Universidad de Alcalá
Escuela Politécnica Superior

Grado en Ingeniería en Electrónica y Automática Industrial

Trabajo Fin de Grado

Diseño y Simulación de un sistema de identificación y seguimiento de objetos para un brazo robótico.

Autor: Pedro José Vidal Moreno

Tutor: Francisco Javier Rodríguez Sánchez

2019

UNIVERSIDAD DE ALCALÁ

ESCUELA POLITÉCNICA SUPERIOR

Grado en Ingeniería en Electrónica y Automática Industrial

Trabajo Fin de Grado

Diseño y Simulación de un sistema de identificación y seguimiento de objetos para un brazo robótico.

Autor: Pedro José Vidal Moreno

Director: Francisco Javier Rodríguez Sánchez

Tribunal:

Presidente: Rafael Barea Navarro

Vocal 1º: Juan Carlos García García

Vocal 2º: Francisco Javier Rodríguez Sánchez

Calificación: .....

Fecha: .....

Agradecimientos

El resultado de este proyecto es el fruto de muchas horas de trabajo, estudio y en algunas ocasiones de improvisación. Pero a pesar de las horas de sacrificio y de todo lo que me dejo en el tintero, me siento contento con el resultado. El TFG marca un punto y aparte en mi paso por la universidad, por lo que no podría entregarlo sin expresar mi agradecimiento a la gente que lo ha hecho posible y que durante estos años me ha acompañado.

Gracias a todos los que me habéis ayudado y seguís haciéndolo. A mis compañeros de biblioteca y a mis profesores. He aprendido mucho de vosotros y espero que aun tengáis mucho que enseñar, me he dado cuenta de lo poco que se y de lo satisfactorio que es entender cosas que pensaba que estaban fuera de mi alcance.

Además me gustaría agradecer, de forma más especifica, a mi tutor y a la empresa Mytra Control S.L. el darme la oportunidad de entrar en el proyecto. Gracias también a mi colega de la cátedra por nuestras lluvias de ideas y por darme siempre otro punto de vista.

Me reservo para el final de este apartado los agradecimientos más importantes. Gracias a mi familia por ayudarme en todo lo que han podido. A mis amigos y a Raquel por saber sacarme una sonrisa hasta en época de exámenes.

Resumen

El presente trabajo ha sido realizado como parte de un proyecto de robótica colaborativa. Se han estudiado técnicas de visión artificial y procesado de imágenes en tres dimensiones tomadas con una cámara de visión estéreo con el fin de localizar espacialmente un determinado volumen.

Palabras clave: Brazo robot, vision artificial, vision estereo.

Abstract

This work has been carried out as part of a collaborative robotics project. Artificial vision techniques and three-dimensional image processing taken with a stereo vision camera have been studied in order to spatially locate a given volume.

Keywords: Robotic arm, computer vision, estereoscopic vision.

Índice general

Índice de figuras

Índice de tablas

Capítulo 1

Introducción

1.1 Presentación

La visión por computador se ha aplicado tradicionalmente en las ramas relacionadas con la industria y la tecnología como el control de calidad o la automatización. Pero hoy en día la tecnología permite aplicarla a la conducción autónoma, la medicina o la realización de tareas domesticas. Esto se consigue apoyándose en los avances hechos en robótica, que han traído al mercado robots cada vez más seguros, capaces de trabajar en colaboración con los humanos. Por otro lado, el aumento de la capacidad de cómputo permite procesar toda la información recibida del exterior a través de sensores cada vez más complejos.

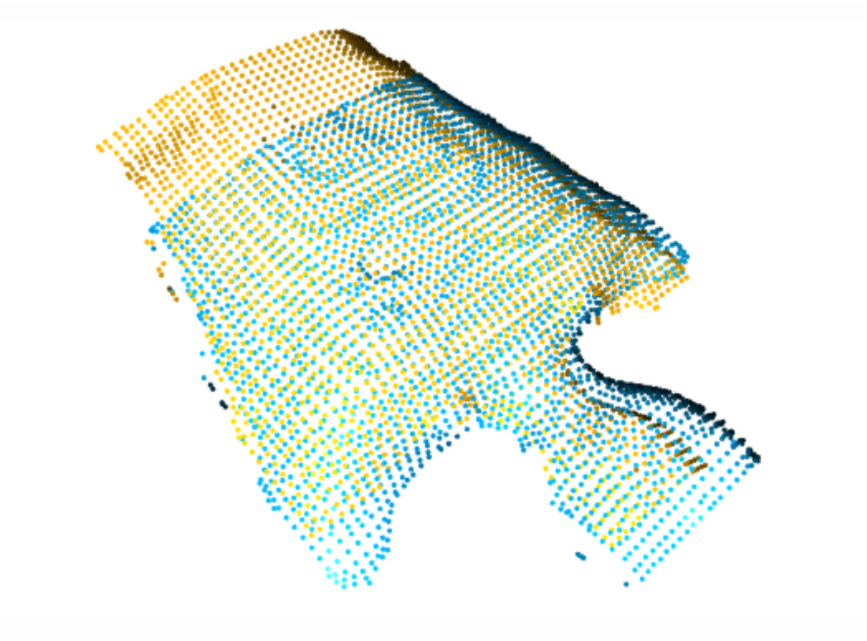
Los sistemas de visión por computador tienen como objetivo procesar las imágenes del mundo real para conseguir información que pueda ser tratada posteriormente. Estas técnicas adquieren mucha importancia como parte de los sistemas robotizados que cada vez están más integrados con el entorno, respondiendo ante sus cambios de forma precisa. Al sacar a los robots de los entornos industriales, que cuentan con condiciones altamente controladas, es necesario añadir robustez extra ante cambios de iluminación o desarrollar algoritmos de segmentación complejos que no solo se basen en características conocidas previamente.

El presente trabajo de fin de grado se enmarca en un proyecto realizado en la Cátedra Mytra de la UAH. Mi participación en dicho proyecto ha consistido en el cálculo del desplazamiento de un paciente dentro de una camilla con respecto a la posición del paciente en el momento en el que se crea un tratamiento. Para ello se ha utilizado una cámara de visión estéreo que captura cuatro canales: Rojo, verde, azul y profundidad (RGBD). El desarrollo práctico del proyecto consiste en la realización de un *"worker"* alojado en la nube, este programa se mantendrá a la espera de que llegue una orden de trabajo en una cola de tareas. Una vez reciba una orden, descargará las imágenes del paciente de la base de datos y ejecutará una serie de algoritmos para calcular el desplazamiento de este. Junto a las imágenes, el *"worker"* recibirá también los parámetros intrínsecos y excéntricos con el fin de poder trabajar con coordenadas reales.

Una vez calculado el desplazamiento del paciente, se convertirá dicho desplazamiento al sistema de coordenadas de un brazo robot para que este pueda aplicar el tratamiento oportuno.

1.2 Entorno experimental previo

En esta sección se explicarán los elementos utilizados para la realización del proyecto. Ha sido necesaria la elección de una cámara de profundidad y un brazo robot.



La cámara se ha buscado que tenga un *SDK* [1] compatible con Linux y disponible para varios lenguajes de programación: *C++* y *Python*. Otro aspecto a tener en cuenta es el rango de distancias en el que puede trabajar. Deberá poder detectar objetos en un rango entre 0.15 y 2 metros. Teniendo en cuenta las necesidades de la aplicación, se han considerado las siguientes tres cámaras. Todas tienen un *SDK* compatible con *Linux*

- La familia *Intel RealSense*, con un *SDK* libre, compatible con *Windows*, *Linux* y *Mac*. Cuenta con dos cámaras, la D415 y D435. La primera con un rango de profundidad de 0.16-10 metros y la segunda de 0.11-10 metros. Ambas tienen una resolución para el canal de profundidad de 16 bits.
- *ASUS Xtion PRO*: Está diseñada para el control de videojuegos mediante gestos por lo que el rango de profundidad es 0.8 a 3.5 m.
- *StereoLab ZED*: Tiene un rango de 0.5 a 20m y una resolución para el canal de profundidad de 32 bits.

Para escoger el robot se ha tenido en cuenta que va a trabajar con personas, por lo que debe de estar preparado para entrar en parada ante el mínimo golpe mientras trabaja a velocidades bajas. Esto hace que se hayan descartado los robots industriales tradicionales. Además debe de poder ser controlado desde un ordenador externo mediante una conexión *TCP*[2].

1.2.1 Cámara de profundidad

La cámara escogida es la Realsense D435 de Intel 1.1. De las tres opciones barajadas, esta la que posee las características más adecuadas para el propósito de la aplicación. Sus aspectos destacados son:

- Dos sensores de profundidad.
- Un sensor de color RGB.

[1] *Software Development Kit*: Bibliotecas para el control de la cámara.
[2] En inglés *Transmission Control Protocol*

- Un procesador de 28 nanometros que admite hasta cinco canales MIPI y dos lineas para computar las imágenes de profundidad en tiempo real.
- Un algoritmo de visión estéreo ya incorporado que descarga de este calculo al ordenador principal.
- Un proyector de matriz de puntos infrarrojos para hacer más robusto el calculo de profundidad.
- Un procesador dedicado para el tratamiento de las imágenes de color.
- Resolución hasta 1280 x 720 en los cuatro canales.
- Compatibilidad con el SDK librealsense2 de codigo abierto y disponible en varios lenguajes como C++ o Python.

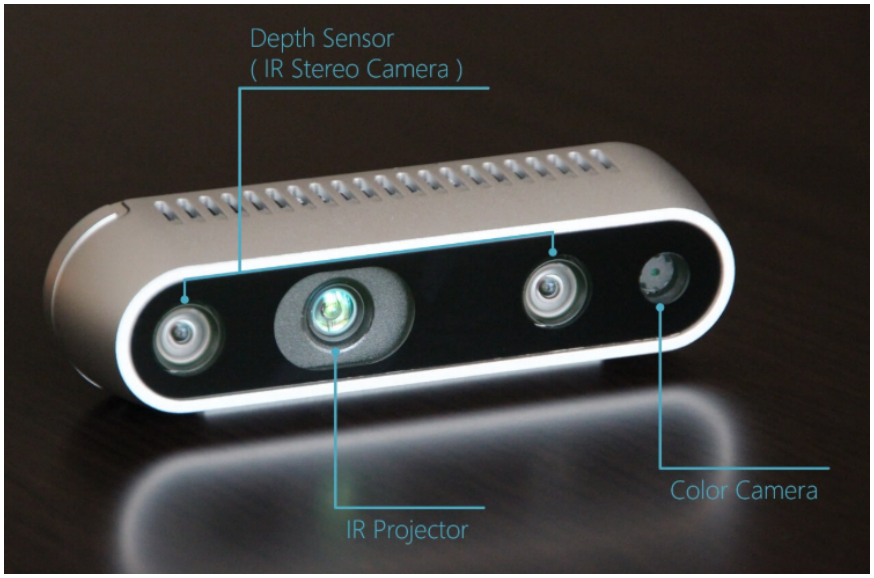


Figura 1.1: Cámara de profundidad *Realsense D435*.

1.2.2 Brazo Robot

El robot utilizado es el UR5 de Universal Robots 1.2 que destaca por su ligereza y cuenta con un modo *free-drive* en el que el usuario puede mover el efector del robot libremente y guardar los puntos deseados. Se trata de un brazo compuesto por juntas y tubos de aluminio extruido. Cuenta con seis grados de libertad y un alcance esférico. En la siguiente tabla se verán las características obtenidas del fabricante y en la figura.

Actuadores.	
Sensor de Fuerza	x-y-z
Rango	50 N
Resolución	2.5 N
Precisión	4.0 N
Sensor de Torque	x-y-z
Rango	10 Nm
Resolución	0.04 Nm
Precisión	0.30 Nm
Rango de temperatura ambiente	0 - 50°C
Carga	5 Kg / 11 lbs
Alcance	850 mm / 33.5 in
Grados de libertad	6 Rotating Joints DOF

Tabla 1.1: Tabla de características de los actuadores.

Movimiento.	
Repetitibilidad	+/- 0.03 mm con carga
Rangos de cada junta	
Base	±360° ±180°/s
Shoulder	±360° ±180°/s
Elbow	±360° ±180°/s
Wrist 1	±360° ±180°/s
Wrist 2	±360° ±180°/s
Wrist 3	±360° ±180°/s
Velocidad del TCP	1m/s

Tabla 1.2: Limites de moviemiento de las articulaciones.

Controlador.	
Puertos I/O	Digital in
	Digital in 16
	Digital out 16
	Analog in 2
	Analog out 2
Alimentación I/O	24V 2A
Comunicaciones	Frecuencia de control 500 Hz
	Modbus TCP. 500 Hz
	ProfiNet y Ethernet IP. 500 Hz
	Puertos un USB 2.0 y un USB 3.0
Fuente de alimentación	100-240V AC, 47-440 Hz

Tabla 1.3: Tabla de características del controlador.

Figura 1.2: Brazo robot UR5e.

1.3 Estructura del documento.

Este trabajo de fin de grado se estructura en 3 capítulos, en el primero, dedicado al estudio teórico, se analizarán los algoritmos de visión por computador utilizados en el desarrollo del proyecto y alguno más que se ha estudiado y que al final se descartó. En el capítulo referente al desarrollo práctico, se explicarán las cuestiones técnicas sobre los programas desarrollados y los distintos componentes utilizados. Por último se analizarán los resultados y se expondrán las conclusiones obtenidas y se hablará de líneas futuras de mejora.

Capítulo 2

Estudio teórico

2.1 Introducción

A la hora de enfrentarnos a una aplicación de visión artificial podemos dividir el flujo de trabajo en distintas etapas por las que irá pasando una imagen. 2.1

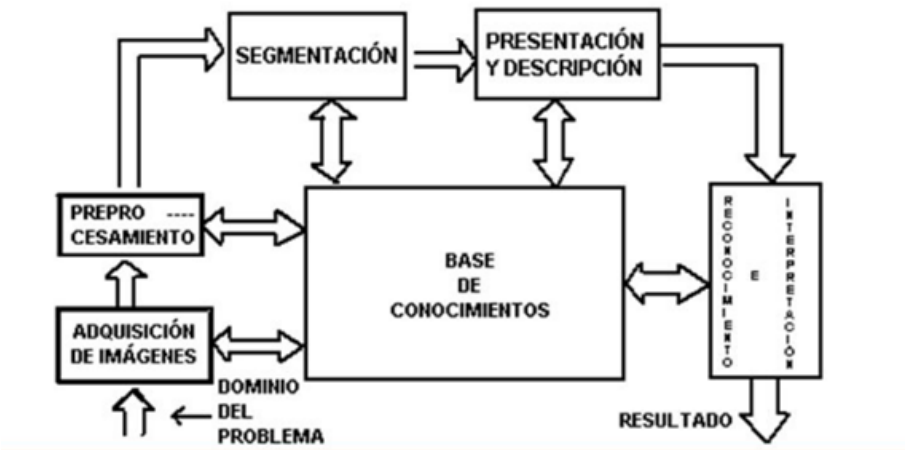


Figura 2.1: Diagrama de flujo por el que pasan las imagenes.

El capítulo se estructura en cuatro apartados, en los que se dará una visión teórica de cada etapa.

En primer lugar se sentarán, las bases teóricas que se han aprendido durante el desarrollo del proyecto sobre la adquisición de imágenes. En la sección 2.2 se estudiará el modelo de una cámara simple y seguidamente se estudiará el sistema de visión estéreo para encontrar las componentes de profundidad de una imagen. Con esto veremos que se podrá reconstruir la escena tridimensional.

A continuación, se estudiarán los algoritmos de preprocesamiento necesarios para aumentar la eficacia en los pasos posteriores, en el apartado 2.3 se verán las operaciones básicas que podemos hacer sobre una imagen en dos dimensiones y en el apartado 2.4 veremos algún algoritmo utilizado para procesar nubes de puntos en tres dimensiones.

A la hora de identificar objetos es fundamental también realizar la segmentación de la imagen en grupos de píxeles pertenecientes a un mismo grupo de características visuales. Una vez segmentada la imagen se busca asociar estas características visuales con las características descriptivas de los objetos que

queramos identificar. Por último se pasa a interpretar los descriptores y a clasificar los objetos contenidos en la imagen.

2.2 Adquisición de imágenes.

En este proyecto las imágenes se toman mediante un módulo de visión estereoscópica. A grandes rasgos, este módulo consta de dos sensores que capturan imágenes por separado y un procesador dedicado que aplica los algoritmos necesarios para obtener la imagen en tres dimensiones. Además cuenta con proyector de matriz de infrarrojos para mejorar la robustez ante cambios en la iluminación. Para entender cómo funciona la obtención de la profundidad de una escena, vamos a estudiar el modelo de un sistema de visión estero básico compuesto por dos cámaras.

2.2.1 Modelo Pin-hole de una camara.

La forma más simple que tenemos de modelar el comportamiento de una cámara es con el modelo Pin-Hole o de cámara estenopeica. Este modelo no tiene en cuenta la óptica usada y la reduce a un punto

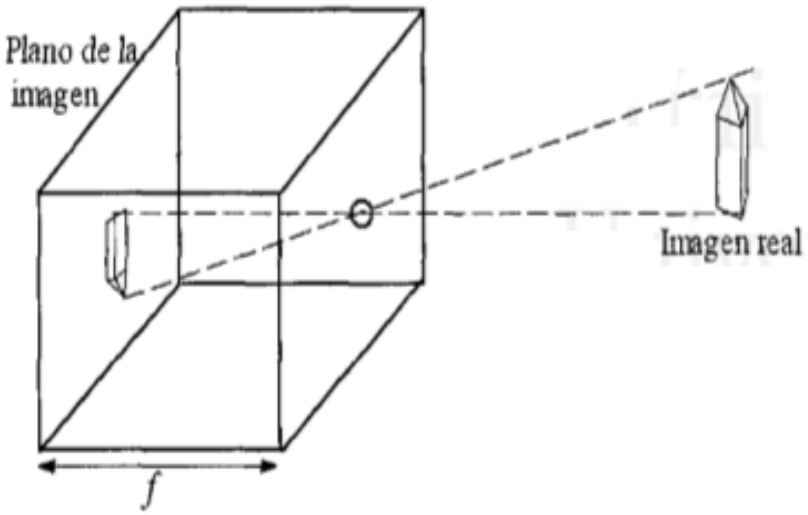


Figura 2.2: Modelo simplificado de una cámara.

infinitesimal, llamado centro óptico, situado a la llamada distancia focal de la imagen (f). Así, de todos los rayos luminosos que refleja un objeto en nuestra escena, estudiamos únicamente aquellos que pasan por este punto. Estos rayos, al proyectarse sobre el plano de la imagen forman el objeto invertido. Esta proyección se conoce como proyección de perspectiva. [1]

Si nos fijamos en la figura de abajo, sean (X,Y,Z) las coordenadas absolutas de cualquier punto de la escena en tres dimensiones. Suponiendo que todos los puntos de interés se encuentran en frente de la lente (Z >f). Nos interesa obtener la relación entre las coordenadas (x,y) del punto en el plano de la imagen entre las coordenadas (X,Y,Z) del punto real.

Haciendo uso de la relación entre triángulos semejantes, se observa que:

$$\frac{x}{f} = \frac{-X}{Z-f} = \frac{X}{f-Z} \tag{2.1}$$

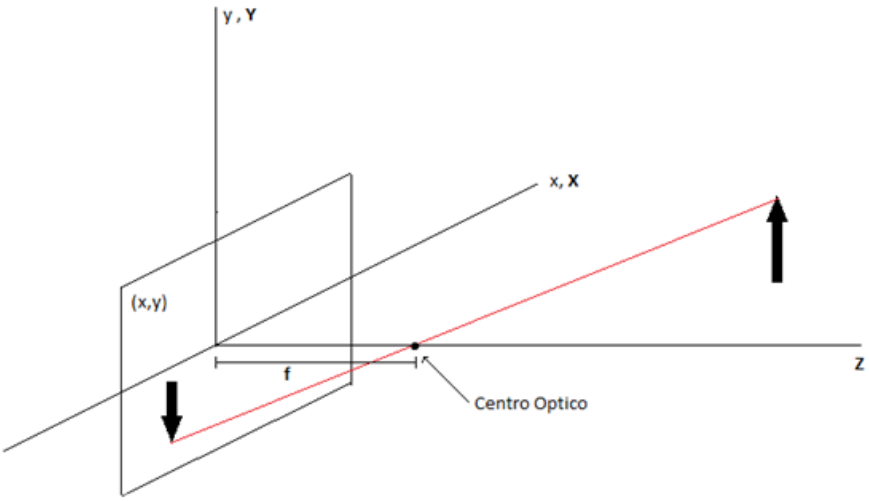


Figura 2.3: Definición gráfica de la distancia focal y centro optico.

$$\frac{y}{f} = \frac{-Y}{Z-f} = \frac{Y}{f-Z} \tag{2.2}$$

Con las ecuaciones 2.1 y 2.2 podemos deducir que para poder obtener la información tridimensional correspondiente a un pixel S(x,y), es necesario conocer al menos la distancia focal f y las coordenadas absolutas (X ,Y) o bien la distancia focal y la coordenada Z del punto respecto a la cámara. Lo habitual es calcular la coordenada Z del punto y, a partir de esta, obtener las otras dos coordenadas tridimensionales. En apartados posteriores veremos cómo se calcula la información de profundidad mediante visión estéreo.

Las ecuaciones 2.1 y 2.2 se pueden aplicar directamente pero, para realizar este cálculo puede ser útil definir una matriz de transformación homogénea. Se define la matriz de transformación de perspectiva como:

$$T = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & \frac{-1}{f} & 1 \end{pmatrix} \tag{2.3}$$

Definiendo los vectores en coordenadas homogéneas c_h para el sistema de referencia de la imagen y w_h. para el sistema de referencia absoluto:

$$\vec{c_h} = T \cdot \vec{w_h} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & \frac{-1}{f} & 1 \end{pmatrix} \cdot \begin{pmatrix} kX \\ kY \\ kZ \\ k \end{pmatrix} = \begin{pmatrix} kX \\ kY \\ kZ \\ \frac{-kZ}{f} + k \end{pmatrix} \tag{2.4}$$

Las coordenadas cartesianas de cualquier punto en el sistema de coordenadas de la cámara [Vision por computador, G pajares] vienen dadas si dividimos las tres primeras componentes de c_h entre el cuarto.

$$\vec{c} = \begin{pmatrix} \frac{fX}{f-Z} \\ \frac{fY}{f-Z} \\ \frac{fZ}{f-Z} \end{pmatrix} \tag{2.5}$$

Así, para convertir un punto cualquiera del sistema de la cámara al sistema absoluto basta con invertir la matriz T.

$$\vec{w_h} = P^{-1} \cdot \vec{c_h} \tag{2.6}$$

Por ultimo, es común utilizar un modelo geométrico equivalente al que aparece en la figura 2.3 para evitar que la imagen aparezca invertida, en este caso, el plano de la imagen se coloca en el centro óptico, además se tiene en cuenta que en la realidad el centro óptico no siempre se encuentra sobre el centro del sensor. De este modelo se obtienen las ecuaciones 2.7 y 2.8 que serán las que se usen para mapear los puntos en coordenadas de la imagen a las coordenadas tridimensionales del mundo.

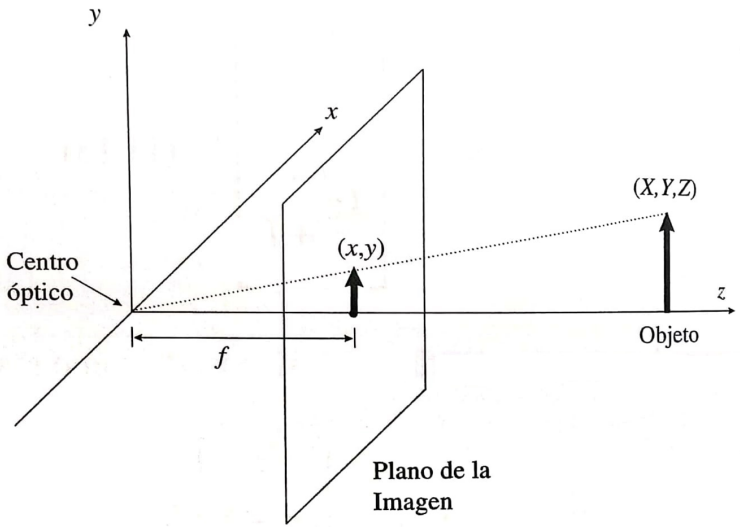


Figura 2.4: Modelo geométrico alternativo de proyección en perspectiva.

$$x - ppx = \frac{f_x \cdot X}{Z} \tag{2.7}$$

$$y - ppy = \frac{f_y \cdot Y}{Z} \tag{2.8}$$

Los nuevos parámetros introducidos son debidos a los aspectos reales de la configuración de la cámara (*ppx, ppy, fx y fy*) pertenecen a los parámetros intrínsecos y se detallaran en el apartado siguiente.

2.2.2 Visión estereoscópica

La visión estereoscópica es el proceso mediante el cual, a partir de dos imágenes desplazadas lateralmente, se obtiene la profundidad que le correspondería a cada píxel en la escena del mundo real. Para analizar este fenómeno comenzaremos con el modelado de un único sensor para posteriormente combinar dos sensores y, a partir del desplazamiento relativo entre los dos, poder calcular la información de profundidad de cada píxel.

Para entender cómo reconstruir la información tridimensional a partir de dos imágenes es necesario distinguir varios procesos. Todos estos tienen como fin obtener un mapa de disparidad de calidad que será el que permita obtener la escena en tres dimensiones. Se llama mapa de disparidad a una tercera imagen formada por la diferencia posicional de las características de la escena entre las dos imágenes fuente.

- Adquisición de las imágenes y calibración de los parámetros intrínsecos y extrínsecos en el modulo estereoscópico.
- Estudio de la geometría y aplicación de restricciones geométricas para simplificar la búsqueda de correspondencias entre pixeles.
- Extracción de las características.
- Buscar la correspondencia entre características de las imágenes para obtener un mapa de disparidad.
- Reconstrucción de la profundidad a partir de la imagen de disparidad.

2.2.2.1 Geometría del sistema de visión estereoscópico

En el apartado 2.2.1 veíamos que para la obtención de las coordenadas absolutas (X,Y) de un punto cualquiera perteneciente a una imagen, es necesario conocer la coordenada de profundidad (Z) y los parámetros intrínsecos de la cámara. Estudiando la geometría del sistema estereoscópica veremos cómo obtener Z.

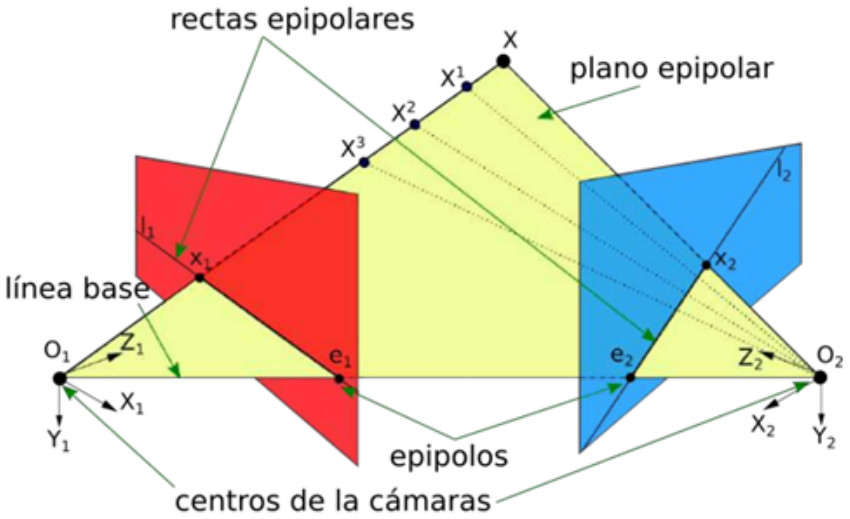


Figura 2.5: Definición de línea base, epipolos, plano y líneas epipolares.

Si miramos desde arriba nuestro sistema nos encontramos la siguiente geometría, donde Oi y Od son los dos puntos centrales de cada cámara y b la distancia entre ellas.

En la figura 2.6 podemos observar la relación de triángulos semejantes que se usa para obtener las dos ecuaciones de nuestro sistema. En este apartado obtenemos solo las ecuaciones para la componente X pero para la Y sería similar. Cabe destacar que si los ejes Zi y Zd no fueran paralelos al eje Z el sistema se complicaría ya que las líneas epipolares no serían horizontales.

$$\begin{cases} O_I : \dfrac{(\frac{b}{2}+X)}{Z} = \dfrac{x_I}{f} \rightarrow x_I = \dfrac{f}{Z}\cdot(X+\dfrac{b}{2}) \\ O_D : \dfrac{(\frac{b}{2}-X)}{Z} = \dfrac{x_D}{f} \rightarrow x_D = \dfrac{f}{Z}\cdot(X-\dfrac{b}{2}) \end{cases} \tag{2.9}$$

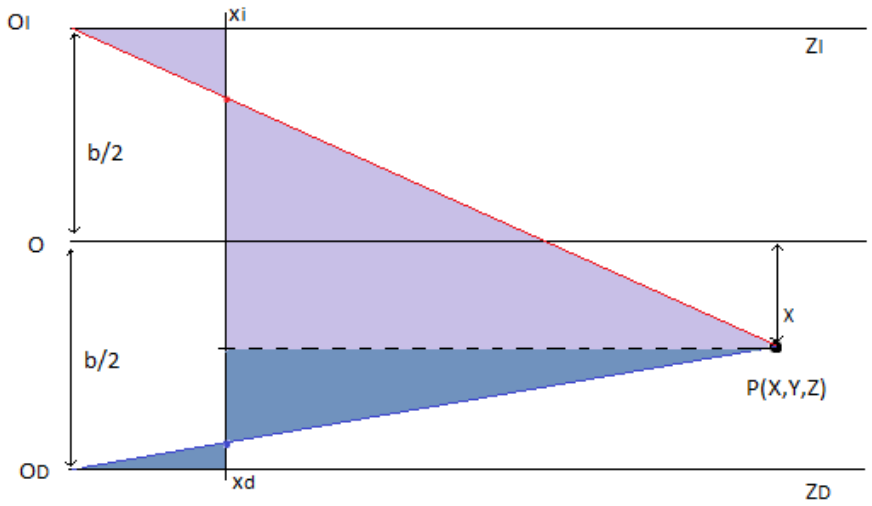


Figura 2.6: Sistema estereoscopio. Relación de triángulos semejantes.

Una vez formamos el sistema de ecuaciones, si conocemos previamente las características de nuestro sistema (f y b), tenemos dos ecuaciones y dos incógnitas (X, Z). Por lo tanto, podemos formar una nueva ecuación que dependa únicamente de Z :

$$x_I - x_D = \frac{f}{Z} \cdot b = d \tag{2.10}$$

$$Z = \frac{f \cdot b}{x_I - x_D} = \frac{f \cdot b}{d} \tag{2.11}$$

Observemos ahora que el problema consiste en determinar qué punto de una imagen **I(xi, yi)** corresponde al mismo punto en la otra imagen **D(xd, yd)**. Una vez resolvemos esta correspondencia podremos obtener la componente de disparidad (d) y con ella la información de la profundidad. En los puntos siguientes veremos que este problema es el más difícil de resolver, aunque la cámara utilizada cuenta con procesadores y un algoritmo propio que se encargarán de este cálculo.

2.2.2.2 Extracción de características y correspondencia entre pixeles.

Para encontrar la correspondencia entre los píxeles de dos imágenes, existen dos tipos de técnicas. Las **técnicas basadas en el área** y las **técnicas basadas en las características** de la imagen.

- Con las técnicas basadas en el área se buscan los puntos que tengan los niveles de intensidad más parecidos, para esto no solo se usa el nivel de intensidad de dicho punto, sino que también el de sus vecinos. Para acotar más esta búsqueda, el cálculo de la correspondencia se limita a los pixeles que se encuentren dentro de la misma línea epipolar. Esto proporciona un mapa de profundidad muy denso ya que idealmente se encuentra la correspondencia de todos los pixeles. Aun así, esta técnica es poco robusta ante distorsiones y es necesaria una calibración muy precisa para obtener las líneas epipolares.

- Las técnicas basadas en la extracción de características, sin embargo, no buscan obtener una correspondencia punto a punto, sino que buscan las estructuras significativas de una escena, como puntos en un borde, segmentos de un borde o regiones.

2.2.2.3 Mapeado 2D a 3D. Parámetros intrínsecos.

La relación entre las imágenes 2D y el sistema de coordenadas 3D viene descrito por los parámetros intrínsecos de la cámara. Esta relación la llamaremos proyección cuando tomamos un punto en el espacio de coordenadas 3D y lo asignamos a una ubicación en el espacio 2D. La deproyección, por el contrario, toma una ubicación de pixeles 2D y su valor de profundidad y le asigna un punto en el espacio de coordenadas 3D.
Los parámetros intrínsecos son:

- Ancho y alto de la imagen.
- Distancia Focal: viene dada por fx y fy como un múltiplo de el ancho y alto de la imagen.
- Centro de proyección: Las coordenadas de la proyección del centro óptico en la imagen que vienen dadas por ppx y ppy.
- Modelo de distorsión: Describe varios posibles modelos de distorsión y en el *SDK* de la cámara viene definido por los parámetros *model*[1] y *coeffs*[2]. Estos modelos sirven para corregir distorsiones radiales en lentes que no estén perfectamente alineadas.
  - None: La imagen no tiene distorsión, como si hubiese sido producida por una cámara estenopeica. Proporciona fórmulas cerradas tanto para la proyección como para la deproyeccion.
  - Brown-Conrady modificada: Este modelo proporciona una fórmula de forma cerrada para asignar puntos sin distorsión a puntos distorsionados, mientras que la asignación en la otra dirección requiere iteración o tablas de búsqueda.
  - Brown-Conrady inversa: Este modelo proporciona una fórmula cerrada para mapear desde puntos distorsionados hasta puntos no distorsionados, mientras que el mapeo en la otra dirección precisa de iteración o tablas de búsqueda.

Para usar el modelo de proyección sin distorsión despejamos las coordenadas (X,Y) de las ecuaciones 2.7 y 2.8 . Véase que la profundidad de cada pixel (Z) la obtenemos de la propia cámara con el cuarto canal:

$$X = \frac{(x - ppx)}{f_x} \cdot Z \tag{2.12}$$

$$Y = \frac{(y - ppy)}{f_y} \cdot Z \tag{2.13}$$

$$Z = Z \tag{2.14}$$

2.2.3 Estructuras de datos para representación tridimensional de imágenes.

Al igual que en las imágenes de dos dimensiones tenemos los píxeles como unidad mínima, existen varias formas de representar computacionalmente una escena tridimensional.

En las resonancias magnéticas o en videojuegos se usa el *vóxel* como unidad mínima, al igual que los pixeles, los voxeles no contienen su posición (x,y,z) en el espacio 3D, sino que esta se deduce por la posición del voxel dentro del archivo de datos. Cada *vóxel* puede contener información de color y de

[1] Nombre del modelo de distorsión utilizado.
[2] Coeficientes para el modelo de distorsión usado.

opacidad, textura, etc. Por lo tanto en esta matriz de datos existen datos incluso en los puntos de la escena en los que no hay nada.

En la presente aplicación, sin embargo, no se utiliza el concepto de *vóxel*. La representación mayormente utilizada en robótica es la nube de puntos, esto es, un conjunto de vértices almacenados en una lista de coordenadas de tres dimensiones (X,Y,Z) respecto a un sistema de coordenadas. En vez de representar una escena completa, lo que se representa son las superficies captadas por los sensores, LIDAR, cámara 3D, etc.

Para obtener la nube de puntos a partir de una imagen RGBD hay que aplicar las ecuaciones 2.12, 2.13 y 2.14 a cada pixel de la imagen. Para cada pixel (x,y) obtendremos una terna de valores (X,Y,Z) que corresponderán a cada punto en el espacio 3D.[3]

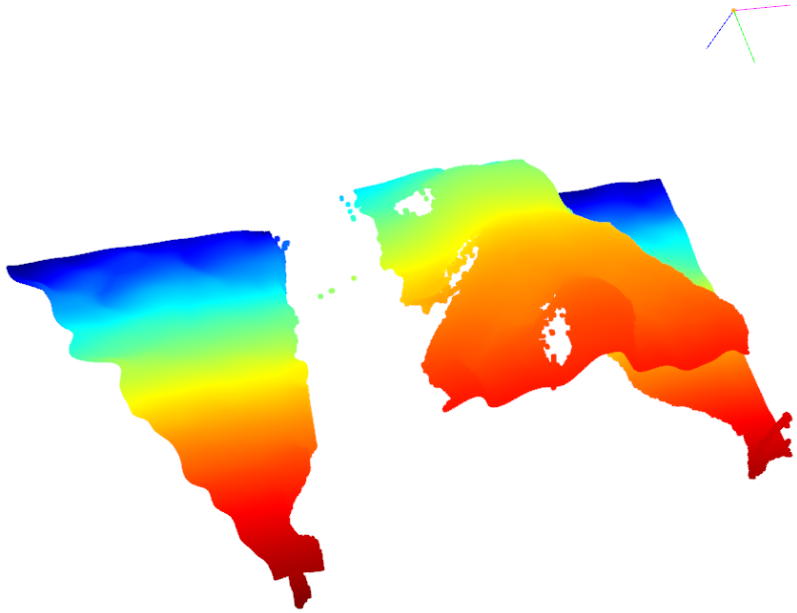


Figura 2.7: Ejemplo de nube de puntos del paciente tumbado en la camilla.

Esta representación permite hacer transformaciones del marco de referencia, análisis estadístico de la escena o búsqueda de planos. Estos algoritmos se estudiarán en capítulos posteriores.

2.3 Preprocesamiento de imágenes bidimensionales.

En los siguientes apartados se definirán las operaciones elementales que se pueden realizar sobre una imagen[4] . Estas serán la base sobre las que se sustentan los algoritmos de más alto nivel necesarios para la segmentación y extracción de características. Estas operaciones se pueden dividir en dos grandes grupos: Transformaciones del nivel de intensidad de cada pixel 2.3.1 y transformaciones geométricas 2.3.2.

2.3.1 Transformaciones del nivel de intensidad de los píxeles

Se modifica el valor de intensidad de cada pixel. A partir de una imagen E, obtendremos otra imagen resultado S.

[3]Cada punto puede considerarse un vector cuyas componentes están en metros.
[4]Para la realización de los ejemplos se ha utilizado el lenguaje Python con las librerías opencv, numpy y matplotlib.

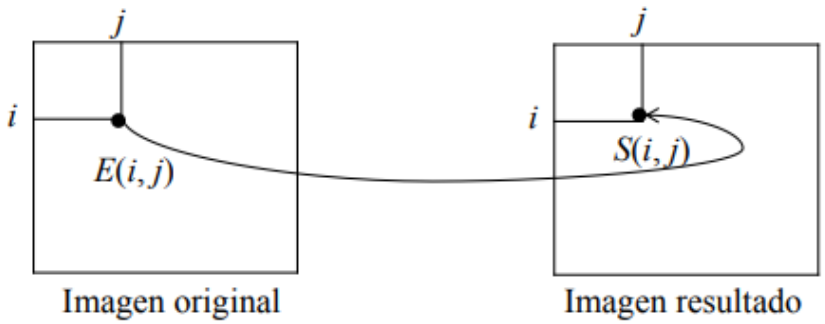


Figura 2.8: Transformación a nivel de pixel en una imagen.

Cada pixel viene representado por su coordenada en dos dimensiones S(i,j).

$$S(i,j) = f(E(i,j))$$

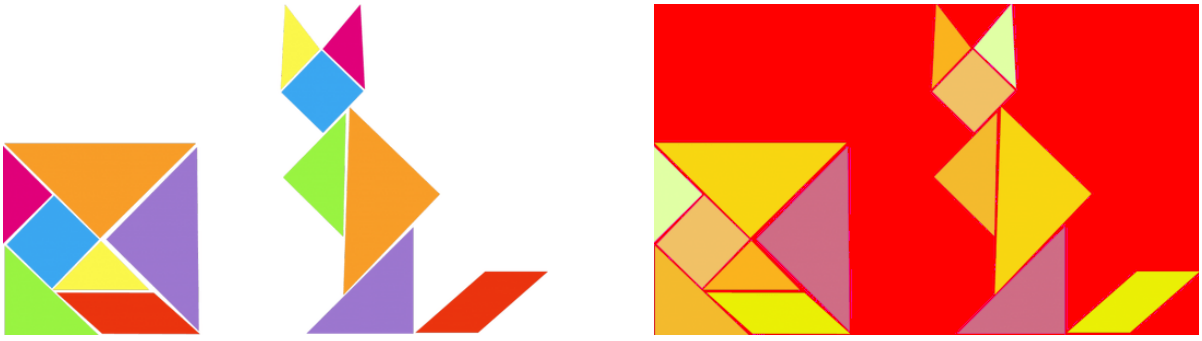
Esta transformación es muy útil para convertir entre espacios de color. En este trabajo no se detallarán todos los espacios de color pero, por ejemplo, para transformar de BGR[5] a HSV[6] se debe aplicar a cada pixel la siguiente función:

Sea MAX el valor máximo de los componentes (R, G, B), y MIN el valor mínimo de esos mismos valores:

$$H = \begin{cases} \text{no definido}, & \text{si } MAX = MIN \\ 60^\circ \times \frac{G-B}{MAX-MIN} + 0^\circ, & \text{si } MAX = R \text{ y } G \geq B \\ 60^\circ \times \frac{G-B}{MAX-MIN} + 360^\circ, & \text{si } MAX = R \text{ y } G < B \\ 60^\circ \times \frac{B-R}{MAX-MIN} + 120^\circ, & \text{si } MAX = G \\ 60^\circ \times \frac{R-G}{MAX-MIN} + 240^\circ, & \text{si } MAX = B \end{cases}$$

$$S = \begin{cases} 0, & \text{si } MAX = 0 \\ 1 - \frac{MIN}{MAX}, & \text{en otro caso} \end{cases}$$

$$V = MAX$$



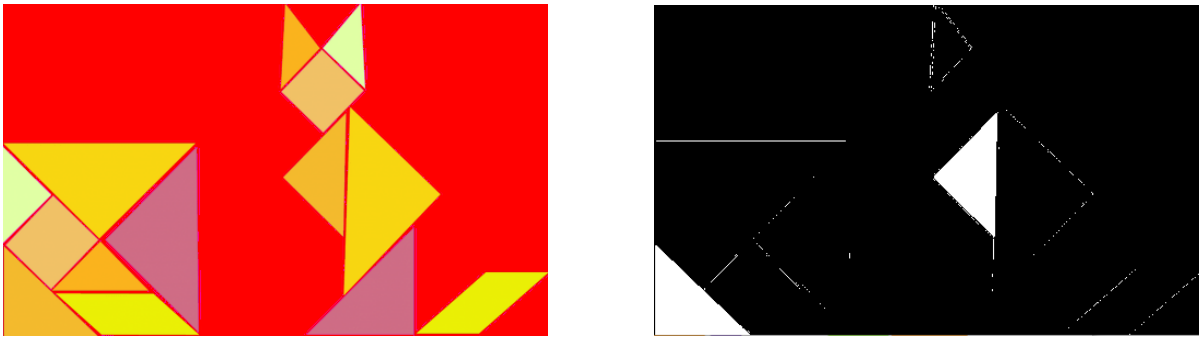
(a) (b)

Figura 2.9: Operación de cambio de espacio de color pixel por pixel. (a) Espacio de color BGR. (b) Espacio de color HSV.

[5]BGR: Blue-Green-Red o Azul-Verde-Rojo)
[6]HSV: Hue-Saturation-Value o Matiz-Saturacion-Valor

2.3.1.1 Operaciones con imágenes binarizadas.

Binarizar una imagen consiste en asignar un nivel máximo a los píxeles que estén por encima de un determinado umbral, y un nivel mínimo a los que estén por debajo. De este modo se obtienen imágenes a las que se pueden aplicar operaciones lógicas pixel por pixel. Esto es muy útil cuando se desea eliminar objetos de una imagen creando máscaras.



(a) (b)

Figura 2.10: Binarizacion para los valores HSV [30,0,0] y [50,255,255]. (a) Imagen original. (a) Imagen binarizada.

2.3.1.2 Convolución.

Otro tipo de transformación perteneciente a este grupo son las convoluciones, aquí la operación aplicada a cada pixel tiene en cuenta también la intensidad de los píxeles vecinos. Consiste en definir un *kernel* o núcleo, que no es más que una matriz, que se deslizará por la imagen original para obtener una nueva. Esta operación es usada en filtros o en las llamadas operaciones morfológicas.

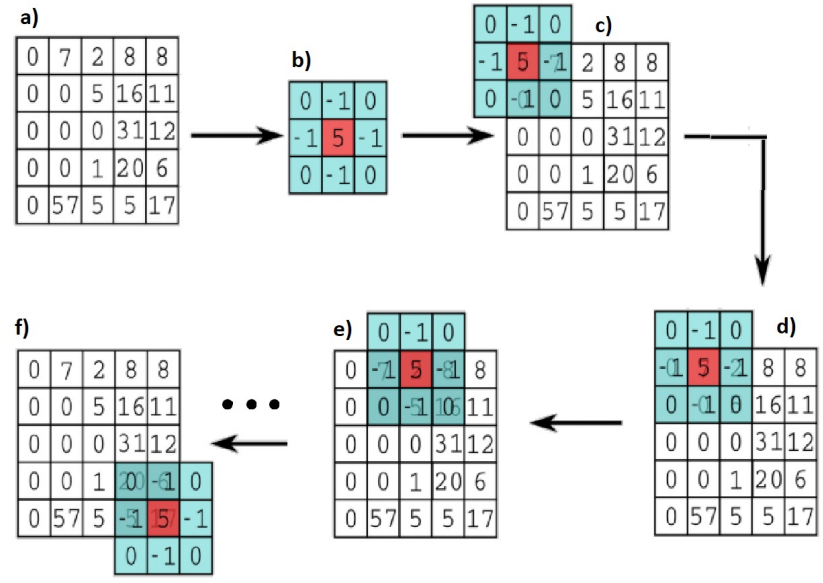


Figura 2.11: Proceso de convolución 2D sobre una imagen en niveles de gris.

$$\sum_{(m,n)=(0,0)}^{(W,H)} \sum_{(i,j)=(1,1)}^{(3,3)} K_{(i,j)} \cdot P_{(i+m,j+n)} \tag{2.15}$$

En la figura 2.11 se detalla esta operación que queda definida, para un kernel de dimensión (3x3) y para una imagen (WxH), por la ecuación 2.15:

- Imagen a la que se aplica la convolución. Una matriz de dimensión w x h en la que cada elemento representa el nivel de gris de un pixel.
- Definición del *kernel* que se pasará por la imagen
- Se aplica la siguiente operación al píxel (0,0):

$$p(0,0) = (5 \cdot 0) + (-1 \cdot 7) + (-1 \cdot 0) + (0 \cdot 0)$$

- Se mueve el *kernel* un píxel a la derecha y se hace la operación:

$$p(1,0) = (-1 \cdot 0) + (5 \cdot 7) + (-1 \cdot 2) + (0 \cdot 7) + (-1 \cdot 0) + (0 \cdot 5)$$

- e) Se mueve el *kernel* un pixel a la derecha y se hace la operación:

$$p(2,0) = (-1 \cdot 7) + (5 \cdot 2) + (-1 \cdot 8) + (0 \cdot 0) + (-1 \cdot 5) + (0 \cdot 16)$$

- f) Una vez se llega al último píxel se hace la operación:

$$p(w,h) = (0 \cdot 20) + (-1 \cdot 6) + (-1 \cdot 5) + (5 \cdot 17)$$

2.3.1.2.1 Gradiente y detección de bordes. Se pueden usar distintos *kernels* para obtener distintos resultados. A continuación se experimentará con alguno de los más comunes. Y también se definirá el concepto de primera y segunda derivada en los niveles de grises de una imagen para su uso en detección de contornos.

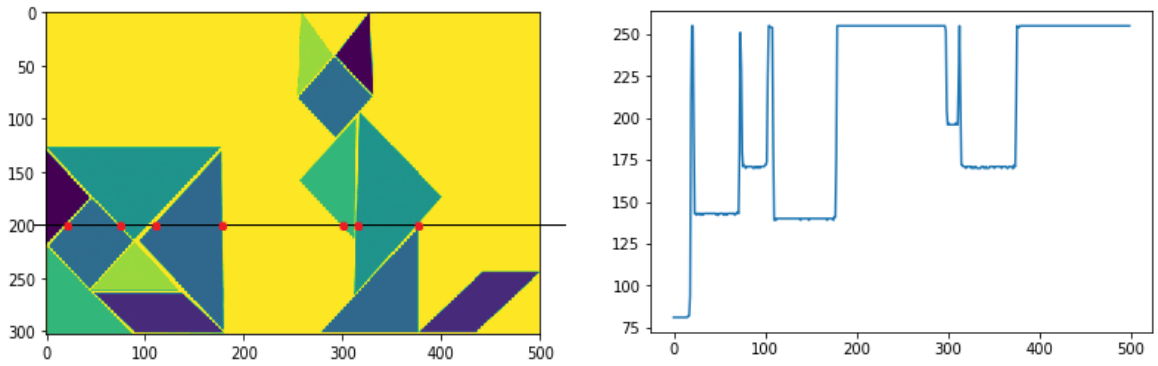
La filosofía básica de la mayoría de algoritmos de detección de bordes es el computo de operadores derivada locales (primera o segunda). Este concepto se visualiza muy bien en la figura 2.12. Haciendo zoom en una zona en la que haya un cambio de intensidad obtenemos la figura 2.13. Se observa que si se aplica la primera derivada a la curva de intensidad (restando a cada pixel su anterior), esta será cero para las regiones en que el nivel es constante y aumentará en las regiones en que haya un cambio de intensidad. Si se aplica la segunda derivada, esta sera cero en todos los puntos excepto en el comienzo y en el final de una transición.

Es interesante poder aplicar el concepto de derivadas locales a todos los píxeles de la imagen tanto en vertical como en horizontal y para esto se usa el concepto de convolución bidimensional. Ahora si se define el kernel *K1* y se desliza por la imagen, será equivalente a la operacion que se ha hecho en la figura 2.12. En este caso, en lugar de aplicar la derivada horizontal a una fila de píxeles, se aplica a toda la imagen. Con esto se detectarán los cambios de intensidad verticales, el resultado se ve en la figura 2.14.

$$K1 = \begin{pmatrix} 0 & 0 & 0 \\ -1 & 2 & -1 \\ 0 & 0 & 0 \end{pmatrix} \qquad K2 = \begin{pmatrix} 0 & -1 & 0 \\ 0 & 2 & 0 \\ 0 & -1 & 0 \end{pmatrix} \qquad K3 = \begin{pmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{pmatrix}$$

Podemos definir otro kernel *K2* para detectar los saltos de intensidad horizontales, esto seria equivalente a aplicar la primera derivada a cada columna de píxeles:

Y por último podemos hacer la derivada vertical y horizontal a la vez con un kernel del tipo *K3*:



(a) (b)

Figura 2.12: Véase que en los puntos pintados de rojo hay un cambio de intensidad en la imagen y coinciden con los picos en la gráfica 2.12b (a) Imagen original (un único canal de intensidad) mostrando una linea de exploración horizontal (linea 200). (b) Representación gráfica de los niveles de intensidad sobre la linea de exploración.

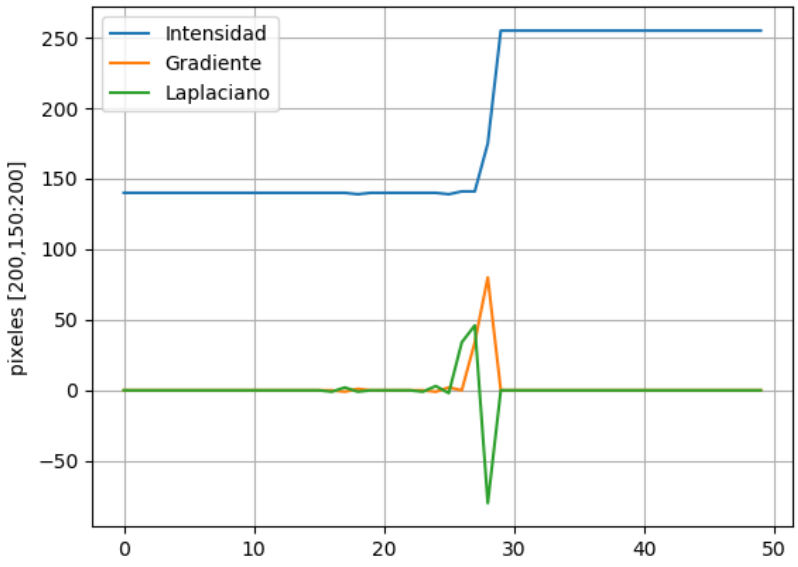


Figura 2.13: Concepto de primera y segunda derivada.

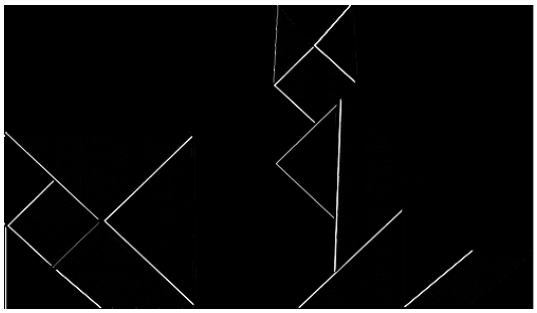


Figura 2.14: Aplicación del kernel K1.

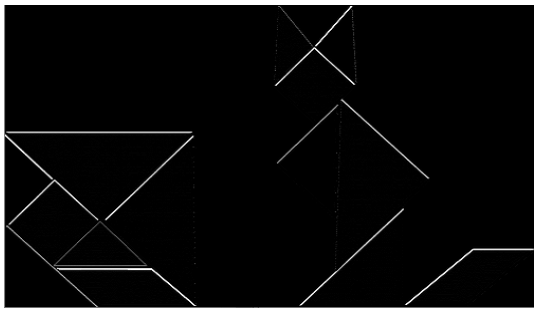


Figura 2.15: Aplicación del kernel K2.

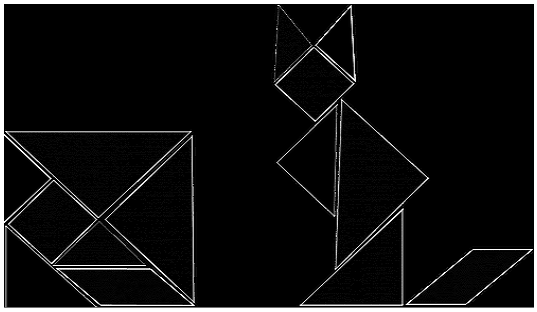


Figura 2.16: Aplicación del kernel K3.

2.3.1.3 Transformaciones morfológicas.

Estas transformaciones se llaman así porque pueden modificar la forma de los objetos de una imagen. Las podemos considerar como una convolución para imágenes binarizadas, en vez de realizar el producto acumulativo de cada elemento, se realiza una operacion lógica. Las dos operaciones principales son la erosión y la dilatación.

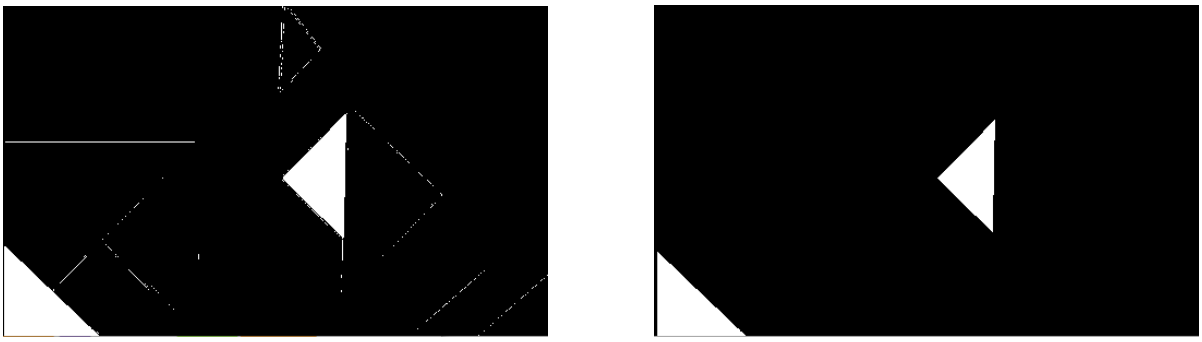
- Erosión (Fig. 2.17): Un pixel en la imagen original (1 o 0) se considerará 1 solo si todos los pixeles debajo del kernel son 1, de lo contrario se erosionará (se pondrá a 0). El resultado es que todos los pixeles cerca del límite de un objeto se descartarán, y por lo tanto, se reducirá su tamaño.

- Dilatación (Fig. 2.18): Es lo contrario a la erosión, un pixel es 1 si al menos un pixel debajo del kernel es 1. Por lo tanto, aumentará el tamaño del objeto.

- Apertura (Fig. 2.19): Se trata de una erosión para eliminar el ruido seguida de una dilatación para recuperar el tamaño original del objeto. Es muy útil para eliminar el ruido exterior al objeto en máscaras binarias.

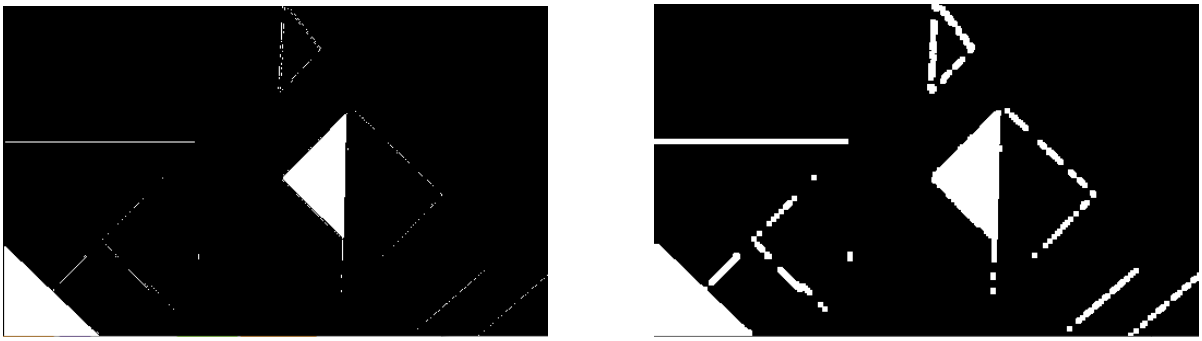
- Cierre (Fig. 2.20): Se trata de una dilatación para eliminar el ruido dentro del objeto seguida de una erosión para recuperar el tamaño original del objeto. Es muy útil para eliminar el ruido interior al objeto en máscaras binarias.

También se puede conseguir el contorno de un objeto haciendo la diferencia entre la dilatación y la erosión de una imagen. A esto se le llama gradiente morfológico porque los resultados son parecidos a los obtenidos con un gradiente convolucional.



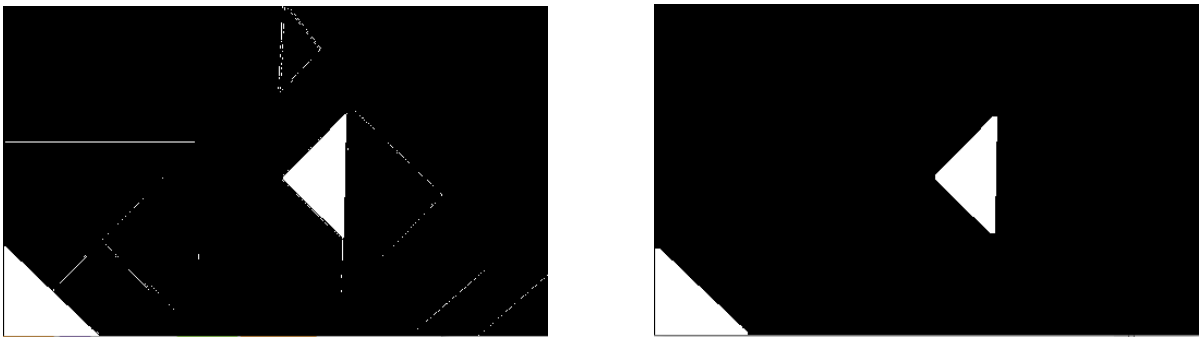
(a) (b)

Figura 2.17: Erosión. Se observa que se elimina el ruido pero se disminuye el tamaño del objeto. (a) Imagen original binarizada. (b) Imagen después de la erosión. .



(a) (b)

Figura 2.18: Dilatación. Se observa que se aumenta el tamaño del objeto y del ruido. (a) Imagen original binarizada. (b) Imagen después de la Dilatación. .



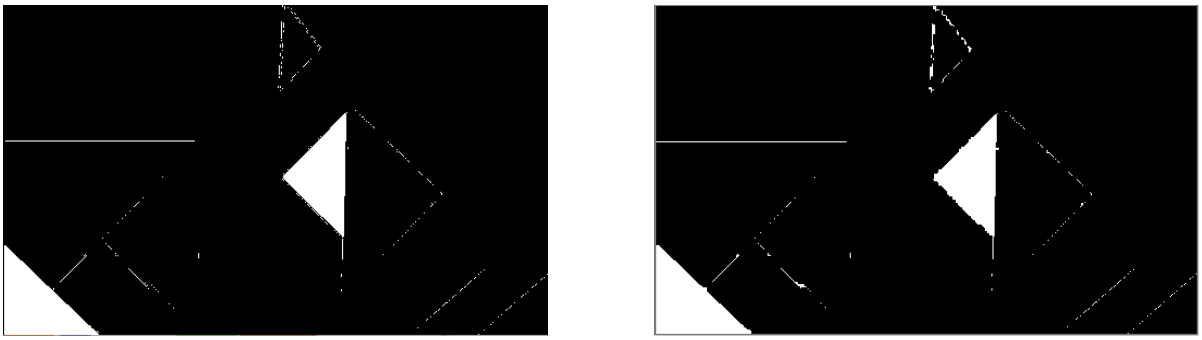
(a) (b)

Figura 2.19: Apertura. Se mantiene el tamaño del objeto y a la vez se elimina el ruido. (a) Imagen original binarizada. (b) Imagen después de la apertura. .

2.3.2 Transformaciones geométricas

Las transformaciones geométricas, como la rotación o traslación de una imagen, modifican la relación espacial entre píxeles. Estas transformaciones consisten en dos operaciones básicas:

- Redefinir la ubicación de los píxeles en el plano de una imagen.
- Interpolar los niveles de gris para asignar los valores de intensidad de los píxeles en la imagen transformada.



(a) (b)

Figura 2.20: Cierre. Se mantiene el tamaño del objeto y a la vez se elimina el ruido que pudiera haber dentro del objeto. (a) Imagen original binarizada. (b) Imagen después del cierre .

2.4 Preprocesamiento de nubes de puntos.

En el apartado 2.2.3 se definió el concepto de nube de puntos, en esta sección estudiaremos las posibles operaciones sobre este tipo de datos. Las transformaciones geométricas serán las aplicables a cualquier vector y se realizarán mediante matrices de transformación homogénea

2.4.1 Búsqueda de puntos cercanos

Cuando se trabaja con nubes de puntos puede resultar muy útil encontrar los puntos más cercanos a un punto dado. Esto se puede usar para encontrar correspondencias entre grupos de puntos o descriptores de características, o bien para definir el vecindario local alrededor de un punto. Por eso los algoritmos para realizar esta operación eficientemente se vuelven una operación muy común. Para realizar este tipo de búsquedas de forma eficiente los puntos se almacenan en una estructura de datos llamada árbol KD.

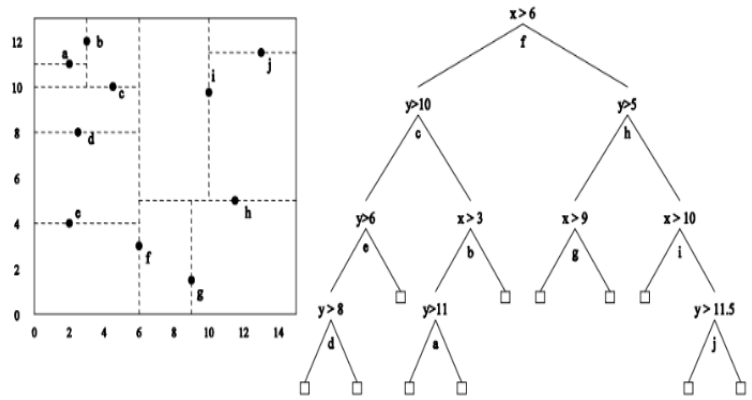
2.4.1.1 Árboles KD

Esta estructura de datos que se corresponde con un tipo de árbol binario de búsqueda, se usa para organizar los puntos de un espacio euclidiano de k dimensiones en subgrupos de puntos cercanos entre ellos, y así hacer más eficientes las búsquedas de grupos de puntos cercanos. Un árbol kd utiliza como ramas planos perpendiculares a uno de los ejes del sistema de coordenadas.

En la figura 2.21 se puede ver un ejemplo de árbol KD de dos dimensiones. La primera división (raíz) se coloca en la mediana en el eje horizontal $M_{e1} = 6$. La siguiente división se hace en el eje vertical, calculando la mediana de los dos grupos de puntos generados con la primera división ($M_{e2} = 10$ y $M_{e3} = 5$). Se repite este proceso ciclicamente hasta que no queden puntos sin su correspondiente hoja en el árbol.

Para realizar una búsqueda se recorre el árbol desde la raíz a los nodos finales u hojas. La búsqueda en estos arboles es muy eficiente, el máximo número de comparaciones que debemos realizar para saber si un elemento se encuentra o no en el árbol, se encuentra entre $log_2(N + 1)$ y N donde N es el número de nodos. Existen dos versiones de estos árboles, en la primera, que será la que se replica en este trabajo, las divisiones se colocan en la mediana de los puntos, con lo que se consigue que cada rama o división pase por un punto. En la segunda versión para colocar las divisiones se utiliza la media geométrica.

En nuestro caso estamos trabajando en el espacio tridimensional por lo que las ramas estarán colocadas en los ejes X, Y, Z.



21: Generación de un árbol kd

Figura 2.21: Ejemplo de dos dimensiones de un árbol KD.

1. Se elije uno de los ejes (En la figura 2.22 el eje Y) y se calcula la mediana de todos los puntos sobre ese eje. Se coloca la primera división sobre este punto mediana. La división será un plano que divida el espacio en dos partes.
2. La siguiente división se realiza en el eje Z.
3. La siguiente división se realiza en el eje X. Quedando el espacio dividido en ocho clusteres de puntos.
4. Se repiten los pasos anteriores hasta que por cada punto pase una rama.

2.4.2 Filtrado de las imágenes RGBD.

Cuando se usan cámaras de profundidad es común encontrar ruido debido a la oclusión entre objetos o a imperfecciones del sensor. El SDK de la cámara D435 cuenta con algoritmos de filtrado. En este apartado se hará un análisis de cada uno de ellos.

2.4.2.1 Submuestreo.

El submuestreo consiste en reducir el número de píxeles de profundidad. Se puede conseguir simplemente decimando el mapa de profundidad tomando, por ejemplo, un píxel por cada n píxeles.

2.4.2.2 Relleno espacial de agujeros.

Las imágenes de profundidad recogidas de la cámara pueden tener píxeles negros llamados agujeros. Estos agujeros son píxeles en vacíos ($Z = 0$) en lugares donde no corresponden y se pueden eliminar con métodos parecidos a los vistos en la sección referente a las convoluciones 2.3.1.2. El SDK cuenta con dos métodos.

- Metodo 1: Se usan los píxeles vecinos a la derecha o a la izquierda dentro de un radio específico para rellenar el orificio. En el SDK este bloque de procesamiento se llama "Filtro espacial".
- Metodo 2: Se usan los píxeles válidos de uno de los dos sensores estéreo para rellenar el orificio. Si el ruido aparece en uno de los sensores pero en el otro no, se puede usar uno de los sensores como referencia para cubrir los agujeros del otro. En el SDK este filtro se llama *Hole-filling*. Se ofrece la posibilidad de elegir entre las tres opciones de la figura 2.23.

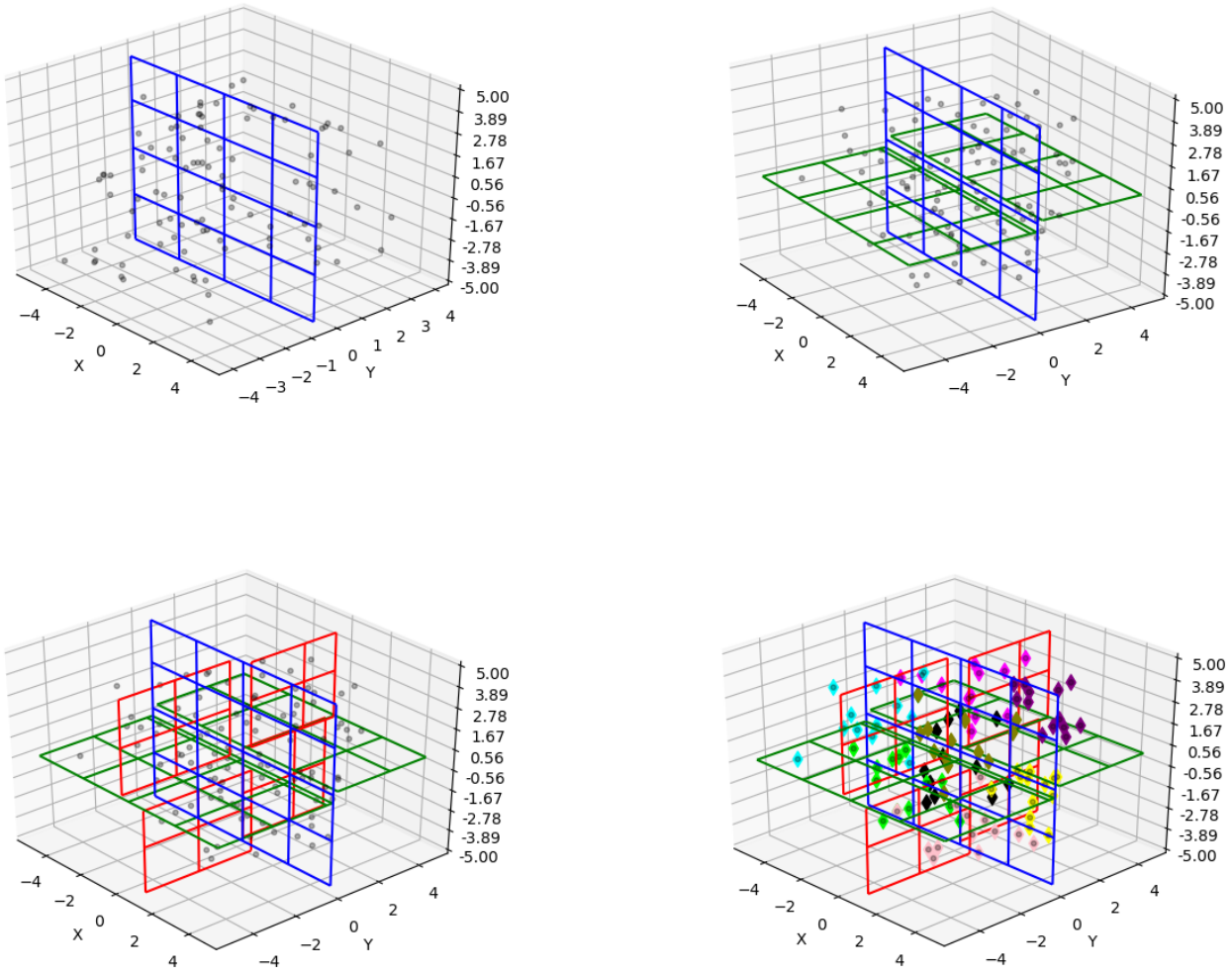


Figura 2.22: Primeros pasos para la creación de un árbol k3.

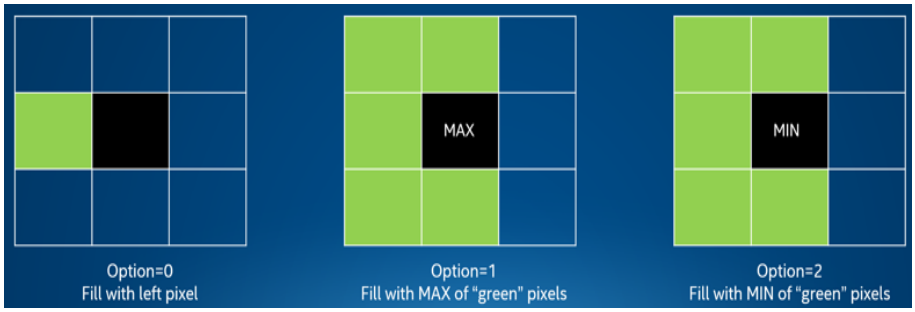


Figura 2.23: La aplicación del bloque "hole filling"tiene las tres opciones.

Para ver el ruido espacial y los efectos del filtro espacial es mejor la visualización en nube de puntos sin aplicar la textura de la imagen a color en la malla, que puede tender a ocultar datos.

2.4.2.3 Filtro temporal.

Volviendo a la tarea de mejorar el ruido del *frame* de profundidad, observamos que otra de las contribuciones al ruido muy importante es el ruido temporal. Los sensores de la cámara no tienen el conocimiento de los *frames* anteriores, por lo que cada imagen de profundidad se calcula de forma completamente independiente de los *frames* anteriores y este cálculo es completamente determinista. Sin embargo, hay ruido que puede provenir de las características inherentes del sensor, como puede ser el ruido ambiental (cambios en la iluminación), el movimiento o el ruido del proyector infrarrojo. Todas estas fuentes contribuyen a un ruido de profundidad constante en el tiempo, por lo que, en general, la calidad de los datos se puede mejorar capturando las imágenes a velocidades de muestreo más bajas, aumentando el tiempo de exposición, o bien aplicando un promediado temporal a cada pixel.

El fabricante recomienda usar el filtrado de promediado temporal siempre que sea posible, teniendo en cuenta que esto será posible solo cuando desarrollemos aplicaciones de vídeo continuo. Dicho filtro consiste en una media móvil exponencial (EMA). En la que se le otorga un peso diferente a cada elemento de la ventana temporal, así, se le dará más peso a los píxeles más actuales.

Los filtros predefinidos en el SDK cuentan con dos parámetros α y δ. De forma que cuando $\alpha = 1$, se comportara como un fltro cero, pero si se reduce $\alpha = 0$, se aumentará el promediado y la suavidad obtenida. El parametro δ funciona como un umbral, de esta manera, se intenta reducir el suavizado temporal cerca de los bordes y excluir los agujeros en el promedio.

2.4.2.4 Secuencia recomendada de filtrado.

El fabricante de la cámara recomienda usar los bloques de procesado en un orden específico (fig. 2.24), además se recomienda aplicar algunos filtros sobre los *frames* de disparidad en lugar de aplicarlo en el *frame* de profundidad final. Por eso, después de aplicar el submuestreo a los píxeles de profundidad, se recomienda convertir la imagen al dominio de la disparidad para ejecutar el filtro espacial y el temporal, después volver otra vez al dominio de la profundidad y aplicar el filtro *hole-filling*.

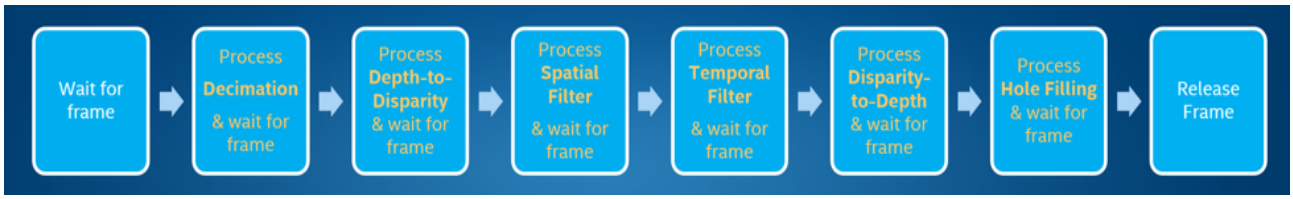


Figura 2.24: Secuencia de aplicación de los filtros recomendada por Intel.

2.4.3 Recortado de nubes de puntos.

Uno de los métodos que se han usado en el proyecto, por su simplicidad a la hora de eliminar información no necesaria de la escena, es el recortado de las nubes de puntos por límites fijos. Aquí, aprovechando que se conoce bien la escena de la imagen y que siempre va a tener la misma estructura, se eliminan las partes de la escena que no nos interesan.

Por ejemplo, si se desea detectar el cuerpo de un paciente tumbado en una camilla y la imagen se toma desde arriba 2.25, podemos suponer que la región de interés de nuestra escena se encontrará en el centro. Conociendo el ancho de la camilla, se pueden descartar los puntos que se encuentren fuera de los laterales de esta como en la figura 2.26.

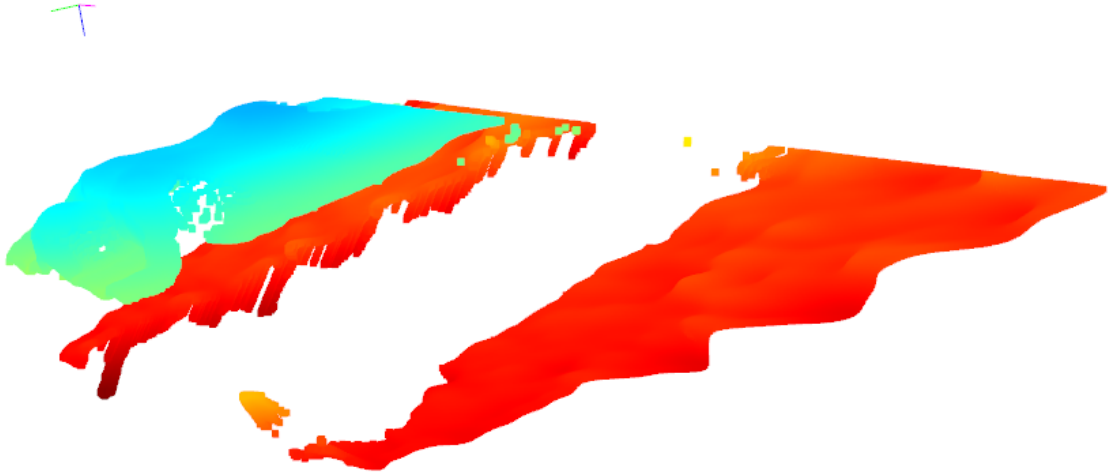


Figura 2.25: Nube de puntos antes de ser recortada.

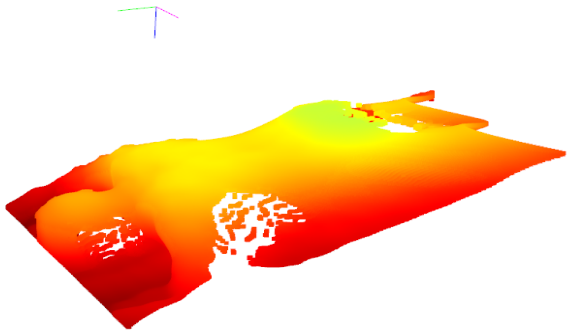


Figura 2.26: Nube de puntos después de ser recortada por unos límites fijos.

No siempre es posible quedarnos con la información necesaria recortando por límites fijos, ya sea porque no conozcamos previamente la estructura de la escena o bien porque la cámara no siempre se encuentre fija. Esto se puede solventar utilizando algoritmos de segmentación y extracción de características para encontrar los límites por los que podremos recortar. En este ejemplo una solución podría consistir en identificar los límites de la camilla mediante el detección de lineas rectas con la transformada de Hough. En la sección siguiente 2.5 se estudiarán algunos métodos útiles para este tipo de tareas.

2.5 Segmentación y extracción de características.

Cuando se habla de segmentación en el campo de la visión por computador, se refiere al proceso de dividir una imagen en grupos de píxeles (2D o 3D) pertenecientes a un mismo objeto o región. En el desarrollo del proyecto esto sería la segmentación de los píxeles pertenecientes al paciente y posteriormente la identificación de las distintas partes del cuerpo, como pueden ser los hombros o la cabeza.

Veremos que la segmentación es un proceso que se vuelve más complicado cuando no se conoce previamente la distribución de los objetos en la escena de la imagen. Sin embargo, si conocemos previamente la estructura de la escena, podemos usar esto a nuestro favor. Se vuelve muy importante en esta fase haber realizado un preprocesamiento correcto de las imágenes antes.

En este apartado se verán distintas formas de segmentación tanto de imágenes 2D como de escenas 3D,

se hará una comparación de los distintos métodos y se estudiara cómo alguno de ellos se complementan.

2.5.1 Aprendizaje no supervisado para agrupamiento.

Se pueden aplicar algoritmos de agrupamiento no supervisado para dividir la escena en los distintos objetos que la contienen. El algoritmo estudiado en este proyecto es K-medias que pertenece a los métodos de aprendizaje automático no supervisados. En la figura 2.27 podemos distinguir tres objetos a simple vista, el problema se reduce a conocer a qué objeto pertenece cada punto. Para esto se realizan los siguientes pasos.

1. Se definen tantos centroides como grupos se quieran diferenciar como en la figura 2.27a. Estos centroides se suelen definir de forma aleatoria.

2. Se calcula la distancia euclidea al cuadrado de cada punto a cada centroide. Así si hay i centroides y j puntos en el espacio tridimensional calcularemos el centroide más cercano a cada punto:

$$min(norm(p_j - c_i)^2) \tag{2.16}$$

3. Se calcula la media de los puntos que compartan el centroide más cercano y se actualiza la posición de cada centroide a esta media.

4. Se repiten los dos pasos anteriores hasta que los centroides ya no cambien más de posición. En la figura 2.27b se observa cómo se van desplazando los centroides. Y en la figura 2.27c se muestra los puntos pertenecientes a cada *cluster*.

En las mejores implementaciones de k-medias se suele ejecutar el algoritmo varias veces y se elije como resultado final el mejor de ellos en términos de inercia de los centroides. En el apéndice A A.1.5 está la prueba de concepto en Python de este algoritmo.

2.5.2 Detección de contornos.

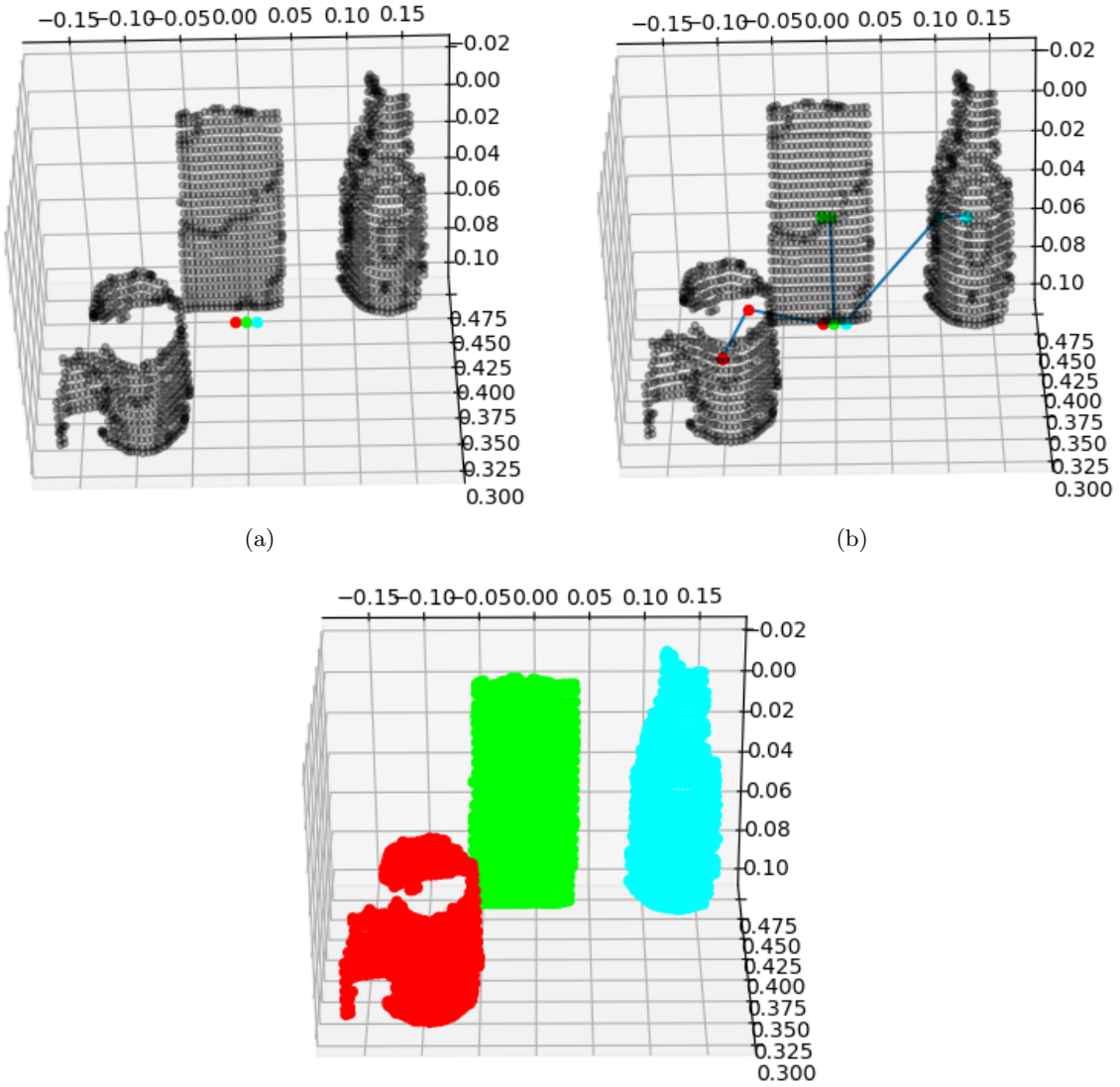
Aplicando el concepto de convolución visto en la sección 2.3.1.2 se puede hacer una detección precisa de los píxeles en los que hay un salto de intensidad, obteniendo así los bordes y límites de los objetos. Para ello, el algoritmo más extendido es el desarrollado por John F. Canny en 1986. Este consta de tres pasos.

1. Suavizado y obtención del gradiente. Se aplica un filtro gaussiano para reducir el ruido y después se calcula el gradiente vertical y horizontal obteniendo así dos imágenes.

2. Supresión no máxima. Se genera una imagen con los bordes reducidos a un píxel de grosor con las dos imágenes del proceso anterior.

3. Umbralización por histéresis. En lugar de aplicar un único umbral se aplica un umbral máximo y uno mínimo. Si el valor del píxel es mayor que el umbral máximo, el píxel es considerado parte del borde. Si es menor que el umbral mínimo, el pixel se clasifica como no borde. Y si el valor está entre los dos umbrales, será parte del borde si uno de sus píxeles vecinos es también parte del borde.

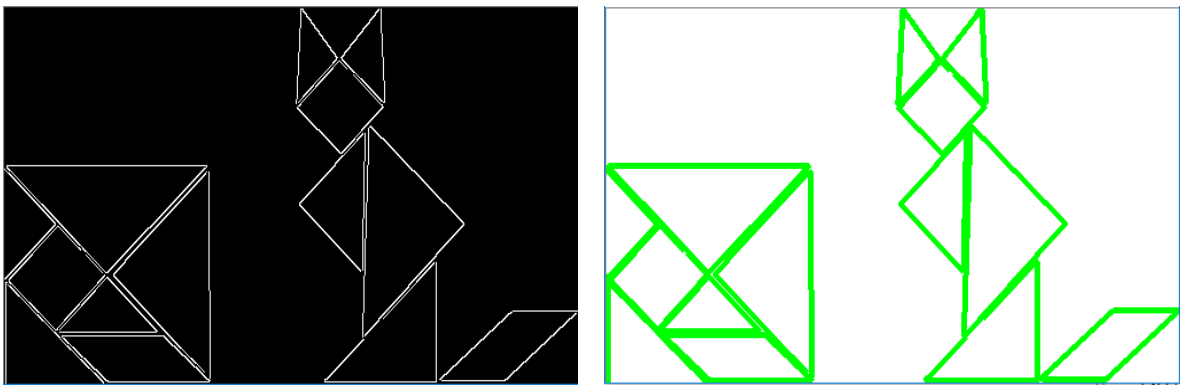
En la figura 2.28a se observa el resultado de aplicar el algoritmo de Canny a la imagen de muestra. Posteriormente, con un algoritmo de agrupación de píxeles según sus píxeles vecinos, se puede separar e identificar cada contorno como en la figura 2.28b.



(a) (b)

(c)

Figura 2.27: Funcionamiento del algoritmo k-medias. (a) Se definen los centroides iniciales. (b) Se desplazan los centroides hasta que se encuentran los *clusters*. (c) *Clusters* encontrados.



(a) (b)

Figura 2.28: Ejemplo de detección de contornos. (a) Resultado de aplicar el algoritmo de Canny . (b)Contornos dibujados con la función para detectar contornos de la librería opencv.

2.5.3 Detección de geometrías. Transformada de Hough.

En 1962 Paul Hough patentó una técnica para la detección de rectas en una imagen. Más tarde se propuso una modificación para detectar cualquier geometría parametrizable matemáticamente. En este apartado se estudiará gráficamente la transformada original de Hough para detectar líneas rectas[7] .

Una vez que se ha aplicado algún algoritmo de detección de bordes y tenemos la imagen binarizada, podemos ver esta imagen como un conjunto de puntos como los de la figura 2.29. Para cada punto de la

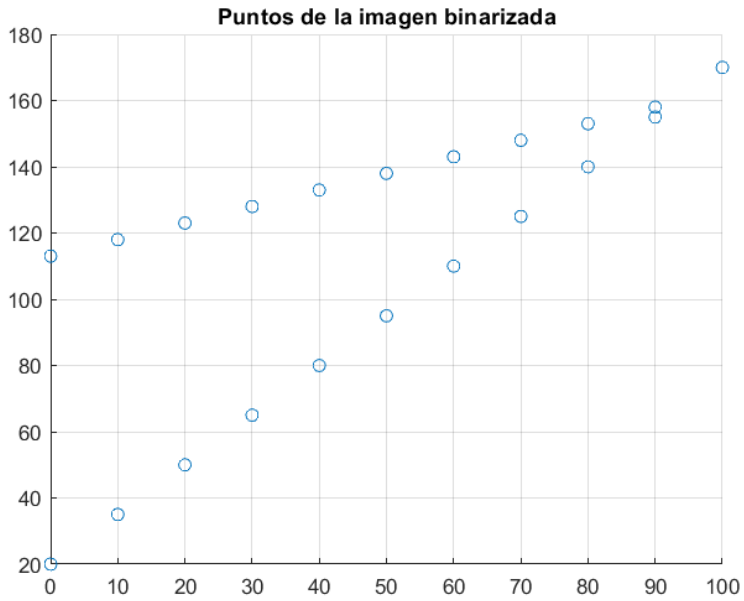


Figura 2.29: Representación simplificada de una imagen binaria que contiene dos rectas.

imagen perteneciente a un borde, se dibujarán un conjunto de las rectas ($R1_i$) que pasan por ese punto con una pendiente de un ángulo α entre -90 y 90 grados. De estas rectas conocemos los puntos de la ecuación 2.17, Donde $[x_i, y_i]$ son las coordenadas de cada punto perteneciente al borde.

$$\begin{aligned} p1_i &= [x_i, y_i] \\ p2_i &= p1_i + [x_i \cdot cos(\alpha), y_i + \cdot sin(\alpha)] \end{aligned} \tag{2.17}$$

Posteriormente, se obtienen y se trazan los segmentos de recta perpendiculares a las líneas del paso anterior que pasen por el origen, estos segmentos corresponden a las líneas discontinuas de la figura 2.31 y las llamaremos $R2_i$. Para cada punto $p1_i$ tendremos una lista con la pendiente y el módulo de estos segmentos. La ecuación de la recta perpendicular a otra y que pasa por un punto es:

$$R2_i: \qquad y - y_0 = -\frac{1}{m_i} \cdot x - x_0 \tag{2.18}$$

Donde m_i es la pendiente de la recta $R1_i$. Mediante un sistema de ecuaciones podemos encontrar el punto de corte entre las rectas $R1_i$ y sus perpendiculares $R2_i$.

$$\begin{cases} R1_i: & x \cdot m_i - x1_i \cdot m_i = y - y1_i \\ R2_i: & y = -\frac{1}{m_i} \cdot x \end{cases} \tag{2.19}$$

[7]Las gráficas han sido obtenidas con MATLAB.

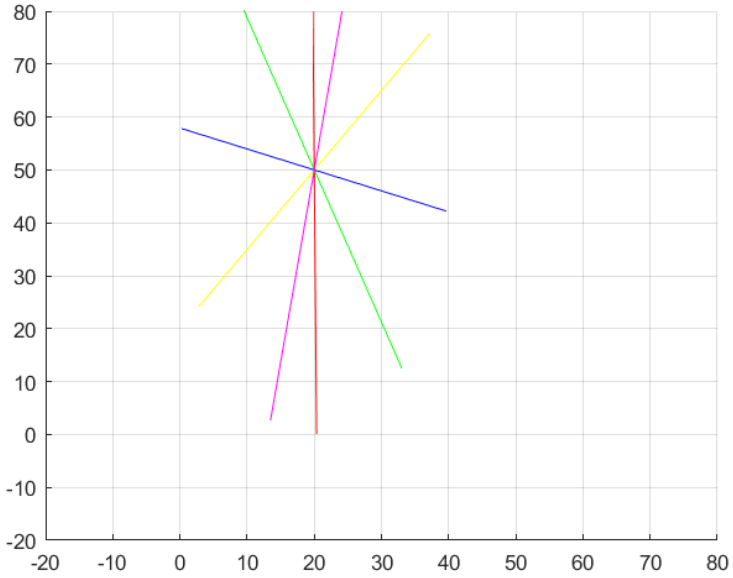


Figura 2.30: Líneas que pasan por un punto perteneciente a un borde. Se han trazado con los ángulos $[-89, -49, -9, 31, 71]$ grados. Este es el primer paso de la transformada de Hough y se hace para cada punto de la imagen binarizada.

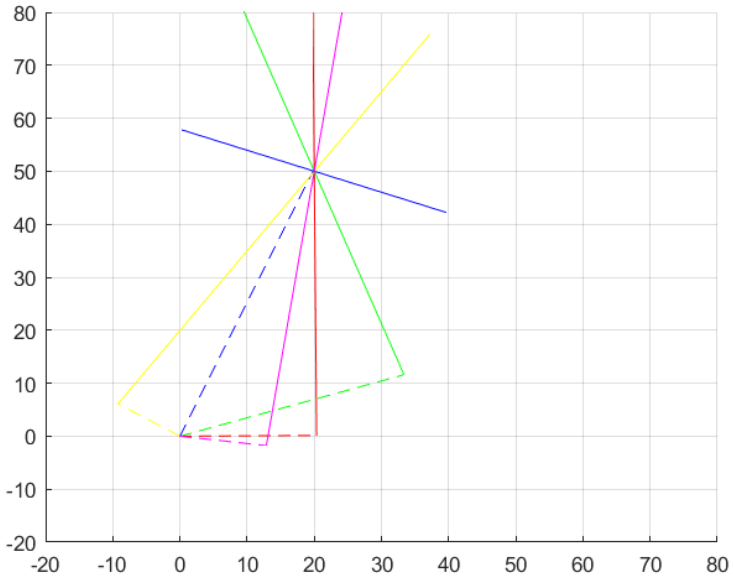


Figura 2.31: Rectas perpendiculares a las anteriores que pasan por el origen.

Despejando, encontramos el punto de cruce $p3_i$ y ya podemos calcular el array de ángulos θ y distancias ρ para cada punto.

$$\theta = atand(\frac{y3_i}{x3_i}) \qquad \rho = x3_i \cdot cos(\theta) \tag{2.20}$$

Una vez tenemos estos *arrays*, los pares (θ, ρ) que más se repitan entre todos los puntos serán los correspondientes a una recta. Una forma de visualizar estos pares más repetidos es graficar para cada punto todos los valores (θ, ρ), se obtendrá un conjunto de funciones sinusoidales y en los puntos en los que se crucen tendremos los parámetros de las rectas. En la figura 2.32 se ve muy bien los dos puntos de

corte en el llamado espacio de Hough.

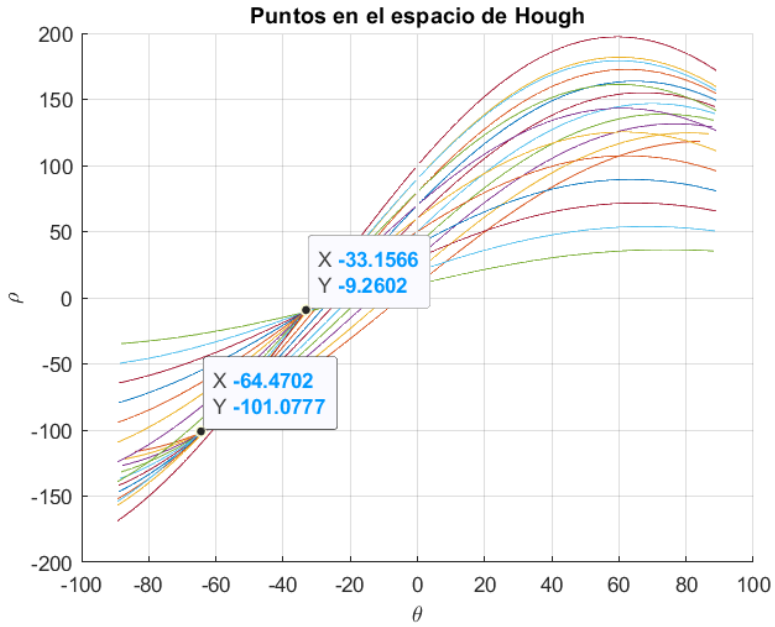


Figura 2.32: Gráfica del espacio de Hough para los puntos de ejemplo de la figura 2.29. Cada sinusoide corresponde a un punto de la imagen binarizada, los puntos de corte entre las sinusoides identifican los valores de (θ, ρ) para los cuales se ha encontrado una recta.

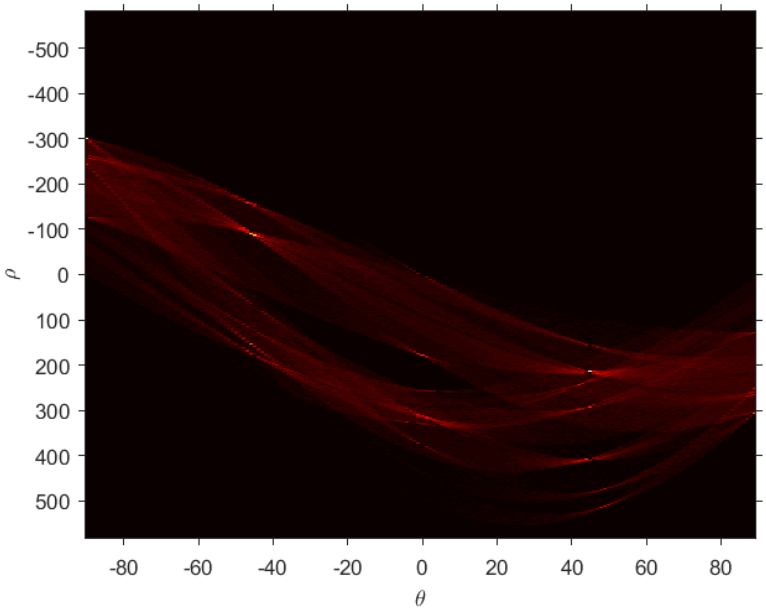


Figura 2.33: Transformada de Hough para cada uno de los bordes de la imagen binarizada por Canny de la figura 2.28a. Los puntos con mayor intensidad dan las posibles rectas encontradas.

2.5.4 Ajuste de planos.

En esta sección se estudiarán algunos métodos propuestos para la detección de planos en imágenes RGBD y en nubes de puntos. Esto es útil cuando se desea conocer la orientación de la cámara respecto a los planos

principales o cuando se desean eliminar las paredes de la escena con el fin de realizar la segmentación correctamente.

2.5.4.1 Ecuación paramétrica del plano.

Si conocemos tres puntos pertenecientes a un plano, por ejemplo tres puntos situados en el suelo o en las paredes de la escena, podemos encontrar los parámetros que definen el plano. Para ello usamos la ecuacion implícita del plano, que viene dada por tres puntos coplanares:

$$\vec{P_1} = (x_1, y_1, z_1) \qquad \vec{P_2} = (x_2, y_2, z_2) \qquad \vec{P_3} = (x_3, y_3, z_3)$$

Para deducir esta ecuación, primero definimos tres vectores. Dos de ellos serán linealmente independientes y el otro sera linealmente dependiente, ya que podrá expresarse como una operación lineal entre los otros dos vectores. Este tercer vector será linealmente independiente si no pertenece al plano y linealmente dependiente si sí que pertenece.

$$\begin{aligned} \vec{P_1P} &= ((x - x_1), (y - y_1), (z - z_1)) \\ \vec{P_1P_2} &= ((x_2 - x_1), (y_2 - y_1), (z_2 - z_1)) \\ \vec{P_1P_3} &= ((x_3 - x_1), (y_3 - y_1), (z_3 - z_1)) \end{aligned}$$

Por lo tanto, el determinante de los tres vectores se hará cero para todo $P = (x, y, z)$ perteneciente al plano.

$$\Pi = \begin{vmatrix} (x - x_1) & (y - y_1) & (z - z_1) \\ (x_2 - x_1) & (y_2 - y_1) & (z_2 - z_1) \\ (x_3 - x_1) & (y_3 - y_1) & (z_3 - z_1) \end{vmatrix} = 0$$

Desarrollando el determinante anterior podemos dejar la ecuación en función de los parámetros que definirán nuestro plano:

$$\theta_0 + x \cdot \theta_1 + y \cdot \theta_2 + z \cdot \theta_3 = 0$$

Donde:

$$\begin{aligned} A &= x_2 - x_1 \qquad B = y_2 - y_1 \qquad C = z_2 - z_1 \\ D &= x_3 - x_1 \qquad E = y_3 - y_1 \qquad F = z_3 - z_1 \end{aligned}$$

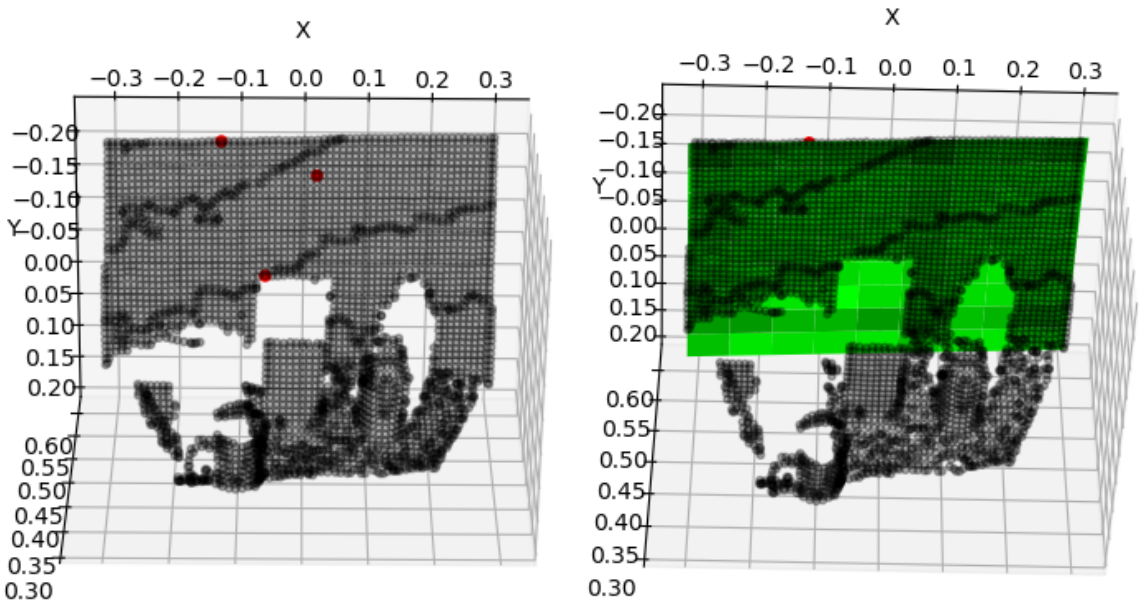
$$\begin{aligned} \theta_0 &= x_1 \cdot (A - F) + y_1 \cdot (B - E) + z_1 \cdot (C - D) \\ \theta_1 &= A - F \qquad \theta_2 = B - E \qquad \theta_3 = C - D \end{aligned}$$

Es importante recordar que los parámetros $\theta_1, \theta_2, \theta_3$ también son las coordenadas del vector director ortogonal al plano. Esto será útil para conocer la orientación de los planos, por ejemplo, la orientación de la cámara respecto a las paredes.

El cálculo de los parámetros del plano que pasa por tres puntos no ofrece una búsqueda robusta de los planos principales de la escena por si solo pero más adelante se verá que es una de las operaciones fundamentales para lograr este objetivo.

2.5.4.2 Regresión Lineal.

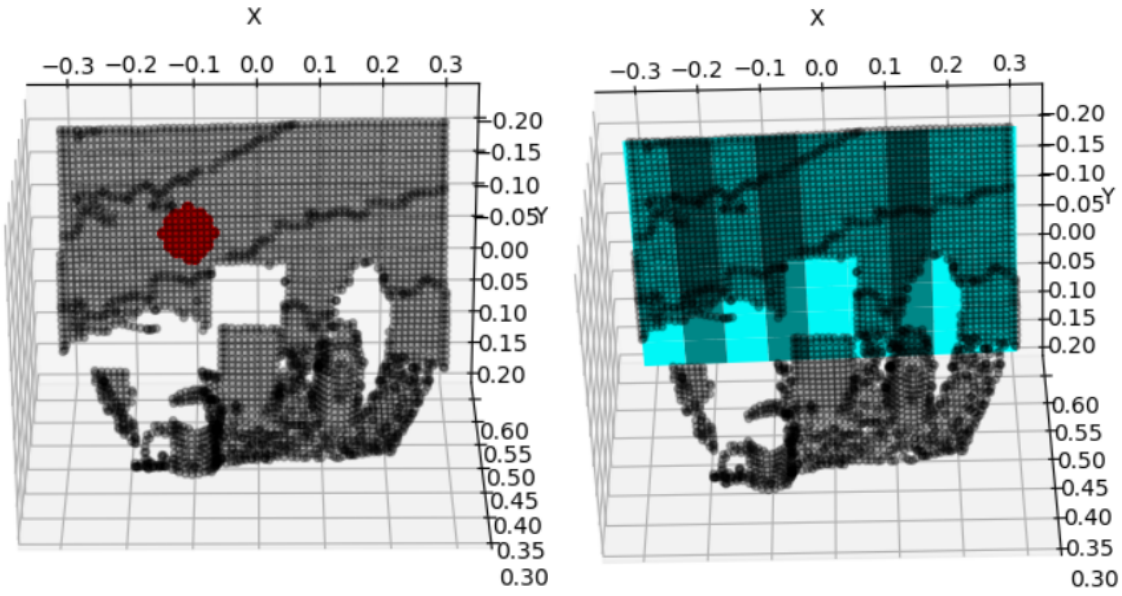
Otro método estudiado para el cálculo de los parámetros de un plano es la regresión lineal. En este caso la entrada ya no son únicamente tres puntos, puede ser un grupo de puntos. Con este método



(a) (b)

Figura 2.34: Detección de un plano que pasa por los tres puntos pintados en rojo mediante la ecuación del plano.

se busca minimizar la función de coste en función de los parámetros de una hipótesis lineal. Como estamos trabajando en el espacio euclidiano 3D, la función hipótesis será la ecuación correspondiente a las coordenadas Z de un plano. Y la función de coste será el error cuadrático medio (MSE) 2.22 entre la hipótesis y los puntos que se supone, pertenecen al plano.



(a) (b)

Figura 2.35: Detección de un plano que pasa por 50 puntos pintados en rojo mediante regresión lineal.

Con este método se obtienen planos que se corresponden más con la realidad que con el método de la sección 2.34b, ya que se tiene en cuenta un número más representativo de la superficie a ajustar. Por otro lado, este método es computacionalmente más costoso. En la sección 2.5.4.3 se verá un algoritmo que usa la regresión lineal o la ecuación del plano para encontrar los parámetros sin tener que conocer

los puntos que pertenecen al plano previamente.

$$h_\theta(x) = \theta_0 + x \cdot \theta_1 + y \cdot \theta_2 = \vec{\Theta} \cdot \vec{x^{(i)}} \tag{2.21}$$

$$J(\theta) = \frac{1}{2m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - z^{(i)}) \tag{2.22}$$

2.5.4.3 RANSAC.

Una forma de determinar los planos principales de una escena es usando el llamado Random Sample Consensus (RANSAC). Este algoritmo permite calcular los parámetros de un modelo a partir de un conjunto de datos aunque estos contengan elementos que no pertenezcan al modelo. Por lo tanto, aplicado a una nube de puntos, es una forma de encontrar los planos principales de esta. El algoritmo es iterativo y consta de cinco partes diferenciadas.

1. Se escoge aleatoriamente un número de puntos. En la implementación del algoritmo desarrollada para este trabajos se escoge un punto aleatorio de toda la nube de puntos, después se buscan los cincuenta puntos más cercanos en el *kdtree* previamente calculado, y por último se cogen tres puntos al azar entre estos cincuenta puntos.

2. Se calculan los parámetros del modelo para los puntos escogidos. Este paso se puede realizar con la ecuación del plano que pasa por tres puntos o mediante regresión lineal.

3. Cada uno de los demás puntos se prueba contra el modelo ajustado para determinar si pertenecen o no este. Se considera un modelo bueno cuando suficientes puntos son clasificados como *inliers*. Para estimar si pertenecen o no al modelo se puede usar una función de pérdida especifica. [8]

4. Se calcula el error del modelo con los puntos estimados como *inliers*. El error puede calcularse como el error cuadrático medio.

5. Se repiten los pasos anteriores hasta encontrar un modelo con un error menor que el deseado.

Este no solo devuelve los parámetros del plano sino que además nos permite conocer los puntos a los que mejor se ajusta. Por lo que se vuelve muy útil a la hora de diferenciar por ejemplo las paredes y el suelo de los objetos que se encuentran en una escena.

2.5.5 Análisis de las componentes principales.

Este método estadístico consiste en una transformación lineal que devuelve un nuevo sistema de coordenadas para un conjunto de puntos en el cual el eje de mayor longitud nos informa de la dirección en la que el conjunto tiene mayor varianza. De esta forma si tenemos una nube de puntos que representa un objeto podemos asignar a este un sistema de coordenadas que nos indique su orientación en el espacio según sus ejes de mayor varianza.

Dado un conjunto de puntos en el espacio, la varianza de una de las variables indica cómo de alejados están los puntos de la media 2.23. Y la covarianza entre dos variables, por ejemplo x e y, nos da una

[8] Una función simple para ilustrar el funcionamiento del algoritmo es el modulo de la distancia de cada punto al plano.

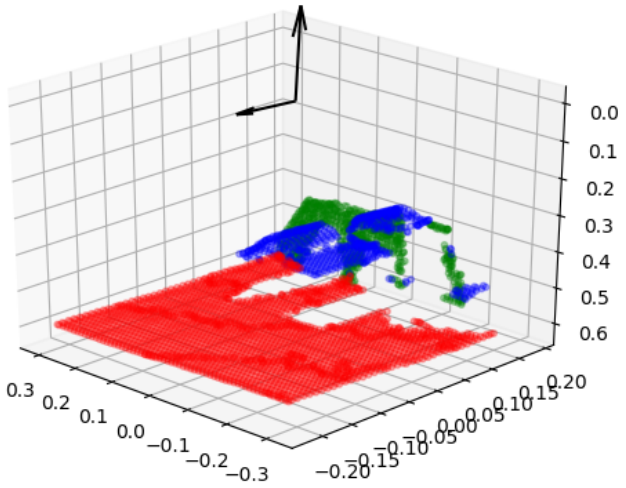


Figura 2.36: Detección automática de los dos planos principales de la imagen mediante RANSAC. En negro los vectores ortogonales que indican la orientación de cada plano.

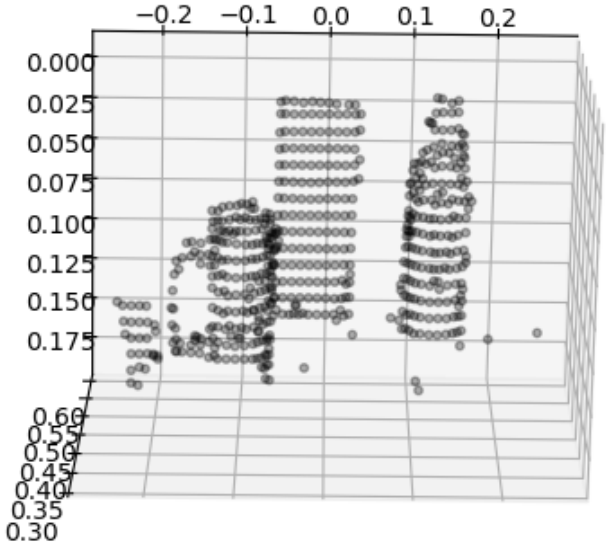


Figura 2.37: Ejemplo de aplicación de RANSAC para separar los objetos de las paredes de una escena.

idea de cómo crecen o decrecen las dos variables juntas 2.24. Si las variables están correlacionadas, la covarianza será distinta de cero y si son variables sin correlación, la covarianza será cero.

$$\sigma^2(x) = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})^2 \tag{2.23}$$

$$\sigma(x,y) = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y}) \tag{2.24}$$

En la matriz de covarianza se reflejan las varianzas de las tres variables en la diagonal principal y las covarianzas en el resto de elementos. Esta matriz es simetrica, ya que de la ecuacion 2.24 podemos deducir que $\sigma(x,y) = \sigma(y,x)$.

$$C = \begin{pmatrix} \sigma^2(x) & \sigma(x,y) & \sigma(x,z) \\ \sigma(y,x) & \sigma^2(y) & \sigma(y,z) \\ \sigma(z,x) & \sigma(z,y) & \sigma^2(z) \end{pmatrix} \tag{2.25}$$

Como se trata de una matriz simétrica, existe una base completa de autovectores. Estos vectores, escalados por sus autovalores asociados, indican las direcciones de mayor covarianza de los puntos. Una buena forma de escalar los ejes es multiplicarlos por la raíz cuadrada de los autovalores.

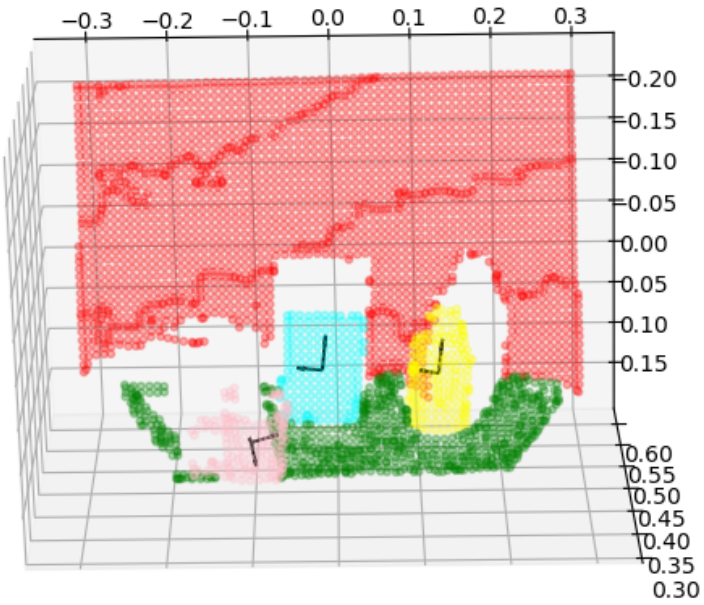


Figura 2.38: Escena segmentada. Primero se ha aplicado RANSAC para eliminar los planos, después se ha ejecutado k-medias a los puntos remanentes para dividirlos en tres objetos. Por último se ha aplicado ACP para encontrar los sistemas de coordenadas de cada objeto.

En la figura 2.38 se puede observar que este método es útil para encontrar la orientación de objetos simétricos y esbeltos (cyan y amarillo). Con los objetos más achatados (rosa) los ejes calculados no darán información válida sobre la orientación del objeto. Sin embargo, existen métodos para poder ajustar modelos de cilindros y otras formas geométricas con los que si se encuentran bien los ejes principales.

2.5.6 Extracción automática de características.

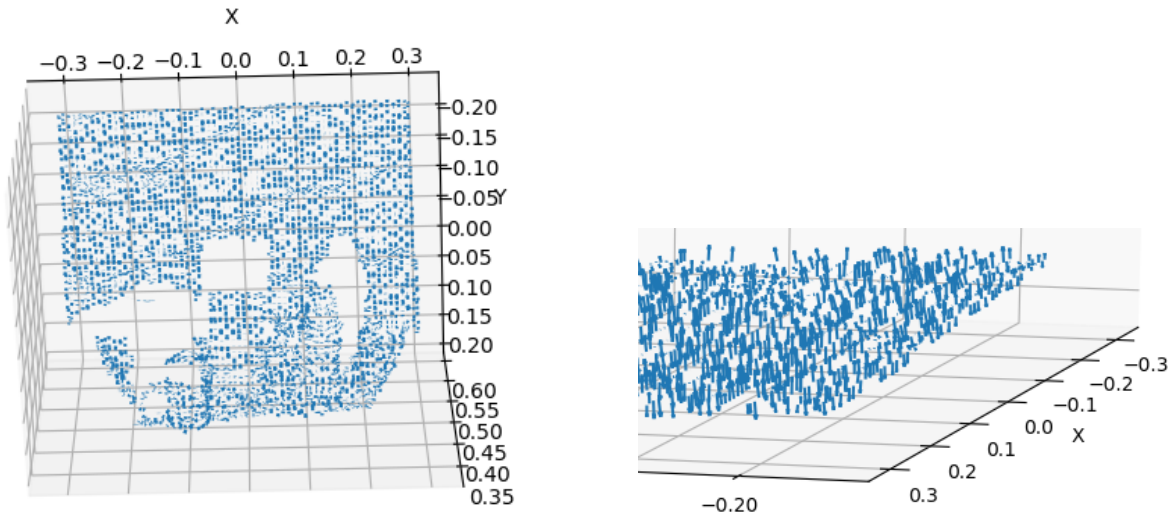
En el mundo real, los objetos que se busca identificar en una escena no siempre pueden ser definidos directamente con parámetros, por lo que se vuelve necesario hacer una descripción matemática de estos mediante la extracción de características geométricas, de color o características semánticas. Estas últimas se obtienen automáticamente mediante redes neuronales profundas, y son estas redes las que se encargan de procesarlas también para hacer la clasificación de los objetos en la escena, por lo que en este apartado se profundizará unicamente en las características que se pueden obtener mediante algoritmos controlados, como son las características SURF o SIFT en imágenes de dos dimensiones, o las características PFH o FPFH en nubes de puntos. La extracción de características no es solo útil en el reconocimiento de objetos si no que se usa también en visión estéreo como se vio en la sección 2.2.2.

2.5.6.1 Transformación de características invariante a la escala.

La mayoría de algoritmos de extracción de características se basan en la búsqueda de esquinas. Un buen algoritmo para esto destaca por poder encontrar estos puntos aunque las esquinas estén rotadas o escaladas. Un punto de esquina se puede definir como el punto de cruce de dos rectas con direcciones distintas, y esto se sigue cumpliendo aunque la imagen esté rotada. El problema viene cuando la imagen se escala en tamaño, ya que un punto esquina puede dejar de serlo al agrandarlo o al disminuirlo. El algoritmo Scale Invariant Feature Transform (SIFT) se utiliza para extraer estas características en imágenes bidimensionales de forma que la detección se mantiene invariante al escalado.

2.5.6.2 Cálculo de normales a las superficies.

El cálculo de los vectores normales a las superficies en una nube de puntos es una aproximación inicial de las características de los alrededores de un punto. Este cálculo se reduce a encontrar los planos que pasan por los puntos cercanos de cada punto. Estos planos pueden obtenerse por los métodos anteriormente estudiados como la regresión lineal o la ecuación del plano, teniendo en cuenta que la regresión dará resultados mejores al utilizar un mayor numero de puntos. Para este cálculo es importante preprocesar la nube de puntos submuestreandola y eliminar así ruido y valores atípicos, también se vuelve necesario calcular primero el árbol kd para poder hacer búsquedas eficientes de los puntos cercanos entre ellos.



(a) (b)

Figura 2.39: Operación de cálculo de normales a la superficie. (a) Normales de la escena. (b) Se observa la orientación aparentemente aleatoria de los vectores normales.

La orientación de los vectores normales obtenidos con este método es impredecible y en una misma superficie habrá vectores apuntando en sentidos contrarios, para algunos algoritmos de más alto nivel que hacen uso de los vectores normales se necesita orientar los vectores pertenecientes a una superficie en el mismo sentido. Esto se hace usando una dirección de referencia o mediante consenso.

2.5.6.3 Histogramas de características de puntos.

Calcular las normales de superficie y las estimaciones de curvatura se vuelve algo básico para representar la geometría alrededor de un punto. Estos cálculos como se ha visto, son extremadametente rápidos y fáciles de implementar pero no capturan demasiados detalles, ya que se intenta describir la geometría del vecindario k de un punto solo con unos pocos valores. Como consecuencia de esto, la mayoría de las

escenas contendrán muchos puntos con valores de características iguales o muy similares, reduciendo así la información extraída.

El objetivo de los histogramas de características (PFH) es codificar las propiedades geométricas del vecindario k de un punto generalizando la curvatura medida alrededor del punto utilizando un histograma de valores multidimensional. Este hiperespacio altamente dimensional proporciona una firma informativa sobre cada característica representada, es invariante a la posición en seis dimensiones de la superficie subyacente y se adapta muy bien a las diferentes densidades de muestreo y niveles de ruido presentes. La representación en PFH se basa en las relaciones entre los puntos en el vecindario k y sus normales. En resumen, se intenta capturar lo mejor posible las variaciones de la superficie muestreada teniendo en cuenta todas las interacciones entre las direcciones de las normales estimadas. El hiperespacio resultante depende, por lo tanto, de la calidad de las normales a la superficie estimadas en cada punto.

La figura 2.40 representa un diagrama de la región de influencia del cálculo PFH para un punto de consulta p_q marcado en rojo y colocado en medio de una esfera con radio **r**. Todos sus vecinos **k** son aquellos cuya distancia es menor que **r**. El descriptor final se calcula como un histograma de las relaciones entre todos los pares de puntos en el vecindario; por lo tanto tiene una complejidad de $O(K^2)$.

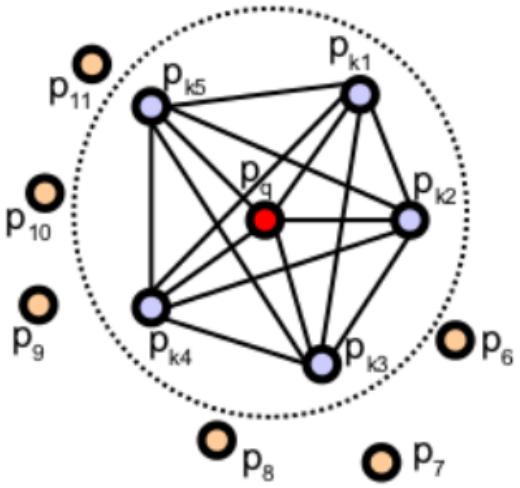


Figura 2.40: Relaciones entre el vecindario del punto p_q

Para calcular la diferencia relativa entre dos puntos p_s y p_t, sean n_s y n_t sus normales asociadas. Definimos un marco de coordenadas fijas en uno de los puntos, donde el eje u corresponderá con su normal correspondiente.

$$u = n_s$$

$$v = u \times \frac{p_s - p_t}{||p_s - p_t||^2}$$

$$w = u \times V$$

Usando el marco **uvw** anterior, la diferencia entre las normales puede expresarse como un conjunto de características angulares.

$$\alpha = V \cdot n_t$$

$$\phi = V \cdot \frac{(p_s - p_t)}{d}$$

$$\theta = arctan(W \cdot n_s, u \cdot n_t)$$

$$d = ||p_s - p_t||^2$$

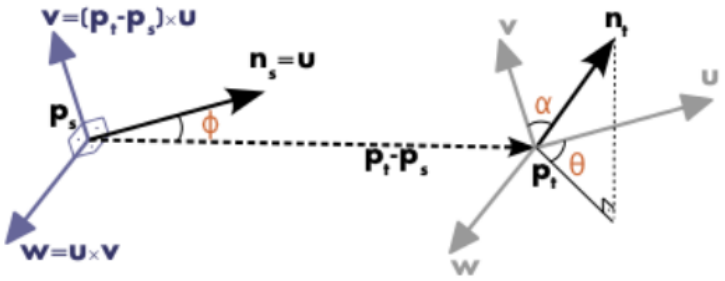


Figura 2.41: Marco de referencia fijo a uno de los puntos.

Donde **d** es la distancia euclidea entre cada par de puntos. La tupla $(\alpha, \phi, \theta, d)$ se calcula para cada par de puntos en la vecindad k, por lo que se reducen los 12 valores (**xyz** y los tres valores de las normales) de los dos puntos a 4 valores.

2.5.6.4 Histogramas rápidos de características de puntos.

La complejidad computacional teórica del PFH visto en la sección anterior para una nube de puntos **P** dada con **n** puntos, es $O(nk^2)$, donde k es el número de vecinos para cada punto **p** perteneciente a **P** y n es el número de puntos de consulta **p**. Para aplicaciones en tiempo real o cercanas a este, el cálculo de las características PFH representan uno de los principales cuellos de botella.

En esta sección se estudia un algoritmo propuesto llamado Histogramas rápidos de características de puntos (FPFH)[9], que reduce la complejidad de PFH a $O(nk)$ manteniendo el poder descriptivo.

- En un primer paso, para cada punto p_s se calculan las tuplas $(\alpha, \phi, \theta, d)$ entre él y sus vecinos como se describía en la sección 2.5.6.3. A esto lo llamaremos Histograma simplificado de características de puntos (SPFH)[10].
- Posteriormente, para cada punto vecino p_k se calcula su SPFH. Estas características son ponderadas por la distancia de cada punto vecino al punto (ω_k) p_s.

$$FPFH(p_s) = SPFH(p_s) + \frac{1}{k} \cdot \sum_{i=1}^{k} \frac{1}{\omega_k} \cdot SPFHp_k \tag{2.26}$$

Este algoritmo se diferencia en el visto en la sección 2.5.6.3 en lo siguiente:

1. En este caso no se interconectan totalmente todos los vecinos de p_s y eso hace que se pierdan algunos pares que podrían contribuir a la descripción de la geometría al rededor de p_s.
2. En PFH se describe el vecindario de cada punto como una esfera perfecta, mientras que en FPFH se incluyen pares de puntos adicionales fuera de la esfera.
3. Finalmente, debido al esquema de re-ponderación, el FPFH combina algunos valores SPFH y re-captura alguno de los pares de valores vecinos del punto.

Como resultado de este proceso tenemos una tupla de 33 valores para cada punto.

[9] En ingles *Fast Point Feature Histograms*
[10] *Simplified Point Feature Histogram*

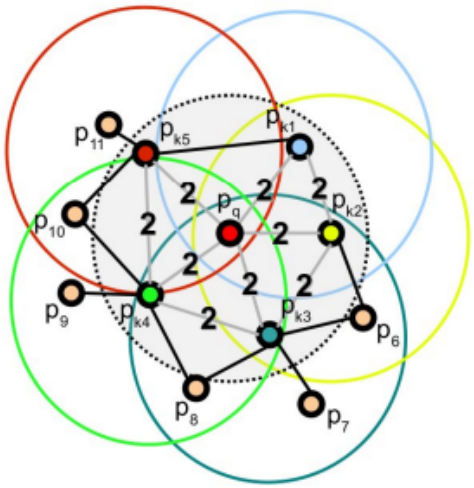


Figura 2.42: Para calcular la tupla de valores se usan puntos fuera de la esfera vecindario.

2.5.7 Registro de nubes de puntos.

En visión por computador el registro de nubes de puntos es el proceso por el cual se busca la transformación espacial que alinea dos nubes de puntos. El propósito es ajustar los datos tomados desde distintas posiciones en un único conjunto de puntos global que corresponderá con un modelo consistente de la escena. Otro de los propósitos de este proceso es encontrar el desplazamiento preciso de un objeto en la escena. Esta aplicación se usará en el desarrollo práctico del proyecto más adelante.

Se han desarrollado diversos algoritmos para este proceso pero en el presente trabajo se verán dos de ellos. El primero usa la extracción de características FPFH 2.42 en ambas nubes de puntos para después hacerlas coincidir basándose en el algoritmo RANSAC de la sección 2.5.4.3. Esto da un resultado inicial aproximado aunque los dos conjuntos de puntos estén muy separados entre ellos en el espacio o estén girados, sin embargo, da resultados distintos para distintas ejecuciones ya que en RANSAC se usan conjuntos de puntos aleatorios para probar cada modelo. Por esto, este algoritmo se usa como entrada al siguiente, que será el encargado de afinar mejor la transformación entre los dos grupos. Para el siguiente paso se usa *Iterative Closest Point*(ICP)

2.5.7.1 Puntos más cercanos iterativos.

ICP[11] Se trata de un algoritmo de optimización empleado para minimizar la diferencia espacial entre dos nubes de puntos. En este método se diferencia la nube de puntos fuente y la nube objetivo o referencia. Esta última se mantiene fija, mientras que la fuente se transforma para coincidir con la referencia. El algoritmo revisa iterativamente la transformación homogénea (traslación y rotación) necesaria para minimizar la métrica del error, generalmente esta métrica es una distancia entre la fuente y el objetivo, por ejemplo, la suma de las diferencias cuadradas entre ambas. Esencialmente, los pasos que se siguen en el algoritmo son:

1. Para cada punto en la nube de puntos de origen, se hace coincidir el más cercano de la nube referencia.

2. Se calcula la combinación de rotación y traslación utilizando una técnica de optimización del error cuadrático medio entre los puntos del origen y del objetivo.

[11] En inglés *Iterative Closest Point*

3. Se transforman los puntos de la fuente usando la matriz obtenida en el paso anterior.

4. Se repiten los pasos anteriores hasta que el error sea menor que el deseado.

Capítulo 3

Desarrollo

3.1 Introducción

En este capítulo se incluirá el desarrollo práctico del trabajo. El proyecto ha consistido en la implementación de una aplicación en lenguaje Python y otra en C++. La aplicación principal se mantendrá a la espera a recibir una orden de trabajo en una lista de tareas, entonces recibirá dos imágenes RGBD, y sus parámetros intrínsecos y extrínsecos, de una camilla médica con el paciente tumbado en ella, las procesará y encontrará la transformación espacial del paciente entre las dos imágenes.

Por otro lado, se ha desarrollado otro programa en C++ encargado de recibir la orden de captura de imágenes y de guardarlas en la base de datos. Este programa espera una orden en una lista de tareas hasta que le llega la orden, entonces se conecta a la cámara mediante su SDK y captura una imagen. Esta imagen se guarda en formato PNG y se codifica en base64 para guardarla en la base de datos.

El capítulo se estructura en cuatro apartados: Colas de tareas y claves, ajuste geométrico del sistema, aplicación para el cálculo del desplazamiento y aplicación de la cámara. En el primero se describirá la forma por la cual se comunican los *workers*. Después se dedicará una sección para el ajuste geométrico del sistema entero, ya que la camilla debe de estar en una posición fija respecto a la cámara. En el tercer apartado se detallará el desarrollo de la aplicación encargada de calcular el desplazamiento de los objetos, así como de las librerías usadas para ejecutar algunos algoritmos de forma eficiente. Y por último, se explica la programación del *worker* encargado de comunicarse con la cámara y obtener las imágenes.

3.2 Colas de tareas y claves.

Como base de datos en memoria y gestor de colas se utiliza Redis. Redis es un motor basado en el almacenamiento de datos clave/valor. En esta aplicación se usarán listas FIFO (First Input First Output) como colas de tareas, y claves como almacenamiento de datos temporal.

La interacción con esta base de datos se hace mediante el uso de comandos. A continuación se hace una lista de los comandos básicos que se usan en el proyecto.

- SET nombreClave "valor": Crea una clave llamada nombreClave y le asigna una cadena de caracteres como valor.
- GET nombreClave : Devuelve el contenido de la clave nombreClave.

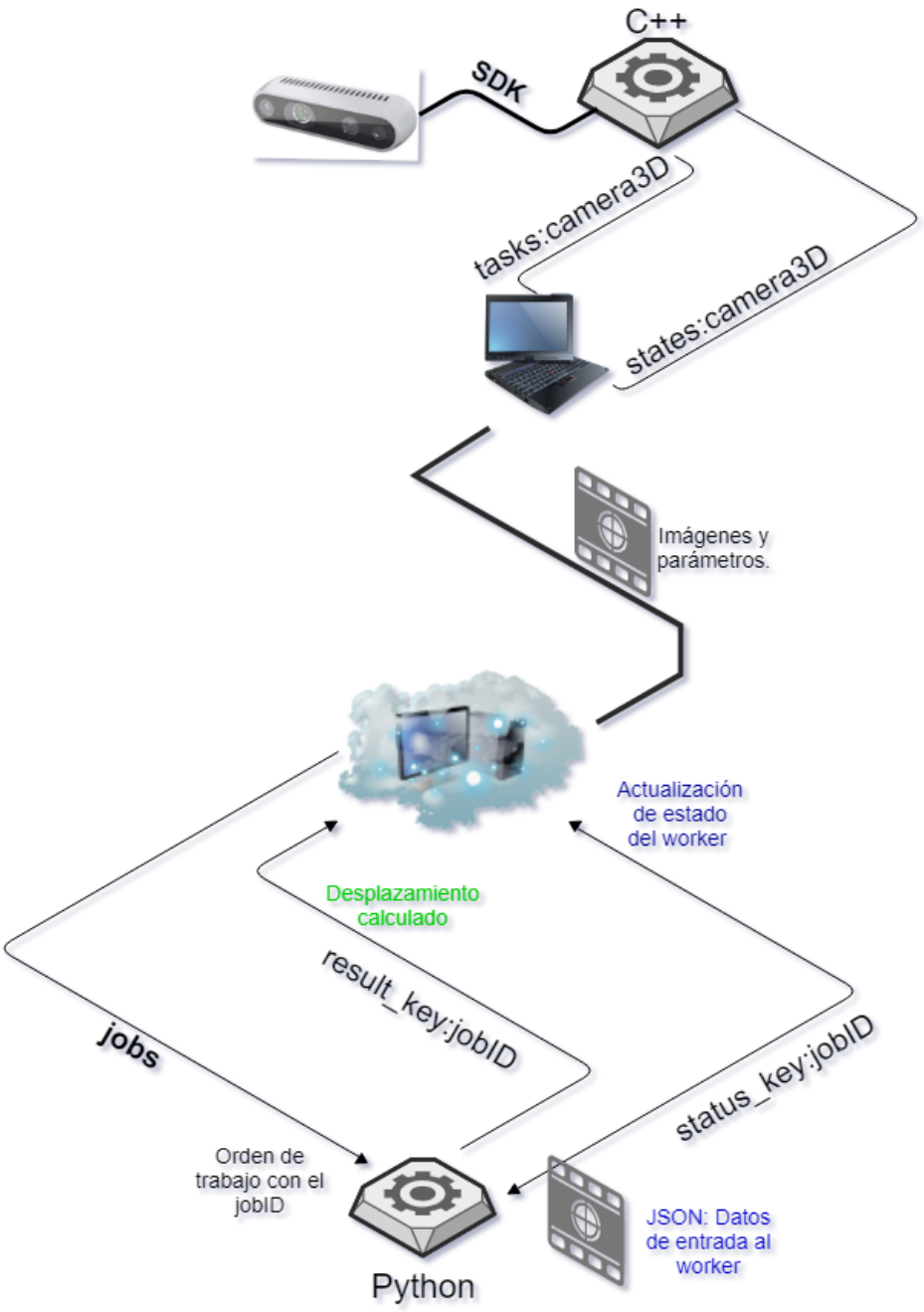


Figura 3.1: Diagrama de claves de Redis utilizadas.

- LPUSH nombreLista "valor": Crea una lista si no está creada y añade el valor en la primera posición por la izquierda, si ya existe, añade un valor en la parte izquierda.
- BLPOP nombreLista: Comando bloqueante que deja el programa en espera hasta que entra un valor en la lista nombreLista y devuelve este valor. Es útil cuando se quiere escuchar en una cola de tareas y bloquear la ejecución del programa hasta que no entre una.

Redis solo está disponible en Linux por lo que si se usa Windows 10 habrá que activar el shell de Ubuntu o usar una maquina virtual. Debemos utilizar los siguientes comandos *sudo apt-get update* para actualizar los repositorios de paquetes de ubuntu. *sudo apt-get upgrade* para actualizar los paquetes que tengamos instalados y por último *sudo apt-get install redis-server* para instalar el servidor de redis.

3.2.0.1 Acceder a Redis desde Python.

Python cuenta con un módulo para conectar con el servidor de redis, este se puede instalar usando el gestor de paquetes pip con el comando *pip install redis*.

En el fragmento de código siguiente se puede ver un ejemplo de conexión a un servidor de Redis local y de introducción de un comando *blpop* para bloquear la ejecución hasta que no entre un trabajo en la lista *jobs*.

```
1 import redis
2
3 r = redis.Redis(host="127.0.0.1", port=6379, db=0)
4 response = r.blpop('jobs', timeout=0)
```

3.2.0.2 Acceder a Redis desde C++.

Para poder acceder a un cliente de Redis desde C++ hay varias librerías, en este proyecto se ha usado *Hiredis*. Para instalarla se introduce el comando *sudo apt-get install libhiredis-dev*.

En el fragmento de código siguiente se puede ver un ejemplo de conexión a un servidor de Redis local y de introducción de un comando. Estos comandos deben estar en formato de cadena de caracteres de C.

```
1 #include <hiredis.h>
2
3 c = redisConnect("127.0.0.1", 6379);
4 reply_tasks = (redisReply *) redisCommand(c,"BLPOP %s 0","tasks:camera3D");
```

3.3 Ajuste geométrico del sistema.

En el momento de instalar el sistema se debe ajustar la posición de la cámara y de la camilla con el fin de poder segmentar correctamente la escena en pasos posteriores. Para esto se ha desarrollado una aplicación en Python que visualiza la nube de puntos en tiempo real y dibuja sobre ella el volumen que se recortará después.

3.4 Aplicación en la nube.

Este programa escrito en Python se considera un *Worker*, pues está constantemente a la espera de recibir una orden de trabajo en la cola *jobs*, una vez le llega dicha orden junto a un *jobID* (que sirve de identificador para cada orden), ejecuta el trabajo, devuelve un resultado en la clave *result_key:jobID* y vuelve a la espera. Mientras se ejecuta el trabajo, se van actualizando los estados por los que va pasando en la clave *status_key:jobID*, de esta forma, la aplicación que controla los *workers* tiene una forma de conocer si el proceso a fallado o ha terminado con éxito. En la figura 3.3 se define el diagrama de flujo de este programa.

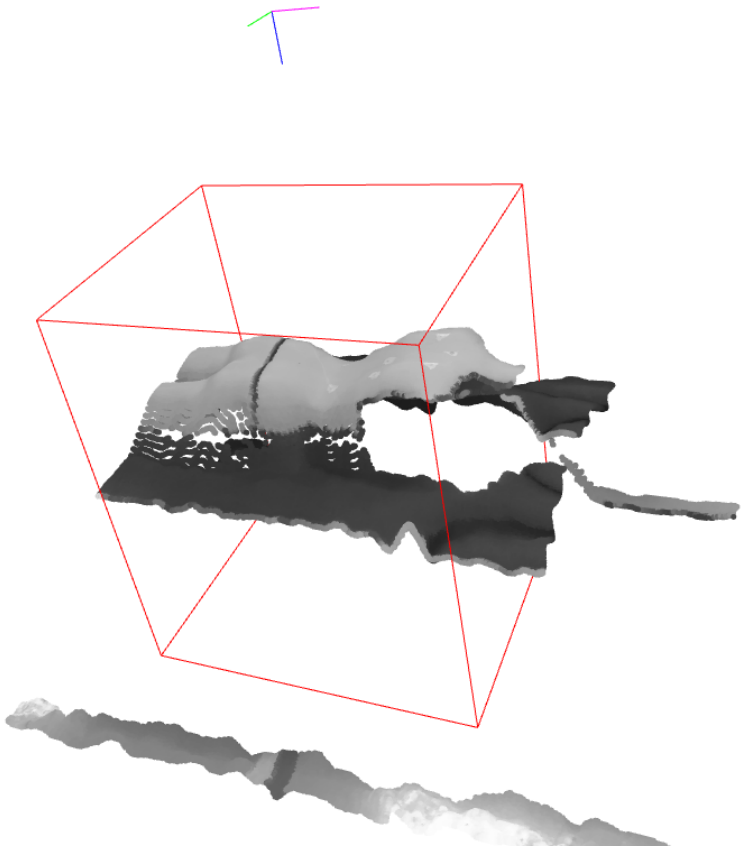


Figura 3.2: Visor en tiempo real de la nube de puntos para ajustar la posición de la cámara y de la camilla.

3.4.1 Cálculo de los desplazamientos.

Una vez vista la estructura y el flujo de ejecución del *worker*, se pasa a explicar los distintos métodos de visión por computador usados para calcular el desplazamiento del paciente. Es necesario remarcar que este desplazamiento no solo corresponde a un cambio en la posición, sino que también podemos encontrarnos un cambio en la orientación del paciente en la camilla. Por lo que no bastará con segmentar el cuerpo del paciente y calcular el desplazamiento de su centroide.

Podemos clasificar los métodos utilizados en tres: Los dos primeros métodos utilizados se basan en asignar unos ejes de coordenadas al cuerpo del paciente una vez segmentado y calcular la diferencia de orientación entre la imagen tomada en la creación del tratamiento y la imagen tomada en la ejecución del tratamiento. El tercer método consiste en aplicar los algoritmos RANSAC generalizado e ICP, que devuelven una matriz de transformación homogénea con el desplazamiento y orientación entre los dos volúmenes.

3.4.1.1 Método 1. Uso de la librería OpenCV en imágenes 2D.

En este método se usa la imagen a color y solo se usará la información de profundidad como ayuda a la hora de segmentar la imagen. Calculando la ecuación del plano al que pertenece la camilla podemos quedarnos con los píxeles por encima de esta, que corresponderán al cuerpo del paciente.

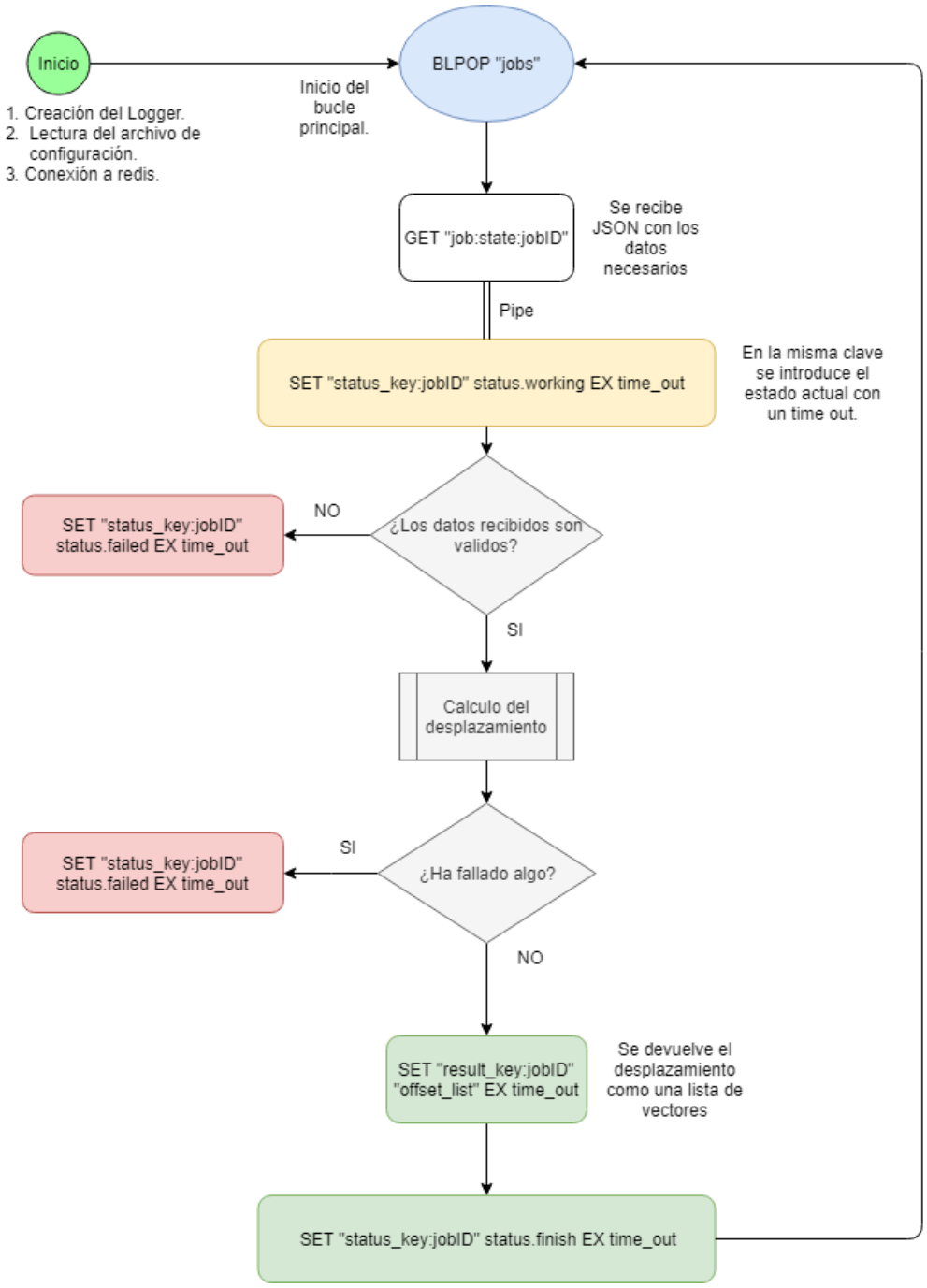


Figura 3.3: Diagrama de flujo del worker en Python.

Una vez la imagen está segmentada, se usará un método de las *opencv* para ajustar una elipse al contorno del cuerpo, este método devuelve tanto los ejes de la elipse como su ángulo. En la figura 3.4 se observa este proceso. El ángulo de inclinación del eje mayor de la elipse es el que nos interesa para conocer la orientación del paciente en ambas imágenes. El flujo de trabajo para este método se define en la figura 3.5.

3.4.1.2 Método 2. Análisis de componentes principales en nubes de puntos 3D.

Primero se realiza la proyección de las imágenes RGBD con el fin de obtener una nube de puntos. Para esto se usa el canal de profundidad y los parámetros intrínsecos recibidos desde *Redis*. Una vez se tienen las

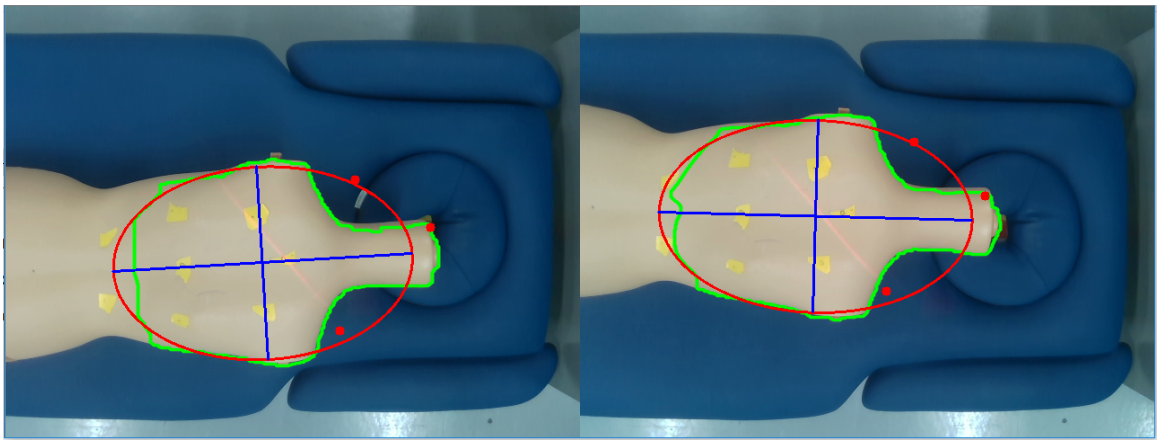


Figura 3.4: En verde el contorno segmentado. En rojo la elipse ajustada al contorno. En azul los ejes principales de la elipse.

nubes de puntos, se puede eliminar la parte que no nos interesa conociendo las características geométricas de nuestro sistema (posición y dimensiones de la camilla). Es por esto por lo que se vuelve necesario el proceso de ajuste visto en la sección 3.3. Una vez tenemos la nube de puntos correspondiente únicamente al cuerpo del paciente, calculamos su media y su matriz de covarianza, de esta matriz se calculan los autovalores y los auto-vectores, estos autovectores escalados por sus autovalores como se vio en la sección 2.5.5, nos dan los ejes de mayor varianza del cuerpo, que coincidirán con los ejes de simetría. En la figura 3.8 se muestra el diagrama de flujo de este programa.

3.4.1.3 Método 3. RANSAC e ICP en nubes de puntos 3D.

Primero se realiza la proyección de las imágenes RGBD con el fin de obtener una nube de puntos. Para esto se usa el canal de profundidad y los parámetros intrínsecos recibidos desde *Redis*. Una vez se tienen las nubes de puntos, se puede eliminar la parte que no nos interesa conociendo las características geométricas de nuestro sistema (posición y dimensiones de la camilla). Es por esto por lo que se vuelve necesario el proceso de ajuste visto en la sección 3.3. Una vez tenemos la nube de puntos correspondiente únicamente al cuerpo del paciente, aplicamos el método de registro RANSAC para obtener una transformación inicial, esta transformación la usamos como inicio del método ICP entre las dos nubes de puntos para dar con una transformación aún más precisa. En la figura 3.9 se visualizan los resultados de cada método por separado. Y en la figura 3.10 se ve el resultado del método 3 aplicado a tres puntos de ejemplo.

Este método es el que mejor resultado ha dado, devolviendo la transformación precisa entre las dos nubes de puntos. La librería open3D ofrece estos dos métodos de forma que se ejecutan eficientemente y permite escoger el número de iteraciones de los algoritmos y el máximo error permitido en el ajuste. En el anexo se podrá ver el código en Python implementado. En la figura 3.11 se muestra el diagrama de flujo de este programa.

3.4.2 Transformación entre los sistemas de coordenadas.

En el flujo de trabajo de los datos tridimensionales obtenidos con la cámara, es necesario hacer una serie de transformaciones geométricas. Esto es debido a que los puntos del tratamiento que devuelve el robot en el proceso de *teaching*, están definidos en el sistema de referencia del robot mientras, que los puntos obtenidos por la cámara toman como base los ejes de la cámara. En la figura 3.12 se muestran los tres sistemas de coordenadas definidos: El de la base del robot, el del efector del robot y el de la cámara.

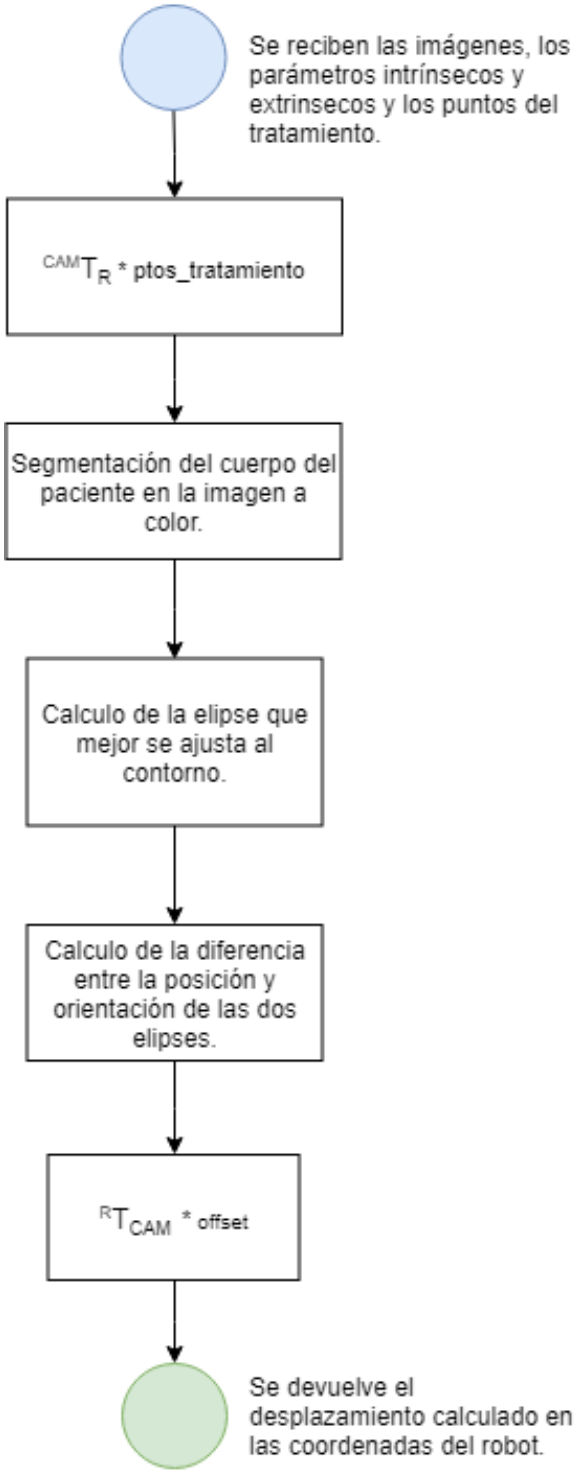
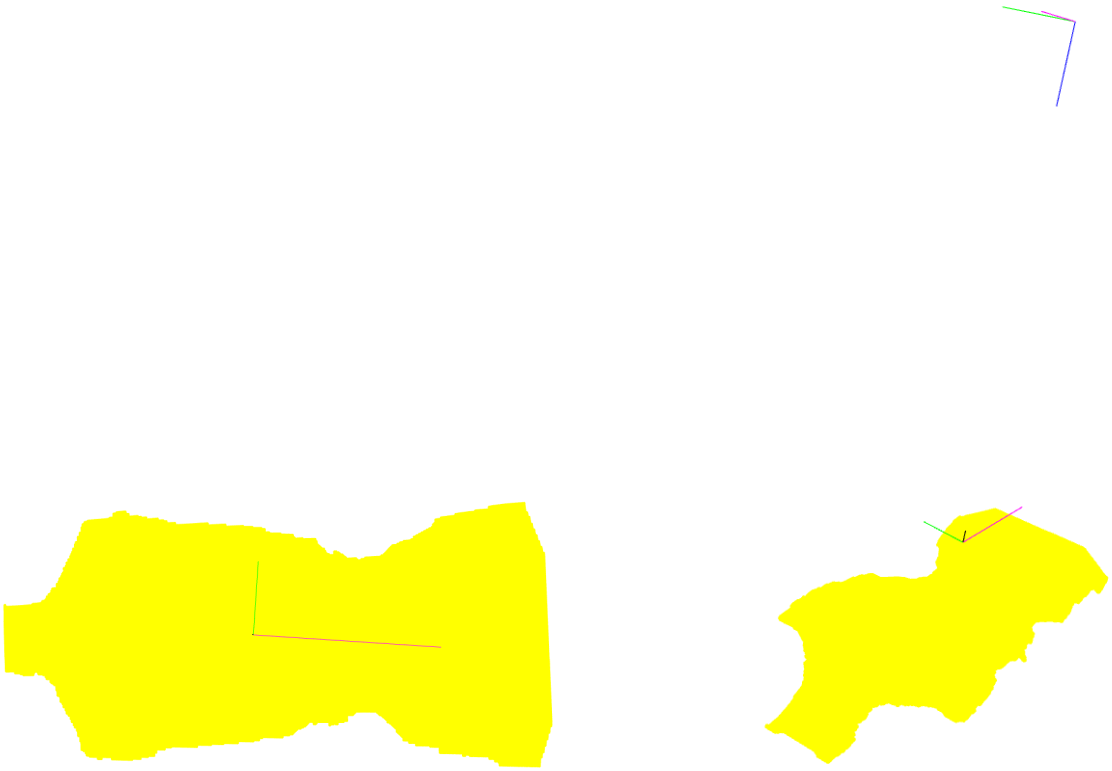


Figura 3.5: Diagrama de flujo del método 1.

En el proceso de entrenamiento se guardan los puntos escogidos mediante el modo *free-drive*[1]. Es importante conocer bien el formato en el que se guardan, ya que existen diversas formas de representar una posición y orientación en el espacio. Por defecto, el robot UR5 utiliza coordenadas XYZ del efector respecto a la base para representar el desplazamiento y un par de rotación para representar la orientación. Se vuelve necesario transformar la nube de puntos obtenida con la cámara al sistema de coordenadas del

[1]Modo en el que el usuario puede mover el efector del robot libremente y guardar los puntos deseados.



(a) (b)

Figura 3.6: Ejes asignados al primer volumen.

robot o bien transformar los puntos del tratamiento al sistema de la cámara, para esta transformación se usarán matrices de transformación homogénea, en este apartado se verá cómo obtener las matrices de transformación a partir de los seis valores obtenidos del robot:

$$pose = [x, y, z, rx, ry, rz] \tag{3.1}$$

Se define un par de rotación como un vector en el cual su modulo corresponde al ángulo de giro y sus vectores directores definen su eje de giro.

$$\begin{aligned} par &= [rx, ry, rz] \\ \theta &= \sqrt{rx^2 + ry^2 + rz^2} \\ \hat{i} &= \tfrac{rx}{\theta} \\ \hat{j} &= \tfrac{ry}{\theta} \\ \hat{k} &= \tfrac{rz}{\theta} \end{aligned} \tag{3.2}$$

Para trabajar más cómodamente se ha transformado el par de rotación en una matriz de rotación, para esto se usa la función *angvec2tr()* de la toolbox de robótica de Matlab. Esta función recibe el vector $v = [\hat{i}, \hat{j}, \hat{k}]$ y el angulo de giro θ alrededor de él y obtiene la matriz de rotación R como se ve en las ecuaciones 3.3 y 3.4. Sea sk la matriz antisimétrica formada a partir de los vectores unitarios del par de rotación.

(a) (b)

Figura 3.7: Ejes asignados al segundo volumen.

$$sk = \begin{pmatrix} 0 & -\hat{k} & \hat{j} \\ \hat{k} & 0 & -\hat{i} \\ -\hat{j} & \hat{i} & 0 \end{pmatrix} \tag{3.3}$$

$$R = I + sin(\theta) \cdot sk + (1 - cos(\theta)) \cdot sk^2 \tag{3.4}$$

Con esto ya podemos obtener la matriz de transformación homogénea entre el efector y la base del robot. Por ejemplo, si el efector está en la $pose = [-0{,}533, 0{,}024, 0{,}745, -2{,}216, -2{,}191, 0{,}163]$ la matriz de transformación sera la siguiente:

$$^{R}T_{EF} = \begin{pmatrix} 0{,}0087 & 0{,}9960 & -0{,}0890 & -0{,}5330 \\ 0{,}9982 & -0{,}0139 & -0{,}0584 & 0{,}0240 \\ -0{,}0594 & -0{,}0883 & -0{,}9943 & 0{,}7450 \\ 0 & 0 & 0 & 1{,}0000 \end{pmatrix} \tag{3.5}$$

La cámara irá colocada en la herramienta del robot (TCP) por lo que su posición respecto a la base del robot será la misma que la del efector, con un offset añadido debido a la geometría del TCP. La matriz de transformación considerando este offset se observa en la ecuación 3.6.

$$^{R}T_{CAM} = ^{R}T_{EF} \cdot transl([0, -0{,}05, 0]) \cdot trotz(\pi) \tag{3.6}$$

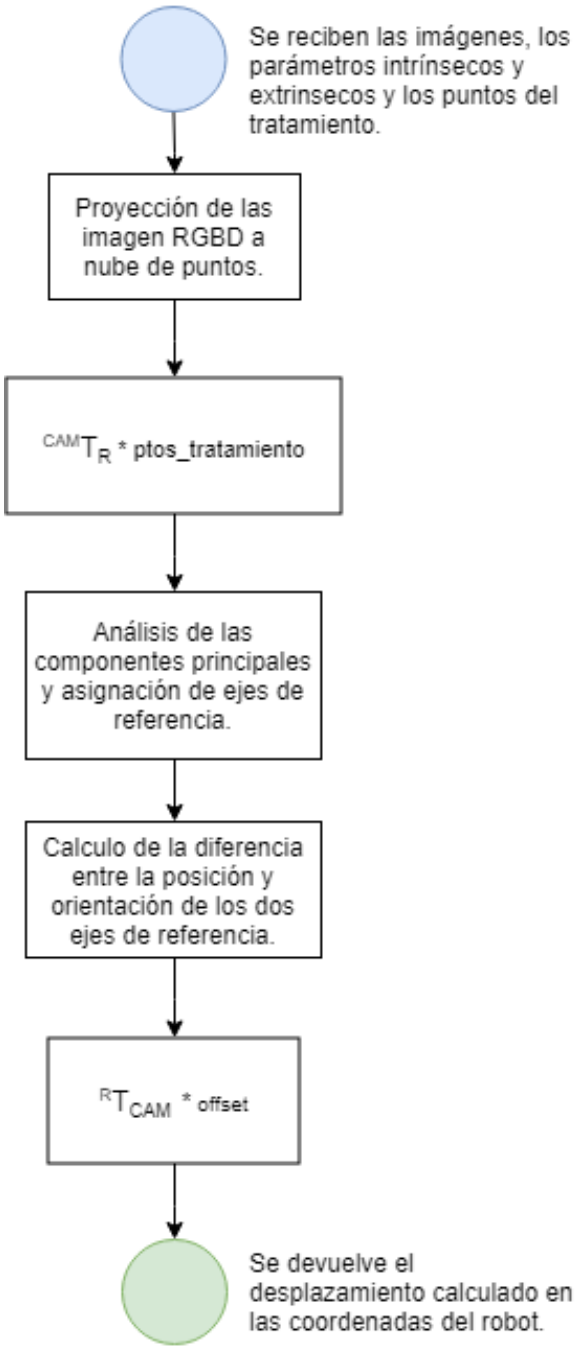
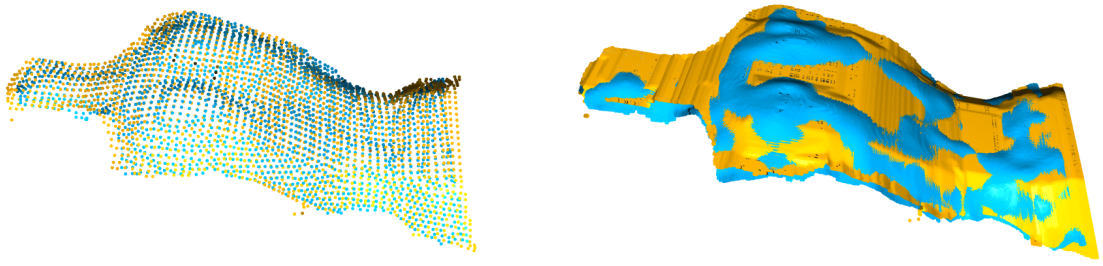


Figura 3.8: Diagrama de flujo del método 2.

Esta matriz se utilizará para transformar el offset calculado en el sistema de la cámara al sistema de coordenadas del robot. Su matriz inversa servirá para transformar los puntos recibidos del robot al sistema de coordenadas de la cámara.

3.5 Aplicación de la cámara RGBD

Este programa es más simple que el anterior pero debe de ser fiable y atender a cualquier error que pueda darse. Estos errores son:



(a) (b)

Figura 3.9: (a) Ajuste por el método RANSAC. (b) Ajuste más fino con el método ICP.

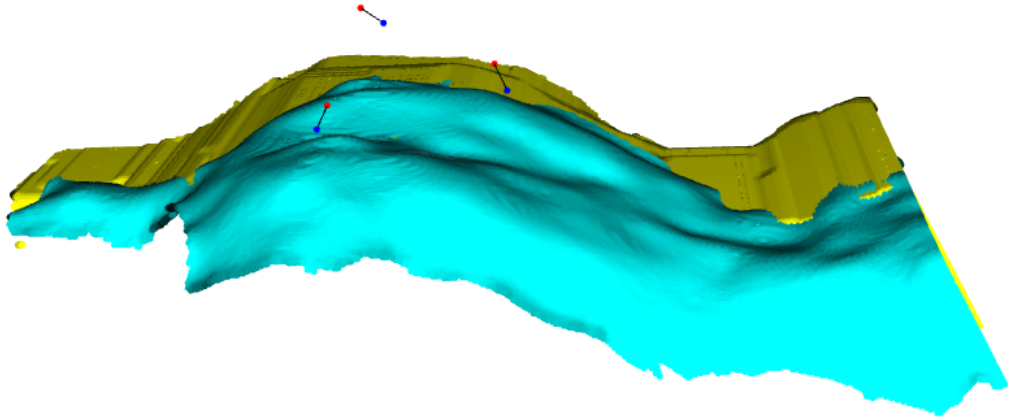


Figura 3.10: Resultado del ajuste completo con el desplazamiento aplicado a los puntos del tratamiento.

- Desconexión repentina de la cámara: La correcta conexión de la camara se comprueba cada vez que llega un trabajo, si la cámara se desconectara durante la ejecución del bucle principal, enviar el estado de error para que el operario pueda comprobar su conexión manualmente y volver a escuchar en la lista de tareas.
- Fallo al adquirir la imagen de color: Si esto ocurre deberá devolver un fallo especifico en la lista de estados para poder depurar correctamente después.
- Fallo al adquirir la imagen de profundidad: Deberá dar otro fallo indicando que ha sido el frame de profundidad el que no se ha podido adquirir.

En la figura 3.13 se define el diagrama de flujo de este programa.

Figura 3.11: Diagrama de flujo del método 3.

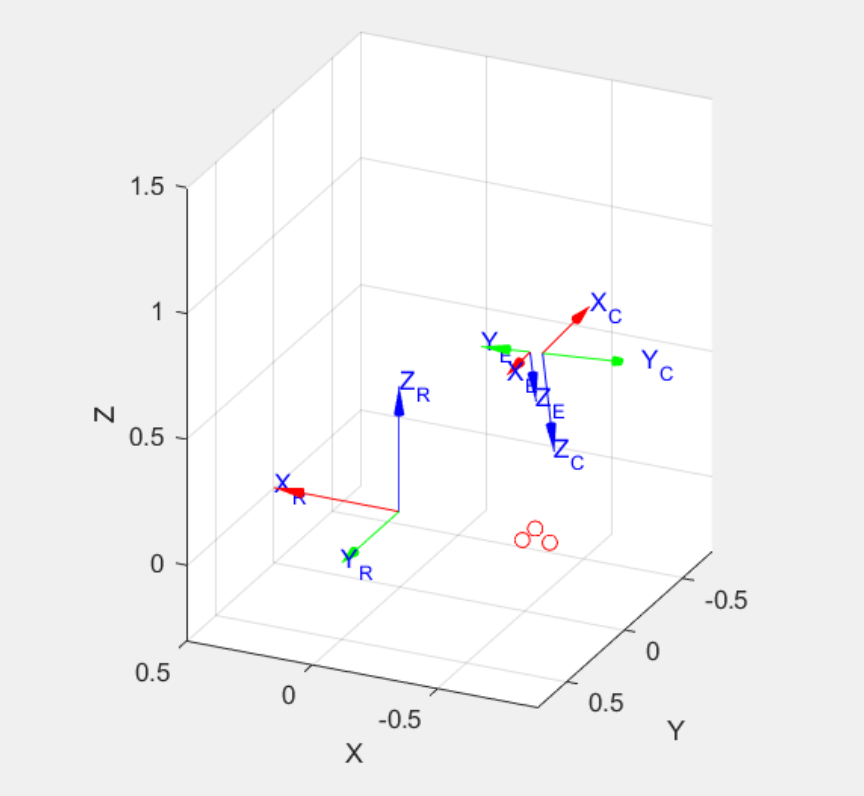


Figura 3.12: Representación de los sistemas de coordenadas a tener en cuenta. De la base del robot(R), del efector del robot(E) y de la cámara(C) solidaria a este. Los tres puntos en rojo representan los puntos escogidos para el tratamiento.

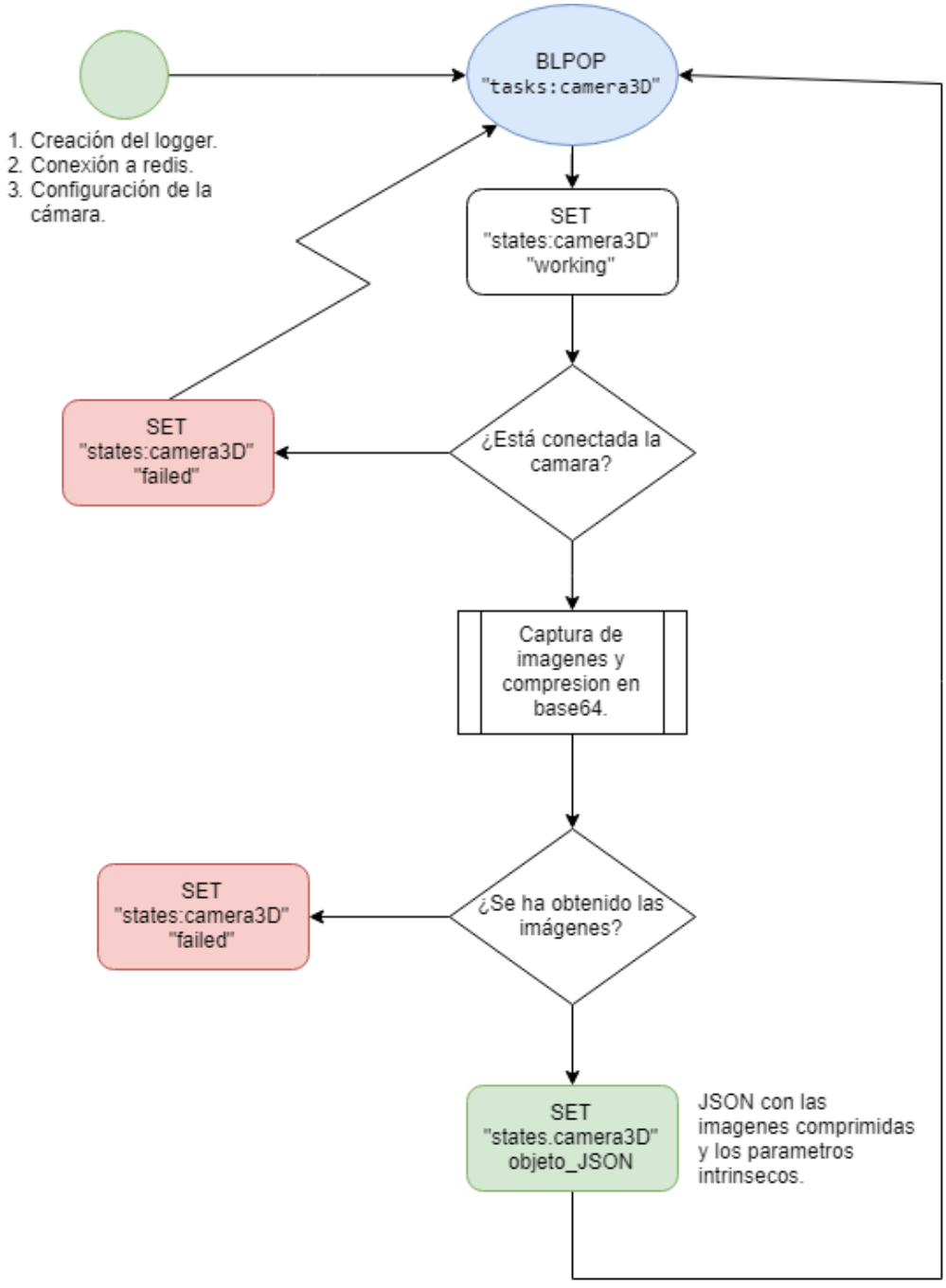


Figura 3.13: Diagrama de flujo del *worker* que controla la cámara de profundidad.

Capítulo 4

Resultados

4.1 Introducción

En este capítulo se introducirán los resultados más relevantes del trabajo. En términos generales, se puede concluir que se ha encontrado una solución al problema propuesto por el proyecto y que se ha implementado de forma que se cumple con los objetivos.

En este capítulo se explicará el entorno experimental, las métricas de calidad usadas para evaluar los resultados, las estrategias seguidas para obtener los resultados y por último, los resultados obtenidos.

4.2 Entorno experimental

El entorno usado para obtener los resultados de este trabajo no es el entorno real con el robot, si no una simulación basada en Python junto con los módulos:

- *Numpy* como núcleo para trabajar con vectores, matrices, y distintas operaciones matemáticas.
- *Matplotlib* para la visualización.
- *Open3D* para realizar las proyecciones de las nubes de puntos, trabajar con ellas y visualizarlas de forma intuitiva.
- *Pyrealsense2* como SDK para comunicarse con la cámara RGBD en la simulación.
- *Redis* como cliente redis desde Python.

Se simulará la llegada de datos en *Redis* en el mismo formato que lo recibe en la aplicación real, con la salvedad que las imágenes se tomarán desde vídeos previamente grabados en el formato de la cámara (.bag). Y se devolverán los resultados a *Redis* en el formato esperado por la aplicación final.

Se ha creado un modulo en *Python* que envía ordenes de trabajo al *worker* del desplazamiento para poder evaluar el comportamiento de este como si estuviese en un entorno real.

4.2.1 Métricas de calidad

Se considerará que la solución es válida cuando se cumplan los siguientes puntos:

- Cálculo del desplazamiento del paciente con un error de offset fijo máximo de 2 cm respecto a las medidas reales.
- Cálculo del desplazamiento del paciente con un error variable (ante los mismos datos de entrada) máximo de 5 mm.
- El tiempo máximo de ejecución de un trabajo por parte del *worker* en *Python* es de 30 segundos.

4.2.2 Estrategia y metodología de experimentación

Para mostrar los resultados obtenidos en cada método, se ha tomado la primera foto al paciente en una posición y se han escogido tres puntos de tratamiento. Después se ha movido al paciente hacia la derecha, midiendo el desplazamiento con un calibre para poder evaluar los resultados de cada algoritmo.

4.3 Resultados

Los resultados obtenidos mediante el procedimiento descrito en 4.2.2 se muestran en la tabla 4.1. En las figuras 4.1 y 4.2 se observan los puntos definidos en el tratamiento en rojo y los puntos calculados en azul para los métodos dos y tres.

Puntos.	Medido	Metodo1	Metodo2	Metodo3
Offset 1	[0.000, -0.400, 0.000]	[0.014, 0.056, 0.000]	[-0.005, -0.024, 0.004]	[0.000, -0.041, 0.005]
Offset 2	[0.010, -0.400, 0.000]	[-0.036, 0.067, 0.000]	[0.032, -0.021, 0.004]	[0.015, -0.040, 0.006]
Offset 3	[0.000, -0.500, 0.000]	[0.006, 0.031, 0.000]	[0.006, -0.043, 0.004]	[0.004, -0.049, 0.005]

Tabla 4.1: Resultados de desplazamiento en el sistema de coordenadas del robot. [x, y, z]

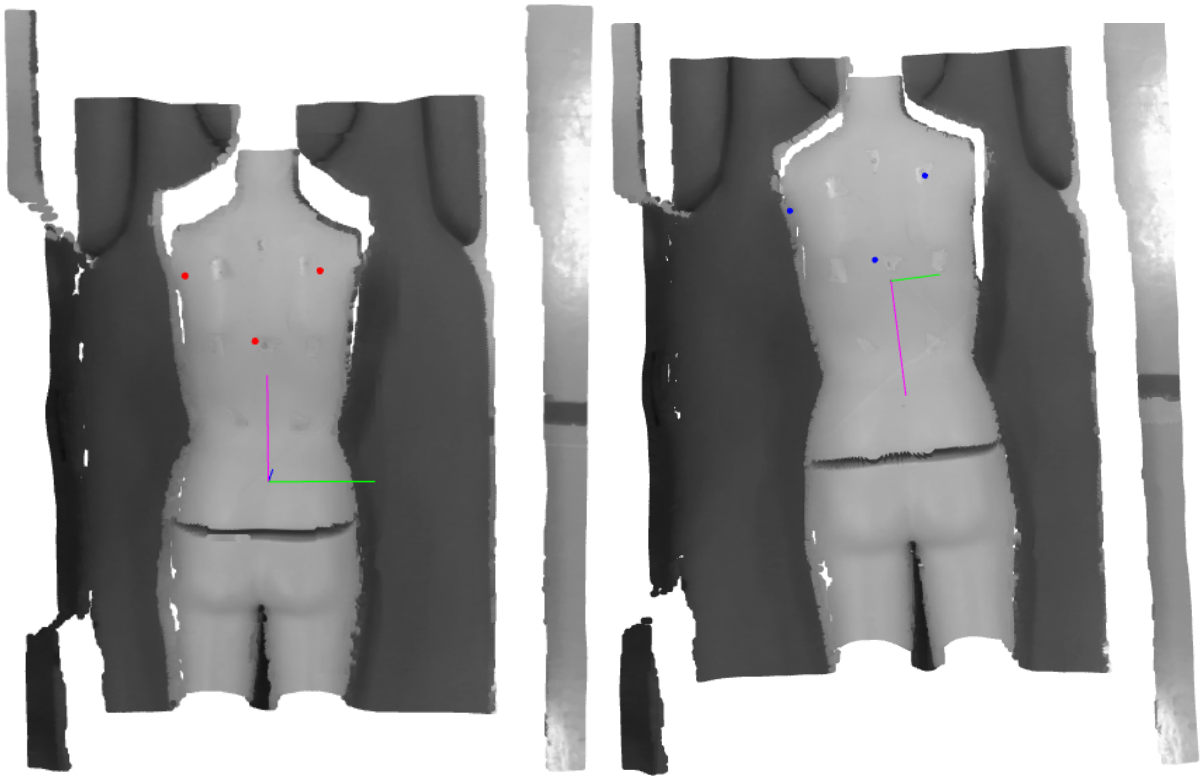
4.3.1 Repetitibilidad de los resultados.

Es importante comentar que, mientras el primer y el segundo método devuelven siempre el mismo resultado ante las mismas imágenes de entrada; el método tres no se comporta igual. Como se vio en el estudio teórico, el algoritmo RANSAC funciona probando unos puntos aleatorios contra el modelo deseado, esto provoca que dos ejecuciones distintas del algoritmo devuelva dos resultados distintos, lo que causa que el desplazamiento calculado bascule alrededor de un valor. Esto puede influir a la hora de evaluar la solución propuesta mediante las métricas de calidad escogidas, ya que puede aparece un error variable.

Para estudiar la variabilidad del resultado por este método, se hace un ensayo definiendo dos puntos de tratamiento y repitiendo el cálculo 50 veces. En las figuras 4.3, 4.4, 4.5 tenemos el desplazamiento obtenido para los dos puntos en metros. Se observa que los resultados no varían de forma significativa para nuestra aplicación (<1mm), ya que uno de los objetivos es que el resultado no varíe más de 5 mm.

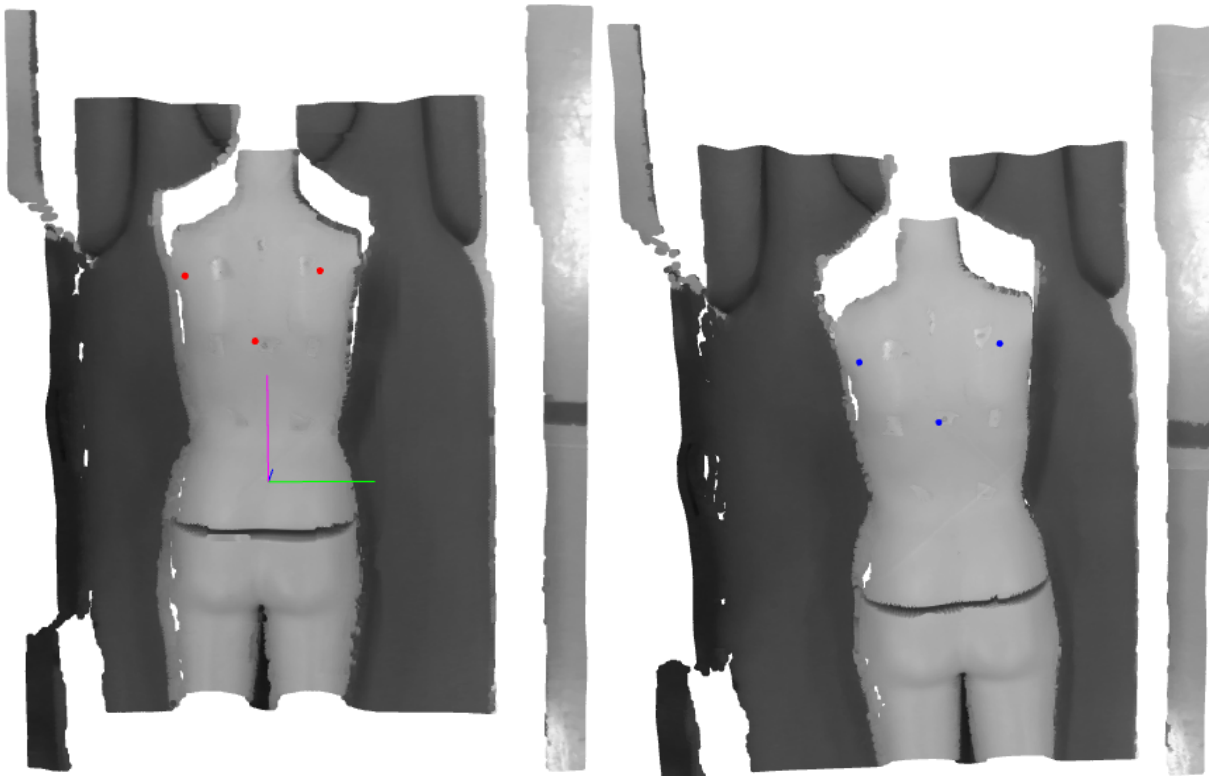
4.3.2 Tiempo de ejecución.

En cuanto al tiempo de ejecución del método tres también cambiará de una ejecución a otra, ya que según los puntos aleatorios que se cojan en el método RANSAC, se tendrán que hacer más o menos iteraciones para conseguir minimizar el error. El resto de métodos no se estudian en este apartado, ya que estos siempre tardan lo mismo y está muy por debajo del tiempo máximo marcado en las métricas de calidad. Como vemos en la figura 4.6, se cumplen con el tiempo máximo de ejecución de 30 segundos. Se consigue



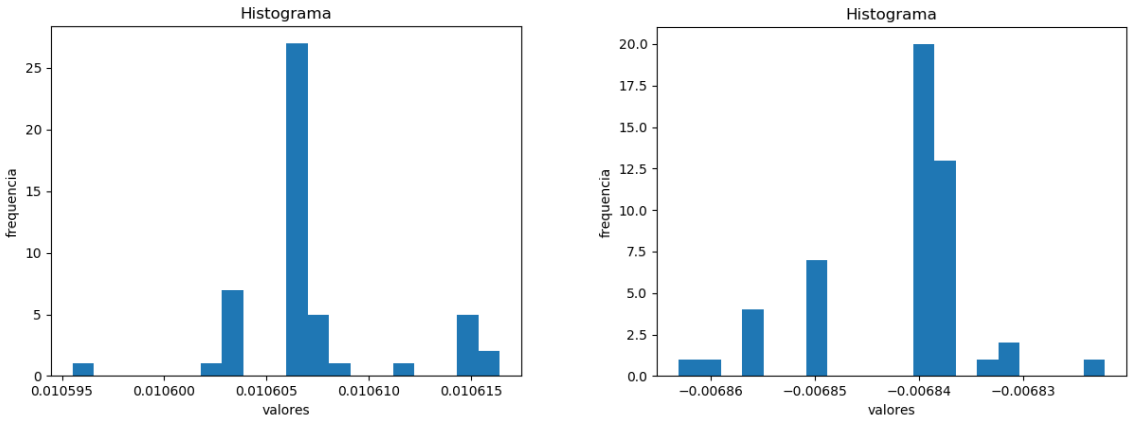
(a) (b)

Figura 4.1: Resultado del método 2 basado en ACP. (a) Puntos en la creación del tratamiento. (b) Puntos en la aplicación del tratamiento.



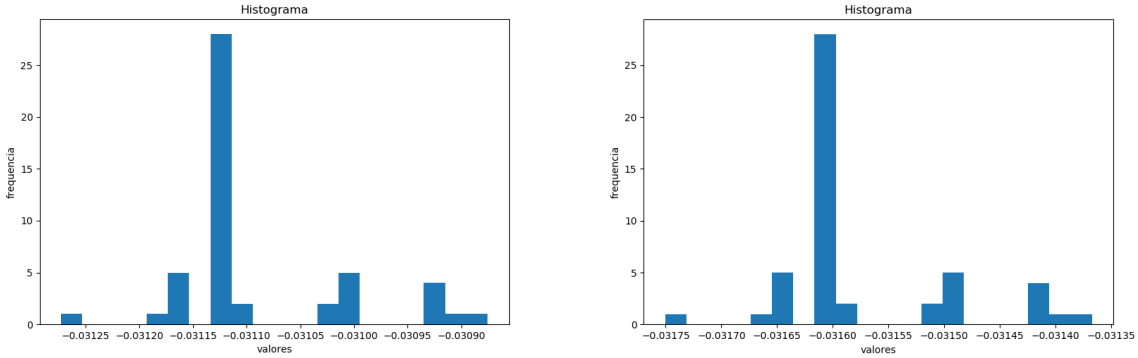
(a) (b)

Figura 4.2: Resultado del método 3 basado en RANSAC e ICP. (a) Puntos en la creación del tratamiento. (b) Puntos en la aplicación del tratamiento.



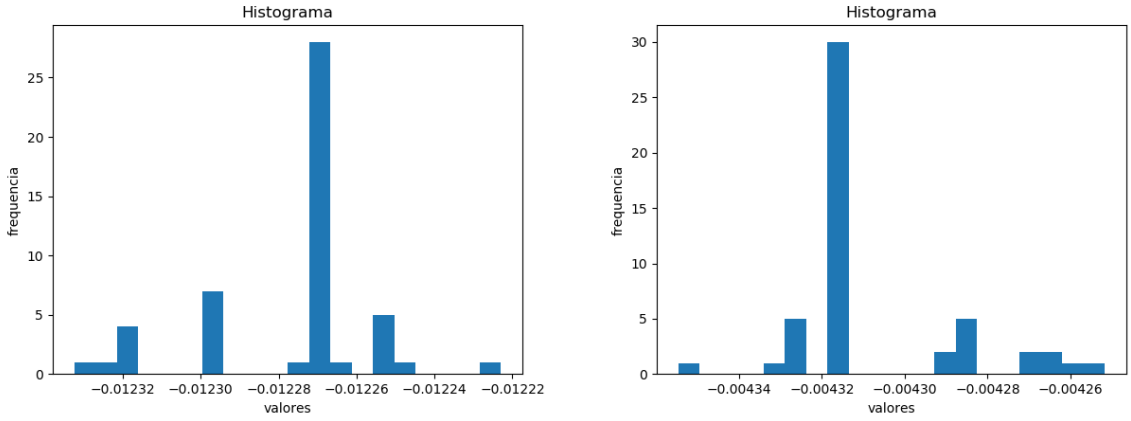
(a) (b)

Figura 4.3: Desplazamiento en el eje X. (a) En el punto 1. (b) En el punto 2.



(a) (b)

Figura 4.4: Desplazamiento en el eje Y. (a) En el punto 1. (b) En el punto 2.



(a) (b)

Figura 4.5: Desplazamiento en el eje Z. (a) En el punto 1. (b) En el punto 2.

una media 1.8 segundos por ejecución en un ordenador 16 Gb de memoria RAM, por lo que, aunque el tiempo de ejecución cambie según el equipo que se use, se podrá tener estos resultados como referencia.

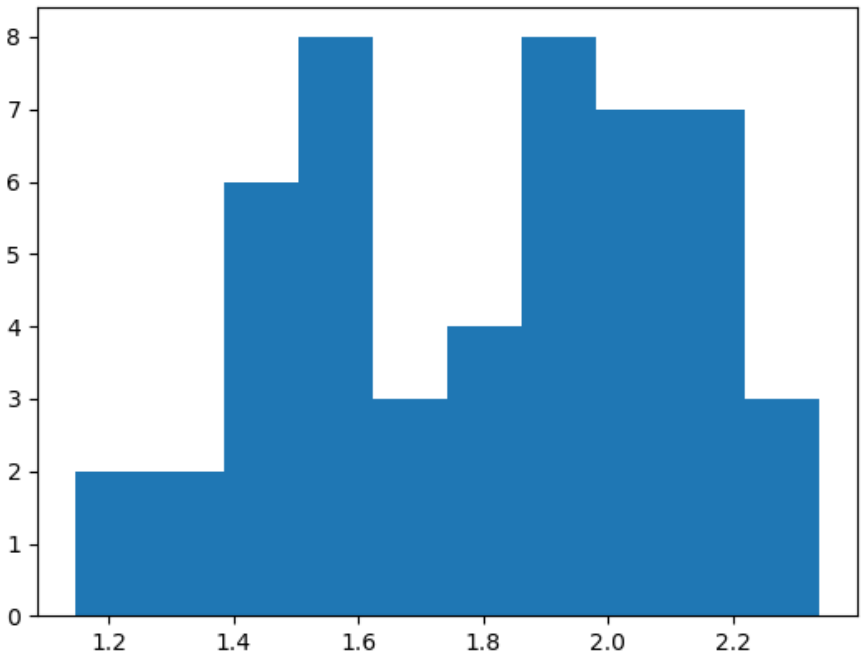


Figura 4.6: Histograma del tiempo de ejecución del método 3 en 50 ejecuciones.

4.4 Conclusiones

De los tres métodos utilizados no todos han dado el mismo resultado, siendo el primer método 3.4.1.1 el menos preciso debido a que en pacientes menos esbeltos o en imágenes mal segmentadas, la elipse ajustada se parece más a una circunferencia y no se obtiene una buena referencia de orientación. Además este método no tiene en cuenta el desplazamiento en el eje Z (arriba-abajo) de la camilla, por lo que en la aplicación final se ha omitido.

Los métodos que trabajan con las nubes de puntos son los que mejor resultados han dado, teniendo en cuenta que el análisis de componentes principales solo nos da una buena medida de la orientación siempre que el cuerpo del paciente sea más alto que ancho. Este método se ha mantenido como una forma de aportar robustez al resultado final. Sin embargo, el método principal y el que da el desplazamiento más preciso es el basado en RANSAC y el algoritmo Iterative Closest Point.

El resultado final del desplazamiento consiste en una media ponderada entre el método dos y el tres, siendo este último el que tiene más peso. Además si el método tres falla porque no consiga alinear las nubes de puntos con un error menor que el indicado, el cálculo del desplazamiento final se deshecha y se devuelve el estado de error para que se repita el proceso completo.

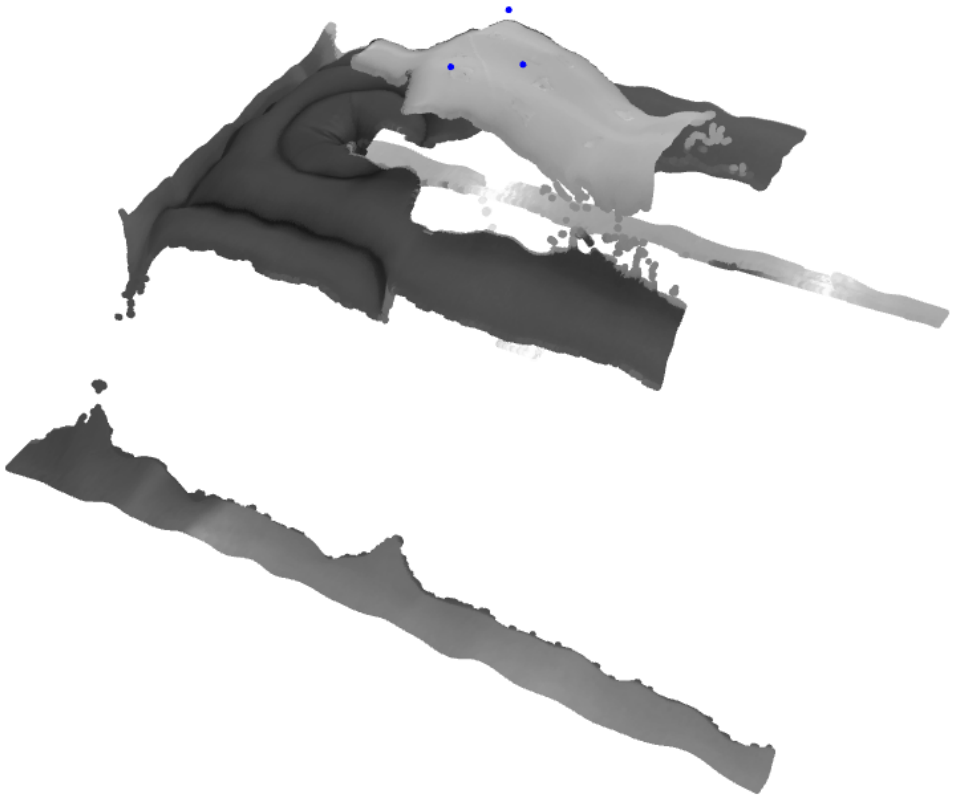


Figura 4.7: Simulación de los puntos del tratamiento (en azul) con el desplazamiento ya aplicado.

Capítulo 5

Conclusiones y líneas futuras

El resultado en el cálculo del desplazamiento de los pacientes ha sido satisfactorio. Se puede concluir que es posible detectar la posición y orientación exacta del cuerpo humano mediante visión estéreo. Los mejores resultados se han obtenido aplicando métodos registro en tres dimensiones de nubes de puntos. Estos métodos son más costosos computacionalmente pero, en aplicaciones que no sean en tiempo real como la referente a este trabajo, se encuentra que son adecuados.

Para realizar el procesamiento de las imágenes y el calculo del desplazamiento se ha subido el algoritmo final a la nube por lo que la aplicación es escalable en el futuro. Como posibles mejoras se valora el uso de computación paralela en unidades de procesamiento gráfico (GPGPU), lo que aceleraría en gran medida los algoritmos de procesado de nubes de puntos en tres dimensiones. Otra propuesta para el futuro sería el uso de redes neuronales convolucionales para la segmentación semántica del cuerpo del paciente. Esto último conlleva la recogida de un gran conjunto de datos por lo que no ha sido viable en esta primera fase del proyecto.

Se han aprendido conceptos actuales como los referentes al funcionamiento de las aplicaciones distribuidas en la nube basadas en *workers*, y se ha utilizado un método de comunicación entre estos como Redis. También se ha estudiado C++ y en mayor medida Python, así como sus paquetes para computación científica. Por último, en lo referente al estudio teórico, se ha buscado realizar pruebas de concepto (PoC) de la mayoría de algoritmos con el fin de entenderlos mejor. Estos ejemplos se encuentran en el apéndice A.

Capítulo 6

Presupuesto

6.1 Equipo de trabajo

Para la realización del proyecto ha sido necesaria la compra de un conjunto de herramientas, materiales y la dedicación de una serie de horas.

6.1.1 Coste del material.

Los costes del material, tanto hardware como software, se recogen en la tabla 6.1.

Objeto.	**Precio Unitario(€).**	**Unidades.**	**Precio Total (€).**
Camara Intel RealSense	180	1	180
Robot colaborativo UR5e	27500	1	27500
Licencia de Matlab	0	1	0
Licencia de Office	0	1	0
HMI SIMATIC Comfort Siemens TP1200	2722,5	1	2722,5
Total(IVA Incluido)			30402,5

Tabla 6.1: Tabla de costes de materiales.

6.1.2 Coste de la mano de obra.

En la Tabla 6.2 se muestra el precio correspondiente a la mano de obra empleada para la realización del proyecto. Se han empleado 840 horas (7 meses) para la realización del proyecto. De estas 840 horas, 600 horas (5 meses) han consistido en labores de investigación y desarrollo software, mientras que el resto del tiempo empleado se ha dedicado a la realización de la documentación necesaria.

Actividad.	**Precio(€).**	**Tiempo(meses).**	**Precio Total (€).**
Ingeniería	1.600	5	8.000
Documentación	1.300	2	2.600
Total(IVA Incluido)			10.600

Tabla 6.2: Tabla de costes de materiales.

6.1.3 Coste de ejecución del material.

Coste.	Total(€).
Coste Materiales	30.402,5
Mano de obra	10.600
Total(IVA Incluido)	41.002.5

Tabla 6.3: Coste total de ejecución material.

6.1.4 Gastos generales y beneficio industrial.

En esta sección se incluyen los gastos derivados del uso de las instalaciones más el beneficio industrial. Se estima que es de un 20 % del coste de ejecución del material.

Gastos generales y beneficio industrial.	**8.200,5 €.**

Tabla 6.4: Gastos generales y beneficio industrial.

6.1.5 Presupuesto de ejecución por contrata:

Se calcula sumando el presupuesto de ejecución material más los gastos generales y de beneficio industrial. El resultado se muestra en la tabla 6.5

Coste.	**Total (€).**
Coste de ejecución material .	41.002.5
Gastos generales y beneficio industrial.	8.200,5
Total.	49.203

Tabla 6.5: Presupuesto de ejecución por contrata.

6.1.6 Honorarios:

Los honorarios facultativos por la ejecución del proyecto se determinan de acuerdo a las tarifas de los honorarios de los ingenieros en trabajos particulares. De acuerdo con la ley 7/1997, de medidas liberalizadoras en materia de suelo y de Colegios Profesionales los Honorarios Profesionales son libres y responden al libre acuerdo entre el profesional y su cliente. Se aplica un coeficiente razonable de un 10 % sobre el importe de ejecución por contrata, debido a que se trata de un proyecto de i+D. Los resultados se encuentran en la Tabla 6.6.

6.1.7 Coste total del proyecto:

Según lo visto en el desglose de las anteriores secciones, se concluye que el coste total del proyecto es de **54.123,3 €**.

Coste.	Total (€).
Importe de ejecución por contrata .	49.203
Honorarios.	4.920,3
Total.	54.123,3

Tabla 6.6: Coste total del proyecto.

Bibliografía

[1] Gonzalo Pajares, Jesús M. de la Cruz. (2001) *Visión por computador. Imágenes digitales y aplicaciones.* Ed: RAMA.

[2] Radu, B. (2009). *Semantic 3D Object Maps for Everyday Manipulation in Human Living Environments* Institut für Informatik der Technischen Universität München

[3] Martin A. Fischler and Robert C. Bolles (1981) *Random Sample Consensus: A Paradigm for Model Fitting with Apphcatlons to Image Analysis and Automated Cartography* SRI International

[4] PointCloud Library Documentation *State of art of Point Cloud processing* http://pointclouds.org/documentation/

[5] OpenCV Documentation, *State of art of computer vision.* https://docs.opencv.org

[6] Open3D Documentation, *Python Wrapper for 3D data processing.* http://www.open3d.org/docs/release/

[7] Redis Documentation, *In-memory data structure store.* https://redis.io/documentation

[8] Librealsense2 Documentation, *Software Development Kit for Intel Realsense Cameras.* https://intelrealsense.github.io/librealsense/doxygen/annotated.html

Apéndice A

Código fuente.

A.1 Pruebas de concepto.

En esta sección se verán los ejemplos de algunos algoritmos del estudio teórico.

A.1.1 Histogramas.

```
1  # -*- coding: utf-8 -*-
2  """
3  Created on Wed Jun 19 22:49:06 2019
4
5  @author: Pedro JosÃ©
6
7  Ejemplo de calculo de histograma con la libreria opencv2.
8  """
9
10 import cv2
11 import numpy as np
12 import matplotlib.pyplot as plt
13
14
15
16 img = cv2.imread("images/tangram.png")
17
18 gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
19 hsv = cv2.cvtColor(img, cv2.COLOR_BGR2HSV)
20 # Se calcula el histograma de niveles de gris
21 hist_gray = cv2.calcHist([gray],[0],None,[256],[0,256])
22 # Se calcula el histograma de niveles de azul
23 hist_b = cv2.calcHist([img],[0],None,[256],[0,256])
24 # Se calcula el histograma de niveles de verde
25 hist_g = cv2.calcHist([img],[1],None,[256],[0,256])
26 # Se calcula el histograma de niveles de rojo
27 hist_r = cv2.calcHist([img],[2],None,[256],[0,256])
28
29 """Visualizacion: """
30 # Figura 1
31 fig = plt.figure()
32 plt.plot(hist_gray)
33 plt.grid()
34 plt.ylabel('NÂ° px')
35 plt.show()
36
```

```
# Figura 2
fig = plt.figure()
plt.plot(hist_b)
plt.plot(hist_r)
plt.plot(hist_g)
plt.gca().legend(('hist_b','hist_r', 'hist_g'))
plt.grid()
plt.ylabel('NÂ° px')
plt.show()

# Imagen
cv2.imshow('color',img)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

A.1.2 Histogramas.

```
# -*- coding: utf-8 -*-
"""
Created on Wed Jun 12 20:33:27 2019

@author: Pedro JosÃ©

Ejemplo de segmentacion de la imagen con una mascara de color,
        de transformaciones morfologicas,
        del concepto de primera y segunda derivada,

"""
# Se importa el modulo opencv2 para trabajar con imagenes en 2D
import cv2
# Se importa el modulo numpy para trabajar con vectores y matrices
import numpy as np
# Se importa el modulo matplotlib para reealizar las graficas
import matplotlib.pyplot as plt


# Se lee la imagen de prueba
img = cv2.imread("images/tangram.png")
# Se guarda una copia en escala de grises
gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

"""Ejemplo de segmentacion de la imagen con una mascara de color"""
# Se guarda una copia en HSV para realizar la mascara
hsv = cv2.cvtColor(img, cv2.COLOR_BGR2HSV)
# Se define el minimo nivel de verde (HSV)
lower_green = np.array([30,0,0])
# Se define el maximo nivel de verde (HSV)
upper_green = np.array([50,255,255])
# Se crea la mascara
mask = cv2.inRange(hsv, lower_green, upper_green)
# se binariza la imagen
img_bin = cv2.bitwise_and(img,img, mask = mask)


"""Ejemplo de transformaciones morfologicas."""
# Se define el nueclo como matriz 5x5 llena de unos
kernel = np.ones((5,5),np.uint8)
erode = cv2.erode(mask, kernel, iterations = 1) # Erosion
dilate = cv2.dilate(mask, kernel, iterations = 1) # Dilatacion
opening = cv2.morphologyEx(mask, cv2.MORPH_OPEN, kernel) # Apertura
closing = cv2.morphologyEx(mask, cv2.MORPH_CLOSE, kernel) # Cierre

"""Concepto de primera derivada"""
```

```
# Se elige la fila 200 de la imagen en escala de grises
fila = gray[200,:]
# Se define el vector de gradientes
grad = []
# Se recorre la fila restando a cada elemento su anterio
for i in range(len(fila)):r
    if i != 0:
        grad.append(fila[i].astype(int)-fila[i-1].astype(int))
# Se convierte a formato numpy para poder hacer la grafica
grad = np.asarray(grad)

"""Concepto de segunda derivada"""
# Se define el vector de laplacianos
lap = []
#Se recorren los gradiente restando a cada elemento su anterior
for i in range(len(grad)):
    if i != 0:
        lap.append(grad[i].astype(int)-grad[i-1].astype(int))
# Se convierte a formato numpy para poder hacer la grafica
lap = np.asarray(lap)


"""Graficas: """
# Se grafica la intensidad de la fila de la imagen original
plt.plot(fila[150:200])
# Se grafica la intensidad de la fila de la primera derivada
plt.plot(grad[150:200])
# Se grafica la intensidad de la fila de la segunda derivada
plt.plot(lap[150:200])
plt.gca().legend(('Intensidad','Gradiente', 'Laplaciano'))
plt.grid()
plt.ylabel('pixeles [200,150:200]')
plt.show()

""" Ejemplo de convoluciones para filtros 2D"""
# Se define el primer nucleo para deteccion horizontal y vertical
kernel1 = np.array([[0, 4, 0],
                    [4,-16, 4],
                    [0, 4, 0]])
# Se define otro nucleo para deteccion vertical
kernel2 = np.array([[0, 0, 0],
                    [1,-2, 1],
                    [0, 0, 0]])
# Se define otro nucleo para deteccion horizontal
kernel3 = np.array([[0, 1, 0],
                    [0,-2, 0],
                    [0, 1, 0]])

result1 = cv2.filter2D(gray, -1, kernel1) # Se aplica el primer nucleo
result2 = cv2.filter2D(gray, -1, kernel2) # Se aplica el segundo nucleo
result2 = cv2.filter2D(gray, -1, kernel3) # Se aplica el tercer nucleo
"""Visualizacion: """
cv2.imshow('gray',gray)
cv2.imshow('filtro vertical',result1)
cv2.imshow('filtro horizontal',result2)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

A.1.3 Canny con OpenCv2.

```
# -*- coding: utf-8 -*-
"""
Created on Wed Jul  3 10:48:58 2019
```

```
4
5  @author: pjvidal
6
7  Ejemplo del uso del algoritmo de Canny para la deteccion de contorno con la
8  libreria opencv.
9  """
10
11 #Se importa el modulo opencv2
12 import cv2
13 #Se importa el modulo numpy
14 import numpy as np
15
16 img = cv2.imread("images/tangram.png")
17 gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
18
19 blank = np.ones(img.shape)
20 #Se aplica el algoritmo de canny a la imagen
21 edges = cv2.Canny(img, 100,200)
22 #Se aplica el algoritmo de busqueda de contornos
23 im2, contours, hierarchy = cv2.findContours(edges,cv2.RETR_TREE,
24                                             cv2.CHAIN_APPROX_SIMPLE)
25 #Se dibujan los contorno en verde
26 cv2.drawContours(blank, contours, -1, (0,255,0), 3)
27 """Visualizacion: """
28 cv2.imshow('color',edges)
29 cv2.waitKey(0)
30 cv2.destroyAllWindows()
```

A.1.4 Arboles KD.

```
1
2  """
3  Created on Mon Jun 17 20:35:13 2019
4
5  @author: Pedro JosÃ©
6  """
7
8  from mpl_toolkits.mplot3d import Axes3D
9  import matplotlib.pyplot as plt
10 from matplotlib import cm
11 from matplotlib.ticker import LinearLocator, FormatStrFormatter
12 import numpy as np
13
14 #Se crea la figura para la visualizacion
15 fig = plt.figure()
16 ax = fig.gca(projection='3d')
17
18 # Se crea la rejilla para las graficas.
19 xs = np.arange(-5, 5, 0.25)
20 ys = np.arange(-5, 5, 0.25)
21 zs = np.arange(-5, 5, 0.25)
22
23
24
25
26
27 # Se crea una nube de puntos aleatoria
28 n_puntos = 100
29 pointcloud = ((np.random.rand(n_puntos,3)*2) + np.array([-1,-1,-1])) * 4
30 #Se visualiza en negro
31 ax.scatter(pointcloud[:,0], pointcloud[:,1], pointcloud[:,2], color = "black",
32                                              s = 10, alpha = 0.3)
33
```

```
34
35  """""""
36  # Plano perpendicular a Y
37  X, Z = np.meshgrid(xs, zs)
38  # Calculo de la mediana que divide la nube de puntos por primera vez
39  medianaY = np.median(pointcloud[:,1])
40  print("La mediana del tronco: ", medianaY)
41  # Se coloca el plano de division coincidente con la mediana
42  Y = np.zeros(np.shape(X)) + medianaY
43  # Se visualiza el plano en azul
44  ax.plot_wireframe(X, Y, Z, rstride=10, cstride=10, color = "blue")
45
46  """******** Segundo nivel de division del espacio. **********************"""
47  #Plano perpendicular a Z
48  X, Y = np.meshgrid(xs, ys)
49  # Rama 1. Segunda division del espacio
50  # Cogemos los puntos por encima de la mediana en el eje Y
51  rama1 = pointcloud[pointcloud[:,1] >=  medianaY]
52  # Se calcula el punto que divide por la mitad a la rama 1.
53  medianaZ1 = np.median(rama1[:,2])
54  print("La mediana de la primera rama: ", medianaZ1)
55  # Se coloca el plano de division coincidente con la mediana
56  Z = np.zeros(np.shape(X)) + medianaZ1
57  # Se buscan los puntos del plano por encima de la mediana para visualizar
58  ids1 = np.where(Y[:,1] >= medianaY)[0]
59  ax.plot_wireframe(X[ids1,:], Y[ids1,:], Z[ids1,:] , rstride=10, cstride=10,
60                                                       color = "green")
61
62
63  # Rama 2. Tarcera division del espacio. En el mismo nivel que la anterior
64  # Cogemos los puntos menores de la mediana en el eje Y
65  rama2 = pointcloud[pointcloud[:,1] <=  medianaY]
66  # Se calcula el punto que divide por la mitad a la rama 2
67  medianaZ2 = np.median(rama2[:,2])
68  print("La mediana de la segunda rama: ", medianaZ2)
69  # Se coloca el plano de division coincidente con la mediana
70  Z = np.zeros(np.shape(X)) + medianaZ2
71  # Se buscan los puntos del plano por debajo de la mediana para visualizar
72  ids2 = np.where(Y[:,1] <= medianaY)[0]
73  ax.plot_wireframe(X[ids2,:], Y[ids2,:], Z[ids2,:] , rstride=10, cstride=10,
74                                                       color = "green")
75
76  """******** Tercer nivel de division del espacio. **********************"""
77  #Plano perpendicular a X
78  Y, Z = np.meshgrid(xs, ys)
79  # Hoja 1. Cuarta division del espacio
80  hoja1 = rama1[rama1[:,2] <= medianaZ1]
81  # Se calcula el punto que divide por la mitad a la hoja 1
82  medianaX1 = np.median(hoja1[:,0])
83  print("La mediana de la hoja 1: ", medianaX1)
84  # Se coloca el plano de division coincidente con la mediana
85  X = np.zeros(np.shape(X)) + medianaX1
86  # Se buscan los puntos del plano por debajo de la mediana para visualizar
87  ids3 = np.where(Z[:,0] <= medianaZ1)[0]
88  ax.plot_wireframe(X[ids3,min(ids1):max(ids1)],
89                    Y[ids3,min(ids1):max(ids1)],
90                    Z[ids3,min(ids1):max(ids1)] ,
91                    rstride=10, cstride=10, color = "red")
92
93
94  # Hoja 2. Quinta division del espacio. Mismo nivel que el anterior.
95  hoja2 = rama1[rama1[:,2] >= medianaZ1]
96  medianaX2 = np.median(hoja2[:,0])
```

```
97  print("La mediana de la hoja 2: ", medianaX2)
98  X = np.zeros(np.shape(X)) + medianaX2
99  ids3 = np.where(Z[:,0] >= medianaZ1)[0]
100 ax.plot_wireframe(X[ids3,min(ids1):max(ids1)],
101                   Y[ids3,min(ids1):max(ids1)],
102                   Z[ids3,min(ids1):max(ids1)],
103                   rstride=10, cstride=10, color = "red")
104
105 # Hoja 3. Sexta division del espacio. Mismo nivel que el anterior.
106 hoja3 = rama2[rama2[:,2] <= medianaZ2]
107 medianaX3 = np.median(hoja3[:,0])
108 print("La mediana de la hoja 3: ", medianaX3)
109 X = np.zeros(np.shape(X)) + medianaX3
110 ids3 = np.where(Z[:,0] <= medianaZ2)[0]
111 ax.plot_wireframe(X[ids3,min(ids2):max(ids2)],
112                   Y[ids3,min(ids2):max(ids2)],
113                   Z[ids3,min(ids2):max(ids2)],
114                   rstride=10, cstride=10, color = "red")
115
116 # Hoja 4. Septima division del espacio. Mismo nivel que el anterior.
117 hoja4 = rama2[rama2[:,2] >= medianaZ2]
118 medianaX4 = np.median(hoja4[:,0])
119 print("La mediana de la hoja 4: ", medianaX4)
120 X = np.zeros(np.shape(X)) + medianaX4
121 ids3 = np.where(Z[:,0] >= medianaZ2)[0]
122 ax.plot_wireframe(X[ids3,min(ids2):max(ids2)],
123                   Y[ids3,min(ids2):max(ids2)],
124                   Z[ids3,min(ids2):max(ids2)],
125                   rstride=10, cstride=10, color = "red")
126
127 """**********************************************************************"""
128 nodo1 = hoja1[hoja1[:,0] >= medianaX1]
129 nodo2 = hoja1[hoja1[:,0] <= medianaX1]
130
131 nodo3 = hoja2[hoja2[:,0] >= medianaX2]
132 nodo4 = hoja2[hoja2[:,0] <= medianaX2]
133
134 nodo5 = hoja3[hoja3[:,0] >= medianaX3]
135 nodo6 = hoja3[hoja3[:,0] <= medianaX3]
136
137 nodo7 = hoja4[hoja4[:,0] >= medianaX4]
138 nodo8 = hoja4[hoja4[:,0] <= medianaX4]
139
140
141 #Visualizacion de los puntos pertenecientes a cada nodo.
142 ax.scatter(nodo1[:,0], nodo1[:,1], nodo1[:,2], color = "yellow",
143            marker = "d", s = 50, alpha = 1.0)
144 ax.scatter(nodo2[:,0], nodo2[:,1], nodo2[:,2], color = "black",
145            marker = "d", s = 50, alpha = 1.0)
146 ax.scatter(nodo3[:,0], nodo3[:,1], nodo3[:,2], color = "purple",
147            marker = "d", s = 50, alpha = 1.0)
148 ax.scatter(nodo4[:,0], nodo4[:,1], nodo4[:,2], color = "magenta",
149            marker = "d", s = 50, alpha = 1.0)
150
151 ax.scatter(nodo5[:,0], nodo5[:,1], nodo5[:,2], color = "pink",
152            marker = "d", s = 50, alpha = 1.0)
153 ax.scatter(nodo6[:,0], nodo6[:,1], nodo6[:,2], color = "lime",
154            marker = "d", s = 50, alpha = 1.0)
155 ax.scatter(nodo7[:,0], nodo7[:,1], nodo7[:,2], color = "olive",
156            marker = "d", s = 50, alpha = 1.0)
157 ax.scatter(nodo8[:,0], nodo8[:,1], nodo8[:,2], color = "cyan",
158            marker = "d", s = 50, alpha = 1.0)
159
```

```
# Se muestra la figura
plt.show()
```

A.1.5 K-Medias.

```
# -*- coding: utf-8 -*-
"""
Created on Sun Jun 23 17:58:26 2019

@author: Pedro JosÃ©
"""


import matplotlib.pyplot as plt
import numpy as np
import mpl_toolkits.mplot3d as plt3d

# Se definen las tres figuras
fig0 = plt.figure()
fig1 = plt.figure()
fig2 = plt.figure()
ax0 = fig0.gca(projection='3d')
ax1 = fig1.gca(projection='3d')
ax2 = fig2.gca(projection='3d')



#Se lee la nube de puntos con las paredes quitadas
pointcloud = np.loadtxt("pcd_3_objetos.txt")
# Se muestran en la figura 0 y 1
ax0.scatter(pointcloud[:,0], pointcloud[:,1], pointcloud[:,2],
            color = "black", s = 10, alpha = 0.3)
ax1.scatter(pointcloud[:,0], pointcloud[:,1], pointcloud[:,2],
            color = "black", s = 10, alpha = 0.3)

# Se definen unos centroides iniciales
initial_centroids = np.array([[-0.01,0.08,0.4],[0.0,0.08,0.4],[0.01,0.08,0.4]])

# Se define un array de colores, uno para cada cluster.
colors = ["red","lime", "cyan"]
# Se muestran y colorean los centroides iniciales en las figuras 0 y 1
for i in range(len(colors)):
    ax0.scatter(initial_centroids[i,0], initial_centroids[i,1],
                initial_centroids[i,2], color = colors[i],
                s = 20, alpha = 1.0)
    ax1.scatter(initial_centroids[i,0], initial_centroids[i,1],
                initial_centroids[i,2], color = colors[i],
                s = 20, alpha = 1.0)

def find_closest_centroids(centroids, points):
    """ Funcion que calcula los centroides mas cercanos a cada punto
     de una nube de puntos.

     Entradas:
         centroids : Conjunto de centroides.
         points : Nube de puntos.
     Salidas:
         distancia_minima[:,0] : Lista de distancias entre cada punto y su
             centroide mas cercano.
         idx[:,0]: Lista de ids que asocia cada punto con un centroide.
     """
    # Se define el array de distancias
    distances = np.zeros(shape = (len(centroids),1))
```

```
59      # Se define el array de distancias minimas
60      distancia_minima = np.zeros(shape = (len(points),1))
61      # Se define el array de ids
62      idx = np.zeros(shape = (len(points),1))
63
64      # Se recorre la nube de puntos
65      for n in range(len(points)):
66          # Para cada punto se calcula la norma euclidiana a cada centroide
67          for j in range(len(centroids)):
68              distances[j] = np.linalg.norm(points[n,:] - centroids[j,:])**2
69
70          # Para cada punto se guarda la distancia minima calculada
71          distancia_minima[n] = min(distances)
72          # Se busca el id del centroide cuya distancia es minima
73          idx[n] = np.where(distances == min(distances))[0][0]
74
75      return distancia_minima[:,0], idx[:,0]
76
77  def compute_centroids(initial_centroids, pointcloud, idx):
78       """ Funcion que desplaza los centroides hacia la media de los puntos mas
79       cercanos a cada uno.
80
81       Entradas:
82           initial_centroids: Posiciones de los centroides que se quieren
83               desplazar
84           pointcloud: Nube de puntos
85           idx: Lista de ids que asocia cada punto con un centroide.
86       Salidas:
87           new_centroids: Posiciones de los nuevos centroides
88       """
89      # Se define el vector de nuevos centroides
90      new_centroids = np.zeros(shape = initial_centroids.shape)
91      # Para cada centroide
92      for i in range(len(initial_centroids)):
93          # Se calcula la media de los puntos que pertenecen a el.
94          new_centroids[i] = np.mean(pointcloud[idx==i], axis = 0)
95
96      return new_centroids
97
98  # Se aplica el elgoritmo n veces
99  for n in range(10):
100     # Se buscan los centroides mas cercanos a cada punto
101     distancia_minima, idx = find_closest_centroids(initial_centroids,
102                                                    pointcloud)
103     # Se actualiza la posicion de los centroides
104     new_centroids = compute_centroids(initial_centroids, pointcloud, idx)
105
106     """ Visualizacion del camino que siguen los centros de los clusters"""
107     for i in range(len(colors)):
108         ax1.scatter(initial_centroids[i,0], initial_centroids[i,1],
109                     initial_centroids[i,2], color = colors[i],
110                     s = 20, alpha = 1.0)
111         xs = [initial_centroids[i,0], new_centroids[i,0]]
112         ys = [initial_centroids[i,1], new_centroids[i,1]]
113         zs = [initial_centroids[i,2], new_centroids[i,2]]
114         line = plt3d.art3d.Line3D(xs, ys, zs)
115         ax1.add_line(line)
116     # Se guardan los centroides nuevos
117     initial_centroids = new_centroids
118
119
120
121 """ Visualizacion de los tres clusters formados en la figura 2. """
```

```
for i in range(len(colors)):
    ax2.scatter(new_centroids[i,0], new_centroids[i,1], new_centroids[i,2],
                color = colors[i], s = 20, alpha = 1.0)
    ax2.scatter(pointcloud[idx==i,0], pointcloud[idx==i,1],
                pointcloud[idx==i,2], color = colors[i],
                s = 20, alpha = 1.0)
```

A.1.6 Transformada de Hough.

```
%% Transformada de Hough con la implementacion de la toolbox de matlab

I= imread('vision_python/images/tangram.png');
gray = rgb2gray(I);
J = imcomplement(gray);

BW = edge(gray, 'canny');
[H,T,R] = hough(BW);

figure(1)
imshow(H,[],'XData', T, 'YData', R,...
            'InitialMagnification', 'fit');
xlabel('\theta'), ylabel('\rho');
axis on, axis normal, hold on;
colormap(gca,hot);
figure(5)
imshow(BW);

[X,Y] = meshgrid(1:1:length(H(1,:)),1:1:length(H(:,1)));
Z = H;
surf(X,Y,Z);

%% Implementacion propia de la transformada de Hough
% Tenemos las lineas que pasan por cada punto (rectas 1) entre los puntos p1 y p2.
% Tenemos las lineas perpendiculares a las lineas anteriores y que pasan
% por los puntos (0,0) y p3 (rectas 2)

%Se define una linea recta
x = 0:10:100
y = x*0.5 + 113;
%Se define otra linea recta
xa = 0:10:100;
ya = xa*1.5 + 20;
%Se juntan los puntos pertenecientes a las dos lineas
x(end:end+length(xa)-1) = xa;
y(end:end+length(ya)-1) = ya;
%Se define el array de angulos
alpha = [-89:0.5:89];

%Se visualizan los puntos de las dos rectas, que corresponden a una
%  simplificacion de la imagen binarizada.
figure(2);
hold on, grid on;
```

```
axis on, axis normal;
title("Puntos de la imagen binarizada")
scatter(x,y)

%Para cada punto se calculan:
for c = 1:length(x)
    x1 = x(c);
    y1 = y(c);
    %Las rectas que pasan por el punto con distinto angulo theta
    x21 = x1+x1*cosd(alpha);
    y21 = y1+y1*sind(alpha);
    x20 = x1-x1*cosd(alpha);
    y20 = y1-y1*sind(alpha);
    x2 = x21
    y2 = y21

    %Las rectas perpendiculares a las anteriores que pasan por el origen
    o = [0,0];
    m1 = (y1-y2)./(x1-x2);
    A = (y2-y1)./(x2-x1);
    B = (x1*y1-x1*y2)./(x2-x1) + y1;

    %Los untos de corte con las rectas1
    x3 = -B./(A + 1./m1);
    y3 = -x3./m1;

    %La pendiente de las lineas ortogonales(rectas2):
    m3 = (y3)./(x3);
    theta = atand(m3);
    %La longitud del segmento de las lineas ortogonales (rectas2)
    %  del origen a las recta1:
    dist = x3./cosd(theta);

    %Visualizacion
    figure(4)
    hold on, grid on;
    axis on, axis normal;
    xlabel('\theta'), ylabel('\rho');
    title("Puntos en el espacio de Hough")
    plot(theta, dist)

end
```

A.1.7 Ajuste de planos.

```
""" Ejemplo de ajuste de plano por tres puntos, mediante la ecuacion del plano."""

import numpy as np

import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

```

```
9
10 def calcular_distancias_al_plano(h, w, parameters, offset_z):
11     plano = np.empty((h, w), dtype=np.float32)
12     for i in range(w):
13         for j in range(h):
14             plano[j,i] = j*parameters[1] + i*parameters[2] + parameters[0] + offset_z
15     return plano
16
17
18 pointcloud = np.loadtxt("pcd_con_paredes.txt")
19
20
21 def ajustar_plano_por_3_puntos(puntos):
22     """Funcion que devuelve los parametros de un plano a partir de los tres
23     puntos que pasan por el. La obtencion del plano se obtiene igualando a 0 el
24     determinante de una matriz de vectores donde uno de ellos es linealmente
25     dependiente.
26     x*a + y*b + z*c + d = 0
27     Entradas:
28         puntos: Tres puntos.
29     Salidas: a,b,c,d"""
30     # Se le da formato alas coordenadas de los puntos para poder operar con
31     # ellos mÃ¡s facil:
32     x1 = puntos[0][0]
33     y1 = puntos[0][1]
34     z1 = puntos[0][2]
35
36     x2 =  puntos[1][0]
37     y2 =  puntos[1][1]
38     z2 =  puntos[1][2]
39
40     x3 =  puntos[2][0]
41     y3 =  puntos[2][1]
42     z3 =  puntos[2][2]
43
44     # Se obtiene cada elemento del determinante
45     A = x2-x1
46     B = y2-y1
47     C = z2-z1
48     D = x3-x1
49     E = y3-y1
50     F = z3-z1
51     # Se opera:
52     a = B*F-C*E
53     b = C*D-A*F
54     c = A*E-B*D
55     d = -(x1*a + y1*b + z1*c)
56
57     return a,b,c,d
58
59
60
61 """Visualizacion"""
62 fig = plt.figure()
63 ax = fig.gca(projection='3d')
64 plt.xlabel("X")
65 plt.ylabel("Y")
66
67 # Se define la rejilla
68 x = np.linspace(min(pointcloud[:,0]),max(pointcloud[:,0]),10)
69 y = np.linspace(min(pointcloud[:,1]),max(pointcloud[:,1]),10)
70 z = np.linspace(min(pointcloud[:,2]),max(pointcloud[:,2]),10)
71
```

```
#Plano perpendicular a z(pared)
puntos = [pointcloud[100].tolist(), pointcloud[120].tolist(),
          pointcloud[148].tolist()] #Pared
pnt = np.asarray(puntos)
a1,b1,c1,d1 = ajustar_plano_por_3_puntos(puntos)
X,Y = np.meshgrid(x,y)
Z = (-d1 - a1*X - b1*Y) / c1
ax.plot_surface(X, Y, Z, rstride=1, cstride=1, color = "lime")

#Plano perpendicular a Y(suelo)
puntos = [pointcloud[3128].tolist(), pointcloud[3584].tolist(),
          pointcloud[113].tolist()] #Pared
pnt = np.asarray(puntos)
a2,b2,c2,d2 = ajustar_plano_por_3_puntos(puntos)
X, Z = np.meshgrid(x, z)
Y = (-d2 - a2*X - c2*Z) / b2
ax.plot_surface(X, Y, Z, rstride=1, cstride=1, color = "cyan")
ax.scatter(pnt[:,0], pnt[:,1], pnt[:,2], color = "red", s = 20, alpha = 1)

#Vectores directores
soa = np.array([[0, 0, -d1/c1, a1*5, b1*5, c1*5],[0, -d2/b2, 0, a2*5, b2*5, c2*5]])
Xvec, Yvec, Zvec, Uvec, Vvec, Wvec = zip(*soa)
ax.quiver(Xvec, Yvec, Zvec, Uvec, Vvec, Wvec)

#Visualizacion de la nube de puntos
ax.scatter(pointcloud[:,0], pointcloud[:,1], pointcloud[:,2],
           color = "black", s = 10, alpha = 0.3)
ax.scatter(pnt[:,0], pnt[:,1], pnt[:,2], color = "red", s = 20, alpha = 1)
```

A.1.8 RANSAC.

```
# -*- coding: utf-8 -*-
"""
Created on Thu Jun 27 21:50:46 2019

@author: Pedro JosÃ©

Prueba de concepto del metodo RANSAC, se usa la ecuacion del plano para probar
los puntos contra el modelo, que en este caso es un plano. Como estimador de
error se usa el error cuadratico medio.
"""

import numpy as np
from sklearn import linear_model, datasets
from sklearn import neighbors
from sklearn.cluster import KMeans

import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

import random

def ajustar_plano_por_3_puntos(puntos):
    x1 = puntos[0][0]
    y1 = puntos[0][1]
    z1 = puntos[0][2]

    x2 =  puntos[1][0]
    y2 =  puntos[1][1]
    z2 =  puntos[1][2]

    x3 =  puntos[2][0]
    y3 =  puntos[2][1]
```

```
33      z3 =  puntos[2][2]
34
35      A = x2-x1
36      B = y2-y1
37      C = z2-z1
38      D = x3-x1
39      E = y3-y1
40      F = z3-z1
41
42      a = B*F-C*E
43      b = C*D-A*F
44      c = A*E-B*D
45      d = -(x1*a + y1*b + z1*c)
46
47      return a,b,c,d
48
49
50  def three_random_index(max_id):
51      a = random.randint(0, max_id)
52      b = random.randint(0, max_id)
53      while b == a:
54          b = random.randint(0, max_id)
55
56      c = random.randint(0, max_id)
57      while c == a or c == b:
58          c = random.randint(0, max_id)
59
60      return [a,b,c]
61
62
63  def ransac_casero(pointcloud_i, error_max, n_valid_inliers, iteracciones, tree):
64      """ FUncion que ejecuta el metodo Ransac en el que el estimador de error
65          es la distancia al plano de cada punto. El modelo se ajusta con tres
66          puntos aleatorios, esto no es muy eficiente, pero para probar es valido. """
67      a1 = 0
68      b1 = 0
69      c1 = 0
70      d1 = 0
71      best_error = 10000
72      best_inliers = 0
73      mejor_distancias_al_plano = 0
74      for k in range(iteracciones):
75          """Se eligen los posible inliers aleatoriamente"""
76          max_id = len(pointcloud_i)-1
77          # Se coge un punto al azar de toda la nube
78          randid_to_query = random.randint(0, max_id)
79          # Se buscan los 50 mas cercanos
80          dist, ind = tree.query([pointcloud_i[randid_to_query]], k=100)
81          ind = np.asarray(ind)[0]
82          max_id = len(ind)-1
83          # cogen tres puntos al azar de entre los 50 mas cercanos, el numero mayor
84          #que se puede generar tiene que ser el maximo id de la nube de puntos.
85          randidx = three_random_index(max_id)
86
87          idx = np.zeros(shape = (len(pointcloud_i)), dtype=bool)
88          idx[randidx] = True
89
90
91          maybe_inliers = pointcloud_i[idx] # maibe inliers
92          """Se calcula el modelo a partir de los puntos aleatorios"""
93          a,b,c,d = ajustar_plano_por_3_puntos(maybe_inliers) # maibe inliers
94
95
```

```
96          also_inliers = []
97          distancias_al_plano = []
98          """Cada uno de los puntos de la nube se prueba contra el modelo ajustado"""
99          for i in range(len(pointcloud_i)):
100             distancia_al_plano = np.sqrt((a*pointcloud_i[i][0] +
101                                           b*pointcloud_i[i][1] +
102                                           c*pointcloud_i[i][2] +
103                                           d)**2)/np.sqrt(a**2 + b**2 + c**2)
104
105             # Si el error es menor que error_max:
106             # se considera valido el modelo para ese punto
107             if distancia_al_plano < error_max:
108                 also_inliers.append(i)
109                 distancias_al_plano.append(distancia_al_plano)
110             # Si el modelo tiene mas de n_valid_inliers se calcula el error
111             #    del modelo con todos los inliers
112         if len(also_inliers) > n_valid_inliers:
113             # Este estimador de error es un poco cutre
114             error_del_modelo = np.mean(distancias_al_plano)
115             """Nos quedamos con el modelo que de un menor error"""
116             if error_del_modelo < best_error:
117                 a1 = a
118                 b1 = b
119                 c1 = c
120                 d1 = d
121                 best_inliers = also_inliers
122                 best_error = error_del_modelo
123                 mejor_distancias_al_plano = np.asarray(distancias_al_plano)
124
125
126     return a1, b1, c1, d1, best_inliers, mejor_distancias_al_plano, best_error
127
128 # Se carga la nube de puntos
129 pointcloud = np.loadtxt("pcd_con_paredes.txt")
130 # Se calcula el arbol kd para buscar los puntos cercanos entre ellos mas adelante
131 tree = neighbors.KDTree(pointcloud, leaf_size=2)
132 # Se define la figura
133 fig = plt.figure()
134 ax = fig.gca(projection='3d')
135 """ Definicion de los parametros del algoritmo: """
136 # Se define el error maximo que los puntos deberÃ¡n tener para considerarlos
137 # pertenecientes al plano.
138 error_max = 0.01
139 # Se define el numero de puntos que deben de cumplir con el error definido
140 # para considerar el plano como un plano valido.
141 n_valid_inliers = len(pointcloud)/2
142 # Numero de iteraciones del metodo Ransac
143 iteracciones = 10
144 # Se ejecuta el metodo Ransac
145 a1, b1, c1, d1, inliers1, mejor_distancias_al_plano1, best_error1 = ransac_casero(
146         pointcloud, error_max, n_valid_inliers, iteracciones, tree)
147 print("RANSAC ha terminado.")
148 print("Se ha encontrado el plano: ", a1, b1, c1, d1)
149 print("Con un error de: ", best_error1)
150 # Se define la rejilla
151 x = np.linspace(min(pointcloud[:,0]),max(pointcloud[:,0]),10)
152 y = np.linspace(min(pointcloud[:,1]),max(pointcloud[:,1]),10)
153 z = np.linspace(min(pointcloud[:,2]),max(pointcloud[:,2]),10)
154 # Se define el plano ajustado por Ransaca para su visualizacion
155 X,Y = np.meshgrid(x,y)
156 Z = (-d1 - a1*X - b1*Y) / c1
157 ax.scatter(pointcloud[inliers1,0],
158            pointcloud[inliers1,1],
```

```
            pointcloud[inliers1,2], s = 10,
            alpha = 0.3, color="red")

# Se elimina el plano de la nube de puntos para la correcta segmentacion
# de los objetos
id_remains = np.ones((len(pointcloud)), dtype=bool)
id_remains[inliers1] = False
pointcloud_remanente = pointcloud[id_remains]


"""Se pasa por segunda vez para encontrar otro plano"""
n_valid_inliers = len(pointcloud_remanente)/2
error_max = 0.01
iteracciones = 50
tree2 = neighbors.KDTree(pointcloud_remanente, leaf_size=2)

a2, b2, c2, d2, inliers2, mejor_distancias_al_plano2, best_error2 = ransac_casero(
        pointcloud_remanente, error_max, n_valid_inliers, iteracciones,tree2)


print("RANSAC ha terminado.")
print("Se ha encontrado el plano: ", a2, b2, c2, d2)
print("Con un error de: ", best_error2)
# Si el error no es muy grande se visualiza, si no, se asume que el metodo
# ha fallado
if best_error2 < 1:
    X,Y = np.meshgrid(x,y)
    Z = (-d2 - a2*X - b2*Y) / c2
    ax.scatter(pointcloud_remanente[inliers2,0],
               pointcloud_remanente[inliers2,1],
               pointcloud_remanente[inliers2,2], s = 10,
               alpha = 0.3, color="green")

    id_obejetos = np.ones((len(pointcloud_remanente)), dtype=bool)
    id_obejetos[inliers2] = False
    pointcloud_obejetos = pointcloud_remanente[id_obejetos]

    ax.scatter(pointcloud_obejetos[:,0],
               pointcloud_obejetos[:,1],
               pointcloud_obejetos[:,2], s = 10,
               alpha = 0.3, color="blue")



# Se dibujan los vectores normales a los dos planos
Xvec, Yvec, Zvec, Uvec, Vvec, Wvec = 0,0,0,a1*2,b1*2,c1*2
ax.quiver(Xvec, Yvec, Zvec, Uvec, Vvec, Wvec, color = "black")
Xvec, Yvec, Zvec, Uvec, Vvec, Wvec = 0,0,0,a2*3,b2*3,c2*3
ax.quiver(Xvec, Yvec, Zvec, Uvec, Vvec, Wvec, color = "black")

# Se visualiza la nube de puntos sin paredes ya
fig1 = plt.figure()
ax1 = fig1.gca(projection='3d')
ax1.scatter(pointcloud_obejetos[:,0],
            pointcloud_obejetos[:,1],
            pointcloud_obejetos[:,2], s = 10,
            alpha = 0.3, color="black")
```

A.1.9 Calculo de vectores normales a una superficie.

```
# -*- coding: utf-8 -*-
"""
Created on Wed Jun 26 19:32:39 2019
```

```
4
5  @author: Pedro JosÃ©
6
7  Calculo de los vectores normales a las superficies de una nube de puntos.
8  """
9  import numpy as np
10 from sklearn import linear_model, datasets
11 from sklearn import neighbors
12 from sklearn.cluster import KMeans
13
14 import matplotlib.pyplot as plt
15 from mpl_toolkits.mplot3d import Axes3D
16
17 def ajustar_plano_por_3_puntos(puntos):
18     x1 = puntos[0][0]
19     y1 = puntos[0][1]
20     z1 = puntos[0][2]
21
22     x2 =  puntos[1][0]
23     y2 =  puntos[1][1]
24     z2 =  puntos[1][2]
25
26     x3 =  puntos[2][0]
27     y3 =  puntos[2][1]
28     z3 =  puntos[2][2]
29
30     A = x2-x1
31     B = y2-y1
32     C = z2-z1
33     D = x3-x1
34     E = y3-y1
35     F = z3-z1
36
37     a = B*F-C*E
38     b = C*D-A*F
39     c = A*E-B*D
40     d = -(x1*a + y1*b + z1*c)
41
42     return a,b,c,d
43
44 pointcloud = np.loadtxt("pcd_con_paredes.txt")
45 tree = neighbors.KDTree(pointcloud, leaf_size=2)
46
47 fig = plt.figure()
48 ax = fig.gca(projection='3d')
49
50 for i in range(0,len(pointcloud)):
51     dist, ind = tree.query([pointcloud[i]], k=3)
52     a,b,c,d = ajustar_plano_por_3_puntos(pointcloud[ind][0])
53     vector_director = np.array([[pointcloud[i,0], pointcloud[i,1],
54                                 pointcloud[i,2], a*100, b*100, c*100]])
55     #vector_director = np.array([[0, 0,0, a*100, b*100, c*100]])
56     if i == 0:
57         soa = vector_director
58     if i != 0:
59         soan = vector_director
60
61         soa = np.append(soa,soan, axis = 0)
62
63     #print(a/norm)
64
65
66 plt.xlabel("X")
```

```
plt.ylabel("Y")
Xvec, Yvec, Zvec, Uvec, Vvec, Wvec = zip(*soa)
ax.quiver(Xvec, Yvec, Zvec, Uvec, Vvec, Wvec)
```

A.1.10 Todo junto:

```
# -*- coding: utf-8 -*-
"""
Created on Thu Jun 27 21:50:46 2019

@author: Pedro JosÃ©

Prueba de los metodos Ransac para eliminar los planos de las paredes,
el metodo kmeans para segmentar los tres objetos y el metodo ACP para encontrar
sus ejes principales.
"""

import numpy as np
from sklearn import linear_model, datasets
from sklearn import neighbors
from sklearn.cluster import KMeans

import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from numpy import linalg as LA
import random

def ajustar_plano_por_3_puntos(puntos):
    x1 = puntos[0][0]
    y1 = puntos[0][1]
    z1 = puntos[0][2]

    x2 =  puntos[1][0]
    y2 =  puntos[1][1]
    z2 =  puntos[1][2]

    x3 =  puntos[2][0]
    y3 =  puntos[2][1]
    z3 =  puntos[2][2]

    A = x2-x1
    B = y2-y1
    C = z2-z1
    D = x3-x1
    E = y3-y1
    F = z3-z1

    a = B*F-C*E
    b = C*D-A*F
    c = A*E-B*D
    d = -(x1*a + y1*b + z1*c)

    return a,b,c,d


def three_random_index(max_id):
    a = random.randint(0, max_id)
    b = random.randint(0, max_id)
    while b == a:
        b = random.randint(0, max_id)

    c = random.randint(0, max_id)
    while c == a or c == b:
```

```
58          c = random.randint(0, max_id)
59
60      return [a,b,c]
61
62  """Ransac en el que el estimador de error es la distancia al plano de cada punto.
63   El modelo se ajusta con tres puntos aleatorios, esto no es muy eficiente"""
64  def ransac_casero(pointcloud_i, error_max, n_valid_inliers, iteracciones, tree):
65
66      a1 = 0
67      b1 = 0
68      c1 = 0
69      d1 = 0
70      best_error = 10000
71      best_inliers = 0
72      mejor_distancias_al_plano = 0
73      for k in range(iteracciones):
74          """Se eligen los posible inliers aleatoriamente"""
75          max_id = len(pointcloud_i)-1
76          # Se coge un punto al azar de toda la nube
77          randid_to_query = random.randint(0, max_id)
78          # Se buscan los 50 mas cercanos
79          dist, ind = tree.query([pointcloud_i[randid_to_query]], k=100)
80          ind = np.asarray(ind)[0]
81
82          max_id = len(ind)-1
83          # cogen tres puntos al azar de entre los 50 mas cercanos
84          randidx = three_random_index(max_id)
85
86          idx = np.zeros(shape = (len(pointcloud_i)), dtype=bool)
87          idx[randidx] = True
88          #idx = idx.tolist()
89
90          maybe_inliers = pointcloud_i[idx] # maibe inliers
91          """Se calcula el modelo a partir de los puntos aleatorios"""
92          a,b,c,d = ajustar_plano_por_3_puntos(maybe_inliers) # maibe inliers
93
94          #resto_de_puntos = pointcloud[~idx]
95          also_inliers = []
96          distancias_al_plano = []
97          """Cada uno de los puntos de la nube se prueba contra el modelo ajustado"""
98          for i in range(len(pointcloud_i)):
99              distancia_al_plano = np.sqrt((a*pointcloud_i[i][0]
100             + b*pointcloud_i[i][1]
101             + c*pointcloud_i[i][2]
102             + d)**2)/np.sqrt(a**2 + b**2 + c**2)
103
104             # Si el error es menor que error_max se considera valido el modelo
105             # para ese punto
106             if distancia_al_plano < error_max:
107                 also_inliers.append(i)
108                 distancias_al_plano.append(distancia_al_plano)
109         """ Si el modelo tiene mas de n_valid_inliers se calcula el error del
110         modelo con todos los inliers"""
111         if len(also_inliers) > n_valid_inliers:
112             # Este estimador de error es un poco cutre
113             error_del_modelo = np.mean(distancias_al_plano)
114             """Nos quedamos con el modelo que de un menor error"""
115             if error_del_modelo < best_error:
116                 a1 = a
117                 b1 = b
118                 c1 = c
119                 d1 = d
120                 best_inliers = also_inliers
```

```
121                 best_error = error_del_modelo
122                 mejor_distancias_al_plano = np.asarray(distancias_al_plano)
123 
124 
125     return a1, b1, c1, d1, best_inliers, mejor_distancias_al_plano, best_error
126 
127 pointcloud = np.loadtxt("pcd_con_paredes.txt")
128 tree = neighbors.KDTree(pointcloud, leaf_size=2)
129 
130 fig = plt.figure()
131 ax = fig.gca(projection='3d')
132 
133 error_max = 0.01
134 n_valid_inliers = len(pointcloud)/2
135 iteracciones = 10
136 a1, b1, c1, d1, inliers1, mejor_distancias_al_plano1, best_error1 = ransac_casero(
137         pointcloud, error_max, n_valid_inliers, iteracciones, tree)
138 print("RANSAC ha terminado.")
139 print("Se ha encontrado el plano: ", a1, b1, c1, d1)
140 print("Con un error de: ", best_error1)
141 
142 x = np.linspace(min(pointcloud[:,0]),max(pointcloud[:,0]),10)
143 y = np.linspace(min(pointcloud[:,1]),max(pointcloud[:,1]),10)
144 z = np.linspace(min(pointcloud[:,2]),max(pointcloud[:,2]),10)
145 
146 X,Y = np.meshgrid(x,y)
147 Z = (-d1 - a1*X - b1*Y) / c1
148 ax.scatter(pointcloud[inliers1,0],
149            pointcloud[inliers1,1],
150            pointcloud[inliers1,2],s = 10, alpha = 0.3, color="red")
151 
152 
153 id_remains = np.ones((len(pointcloud)), dtype=bool)
154 id_remains[inliers1] = False
155 pointcloud_remanente = pointcloud[id_remains]
156 
157 
158 """Se pasa por segunda vez para encontrar otro plano"""
159 n_valid_inliers = len(pointcloud_remanente)/2
160 error_max = 0.01
161 iteracciones = 50
162 tree2 = neighbors.KDTree(pointcloud_remanente, leaf_size=2)
163 
164 a2, b2, c2, d2, inliers2, mejor_distancias_al_plano2, best_error2 = ransac_casero(
165         pointcloud_remanente, error_max, n_valid_inliers, iteracciones,tree2)
166 
167 
168 print("RANSAC ha terminado.")
169 print("Se ha encontrado el plano: ", a2, b2, c2, d2)
170 print("Con un error de: ", best_error2)
171 if best_error2 < 1:
172     X,Y = np.meshgrid(x,y)
173     Z = (-d2 - a2*X - b2*Y) / c2
174     ax.scatter(pointcloud_remanente[inliers2,0],
175                pointcloud_remanente[inliers2,1],
176                pointcloud_remanente[inliers2,2],s = 10, alpha = 0.3, color="green")
177 
178     id_obejetos = np.ones((len(pointcloud_remanente)), dtype=bool)
179     id_obejetos[inliers2] = False
180     pointcloud_obejetos = pointcloud_remanente[id_obejetos]
181 
182     #ax.scatter(pointcloud_obejetos[:,0],
183                 #pointcloud_obejetos[:,1],
```

```
                #pointcloud_obejetos[:,2],s = 10, alpha = 0.3, color="blue")


""" Se aplica kmeans para segmentar cada objeto. """

# 10 clusters
n_clusters = 3
# Runs in parallel 4 CPUs
kmeans = KMeans(n_clusters=n_clusters, n_init=20, n_jobs=4).fit(pointcloud_obejetos)
idx = kmeans.labels_

colors = ["yellow","lime", "cyan", "red", "pink"]

def eliminar_outliers(puntos):
    # Si hay outliers en los laterales los quita
    puntos = puntos[(puntos[:,0] < np.mean(puntos[:,0]) + 2*np.std(puntos[:,0]))]
    puntos = puntos[(puntos[:,0] > np.mean(puntos[:,0]) - 2*np.std(puntos[:,0]))]
    # Si hay outliers en los longitudinales los quita
    puntos = puntos[(puntos[:,1] < np.mean(puntos[:,1]) + 2*np.std(puntos[:,1]))]
    puntos = puntos[(puntos[:,1] > np.mean(puntos[:,1]) - 2*np.std(puntos[:,1]))]
    # Si hay outliers en la altura los quita
    puntos = puntos[(puntos[:,2] < np.mean(puntos[:,2]) + 2*np.std(puntos[:,2]))]
    puntos = puntos[(puntos[:,2] > np.mean(puntos[:,2]) - 2*np.std(puntos[:,2]))]

    return puntos

obj1 = eliminar_outliers(pointcloud_obejetos[idx==0])
obj2 = eliminar_outliers(pointcloud_obejetos[idx==1])
obj3 = eliminar_outliers(pointcloud_obejetos[idx==2])

ax.scatter(obj1[:,0], obj1[:,1], obj1[:,2], color = "yellow", s = 10, alpha = 0.3)
ax.scatter(obj2[:,0], obj2[:,1], obj2[:,2], color = "pink", s = 10, alpha = 0.3)
ax.scatter(obj3[:,0], obj3[:,1], obj3[:,2], color = "cyan", s = 10, alpha = 0.3)

"""Se aplica ACP para encontrar los ejes principales de cada objeto."""
def obtain_reference_frame(pcd):

    cov = np.cov(pcd.T)
    med = np.mean(pcd, axis = 0)
    w, v = LA.eig(cov)
    v[0] = v[0]*np.sqrt(w[0])
    v[1] = v[1]*np.sqrt(w[1])
    v[2] = v[2]*np.sqrt(w[2])

    soa = np.array([[med[0], med[1], med[2], v[0][0], v[0][1], v[0][2]]])
    soan = np.array([[med[0], med[1], med[2], v[1][0], v[1][1], v[1][2]]])
    soa = np.append(soa,soan, axis = 0)
    soan = np.array([[med[0], med[1], med[2], v[2][0], v[2][1], v[2][2]]])
    soa = np.append(soa,soan, axis = 0)
    return soa


acp1 = obtain_reference_frame(obj1)
Xvec, Yvec, Zvec, Uvec, Vvec, Wvec = zip(*acp1)
ax.quiver(Xvec, Yvec, Zvec, Uvec, Vvec, Wvec, color = "black")

acp2 = obtain_reference_frame(obj2)
Xvec, Yvec, Zvec, Uvec, Vvec, Wvec = zip(*acp2)
ax.quiver(Xvec, Yvec, Zvec, Uvec, Vvec, Wvec, color = "black")

acp3 = obtain_reference_frame(obj3)
Xvec, Yvec, Zvec, Uvec, Vvec, Wvec = zip(*acp3)
ax.quiver(Xvec, Yvec, Zvec, Uvec, Vvec, Wvec, color = "black")
```

```
247
248 #fig2 = plt.figure()
249 #ax2 = fig2.gca(projection='3d')
250 #ax2.scatter(obj1[:,0], obj1[:,1], obj1[:,2], color = "yellow", s = 10, alpha = 1)
```

A.2 Objeto JSON de ejemplo para el Worker en Python

```
1  {
2    "firstImage": {
3      "idColor" : "https://i.ibb.co/xj4vDyT/color-image1.png",
4      "idDepth" : "https://i.ibb.co/S6CYjM0/depth-image1.png",
5      "width" : "1280",
6      "height" : "720",
7      "ppx" : "326.79736328125",
8      "ppy" : "239.5101776123047",
9      "fx" : "614.009521484375",
10     "fy" : "614.193603515625",
11     "model" : "None",
12     "coeff" : "[0, 0, 0, 0, 0]",
13     "depthScale" : 0.001
14   },
15   "lastImage": {
16     "idColor" : "https://i.ibb.co/brJMsdT/color-image2.png",
17     "idDepth" : "https://i.ibb.co/5hfr5mR/depth-image2.png",
18     "width" : "1280",
19     "height" : "720",
20     "ppx" : "326.79736328125",
21     "ppy" : "239.5101776123047",
22     "fx" : "614.009521484375",
23     "fy" : "614.193603515625",
24     "model" : "None",
25     "coeff" : "[0, 0, 0, 0, 0]",
26     "depthScale" : 0.001
27   },
28   "points": [
29     {
30       "x": -0.65,
31       "y": -0.2,
32       "z": 0.3,
33       "rx": 0,
34       "ry": 0,
35       "rz": 0,
36       "height": 11
37     },
38     {
39       "x": -0.75,
40       "y": -0.3,
41       "z": 0.3,
42       "rx": 0,
43       "ry": 0,
```

```
44        "rz": 0,
45        "height": 0
46      },
47      {
48        "x": -0.85,
49        "y": -0.2,
50        "z": 0.3,
51        "rx": 0,
52        "ry": 0,
53        "rz": 0,
54        "height": 12
55      }
56    ]
57  }
```

Apéndice B

Herramientas y recursos

Las herramientas necesarias para la elaboración del proyecto han sido:

- PC compatible
- Sistema operativo GNU/Linux.
- Entorno de desarrollo CodeBlocks.
- Entorno de desarrollo Spyder.
- Procesador de textos LaTeX.
- Lenguaje de procesamiento matemático Matlab.
- Control de versiones Anaconda.
- Compilador C/C++ gcc.
- Interprete de Python 3.6.
- Gestor de compilaciones make.

B.1 Glosario de acrónimos.

- SURF: Speeded-Up Robust Features.
- SIFT: Scale-invariant feature transform.
- PFH: Point Features Histogram.
- FPFH: Fast Point Features Histogram.
- RANSAC: Random Sample Consensus.
- ICP: Iterative Closest Point.

Universidad de Alcalá
Escuela Politécnica Superior

ESCUELA POLITECNICA
SUPERIOR

Universidad
de Alcalá