Trabajo de Fin de Grado

Grado en Ingeniería de las Tecnologías Industriales

Desarrollo de un sistema de visión artificial para la medida de deflexión en un brazo robótico de articulaciones flexibles

Autor: Pablo Russo del Río

Tutores: Alejandro Suárez Fernández-Miranda

Jose Guillermo Heredia Benot

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

Sevilla, 2022

Trabajo de Fin de Grado

Grado en Ingeniería de las Tecnologías Industriales

Desarrollo de un sistema de visión artificial para la medida de deflexión en un brazo robótico de articulaciones flexibles

Autor:

Pablo Russo del Río

Tutores:

Alejandro Suárez Fernández-Miranda

Jose Guillermo Heredia Benot

Profesor Titular

Dpto. Ingeniería de Sistemas y Automática

Escuela Técnica Superior de Ingeniería

Universidad de Sevilla

Sevilla, 2022

Trabajo de Fin de Grado: Desarrollo de un sistema de visión artificial para la medida de deflexión en un brazo robótico de articulaciones flexibles

Autor: Pablo Russo del Río
Tutores: Alejandro Suárez Fernández-Miranda Jose Guillermo Heredia Benot

El tribunal nombrado para juzgar el trabajo arriba indicado, compuesto por los siguientes profesores:

Presidente:

Vocal/es:

Secretario:

acuerdan otorgarle la calificación de:

El Secretario del Tribunal

Fecha:

Agradecimientos

Transmitir mi más sincero agradecimiento a todos aquellos que me han ayudado a lo largo de esta etapa y me han apoyado.

Especial agradecimiento a mis tutores Alejandro Suárez Fernández-Miranda y Jose Guillermo Heredia Benot, por su gran implicación, y por la excelente orientación en la elección del tema de este Trabajo de Fin de Grado.

Resumen

Este trabajo propone una solución frente a un problema presente en los brazos robóticos flexibles: **la deflexión**. Se ha desarrollado un sistema de visión que utiliza varios métodos simultáneamente, buscando la mayor precisión y fiabilidad a la hora de medirla. Los métodos utilizados son:

- **la detección por umbralización del color**: Se utilizará con el objetivo de encontrar un marker rojo en la muñeca del robot.
- **filtrado de fondo por profundidad**: Aprovecha el conocimiento de las dimensiones del robot para su desempeño
- **CamShift**: Algoritmo ampliamente utilizado en el tracking de objetos por su color.

Se calculará la deflexión como la **diferencia entre la posición cartesiana del marker**, proporcionada por el sistema de visión, y los **puntos obtenidos del modelo rígido**, calculados del modelo cinemático del robot.

Mediante dos experimentos distintos en un entorno controlado se ha comprobado la eficacia del sistema de visión en la precisión de las medidas obtenidas por la cámara y su cálculo de la deflexión.

Abstract

This Bachelor Thesis proposes a solution to a common problem in flexible robotic arms: **the deflection**.

A vision system has been developed including several methods simultaneously, seeking the highest accuracy and reliability when measuring it. The methods used are:

- **detection by color thresholding**: It will be used with the objective of finding a red marker on the wrist of the robot.
- **depth background filtering**: It takes advantage of the knowledge of the robot's dimensions for its performance.
- **CamShift**: Algorithm widely used in object tracking by color.

.

The deflection will be calculated as the **difference between the Cartesian position of the marker**, provided by the vision system, and the points obtained from the **stiff-joint model**, obtained from the kinematic model of the robot.

Two experiments in a controlled environment tested the effectiveness of the vision system in the accuracy of the measurements obtained by the camera and its deflection calculation.

Índice

1 Introducción

El uso de brazos robóticos no rígidos y ligeros tiene grandes ventajas frente a los brazos robóticos rígidos. Su bajo peso y resistencia a impactos, gracias a sus amortiguadores, permite implementar estos brazos en drones que podrían realizar labores de mantenimiento e inspección que los seres humanos no podrían.

La gran desventaja que presentan es la menor precisión del cálculo de la posición del efector final. Esto es debido a diversos factores como la inercia, la gravedad, impactos o el agarre de objetos de mayor peso.

El **objetivo principal** de este trabajo es desarrollar un sistema de visión que permita calcular la deflexión producida en el robot, que él mismo no sería capaz de medir. Se buscará utilizar todas las herramientas que la cámara Intel REALSENSE D435i dispone buscando un desempeño que se adapte a diversos escenarios y sea lo más fiable posible para conseguir un sistema flexible y robusto con la menor interacción con el usuario posible. La implementación de brazos robóticos no rígidos es un campo relativamente novedoso y en auge, por lo que la aplicabilidad de este tipo de proyectos es notoria.

Se ha probado experimentalmente el sistema de visión para mostrar sus ventajas, aunque también las limitaciones en su desempeño.

El documento se ha estructurado de la siguiente manera:

- **Descripción del sistema**: Se describirá y explicará los diferentes elementos que componen nuestro sistema.
- **Conceptos teóricos**: Se desarrollará la base teórica necesaria para la comprensión del proyecto.
- **Diseño de la solución**: Se mostrará las herramientas utilizadas y se desarrollará los puntos más importantes del diseño del programa.
- **Resultados experimentales**: Se mostrarán los resultados obtenidos y se discutirán
- **Conclusiones**: Se desarrollará las conclusiones del proyecto, así como sus ventajas y limitaciones y posibles ampliaciones.

2 Descripción del sistema

En este capítulo se describirá y explicará los diferentes elementos que componen nuestro sistema.

2.1 Doble brazo antropomórfico, adaptable y ligero

Existen actividades de inspección y mantenimiento que conllevan un riesgo si son realizadas por un humano. Estos brazos robóticos, capaces de imitar la cinemática de unos brazos humanos, podrían realizar estas actividades. En la Figura 2.1 podemos ver los brazos robóticos.

Los brazos utilizados son los **LiCAS A1** que han sido creados por el grupo de visión y control de la Universidad de Sevilla. Su diseño busca la mejor adaptabilidad, bajo peso, inercia y resistencia. Cada brazo tiene tres articulaciones en el hombro y una articulación más en el codo. Los eslabones que unen la muñeca y el codo y este último con el hombro son de 25cm. La separación entre brazos es de **32 cm**, la distancia vertical entre el hombro y la cámara es de 23 cm. Cada brazo tiene un **marker rojo** en su muñeca que se utilizará en el algoritmo de visión.

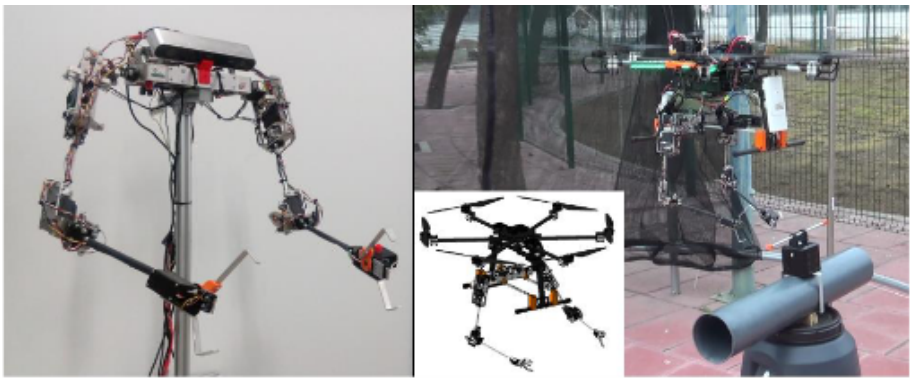


Figura 2.1 Brazos robóticos(Suarez et al. (b)) .

2.1.1 Modelo cinemático

En la Figura 2.2 podemos ver los ejes del modelo cinemático y otros datos de interés.

$X^iY^iZ^i$ son los ejes del i eslabón del brazo y $X^cY^cZ^c$ son los ejes de la cámara. Los superíndices i=1,2 denotan el brazo izquierdo y derecho respectivamente y los subíndices j=1,2,3,4 denotan las articulaciones en el siguiente orden:Pitch, roll y yaw del hombro, y pitch del codo.

Los brazos no son rígidos, por lo que se producirá un fenómeno llamado **deflexión**. La deflexión

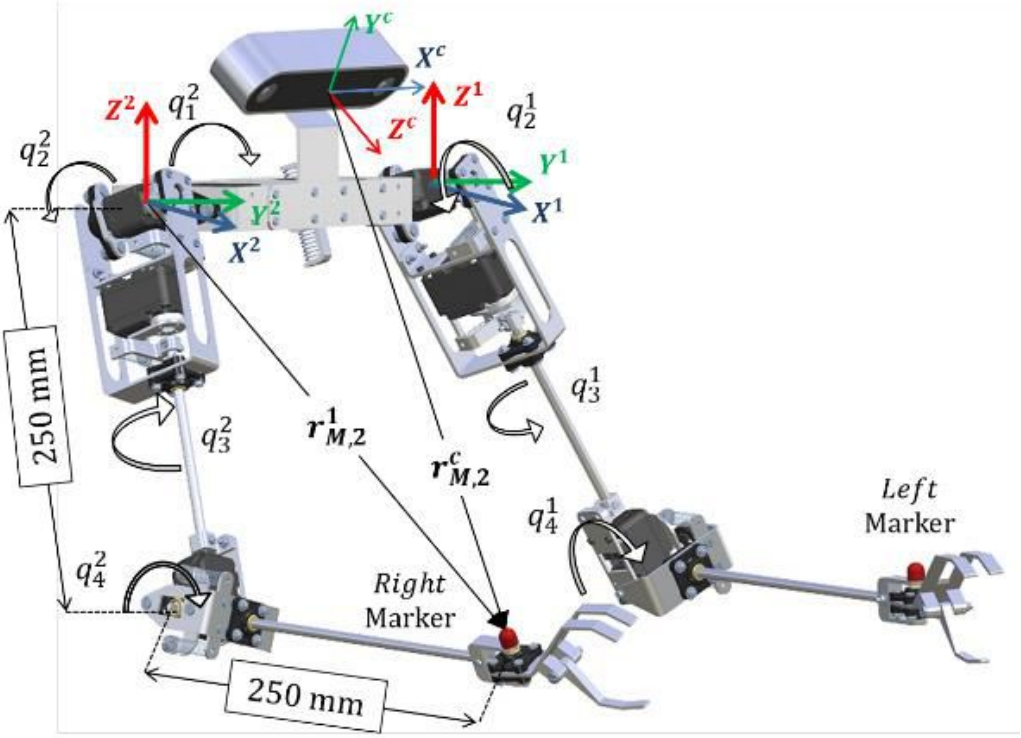


Figura 2.2 Ejes de coordenadas del modelo cinemático del robot y ejes de coordenadas genéricos de la cámara (Suarez et al. (b)) .

cartesiana la denominaremos como la diferencia entre la posición del marker proporcionada por la cámara, que corresponde con la **posición real**, y la proporcionada por la cinemática del robot rígido, que corresponde con la **posición ideal**. Al ser los ejes de la cámara y del brazo diferente será necesario calcular una **matriz de transformación** ${}_c^i T \in \Re^{3x3}$ entre ambos sistemas de referencia. Esta matriz la podemos construir debido a que conocemos la localización y orientación de la cámara. Se calculará la deflexión usando la fórmula 2.1 donde Δl^i representa la deflexión, r_c^i representa la posición en ejes de cámara del marker y θ^i la posición angular de los servos.

$$\Delta l^i =_c^i T \cdot r_c^i - F_i^i(\theta^i) \tag{2.1}$$

La deflexión es una magnitud útil que se puede utilizar en labores de control, estimación del peso de un objeto que sujete el brazo, etc.

2.2 Cámara Intel RealSense D435i con sensor de profundidad

La Cámara **Intel RealSense D435i** es un sistema de visión estereoscópica capaz de medir la profundidad gracias a su sensor de profundidad. Este sistema incluye procesador de visión, el módulo de profundidad estereoscópica, el sensor RGB con procesamiento de señal de imagen en color y una unidad de medición inercial compuesta por un acelerómetro y un giroscopio. Figura 2.3, Figura 2.4

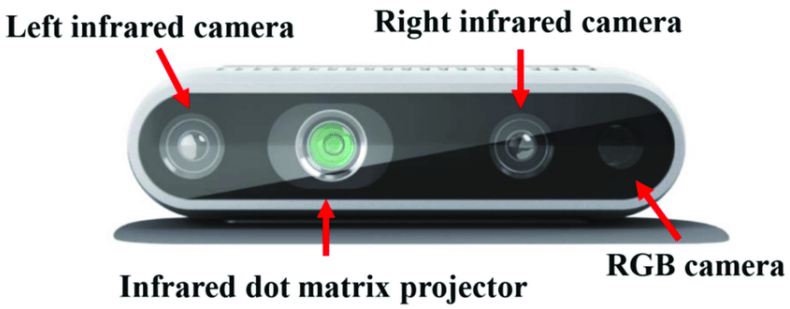


Figura 2.3 Intel RealSense D435i .

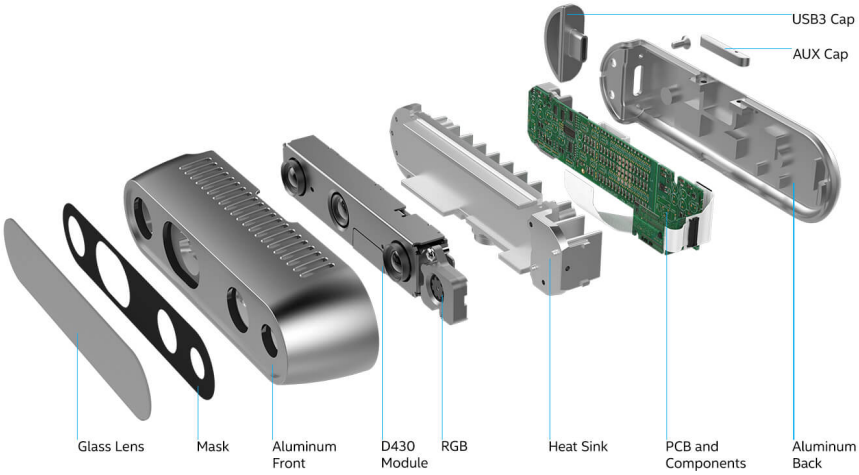


Figura 2.4 Componentes del ensamblado de la cámara Intel RealSense D435i .

3 Conceptos teóricos

En este capítulo se explicará los conceptos teóricos necesarios para el desarrollo del proyecto.

3.1 Visión artificial

La **visión artificial** se basa en adquirir, procesar y analizar datos numéricos obtenidos de imágenes o vídeos con un fin particular. La adquisición de imágenes no es un problema trivial debido a que los datos presentan ruidos y distorsiones a causa de imperfecciones en la lente, discretización en píxeles de los datos, ruido electrónico presente en los sensores, etc. Para solventar estos problemas se utilizarán los procedimientos explicados en los siguientes apartados.

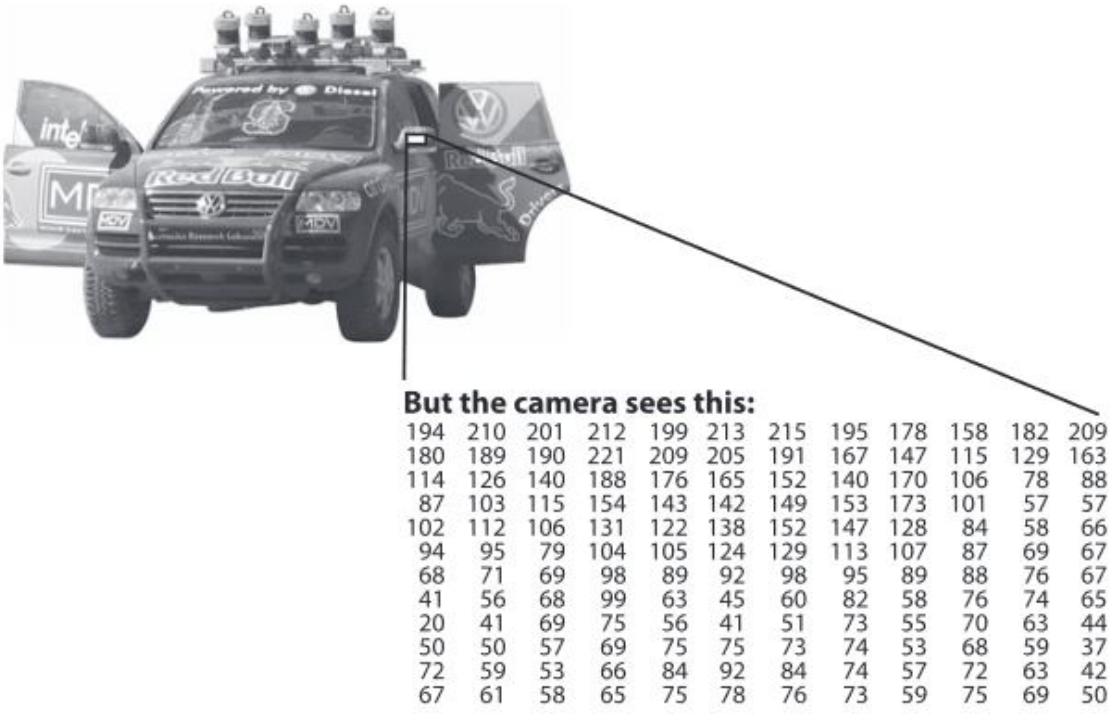


Figura 3.1 Matriz de datos que obtiene el ordenador de la imagen (Bradski y Kaehler).

3.2 Proyección de imágenes y modelo Pinhole

En este apartado se describirá las herramientas matemáticas para la transformación de los píxeles de la imagen a puntos en un sistema de coordenadas en 3D. El modelo utilizado de la cámara será el modelo Pinhole

3.2.1 Píxel

Las imágenes tienen asociado un sistema de coordenadas en 2 dimensiones divididos en píxeles, estos se pueden definir como índices que se utilizan para moverse por la imagen. El origen se encuentra en la esquina superior izquierda. Tomando como punto de vista la perspectiva de la cámara, el eje x tiene sentido positivo hacia la derecha, mientras que el eje y tiene sentido positivo hacia abajo.

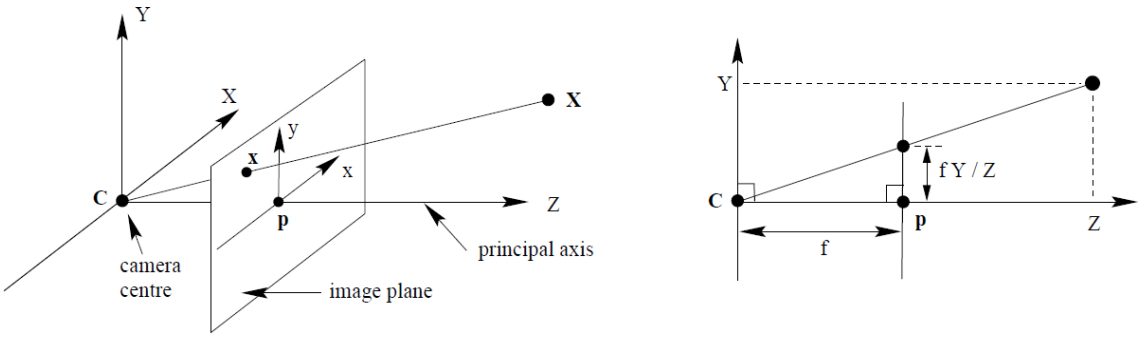


Figura 3.2 Modelo Pinhole.

3.2.2 Coordenadas en 3D

La cámara tiene un sistema de coordenadas en 3D que correspondería con el "mundo real". El origen se encuentra en el centro físico de la cámara.

3.2.3 Parámetros intrínsecos

Los parámetros intrínsecos permiten cambiar de los ejes 2D a 3D y viceversa. Los parámetros son los siguientes:

- **Distancia focal**: Distancia desde el centro físico de la cámara hasta la imagen.
- **Centro de proyección**: Punto donde corta el eje Z con el plano imagen
- **Coeficientes de distorsión**: Todas las imágenes reales tiene una distorsión rectificada gracias a estos coeficientes

En la Figura 3.2 podemos ver una representación del modelo Pin hole, donde P se correspondería con el Centro de proyección, C con el centro físico de la cámara y f con la distancia focal. Las ecuaciones del modelo son:

$$x - P_x = \frac{f_x \cdot X}{Z}, \tag{3.1}$$

$$y - P_y = \frac{f_y \cdot Y}{Z}, \tag{3.2}$$

x e y son las coordenadas en píxeles, mientras que X, Y, Z son las coordenadas en 3D. f_x y f_y son las distancias focales y P_x y P_y las coordenadas del centro de proyección.

3.2.4 Parámetros extrínsecos

Los parámetros extrínsecos de la cámara nos permite cambiar de los ejes de coordenadas 3D de la cámara a otros ejes diferentes en 3D. Los parámetros extrínsecos son los siguientes:

- **Traslación**: Vector que une los centros de ambos ejes de coordenadas.
- **Rotación**: Matriz 3x3 ortonormal de rotación entre ambos ejes de coordenadas.

Conociendo estos parámetros extrínsecos se puede crear la matriz de transformación para cambiar de un eje de coordenadas a otro.

3.3 Procesamiento de imágenes

Es muy ventajoso tratar las imágenes adquiridas por la cámara como paso previo a trabajar con ellas. El resultado de este procesamiento es una imagen modificada más adecuada para el objetivo deseado.

3.3.1 Espacio de color HSV

Las imágenes obtenidas con la cámara tienen el espacio de color en formato **RGB**(Red, Green, Blue). Es muy habitual transformar este espacio a modelo **HSV**(Hue, saturation, value) cuando se desea segmentar objetos de una imagen por su color. Este modelo fue creado para aportar tanto información sobre el color en sí como sobre la cantidad y brillo del mismo, representando todo a la vez en un mismo diagrama cromático. El espacio HSV se representa tridimensionalmente como un cono (Figura 3.3). Las siglas HSV corresponden a **Hue**(tono), **saturation**(saturación) y **value**(brillo). El **hue** es un ángulo cuyo rango es $[0,2\pi]$. La **saturación** es la distancia radial desde el eje central hasta la superficie exterior. El **brillo** es el eje central vertical del cono. La cámara adquiere las imágenes en **RGB** y se le aplica la siguiente transformación a los niveles de intensidad de cada píxel. En las ecuaciones (3.3)-(3.4)-(3.5), los valores en R,G,B se divide entre 255 para cambiar su rango de $[0,255]$ a $[0,1]$.

$$R' = R/255, \tag{3.3}$$
$$G' = G/255, \tag{3.4}$$
$$B' = B/255, \tag{3.5}$$
$$C_(max) = max(R', G', B'), \tag{3.6}$$
$$C_(min) = min(R', G', B'), \tag{3.7}$$
$$\Delta = C_(max) - C_(min), \tag{3.8}$$

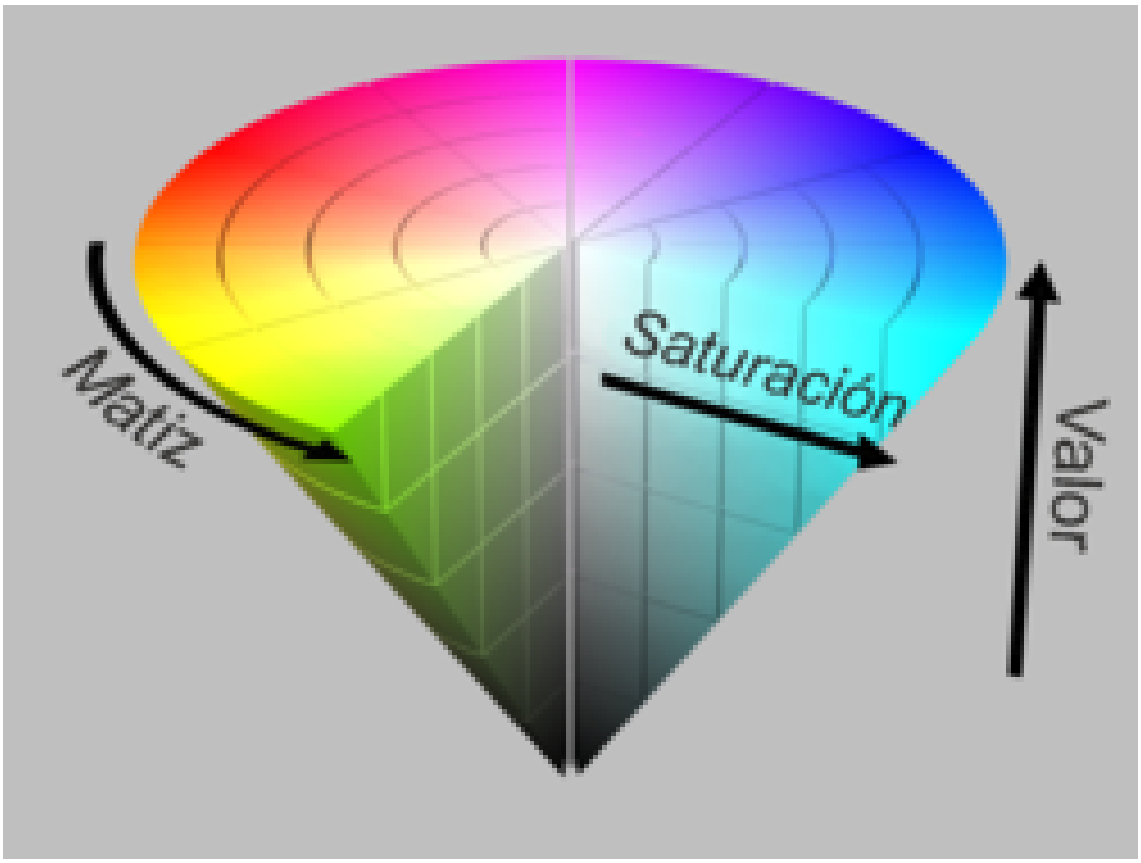


Figura 3.3 Representación tridimensional del espacio HSV.

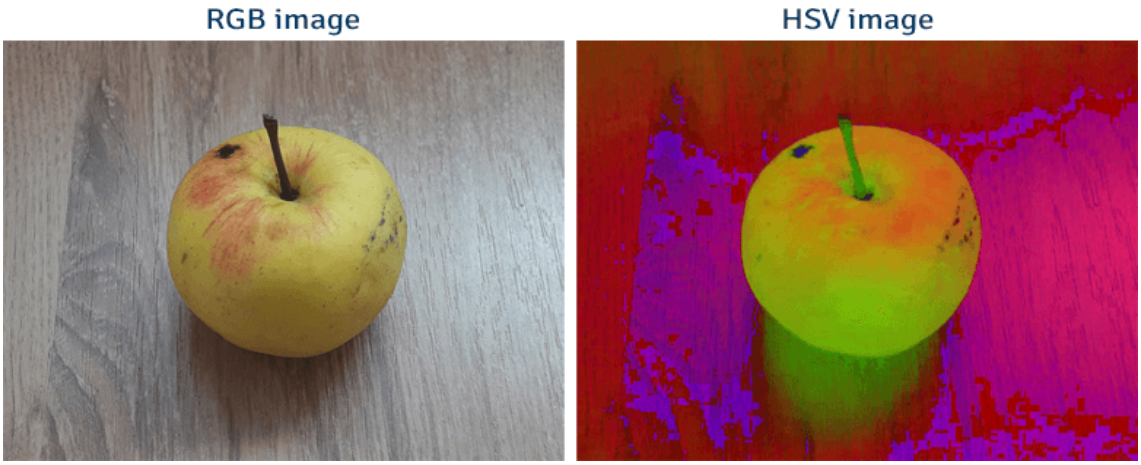


Figura 3.4 Imagen en RGB y en HSV.

Las fórmulas para calcular hue(H), saturación(S) y value(V) son:

$$H = \begin{cases} 0 & \Delta = 0 \\ 60(\frac{G'-B'}{\Delta} mod 6) & C_(max) = R' \\ 60(\frac{B'-R'}{\Delta} + 2) & C_(max) = G' \\ 60(\frac{R'-G'}{\Delta} + 4) & C_(max) = B' \end{cases}$$

$$S = \begin{cases} 0 & C_{(}max) = 0 \\ \frac{\Delta}{C_{(}max} & C_{(}max) \neq 0 \end{cases}$$

$$V = C_{(}max)$$

3.3.2 Blurring (Suavizado)

El **Blurring** es un proceso habitual en el procesado de imágenes, ya que es sumamente útil para la reducción del ruido. Existen diferentes formas de llevar a cabo el Blurring (filtro de media, filtro gaussiano, etc) aunque nos centraremos en **The simple blur operation**(SBO), ya que es la que tiene un coste computacional menor y es la más adecuada para nuestro sistema en tiempo real.

SBO utiliza **kernel morfológicos** que a diferencia de los kernel convolucionales, no requieren de ningún valor numérico. Los elementos del kernel simplemente indican donde se realiza la operación mientras él se mueve por la imagen. En la figura 3.5 podemos ver un ejemplo de kernel. El SBO

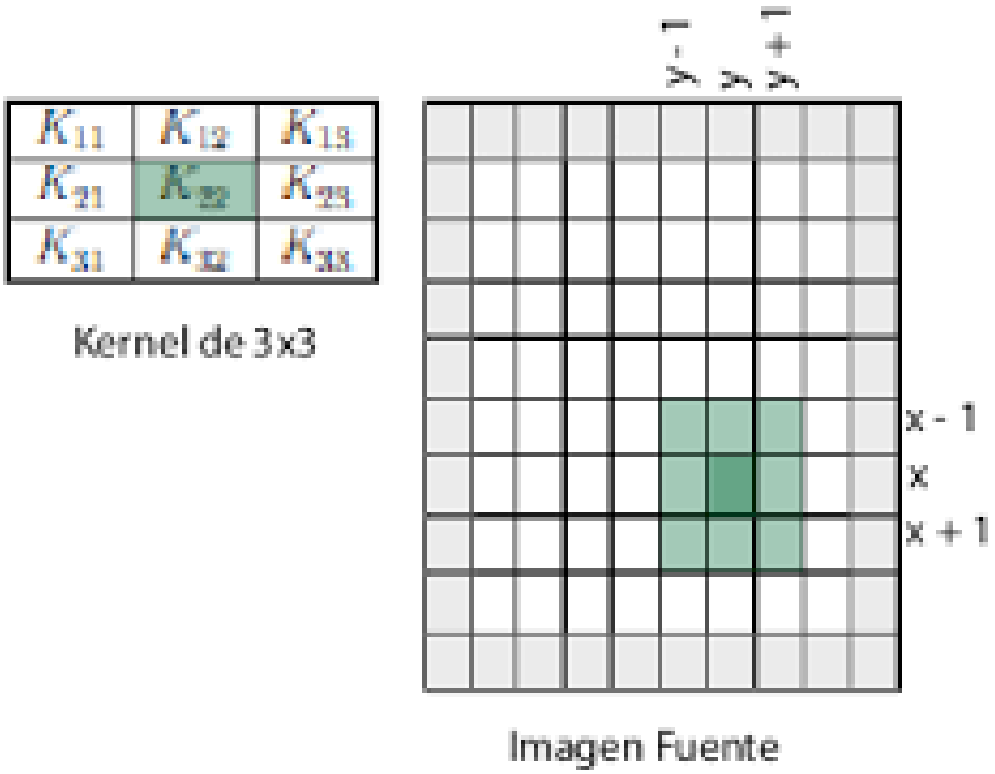


Figura 3.5 Kernel $3x3$.

funciona haciendo la media de todos los píxeles de una ventana cuyo centro es el píxel seleccionado. En la figura 3.6 podemos ver un ejemplo de **Blurring**.

Figura 3.6 Imagen original(izquierda); Imagen tratada con **Blurring** (derecha) (REF:Bradski y Kaehler).

3.3.3 Transformaciones morfológicas

Las **transformaciones morfológicas** son operaciones no lineales a diferencia del Blurring que si es lineal. Las principales transformaciones morfológicas son la **erosión** y la **acreción**. Son muy utilizadas para eliminar ruidos, separar objetos o unir diferentes elementos.

3.3.3.1 Erosión y Acreción

La **acreción** es la convolución de una imagen(o una parte de ella) con un **kernel**. El kernel se puede entender como una máscara cuyo efecto es el de un operador de máximos locales. Un kernel puede ser de cualquier tamaño y forma. Cuando el kernel es aplicado, se encuentra el píxel de máximo valor dentro de una ventana de dimensiones del kernel y se reemplaza el valor del centro de dicho ventana por el máximo encontrado. Este proceso aumenta el área de los contornos.(Figura 3.7).

La **erosión** es la operación contraria. Aplicar una erosión es el equivalente a utilizar un operador de mínimos locales. Este proceso disminuye el área de los contornos encontrados.(Figura 3.8) Las fórmulas matemáticas de la erosión y la acreción están descritas en la figura 3.9

3.3.3.2 Apertura y cierre

Se puede combinar las operaciones anteriores. Una **apertura** es una erosión seguida de una acreción y puede emplearse con contornos para eliminar ruido. Un **cierre** es una acreción seguida de una erosión y puede emplearse para rellenar los contornos.

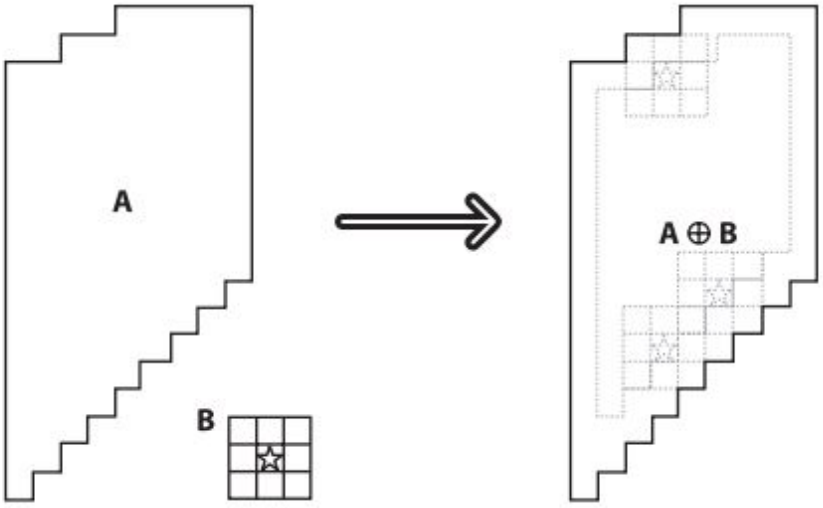


Figura 3.7 acreción (REF:Bradski y Kaehler).

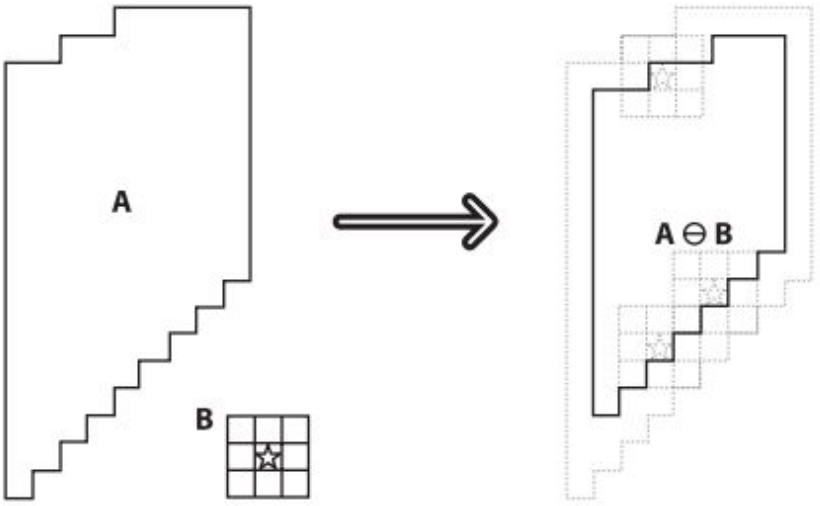


Figura 3.8 Erosión (REF:Bradski y Kaehler).

$$\text{erode}(x,y) = \min_{(x',y')\in\text{kernel}} \text{src}(x+x',y+y')$$
$$\text{dilate}(x,y) = \max_{(x',y')\in\text{kernel}} \text{src}(x+x',y+y')$$

Figura 3.9 Fórmulas matemáticas de la erosión y la acreción.

3.4 Segmentación

La **segmentación** se centra en aislar objetos o partes de objetos del resto de la imagen. Existen diferentes métodos de segmentación que describimos a continuación.

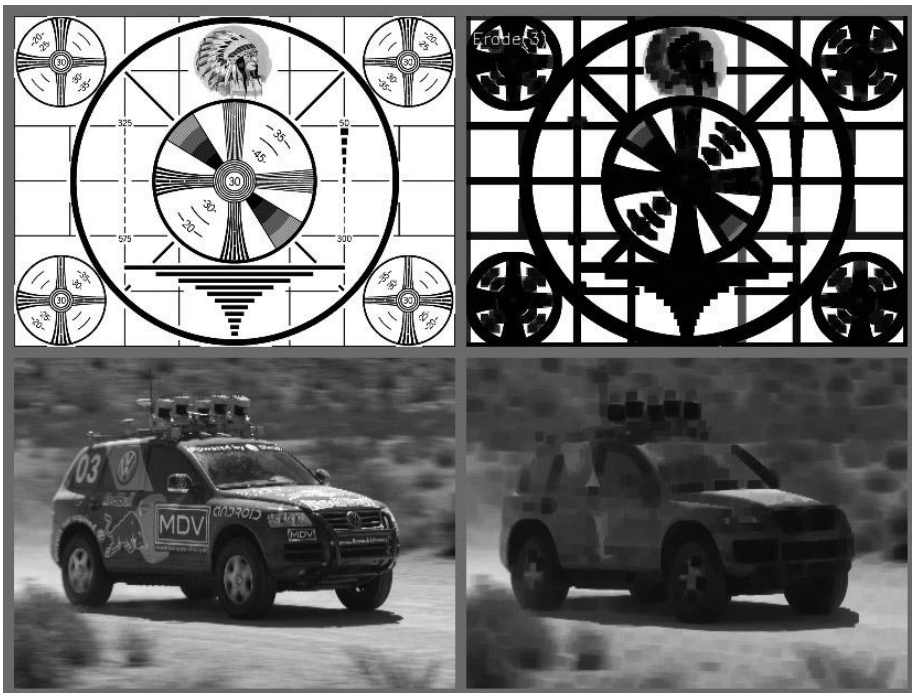


Figura 3.10 Ejemplo de una erosión (REF:Bradski y Kaehler).

3.4.1 Umbralización

La **umbralización** es el método de segmentación más simple. Separa los objetos que nos interesa de la imagen debido a la variación de la intensidad de los píxeles del objeto con los píxeles del fondo. Para diferenciar los píxeles que nos interesan se compara la intensidad de cada píxel con un umbral, el cual se escogerá dependiendo de las condiciones del problema. Una vez se han separado los píxeles se crea una nueva imagen en la cual se le asigna los valores que nos interese.

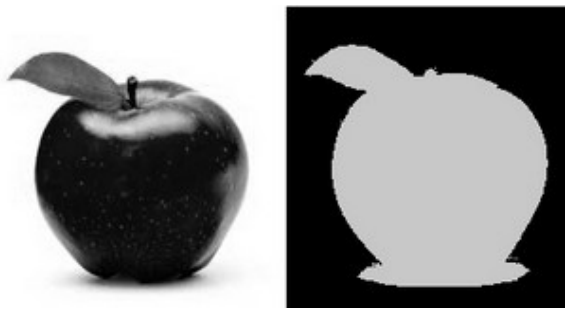


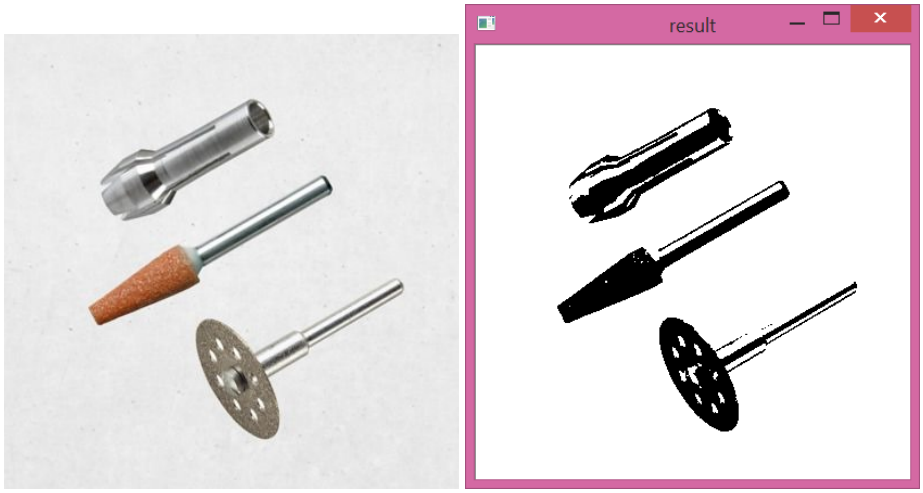
Figura 3.11 Imagen original (izquierda); imagen segmentada por umbralización (derecha) .

3.4.1.1 Umbralización binaria

La operación se expresa asi:

$$dst(x,y) = \begin{cases} maxval & src(x,y) > umbral \\ 0 & \text{en cualquier otro caso} \end{cases}$$

dst(x,y) es el valor del píxel final y src(x,y) el original.



(a) Imagen original. **(b)** Imagen umbralizada.

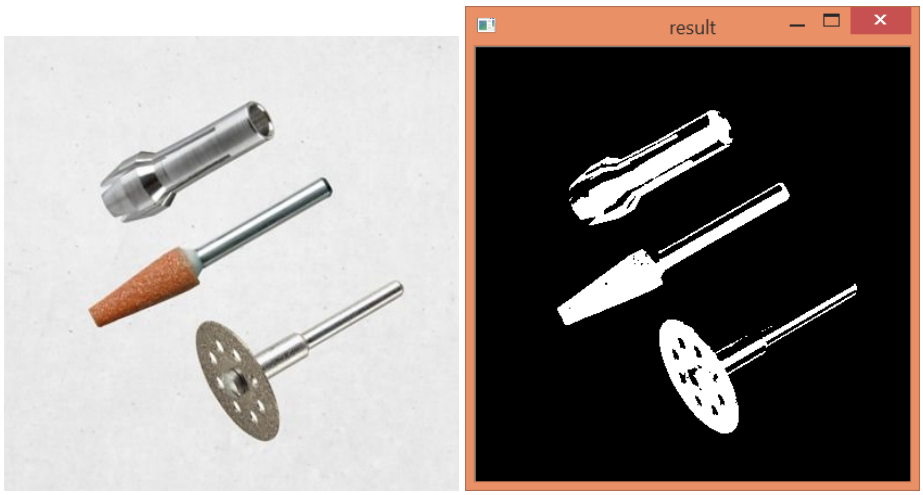
Figura 3.12 Umbralización binaria.

3.4.1.2 Umbralización inversa

La operación se expresa así:

$$dst(x,y) = \begin{cases} 0 & src(x,y) > umbral \\ maxval & \text{en cualquier otro caso} \end{cases}$$

dst(x,y) es el valor del píxel final y src(x,y) el original.



(a) Imagen original. **(b)** Imagen umbralizada.

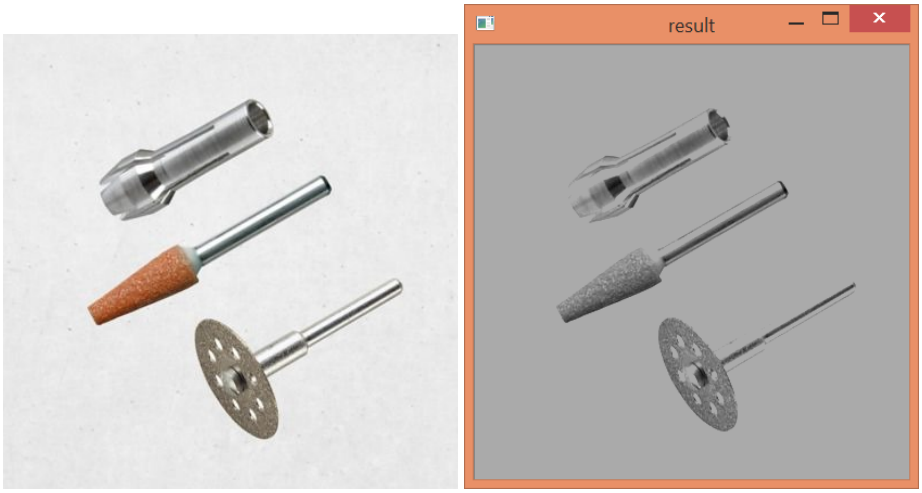
Figura 3.13 Umbralización binaria inversa.

3.4.1.3 Umbralización truncada

La operación se expresa asi:

$$dst(x,y) = \begin{cases} Umbral & src(x,y) > umbral \\ src(x,y) & \text{en cualquier otro caso} \end{cases}$$

dst(x,y) es el valor del píxel final y src(x,y) el original.



(a) Imagen original. **(b)** Imagen umbralizada.

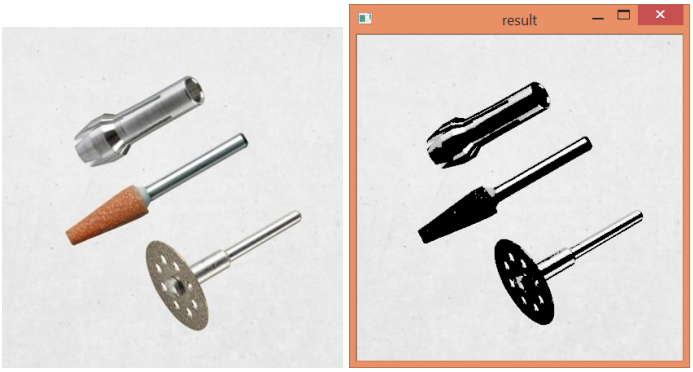
Figura 3.14 Umbralización truncada.

3.4.1.4 Umbralización a cero

La operación se expresa asi:

$$dst(x,y) = \begin{cases} src(x,y) & src(x,y) > umbral \\ 0 & \text{en cualquier otro caso} \end{cases}$$

dst(x,y) es el valor del píxel final y src(x,y) el original.



(a) Imagen original. **(b)** Imagen umbralizada.

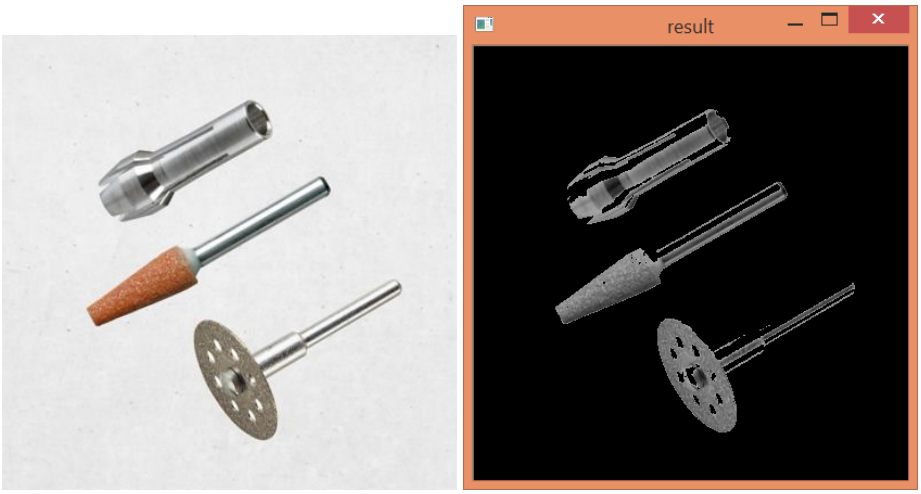
Figura 3.15 Umbralización a cero.

3.4.1.5 Umbralización a cero inversa

La operación se expresa asi:

$$dst(x,y) = \begin{cases} 0 & src(x,y) > umbral \\ src(x,y) & \text{en cualquier otro caso} \end{cases}$$

dst(x,y) es el valor del píxel final y src(x,y) el original.



(a) Imagen original. **(b)** Imagen umbralizada.

Figura 3.16 Umbralización a cero inversa.

3.4.2 Momentos

Una de las formas más simples de obtener información sobre la forma de un contorno es obtener sus **momentos**.

Un **momento** se puede entender a grosso modo como una característica de un contorno obtenida de integrar a través de todos los píxeles del mismo. En general, se define el (p,q) momento de un contorno como la ecuación (3.9).

$$m_{p,q} = \sum_{i=1}^{N} I(x,y) x^p y^p \tag{3.9}$$

p es el orden de la x y **q** es el orden de la i. Con orden nos referimos al peso que cada componente tienen en el sumatorio. El sumatorio se lleva a cabo en todos los píxeles del contorno(n).

Los **momentos** calculados con (3.9) no son demasiado útiles en la práctica. Resulta mucho más ventajoso utilizar momentos normalizados, ya que los objetos con la misma forma, pero distinto tamaño, dan valores similares.

los **momentos centrales** se definen de igual manera que los momentos descritos anteriormente pero usando las medias en vez de los valores de x e y.

$$x_{avg} = \frac{m_{10}}{m_{00}}$$
$$y_{avg} = \frac{m_{01}}{m_{00}}$$
$$\mu_{p,q} = \sum_{i=1}^{N} I(x,y)(x - x_{avg}^{p})(y - y_{avg}^{p})$$

Los **momentos normalizados** se definen de igual manera que los momentos centrales, con la diferencia de que están divididos por un valor concreto de m_{00}.

$$\eta_{p,q} = \frac{\mu_{p,q}}{m_{00}^{\frac{p+q}{2+1}}} \tag{3.10}$$

Los momentos invariantes de Hu son una combinación lineal de los **los momentos centrales**. El concepto consiste en combinar los momentos centrales normalizados, para crear funciones que dan información sobre diferentes características de un contorno, siendo estas invariantes con respecto escalado y rotaciones.

En la figura 3.17 están descritos los momentos de Hu.

$$h_1 = \eta_{20} + \eta_{02}$$
$$h_2 = (\eta_{20} - \eta_{02})^2 + 4\eta_{11}^2$$
$$h_3 = (\eta_{30} - 3\eta_{12})^2 + (3\eta_{21} - \eta_{03})^2$$
$$h_4 = (\eta_{30} + \eta_{12})^2 + (\eta_{21} + \eta_{03})^2$$
$$h_5 = (\eta_{30} - 3\eta_{12})(\eta_{30} + \eta_{12})((\eta_{30} + \eta_{12})^2 - 3(\eta_{21} + \eta_{03})^2) + (3\eta_{21} - \eta_{03})(\eta_{21} + \eta_{03})(3(\eta_{30} + \eta_{12})^2 - (\eta_{21} + \eta_{03})^2)$$
$$h_6 = (\eta_{20} - \eta_{02})((\eta_{30} + \eta_{12})^2 - (\eta_{21} + \eta_{03})^2) + 4\eta_{11}(\eta_{30} + \eta_{12})(\eta_{21} + \eta_{03})$$
$$h_7 = (3\eta_{21} - \eta_{03})(\eta_{21} + \eta_{03})(3(\eta_{30} + \eta_{12})^2 - (\eta_{21} + \eta_{03})^2) - (\eta_{30} - 3\eta_{12})(\eta_{21} + \eta_{03})(3(\eta_{30} + \eta_{12})^2 - (\eta_{21} + \eta_{03})^2)$$

Figura 3.17 Momentos de Hu.

3.4.3 GrabCut

Grabcut es un algoritmo novedoso de segmentación en 2D diseñado para la extracción del fondo buscando la interacción mínima con el usuario.

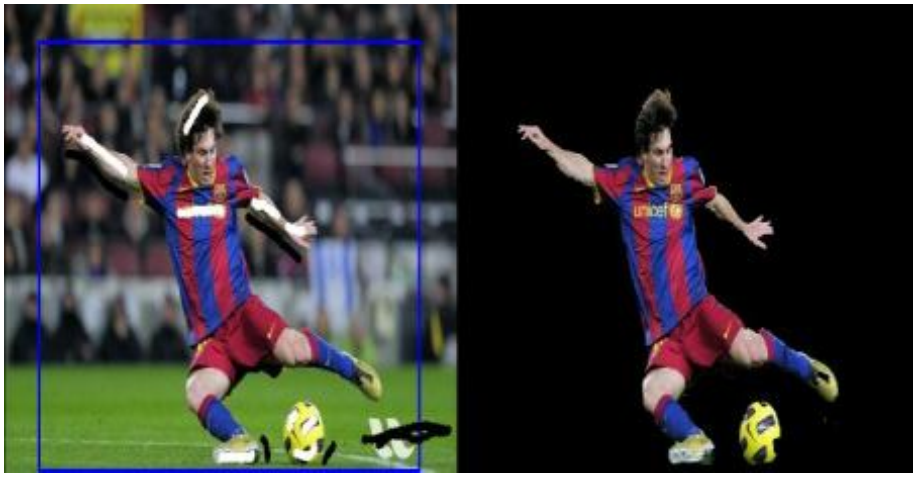


Figura 3.18 Ejemplo GrabCut.

El algoritmo Grabcut recibe una imagen con un **bounding box** que encierra los píxeles donde se encuentra el objeto que queremos segmentar o una máscara aproximada de la segmentación. Luego sigue los siguientes pasos iterativamente:

- Estima la distribución de color del fondo y del primer plano utilizando un **Gaussian Mixture Model**.
- Construye un campo aleatorio de **Markov** (crea un grafo).
- Aplica una optimización mediante **cortes de grafos** para lograr la segmentación.

3.4.3.1 Gaussian Mixture Model (GMM)

Lo primero que debemos entender antes de explicar el algoritmo **GrabCut** es el Algoritmo de mezclas gaussianas o **Gaussian Mixture Model (GMM)**. No vamos a profundizar mucho en esto, pero con una visión superficial será suficiente para poder utilizarlo en el proyecto. Se comenzará por explicar que es el **clustering**, lo cual consiste en la división de un conjunto de datos en grupos con características similares. El GMM descompone los datos en varias campanas gaussianas correspondientes a los clusteres obtenidos de las observaciones.

3.4.3.2 Graph cut

En este apartado se explicará el proceso de **corte de grafos**. Se parte de un grafo con múltiples conexiones entre sus elementos y se pretende cortar esas conexiones para obtener dos grafos totalmente desconectados que corresponderían con los segmentos de la imagen.

El coste del corte es la suma de los costes de cada segmento que rompiste y el algoritmo busca conseguir su objetivo con el mínimo coste.

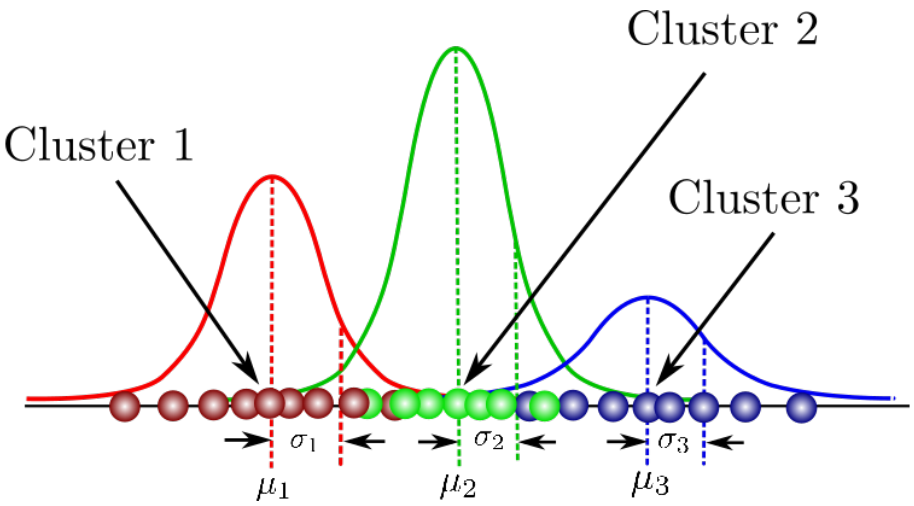


Figura 3.19 Ejemplo GMM con 3 cluster.

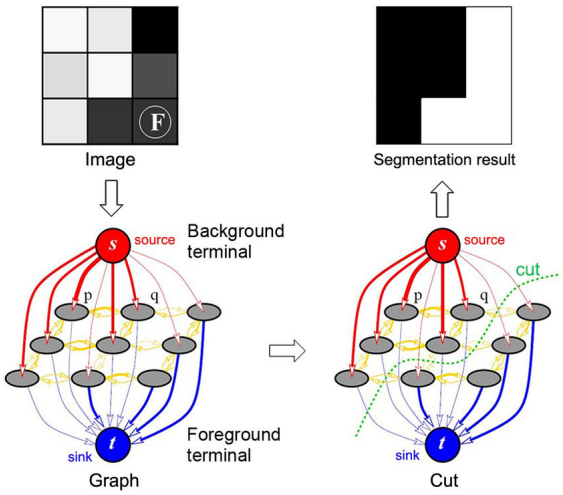


Figura 3.20 Ejemplo algoritmo Grabcut.

3.5 Tracking

El **tracking** consiste en seguir objetos que se encuentren en el campo visual de la cámara. A diferencia de las técnicas explicadas anteriormente que se aplican a imágenes, esta técnica se utiliza en vídeo.

El tracking tiene dos elementos muy importantes:

- **Identificación**:Consiste en encontrar el objeto que queremos seguir en un frame del vídeo. Es mucho más fácil seguir un objeto conocido que uno desconocido.
- **Modelado**:Un buen modelado del sistema de Tracking permite mitigar gran parte del ruido que suele estar asociado a estas técnicas.

3.5.1 Camshift (Continuously Adaptive Mean Shift)

Camshift es un algoritmo para el seguimiento de objetos basados en el **histograma** de color en HSV.

La única interacción necesaria con el usuario para que funcione este algoritmo es indicarle una **ventana inicial** que contenga el objeto que queremos seguir. El primer paso es crear el **histograma**

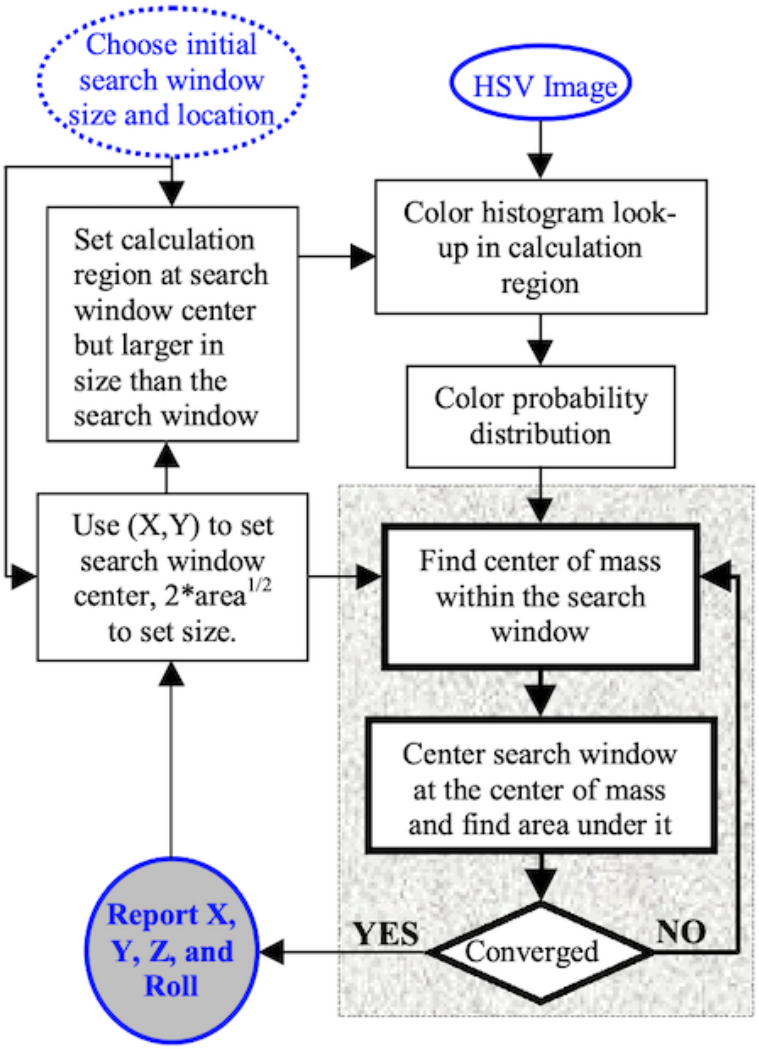


Figura 3.21 Diagrama de bloques del algoritmo Camshift(Bradski).

del objeto a seguir que se encuentra en la **ventana inicial** de la cual debemos indicar su tamaño y localización

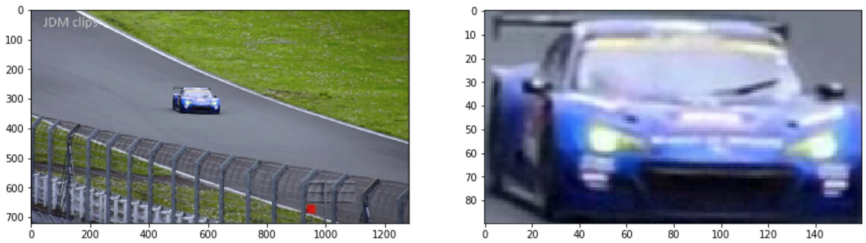


Figura 3.22 Imagen original(izquierda); ventana inicial(derecha).

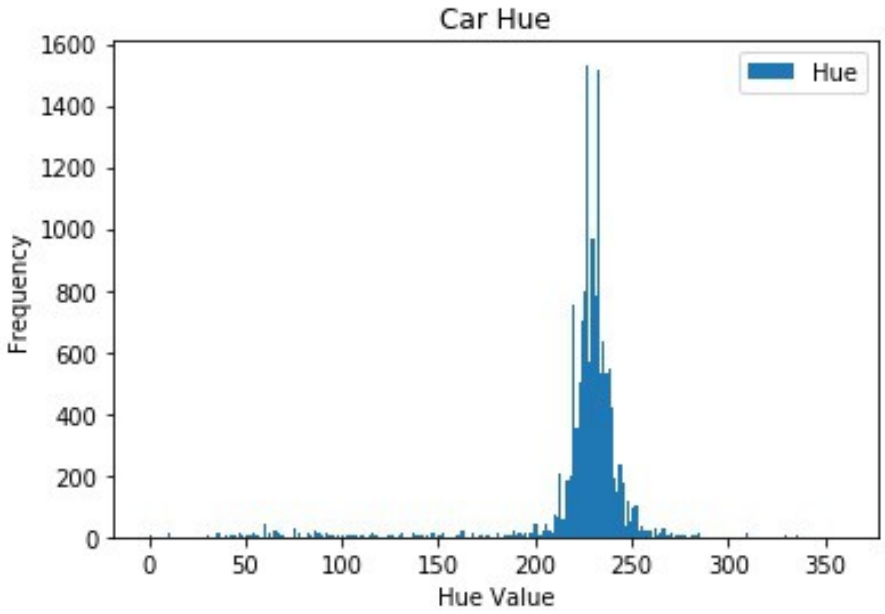


Figura 3.23 Histograma coche azul.

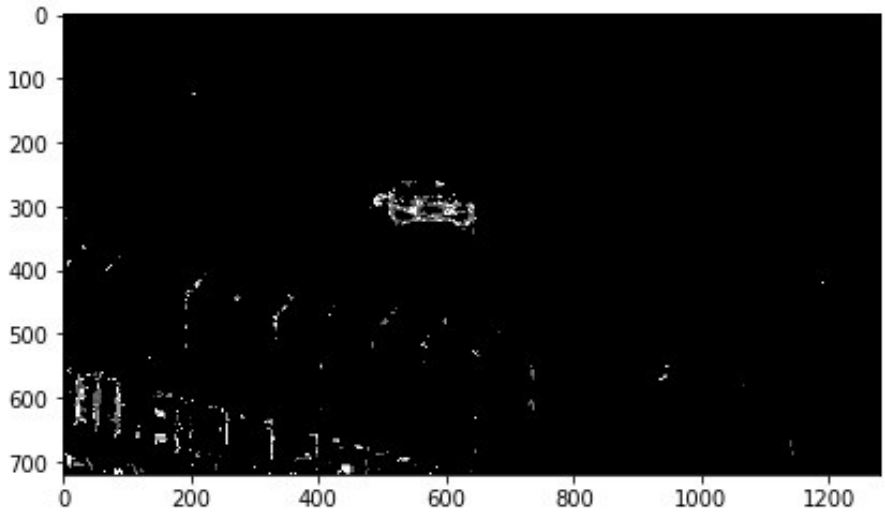


Figura 3.24 Imagen de probabilidad o "back project".

El algoritmo utiliza este histograma para convertir cada frame a su correspondiente imagen de probabilidad. Esto se consigue normalizando los valores del histograma entre 0 y 1 y se interpretarán estos valores como probabilidad.

Una vez se han convertido los frames, se mueve la ventana al área de máxima densidad por pixel de la distribución de probabilidad, para ello se usará el algoritmo **mean shift**. El nuevo centroide de la ventana es calculado con los **momentos** de la imagen con las ecuaciones de la figura 3.25

El algoritmo **mean shift** se repetirá sobre los nuevos **centroides** encontrados hasta que converja.

Camshift es una mejora del algoritmo mean shift, en el cual la ventana es siempre del mismo tamaño y orientación. Camshift adapta estos parámetros a la nueva posición del objeto. El tamaño de la ventana de búsqueda se calculará con la siguiente fórmula $s = 2\sqrt{\frac{M_00}{256}}$.

Find the zeroth moment

$$M_{00} = \sum_x \sum_y I(x, y).$$

Find the first moment for x and y

$$M_{10} = \sum_x \sum_y xI(x, y);\ M_{01} = \sum_x \sum_y yI(x, y).$$

Then the mean search window location (the centroid) is

$$x_c = \frac{M_{10}}{M_{00}};\quad y_c = \frac{M_{01}}{M_{00}};$$

Figura 3.25 Ecuaciones momentos *mean shift*. Bradski .

3.6 Filtro complementario

El **Filtro complementario** es ampliamente usado a la hora de incrementar la precisión de las medidas de **ángulos** proprocionados por un **acelerómetro** y un **giroscopio** como los que tiene integrado nuestro dispositivo. Existen otros filtros ampliamente conocidos como el filtro de Kalman que dan resultados excelentes, aunque su complejidad matemática y su excesivo coste computacional hace que en muchos casos sea más útil utilizar el filtro antes mencionado.

Los **giroscopios** suelen estar diseñados para medir la velocidad angular en una rotación. La medición debe estar integrada con respecto al tiempo. Esta integración provoca desviaciones en las medidas debido a la presencia de errores de sesgo, incluso si son pequeños.

Los **acelerómetros** están diseñados para medir las aceleraciones provocadas por las fuerzas exteriores. El ángulo absoluto se puede obtener debido a la fuerza provocada por la gravedad, que siempre será perpendicular al suelo, aunque las fuerzas de traslación provocan errores en las medidas.

El **Filtro complementario** es una solución simple y eficaz que utiliza los resultados de ambos sensores y los filtra buscando utilizar los puntos fuertes de cada sensor.

La estimación del ángulo se obtiene de la suma de las medidas como se muestra en la figura 3.26. Debido a que el giroscopio tiene mejor rendimiento a frecuencias altas, se le aplicará a su medida un filtro de paso alto. El acelerómetro tiene un mejor rendimiento a frecuencias bajas, por lo que se le aplicará un filtro de paso bajo.

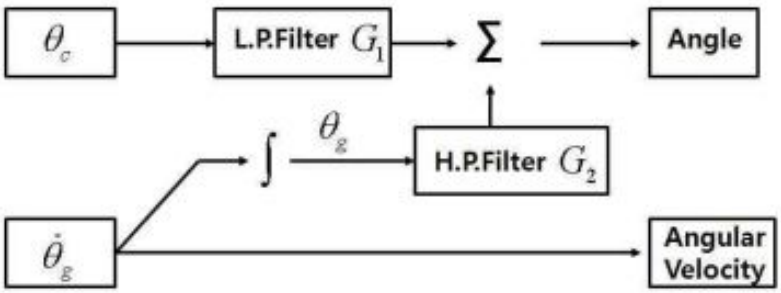


Figura 3.26 Filtro complementario (Min y Jeung) .

4 Diseño de la solución

En este capítulo desarrollaremos los algoritmos y métodos utilizados para lograr nuestros objetivos, así como una explicación de los puntos y funciones más importantes del código.

4.1 Software

Debido a que se trabajará con un sistema en **tiempo real** es muy importante la velocidad de ejecución del programa. El código se ha desarrollado en una máquina virtual con el sistema operativo(SO) **UBUNTU**. Se ha escogido Ubuntu porque es muy sólido, es mucho más difícil que se produzcan errores o fallos en este SO que en otros más populares como WINDOWS. El lenguaje de código escogido es **C++**, ya que en cuanto a velocidad, no tiene competidor. Existen otros lenguajes ampliamente utilizados y menos complejos como python, pero su rendimiento es menor. Se ha utilizado la librería **Opencv** y el software **CMake**.

4.1.1 Opencv

Opencv es una librería de visión artificial de código abierto que está escrita en **C++**. Esta librería se usa principalmente en aplicaciones en tiempo real debido a su alta **eficiencia** computacional. **Opencv** contiene más de 500 funciones capaces de realizar prácticamente cualquier algoritmo de visión, incluyendo incluso una librería para Machine Learning.

Intel proporciona a los productos de la familia **D400 Intel® RealSenseTM** la librería **librealsense** que contiene rutinas optimizadas de bajo nivel para facilitar el uso de sus cámaras. Además están disponibles una amplia documentación con ejemplos y explicaciones de como implementar **opencv** y **librealsense** en sus productos.

4.1.2 CMake

CMake es un software libre y de código abierto multiplataforma para gestionar la automatización de la construcción del software utilizando un método independiente del compilador. El proceso de construcción de **CMake** tiene dos etapas:

- Creación de los archivos de construcción estándar a partir de los archivos de configuración.
- Construcción real a través de las herramientas de construcción nativas.

Cada proyecto de construcción contiene un archivo **CMakeLists.txt**, en cada directorio, que controla el proceso de construcción.

4.1.3 Matlab

Se ha utilizado Matlab para el tratamiento y representación de datos.

4.2 Diagrama de flujo del sistema de visión

Se ha representado en la Figura 4.1 el diagrama de flujo del sistema de visión que será tratado en las secciones siguientes.

4.3 Umbralización por color

En este apartado se explicará todos los procedimientos realizados que han utilizado el color del marker.

4.3.1 Toma de imágenes RGB y sensor de profundidad

En el extracto mostrado en el código 4.1 se declara un **pipeline** que incluirá la cámara y los sensores. Posteriormente, se elige la configuración deseada y se inician para que empiecen a transmitir información.

Código 4.1 Toma de imágenes RGB y sensor de profundidad.

```
// Declare RealSense pipeline,
   pipeline pipe;

   // Crea una configuración
   config cfg;

   // Incluyo la configuración deseada

 cfg.enable_stream(RS2_STREAM_COLOR);
 cfg.enable_stream(RS2_STREAM_ACCEL, RS2_FORMAT_MOTION_XYZ32F);
   cfg.enable_stream(RS2_STREAM_GYRO, RS2_FORMAT_MOTION_XYZ32F);
   cfg.enable_stream(RS2_STREAM_DEPTH);
 pipe.start(cfg);
```

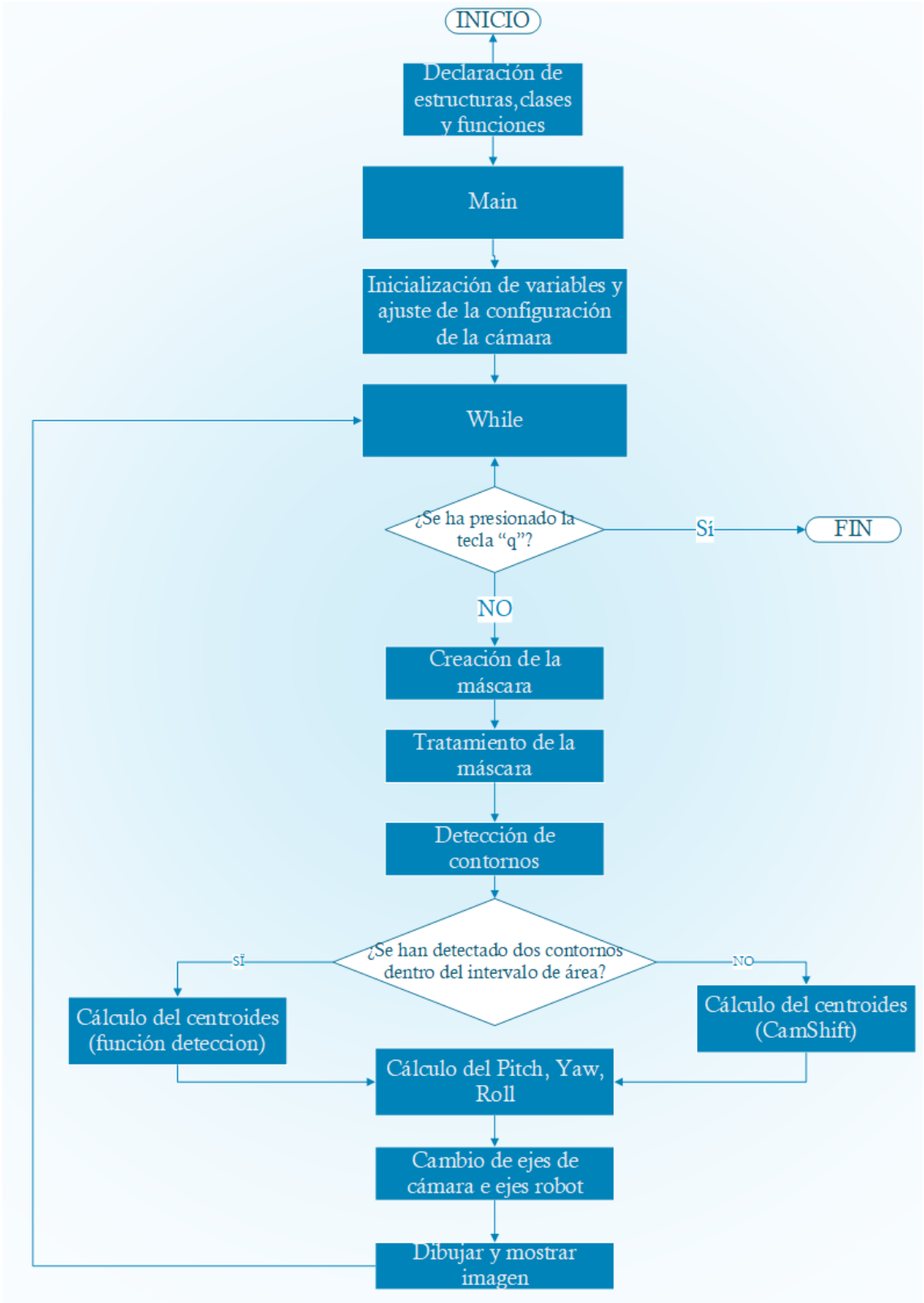


Figura 4.1 Diagrama de flujo del sistema de visión.

4.3.2 Alineación imagen RGB y profundidad

En el extracto mostrado en Código 4.2 se capturan imágenes del vídeo y se alinean la imagen RGB y los datos del sensor de profundidad. El extracto de código está basado en el ejemplo **rs-align(Intel,2019)**.

Código 4.2 Alineación imagen RGB y profundidad.

```
        rs2::align align_to(RS2_STREAM_COLOR);
        while ((getWindowProperty(window_name, WND_PROP_AUTOSIZE) >= 0)
           && (waitKey(1) != 'q'))
    {
        auto data = pipe.wait_for_frames();
        //alineo los frames
        data = align_to.process(data);

        auto color_frame = data.get_color_frame();
        auto depth_frame = data.get_depth_frame();

    .
    .
    .
      }
```

Las imágenes se obtienen guardando los frames del vídeo retransmitidos por la cámara. Estas se obtienen periódicamente mediante el uso de un bucle while.
La cámara RGB nos proporciona una matriz con los tres canales de color y el sensor de profundidad nos proporcionará una matriz con las profundidades.
Estas matrices no coinciden físicamente, es decir, un píxel de la imagen RGB no tiene su valor de profundidad correspondiente en el mismo píxel de la imagen de profundidades.
El proceso de alineación nos permite generar una **retransmisión sintética** que nos devolverá una matriz con 4 canales, los 3 canales RGB y el canal de profundidad.

4.3.3 Función máscara

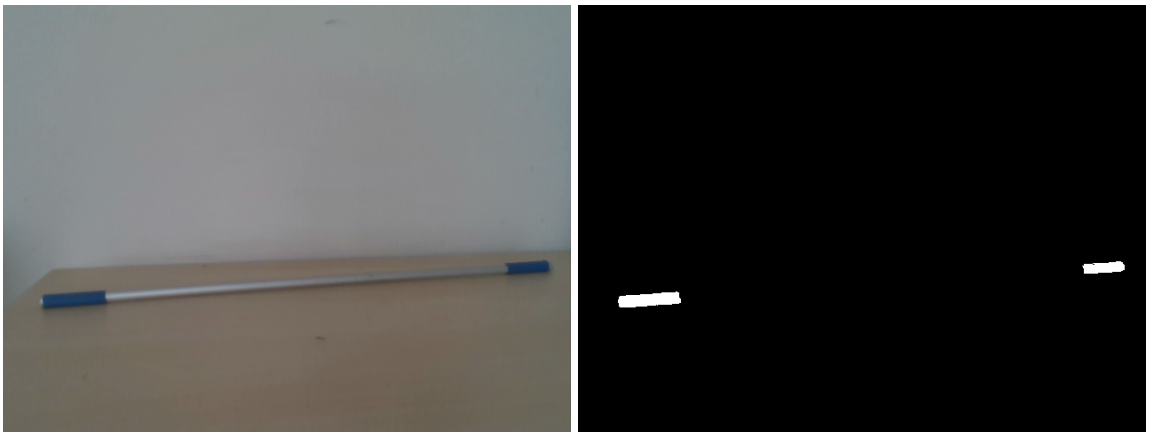
La función **máscara** recibe como parámetros de entrada la matriz de la imagen a color en espacio RGB y los umbrales y devuelve la máscara binaria. Esta función aplica los conceptos tratados en las secciones 3.3.1 y 3.4.1

Código 4.3 Función máscara.

```
Mat mascara(Mat image,int hmin,int hmax,int smin,int smax,int vmin,int
    vmax )
{
  Mat HSV;
  Mat masc;
  cvtColor(image,HSV,COLOR_BGR2HSV);//convertir a HSV
  Scalar lower(hmin,smin,vmin);// Matiz(hue),saturacion minima,valor
     minimo
  Scalar upper(hmax,smax,vmax);
```

```
  inRange(HSV,lower,upper,masc);//te hace la mascara
  return masc;
}
```

Esta función convierte la imagen a espacio HSV con la función **cvtColor** y aplica los umbrales con la función **inRange**. Los píxeles que se encuentren entre los umbrales se les asignará el valor de 255 (blanco) y los que estén fuera se les asignará el valor 0 (negro). Se han usado barras deslizantes para cambiar los valores de los umbrales en tiempo real que se pueden ver en la figura 4.3. De esta forma no necesitamos saber los valores exactos del hue, la saturación o el value, ya que se pueden obtener experimentalmente. Las ventajas de usar el espacio de color HSV se hacen patentes en esta función debido a que el valor del hue será siempre igual para el mismo objeto, debido a que es independiente de la iluminación. Se ha comprobado experimentalmente que el valor del value tiene muy poca influencia en la umbralización por lo que no será necesario modificarlo en prácticamente ningún caso. El elemento más sensible a la iluminación es la saturación, la cual si suele ser necesario ajustarla, aunque con una saturación entre 60 y 255 suele dar buenos resultados en la amplia mayoría de los casos.



(a) Imagen RGB. **(b)** Máscara binarizada.

Figura 4.2 Máscara de objeto con puntas azules.

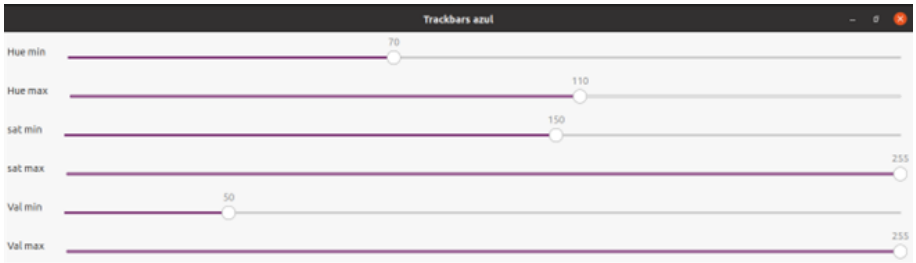


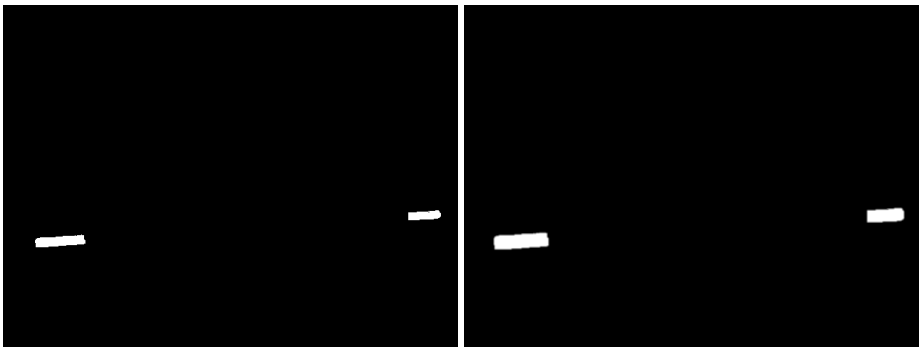
Figura 4.3 Barras deslizantes.

4.3.4 Función procesar imagen

La función **procesar imagen** recibe como argumento la máscara obtenida por la función anterior y devuelve otra matriz procesada. Esta función utiliza los conceptos explicados en las secciones

3.3.3 y 3.3.2.

Procesar imagen utiliza la función **blur** la cual filtra ruido mediante una operación de suavizado. Luego aplica una acreción y una erosión en la misma imagen. Los kernel se han escogido de forma experimental. Cuanto más grande es el kernel más agresivo será el procesado.



(a) Máscara sin procesar. **(b)** Máscara procesada.

Figura 4.4 procesamiento de la máscara de un objeto con puntas azules.

4.3.5 Función detección

La Función detección recibe como argumento la imagen binarizada procesada, la imagen en RGB y la matriz de profundidades y devuelve una estructura donde guardará las coordenadas del centroide de la pieza. Esta función se basa en los conceptos explicados en las secciones 3.3.3 y 3.4.2.

Código 4.4 Función detección.

```
struct coord deteccion(Mat procesada,Mat image,rs2::depth_frame
    depth_frame)
{
//ordenar
struct contour_sorter // 'less' for contours
{
    bool operator ()( const vector<Point>& a, const vector<Point> & b )
    {
        Rect ra(boundingRect(a));
        Rect rb(boundingRect(b));
        return ( (ra.x ) < (rb.x ) );
    }
};
.
.
.
  findContours(procesada,contours,RETR_EXTERNAL,CHAIN_APPROX_SIMPLE);//
      encuentra contornos,aproximacion simple para ahorrar puntos
    if(contours.size()==2)//solo opera cuando ha detecto 2 objetos o
        menos
    {
```

```
    sort(contours.begin(), contours.end(), sorter);

  .
  .
  .

    for(int i=0;i<contours.size();i++) //recorro contorno
    {
        int area=contourArea(contours[i]);//calculo areas de los
            contornos



      if(area>Amin && area<Amax) //elimino posibles ruidos y errores
      {


            .
            .
            .
```

En el Código 4.4 podemos ver como se utiliza la función **findContours** la cual, al proporcionarle la máscara, nos devuelve un vector que contiene los contornos encontrados. **FindContours** funciona hallando una curva continua que une todos los puntos a lo largo de la frontera del objeto. El tercer argumento de la función indica el método de aproximación. Si se le pasa el argumento `CHAIN_APPROX_NONE` se guardarán todos los puntos fronteras, en cambio, si se le pasa el argumento `CHAIN_APPROX_SIMPLE` guardará solo los puntos estrictamente necesarios para definir el contorno. Se ha escogido esta última opción, ya que ahorra mucha memoria. Para ordenar los contornos se ha utilizado la función **sort**. Anteriormente, se ha declarado la estructura `contour_sorter` la cual contiene la condición que utilizará sort para llevar a cabo su cometido. La condición es almacenar primero los contornos que tengan una coordenada x menor, por lo que el marker izquierdo se guardará primero y luego el derecho.

La función detección tiene dos condiciones para actuar:

- *Detectar dos objetos*: Debido a que el robot tiene un marker en cada muñeca, si detecta un número mayor o menor a dos objetos significará que existe un error.
- *Filtrado por área*: Debido a que conocemos la longitud de los brazos y el tamaño del objeto, sabemos de manera experimental el rango de valores de área en el que se encuentra nuestro objeto. Eliminar las áreas muy pequeñas es esencial para eliminar ruido.

En el código Código 4.5 se ha utilizado la función **boundingRect** la cual crea una estructura del tipo *Rect* donde guardará los contornos encontrados. Posteriormente, se calcula los momentos de cada contorno con la función *moments* y con esos momentos se calcula el centroide. Una vez se han hallado los contornos se calcula su distancia llamando a la función `depth_frame.get_distance`. Esta función nos devuelve el valor de la matriz de profundidad de un píxel de la imagen.

Código 4.5 Función detección.

```
boun[i]=boundingRect(contours[i]);  //creo estructura con cada contorno
    detectado
```

```
 //calculo del centroide.
mu[i] = moments( contours[i] );//calculo momentos
 mc[i] = Point2f( static_cast<float>(mu[i].m10 / (mu[i].m00 +1e-5)),
     static_cast<float>(mu[i].m01 / (mu[i].m00 + 1e-5)) );//calculo
     centroide


//distancia al centroide
dist=depth_frame.get_distance(mc[i].x,mc[i].y);
```

La Función detección finalmente guardará los píxeles x e y junto con su valor de profundidad y dibujará el **centroide** y la **Bounding box del objeto**. Podemos ver el resultado en la figura 4.5.

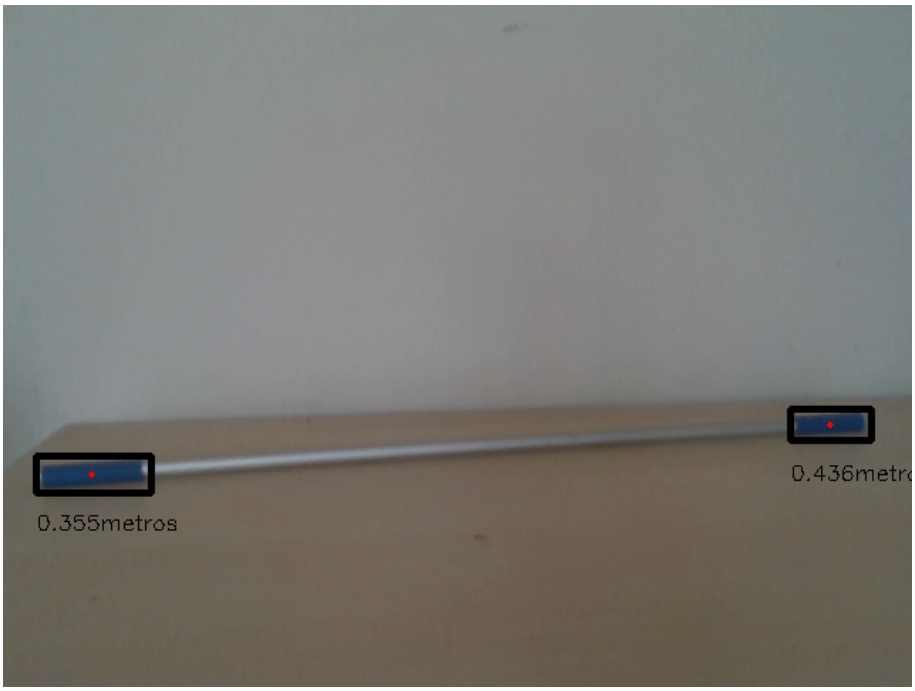


Figura 4.5 Representación del centroide, Bounding box y distancia a la cámara objeto azul.

4.4 Filtrado por profundidad

Para eliminar el fondo se ha usado el algoritmo tratado en la seccion 3.4.3. La función `depth_filter` recibe como argumentos de entrada la imagen de la cámara, la matriz de profundidades y la profundidad que se desea filtrar. Esta devuelve la imagen con el fondo retirado.

El Código 4.5 está basado en el ejemplo rs-grabcuts(Intel,2019).

Código 4.6 Función detección.

```
 Mat depth_filter(Mat image,rs2::depth_frame depth_frame,float max_depth
     )
{
   Mat bgModel,fgModel,mask;
```

```
    Mat depth_mat = depth_frame_to_meters(depth_frame);//matriz de
        profundidad
    Mat BG=Mat::zeros(depth_mat.size(),CV_8UC1);//matriz de ceros
    Mat FG=Mat::zeros(depth_mat.size(),CV_8UC1);//matriz de ceros
    BG.setTo(255,depth_mat>max_depth);//si esta mas lejos de la distancia
         maxima 0
    FG.setTo(255,depth_mat<max_depth);//si esta mas cerca de la distancia
         maxima 1

    ////////////
    /////creo matriz de la forma de la funcion grabCut
     mask.create(depth_mat.size(), CV_8UC1);
     mask.setTo(Scalar::all(GC_BGD)); // Asigna a todos los píxeles como
         fondo
     mask.setTo(GC_PR_BGD, BG ==0); // Asigna a los píxeles de la región
         lejana como "probablemente fondo"
     mask.setTo(GC_FGD, FG == 255); // Asigna a los píxeles de la región
         cercana como foreground
     //////
     grabCut(image, mask, Rect(), bgModel, fgModel, 1, GC_INIT_WITH_MASK)
         ;
     //Extrae el fondo a partir de la máscara proporcionada por el
         algoritmo
     Mat3b foreground = Mat3b::zeros(image.rows, image.cols);
     image.copyTo(foreground, (mask == GC_FGD) | (mask == GC_PR_FGD));

     return foreground;
}
```

La función realiza lo siguiente:

- **Generar la máscara "cercana" y la máscara "lejana"**: Previamente, se ha escogido una distancia que diferencia el fondo(background) de la zona que nos interesa(foreground). Debido a que se conocen las dimensiones de los brazos robóticos, sabemos a partir de que distancia los píxeles pertenecerán al background. La máscara cercana contendrá los píxeles que se encuentren a una distancia menor que la establecida, dándole un valor de 255(blanco) a estos y de 0 (negro) al resto. La máscara lejana contendrá los píxeles que se encuentren a una distancia mayor que la establecida, dándole un valor de 255(blanco) a estos y de 0 (negro) al resto.
- **Llamada a la función GrabCut**: Se combinan ambas máscaras y se le da a la función como argumento. Esta función genera una máscara refinada de la cual solo guardamos los píxeles que el algortimo considera foregorund o posible foreground dejando el resto de píxeles a 0.

En la figura 4.6 se muestran los resultados del algoritmo, una vez eliminada la pared de la imagen y mostrando solo los píxeles que nos interesan. En las figuras 4.7 se observa el filtro utilizado en la detección de los marker y el objeto azul apoyado en la mesa. En la figura 4.8 podemos observar el filtro utilizado en el robot.

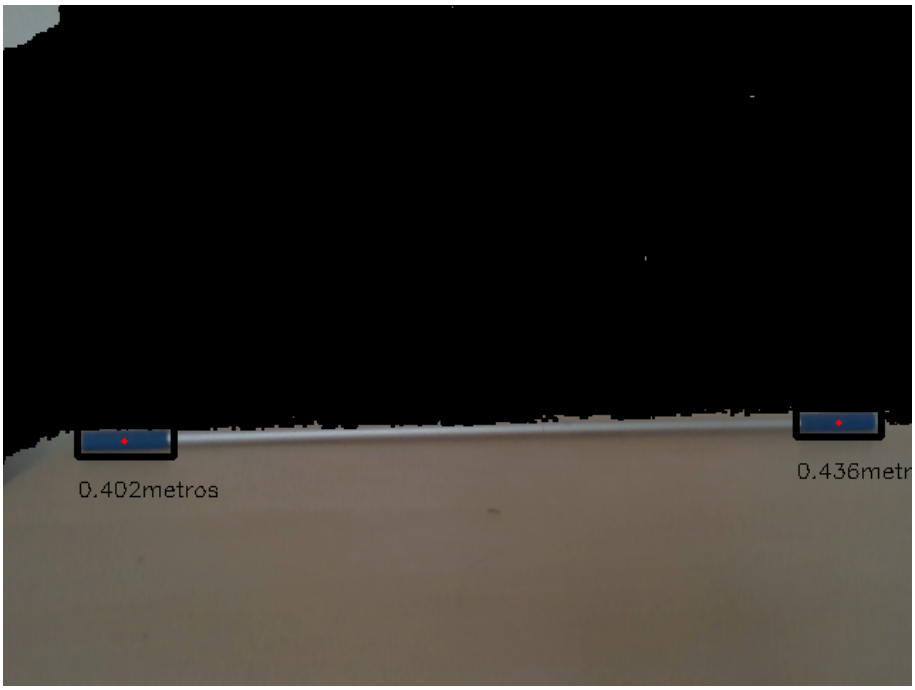
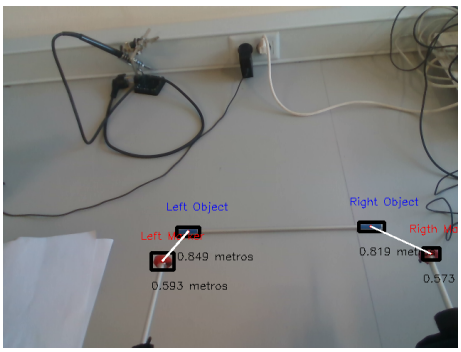
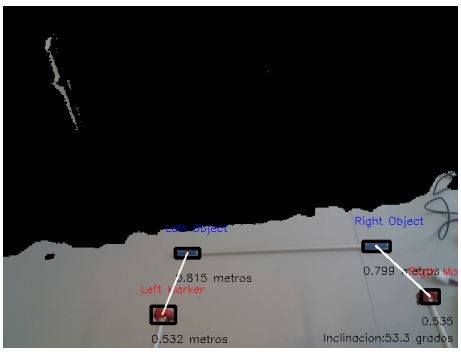


Figura 4.6 Imagen con máscara del algoritmo GrubCut aplicada.



(a) Filtro de profundidad desactivado.



(b) Filtro de profundidad activado.

Figura 4.7 Detección del marker rojo y objeto azul en el robot .

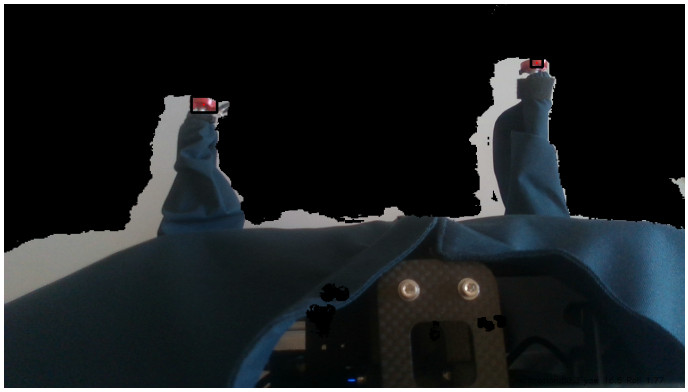
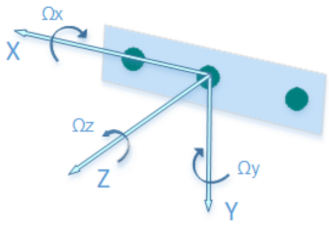


Figura 4.8 Detección del marker rojo con filtro de profundidad.

4.5 Cálculo del Pitch, Yaw y Roll

El Pitch, Yaw y Roll traducidos al español serían Balanceo, cabeceo y guiñada respectivamente. La cámara puede moverse en tres dimensiones del espacio, lo cual hace su movimiento bastante complejo. El Pitch se definen como el giro alrededor del eje X, el Yaw alrededor del eje Y, y el

The resulted orientation angles and acceleration vectors share the coordinate system with the depth sensor.



1. The positive x-axis points to the right.
2. The positive y-axis points down.
3. The positive z-axis points forward

Figura 4.9 Representación de los ejes de coordenadas de la cámara .

Roll alrededor del eje Z. Su cálculo es muy importante, ya que nos proporcionará los parámetros extrínsecos de la cámara. Esto nos permitirá transformar los puntos expresados en los ejes de coordenadas-cámara a los ejes de coordenadas- brazo. En la Figura 4.9 podemos observar los ejes de coordenadas de la cámara Intelrealsense D435i. Ω_X,Ω_Y,Ω_Z representan el Pitch, Yaw y Roll respectivamente.

El fragmento de código que se encarga de calcular el ángulo está basado en el ejemplo rs-motion(Intel,2020).

Se ha creado la clase `rotation_estimator` la cual calcula el ángulo de rotación mediante los datos obtenidos por el giroscopio y el acelerómetro que están integrados en el dispositivo. Este ángulo se guardará en el vector *theta*. Debido a que tanto el acelerómetro como el giroscopio querrán actualizar la variable **theta** será necesario el uso de un **mutex**. La información obtenida por el acelerómetro y el giroscopio será filtrada por el **Complementary Filter** explicado en la sección 3.6.

Se combinarán las medidas de ambos sensores con el parámetro **alpha** que tendrá un valor entre 0 y 1. Este parámetro pondera la influencia del giroscopio y el acelerómetro en la medida. Cuanto más cercano a 1 sea alpha más influencia tendrá el giroscopio y menos el acelerómetro, y cuanto más cercano sea a 0 será al contrario.

```
float3 theta; // theta es el angulo de rotacion de la cámara en x,y,z.
std::mutex theta_mtx;
float alpha = 0.98;
bool first = true; // primera iteración
double last_ts_gyro = 0;
```

La función descrita en el código 4.7 calcula el cambio en el ángulo de rotación con la información proporcionada por el giroscopio. Recibe como argumento las medidas del giroscopio y la medida de tiempo actual. Se multiplica las medidas del giroscopio por el tiempo transcurrido desde la última medida para así calcular el cambio en la dirección del movimiento. Finalmente, bloquea el mutex

para sumar o restar el cambio del ángulo en **theta**. La posición inicial está proporcionada por el acelerómetro, ya que el giroscopio solo puede detectar cambios en la velocidad angular y le es imposible hallar por él mismo un ángulo inicial. Debido a esto se ignorarán las medidas tomadas por el giroscopio hasta que el acelerómetro proporcione un ángulo inicial.

Código 4.7 process-gyro.

```
void process_gyro(rs2_vector gyro_data, double ts)
{
    if (first) //primera iteracion,solo se usa datos del acelerómetro
    {
        last_ts_gyro = ts;
        return;
    }
    // guarda el angulo calculado por el girsocopio
    float3 gyro_angle;

     // Inicializa gyro_angle con la información del giroscopio
    gyro_angle.x = gyro_data.x; // Pitch
    gyro_angle.y = gyro_data.y; // Yaw
    gyro_angle.z = gyro_data.z; // Roll


    double dt_gyro = (ts - last_ts_gyro) / 1000.0;
    last_ts_gyro = ts;

    //cambio en el angulo
    gyro_angle = gyro_angle * dt_gyro;

    // guardar en theta
    std::lock_guard<std::mutex> lock(theta_mtx);
    theta.add(-gyro_angle.z, -gyro_angle.y, gyro_angle.x);
}
```

La función `process_accel` descrita en los fragmentos de código 4.8 y 4.9 recibe como argumento la información del acelerómetro y actualiza el ángulo theta. Los ángulos en los ejes Z y X se calculan utilizando funciones trigonométricas. El movimiento alrededor del eje Y (perpendicular al suelo) nunca puede ser estimado inicialmente por un acelerómetro, por lo que se le dará un valor inicial arbitrario.

Código 4.8 process-accel.

```
void process_accel(rs2_vector accel_data)
{
    // Holds the angle as calculated from accelerometer data
    float3 accel_angle;
    // Calculate rotation angle from accelerometer data
    accel_angle.z = atan2(accel_data.y, accel_data.z);
    accel_angle.x = atan2(accel_data.x, sqrt(accel_data.y * accel_data.y
        + accel_data.z * accel_data.z));
```

```
// If it is the first iteration, set initial pose of camera
    according to accelerometer data
// (note the different handling for Y axis)
std::lock_guard<std::mutex> lock(theta_mtx);
if (first)
{
    first = false;
    theta = accel_angle;
    // Since we can't infer the angle around Y axis using
        accelerometer data, we'll use PI as a convetion
    // for the initial pose
    theta.y = PI;
}
```

Después de la primera iteración se usará una versión aproximada del **Complementary Filter**. Se utilizará este filtro para solucionar dos problemas:

- **Giroscopio**: No retorna a cero cuando el sistema vuelve a su posición inicial.
- **Acelerómetro**: Mucho más sensible al ruido que el giroscopio.

Para lidiar con estos inconvenientes se aplicarán dos procesos:

- **Filtro de paso alto (theta * alpha)**: Filtra las señales que se mantienen en el tiempo y deja pasar las de corta duración. Busca mejorar la medida del giroscopio.
- **Filtro de paso bajo (accel * (1- alpha))**: Filtra las señales de corta duración (que seguramente tengan mucho ruido) y deja pasar las de larga duración. Busca mejorar la medida del acelerómetro.

Este filtro se usará para ponderar las medidas tomadas por el acelerómetro y el giroscopio en el eje X y en el eje Z. En el eje Y no se ponderará, ya que solo la calcula el giroscopio.

Código 4.9 process-accel.

```
else
      {
          /*
          Apply Complementary Filter:
              - "high-pass filter" = theta * alpha: allows short-
                duration signals to pass through while
                filtering out signals that are steady over time, is
                   used to cancel out drift.
              - "low-pass filter" = accel * (1- alpha): lets through
                long term changes, filtering out short
                term fluctuations
          */
          theta.x = theta.x * alpha + accel_angle.x * (1 - alpha);
          theta.z = theta.z * alpha + accel_angle.z * (1 - alpha);
      }
  }
```

El fragmento de código 4.10 nos indica cómo utiliza el main la clase `rotation_estimator` Se obtienen los datos proporcionados por los sensores y se les pasa como argumentos a las funciones. Finalmente solo será necesario acceder al mutex para poder trabajar con el ángulo.

Código 4.10 process-accel.

```
//calculo de los angulos

   //acelerometro
      auto fa = data.first(RS2_STREAM_ACCEL, RS2_FORMAT_MOTION_XYZ32F);

      rs2::motion_frame accel = fa.as<rs2::motion_frame>();
     //giroscopio
      auto fg = data.first(RS2_STREAM_GYRO, RS2_FORMAT_MOTION_XYZ32F);
      rs2::motion_frame gyro = fg.as<rs2::motion_frame>();
  if (gyro)
      {
    double ts = data.get_timestamp();
          rs2_vector gyro_data = gyro.get_motion_data();
    estimador.process_gyro(gyro_data, ts);

  }
  if (accel)
      {
          rs2_vector accel_data= accel.get_motion_data();
    estimador.process_accel(accel_data);
  }

      theta_rad=estimador.get_theta();//obtengo theta del mutex
```

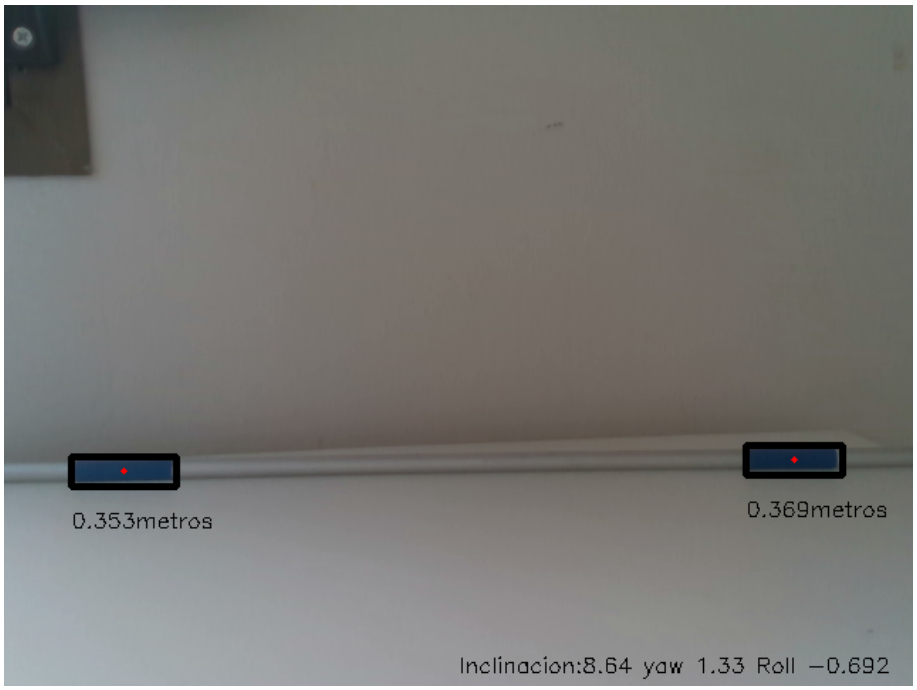


Figura 4.10 Pitch, Yaw y Roll en grados de la cámara en tiempo real.Yaw inicial=0.

En la figura 4.10 podemos ver el valor del pitch, el Yaw y el Roll en la esquina inferior derecha. Estos valores son en tiempo real, por lo que la cámara podría modificar su orientación sin ningún problema.

4.6 Camshift

El algoritmo de **Camshift** ha sido previamente tratado en la sección 3.5.1. En el código hay dos funciones que se encargan de este algoritmo para las que se ha tomado base el ejemplo **Meanshift and Camshift(Opencv)**. La función **Roi** está descrita en el código 4.11. Esta función recibe como argumentos la imagen de la cámara en RGB y una variable de tipo RECT que contiene la ventana inicial donde empezará a buscar el algoritmo. Esta ventana estará proporcionada por la última bounding box que haya encontrado la función **coordenadas**. La función **Roi** recorta la imagen a la ventana inicial, posteriormente transforma la imagen a espacio de color HSV y elimina los valores de baja saturación, que suelen provocar errores. Posteriormente calcula el histograma de la ventana y lo normaliza.

Código 4.11 ROI.

```
Mat ROI(Mat image, Rect bo)
 {

    Mat hsv_roi, mask,Roi;
    Roi = image(bo);
    cvtColor(Roi, hsv_roi, COLOR_BGR2HSV);
    inRange(hsv_roi, Scalar(0, 60, 32), Scalar(180, 255, 255), mask);
    float range_[] = {0, 180};
    const float* range[] = {range_};
    Mat roi_hist;
```

```
    int histSize[] = {180};
    int channels[] = {0};
    calcHist(&hsv_roi, 1, channels, mask, roi_hist, 1, histSize, range);
    normalize(roi_hist, roi_hist, 0, 255, NORM_MINMAX);
    return roi_hist;

 }
```

La función `Cam_shift` recibe como argumento el histograma, la imagen, la ventana de búsqueda y la matriz de profundidades. Devuelve una estructura Camshif, que hemos declarado anteriormente, conteniendo un vector tipo Rect con la ventana inicial de la siguiente iteración, y la bounding box del objeto. Podemos ver la función en el código 4.12. La función transforma la imagen a HSV y llama a la función **calcBackProject**, la cual lee el histograma que se le ha proporcionado y lo escala. El resultado de esta operación es el "back project". En términos estadísticos, calcula la probabilidad de un valor en cada elemento de la imagen completa con respecto a la probabilidad empírica de la distribución de probabilidad representada por el histograma hallado anteriormente de la ventana.

Finalmente se llama a la función **CamShift** la cual aplicará el algoritmo recursivo del camshift y nos devolverá la nueva bounding box y la siguiente ventana inicial para la próxima iteración.

Código 4.12 Cam-shift.

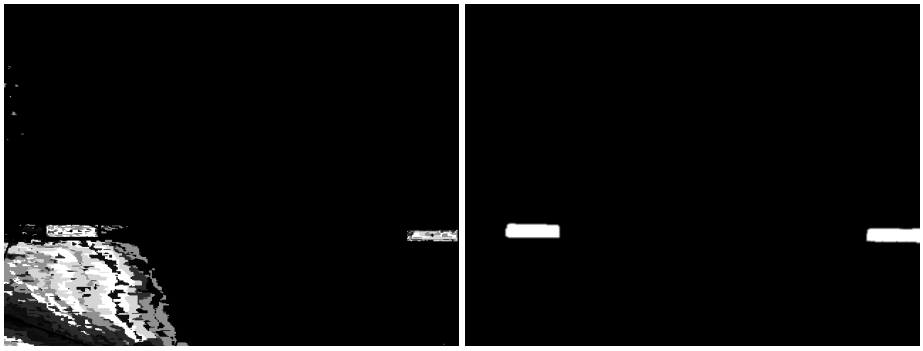
```
struct Camshif Cam_shift(Mat rui,Mat image,Rect bound,rs2::depth_frame
    depth_frame)
{
    Mat hsv,dst;
  struct Camshif Cam;
    int channels[] = {0};
    float range_[] = {0, 180};
    const float* range[] = {range_};const
    //criterio de terminación de 10 iteraciones
    TermCriteria term_crit(TermCriteria::EPS | TermCriteria::COUNT, 100,
        1);
  cvtColor(image, hsv, COLOR_BGR2HSV);
    calcBackProject(&hsv, 1, channels, rui, dst,range);
    RotatedRect rot_rect = CamShift(dst, bound, term_crit);
    // Dibujar en la imagen
    Point2f points[4];
    rot_rect.points(points);
    for (int i = 0; i < 4; i++) line(image, points[i], points[(i+1)%4],
       255, 2);
  circle(image,rot_rect.center,2,Scalar(0,0,255),FILLED);// Dibujo
      centroide
  Cam.vent_inicial=bound;
  Cam.bou=rot_rect;
    return Cam;

}
```

En la figura 4.11 se encuentra representado el back project de la pieza y la máscara obtenida por

umbralización mediante barras deslizantes. En el back project los píxeles más blancos son los que más probabilidades de pertenecer la pieza. Se observa claramente como los resultados son mucho mejores por umbralización.

Una máscara y un back project no son lo mismo. La máscara es binaria, por lo que los píxeles pueden tener un valor de 0(negro) o 255(blanco). El back project tiene valores que se encuentran en un rango entre 0 y 255. Podríamos interpretar la máscara como si fuera un back project y dárselo como argumento a la función **CamShift**. De esta forma desarrollamos un método mixto. Se le aporta al algoritmo **CamShift** la máscara calculada por umbralización, la cual interpretará como si fuera un back project. Los puntos que valen 255 son puntos que el camshift interpreta como una pieza con absoluta seguridad, y los que valen 0 los interpreta como fondo con absoluta seguridad.



(a) Back project. **(b)** Máscara hallada por umbralización.

Figura 4.11 Máscara por umbralización y Back project por Camshift.

4.7 Cambio de ejes

Hasta ahora hemos obtenido los píxeles en los que se encuentran los centroides de cada pieza y su distancia hasta la cámara, pero se buscan las coordenadas en 3 dimensiones del centroide en ejes del brazo. Por tanto, habría que transformar los puntos de ejes de cámara a ejes del brazo. La función `imagen_a_robot` se encargará de este cometido y la podemos ver en los códigos 4.13 y 4.14. Los conceptos teóricos se han explicado en la sección 3.2. `imagen_a_robot` recibe como argumento un vector que contendrá los píxeles y la distancia a la que se encuentra el centroide, los parámetros intrínsecos de la cámara, el pitch y un flag para diferenciar si es el brazo izquierdo o el derecho. Nos devolverá el punto expresado en ejes brazo.

Los parámetros intrínsecos se encuentran guardados dentro del dispositivo, ya que esta cámara viene calibrada de fábrica, por lo que no será necesario calcularlos.

En el código 4.13 se utiliza la función `rs2_deproject_pixel_to_point` la cual recibe como argumento Los parámetros intrínsecos, las coordenadas en píxeles y la distancia del centroide y nos devolverá las coordenadas del centroide expresado en ejes cámara.

Código 4.13 imagen-a-robot.

```
struct punto imagen_a_robot(float punto[3],rs2_intrinsics intrin,float
    pitch,int L)
{
  rs2_extrinsics extr;
```

```
struct punto punt;
  float point[3];
const float pixel[2]={punto[0],punto[1]};
float depth=punto[2];
float cam[3];
float arm[3];
rs2_deproject_pixel_to_point(point,&intrin, pixel,depth);
```

En el código 4.14 se transformará de ejes de coordenadas-cámara a ejes de coordenadas-robot. Se utilizará la geometría de los brazos descritos en la sección 2.1.1. En la figura 4.12 se observa los

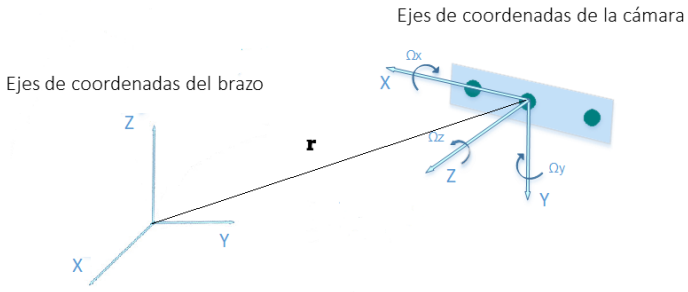


Figura 4.12 Representación de la rotación y traslación de los ejes de coordenadas del sistema.

ejes de referencia de la cámara y de los brazos, donde el vector *r* representa la translación a aplicar y Ω_x el pitch. Se transforma aplicando una traslación en el eje X de 3 cm. Una traslación en el eje Y de 18 cm en el brazo izquierdo o -18 cm para el brazo derecho. Una traslación en el eje Z de 23 cm y una rotación alrededor del eje X cuyo valor será el picth. Además la dirección y sentido de los ejes cámara han sido intercambiados para que coincidan con los ejes brazo.

Código 4.14 imagen-a-robot.

```
 //cambio de ejes///
  cam[0]=point[2]-X;
if (L==1)cam[1]=-point[0]-Y; else cam[1]=-point[0]+Y;
cam[2]=-point[1]+Z;

///
arm[0]=cam[0]*cos(pitch)+cam[2]*sin(pitch);
arm[1]=cam[1];
arm[2]=-cam[0]*sin(pitch)+cam[2]*cos(pitch);

punt.punto_XYZ[0]=arm[0];
```

```
  punt.punto_XYZ[1]=arm[1];
  punt.punto_XYZ[2]=arm[2];
  return punt;
}
```

5 Resultados experimentales

5.1 Evaluación de la precisión del sistema de visión

Con el objetivo de comprobar la **precisión** del sistema de visión a la hora de tomar medidas de distancia, se ha realizado un experimento en el que la cámara y el robot registran simultáneamente los datos de la posición de los marker obtenidos por sus sensores. El **robot** nos aportará las coordenadas calculadas mediante **la cinemática del sólido rígido**, es decir, **la medida ideal**. Debido a que durante el experimento el robot no sufrirá ninguna perturbación, se asumirá que las medidas son reales y se usará como "ground truth". De esta forma, podemos comprobar la precisión de la cámara, la cual será mayor cuanto más parecidas sean sus medidas a las del robot. Los brazos describirán una trayectoria arbitraria y el sistema de visión deberá seguirla.

En las Figuras 5.1, 5.2, 5.3, 5.4, 5.5, 5.6, podemos observar la representación gráfica de las coordenadas de los marker y su error.

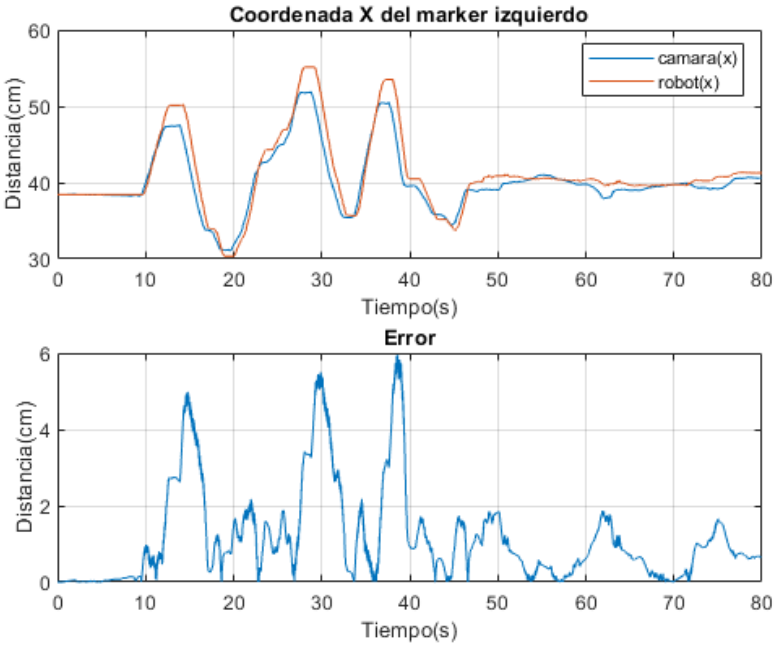


Figura 5.1 Posición y estimación del error en el eje X del marKer izquierdo .

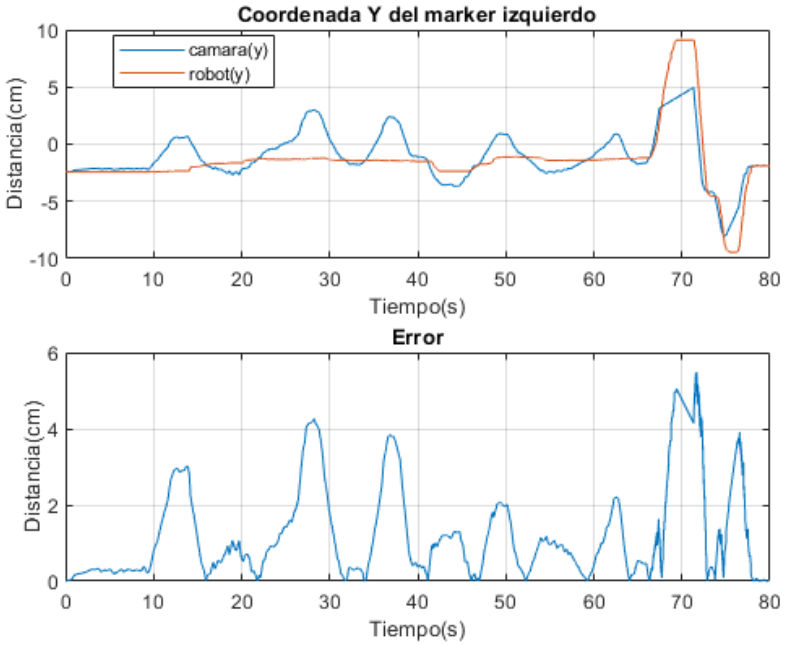


Figura 5.2 Posición y estimación del error en el eje Y del marKer izquierdo .

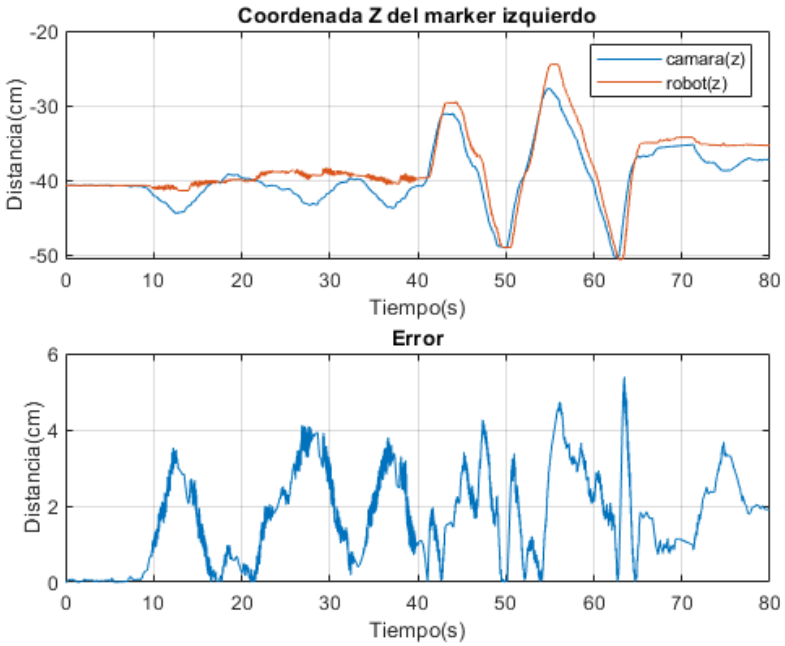


Figura 5.3 Posición y estimación del error en el eje Z del marKer izquierdo .

Se puede observar en todas las gráficas el buen rendimiento del sistema de visión cuando los brazos se encuentran **estáticos, con error inferior a 1 cm**, aunque cuando se encuentra en **movimiento** ese error puede subir hasta los **6 cm**. El error será mayor cuanto más rápido y brusco sea el movimiento.

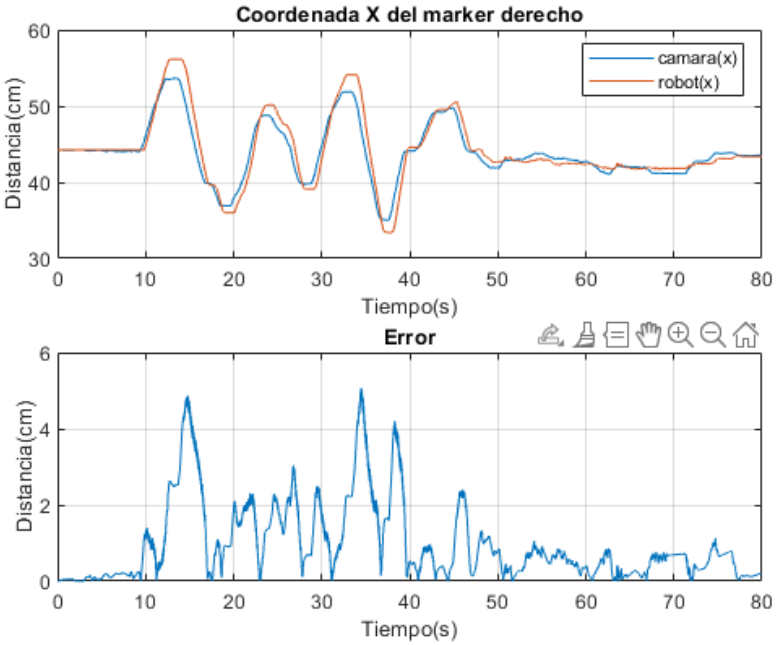


Figura 5.4 Posición y estimación del error en el eje X del marKer derecho .

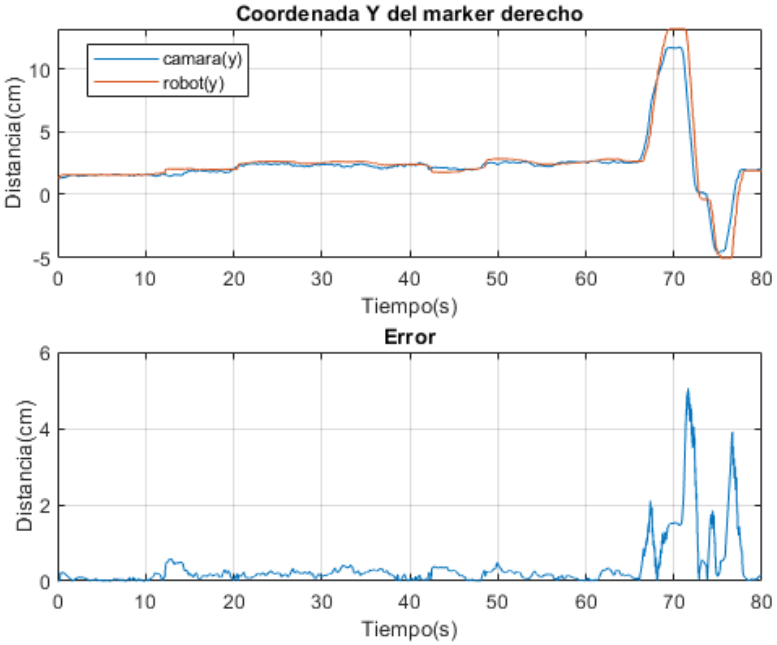


Figura 5.5 Posición y estimación del error en el eje Y del marKer derecho.

Este error se puede deber a varios factores:

- **Diferentes marcas de tiempo**: El tiempo de muestreo del robot y de la cámara son diferentes, por lo que se ha interpolado linealmente los datos de la **cámara** para poder trabajar con ambas medidas. Esta interpolación puede provocar errores. Además, el tiempo de muestreo de la cámara tiene mucha importancia en los movimientos bruscos, ya que si no es suficientemente corto, puede ser que no detecte algunos puntos.

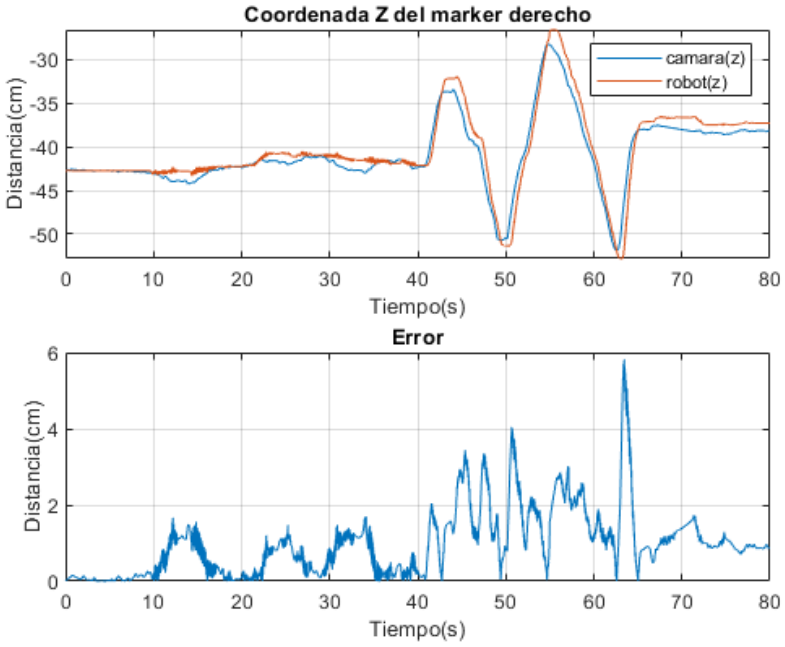


Figura 5.6 Posición y estimación del error en el eje Z del marKer derecho .

- **Eliminación de outliers**: El seguimiento de la cámara no es perfecto, por lo que existen algunos valores llamados "outliers" que se generan al fallar la detección del marker durante algunas iteraciones. La eliminación de los mismos podría alterar los valores que se obtienen cuando existen cambios bruscos, detectándolo como "outlier" y eliminado medidas correctas.
- **Deflexión**: Es también posible que las medidas del robot no sean totalmente ideales y que al realizar estos movimientos bruscos deflecte ligeramente, de forma que las medidas tomadas por la cámara no sean tan erróneas.

Existen otras posibles fuentes de error como una mala calibración de la cámara, ruido en la medición del ángulo o incluso errores en la medida de la posición relativa de la camára con respecto a los brazos, aunque no es probable que influyan mucho en el error, ya que cuando el marker está inmóvil las medidas tienen un error pequeño.

5.2 Cálculo de la deflexión

En las mismas condiciones que en el experimento anterior se ha aplicado a los marker perturbaciones para provocar una deflexión y poder medirla desde el sistema de visión. En las Figuras 5.7, 5.8, 5.9, 5.10, 5.11, 5.12, podemos observar la representación gráfica de las coordenadas de los marker y su deflexión.

Transcurridos **10 segundos** del inicio se ha aplicado una pertubación en ambos marker cuya duración es de 20 segundos. Alrededor de los **50 segundos** del inicio se ha aplicado una perturbación mayor de una duración de unos 40 segundos.

Se puede observar como vuelve a tener un muy buen rendimiento con los marker estáticos (error<0.2cm). Se aprecia en las gráficas como **el robot no es capaz de percibir esas perturbaciones**, mientras que el sistema de visión si es capaz de percibirlas y medirlas. También podemos apreciar como vuelven a coincidir las gráficas cuando la perturbación desaparece.

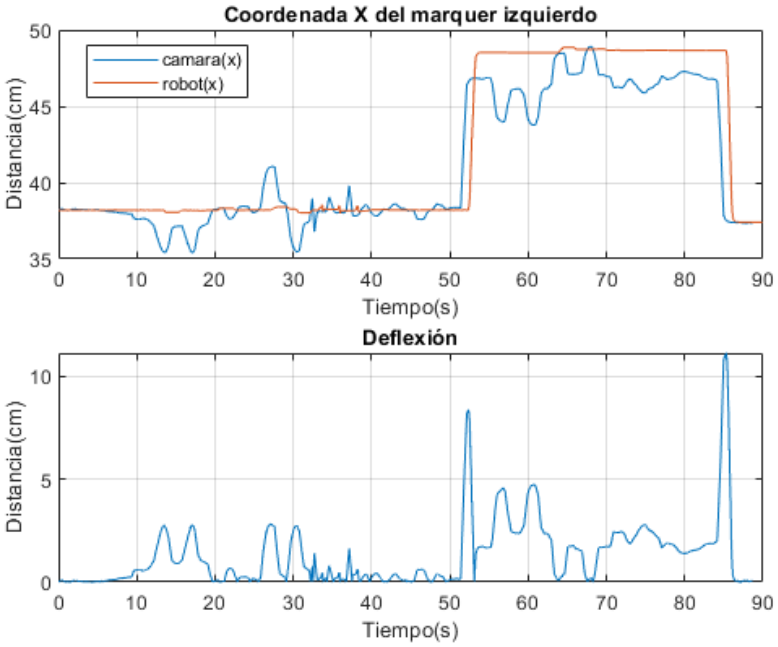


Figura 5.7 Posición y estimación de la deflexión en el eje X del marKer izquierdo .

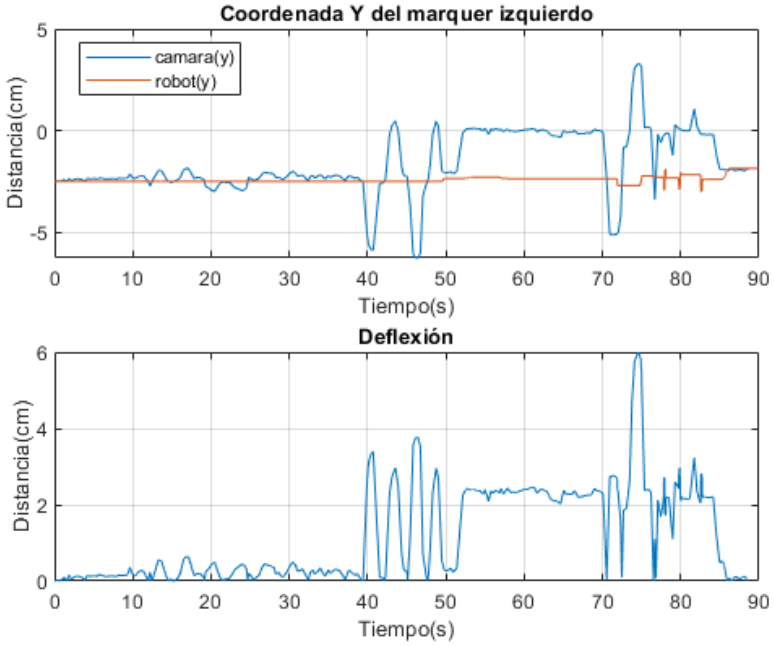


Figura 5.8 Posición y estimación de la deflexión en el eje Y del marKer izquierdo .

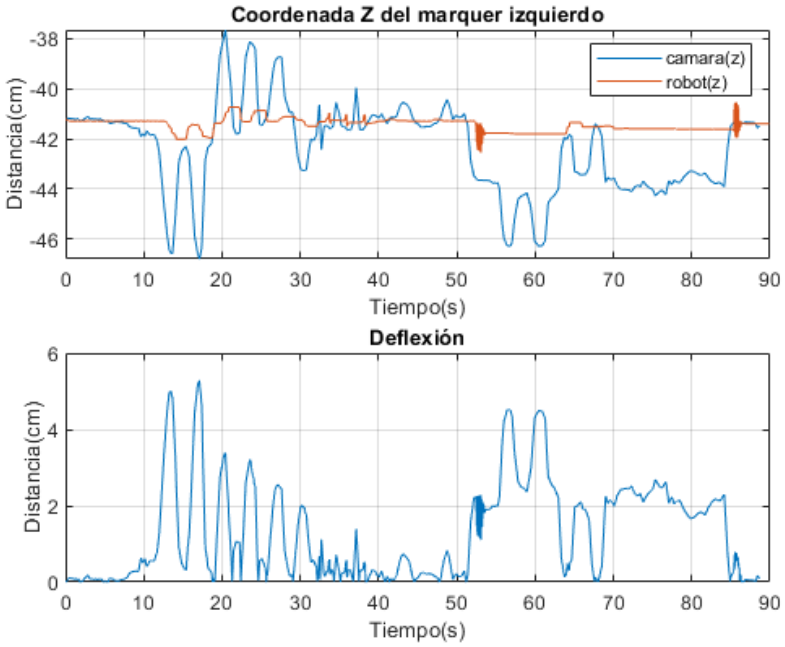


Figura 5.9 Posición y estimación de la deflexión en el eje Z del marKer izquierdo .

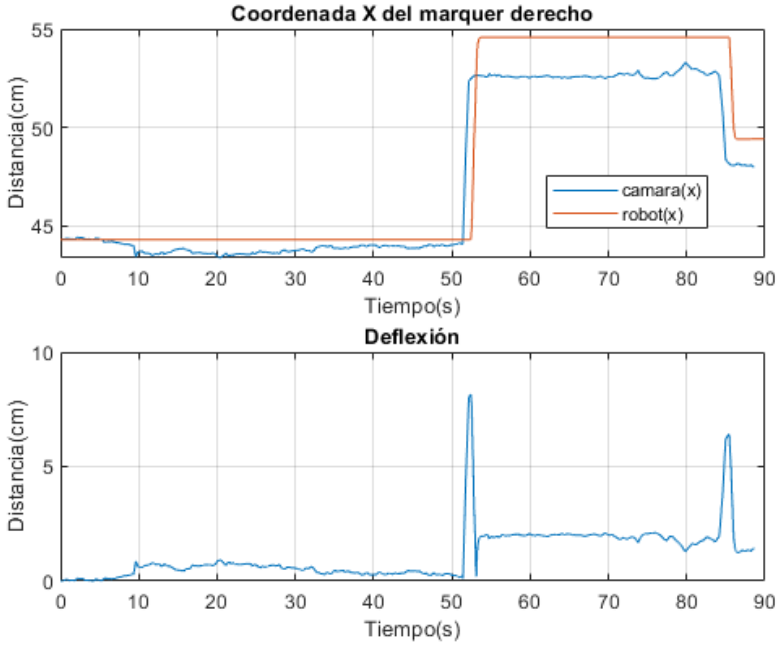


Figura 5.10 Posición y estimación de la deflexión en el eje X del marKer derecho .

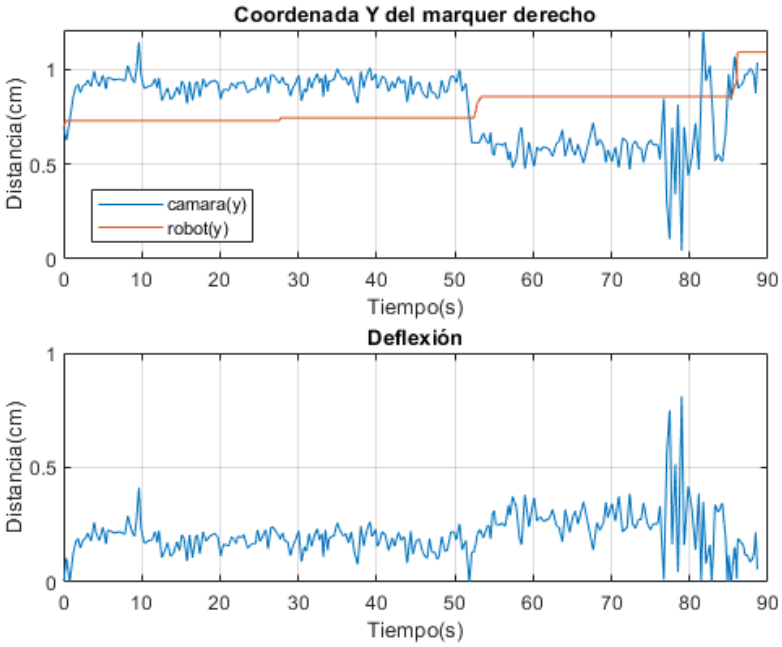


Figura 5.11 Posición y estimación de la deflexión en el eje Y del marKer derecho.

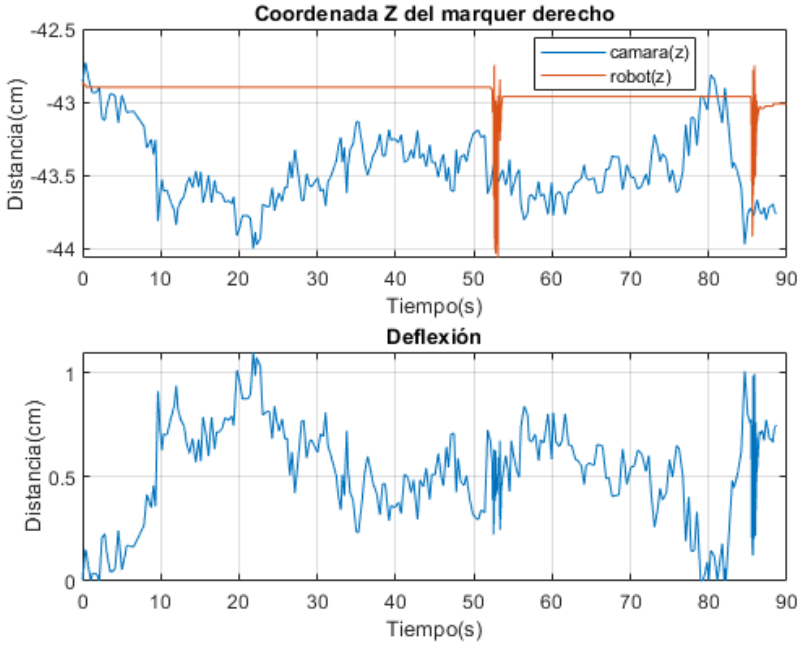


Figura 5.12 Posición y estimación de la deflexión en el eje Z del marKer derecho .

6 Conclusiones

Con los resultados obtenidos en el Capítulo 5 extraeremos algunas conclusiones. La Umbralización por colores da buenos resultados a la hora de localizar el marker, aunque requiere unas condiciones de luz bajas. Cuando la iluminación es muy fuerte, como puede ser en el exterior, es bastante probable que encuentre inconvenientes a la hora de localizar el objeto. Una gran desventaja es la presencia de objetos del mismo color y área parecida a la del marker en el campo de visión de la cámara, provocando que el sistema dejará de funcionar completamente. Esta última limitación se soluciona con el Camshift, ya que solo seguirá a los objetos que se encuentran en la ventana de seguimiento. Uniendo ambos métodos y haciendo actuar el Camshift solo cuando la umbralización falla, tenemos las ventajas de ambos algoritmos. El filtro de profundidad mejora por lo general el proceso de detección, ya que elimina píxeles de la imagen que no corresponden con la pieza. El único inconveniente de este filtro es el alto coste computacional que presenta, por lo que para ordenadores poco potentes puede ser un recurso demasiado exigente.

La precisión de la medida del marker en estático es bastante alta (error<0.2 cm). Sin embargo, cuando el marker realiza movimientos, este error se incrementa hasta valores de 6 cm. Cuanto más rápido sea el movimiento del marker, mayor será ese error. Estos resultados no han sido satisfactorios y hace inviable la medición de perturbaciones puntuales bruscas, aunque sería útil en la medición de deflexiones provocadas por el peso de un objeto, donde no existirían estos movimientos tan bruscos.

Una serie de posibles líneas de trabajo futuras con las que se podría mejorar e implementar este sistema serían:

- Rotación de la cámara: El programa desarrollado calcula el Pitch, el Yaw y el Roll de la cámara en tiempo real. En nuestro caso, la cámara estaba rotada solamente sobre el eje X, por lo que solo nos interesaba el pitch. Con algunas pequeñas modificaciones en el cálculo de los parámetros extrínsecos de la cámara, gracias a los ángulos mencionados anteriormente, se podría rotar la cámara en cualquier dirección.

- Comunicación con el robot: El programa tiene incluido un socket para intercambiar datos con el robot, aunque no se llegó a utilizar. El intercambio directo de datos entre la cámara y los brazos nos permitiría, por ejemplo, establecer un lazo de control. En la Figura 6.1 podemos observar la aplicación de un sistema de visión parecido.

- Machine learning: Opencv posee algoritmos propios de machine learning como YOLO. Estos algoritmos podrían reemplazar totalmente a los algoritmos de visión anteriores, ya que suelen dar resultados más precisos e incluso a veces de menor coste computacional. El gran inconveniente es la creación de un conjunto de datos propio para entrenar el algoritmo.

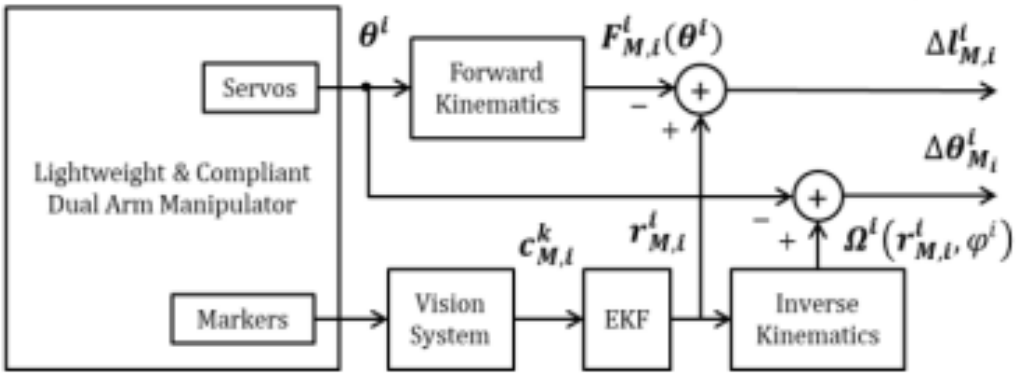


Figura 6.1 Lazo de control(Suarez et al. (b)).

Apéndice A
Código fuente

```
// Standard library
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>

#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <sys/time.h>
#include <fstream>

// OpenCV library
#include <opencv2/opencv.hpp>
#include "opencv2/core/core.hpp"
#include "opencv2/highgui/highgui.hpp"
#include "opencv2/imgproc/imgproc.hpp"
#include <librealsense2/rs.hpp> // Include RealSense Cross Platform API
#include <opencv2/opencv.hpp> // Include OpenCV API

#include <cmath>
#include "cv-helpers.hpp"
#include<iostream>
#include<vector>

#include <mutex>      // Include short list of convenience functions for
    rendering
#include <cstring>

#include <sys/wait.h> /* Para wait */
#include <sys/types.h> /* tipo pid_t */

//macros
#define Amin 100
```

```
#define Amax 10000
#define SminR 150 //saturacion minima rojo
#define SminA 120//saturacion minima azul
#define DEPTH 0.55//filtro profundidad 0.85 en lab
#define X 0.03//dimensiones del robot
#define Y 0.18
#define Z 0.23
// Namespaces
using namespace cv;
using namespace std;

using namespace cv::dnn;
using namespace rs2;


// Structure shared between threads
typedef struct
{
  int x0;
  int x1;
  int y0;
  int x2;

  uint8_t endFlag;
} THREAD_ARGS;


///
struct coord //guardo coordenadas de findcontours
  {
       float izq[3];
    float derech[3];
      Rect BBleft;
    Rect BBright;
  };
struct punto //guardo coordenadas de findcontours
  {
       float punto_XYZ[3];
  };

struct Camshif //guardo coordenadas de findcontours
  {
       RotatedRect bou;
    Rect vent_inicial;
  };

struct float3 { //Facilita operaciones
    float x, y, z;
    float3 operator*(float t)
    {
```

```
        return { x * t, y * t, z * t };
    }

    float3 operator-(float t)
    {
        return { x - t, y - t, z - t };
    }

    void operator*=(float t)
    {
        x = x * t;
        y = y * t;
        z = z * t;
    }

    void operator=(float3 other)
    {
        x = other.x;
        y = other.y;
        z = other.z;
    }

    void add(float t1, float t2, float t3)
    {
        x += t1;
        y += t2;
        z += t3;
    }
};
class rotation_estimator
{

    float3 theta;
    std::mutex theta_mtx;

    float alpha = 0.98f;
    bool firstGyro = true;
    bool firstAccel = true;

    double last_ts_gyro = 0;
public:

    void process_gyro(rs2_vector gyro_data, double ts)
    {
        if (firstGyro)
        {
            firstGyro = false;
            last_ts_gyro = ts;
            return;
        }
```

```
    float3 gyro_angle;


    gyro_angle.x = gyro_data.x; // Pitch
    gyro_angle.y = gyro_data.y; // Yaw
    gyro_angle.z = gyro_data.z; // Roll




    double dt_gyro = (ts - last_ts_gyro) / 1000.0;
    last_ts_gyro = ts;


    gyro_angle = gyro_angle * static_cast<float>(dt_gyro);

    std::lock_guard<std::mutex> lock(theta_mtx);
    theta.add(-gyro_angle.z, -gyro_angle.y, gyro_angle.x);
}

void process_accel(rs2_vector accel_data)
{

    float3 accel_angle;


    accel_angle.z = atan2(accel_data.y, accel_data.z);
    accel_angle.x = atan2(accel_data.x, sqrt(accel_data.y *
        accel_data.y + accel_data.z * accel_data.z));


    std::lock_guard<std::mutex> lock(theta_mtx);
    if (firstAccel)
    {
        firstAccel = false;
        theta = accel_angle;

  theta.y = 0;//angulo inicial

    }
    else
    {

        theta.x = theta.x * alpha + accel_angle.x * (1 - alpha);
        theta.z = theta.z * alpha + accel_angle.z * (1 - alpha);

    }

}
```

```
    float3 get_theta()
    {
        std::lock_guard<std::mutex> lock(theta_mtx);
        return theta;
    }

};

// Data packet sent to the dual arm control program with the markers
    position
typedef struct
{
  char header[3];  // "VIS" character sequence
  float leftMarkerPosition[3];
  float rightMarkerPosition[3];
} PACKET_MARKER_POSITION_VISION;


// Data packet received from the dual arm control program with the
    markers position (equivalent stiff joint arm)
typedef struct
{
  char header[3];  // "ARM" character sequence
  float leftMarkerPosition[3];
  float rightMarkerPosition[3];
} PACKET_MARKER_POSITION_ARMS;


// Thread function declaration
static void * keyboardThreadFunction(void * args);




//funciones
Mat procesar_imagen(Mat Mask);
coord deteccion(Mat procesada,Mat image,rs2::depth_frame depth_frame);
Mat mascara(Mat image,int hmin,int hmax,int smin,int smax,int vmin,int
    vmax );
Mat depth_filter(Mat image,rs2::depth_frame depth_frame,float max_depth)
    ;
struct Camshif Cam_shift(Mat rui,Mat image,Rect bound,rs2::depth_frame
    depth_frame);
struct punto imagen_a_robot(float punto[3],rs2_intrinsics intrin,float
    pitch,float yaw,float Roll,int L);
void dibujar(Mat image,struct punto puntoL,Rect BB);
THREAD_ARGS threadArgs;
int main(int argc, char** argv )
```

```
{

  pthread_t keyboardThread;
    Mat image;
  Mat MaskR,MaskR1,MaskR2,MaskA,Mask_depth;
  Mat procesada,procesadaR,procesadaA;
  struct coord coordenadasR,coordenadasA;
  struct rotation_estimator estimador;
    Mat roiRleft,roiRright,roiAleft,roiAright;
  float max_depth=DEPTH ;
  struct Camshif CSRI,CSRD,CSAI,CSAD;
  Rect BBRi_ant,BBRd_ant,BBAi_ant,BBAD_ant;
  struct punto RL,RR,AL,AR;
  float Pitch,yaw,Roll;



  //angulos
  float3 theta_rad,theta_grados;

    rotation_estimator estimador_angulo;

  /////rojo////////////

    int rhmin1=0,rhmax1=4,rhmin2=178,rhmax2=179,rsmin=SminR;//el hue es
        siempre <5 independientemente de la luz
    int rsmax=255,rvmin=40,rvmax=255;

  /////azul//////////
  int ahmin=70,ahmax=110,asmin=SminA;//dejar sat max a 255,vmin y vmax
      poca importancia */
    int asmax=255,avmin=50,avmax=255;


  //offstream
  ofstream dataFile;

    struct timeval tA;

    struct timeval tB;

    double t = 0;

    dataFile.open("Distancias.txt");//abro el archivo de texto

  PACKET_MARKER_POSITION_VISION dataPacketVision;  // Data packet sent
      to the dual arm control program with the marker position
  PACKET_MARKER_POSITION_ARMS dataPacketArms;  // Data packet received
      from the dual arm control program with the TCP position
  struct sockaddr_in addrHost;
```

```
  struct hostent * host;
int socketPublisher = -1;
int socketReceiver = -1;
int error = 0;




    // Init thread args structure
    threadArgs.endFlag = 0;

    // Init data packet fields
    strncpy(dataPacketVision.header, "VIS", 3);
    dataPacketVision.leftMarkerPosition[0] = 0;
    dataPacketVision.leftMarkerPosition[1] = 0;
    dataPacketVision.leftMarkerPosition[2] = 0;

  // Create the keyboard thread for managing the program execution
  if(pthread_create(&keyboardThread, NULL, &keyboardThreadFunction, (
      void*)&threadArgs))
    printf("ERROR: [in main] could not create keyboard thread.\n");
  usleep(1000);

  // Open the UDP socket for sending the visual corrections
  socketPublisher = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
    if(socketPublisher < 0)
    {
      fprintf(stderr, "ERROR: [in main] could not open socket.\n");
      error = 1;
    }

    host = gethostbyname("localhost");
    if(host == NULL)
  {
      fprintf(stderr, "ERROR: [in main] could not get host by name.\n")
          ;
      error = 1;
  }

  // Set the address of the host
  if(error == 0)
  {
    bzero((char*)&addrHost, sizeof(struct sockaddr_in));
    addrHost.sin_family = AF_INET;
    bcopy((char*)host->h_addr, (char*)&addrHost.sin_addr.s_addr, host
        ->h_length);
    addrHost.sin_port = htons(43000);
  }
  else
```

```
    close(socketPublisher);


  // Declare RealSense pipeline, encapsulating the actual device and
     sensors
  pipeline pipe;

  // Create a configuration for configuring the pipeline with a non
     default profile
  config cfg;

  // Add desired streams to configuration

cfg.enable_stream(RS2_STREAM_COLOR);
cfg.enable_stream(RS2_STREAM_ACCEL, RS2_FORMAT_MOTION_XYZ32F);
  cfg.enable_stream(RS2_STREAM_GYRO, RS2_FORMAT_MOTION_XYZ32F);
  cfg.enable_stream(RS2_STREAM_DEPTH);
pipe.start(cfg);
rs2::align align_to(RS2_STREAM_COLOR);
  const auto window_name = "Color";
  namedWindow(window_name, WINDOW_AUTOSIZE);

auto const intrin = pipe.get_active_profile().get_stream(
    RS2_STREAM_COLOR).as<rs2::video_stream_profile>().get_intrinsics();
    //parametros intrinsecos




// Skips some frames to allow for auto-exposure stabilization
  for (int i = 0; i < 10; i++) pipe.wait_for_frames();
// Antes del bucle obtienes la marca de tiempo inicial

gettimeofday(&tA, NULL);


  while ((getWindowProperty(window_name, WND_PROP_AUTOSIZE) >= 0) && (
     waitKey(1) != 'q') )
  {

      // Obtienes la marca de tiempo actual
      gettimeofday(&tB, NULL);

  t = (tB.tv_sec - tA.tv_sec) + 1e-6*(tB.tv_usec - tA.tv_usec);
      auto data = pipe.wait_for_frames();
      // Make sure the frames are spatially aligned
      data = align_to.process(data);
```

```
        auto color_frame = data.get_color_frame();
        auto depth_frame = data.get_depth_frame();

    // Query frame size (width and height)
        const int w = depth_frame .as<rs2::video_frame>().get_width();
        const int h = depth_frame .as<rs2::video_frame>().get_height();


    //Utilizo la funcion helper lo paso de intelrealsense a opencv
    auto image = frame_to_mat(color_frame);

         //ajuste

     namedWindow("Trackbars azul",(640,200));//crear ventana
     namedWindow("Trackbars rojo",(640,200));//crear ventana
        createTrackbar("Hue min", "Trackbars azul", &ahmin, 179);
       createTrackbar("Hue max", "Trackbars azul", &ahmax, 179);
       createTrackbar("sat min", "Trackbars azul", &asmin, 255);
       createTrackbar("sat max", "Trackbars azul", &asmax, 255);
       createTrackbar("Val min", "Trackbars azul", &avmin, 255);
       createTrackbar("Val max", "Trackbars azul", &avmax, 255);
       createTrackbar("Hue min", "Trackbars rojo", &rhmin1, 179);
       createTrackbar("Hue max", "Trackbars rojo", &rhmax1, 179);
       createTrackbar("sat min", "Trackbars rojo", &rsmin, 255);
       createTrackbar("sat max", "Trackbars rojo", &rsmax, 255);
       createTrackbar("Val min", "Trackbars rojo", &rvmin, 255);
       createTrackbar("Val max", "Trackbars rojo", &rvmax, 255);

    ///rojo/////
    MaskR1=mascara(image,rhmin1,rhmax1,rsmin, rsmax,rvmin,rvmax);//hago
        mascara de color (0,4)
    MaskR2=mascara(image,rhmin2,rhmax2,rsmin, rsmax,rvmin,rvmax);//hago
        mascara de color(170,179)
    add(MaskR1,MaskR2,MaskR);//sumo las mascaras
    procesadaR=procesar_imagen( MaskR);//proceso la imagen
    coordenadasR=deteccion(procesadaR,image,depth_frame); //devuelve x,y,
        z de los contornos detectados

     ///azul//////

    MaskA=mascara(image,ahmin,ahmax,asmin, asmax,avmin,avmax);//hago
        mascara de color
    procesadaA=procesar_imagen( MaskA);//proceso la imagen
    coordenadasA=deteccion(procesadaA,image,depth_frame); //devuelve x,y,
        z de los contornos detectados


        if(threadArgs.y0==1)
{
```

```
  if (coordenadasR.BBleft.empty()!=true) BBRi_ant=coordenadasR.
      BBleft;//si esta lleno lo copio
else
{
CSRI=Cam_shift(procesadaR, image, BBRi_ant,depth_frame);
coordenadasR.izq[0]=CSRI.bou.center.x;
coordenadasR.izq[1]=CSRI.bou.center.y;
coordenadasR.izq[2]=depth_frame.get_distance(CSRI.bou.center.x,CSRI.
   bou.center.y);
BBRi_ant=CSRI.vent_inicial;


}
if (coordenadasR.BBright.empty()!=true) BBRd_ant=coordenadasR.
   BBright;
else
{
  CSRD=Cam_shift(procesadaR, image, BBRd_ant,depth_frame);
  coordenadasR.derech[0]=CSRD.bou.center.x;
    coordenadasR.derech[1]=CSRD.bou.center.y;
    coordenadasR.derech[2]=depth_frame.get_distance(CSRD.bou.center.x
       ,CSRD.bou.center.y);
  BBRd_ant=CSRD.vent_inicial;


}
  if (coordenadasA.BBleft.empty()!=true) BBAi_ant=coordenadasA.
      BBleft;
else
{
  CSAI=Cam_shift(procesadaA, image, BBAi_ant,depth_frame);
  coordenadasA.izq[0]=CSAI.bou.center.x;
  coordenadasA.izq[1]=CSAI.bou.center.y;
  coordenadasA.izq[2]=depth_frame.get_distance(CSAI.bou.center.x,
      CSAI.bou.center.y);
  BBAi_ant=CSAI.vent_inicial;

}

if (coordenadasA.BBright.empty()!=true) BBAD_ant=coordenadasA.
   BBright;
else
{
  CSAD=Cam_shift(procesadaA, image, BBAD_ant,depth_frame);
    coordenadasA.derech[0]=CSAD.bou.center.x;
  coordenadasA.derech[1]=CSAD.bou.center.y;
  coordenadasA.derech[2]=depth_frame.get_distance(CSAD.bou.center.x,
      CSAD.bou.center.y);
  BBAD_ant=CSAD.vent_inicial;
}
```

```
}


   //calculo de los angulos

   //acelerometro
      auto fa = data.first(RS2_STREAM_ACCEL, RS2_FORMAT_MOTION_XYZ32F);

      rs2::motion_frame accel = fa.as<rs2::motion_frame>();
     //giroscopio
      auto fg = data.first(RS2_STREAM_GYRO, RS2_FORMAT_MOTION_XYZ32F);
      rs2::motion_frame gyro = fg.as<rs2::motion_frame>();
  if (gyro)
      {
    double ts = data.get_timestamp();
         rs2_vector gyro_data = gyro.get_motion_data();
    estimador.process_gyro(gyro_data, ts);

  }
  if (accel)
      {
         rs2_vector accel_data= accel.get_motion_data();
    estimador.process_accel(accel_data);
  }

      theta_rad=estimador.get_theta();//obtengo theta del mutex



   theta_grados = theta_rad * static_cast<float>(180.0/M_PI);//paso a
       grados para representar
   Pitch=(-theta_grados.z-90)*(M_PI/180.0);

   //cout << "Pitch:" << theta_grados.z << " Yaw:" << theta_grados.y
       << " Roll:" << theta_grados.x<< std::endl;//represento theta */


      if(threadArgs.x1==1) image=depth_filter(image,depth_frame,
         max_depth);//elimino todo lo mayor a la distancia de brazo

    RL=imagen_a_robot(coordenadasR.izq,intrin,Pitch,yaw,-theta_rad.x
        ,1);
  RR=imagen_a_robot(coordenadasR.derech,intrin,Pitch,yaw,-theta_rad.x
      ,0);
  AL=imagen_a_robot(coordenadasA.izq,intrin,Pitch,yaw,-theta_rad.x,1);
  AR=imagen_a_robot(coordenadasA.derech,intrin,Pitch,yaw,-theta_rad.x
      ,0);


        if(threadArgs.y0==1)
```

```
{
      //dibujar coordenadas
  dibujar(image,RL, BBRi_ant);
  dibujar(image,RR,BBRd_ant);
  dibujar(image,AL,BBAi_ant);
  dibujar(image,AR,BBAD_ant);
}
else
{
       //dibujar coordenadas
  dibujar(image,RL, coordenadasR.BBleft);
  dibujar(image,RR,coordenadasR.BBright);
  dibujar(image,AL,coordenadasA.BBleft);
  dibujar(image,AR,coordenadasA.BBright);
}
  //dibujar Pitch


   Point letrero_Pitch (w-320, h-10);
   ostringstream ss_Pitch;
       ss_Pitch << std::setprecision(3) << "Inclinacion:" <<-
           theta_grados.z-90 << " yaw "<< theta_grados.y<<" Roll "<< -
           theta_grados.x;
       String conf_Pitch(ss_Pitch.str());
       putText(image, ss_Pitch.str(), letrero_Pitch,
           FONT_HERSHEY_SIMPLEX, 0.5, Scalar(0,0,0));







  //sacar datos////////////

      dataFile << t << "\t" << RL.punto_XYZ[0]<< "\t" << RL.punto_XYZ
          [1] << "\t"<<RL.punto_XYZ[2] <<"\t"<<RR.punto_XYZ[0]<< "\t"
          << RR.punto_XYZ[1] << "\t"<<RR.punto_XYZ[2] <<"\t"<<endl;
      imwrite("/home/pablo/Escritorio/TFG/tfg_2.0/build/Robot.PNG",
          image);
  imshow(window_name, image);
  imshow("masc roja", procesadaR);
      imshow("masc azul", procesadaA);


  imwrite("/home/pablo/Escritorio/TFG/tfg_2.0/build/depth.PNG",image);
// Send the packet
if(sendto(socketPublisher, (char*)&dataPacketVision, sizeof(
    PACKET_MARKER_POSITION_VISION), 0, (struct sockaddr*)&addrHost,
    sizeof(struct sockaddr)) < 0)
  {
```

```
          fprintf(stderr, "ERROR: [in main] could not send data.\n");
          error = 1;
    }

    }



   //fuera del loop
    // Finish threads
    threadArgs.endFlag = 1;
    usleep(250000);


    // Close socket
    close(socketPublisher);
  dataFile.close();
  printf("Cerrado con exito.\n");
    return 0;
}


/*
 * Thread for reading the keyboard inputs
 */
static void * keyboardThreadFunction(void * args)
{
  THREAD_ARGS * threadArgs = (THREAD_ARGS*)args;
  extern int code;


  do
  {
    //ajuste
    printf("activar filtrado profundidad(1/0)/activar Camshift(1/0).\n")
        ;
      scanf(" %d %d ",&threadArgs->x1,&threadArgs->y0);

  }while(threadArgs->endFlag == 0);

  return 0;
}
Mat procesar_imagen(Mat Mask)
{
        //////////procesamiento de la imagen/////////////
    Mat filtrada;
        Mat dilatada;
      Mat erosionada;
```

```
    blur(Mask,filtrada,Size(3,3));//filtro para rudio en general,cuanto
        mas grande el kernel mas efecto tiene.He escogido este poruq
        edifumina un poco los bordes pero nos da igual
    Mat kerneldil=getStructuringElement(MORPH_RECT,Size(7,7));//crea un
        kernel compatible con dilate.
    Mat kernelerode=getStructuringElement(MORPH_RECT,Size(3,3));
    dilate(filtrada,dilatada,kerneldil);//dilato la imagen
    erode(dilatada,erosionada,kernelerode);//erosiono
    return dilatada;

}
struct coord deteccion(Mat procesada,Mat image,rs2::depth_frame
    depth_frame)
{
//ordenar
struct contour_sorter // 'less' for contours
{
    bool operator ()( const vector<Point>& a, const vector<Point> & b )
    {
        Rect ra(boundingRect(a));
        Rect rb(boundingRect(b));
        // scale factor for y should be larger than img.width
        return ( (ra.x ) < (rb.x ) );
    }
};

        struct contour_sorter sorter;
      struct coord cord;
        float dist;
    float distancia[]={0,0};
      vector<vector<Point>>contours;//vector de vectores de puntos=
          contorno
    findContours(procesada,contours,RETR_EXTERNAL,CHAIN_APPROX_SIMPLE);
        //encuentra contornos,aproximacion simple para ahorrar puntos
    if(contours.size()==2)//solo opera cuando ha detecto 2 objetos
    {
      sort(contours.begin(), contours.end(), sorter);
      vector<Moments> mu(contours.size() );//vector de mometos del
          contorno
        vector<Point2f> mc(contours.size());//vector de centroides
      vector<Rect> boun(contours.size());

      for(int i=0;i<contours.size();i++) //recorro contorno
      {
          int area=contourArea(contours[i]);//calculo areas de los
              contornos



        if(area>Amin && area<Amax) //elimino posibles ruidos y errores
```

```
{


  boun[i]=boundingRect(contours[i]);  //creo estructura con cada
      contorno detectado

          //calculo del centroide.
  mu[i] = moments( contours[i] );//calculo momentos
           mc[i] = Point2f( static_cast<float>(mu[i].m10 / (mu[i
              ].m00 + 1e-5)),static_cast<float>(mu[i].m01 / (mu[
              i].m00 + 1e-5)) );//calculo centroide

    //creo estructura con cada contorno detectado



  //distancia al centroide
    dist=depth_frame.get_distance(mc[i].x,mc[i].y);

  distancia[i]=dist;//calculo vector distancias

     //representación en pantalla//

    rectangle(image,boun[i].tl(),boun[i].br(),(0,0,0),3); //
        dibujo rectangulo en la imagen
    circle(image,mc[i],2,Scalar(0,0,255),FILLED);// Dibujo
        centroide


        //guardar bounding boxes



    cord.BBleft=boun[0];
  cord.BBright=boun[1];

  cord.izq[0]=mc[0].x;
  cord.derech[0]=mc[1].x;

  cord.izq[1]=mc[0].y;
  cord.derech[1]=mc[1].y;

  cord.izq[2]=distancia[0];
  cord.derech[2]=distancia[1];


}
/* else if(area<Amin) cout<<"Marker con área demasiado pequeña
    "<<endl;
```

```
        else if(area>Amax) cout<<"Marker con área demasiado grande"<<
            endl; */
      }


    }
    /* else if(contours.size()==1) cout<<"No se han detectado
        suficientes markers"<<endl;
    else if(contours.size()==0) cout<<"No se han detectado markers"<<
        endl;
    else cout<<"Se han detectado demasiados markers"<<endl; */
    return cord;
}
Mat mascara(Mat image,int hmin,int hmax,int smin,int smax,int vmin,int
    vmax )
{
  Mat HSV;
  Mat masc;
  cvtColor(image,HSV,COLOR_BGR2HSV);//convertir a HSV
  Scalar lower(hmin,smin,vmin);// Matiz(hue),saturacion minima,valor
      minimo
  Scalar upper(hmax,smax,vmax);
  inRange(HSV,lower,upper,masc);//te hace la mascara
  return masc;
}
 Mat depth_filter(Mat image,rs2::depth_frame depth_frame,float max_depth
     )
{
   Mat bgModel,fgModel,mask;

   Mat depth_mat = depth_frame_to_meters(depth_frame);//matriz de
       profundidad
   Mat BG=Mat::zeros(depth_mat.size(),CV_8UC1);//matriz de ceros
   Mat FG=Mat::zeros(depth_mat.size(),CV_8UC1);//matriz de ceros
   BG.setTo(255,depth_mat>max_depth);//si esta mas lejos de la distancia
       maxima 0
   FG.setTo(255,depth_mat<max_depth);//si esta mas cerca de la distancia
       maxima 1

   ////////////
   /////creo matriz de la forma de la funcion grabCut
    mask.create(depth_mat.size(), CV_8UC1);
    mask.setTo(Scalar::all(GC_BGD)); // Set "background" as default
        guess
    mask.setTo(GC_PR_BGD, BG ==0); // Relax this to "probably background
        " for pixels outside "far" region
    mask.setTo(GC_FGD, FG == 255); // Set pixels within the "near"
        region to "foreground"
    //////
```

```
    grabCut(image, mask, Rect(), bgModel, fgModel, 1, GC_INIT_WITH_MASK)
        ;
    // Extract foreground pixels based on refined mask from the
        algorithm
    Mat3b foreground = Mat3b::zeros(image.rows, image.cols);
    image.copyTo(foreground, (mask == GC_FGD) | (mask == GC_PR_FGD));
    //imshow("foreground", image);
    return foreground;
}
struct Camshif Cam_shift(Mat rui,Mat image,Rect bound,rs2::depth_frame
    depth_frame)
{
    Mat hsv,dst;
  struct Camshif Cam;
  /*   int channels[] = {0};
    float range_[] = {0, 180};
    const float* range[] = {range_};const */
     // Setup the termination criteria, either 10 iteration or move by
         atleast 1 pt
    TermCriteria term_crit(TermCriteria::EPS | TermCriteria::COUNT, 10,
        1);
  //cvtColor(image, hsv, COLOR_BGR2HSV);
    //calcBackProject(&hsv, 1, channels, rui, dst,range);

    RotatedRect rot_rect = CamShift(rui, bound, term_crit);
    // Draw it on image
    Point2f points[4];
    rot_rect.points(points);
    for (int i = 0; i < 4; i++) line(image, points[i], points[(i+1)%4],
        255, 2);
  circle(image,rot_rect.center,2,Scalar(0,0,255),FILLED);// Dibujo
      centroide
  Cam.vent_inicial=bound;
  Cam.bou=rot_rect;
    return Cam;

}
struct punto imagen_a_robot(float punto[3],rs2_intrinsics intrin,float
    pitch,float yaw,float Roll,int L)
{
  rs2_extrinsics extr;
  struct punto punt;
    float point[3];
  const float pixel[2]={punto[0],punto[1]};
  float depth=punto[2];
  float cam[3];
  float arm[3];

  rs2_deproject_pixel_to_point(point,&intrin, pixel,depth);
   //cambio de ejes///
```

```
    cam[0]=point[2]-X;
  if (L==1)cam[1]=-point[0]-Y; else cam[1]=-point[0]+Y;
  cam[2]=-point[1]+Z;


  arm[0]=cam[0]*cos(pitch)+cam[2]*sin(pitch);
  arm[1]=cam[1];
  arm[2]=-cam[0]*sin(pitch)+cam[2]*cos(pitch);



  punt.punto_XYZ[0]=arm[0];
  punt.punto_XYZ[1]=arm[1];
  punt.punto_XYZ[2]=arm[2];
  return punt;
}
void dibujar(Mat image,struct punto punto,Rect BB)
{


  Point letrero (BB.x-20,BB.y-20);
  ostringstream ss;
  ss<<"["<<setprecision(2)<<punto.punto_XYZ[0]*100<<","<<setprecision(2)
      <<punto.punto_XYZ[1]*100<<","<<setprecision(2)<<punto.punto_XYZ
      [2]*100<<"]";
  string conf(ss.str());
  putText(image,ss.str(),letrero,FONT_HERSHEY_SIMPLEX,0.5,Scalar(0,0,0))
      ;

}
```

Índice de Figuras

Índice de Códigos

Bibliografía

«Inteligencia artificial fácil - Machine Learning y Deep Learning prácticos - Algoritmo de mezclas gaussianas o Gaussian Mixture Model (GMM) | Editiones ENI».
https://www.ediciones-eni.com/open/mediabook.aspx?idR=1922ca92926a43a880845e5c025b168e.

«LiCAS Robotic Arms – Lightweight and Compliant Robotic Arms Manufacturer».
https://licas-robotic-arms.com/.

«OpenCV: Interactive Foreground Extraction using GrabCut Algorithm».
https://docs.opencv.org/3.4/d8/d83/tutorial_py_grabcut.html.

«OpenCV: Meanshift and Camshift».
https://docs.opencv.org/4.x/d7/d00/tutorial_meanshift.html.

«Projection in Intel RealSense SDK 2.0».
https://dev.intelrealsense.com/docs/projection-in-intel-realsense-sdk-20.

«Projection in RealSense SDK 2.0 · IntelRealSense/librealsense Wiki».
https://github.com/IntelRealSense/librealsense.

«rs-align».
https://dev.intelrealsense.com/docs/rs-grabcuts.

«rs-grabcuts».
https://dev.intelrealsense.com/docs/rs-grabcuts.

«rs-motion».
https://dev.intelrealsense.com/docs/rs-motion.

Bradski, Gary R «Computer Vision Face Tracking For Use in a Perceptual User Interface». p. 15.

Bradski, Gary R. y Kaehler, Adrian *Learning OpenCV: computer vision with the OpenCV library*. Software that sees. O'Reilly, 1. ed., [nachdr.].ª edición. ISBN 978-0-596-51613-0.

Min, Hyung Gi y Jeung, Eun Tae «Complementary Filter Design for Angle Estimation using MEMS Accelerometer and Gyroscope». p. 11.

Murtaza's Workshop - Robotics and AI «LEARN OPENCV C++ in 4 HOURS | Including 3x Projects | Computer Vision».
https://www.youtube.com/watch?v=2FYm3GOonhk.

Rother, Carsten; Kolmogorov, Vladimir y Blake, Andrew «"GrabCut" — Interactive Foreground Extraction using Iterated Graph Cuts». p. 6.

Suarez, Alejandro; Heredia, Guillermo y Ollero, Anibal «Physical-Virtual Impedance Control in Ultralightweight and Compliant Dual-Arm Aerial Manipulators». **3(3)**, pp. 2553–2560. ISSN 2377-3766, 2377-3774. doi: 10.1109/LRA.2018.2809964. *https://ieeexplore.ieee.org/document/8302932/*.

Suarez, Alejandro; Heredia, Guillermo y Ollero, Anibal «Vision-Based Deflection Estimation in an Anthropomorphic, Compliant and Lightweight Dual Arm». En: Anibal Ollero; Alberto Sanfeliu; Luis Montano; Nuno Lau y Carlos Cardeira (Eds.), *ROBOT 2017: Third Iberian Robotics Conference*, tomo 694, pp. 332–344. Springer International Publishing. ISBN 978-3-319-70835-5 978-3-319-70836-2. doi: 10.1007/978-3-319-70836-2_28. *http://link.springer.com/10.1007/978-3-319-70836-2_28*. Series Title: Advances in Intelligent Systems and Computing.

Sural, S.; Gang Qian y Pramanik, S. «Segmentation and histogram generation using the HSV color space for image retrieval». En: *Proceedings. International Conference on Image Processing*, tomo 2, pp. II–589–II–592. IEEE. ISBN 978-0-7803-7622-9. doi: 10.1109/ICIP.2002.1040019. *http://ieeexplore.ieee.org/document/1040019/*.

Temoche, Pablo; Ramírez, Esmitt y Carmona, Rhadamés «3D GrabCut: Una segmentación de volúmenes basada en la técnica GrabCut utilizando la GPU». p. 8.