto en nuestros ojos. El detector FREAK pretende establecer una analogía entre las densidad de las células ganglion(ilustración 14) y el uso del ruido Gaussiano.

El algoritmo cambia el tamaño del ruido con respecto a un patrón en el que cada círculo se corresponde con un campo receptivo.

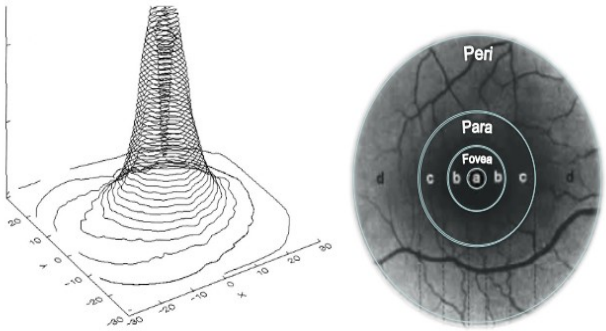


Ilustración 14: Densidad de las células ganglion de la retina

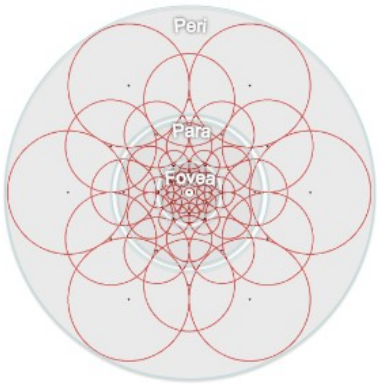


Ilustración 15: Ejemplo del patrón FREAK similar a las células ganglion de la retina. Cada circulo corresponde a un campo receptivo

Su descriptor binario, F se construye mediante el umbral de la diferencia entre pares de campos receptivos con su correspondiente ruido Gaussiano. En otras palabras, F es una cadena binaria formada por una secuencia de 1-bit Diferencia de Gaussianas(DoG).

$$F = \sum_{0 \le a \le N} 2^a T(P_a)$$

Donde P_a es una pareja de campos receptivos, N es el tamaño deseado para el descriptor y,

$$T(P_a) = \begin{cases} 1, & \left(I(P_a^{r_1}) - I(P_a^{r_2})\right) \\ 0, & para\, otros\, casos \end{cases}$$

con $I(P_a^{r1})$ es la intensidad suavizada el primer campo receptivo.

Con pocas docenas de campos receptivos, miles de pares son posibles conduciendo a un descriptor grande. Aunque, muchos de los pares pueden no ser útiles para describir eficientemente una imagen. Una estrategia posible sería la de seleccionar los pares dada su distancia espacial similar a BRISK. Sin embargo, éstos podrían estar altamente correlacionados y no ser discriminantes.

3.4.3 Features Matching

El principal objetivo de un feature matcher es el de encontrar las correspondencias entre una imagen u otra dado un vector de características (las características del objeto y del entorno donde se encuentra).

Para implementar estos features matchers he usado la librería suministrada por OpenCV denominada **Flann** (Fast library for Approximate Nearest Neighbors) que te ofrece una colección de algoritmos optimizados para encontrar el vecino más cercano en un conjunto de datos con un gran tamaño y dimensión [16].

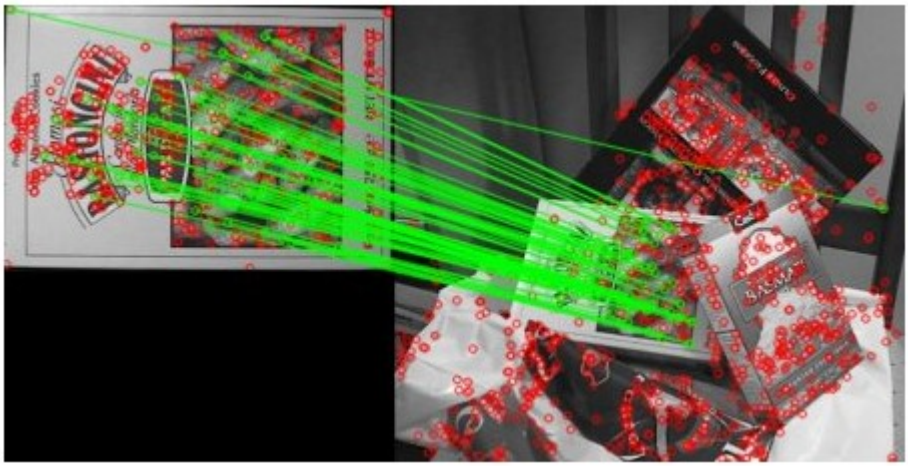


Ilustración 16: Ejemplo de Matcher

Los algoritmos que usa la librería son los siguientes:

1. **Árbol-kd aleatorio**

   El algoritmo original de kd-tree divide los datos por la mitad en cada nivel del árbol en la dimensión para la cual los datos presentan una mayor varianza. En comparación, los árboles al azar se construyen al al azar desde la primera dimensión D que posee unos datos con mayor varianza.

   En la implementación de FLANN utiliza el valor fijo D = 5, ya que funciona bastante bien en la mayoría de conjunto de datos.

   Al buscar en los árboles, se mantiene una cola de prioridad única en todos los árboles asignados al azar para que la búsqueda se pueda ordenar aumentando la distancia a cada límite del contenedor. El grado de aproximación se determina examinando un número fijo de nodos de hojas, momento en el que la inspección termina y los mejores candidatos regresan.

2. **Árbol k-medios jerárquico**

El árbol de k-medios jerárquico se construye dividiendo los puntos de datos en cada nivel en K regiones distintas utilizando un agrupamiento de k-medios, aplicándose así el mismo método recursivamente a todos los puntos en cada región. Pararemos la recursión cuando el número de puntos en una región sea menor que K.

A la hora de explorar el árbol, el algoritmo realiza inicialmente un solo recorrido a través de éste y agrega a una cola de prioridades todas las ramas inexploradas en cada nodo a lo largo del camino. A continuación, extrae de la cola de prioridad la rama que tenga el centro más cercano al punto de consulta y reinicia el recorrido del árbol desde esa rama. En cada recorrido el algoritmo sigue agregando a la cola de prioridades las ramas inexploradas.

El grado de aproximación se especifica de la misma manera que para los kd-árboles aleatorios, es decir, deteniendo la búsqueda después de que se haya examinado un número predeterminado de nodos de hojas.

3.5 Evaluación

Para poder comprobar el rendimiento de nuestros detectores al completo me he basado en los conceptos explicados en [17].

Para poder decidir si la decisión final es buena o no necesitamos del Ground Truth sobre el conjunto de las imágenes que usaremos y la medida de solapamiento entre las ventanas del Ground Truth y de las ventanas detectadas.

Ilustración 17: Detección GT(vgt): verde; Detección Detector(vdt): azul

La medida de solapamiento se define como el cociente entre la área de intersección y el área unión que forman las dos ventanas:

$$Solapamiento = \frac{A(vdt \cap vgt)}{A(vdt \cup vgt)}$$

Ilustración 19: Marcada área de intersección

Ilustración 18: Marcada área de unión

Tal y como está definido el solapamiento una detección perfecta tendría el valor de 1 y una detección errónea el valor de 0. Así pues nos vemos en la necesidad de decidir un cierto umbral en la medida de solapamiento para decidir si la detección es correcta o no.

Aquellas ventanas que superen el grado de solapamiento reciben el nombre de **Reales Positivos,** mientras que aquellas que no lo logran, **Falsos Positivos**. Cabe mencionar que si sobre una detección del Ground Truth se encuentran varias ventanas, la ventana con mayor solapamiento (si supera el umbral) será considerada como la única Real Positiva y las demás como Falsos Positivos penalizando así al detector.

Finalmente para evaluar el rendimiento se suele utilizar las siguientes definiciones:

- **Tasa de detección**: cociente entre el número de Reales Positivos y el número de Objetos del Ground Truth.

$$Tasa\,de\,detección = \frac{Reales\ Positivos}{n^o\ de\ objetos\ Ground\ Truth}$$

- **Tasa de error**: que se define como la uno menos Tasa de detección.

$$Tasa\,de\ Error = 1 - Tasa\,de\,Detección$$

- **Falsos Positivos por Imagen**: que se calcula como el cociente entre el nº de Falsos Positivos y número de imágenes utilizadas

$$FPPI = \frac{Falsos\ Positivos}{n^o\,de\,imágenes}$$

4. Implementación

4.1 Clases

Debido a que en multitud de nodos tenía que realizar las mismas funciones, decidí crear una serie de ficheros que las agrupase y me facilitase su uso en los diferentes códigos agrupando las funciones en base a su finalidad.

4.1.1 Clase features

Clase que recoge todas las funciones relacionadas con el paquete de features.

Funciones:

- Mostrar uso del nodo correspondiente por pantalla: showUsage
- Pintar en una única imagen todos los objetos que deseemos detectar: drawMultipleObject
- Calcular si la detección es válida: calcula_detectado
- Borrar una sección de una imagen (volver negro): borrar_seccion
- Calcular los límites de una caja contenedora tanto en los ejes x como y: limits
- Dibujar la caja contenedora con los inliers,outliers y el nombre del objeto: DrawBoundingBox
- Dibuja la caja contenedora con sus coordendas (x,y,z): drawBBXYZ
- Calcular los puntos de interés y el descriptor del objeto: detect_extract_object
- Calcular los puntos de interés y el descriptor del entorno: detect_extract_scene
- Devolver una imagen escalada: image_input
- Calcular el centro de la caja contenedora: centroUV
- Interpola linealmente el centro de la caja contenedora para la imagen de profundidad: escaladoCentroUV

4.1.2 Clase Timer

Clase que se utiliza con el propósito de conocer cuánto tiempo ha transcurrido desde una línea de código hasta otra.

Funciones:

- Establecemos el timer: constructor de la clase
- Calcula el tiempo transcurrido desde que se declaro la variable: intervalo
- Reseteo del timer: reset

4.1.3 Clase accessFileString

Clase que agrupa todas las funciones referente a la lectura/escritura de ficheros o de strings.

Funciones:

- Devuelve el nombre del archivo una vez introducido su ruta: readName
- Calcula el número de imágenes en un fichero .txt
- Devuelve la ruta de una imagen de un fichero en base al número introducido: getRuta
- Comprueba si existe en un fichero una línea con el nombre introducido: checkname
- Escribe en un fichero la detección calculada a continuación del nombre de la imagen: writeinfile
- Calcula el número de detecciones que posee un imagen en un fichero: numberofdeteccions
- Devuelve la detección correspondiente de la imagen en el fichero en base al índice introducido: extractDetecciones

4.1.4 Clase eval

Clase que agrupa las funciones relacionadas con la evaluación del detector.

Funciones:

- Calcula la área unión entre las dos ventanas: area_union
- Calcula la área intersección entra las dos ventanas: area_interseccion
- Calcula el grado de solapamiento entre las dos ventanas y devuelve un 1 lógico si supera el umbral introducido: solapamiento

4.1.5 Clase hogF

Clase que agrupa las funciones relacionadas con el paquete de hog.

Funciones:

- Guardamos los descriptores en un archivo: saveDescriptorVectorToFile
- Guardamos en un vector los archivos presentes en un directorio: getFilesInDirectory
- Calculamos las características de una imagen: calculateFeaturesFromInput
- Escribimos las características en un fichero: writefeaturesHOG

4.2 ROS

El principal problema que existe al usar ROS para captar las imágenes de los sensores es que éste publica la información en un formato distinto al que usa OpenCV. Por ello debemos utilizar una herramienta que haga la función de "puente" entre ROS y OpenCV.

Esta herramienta es conocida como **CvBridge** [18], la cual provee las librerías necesarias como para actuar como un puente.

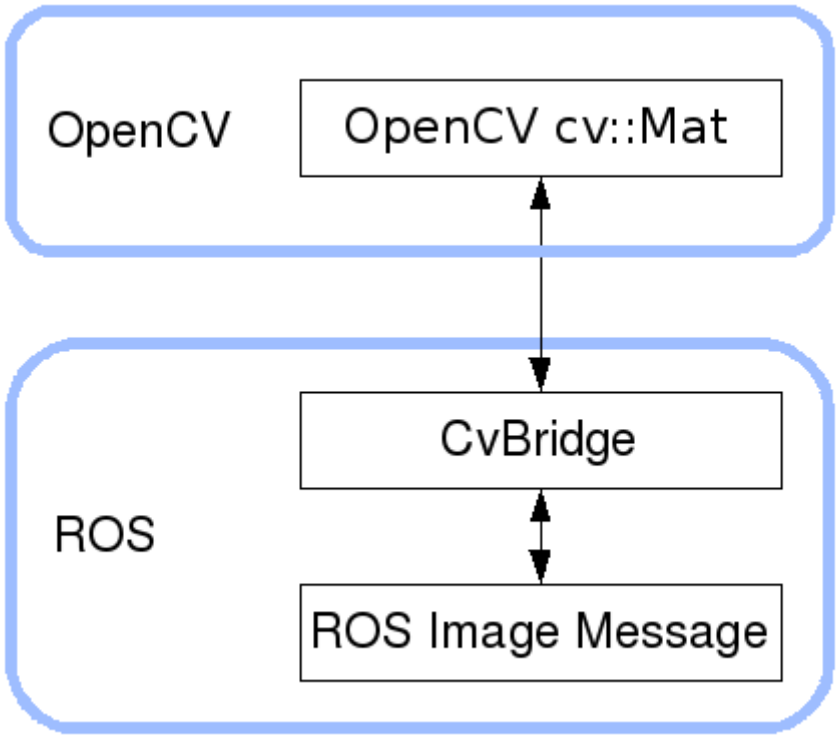


Ilustración 20: Gráfico de CvBridge

Para ello simplemente definimos la variable de cv_bridge y dentro de la función de llamada que se encuentra suscrita al topic de la imagen copiamos la información.

```
void imageCb(const sensor_msgs::ImageConstPtr& msg)
{
    cv_bridge::CvImagePtr cv_ptr;

    cv_ptr                  =                  cv_bridge::toCvCopy(msg,
sensor_msgs::image_encodings::BGR8);

}
```

Para que no haya problemas en la compilación deberíamos haber incluido las dependencias de *cv_bridge* e *image_transport* en nuestro CmakeLists.txt y en nuestro package.xml.

4.3 Kinect v2

Para poder usar la cámara Kinect v2 en el entorno de ROS he usado el paquete de iai_kinect2 [19] que te ofrece una serie de herramientas necesarias para recibir la información del sensor. Para ello el paquete a su vez posee los siguientes paquetes:

- Kinect2_bridge: Actúa como puente entre libfreenect2(driver para la kinect) y ROS. Al lanzarlo publica en una serie de topics la información del sensor.

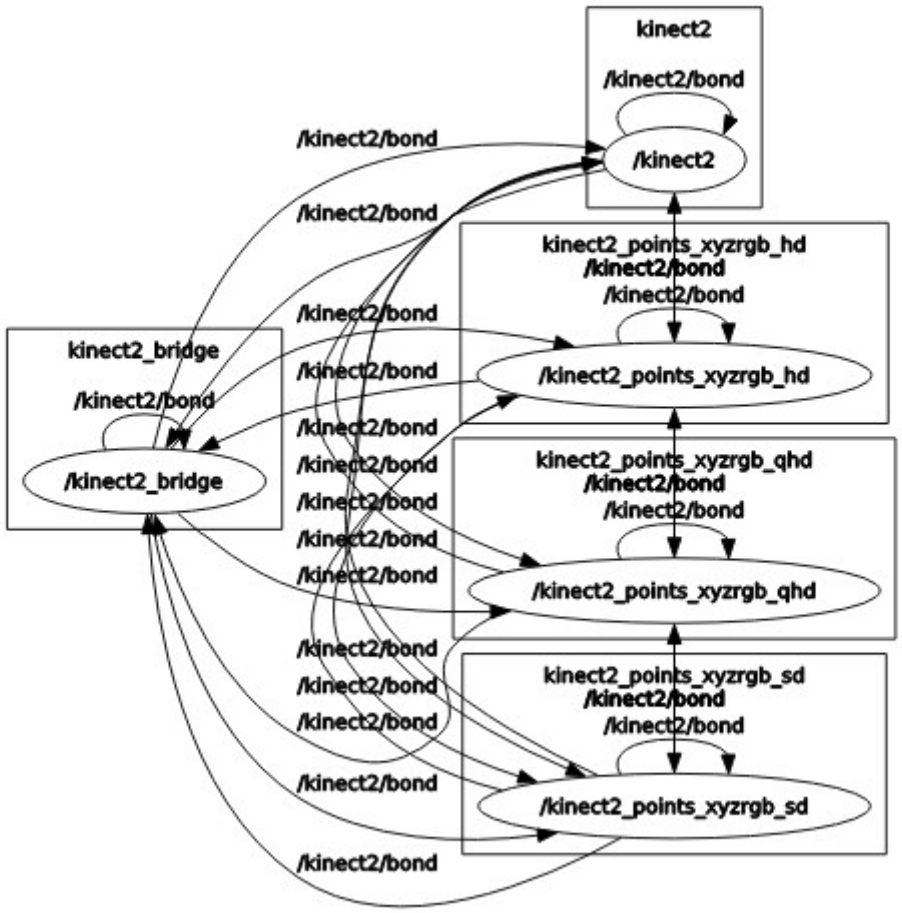


Ilustración 21: rqt_graph una vez lanzado el ".launch" de kinect2_bridge

Como podemos observar la información de las distintas configuraciones de la kinect (hd,qhd,sd) van a parar a kinect2 a través de /kinect2/bond que las transformará en un formato adecuado.

Para poder conocer los tópicos que transmiten esta información podemos utilizar el comando "`rostopic list`", ofreciendo una lista completa de todos los tópicos activos:

```
/kinect2/hd/camera_info
/kinect2/hd/image_color
/kinect2/hd/image_color/compressed
/kinect2/hd/image_color_rect
/kinect2/hd/image_color_rect/compressed
/kinect2/hd/image_depth_rect
/kinect2/hd/image_depth_rect/compressed
/kinect2/hd/image_mono
/kinect2/hd/image_mono/compressed
/kinect2/hd/image_mono_rect
/kinect2/hd/image_mono_rect/compressed
/kinect2/hd/points

/kinect2/qhd/camera_info
/kinect2/qhd/image_color
/kinect2/qhd/image_color/compressed
/kinect2/qhd/image_color_rect
/kinect2/qhd/image_color_rect/compressed
/kinect2/qhd/image_depth_rect
/kinect2/qhd/image_depth_rect/compressed
/kinect2/qhd/image_mono
/kinect2/qhd/image_mono/compressed
/kinect2/qhd/image_mono_rect
/kinect2/qhd/image_mono_rect/compressed
/kinect2/qhd/points

/kinect2/sd/camera_info
/kinect2/sd/image_color_rect
/kinect2/sd/image_color_rect/compressed
/kinect2/sd/image_depth
/kinect2/sd/image_depth/compressed
/kinect2/sd/image_depth_rect
/kinect2/sd/image_depth_rect/compressed
/kinect2/sd/image_ir
/kinect2/sd/image_ir/compressed
/kinect2/sd/image_ir_rect
/kinect2/sd/image_ir_rect/compressed
/kinect2/sd/points
```

- Kinect2_viewer: Este nodo se utiliza para visualizar el contenido de los topics publicados por kinect_bridge.

- Kinect2_registration: convierte la imagen de profundidad en una imagen de color.

- Kinect2_calibration: Como su propio nombre indica se utiliza para obtener los parámetros de calibración de la cámara.

4.4 HOG+SVM

En este paquete he implementado la teoría expuesta en el apartado de conceptos teóricos sobre HOG y SVM. Para ello debemos crear un modelo y luego utilizarlo para detectar el objeto deseado.

Para poder llevarlo a cabo he modificado el código de Anmol Sharma subido a la comunidad de github [20], añadiendo el código necesario para ubicar en el espacio al objeto y desglosando el código en dos nodos(uno para crear el modelo y otro para detectar el objeto).

Además de las librerias de OpenCV, cvBridge y el paquete iai_bridge, este paquete hace uso de la librería libre de SVMlight desarrollada por la Universidad de Dortmund [21] para crear el modelo.

La principal ventaja que posee este paquete con respecto al de features que veremos luego, es el de que el **tiempo de detección** de un objeto es independiente del número de veces que se encuentra repetido en la escena.

Lista de Nodos:

1. hog_train: Nodo encargado de crear el clasificador SVM y el Descriptor HOG.

   Uso:

   ```
   rosrun hog train argv[1] argv[2] argv[3]
   ```
   argv[1]-> ruta al directorio imágenes positivas
   argv[2]-> ruta al directorio imágenes negativas
   argv[3]-> ruta al directorio de para guardar el modelo

2. hog_deteccion: Nodo encargado de detectar el objeto deseado en tiempo real. Para ello carga el clasificador y descriptor previamente calculado, se suscribe al topic de la imagen de la kinect y publica las coordenadas en píxeles del centro de la Bounding Box para su posterior conversión en coordenadas xyz.

   Uso:
   Una vez lanzado `roslaunch features kinect2_coordenadas.launch`
   ```
   rosrun hog deteccion argv[1]
   ```
   argv[1]-> ruta al directorio de modelo guardado

3. hog_ruta: Nodo encargado de la publicación de resultados de la detección del objeto. Para ello debe cargar el clasificador y descriptor y la batería de imágenes que utilizaremos como pruebas. Éste deberá publicar los resultados en un fichero para su posterior evaluación.

   Uso:
   ```
   rosrun hog ruta argv[1] argv[2] argv[3]
   ```
   argv[1]-> ruta al fichero con la lista de imágenes.
   argv[2]-> ruta al fichero de los resultados.
   argv[3]-> ruta al directorio del modelo guardado

4.4.1 hog_train

Pseudocódigo:

1.Obtiene la lista de imágenes positivas y negativas de los directorios introducidos
2.En caso de que el número de imágenes positivas sea 0 → fin programa
3.Guardamos los vectores del descriptor en un fichero con un formato que pueda ser utilizado por la librería SVMlight
    3.1. Abrimos el fichero de la características
    3.2. Calculamos las características HOG de la imagen y lo guardamos en un vector
    3.3 Si el vector no está vacío
        3.3.1 Introduce los datos en el fichero
4. llamamos a las funciones de SVMlight y entrenamos SVM
5. Guardamos el modelo SVM en un fichero para su posterior carga en hog_deteccion
6. Obtenemos vector de características para usarlo con el algoritmo de HOG

4.4.2 hog_deteccion

Pseudocódigo:
Main:
0. Inicialización del nodo hog_deteccion
1.Ejecutamos una única vez:
    1.1 Inicialización de los parámetros del HOG
    1.2. Cargamos el modelo SVM
    1.3. Obtenemos el umbral de tolerancia de la detección del Detector
    1.4. Cargamos el HOG para su uso en *hog.multiscale* de la librería OpenCV
2. Llamamos a la clase ImageConverterHOG
3. ros::spin() para que se ejecute continuamente

La clase ImageConverter:

1. constructor:
    1.1. Se suscribe al topic de la cámara rgb() con la función de llamada: ImageCb
    1.2. Inicializa el publicador image_pub_ con el fin de publicar la imagen resultante en otro topic

2. ImageCbHOG
    2.1. Se convierte el mensaje del topic en el formato adecuado para OpenCV a través de cv_bridge
    2.2. Se cambia el frame a blanco y negro para los calculos
    2.3. Utilizamos la función “detectTest” para detectar el objeto introduciendo el hog cargado del main, el umbral inicializado en el main, la imagen en blanco y negro para los calculos y la imagen a color para pintar la bounding box.
    2.4 Una vez llamada la función se muestra la imagen a color resultante y se publica.

3. DetectTest
    3.1. Inicializamos los datos para la llamada detectMultiScale
    3.2. Llamamos a la función detectMultiScale para detectar el objeto
    3.3. llamamos a showDetection para que muestre la caja contenedora

4.showDetections
    4.1 filtramos las detecciones
    4.2 Obtenemos el centro del objeto
    4.3. Llamamos al servicio para obtener las coordenadas
    4.4. Si la llamada tiene éxito mostramos la caja contenedora más sus coordenadas, si no sólo la caja.

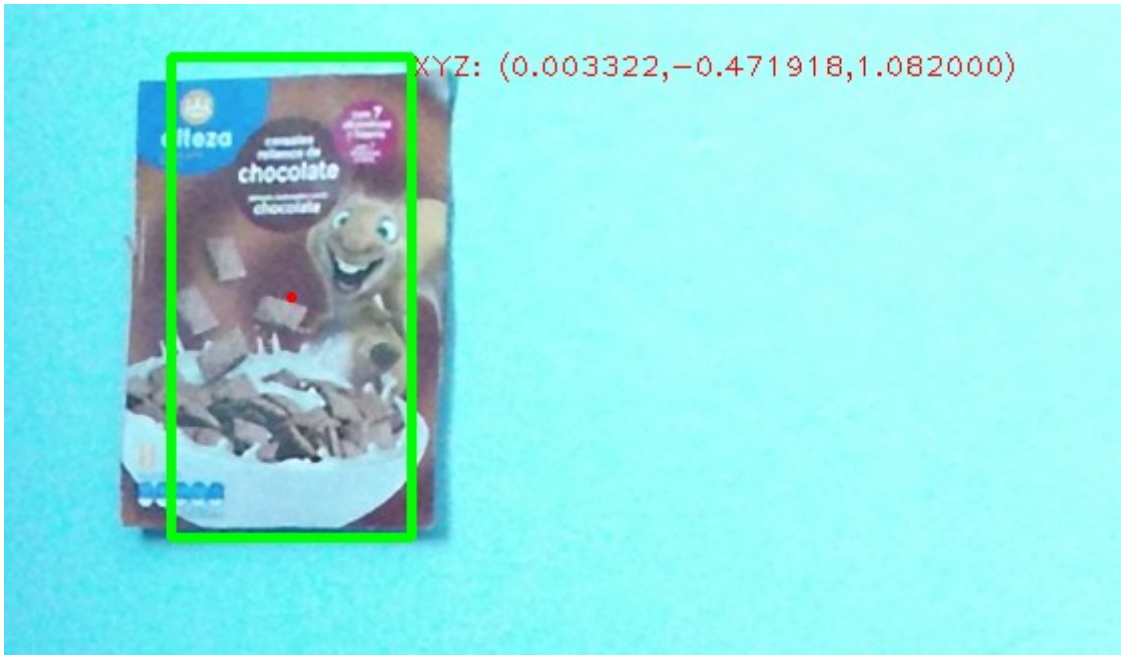


Ilustración 22: Detección HOG con sus coordenadas

4.4.3 hog_ruta

Posee el mismo esquema que hog_detections pero con algunas modificaciones, y es que en lugar de suscribirnos a la información de la cámara tenemos una batería de imágenes en las que debemos detectar el objeto.
Por ello, no utilizo la clase de ImageConverterHOG, todo el código se encuentra en el main.

Pseudocódigo:

1 Inicialización de los parámetros del HOG
2. Cargamos el modelo SVM
3. Obtenemos el umbral de tolerancia de la detección del Detector
4. Cargamos el HOG para su uso en *hog.multiscale* de la librería OpenCV
5. Obtenemos el numero de imágenes que vamos a utilizaremos
6. Bucle for en el que cada iteración pertenece a una imagen distinta
    6.1. Obtenemos la imagen correspondiente al número de iteraciones
    6.2. Convertimos la imagen a blanco y negro
    6.3. Llamamos a la función detectTest

6.4. Abrimos el fichero en el que vamos a volcar todos los resultados.
6.5. Escribimos el nombre de la ruta de la imagen en el fichero
6.6. Si ha habido alguna detección
6.7. Se publica los datos en el fichero con la función writeinfile
7. Una vez que se ha calculado las posibles detecciones para todas las imágenes, acaba el programa.

4.5 Features

Features es un paquete que implementa la teoría descrita en la parte de descriptores, extractores y flann.

Éste es una modificación del paquete existente de **find_object_2d** diseñado por Mathieu Labbé para ROS [22] en el que me ha basado para la detección del objeto.

Lista de nodos:

1. features_video: Nodo encargado de la detección del objeto deseado en la imagen suministrada por la cámara eligiendo el método deseado.

   Uso:
   Una vez lanzado el ".launch" de "kinect2_coordenadas.launch":

   ```
   rosrun features features_video argv[1] argv[2] argv[3] …
   ```

   argv[1] → Método del detector
   argv[2] → Método del extractor
   argv[3,4,...] → rutas a imágenes a detectar

2. features_video_cambio: Nodo encargado de la detección del objeto deseado en la imagen suministrada por la cámara con un método predefinido.

   Uso:
   Una vez lanzado el ".launch" de "kinect2_coordenadas.launch":

   ```
   rosrun features features_video_cambio argv[1] …
   ```

   argv[1,2,,...] → rutas a imágenes a detectar

3. features_evaluación: Nodo encargado de la publicación de resultados de la detección del objeto de una batería de de imágenes. Al igual que hog_deteccion éste también publicará las coordenadas en píxeles del centro de la Bounding Box.

   Uso:
   ```
   rosrun features features_evaluacion argv[1] argv[2] argv[3]
   ```

   argv[1] → ruta a la imagen para detectar
   argv[2] → ruta al fichero con la lista de imágenes del entorno
   argv[3] → ruta a la carpeta de resultados

4.5.1 Features_video.

Pseudocódigo:

Main:
0. Inicialización del nodo “features_video”
1. Ejecutar una única vez
    1.1. Comprobamos que la introducción de los argumentos es correcta (número de argumentos mayor que 4, y el argumento 1 y 2 mayor que 8 y 5 respectivamente)
    1.2. Pintamos en una única imagen todas las imágenes de los objetos
2. Creamos una clase de imageConverterFeature
3. ros::spin() para la ejecución continua

imageConverterFeature:
Constructor:
1. Nos suscribimos al topic de la imagen RGB de la cámara con la función de llamada imageCbFeature

imageCbFeature:

1. Tomamos un frame de la cámara y lo convertimos a blanco y negro para su procesamiento
2. Bucle **for** en el cada iteración pertenece a una imagen distinta de la línea de argumentos.
    2.1. Cargamos la imagen del objeto correspondiente
    2.2. Calculamos los puntos claves de la imagen del objeto y su descriptor dependiendo de que argumento hayamos introducido.
    2.3. Bucle **while** que termina cuando no se reconozca más el objeto en la imagen de la cámara(while(fin==0))
        2.3.1. Extraemos los puntos clave y descriptor de la cámara.
        2.3.2. Dependiendo del descriptor para el objeto que hayamos elegido, se introducirán distintas argumentos en flannIndex. Independientemente de cúal se utilice se calculará las distancias más cercanas entre los puntos clave de la cámara y del objeto

        2.3.3. Bucle for entre 0 y las filas del descriptor del objeto
            2.3.3.1. Si el índice del vecino más cercano del objeto y de la cámara es mayor que 0 y la distancia es menor que 0,8 guardamos el punto clave

        2.3.4. Si se ha más de 8 puntos que cumplan la condición anterior se calcula la matriz de homografía para hallar la Bounding Box
        2.3.5. Se calcula las esquinas del objeto y aplicando la transformación de la matriz de homografía las esquinas del objeto
        2.3.6 Se introduce las esquinas del objeto en una función para determinar si ha sido una detección válida(la diferencia entre sus diagonales sea mayor que 40 y que su diagonal sea mayor que 50 píxeles)
        2.3.7. Si la BB calculada es válida
            2.3.7.1. Se calcula el centro.
            2.3.7.2 Se escala el punto a la resolución de image_sd (512*424)
            2.3.7.3 Se solicita el servicio de “coordenadas_xyz” para obtener sus coordenadas.
            2.3.7.4 Si el servicio le responde, se dibuja la BB con las coordenadas si no tiene respuesta si dibuja sin ellas.

2.3.7.2. Se borra la sección de la imagen correspondiente a la bounding box para que se busque otra vez el objeto por si se encuentra repetido.

2.3.8 Si no se ha detectado nada fin = 1 para salir del bucle while

Ilustración 23: Ejemplo de drawMultipleObject

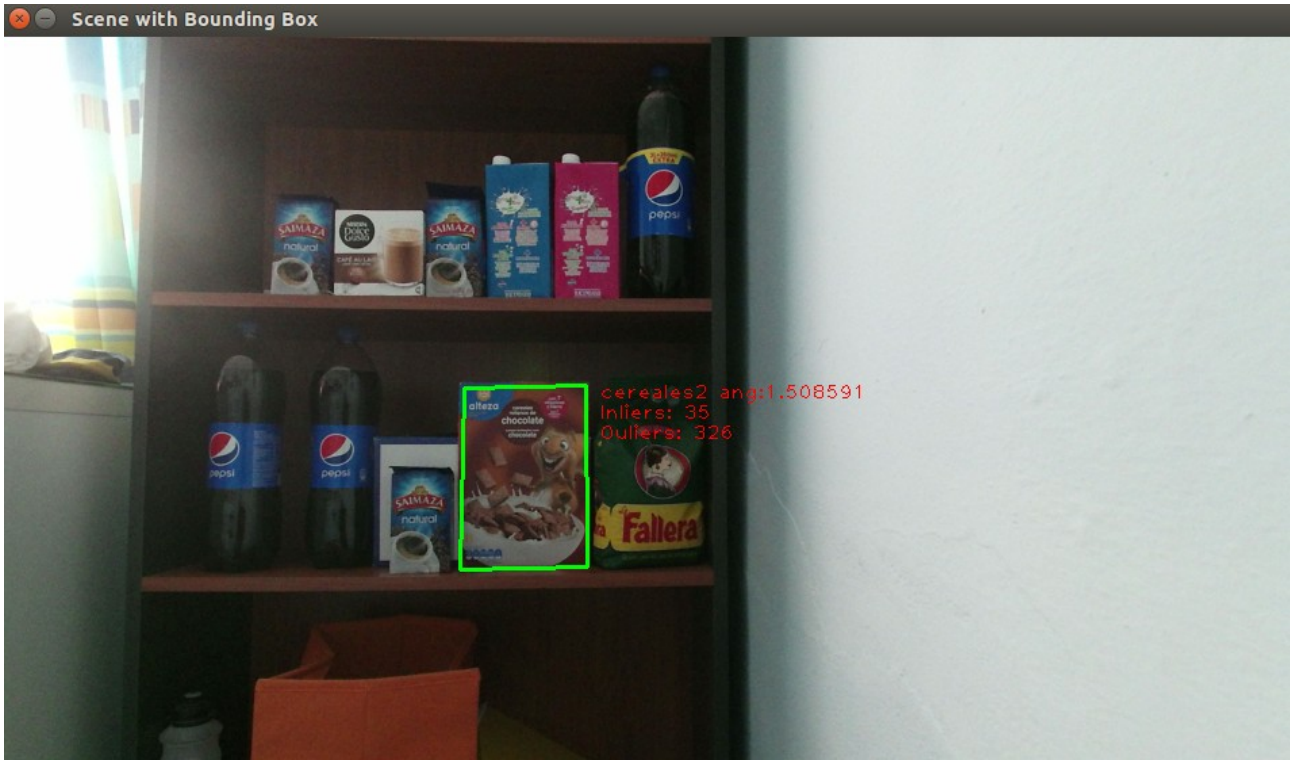


Ilustración 24: Ejemplo de detección cereales sin coordenadas

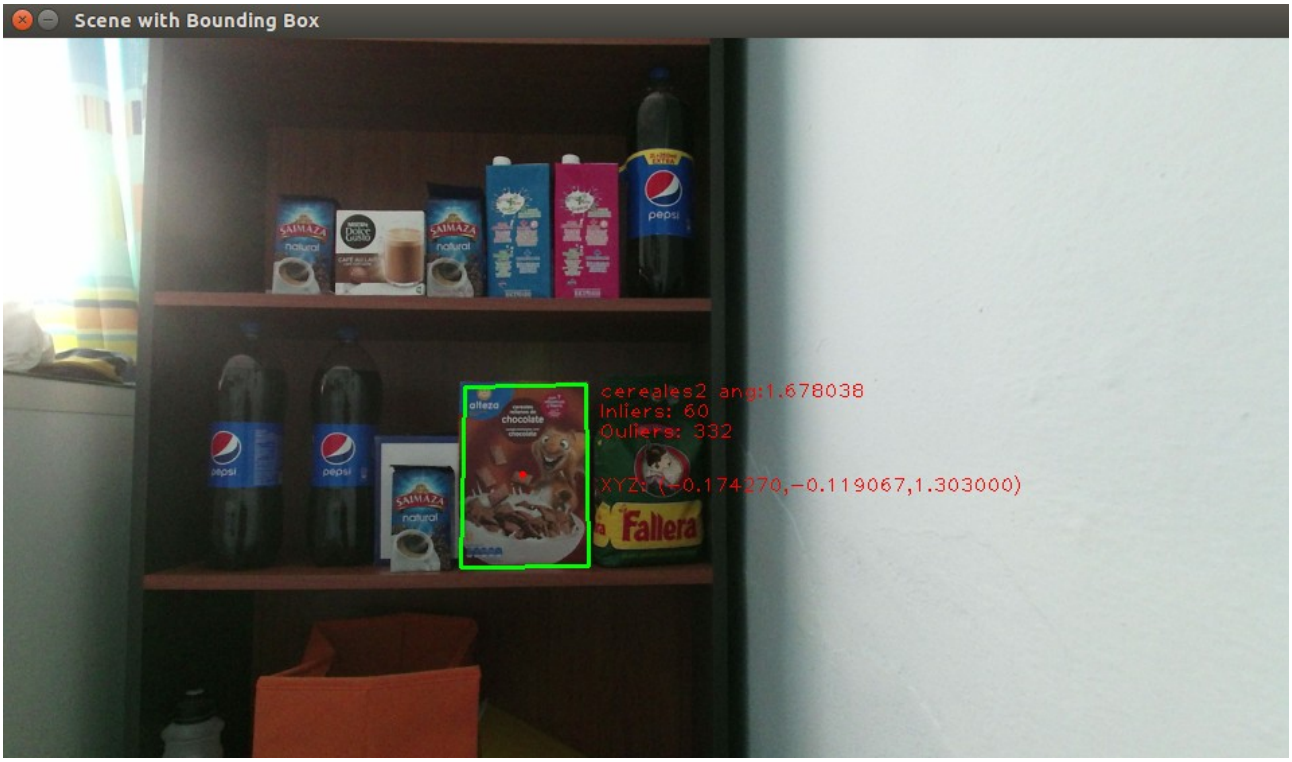


Ilustración 25: Ejemplo de detección cereales con coordenadas

4.5.2 features_video_cambio

Modificación del nodo “features_video” en el que inicialmente el método de detección/extracción es SIFT/SIFT y con forme nos acercamos al objeto y supera un cierto umbral cambiamos a SURF/SURF.

Este nodo está pensado para que se localice el objeto con el método más robusto y cuando nos encontremos más cerca que un cierto umbral utilicemos uno que posea menor tiempo de respuesta pero que nos ofrezca la misma calidad de detección a esas distancias.

Si cambiamos y pasa N ciclos de detecciones fallidas, reducimos el umbral en el que cambia para que cometa menos fallos. En el caso de que al reducirlo se encuentre el objeto a un distancia mayor que éste, volverá a SIFT/SIFT.

4.5.3 features_evaluación

Este nodo gran parte del código de “features_video” por lo que sólo voy a explicar en qué se diferencia. Dado que su finalidad es saber que tan bueno es la metodología de detector/extractor para localizar un objeto, este nodo cambiará entre todos los detector y extractores de la lista e intentará localizar el objeto en una lista de imágenes.

Al igual que features_video calculará los puntos claves, calculará las distancias entre puntos....

Además publicará sus resultados en un fichero con el nombre del detector y extractor utilizado y las detecciones en las imágenes.

4.6 Evaluación

Evaluación es un paquete que se utiliza como herramienta para facilitarte el trabajo en otros nodos con los ejecutables de toma_foto y recorte. Además como su propio nombre indica evalúa la calidad de los detectores con “comparacion” que a su vez requiere del uso de create_gt.

Lista de nodos:

1. toma_foto: Nodo encargado de tomar imágenes de la cámara cuando se presione la tecla del ordenador 's'. Se utilizarán estas imágenes para comprobar la eficacia de los algoritmos de reconocimiento.

   Uso:
   Lanzamos: `roslaunch kinect2_bridge kinect2_bridge.launch`
   A continuación: `rosrun eval toma_foto argv[1]`
   argv[1] → nombre de la carpeta que queremos crear
   Con la tecla 's' del teclado guardamos el frame

2. Create_gt: Nodo encargado de la creación del fichero que contiene la posición teórica de la posición del objeto en la imagen (Ground Truth) para su posterior comparación con los resultados obtenidos de los nodos hog_ruta y features_evaluacion.

   Uso:
   `rosrun eval toma_foto argv[1] argv[2]`
   argv[1] → ruta al fichero con la lista de imágenes
   argv[2] → ruta al fichero en el que vamos a guardar los resultados

   A través del botón izquierdo del ratón seleccionamos los dos vértices opuestos de la caja contenedora. Por otro lado si presionamos el botón derecho del ratón dibujamos encerramos el objeto dibujando 4 líneas generándose una caja contenedora que contenga al rectángulo formado.

   Para pasar a la siguiente imagen debemos pulsar la tecla 'n'.

3. Recorte: nodo encargado de recortar la imagen de forma manual para obtener la imagen que utilizaremos como plantilla para el paquete de features.

   Uso:
   `rosrun eval recorte argv[1]`
   argv[1] → ruta a la imagen que deseamos seleccionar una sección

   A través del botón izquierdo del ratón seleccionamos los dos vértices opuestos de la caja contenedora. Por otro lado si presionamos el botón derecho del ratón dibujamos encerramos el objeto dibujando 4 líneas generándose una caja contenedora que contenga al rectángulo formado.

   Una vez seleccionado la sección pulsamos 's' si deseamos guardar la parte seleccionada o 'n' si no estamos interesados.

4. Comparacion: Nodo encargado de comparar los resultados teóricos con los resultados prácticos calculando la Tasa de Detección, la Tasa de error y los Falsos Positivos por Imagen (FPPI).

   Uso:
   `rosrun eval comparacion argv[1] argv[2]`
   argv[1] → ruta al fichero con la lista de imágenes
   argv[2] → ruta al fichero con la lista de resultados del nodo “create_gt”
   argv[3] → ruta al fichero con la lista de resultados de las detecciones del detector completo

toma_foto

Pseudocódigo:
Main:
0. Inicialización del nodo toma_foto
1. Si el número de argumentos es menor que 2 salta un mensaje de uso y termina el programa
2. Ejecutar una vez
    2.1. obtenemos el listado de archivos en la carpeta de fotos del paquetes
    2.2. comprobamos si existe una carpeta con el nombre que hemos introducido en el primer argumento de la línea de comando
    2.3. Si no existe la carpeta
        2.3.1 creamos la carpeta e inicializamos a 0 la variable numSnapshot que será la encargada de nombrar los pantallazos junto con la extensión “.png”
    2.4. Si existe la carpeta
        2.4.1. obtenemos el numero de archivos para inicializar la variable numSnapshot con el número de archivos para no sobrescribirlos.
3. Llamamos a la clase ImageConverterTomaFoto.

ImageConverterTomaFoto:
1. Constructor
    1.1. Nos suscribimos al topic RGB de la cámara con la funcion de llamada ImageCbShot
2. ImageCbTomaFoto
    2.1. Inicializamos la variable key con 0 (tecla del ordenador)
    2.2. Obtenemos el valor de key con waitKey
    2.3. Si hemos presionado la tecla 's'
        2.3.1 Se guarda el frame con el nombre de numSnapshot más la extensión “.png” en la ruta correspondiente.
        2.3.2. Incrementamos numSnapShot.

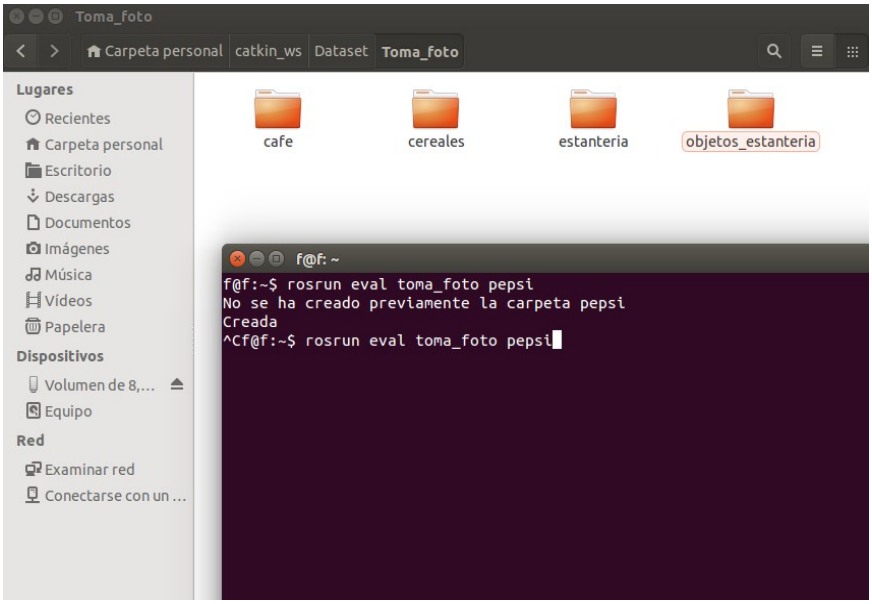


Ilustración 26: Queremos crear una carpeta no existente

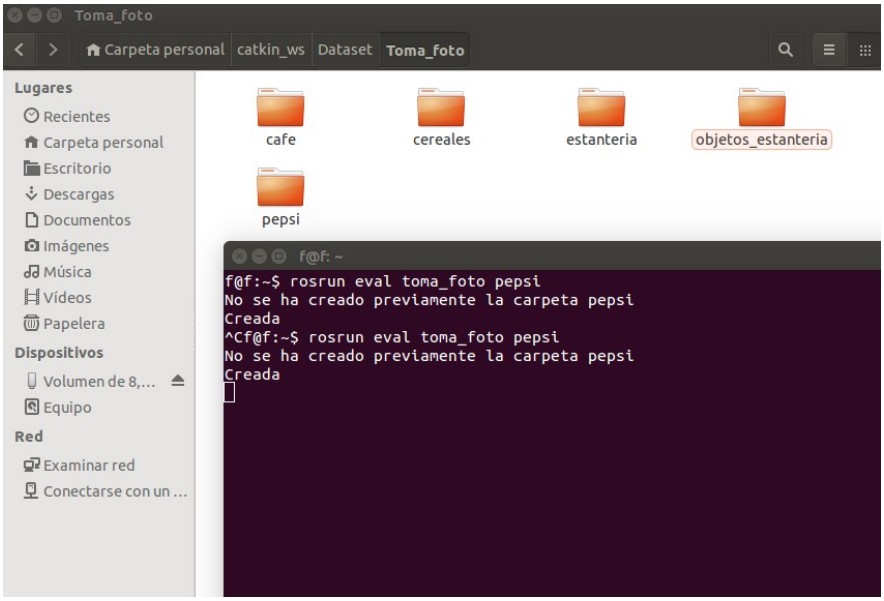


Ilustración 27: Una vez ejecutado, se crea la carpeta

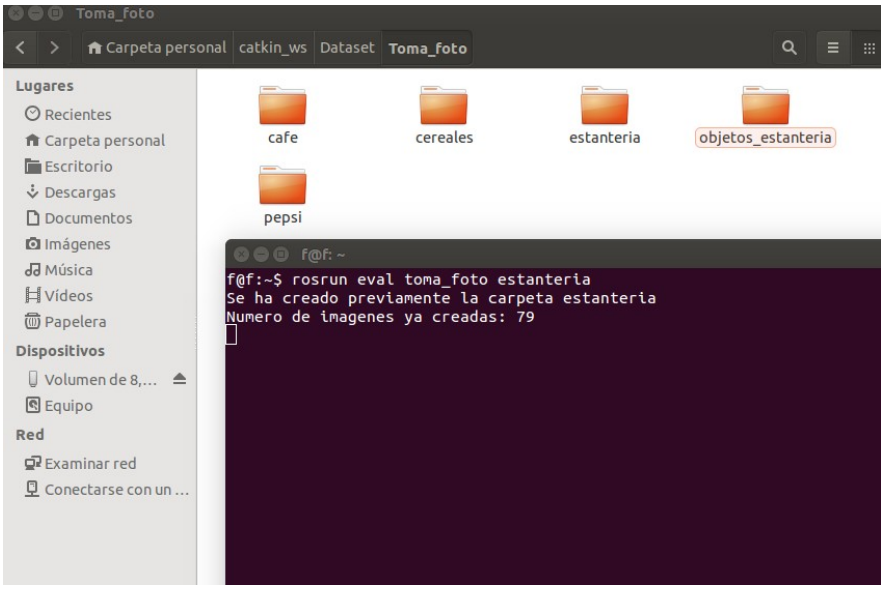


Ilustración 28: Si intentamos crear una carpeta existente

Create_gt

Pseudocódigo:
1. Si el número de argumentos es menor que 3 salta un mensaje de uso y termina el programa
2. Obtenemos el número de imágenes del archivo txt encargado de tener el listado de imagenes
3. Bucle for en el que cada iteración pertenece a una imagen distinta
    3.1 Obtenemos la ruta a la imagen correspondiente
    3.2. Cargamos la imagen a color
    3.3. inicializamos la variable key a '0' (tecla del ordenador)
    3.4. inicializamos la función de llamada para el evento del ratón (setMouseCallback)
    3.5. Si nunca se ha escrito la ruta de la imagen en el fichero de resultados se escribe su nombre
    3.6. Inicializamos el modo a “dibuja”. Este modo dibuja las posibles Bounding Box que se hayan guardado ya en el fichero de resultados.
    3.6. Bucle while de espera a que se presione la tecla 'n' para pasar a la siguiente foto

        3.6.1 Si el modo es “dibuja”
            3.6.1.1. Cambiamos el modo a “0” para evitar volver a entrar
            3.6.1.2 Obtenemos el número de detecciones escritas en en fichero
            3.6.1.3. Bucle for para extraer los datos del fichero y dibujar un rectángulo simbolizando la bounding box

        3.6.2. Si el modo es “rectangulo”
            3.6.2.1. Cambiamos el modo a “0” para evitar volver a entrar
            3.6.2.2. Dibujamos el rectángulo y escribimos los puntos opuestos de la Bounding Box en el fichero de resultados

        3.6.3. Si el modo es “line”
            3.6.3.1. Cambiamos el modo a “0” para evitar volver a entrar
            3.6.3.2 Dibujamos la línea de pt1 a pt2
            3.6.3.3 Introducimos el valor de pt2 en pt1 (para no tener que marcar los dos puntos de la línea) y guardamos el valor de pt2 en un vectores
            3.6.3.4 Una vez que se ha dibujado las 4 lineas, se busca la caja contenedora del rectángulo dibujado
            3.6.3.5 calculado los datos para dibujar la caja contenedora se escribe en elG fichero
    3.6.4 Mostramos la imagen y obtenemos el valor de key

Una vez pasadas todas las imágenes con 'n' de la lista se acaba el programa.

Cabe mencionar que el proceso de guardado de las BB tiene 2 modos debido a que si el objeto no se encuentra inclinado es mas cómodo enmarcarlo presionando los vértices opuestos, mientras que si se encuentra inclinado se puede cometer errores a la hora de seleccionarlos.

Para poder seleccionar el modo utilizamos la función de interrupción del ratón en el que el botón derecho selecciona el modo linea y el botón izquierdo el modo rectángulo.

Ilustración 29: modo rectángulo, presionamos los vértices opuestos

Ilustración 30: Modo línea: presionamos los 4 vértices

Recorte

Modificación del nodo de create_gt con el fin de recortar una parte de la imagen para utilizarlo en otros nodos.

El código es similar a create_gt sin la utilización de ficheros y se ha introducido la modificación en la que una vez construida la Bounding Box, independientemente del modo utilizado, se crea una imagen con la parte seleccionada y se guarda con el nombre de la imagen seleccionada mas "_recortada%d.png" siendo %d el número de recortes que se ha realizado a la imagen.

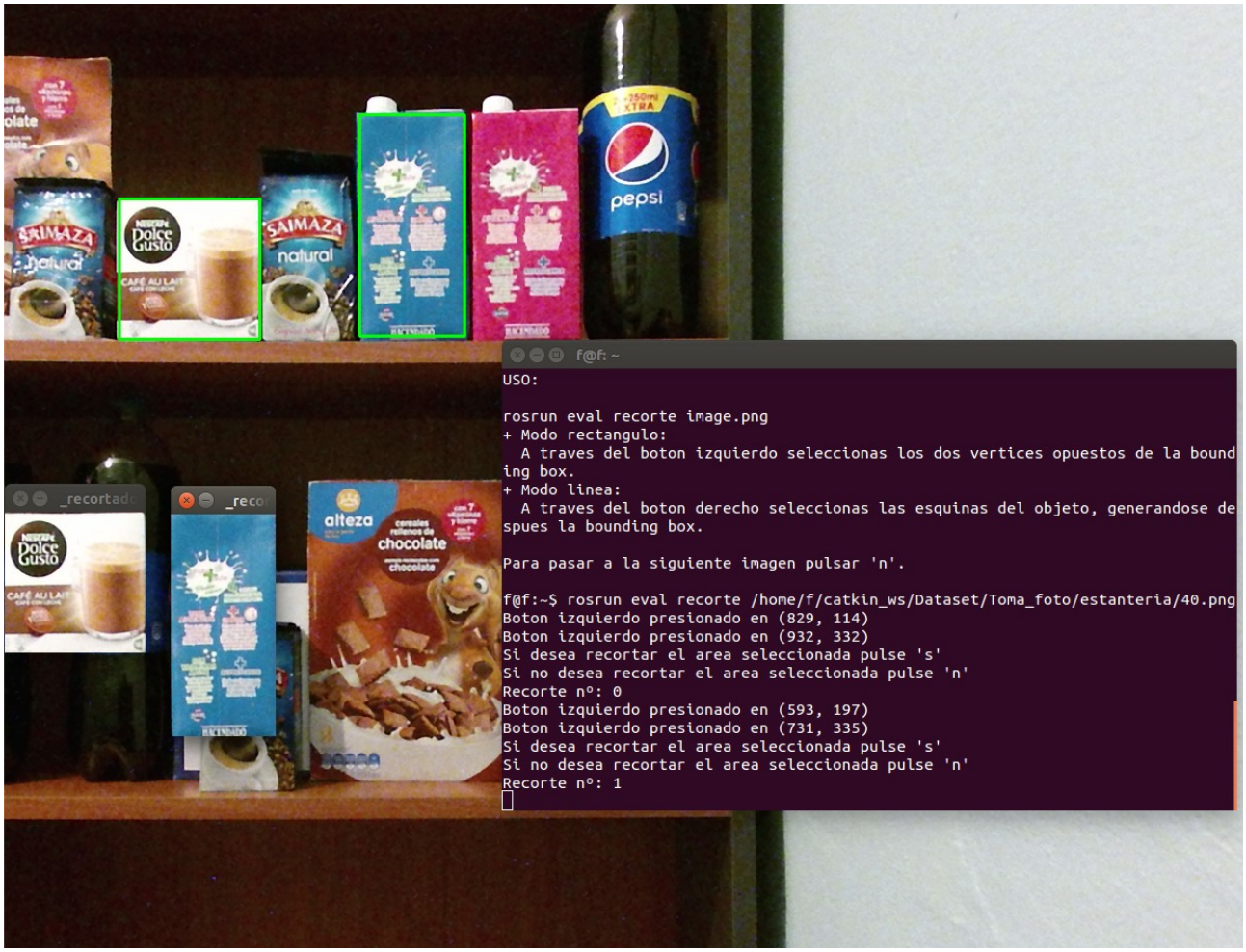


Ilustración 31: Ejemplo del uso de recorte

Comparacion

rosrun eval comparacion list_of_scene.txt list_of_gt.txt list_of_gd.txt

Pseudocódigo:
1. Comprobamos que el número de argumentos de la línea de comando es mayor que 3, si es menor mensaje de uso y salimos del programa.
2. Calculamos el número de imágenes.
3. Inicializamos a 0 los Reales Positivos (detecciones correctas del detector) y los Falsos positivos (detecciones erróneas del detector)
4. Bucle **for** en el que en cada iteración aumenta el umbral de solapamiento
    4.1. Bucle **for** en el que en cada iteración pertenece a una imagen distinta de la lista
        4.1.1 Se calcula el número de detecciones que se encuentra escrito en el fichero de list_of_gt.txt y list_of_gd.txt correspondiente a la imagen
        4.1.2. Incrementamos numero de objetos Ground Truth con las detecciones tericas de list_of_gt.txt(utilizaremos esta variable para calcular la tasa de Detección)
        4.1.3 Cogemos la ruta de la imagen y la modificamos con "_resultante.png" con el fin de guardar posteriormente la imagen con las cajas contenedoras tanto teóricas como las del detector
        4.1.4. Bucle **for** para las detecciones teóricas
            4.1.4.1 Extraemos la BB correspondiente y la dibujamos en la imagen con el color rojo
            4.1.4.2. Bucle **for** para las detecciones de nuestro detector
                4.1.4.2.1. Extraemos la BB correspondiente y la dibujamos en la imagen con el color verde.
                4.1.4.2.2. Calculamos la área unión entre las dos ventanas
                4.1.4.2.3. Calculamos la área intersección de las dos ventanas
                4.1.4.2.4. Si el nivel de solapamiento entre las dos ventanas para el umbral correspondiente es apto, nos salimos del bucle for
            4.1.4.3. Si alguna de las cajas de los detectores ha superado el nivel de solapamiento exigido se aumenta los RealesPositivos
            4.1.4.4. Si no supera se incrementa los FalsosPositivos
        4.1.5 Si el número de detecciones del detector supera el de la teoria se incrementa los FalsosPositivos en la diferencia entre el número de detecciones del detector y del Ground Truth.
        4.1.6 Se guarda la imagen resultante
    4.2 Se calcula los Falsos Positivos por Imagen, Tasa de Detección y la Tasa de error y lo guardamos en arrays.
5. Mediante un bucle **for** mostramos los resultados.

4.7 Coordenadas

Una vez reconocido el objeto debemos conocer dónde se encuentra en el espacio. Para ello he creado un paquete que realice esta función.

Es el único paquete en el que he implementado la función de un **servicio**. Debido a que el objetivo que persigo con este paquete es el de **recibir** el centro del objeto en coordenadas píxeles y devolver como **respuesta** dónde se encuentra en el espacio, y el modelo suscriptor/publicador no me ofrece esta alternativa.

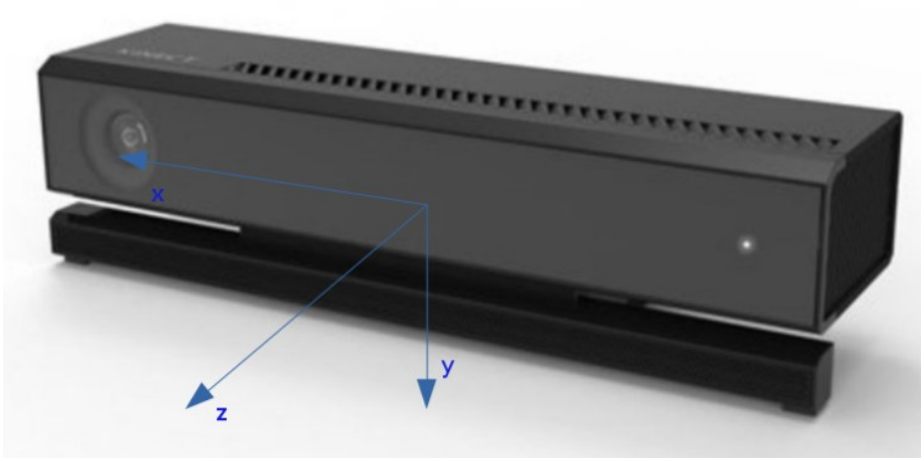


Ilustración 32: Eje de referencia kinect

Hay que tener en cuenta que las coordenadas que te devuelve la Kinect, se encuentran referidas a su sistema de referencia, el cual se encuentra representado en la ilustración.

Lista de nodos:

1. image_depth: nodo encargado de publicar las coordenadas xyz del objeto. Para ello se encontrará suscrito a la información de profundidad de la cámara Kinect y a las coordenadas del centro de la Bounding Box (u,v) en píxeles.

   Uso: una vez lanzado
   ```
   roslaunch kinect2_bridge kinect2_bridge.launch
   rosrun coordenadas image_depth
   ```

2. pcl: nodo encargado de publicar las coordenadas xyz del objeto. Para ello se encontrará suscrito a la información de profundidad de la cámara Kinect y a las coordenadas del centro de la Bounding Box (u,v) en píxeles.

Uso:

```
roslaunch kinect2_bridge kinect2_bridge.launch
rosrun coordenadas pcl
```

3. Envia: nodo encargado de enviar un valor (u,v) para comprobar el funcionamiento del nodo image_depth o pcl. Se ha utilizado a modo de depuración.

   Uso:

```
roslaunch kinect2_bridge kinect2_bridge.launch
rosrun coordenadas pcl
rosrun coordenadas image_depth
rosrun coordenadas envia argv[1] argv[2]
```

   argv[1] → u en coordenadas píxeles
   argv[2] → v en coordenadas píxeles

Funcionamiento:

Tanto el nodo pcl como image_depth están suscritos a un topic para conocer la información de la profundidad: pcl a una nube de puntos y image_depth a una imagen de profundidad. Éstos copian la información a una variable global y desde el servicio a través de unos cálculos obtienen las coordenadas.

El principal problema que tenía es que desconocía una forma eficiente de copiar la información del topic de PCL a la variable global. Mi opción era la de copiarla en un bucle for con número de iteraciones igual a la multiplicación de la resolución de la imagen suscrita. Esto provocaba un mayor coste computacional y por con siguiente un tiempo de respuesta mayor.

La solución a este problema fue la creación del nodo image_depth en el que sabía cómo copiar la información de forma eficiente.

pcl

Nos suscribimos a la información de la profundidad que viene dada por "/kinect2/(qhd,hd,sd)/points" el cual copiamos los tres canales a tres arrays x,y,z.

En el servicio obtenemos la componente asociada al punto recibido y le devolvemos en forma de array la respuesta con las coordenadas correspondientes.

image_depth

En el caso de image_depth, éste se encuentra suscrito a "/kinect2/sd/image_depth" el cual es una imagen de profundidad con formato 16UC con una resolución de 512x424, la codificamos como siempre con cvBridge y copiamos la variable en una matriz en la que en cada componente es la profundidad correspondiente a ese píxel y la mostramos en escala de grises junto con las coordenadas (u',v') correspondientes al punto que deseamos conocer su localización (ilustración 34).

En el servicio accedemos a la matriz para conocer la profundidad y a través de los parámetros intrínsecos de la cámara obtenemos las coordenadas “x” e “y” tal y como se describe en [23]:

```
P3D.x = (x_d - cx_d) * depth(x_d,y_d) / fx_d
P3D.y = (y_d - cy_d) * depth(x_d,y_d) / fy_d
P3D.z = depth(x_d,y_d)
```

Una vez hallado las coordenadas cartesianas, las devolvemos en forma de array.

envio

Es un nodo simple en el envío un punto a los servicios de pcl e image_depth, recibo la respuesta y mido su tiempo.
Obteniéndose un tiempo de respuesta para el nodo image_depth de unos 1.2 milisegundos y para el nodo pcl de unos 62 milisegundos.

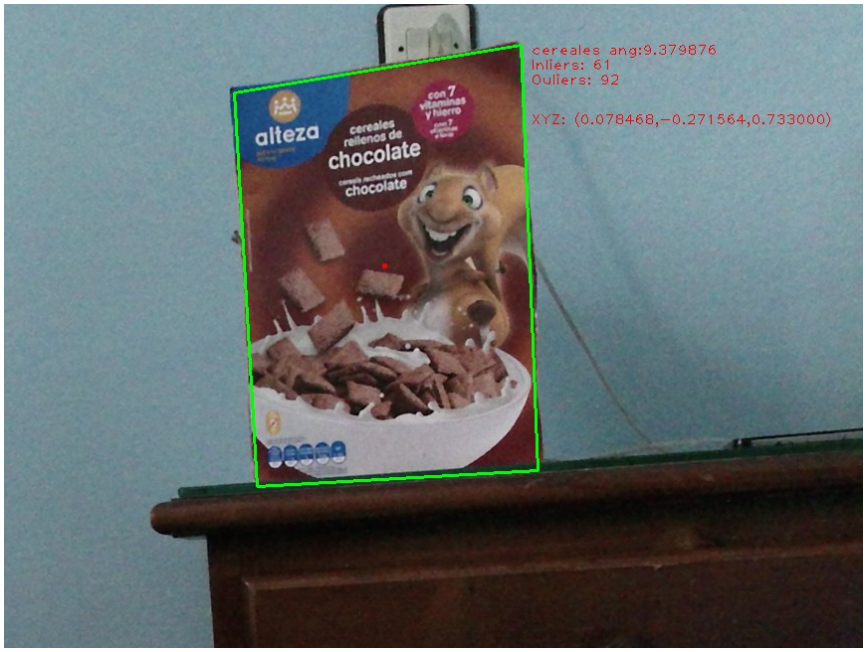


Ilustración 33: Cereales detectado con coordenadas (u,v), envía las coordenadas escaladas (u',v') al servicio (punto rojo)

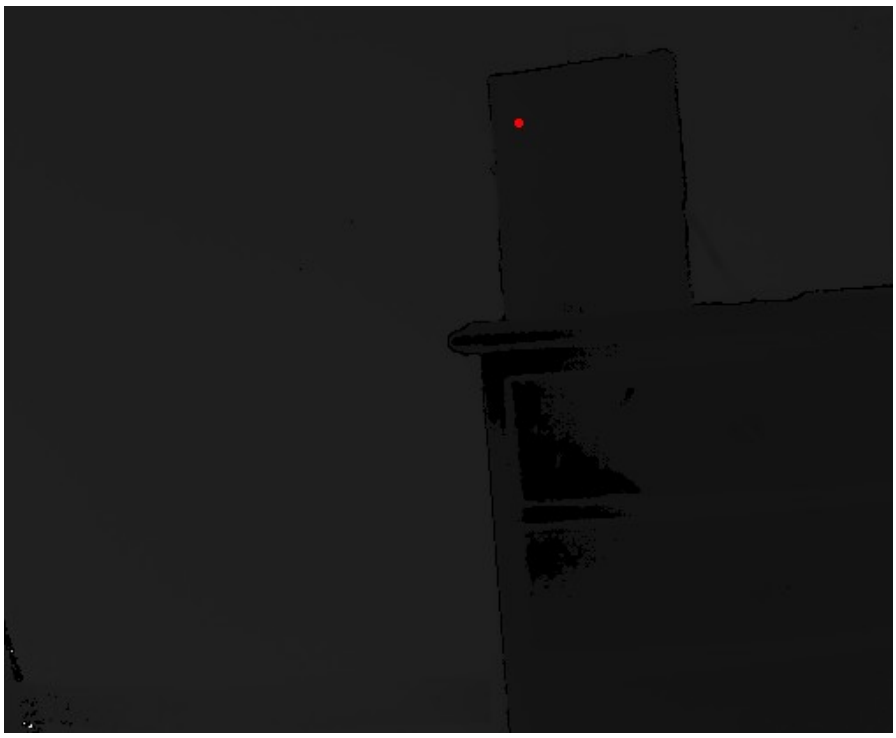


Ilustración 34: Equivalencia de la coordenada (u,v) a la imagen de profundidad coordenada (u',v'')

Como podemos comprobar en las ilustraciones 33 y 34 al interpolar el punto con coordendas (u,v) de la resolución HD (1920x1080) a la imagen de profundidad(515x424) (u',v') éste se encuentra desplazado.

Esto conlleva a que si el objeto se encuentra lo suficientemente lejos de la cámara, el punto puede caer fuera del objeto provocando unas mediciones erróneas. Para poder evitar esto en primer lugar escalé la imagen de entrada a la resolución de la imagen de profundidad provocando que el punto (u,v) se corresponda con el punto (u',v'), no haría falta escalar los puntos.

Sin embargo, esta solución no es tan aconsejable dado que se produciría una pérdida de información y los algoritmos de detección del objeto funcionarían peor. En el caso del detector/extractor con mayor distancia de detección, SIFT/SIFT (como veremos más adelante) pasa de detectar el objeto a más de 2 metros a 1,2 metros.

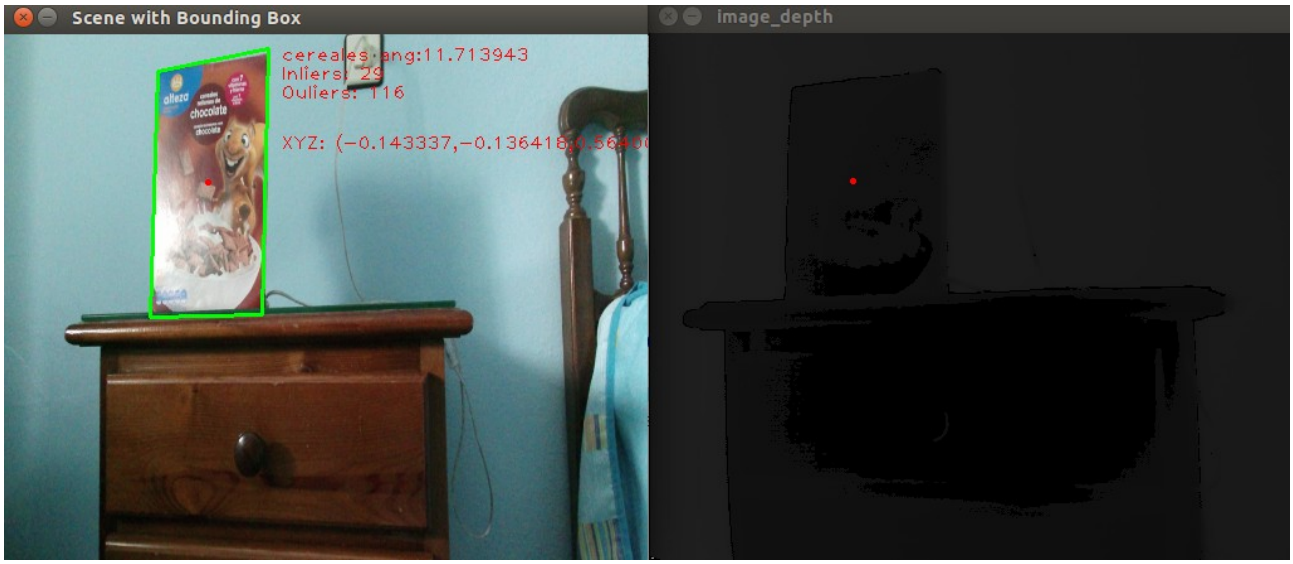


Ilustración 35: Imagen hd escalada e imagen de profundidad correspondiente

Además como podemos ver en la ilustración 35 no basta con sólo escalar la imagen para que el píxel se corresponda con la imagen de profundidad, porque el sensor de profundidad posee un rango diferente al rango de visión: rango menor en el eje horizontal y un rango mayor en el eje vertical. Esto se puede apreciar en que en la imagen a color muestra una cama y en la imagen de profundidad no se aprecia.

Para poder solucionarlo se utilizaron las siguientes medidas:

1. Nos suscribimos al topic de "/kinect2/hd/image_depth_rect" para no tener que reescalar el punto, pero tras un intento de implementación los tiempos de ejecución se dispararon tanto de la respuesta del servicio como en el algoritmo de detección porque el procesador debía ejecutar más cálculos en el nodo de image_depth. Por lo que no se implementó esta solución.

2. No escalar la imagen de entrada de la cámara y cuando se acerque al objeto, que aumente las posibilidades de que el punto caiga en el objeto. Sin embargo, esta solución generaría una gran incertidumbre porque no sabemos a ciencia exacta si el punto está en el objeto.

3. Limitar los rangos tantos de la visión como de la imagen de profundidad: Cuando se obtiene las coordenadas (u,v) se interpola para las dimensiones de la imagen de profundidad 512x424, se comprueba si u' se encuentra en el rango en el que el sensor de profundidad tiene información y si es así, se vuelve a interpolar para adecuarlo al tamaño de la imagen de profundidad u". Para el caso de v dado que el sensor de profundidad es el que tiene un mayor rango, debemos limitar el rango en el eje vertical de éste y escalarlo a las dimensiones originales. Si interpolamos cada punto de la imagen de entrada y modificamos la imagen de profundidad obtendríamos la siguiente ilustración:

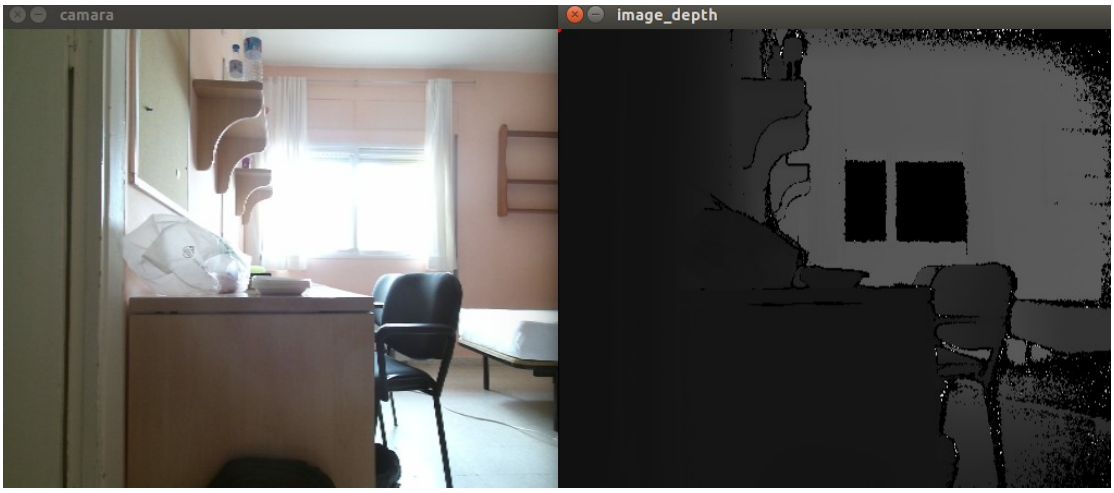


Ilustración 36: Rangos de visión y profundidad adecuados para que coincidan

Como podemos los píxeles de la imagen de color se encuentran en la misma posición que los puntos de la imagen de profundidad.

En esta ocasión si ejecutamos los algoritmos de detección podemos ver como el punto cae en el objeto:

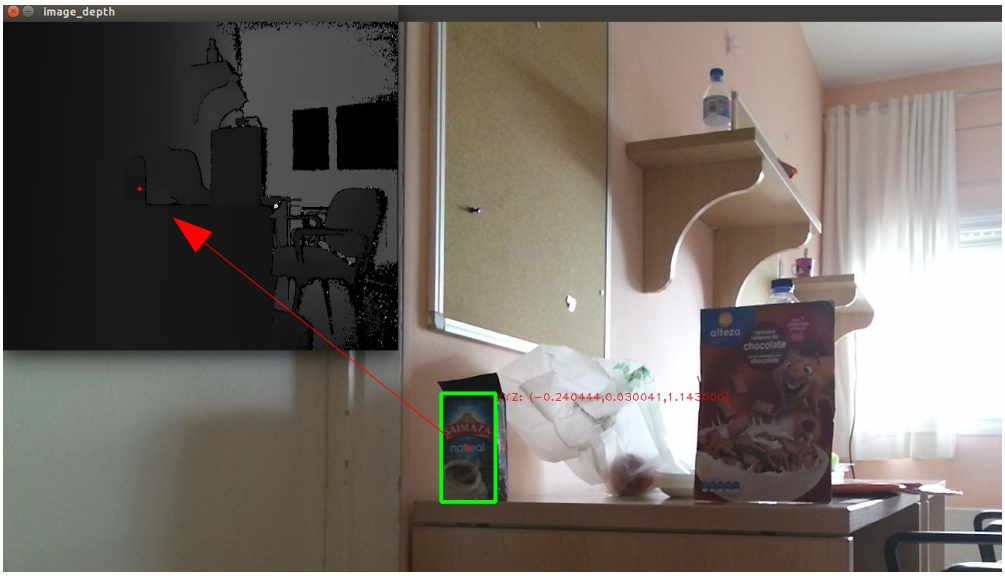


Ilustración 37: Correspondencia del punto (u,v) con el punto (u'',v'')

He de decir que el punto se encuentra ligeramente desviado debido a que las limitaciones en el rango no son perfectas pero ofrece unos mejores resultados que si no escalamos los rangos.

5. Pruebas y resultados

5.1 Tiempos de ejecución

Para medir los tiempos de ejecución de cada detector/extractor en “features_video” he utilizado un fondo lo más simple posible y he calculado el tiempo promedio en detectar el objeto.

Cabe decir que estos tiempos no son absolutos, es decir puede depender del estado de carga computacional que presente en ese momento el computador, el fondo que se utilice,... Sin embargo, es una buena estimación de cuánto tardaría cada algoritmo **por cada imagen que reconozca**.

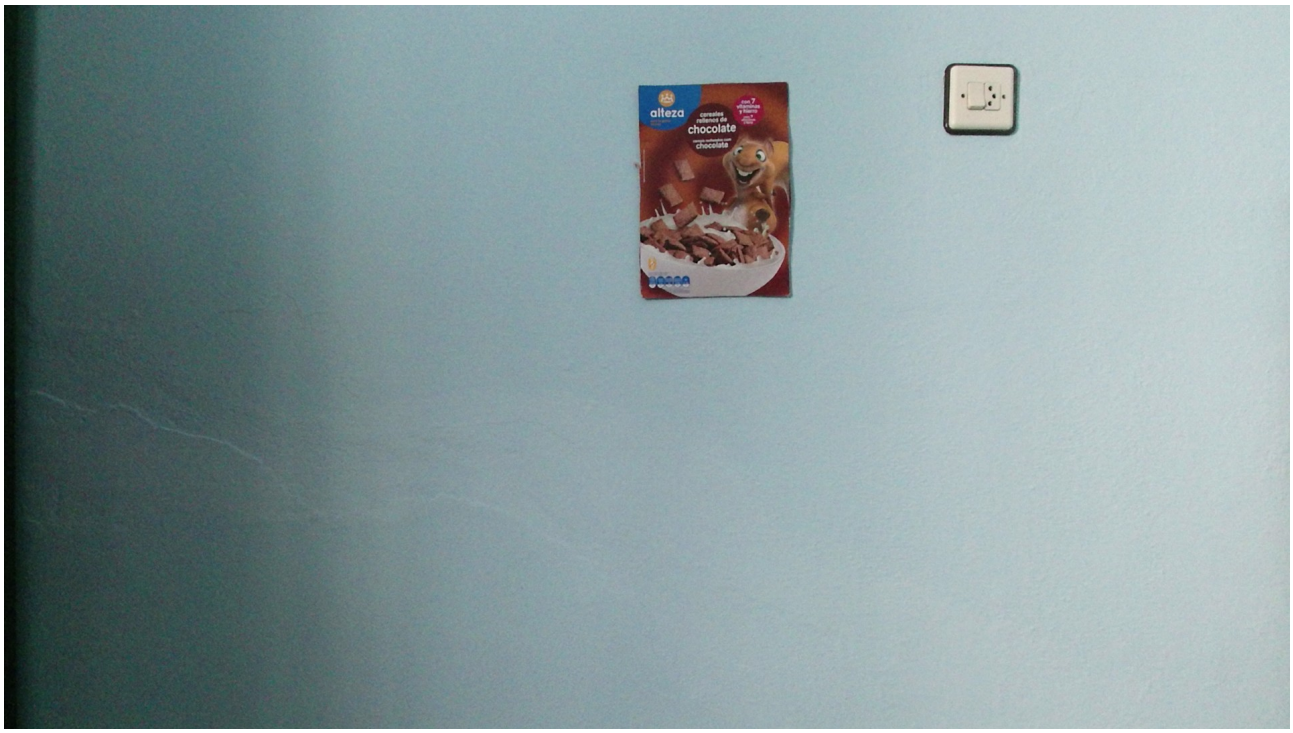


Ilustración 38: Fondo para medir los tiempos

Extractor\ detector	Dense	Fast	GFTT	MSER	ORB	SIFT	Star	SURF	BRISK
BRIEF	2,5s	0,27s	0,4s	0,6s	0,15s	1,7s	-	0,5s	1,7s
ORB	3s	0,26s	0,4s	0,5s	0,25s	-	0,3s	0,7s	1,7s
SIFT	-	1s	0,7s	1,3s	-	3s	0,7s	1,6s	2,2s
SURF	-	0,36s	-	0,6s	0,35s	1,7s	0,2s	0,7s	1,7s
BRISK	5s	1,5s	2s	1,9s	1,5s	3s	1,1s	2s	3s
FREAK	5s	0,5s	0,7s	0,7s	-	1,8s	0,4s	0,6s	1,9s

Tabla 1: Tiempos de detección para la caja de cereales

Las casillas marcadas con un "-" muestran aquellos detectores/extractores que no son capaces de identificar correctamente el objeto.

HOG

El algoritmo de detección de HOG posee un tiempo de ejecución de unos 800ms pero a diferencia de features este tiempo es **independiente** de la cantidad de objetos que se encuentran en la imagen.

5.2 Ángulo límite

Finalmente para poder medir los ángulos límite utilicé el método que proporciona el mejor resultado a costa de un mayor tiempo de ejecución: detector **SIFT**, extractor **SIFT**. Coloqué un objeto a una distancia de 1 metro (distancia estándar) y calculé el ángulo límite al que deja de detectar el objeto.

El resultado fue que podía girar el objeto hasta unos 55-60 grados y el objeto seguía siendo reconocido. En la mayoría de los extractores/detectores pueden llegar hasta unos 40 grados sin problemas.

Para el caso de la detección a través de **HOG** su ángulo límite se encuentra entre unos 32-37 grados.

5.3 Resultados de las pruebas de rendimiento

Para poder evaluar el rendimiento de los detectores al completo he usado una batería de imágenes en las que he cambiado de posición el objeto que deseamos localizar, su distancia hacia el sensor,he cambiado la cantidad y la incidencia de la luz sobre una estantería simulando un estante de un supermercado o almacén.

Los parámetros que voy a calcular son: la Tasa de Detección, Tasa de error, FPPI y la distancia límite (ésta será crucial para luego interpretar los resultados).

Para poder medir la distancia límite he utilizado un fondo simple con el objeto y he alejado la cámara hasta que la detección dejaba de ser estable.

Para medir los parámetros de evaluación he seguido los siguientes pasos:

1. Creamos la batería de imágenes con el nodo toma_foto: variando distancias, posiciones, ángulos, luminosidad...
2. Creamos el fichero .txt con las ventanas del Ground Truth mediante el nodo create_gt
3. Para el caso de HOG:
    1. Nos creamos el modelo:
        1. Tomamos unas más de 50 fotos del objeto con diferentes ángulos y luminosidad.
        2. Recortamos la imagen seleccionando sólo el objeto con el nodo recorte.
        3. Entrenamos al modelo con el nodo train.
    2. A través del nodo ruta del paquete de hog creamos el fichero .txt de las detecciones de nuestro algoritmo
4. Para el caso de Features:
    1. Nos creamos la plantilla
        1. Tomamos sólo una imagen del objeto y la recortamos con recorte
    2. A través del nodo features_evaluacion obtenemos todos los ficheros .txt de las detecciones de los detectores/extractores correspondientes

5.3.1 Experimento I

En este primer experimento voy a utilizar la misma caja de cereales que he estado utilizando a lo largo del Trabajo Fin de grado. con unas dimensiones de 19,2x27,2 cm.

En el caso de HOG he utilizado **64 imágenes** para crearme el modelo.

5.3.1.1 Distancia límite

Extractor\ detector	Dense	Fast	GFTT	MSER	ORB	SIFT	Star	SURF	BRISK
BRIEF	1,10m	0,9m	1,25m	0,8m	1m	1,1m	X	1,2m	1,15m
ORB	1m	1,2m	0,9m	0,8m	2m	X	1m	1,2m	1m
SIFT	X	1,5m	1m	1,3m	1,8m	2,8m	2m	1,9m	1,1m
SURF	X	0,9m	X	1,9m	1,7m	1m	0,8m	2,5m	1,1m
BRISK	1,3m	1,2m	0,9m	2,4m	1,5m	1,7m	1,4m	2,4m	1,1m
FREAK	0,9m	1,15m	0,5m	1,2m	X	1,2m	1m	1,33m	1,2m

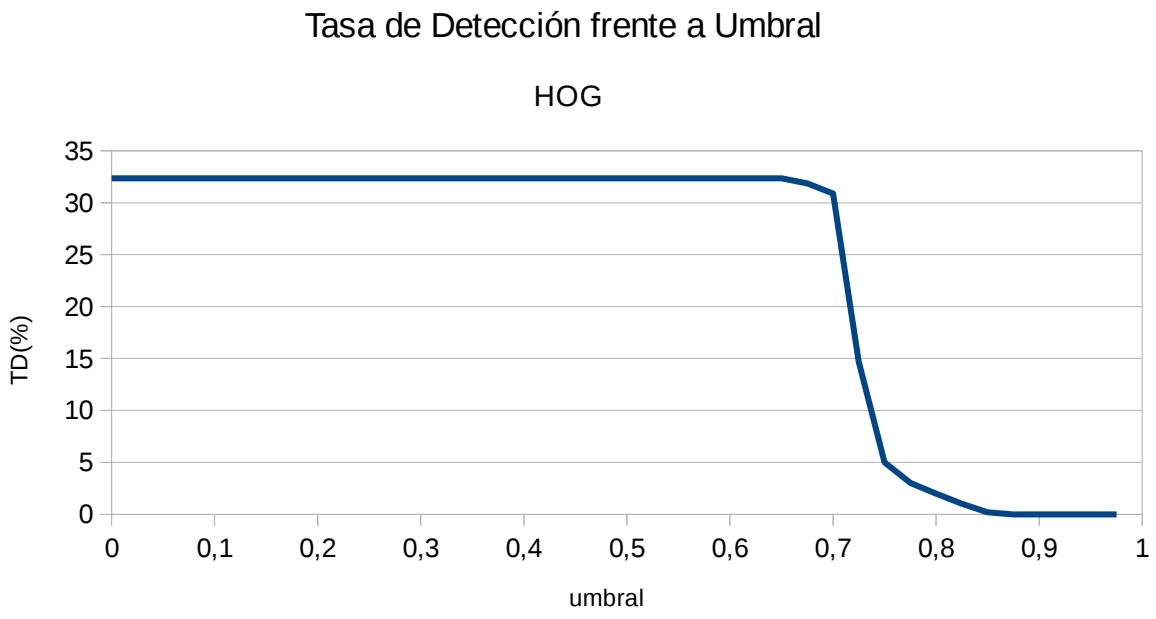
Tabla 2: Experimento I: Distancia límite

HOG

Para el caso de HOG el objeto podía alejarse hasta unos 2,30 metros.

5.3.1.2 Tasa de Detección

HOG



Como podemos apreciar la Tasa de detección se mantiene constante (32,35) hasta un grado de solapamiento del 65% , momento en el que empieza a decaer.

Para el caso de **HOG** el 70 % equivale a una Tasa de Detección del 30,88 %

Features

Como en el caso de features hay tantos tipos de detector/extractor he decidido mostrar sólo el porcentaje que define una buena detección: 70 %.Como podemos comprobar en el caso de HOG en el que se ha representado todo la TD frente al umbral antes del ~60% siempre se mantiene constante y a partir del 70 comienza a decaer.

Debido a que es muy complicado que coincida en una gran medida (+80 %) la ventana del Ground Truth creada manualmente con la detección del objeto.

Extractor\ detector	Dense	Fast	GFTT	MSER	ORB	SIFT	Star	SURF	BRISK
BRIEF	33,82	73,03	60,7	20,09	12,7	63,725	-	64,7	39,21
ORB	34,80	71,5686	44,60	15,19	23,03	-	5,88	55,88	9,8
SIFT	-	74,0196	61,76	44,60	1,47	78,57	28,43	51,9	16,66
SURF	-	4	-	33,33	21,568	11,57	0,5	64,7	2,4
BRISK	48,52	71,07	58,33	53,43	14,7	70,9	7,35	72,54	22,05
FREAK	32,84	70,58	56,86	11,76	-	62,25	2,4	46,07	25,49

Tabla 3: Experimento I: TD

Como podemos comprobar existen algunos métodos que no logran detectar el objeto.

5.3.1.3 Tasa de error

HOG
Para el caso del nodo HOG la tasa de error es del 69.12%

Features

Extractor\ detector	Dense	Fast	GFTT	MSER	ORB	SIFT	Star	SURF	BRISK
BRIEF	66,18	26,97	39,3	79,91	87,3	36,275	100	35,3	60,79
ORB	65,2	28,4314	55,4	84,81	76,97	100	94,12	44,12	90,2
SIFT	100	25,98	38,24	55,4	98,53	21,43	71,57	48,1	83,34
SURF	100	96	100	66,67	78,432	88,43	99,5	35,3	97,6
BRISK	51,48	28,93	41,67	46,57	85,3	85,3	92,65	27,46	77,95
FREAK	67,16	29,42	43,14	43,14	100	100	97,6	53,93	74,51

Tabla 4: Experimento I: TE

5.3.1.4 FPPI

Debido a las filtraciones de las detecciones en el caso de HOG y en el uso del criterio en features, los el número de falsos positivos resulta despreciable frente al número de detecciones correctas, en ambos métodos no superan los 0,1 falsos positivos por imagen.

5.3.2 Experimento *II*

Para comprobar la eficiencia de los algoritmos he decidido ponerlo a prueba con un objeto más pequeño y recubierto de un material que refleja más la luz. El objeto elegido es bolsa de café de la marca Saimaza con una dimensiones de 7.9x14.4 cm.

Ilustración 39: Objeto utilizado en el Experimento II

5.3.2.1. Distancia límite

Features

Extractor\ detector	Dense	Fast	GFTT	MSER	ORB	SIFT	Star	SURF	BRISK
BRIEF	1,1m	1,25	1,2	-	-	1,2	<0,5	1	<0,5
ORB	1,1m	1,25	1,1	-	1,1	-	<0,5	1,1	<0,5
SIFT	-	1,4	1,48	-	-	2,1	1	1,1	<0,5
SURF	-	1,2	-	0,7	-	-	0,5	1,4	<0,5
BRISK	1,25	1,4	1,3	-	-	1,3	0,6	1,2	0,6
FREAK	-	1,24	1,4	-	-	1,1	0,5	-	-

Tabla 5: Experimento II: Distancia límite

HOG: el objeto podía alejarse hasta unos 1,20 metros.

5.3.2.2. Tasa de Detección

HOG:

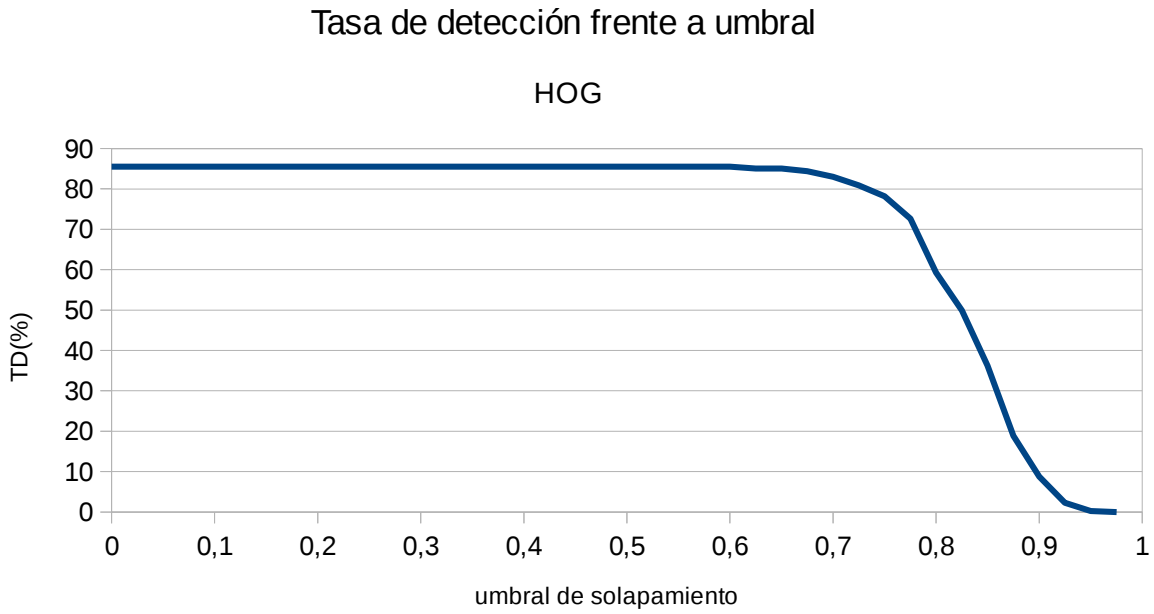


Ilustración 40: Gráfico de Tasa de Detección frente a umbral

Siendo **82.98%** el valor equivalente a un grado de solapamiento del 0..

Features

Extractor\ detector	Dense	Fast	GFTT	MSER	ORB	SIFT	Star	SURF	BRISK
BRIEF	47,81	43,21	26,67	0,2	11,5	32,64	-	20,45	14
ORB	28,73	37,24	11,03	-	20,6	-	-	21,3	7,8
SIFT	1,14	48,27	48,96	-	-	50,34	-	20,4	10
SURF	-	24,59	-	3,2	20	0,2	-	35	8,7
BRISK	30,11	51,72	50,11	0,4	13,79	47,58	-	45	15
FREAK	-	-	-	-	-	-	-	23	8,5

Tabla 6: Experimento II: TD

5.3.2.3 Tasa de Error

HOG

La tasa de error de HOG para este experimento es del

Features

Extractor\ detector	Dense	Fast	GFTT	MSER	ORB	SIFT	Star	SURF	BRISK
BRIEF	52,19	56,79	73,33	99,8	88,5	67,36	100	79,55	86
ORB	71,27	62,76	88,97	100	79,4	100	100	78,67	92,2
SIFT	98,86	51,73	51,04	100	100	49,66	100	79,6	90
SURF	100	75,41	100	96,8	80	99,8	100	65	91,3
BRISK	69,89	48,28	49,89	99,6	86,21	52,42	100	55	85
FREAK	100	100	100	100	100	100	100	77	91,5

Tabla 7: Experimento II: TE

5.3.2.4 FPPI

Al igual que el primer experimento los FPPI no llegan a superar el valor de 0,1.

5.3.3 Experimento III

Para poder comprobar una vez más el rendimiento de los algoritmos, he usado en esta ocasión una botella de plástico de agua. Concretamente una botella de 0,5 litros de la marca Bezoya (con unas diemensiones de 6 cm de diámetro y 20 cm de alto) como se puede ver en la siguiente ilustración:

Ilustración 41: Botella de agua Bezoya

Para poder detectar este tipo de objetos hay dos alternativas: o bien se crea el modelo (hog) o la imagen de plantilla (features) con la botella entera o se utiliza la imagen de la etiqueta. Ambas opciones presentan desventajas, tales como en el caso de la imagen entera se transparenta el fondo haciendo la detección más difícil, y en el caso de la etiqueta el algoritmo deberá encontrar un objeto con unas dimensiones muy pequeñas.

Ante estas dos opciones la medida adoptada ha sido la de utilizar la botella entera. Además en este caso se ha utilizado un conjunto total de unas **50** imágenes para probar los algoritmos.

5.3.3.1 Distancia límite

En este experimento no se ha medido las distancias límites a los que los algoritmos deja de detectar el objeto porque en la mayoría de los casos requería de una distancia entorno a los 50 cm. Esto se verá reflejado en las bajas tasas de detección del objeto.

5.3.3.2 Tasa de detección

HOG:

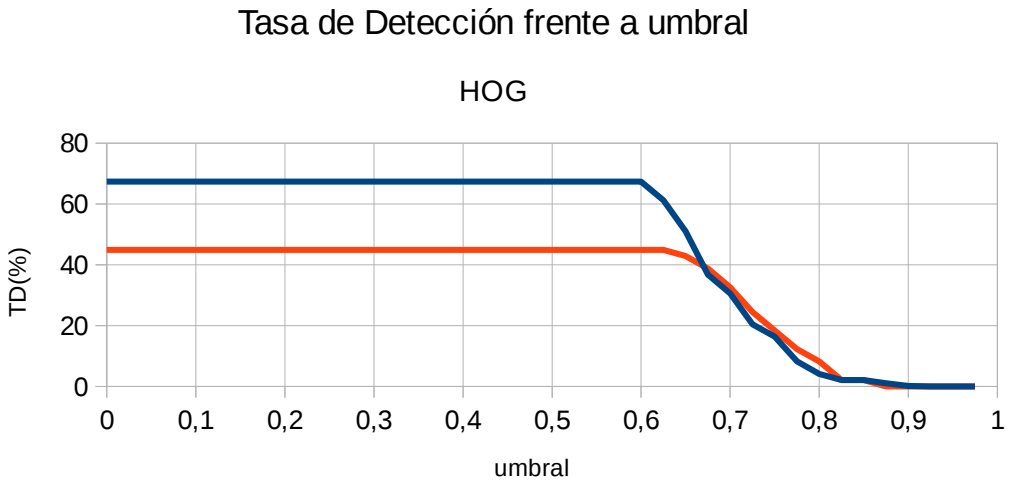


Ilustración 42: TD frente a umbral: Rojo modelo a partir 50 imágenes positivas, azul modelo a partir 61 imágenes positivas

Features:

Extractor\ detector	Dense	Fast	GFTT	MSER	ORB	SIFT	Star	SURF	BRISK
BRIEF	8,16	8,16	8,16	6,12	-	6,12	-	-	2
ORB	10,2	10,2	8,16	6,12	12,25	-	-	5	-
SIFT	-	8,16	12,25	10,2	2	32,65	-	5	-
SURF	-	8,16	-	6,12	4	2	-	6,12	-
BRISK	4,08	8,16	8,16	6,12	-	8,1	-	6,12	-
FREAK	4,08	10,2	10,2	2	-	10,2	-	4	-

Tabla 8: Experimento III: TD

5.3.3.3 Tasa de error

HOG:

Para el caso del 70% el modelo creado a partir de 50 imágenes tiene una tasa de error del 67,16 y para el creado a partir de 60 imágenes 69,4

Features:

Extractor\ detector	Dense	Fast	GFTT	MSER	ORB	SIFT	Star	SURF	BRISK
BRIEF	91,84	91,84	91,84	93,88	100	93,88	100	100	98
ORB	89,8	89,8	91,84	93,88	97,75	100	100	95	100
SIFT	100	91,84	87,75	89,8	98	77.35	100	95	100
SURF	100	91,84	100	93,88	96	98	100	93,88	100
BRISK	93,92	91,84	91,84	93,88	100	91,9	100	93,88	100
FREAK	93,92	89,8	89,8	98	100	89,8	100	96	100

Tabla 9: Experimento III: TE

5.3.3.4 FPPI

Al igual que en anteriores experimentos los falsos positivos por imagen no llegan a superar los 0,1.

6. Conclusiones

6.1 Conclusiones de los experimentos

En primer lugar, nos llama la atención las diferencias que existen entre las distancias límites entre los objetos del experimento I y II, y es que esto es perfectamente lógico ya que el objeto usado en el segundo experimento es 4 veces más pequeño .

Pero la diferencia más importante es el descenso de calidad de la detección pasando métodos como SIFT/SIFT con una TD del 78,5% al 50,34% y algunos métodos incluso no son ni capaces de detectar el objeto.

Para el caso del HOG la situación ha sido al contrario. ¿A qué se debe esto? Principalmente a dos motivos: El método de HOG requiere que el objeto se muestre al completo y gran parte de las imágenes en el experimento tapan a la caja de cereales para poder probar esto. Si no se tapa al objeto daría resultados similares a los del experimento I.

Ilustración 43: Muestras de la base de datos de las imágenes usadas en los experimentos I y II

En el caso del tercer experimento, nos encontramos ante un objeto con una forma totalmente distinta a los anteriores: cilíndrica. Esto provoca una mayor dificultad a la hora de detectarlo dado que no es lo mismo girar un producto en el que se encuentra presente la imagen que girar una botella en la que la imagen cambia y no se visualiza el mismo contenido.

Por ello necesitamos una mayor cantidad de información de la botella a parte de imágenes en las que se varía la luminosidad, haciendo que el método perfecto para reunir esta información sea el de HOG. Ya que si intentamos captar la botella con más imágenes usando como plantilla se dispararía los tiempos de detección en el caso de features.

Todo esto hace que ambos métodos tengan sus fortalezas y sus debilidades.

En el caso de features el método SIFT/SIFT ofrece unos buenos resultado con la mayor distancia límite con sólo una imagen a costa de un gran coste computacional. Y otros métodos como SURF/SURF ofrecen un gran balance entre tiempos y una buena detección.

Por otro lado el detector HOG requiere de más de una imagen para crear el modelo pero posee unos tiempos menores (siendo éstos independientes del número de objetos en pantalla) en los que el objeto debe mostrarse al completo.

Además, ambos métodos poseen una tasa de falsos positivos por imágenes ínfima debido a que en el propio código se intenta evitar estas detecciones erróneas.

6.2 Valoración del Trabajo Fin de Grado

A pesar de las dificultades que conlleva el trabajar con nuevas tecnologías como ROS en las que la curva de aprendizaje es lenta, no me arrepiento de haber escogido este Trabajo Fin de Grado. Dado que siempre me ha interesado el campo de la robótica y el aprender este *framework* lo considero vital para mi futuro.

Además he descubierto mi afán por el campo de la visión por computador tan vital en el campo de la industria y en el que sólo he visto unas pinceladas a lo largo del grado.

Y ya no sólo cómo funciona una serie de algoritmos si no cómo debe uno enfrentarse a un reto buscando la información, planificando el tiempo y documentando un trabajo.

7. Especificaciones técnicas

7.1 Ordenador

Ilustración 44: Ordenador Acer Aspire v3 571G

Procesador: Intel Core i7 3632QM (Quad Core) 2.20GHz (3.20GHz Turbo Boost)

- Memoria RAM: 4GB SO-DIMM DDR3
- Disco duro: 750 GB SATA
- Gráfica: nVidia GeForce 710M 2GB dedicados

Con sistema operativo Ubuntu 14.04 Trusty.

7.2 Cámara Kinect v2

Ilustración 45: Kinect v2

Características:

Vídeo: 1920x1080 a 30 fps

Profundidad: 512x424 a una distancia de 0.5 a 4.5 metros

Ángulo de visión: 70 grados de campo horizontal y 60 grados de campo vertical

USB: 3.0

8. Manual usuario

8.1 Guía de instalación

ROS

Para instalar la distribución de indigo en nuestro computador debemos seguir los pasos descritos en [24]:

1. Inicializamos nuestro ordenador para aceptar el sofware de packages.ros.org:

   ```
   sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu $
   (lsb_release -sc) main" > /etc/apt/sources.list.d/ros-
   latest.list'
   ```

2. Inicializamos nuestras claves:

   ```
   sudo apt-key adv --keyserver hkp://ha.pool.sks-
   keyservers.net:80 --recv-key
   421C365BD9FF1F717815A3895523BAEEB01FA116
   ```

3. Instalación:

   1. Actualizamos:

      ```
      sudo apt-get install update
      ```

   2. Instalamos el paquete completo de ROS

      ```
      sudo apt-get install ros-indigo-desktop-full
      ```

4. Inicializamos rosdep

   ```
   sudo rosdep init
   rosdep update
   ```

5. Añadimos la variable de entorno a nuestra bashrc

   ```
   echo "source /opt/ros/indigo/setup.bash" >> ~/.bashrc
   ```

6. Ejecutamos nuestra bashrc

   ```
   source ~/.bashrc
   ```

7. Finalmente para comprobar si tenemos ROS bien instalado ejecutamos:

   ```
   roscore
   ```

Si nos aparece la siguiente imagen, nuestro ROS se encuentra correctamente instalado.

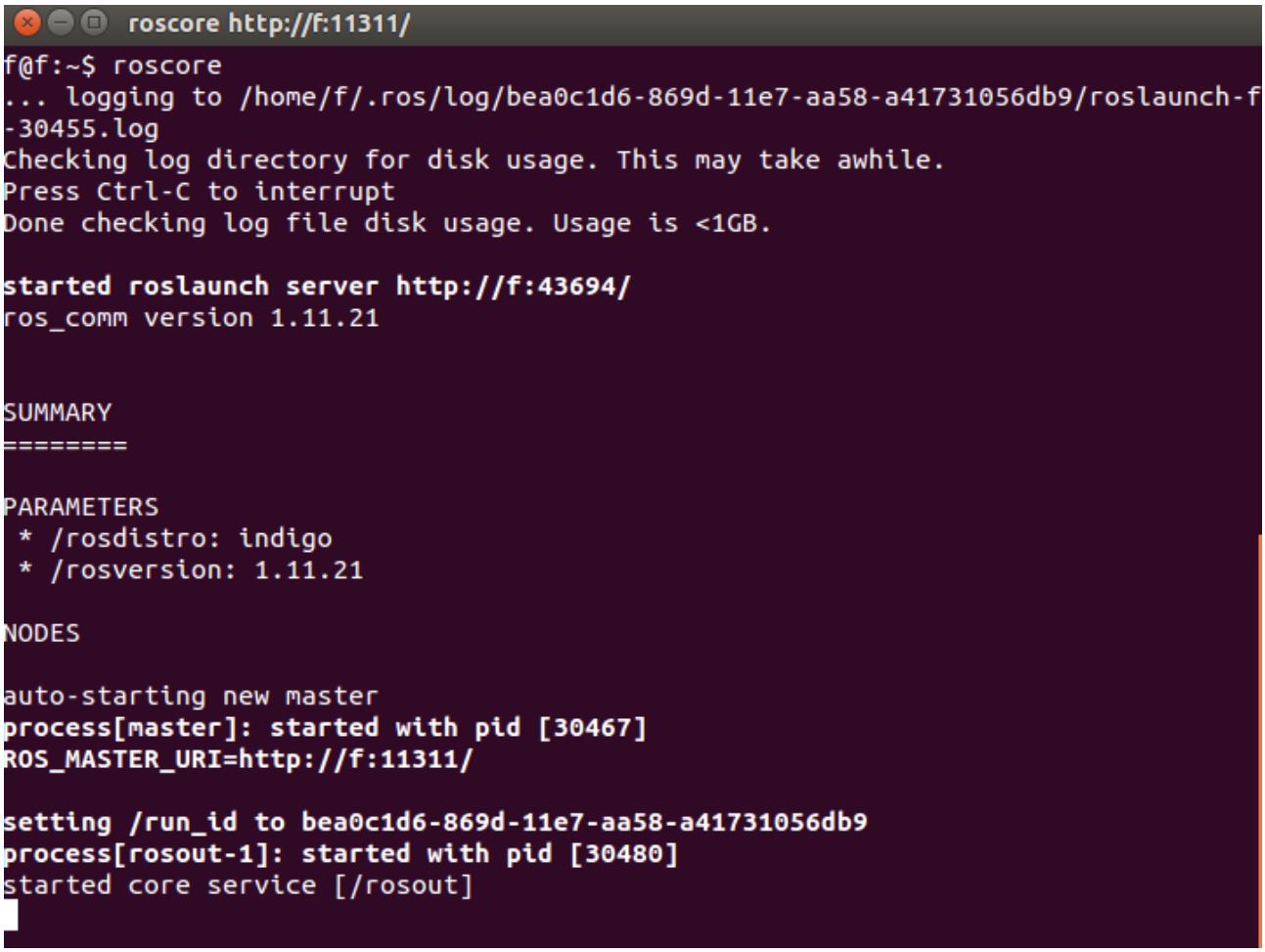


Ilustración 46: Ejecución del comando Roscore

Opencv_nonfree

Al haber instalado el paquete completo de ROS, no necesitamos descargarnos las librerías de OpenCV ya que éstas ya se encuentran en el paquete de ros-indio-vision-opencv. Sin embargo, el proyecto hace uso de unas librerías que no se encuentran integradas: **opencv_nonfree**.

Para instalar debemos ejecutar los siguientes comandos:

```
sudo add-apt-repository --yes ppa:xqms/opencv-nonfree
sudo apt-get update
sudo apt-get install libopencv-nonfree-dev
```

Kinect v2

Como bien he comentado para poder usar la kinect utilizo el paquete de iai_kinect como puente entre libfreenect2 y ROS.

Para instalar el paquete de iai_kinect se debe seguir los siguientes pasos denotados en su github[19.1]:

1. Instalar **libfreenect2** [25]
    1. Descargamos el código fuente

       ```
       git clone https://github.com/OpenKinect/libfreenect2.git
       cd libfreenect2
       ```

    2. Descargamos los archivos de actualización

       ```
       cd depends; ./download_debs_trusty.sh
       ```

    3. Instalamos las herramientas build

       ```
       sudo apt-get install build-essential cmake pkg-config
       ```

    4. Instalamos libusb

       ```
       sudo dpkg -i debs/libusb*deb
       ```

    5. Instalamos TurboJPEG

       ```
       sudo apt-get install libturbojpeg libjpeg-turbo8-dev
       ```

    6. Instalamos OpenGL (si el último entra en conflicto con los otros no lo instalamos)

       ```
       sudo dpkg -i debs/libglfw3*deb; sudo apt-get install -f;
       sudo apt-get install libgl1-mesa-dri-lts-vivid
       ```

    7. OpenCL (opcional) para Intel(mi caso)

       ```
       sudo apt-add-repository ppa:floe/beignet; sudo apt-get
       update; sudo apt-get install beignet-dev; sudo dpkg -i
       debs/ocl-icd*deb
       ```

    8. Compilamos e instalamos la librería:

       ```
       cd ..
       mkdir build && cd build
       cmake .. -DCMAKE_INSTALL_PREFIX=$HOME/freenect2
       make
       make install
       ```

9. Concedemos los permisos para el acceso al dispositivo:

```
sudo cp ../platform/linux/udev/90-kinect2.rules
/etc/udev/rules.d/
```

10. Finalmente comprobamos su funcionamiento ejecutan el siguiente comando:

```
./bin/Protonect
```

Ilustración 47: Captura de pantalla usando ./Protonect

2. Clona este repositorio dentro de tu catkin workspace, instala las dependencias y compílalo.

```
cd ~/catkin_ws/src/
git clone https://github.com/code-iai/iai_kinect2.git
cd iai_kinect2
rosdep install -r --from-paths .
cd ~/catkin_ws
catkin_make -Dfreenect2_DIR=$HOME/lib/cmake/freenect2
```

PROYECTO

En primer lugar, debemos crear nuestra zona de trabajo para ello:

```
mkdir -p catkin_ws/src
cd ~/catkin_ws/src
catkin_init_workspace
```

A continuación debemos copiar el contenido de la carpeta de proyecto sobre la carpeta src que nos hemos creado y ejecutamos:

`cd ..` para volver a nuestra zona de trabajo

Instalamos los paquetes necesarios para la kinect v2

`catkin_make install` para instalar los servicios y

`catkin_make` para compilar nuestros paquetes.

Añadimos nuestra variable de entorno a la bashrc para que se ejecute cada que que abramos una nueva terminal y asi se añada nuestros paquetes a la lista de comando en la terminal.

```
echo "source ~/catkin_ws/devel/setup.bash" >> ~/.bashrc
```

9. Referencias

[0] http://erlerobotics.com/blog/ros-introduction-es/

[1] https://www.coursera.org/learn/deteccion-objetos/lecture/WSJ9t/l4-2-hog-calculo-del-gradiente

[2] https://www.coursera.org/learn/deteccion-objetos/lecture/GBJYS/l4-3-hog-calculo-de-los-histogramas

[3] https://www.coursera.org/learn/deteccion-objetos/lecture/uAEKB/l4-4-hog-calculo-del-descriptor

[4] https://www.coursera.org/learn/deteccion-objetos/lecture/iGdzT/l4-5-support-vector-machines-svm-conceptos-basicos

[5] https://www.coursera.org/learn/deteccion-objetos/lecture/JXyER/l4-6-support-vector-machines-svm-desarrollo-matematico

[6] http://aishack.in/tutorials/sift-scale-invariant-feature-transform-scale-space/

[6.1] http://aishack.in/tutorials/sift-scale-invariant-feature-transform-features/

[7] Herbert Bay, Tinne Tuytelaars, and Luc Van Gool, "Speeded Up Robust Features", ETH Zurich, Katholieke Universiteit Leuven. Apartado 3 : Fast-Hessian Detector

[7.1] Herbert Bay, Tinne Tuytelaars, and Luc Van Gool, "Speeded Up Robust Features", ETH Zurich, Katholieke Universiteit Leuven. Apartado 4: SURF Descriptor

[8] *Rosten, Edward; Tom Drummond (2006). "Machine learning for high-speed corner detection". European Conference on Computer Vision*

[9] Ethan Rublee, Vincent Rabaud, Kurt Konolige, Gary R. Bradski: "ORB: An efficient alternative to SIFT or SURF". Apartado 3: oFAST: FAST Keypoint Orientation

[9.1] http://docs.opencv.org/3.0-beta/doc/py_tutorials/py_feature2d/py_orb/py_orb.html

[10] Akash Patel, D.R. Kasat, Sanjeev Jain, V.M. Thakare: "Performance Analysis of Various Feature Detector and Descriptor for Real-Time Video based Face Tracking". Apartado 2.3: CenSurE based STAR feature detector

[11] http://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_feature2d/py_shi_tomasi/py_shi_tomasi.html

[12] Presentación en pdf:
http://www.micc.unifi.it/delbimbo/wp-content/uploads/2011/03/slide_corso/A34%20MSER.pdf

[13] Stefan Leutenegger, Margarita Chli and Roland Siegwart: BRISK: Binary Robust Invariant Scalable Keypoints. Apartado 3.1 : Scale-Space Keypoint Detection

[13.1] Stefan Leutenegger, Margarita Chli and Roland Siegwart: BRISK: Binary Robust Invariant Scalable Keypoints. Apartado 3.2: Keypoint Description

[14] Michael Calonder, Vincent Lepetit, Christoph Strecha, and Pascal Fua, “BRIEF: Binary Robust Independent Elementary Features”, 11th European Conference on Computer Vision (ECCV), Heraklion, Crete. LNCS Springer, September 2010.

[15] Alahi, R. Ortiz, and P. Vandergheynst. FREAK: Fast Retina Keypoint. In IEEE Conference on Computer Vision and Pattern Recognition, 2012. CVPR 2012 Open Source Award Winner.

[16] Marius Muja and David G. Lowe: "Scalable Nearest Neighbor Algorithms for High Dimensional Data". Pattern Analysis and Machine Intelligence (PAMI), Vol. 36, 2014.

[17] https://www.coursera.org/learn/deteccion-objetos/lecture/w9oWH/l3-6-evaluacion-del-rendimiento-evaluacion-del-detector

[18]http://wiki.ros.org/cv_bridge/Tutorials/UsingCvBridgeToConvertBetweenROSImagesAndOpenCVImages

[19] https://github.com/code-iai/iai_kinect2

[19.1] https://github.com/code-iai/iai_kinect2#install

[20] https://github.com/trane293/Object-Detection-Framework-Using-HOG-And-SVM

[21] http://svmlight.joachims.org/

[22] http://wiki.ros.org/find_object_2d con el repositorio https://github.com/introlab/find-object.git

[23] http://rgbdemo.org/index.php/Documentation/KinectCalibrationTheory

[24] http://wiki.ros.org/indigo/Installation/Ubuntu

[25] https://github.com/OpenKinect/libfreenect2/blob/master/README.md#linux

[26] Noticia escrita por el usuario Miguel López del portal de xataka: https://www.xataka.com/robotica-e-ia/este-es-el-primer-almacen-robotizado-de-amazon-en-espana