

PRÁCTICA 2
Grado en Ingeniería Robótica

Detección de objetos usando el pipeline de reconocimiento tradicional



Autores: Zuleika María Redondo García, Cristina Romero Mirete

Asignatura: Sistemas de Percepción

Fecha: Mayo 2022

Grupo: 1

GitHub: <https://github.com/crisrm128/Sistemas-de-Percepcion>

Drive: <https://drive.google.com/drive/folders/1G7H2kGEwngky0RnjuZXGQ0zJ3V1c47Z4?usp=sharing>

Índice general

1. Introducción	3
2. Estado del Arte	4
3. Desarrollo	9
4. Experimentación	19
4.1. Cambio de valores de los parámetros	19
4.2. Resultados finales de la experimentación	40
5. Conclusiones	44
6. Referencias	46

Parte 1

Introducción

El principal objetivo de este trabajo es llevar a cabo un proceso de reconocimiento de objetos 3D. Por lo tanto, se va a tratar de etiquetar cada uno de los objetos que hayan sido percibidos mediante, en este caso, una cámara Kinect, que obtiene nubos de puntos.

Así, partiendo de una nube de puntos de una escena y las nubes de puntos específicas de cada uno de los objetos a identificar en la escena anterior, se debe localizar, a su vez, la posición de cada uno de los objetos dentro de la escena.

Tal y como se hablará de ello en apartados posteriores, el reconocimiento de objetos 3D presenta una dificultad mucho más elevada con respecto al 2D, pues en una imagen es mucho más sencillo detectar, reconocer y localizar objetos, que se puede hacer tanto con métodos tradicionales o, de forma más eficaz, mediante *Deep Learning*. Sin embargo, extrapolar esta tarea al 3D resulta mucho más complejo, pues ya se debe tener en cuenta el factor espacial, por ejemplo, al aplicar convoluciones (metodología muy empleada en 2D), pues se deberán seguir cumpliendo las relaciones de profundidad.

De esta forma, se aplicará un algoritmo o *pipeline* tradicional para reconocimiento de objetos 3D, capaz de emparejar e identificar si cierto objeto está presente en la escena.

Parte 2

Estado del Arte

En la actualidad, el problema de reconocimiento de objetos en imágenes se ha considerado completamente solucionado, de hecho, con diferentes métodos que resultan muy eficientes computacionalmente desarrollados recientemente.

Sin embargo, ha surgido un nuevo problema más adecuado a la complejidad a la que nos enfrentamos hoy en día, este es, el reconocimiento de objetos en 3D. Se ha comenzado a investigar en profundidad acerca de este problema para poder encontrar soluciones robustas y eficientes en función de los diferentes campos.

- **Métodos tradicionales:** En el *paper* titulado ‘*Scale-hierarchical 3D object recognition in cluttered scenes*’[1] se trata el tema en profundidad y se exponen varias soluciones:

- En primer lugar, Johnson y Hebert en su *paper* ***Using Spin Images for Efficient Object Recognition in Cluttered 3D Scenes***[2] presentan el ‘matching’ de superficies para obtener el reconocimiento de las imágenes.

Así pues, se obtiene la representación de las **superficies** en función del sistema de coordenadas del propio objeto y su detección por medio de las características globales. Se obtienen las **normales** de los puntos de la superficie, así como la **orientación** de cada punto.

Una vez obtenidos estos datos, se obtienen las ***spin images*** que son, en palabras generales, representaciones 2D de los puntos del objeto que se construyen a partir de la información de orientación de los puntos. Estas imágenes serán **similares si representan al mismo objeto**.

Finalmente, gracias a las *spin images* obtenidas y una librería de imágenes similares referenciadas a objetos, se realiza **matching** con el método de *computing correlation coefficient*.

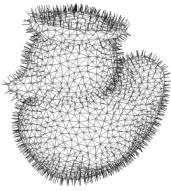


Figura 2.1: Representación de la superficie de un objeto.

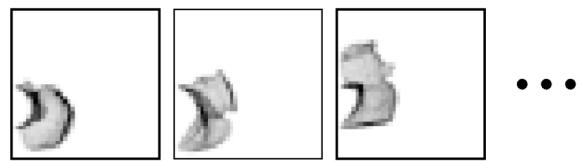


Figura 2.2: Algunas de las *spin images* obtenidas.

- Por otro lado, Stein y Medioni en su trabajo *Structural Indexing: efficient 3-D Object Recognition*[3], presentan otro método basado en la distribución de normales.

De esta manera, las propiedades de los puntos se representan por medio de “splashes”, es decir, distribuciones de normales a través de puntos de interés y “curvas en 3D” que son codificados posteriormente.

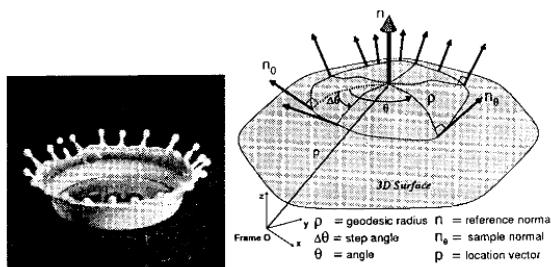


Figura 2.3: Ejemplo de *splash*.

Gracias a estos datos y otros guardados en la librería de posibles objetos, se establecen **correspondencias** y se hacen **hipótesis** entre las características obtenidas de la escena y los modelos almacenados.

Finalmente, se obtiene la **matriz de transformación** que produce el menor error posible en las correspondencias obtenidas de las hipótesis anteriores utilizando *least squares method*.

- **Métodos basados en deep learning:** En el *paper* titulado ‘*Review of multi-view 3D object recognition methods based on deep learning*’[4] se expone la existencia de 3 vertientes de este tipo de soluciones:

- **Basados en véxeles** Un ejemplo sería el expuesto en el *paper* *Sliced voxel representations with LSTM and CNN for 3D shape recognition*[5].

Sus autores proponen una representación de véxel cortado que llaman “**Sliced Square Voxels**” (**SSV**) con **LSTM** (**Long Short-Term Memory**) y una **red convolucional** (**CNN**) para el conseguir el reconocimiento de la forma de objetos en 3D.

Primeramente, la representación del véxel se refiere a atribuir información de profundidad a información en 2D, por tanto, se consigue una representación de la forma del objeto de una forma más natural, es decir, el espacio se divide en véxeles, se cortan estos mismos verticalmente a lo largo de la dirección de profundidad y se tratan como información espacial para la información del espacio 2D.

Entonces, se utiliza la red convolucional para expresar características de esta información bidimensional, lo que hace posible expresar la forma de la sección trasversal del modelo tridimensional.

Seguidamente se aplica LSTM para representar la profundidad de la figura del modelo 3D con la conexión espacial de las caras cortadas.

El resultado final son unas **características que expresan correctamente la forma tridimensional del objeto**. Estas permiten obtener porcentajes de correspondencia con objetos preprocesados.

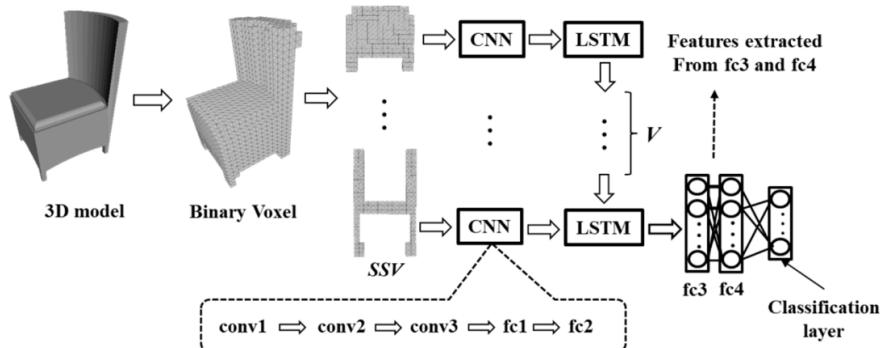


Figura 2.4: Pipeline de *Sliced Voxel representation*.

- **Basados en conjuntos de puntos**

En el trabajo *Escape from Cells: Deep Kd-Networks for the Recognition of 3D Point Cloud Models*[6] se propone el uso de una nueva arquitectura de deep learning a la que han llamado **Kd-network**, que trabaja con nubes de puntos desestructuradas para resolver el problema.

Esta realiza varias trasformaciones y devuelve características de estas transformaciones en función de las divisiones que se hacen del espacio de puntos con el **método de kd-trees**.

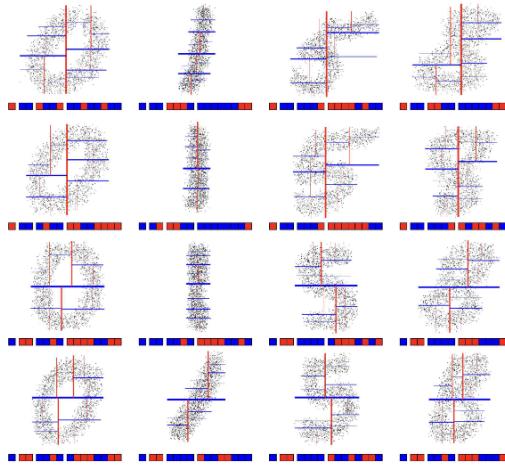


Figura 2.5: Ejemplo de distintas divisiones del espacio mediante **kd-tree** en varias nubes de puntos.

No utiliza ningún tipo de escalado por medio de véxels para evitar malos comportamientos de la red debidos a este mismo escalado.

- **Basados en vistas**

Este método se ejemplifica correctamente en el *paper* titulado *RotationNet: Joint Object Categorization and Pose Estimation Using Multiviews from Unsupervised Viewpoints*[7].

Sus tres autores exponen el uso de una red convolucional basado en el modelo de “*RotationNet*”, con el que se introducen imágenes de un mismo objeto desde diferentes puntos de vista (rotaciones) y trata de obtener de forma correcta su posición y categoría.

Este método trata las etiquetas de los puntos de vista como variables latentes, que son aprendidas de manera no supervisada durante el entrenamiento usando un dataset de objetos.

RotationNet está diseñado para ser usado solamente con un número parcial de imágenes de varias vistas y esta propiedad es muy práctica en escenarios con oclusión.

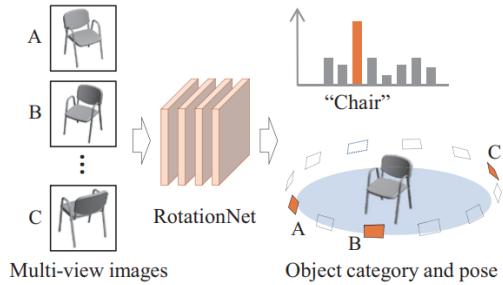


Figura 2.6: Pasos en la aplicación de CNN basada en RotationNet

Parte 3

Desarrollo

El *pipeline* implementado para reconocimiento de objetos 3D sigue los siguientes pasos para llevar a cabo esta tarea:

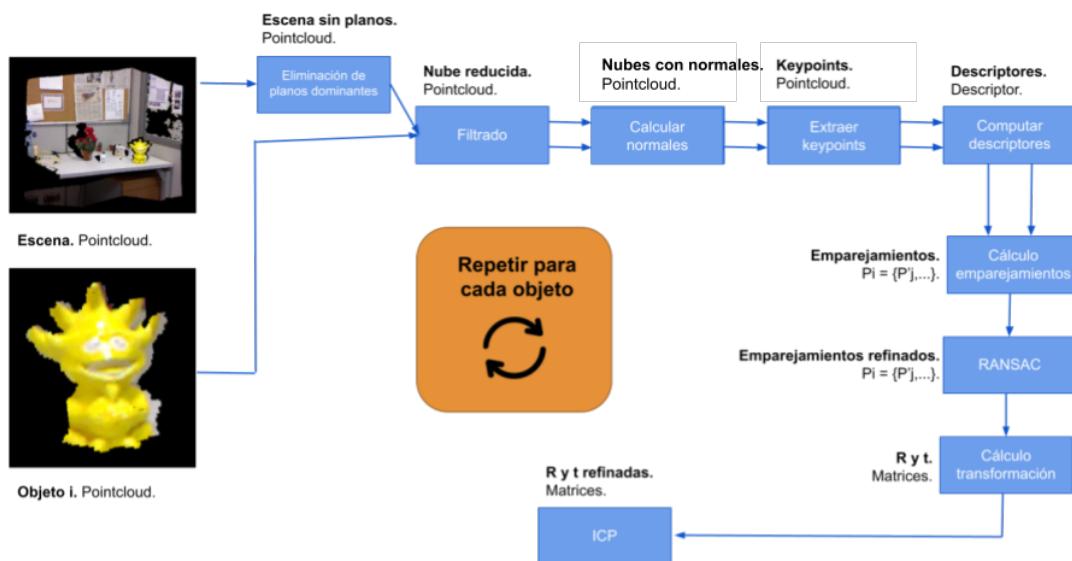


Figura 3.1: Algoritmo tradicional implementado para el reconocimiento de objetos 3D.

A continuación se detallarán cada uno de los elementos de este *pipeline* y cómo se han implementado en el código:

1. **Eliminación de planos dominantes:** Es importante eliminar planos de la escena que no interesan y pueden resultar conflictivos en tareas como cálculo de las normales o *keypoints*,

falseando información que resulta ser irrelevante.

Un ejemplo de este tipo de planos son las paredes y, en el caso de esta escena en concreto, la superficie del escritorio donde reposan los objetos. De esta manera, se plantea un algoritmo que, de forma iterativa, va eliminando un plano cada vez.

Para ello, se ha utilizado la función `segment_plane` de la librería Open3D, que como su propio nombre indica permite la segmentación de los planos principales de la escena y su posterior eliminación. La función se basa en utilizar RANSAC, un algoritmo capaz de estimar, iterativamente, los parámetros de un modelo de forma robusta. En este caso el modelo se trata de los planos a eliminar en cada caso.

Esta función cuenta con tres parámetros de entrada: `distance_threshold`, que es la máxima distancia que un punto de la nube debe tener con respecto al plano estimado para ser considerado un *inlier*, es decir, el umbral de distancia para saber si ese dato encaja con el modelo; `ransac_n`, que define el número de puntos que son aleatoriamente seleccionados para estimar el plano; y `num_iterations`, que se corresponde con la frecuencia con la que un plano aleatorio es muestreado y verificado.

Esta segmentación de planos se ha realizado 3 veces, la primera de ellas para la pared trasera, la segunda para la pared lateral derecha y la tercera para la superficie del escritorio. Los umbrales de distancia de las dos paredes han tenido que aumentarse del valor por defecto de 0.01 a 0.05 metros del plano original obtenido, pues si no no se consigue eliminar por completo (véase figura 3.2). El resto de parámetros se ha dejado con los valores por defecto: `ransac_n` son 3 puntos, los mínimos necesarios para definir un plano, y `num_iterations` igual a 1000, que resultan suficientes para poder encontrar el plano correcto a eliminar en cada caso.

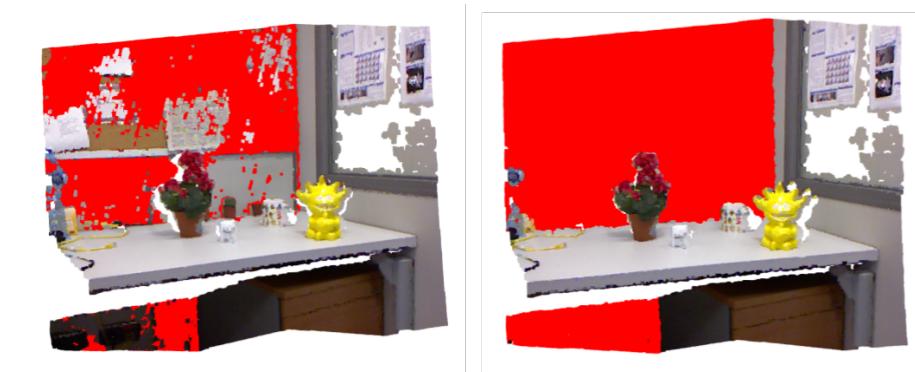


Figura 3.2: Diferencia al utilizar un umbral de distancia de 0.01 (izquierda) a 0.05 (derecha).

Así, para poder eliminar el plano basta con seleccionar los puntos invertidos, es decir, los *outliers*. Para ello se utiliza la función `select_by_index` y así invertir los *inliers*. De manera progresiva se irán eliminando los *inliers* (como se van eliminando desde el plano más grande, se eliminarán siempre en cierto orden de mayor a menor), obteniendo el siguiente resultado:

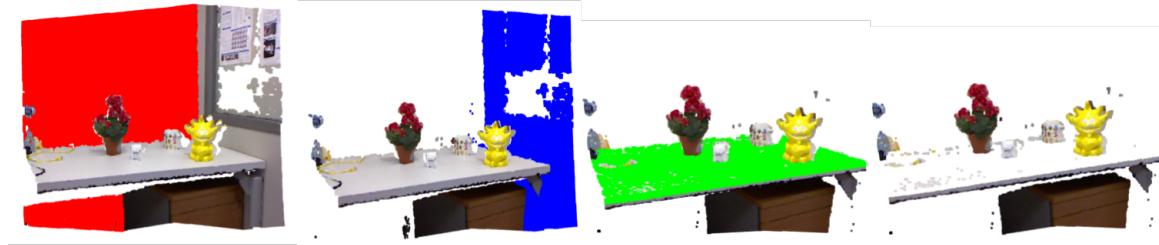


Figura 3.3: Evolución de las iteraciones de eliminación de planos (izquierda a derecha) y resultado final.

2. **Filtrado de la nube de puntos:** El principal objetivo de este paso es reducir la dimensionalidad de las nubes, pues el número de puntos de cada una de ellas puede resultar elevado y ralentizar otros posteriores procesos de cálculo. Para ello, existen dos importantes algoritmos de filtrado: *Voxelgrid* y *Uniform Sampling*, ambos utilizados para hacer esta compresión de datos.

En el caso de la librería Open3D, se encuentra definido el primer método bajo la función `voxel_down_sample`, que se basa en establecer un tamaño de celdas (*boxes*) en las cuales caerán una serie de puntos de la nube dentro. Con todos estos puntos en el interior de la celda, se hará la media y ese valor obtenido sera el punto representativo de esa celda que formará parte de esa nueva nube de puntos reducida. Por lo tanto, el único parámetro a configurar en esta función es el tamaño de esa celda, que se denomina `voxel_size`.

En Open3D también existe una función denominada `uniform_down_sample`, que con el nombre se puede llegar a intuir que se trata del otro algoritmo muy utilizado, *Uniform Sampling*, cuando no es así. *Uniform Sampling* se basa en el mismo principio que *Voxelgrid*, pero en vez de calcular la media de todos los puntos selecciona directamente el punto central, lo cual puede llegar a desvirtuar la nube de puntos en determinados casos.

Con respecto a `uniform_down_sample`, funciona igual que `voxel_down_sample` pero la celda se irá obteniendo a partir de la lista de puntos, es decir, cada ‘k’ puntos (parámetro de entrada en vez de `voxel_size`) cogerá el punto central de estos. Así, esta opción queda totalmente rechazada porque sus resultados son mucho peores (con más ruido) que utilizando la función `voxel_down_sample`, pues el resultado es mucho más fiel a la nube de puntos original.



Figura 3.4: Diferencia entre filtrado con `uniform_down_sample` cada 5 puntos (arriba) y `voxel_down_sample` con tamaño de celda de 2 milímetros (abajo).

Este filtrado hay que hacerlo para las dos nubes, la escena y los puntos, con el mismo parámetro de tamaño de celda. Este valor debe ser un radio lo más pequeño posible para no falsear las normales (por eso en la figura 3.4 apenas se nota la diferencia con la nube de puntos original), así dará las normales muy fiables al tratarse de una zona muy local, por lo que se trata de la mejor resolución disponible.



Figura 3.5: Diferencia entre `voxel_size` de 2 (arriba) y 10 milímetros (abajo).

Se puede observar en la figura 3.5 cómo afecta a las normales aumentar demasiado el tamaño de la celda, y se ve claramente que las normales del borde de la mesa en la imagen de abajo no se corresponden con la dirección real del plano (horizontal), mientras que en la imagen de arriba el resultado es mucho más fiable.

Cabe destacar que las normales visualizadas no han sido extraídas mediante el código, sino que forman parte del visualizador de Open3D pulsando la tecla N.

Por último, es necesario destacar que el tamaño de la celda utilizado para el objeto PLC es menor que el resto, pasando de 2 milímetros a 1.45 milímetros. Esto es debido a que, al tratarse de un objeto mucho más pequeño que el resto, va a contar menos con puntos que el resto. Esto significa que sucederá el mismo efecto que aumentando el tamaño, se falsearán las normales y el resultado obtenido no será representativo de la realidad.

3. **Cálculo de las normales:** Para esta tarea se crea una función propia llamada '*estimate_normals*' a la que se pasa la *pcd* y el tamaño del *voxel*.

Para calcular las normales de cada uno de los puntos de la *pcd* se utiliza la función propia de *open3d*, *estimate_normals*. Esta función utiliza un valor de radio que se le pasa para calcular la orientación de las normales de los puntos con la información aportada por los puntos dentro de este radio. El número de puntos no podrá pasar de un valor indicado, que será el máximo número de vecinos más cercanos, pues la función implementada en Open3D utiliza la estructura de KDTree para buscar los ‘n’ vecinos más cercanos al punto en cuestión.

La función propia devuelve la “*pcd*” con las normales ya calculadas y el tiempo que tarda en realizar la tarea.

Por otro lado, cabe destacar que el valor de el radio introducido es calculado mediante la multiplicación del tamaño del *voxel* por un escalar. Este último ha sido ajustado por medio de la práctica, teniendo en cuenta los siguientes factores:

- Si el valor del **radio** es **muy pequeño**, las normales serán falseadas por falta de información.
- Si el valor del **radio** es **muy grande**, se tendrá en cuenta parte de superficie muy lejana al punto y poco representativa, por tanto, el resultado de la normal no será el correcto(tiendiendo en cuenta el consecuente aumento de puntos a coger).

4. **Cálculo de los puntos clave (keypoints):** Cabe destacar que, no todos los puntos son útiles para encontrar correspondencias y realizar el *matching* de las nubes de puntos de la escena y del objeto en cuestión. Para ello, es necesario trabajar con aquellos puntos que sean característicos tanto de la escena, como del objeto, de manera que se buscarán los *keypoints* que coincidan en ambas nubes de puntos para hacer los emparejamientos.

Asimismo, es necesario que estos *keypoints* tengan las características de ser **repetibles**, de forma que puedan ser encontrados inequívocamente desde cualquier punto de vista de la nube de puntos, así como **distinguibles** pues, es necesario identificarlos de manera única (este paso se realizará en el punto del *pipeline*, 5).

El extractor de puntos característicos recibirá una nube de puntos y devolverá un subgrupo de esta misma nube, por lo que permite obtener una representación mucho más identificativa y consistente tanto de la escena como de cada uno de los objetos de ésta. Así, el algoritmo empleado de la librería Open3D es el denominado ISS, *Intrinsic Shape Signatures*. Este método se basa en la descomposición en valores singulares, es decir, autovalores y autovectores, que se suelen usar para reducir dimensionalidad, pero en este caso se utilizan para medir la dispersión entre ejes de coordenadas, pues las direcciones principales indicarán las direcciones donde hay más dispersión en los datos.

Para ello, el algoritmo empieza con la iteración de todos los puntos de la nube y para cada uno se debe calcular un peso, que será inversamente proporcional a la densidad de puntos dentro de la vecindad del punto seleccionado en esa iteración.

A continuación, se calcula una matriz de dispersión (que es una aproximación de la matriz de covarianza para que los cálculos sean menos costosos, aunque el resultado es prácticamente el mismo) y sobre ella se realiza la descomposición en valores singulares. El resultado serán los autovalores ordenados de mayor a menor (los de mayor magnitud suelen corresponderse con los ejes ‘x’ e ‘y’) y se establecerán dos parámetros *gamma* para controlar el *ratio* o relación entre estos autovalores, pues si ambos *ratios* están por debajo de esos valores, significa que los autovalores son diferentes, lo que lleva a la conclusión de que ese punto es interesante y se encuentra en una zona no monótona, no plana.

Así, se puede decir que en el algoritmo *ISS* se detectan aquellos puntos cuyas magnitudes entre las direcciones de los sistemas de referencia de los puntos sufran mucha diferencia. Open3D lo hace así porque luego los elementos son más robustos. La idea en la que se basa este algoritmo es la de eliminar aquellos puntos que presentan una distribución similar de los puntos en los ejes de la vecindad y quedarse con el más representativo de esa zona.

Este método recibe una serie de valores [11]:

- **salient_radius:** Radio esférico de selección de vecindad para cada uno de los puntos.
- **nom_max_radius:** Radio en el que se escoge el punto con el mayor salient_value y, se descartan el resto (técnica para escoger máximos locales), es decir, se comprueba para cada punto la vecindad y prevalece el que sea más interesante, para evitar que haya muchos *keypoints* en el mismo espacio 3D, es decir, puntos clave redundantes que no aportan más información. Este parámetro será calculado por medio de la distribución más pequeña, la del eje ‘z’.
- **gamma_21:** Relación o *ratio* entre el segundo y el primer autovalor, es decir, en este caso las distribuciones de puntos en el eje ‘y’ y ‘x’.
- **gamma_32:** Relación o *ratio* entre el tercero y el segundo autovalor, es decir, en este caso las distribuciones de puntos en el eje ‘z’ e ‘y’.

5. **Cálculo de los descriptores (FPFH):** A partir de los *keypoints* obtenidos, tal y como se explicó anteriormente, es necesario identificarlos de forma única. Para ello, se calcula un descriptor de cada punto y la característica que permitirá contrastar los puntos vecinos entre sí, será la normal ya calculada.

Así pues, el algoritmo recibe la nube de puntos de los *keypoints* y un número de puntos más cercanos contenidos en un radio, cuyo valor fue obtenido por medio de la práctica.

Una vez conseguido ese número de puntos más cercanos por medio de KDTree, el método ISS se basa en la obtención de las características geométricas locales de esos puntos por medio de sus normales para conseguir el descriptor de cada uno de los puntos. [13]

A diferencia de lo que se realiza con *Point Feature Histogram (PFH)*, que capta la información de la geometría del punto analizando la diferencia entre las direcciones de las normales en la vecindad, emparejando todos los puntos dentro de esa región entre sí, FPFH considera solamente las conexiones directas entre el *keypoint* y sus vecinos, descartando las conexiones adicionales entre vecinos.

Así, se calculan la diferencia entre las normales de los dos puntos seleccionados y se codifican como tres variables angulares, que junto con la distancia euclídea de los puntos serán los valores característicos almacenados en el histograma.

De esta manera, lo que se obtiene es un vector (que es un histograma) de 33 valores para describir de forma local ese punto. Cabe destacar que son 33 valores y no 4 porque se crean ‘d’ número de histogramas de características separados, uno para cada dimensión de característica, y se concatenan. [19]

Este proceso se realizará, tal y como se ha ido comentando, tanto para los *keypoints* de la escena como para los del objeto. Asimismo, hay el mismo número de descriptores que de puntos clave, porque se supone que con este paso se describen esos puntos.

6. **Cálculo de emparejamientos y transformación:** Para realizar el *matching*, eliminación de falsos emparejamientos y cálculo de rotación y traslación (transformación del objeto para colocarlo en la escena), se utilizará RANSAC en el *pipeline* que maneja Open3D.

RANSAC permite, de manera iterativa, estimar los parámetros de un modelo de manera robusta, que en este caso será una matriz de transformación que mejor se ajustará a los emparejamientos de los puntos.

El método recibe varios valores [13]:

- Nubes de puntos de *keypoints* tanto de la fuente como del objetivo.
- Valores de descriptores de los keypoints tanto de la fuente como del objetivo.
- **mutual_filter:** Valor booleano que indica que se puede realizar *matching* con la propia nube.
- **max_correspondence_distance:** Valor de la distancia máxima a la que se pueden encontrar las correspondencias. Se eliminarán aquellos emparejamientos que superen este umbral de distancia.
- **estimation_method:** Corresponde al método de estimación de la transformación, en este caso, se utiliza el método punto a punto.
- **ransac_n:** Nº de puntos elegidos al azar de la nube de puntos de la fuente para hacer la correspondencia con los puntos del objetivo.
- **checkers:** Métodos para comprobar si dos nubes de puntos pueden ser alineadas. Se utiliza uno basado en la distancia para no alinear dos puntos a más distancia de la max_correspondence_distance. Y también se utiliza el basado en la distancia de los ejes, para comprobar que existe una similitud en cualquiera de las distancias entre 2 puntos, en función del valor de *similarity_threshold* especificado (siempre menor que 1). [14]
- **criteria:** Criterio de convergencia del método. Define el número de iteraciones a realizar por RANSAC y el valor de probabilidad de éxito deseada. Este último se utiliza para estimar la terminación anticipada del método, sin embargo, se puede utilizar el valor de 1.0 para evitar terminar anticipadamente.

Cuando mayores sean estos valores, con mayor precisión contará el método pero, más lenta será su ejecución. [15]

En palabras generales, el algoritmo escoge al azar, en este caso, 3 puntos de la nube fuente para hacer las correspondencias con los puntos de la nube objetivo y así, ajustar en cada iteración el alineamiento entre las dos nubes de puntos.

Para ello, tiene en cuenta los descriptores basados en las normales de los puntos clave y en que la distancia entre ellos no sea mayor que un valor máximo especificado. Tras varias iteraciones, el método convergerá al llegar al máximo número de estas indicado por el programador en el atributo de *criteria* o al llegar de forma anticipada a la estimación de éxito indicada.

El resultado que ofrece este método son 3 valores [17]:

- **fitness:** Área de solapamiento entre los puntos de la nube fuente y las correspondencias estimadas en la nube objetivo (el objeto y la escena).
- **inlier_rmse:** Error de las correspondencias basado en el método de estimación RMSE (root-mean-square error). Calculado entre la posición predicha y la posición real. [18]
- **correspondence_set:** Valores de correspondencia entre la nube de la escena y el objeto ($n \times 2$, siendo n el nº de correspondencias).

7. **Refinamiento de la transformación:** Sabiendo que la transformación se ha obtenido del algoritmo RANSAC descrito en el apartado anterior, a veces es necesario refinarla porque, a pesar de haber obtenido una buena traslación y rotación, puede no haber sido la óptima.

En el caso que nos concierne, como los objetos han sido extraídos directamente de la escena, los emparejamientos y transformaciones resultarán los idóneos, pero normalmente las nubes de puntos de la escena y los objetos vendrán dadas por sensores o cámaras distintas, así que no es posible asegurar que sean exactamente los mismos puntos en ambas nubes.

Así, el principal algoritmo usado para realizar este refinamiento es *ICP*, *Iterative Closest Points*, que como se ha indicado permite **refinar** una transformación ya obtenida a partir de las correspondencias, únicamente intenta mejorarla pero ya se debe partir de una estimación buena, no puede mejorar la transformación si cuenta con demasiado error acumulado.

Las entradas del algoritmo son las siguientes [16]:

- **Nube de puntos** del objeto (que serán sus *keypoints*).
- **Nube de puntos** de la escena (que serán sus *keypoints*).
- **Transformación inicial** calculada previamente por RANSAC.
- **Criterio de parada**, que en este caso será minimizando la función establecida según si se utiliza una métrica de punto-a-punto o punto-a-plano, un umbral de distancia que si el resultado obtenido es menor significa que ya se ha conseguido mejorar la transformación.

PARTE 3. DESARROLLO

Así, el algoritmo se basa en aplicar la transformación actual a los puntos de la escena, y para cada uno de ellos, emparejarlos con el punto más cercano de la nube del objeto, para estimar la transformación a partir de los emparejamientos mediante un método punto-a-punto o punto-a-plano, de manera que si el error es menor que un umbral de distancia (criterio de parada), significa que ya ha mejorado la transformación (pues lo que se hace es eliminar los pares de puntos cuya distancia sea mayor a un umbral) y se almacena este resultado.

Con respecto a las métricas de distancia ya mencionadas, el error **punto-a-punto** se refiere a calcular la distancia entre puntos más cercanos, mientras que el error **punto-a-plano** significa usar la distancia entre el punto origen y el plano descrito por el punto y su superficie normal local. La principal diferencia entre ambas métricas es que punto-a-plano suele converger antes, pero la diferencia de tiempo es ínfima y la diferencia en el resultado, en este caso en concreto, es despreciable.

Parte 4

Experimentación

4.1. Cambio de valores de los parámetros

Dentro de cada uno de los apartados que componen el *pipeline* utilizado para el reconocimiento de objetos 3D, se han obtenido una serie de resultados que se interpretarán a continuación:

1. **Eliminación de planos:** Los parámetros a modificar en esta etapa son el umbral de distancia entre los puntos al plano principal, y el número de iteraciones para obtener el plano. Tal y como se adelantó en su apartado 1, el aumentar demasiado el umbral de distancia, menos puntos serán obtenidos de ese plano dominante y el resultado será peor (véase figura 3.2). A continuación se muestra una gráfica con la relación entre el número de puntos eliminados del plano con el umbral de distancia.

En este caso, se ha utilizado el plano del fondo para estudiar su valor.

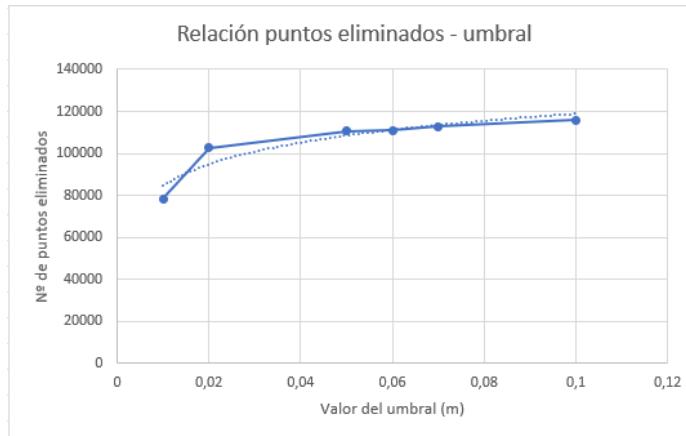


Figura 4.1: Gráfica de relación entre los puntos eliminados y el umbral elegido.

PARTE 4. EXPERIMENTACIÓN

Como se muestra en la gráfica 4.1, el crecimiento de número de puntos es similar a un crecimiento logarítmico, así pues, a más umbral, elimina más número de puntos pero, crece de esta manera porque el plano se desplaza.

Sin embargo, hay que tener en cuenta que si se aumenta demasiado el umbral, como se acaba de nombrar, el plano se desplazará y, por tanto, se comienza a dejar de detectar el plano deseado como se muestra en el figura 4.2.



Figura 4.2: Eliminación errónea del plano de la Figura 4.3: Eliminación errónea del plano de la pared de fondo.
pared de la mesa.

Asimismo, si el valor del umbral es excesivamente elevado, como se expone en la figura 4.3, también se eliminan puntos pertenecientes a los objetos, por tanto, no es lo óptimo.

De esta manera, se han elegido los valores que eliminan mayor número de puntos pertenecientes a los planos y que no eliminan parte de los objetos con los que hacer *matching*.

Por otro lado, se puede modificar el número de iteraciones para ajustar el resultado. El resultado de estos cambios se muestra en las gráficas 4.4 y 4.5, comparándolo por un lado con el número de puntos que se consiguen eliminar y por otro con el tiempo que tarda en ejecutarse. Se ha utilizado el plano de fondo para estudiarlos.

PARTE 4. EXPERIMENTACIÓN

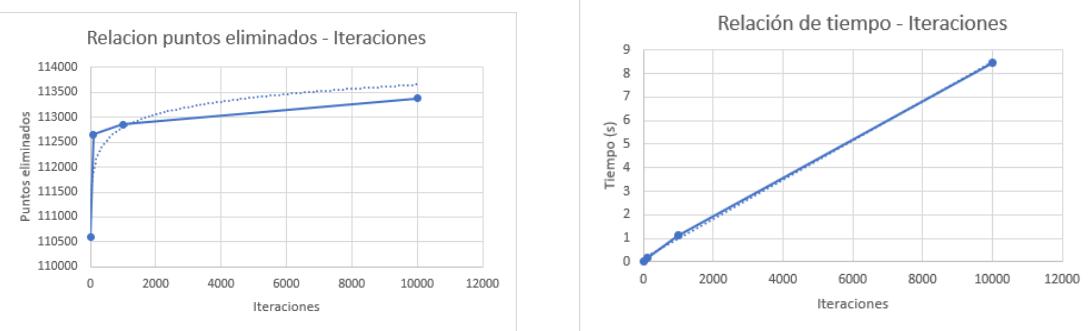


Figura 4.4: Gráfica de relación entre el nº de iteraciones y los puntos eliminados. Figura 4.5: Gráfica de relación entre el nº de iteraciones y el tiempo de ejecución.

Por un lado, el número de puntos eliminados vuelve a tener un crecimiento logarítmico, sin embargo, el aumento del tiempo de cómputo tiene un crecimiento lineal. Esto denota que el cambio de iteraciones produce un severo aumento del tiempo de ejecución, el cual, no es eficiente, dado que no elimina gran cantidad de puntos adicionales.

Es por esto, que se trata de disminuir el número de iteraciones, para que el algoritmo tarde menos en ejecutarse, aunque, se tiene que solucionar la pérdida de precisión. Para esto, se utilizará el siguiente parámetro.

Una vez ajustados los valores de umbral e iteraciones, se estudian el posible nº de puntos iniciales a establecer; para ello, se utiliza un número de iteraciones común de 100.

Se ha comprobado la variación de los valores de los puntos eliminados 4.6 y la variación del tiempo 4.7.

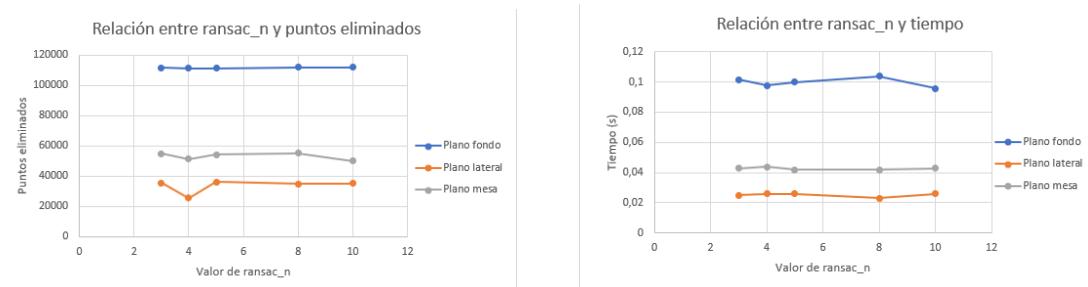


Figura 4.6: Gráfica de relación entre el nº de puntos iniciales y los puntos eliminados. Figura 4.7: Gráfica de relación entre el nº de puntos iniciales y el tiempo de ejecución.

PARTE 4. EXPERIMENTACIÓN

Se han evaluado los valores con los 3 planos a eliminar, a pesar de no mostrar gran diferencia de eliminación de puntos, en la práctica se observa una diferencia en el resultado, sobre todo al detectar la taza (objeto delicado de alinear).

Finalmente, se emplearon los valores de 100 iteraciones, 10 puntos iniciales y 0.06, 0.05 y 0.05 respectivamente de valor umbral (obtenidos por medio de validación cualitativa).

Cabe destacar, por el coste temporal, que este proceso, como se ha mostrado, se realiza 3 veces dado que, los planos a eliminar son 3: El plano de la pared de fondo, el de la pared lateral y el de la mesa. Es por ello que, los costes temporales se multiplican.

Tras estos resultados se ajustaron los valores 4.1:

	Umbral	Nº de puntos iniciales	Nº de iteraciones
a	0.05, 0.05, 0.01	3, 3, 3	1000, 1000, 1000.
b	0.06, 0.06, 0.01	3, 3, 3	100, 100, 100.
c	0.06, 0.06, 0.01	10, 10, 10	100, 100, 100.
d	0.06, 0.06, 0.01	10, 10, 10	500, 500, 500.
e	0.06, 0.06, 0.008	5, 5, 5	500, 500, 500.

Cuadro 4.1: Valores para la eliminación de planos.

- a) Al principio se dejaron los valores por defecto de puntos iniciales y nº de iteraciones y se ajustaron los umbrales para poder detectar correctamente los planos. Sin embargo, el coste computacional era muy elevado.
- b) El nº de iteraciones se disminuyó a 100. Esto agilizó el método, sin embargo, los resultados de la taza y el plc se vieron alterados.
Además, se detectó que se podía aumentar el umbral para las dos primeras paredes.
- c) Para aumentar la precisión, se aumentó el nº de puntos iniciales pero, a pesar de aumentar la precisión, no se conseguía que el error de la taza y plc disminuyeran lo suficiente.
- d) Para conseguir elevar en mayor cantidad la precisión, se aumenta el nº de iteraciones.
- e) Para ajustar todo lo posible el tiempo de ejecución, se disminuye el nº de puntos iniciales sin prácticamente perder precisión.

Por otro lado, se detectó que con el último valor de umbral, se eliminaban algunos puntos de la base de la taza, por tanto, se disminuyó ligeramente su valor.

2. **Resumen de puntos (filtrado):** Para esta práctica, se presentan 2 métodos posibles para realizar el filtrado de los puntos:

PARTE 4. EXPERIMENTACIÓN

- **`voxel_down_sample(<voxel_size>)`:** Este método divide el espacio en cuadrículas del tamaño indicado, se hace la media de los puntos dentro de estas cuadrículas y para cada uno de ellos se queda con la media.

A mayor tamaño de *voxel*, menor número de puntos se obtendrán. En caso de exceder un tamaño del *voxel* provoca que se detecten pocos puntos y, por tanto, que no se obtengan suficientes *keypoints*, como se expone en las figuras 4.8 y 4.9.



Figura 4.8: Número de puntos obtenidos tras aplicar el método de `voxel_down_sample` con un tamaño de *voxel* de 6mm.
Figura 4.9: Número de *keypoints* tras aplicar el método de `voxel_down_sample` con un tamaño de *voxel* de 6mm.

De esta forma, el valor de tamaño de *voxel* a elegir tiene que ser suficientemente grande como para no aumentar demasiado el coste computacional y suficientemente pequeño como para no falsear las normales y tener suficientes puntos de muestra para obtener características geométricas locales adecuadas a los puntos y los consecuentes *keypoints*.

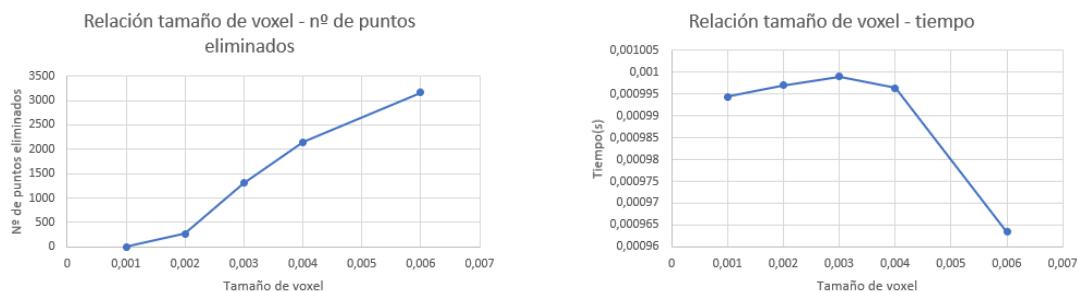


Figura 4.10: Gráfica de relación entre el tamaño de *voxel* y los puntos eliminados.
Figura 4.11: Gráfica de relación entre el tamaño de *voxel* y el tiempo.

Dado que, la variación del número de puntos eliminados es mucho más representativa que la variación del tiempo , prevalecerá a la hora de elegir el tamaño del *voxel*.

Finalmente, se eligió un tamaño de 0.002 (2 milímetros) para todos los objetos excepto para el plc, que será de 0.00145 (1.45 milímetros). Este se disminuyó ligeramente porque tiene menos puntos y, por lo tanto, sus superficies serán más pequeñas que el resto de objetos, de mayor tamaño, por lo que será más susceptible a falsear las normales. Además, también es importante para ajustar el valor del radio en el método del cálculo de las normales (3).

- **uniform_down_sample(<k points>):** Para este método, tal y como se adelantó en 2, se escoge el punto medio cada “k” puntos. No se basa en ningún otro tipo de criterio, es decir, no tiene en cuenta que ese punto corresponda a la misma superficie que el resto de puntos de su vecindad en la lista de puntos.

Es por ello que, con un número similar de puntos después del método, se consiguen menos *keypoints*, como se expone en las figuras 4.12 y 4.13.



Figura 4.13: Número de *keypoints* obtenidos con

Figura 4.12: Número de puntos obtenidos tras el los puntos que se consiguen tras aplicar el método de uniform_down_sample.

3. Cálculo de las normales:

Tal y como se mostró en la figura 3.1, el siguiente paso era calcular las normales de la nube de puntos ya filtrada. Se elige un valor de radio y un número de puntos más cercano, es decir, por medio de KDTree se obtienen los puntos más cercanos dentro de un radio para conseguir las normales de los puntos de la nube.

Se tienen que tener en cuenta los siguientes factores estudiados en función de que en un radio “r” existan “x” número de puntos:

- Si se indica en el KDTree un número de puntos menor que “x”, se obtendrá un resultado diferente.

PARTE 4. EXPERIMENTACIÓN

- Si se indican en el KDTree un número de puntos mayor que “x”, el resultado será como si se indicaran en el KDTree el número máximo de puntos dentro del radio, no más.
- Si se disminuye el radio, el número de puntos de KDTree será el máximo de puntos de ese nuevo radio, no “x”.
- Si se aumenta el radio pero no el número de puntos, no se verá variado el resultado, principalmente porque el número de puntos más cercano continúa siendo el mismo.

Como se ha comentado anteriormente en el apartado de desarrollo, se tienen que encontrar unos valores adecuados para no falsear las normales. Además, se tiene que elegir un tamaño de radio mayor que el *voxel* para abarcar más de un punto (su vecindad).

Se obtuvieron valores para los que todos los objetos muestran buenos resultados excepto la taza. Anteriormente, se disminuyó ligeramente el tamaño de *voxel* del plc para que tener más puntos y que en el cálculo de las normales se tuviera la suficiente información local geométrica para dar buenos resultados.

	Radio	max_nn
a	voxel_size*4	50
b	voxel_size*4	40
c	voxel_size*3	30

Cuadro 4.2: Valores para la obtención de las normales.

- a) No daba buenos resultados por abarcar demasiados puntos en la obtención de las normales, sobre todo en el borde de la taza, donde la información geométrica es más dispar.

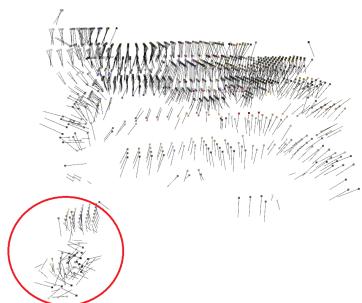


Figura 4.14: Resultado de las normales con valores a).



Figura 4.15: Resultado con valores a).

- b) Da mejores resultados en la obtención de normales, como se muestra en la figura 4.16 pero el resultado final continua sin ser el óptimo.

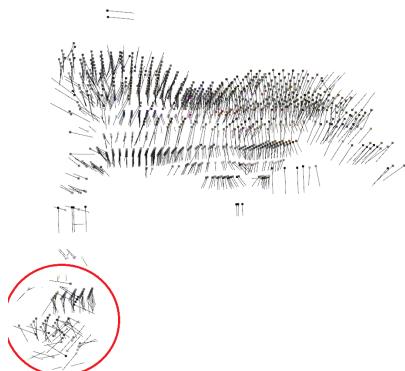


Figura 4.16: Resultado de las normales con valores b).



Figura 4.17: Resultado con valores b).

- c) Finalmente, se obtienen resultados correctos y acordes al resultado de las normales, el error obtenido en la taza es aproximadamente de $6,704 \times 10^{-5}$ metros.

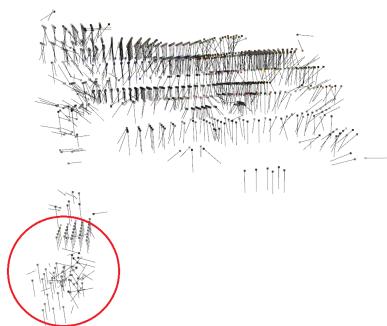


Figura 4.18: Resultado de las normales con valores c).

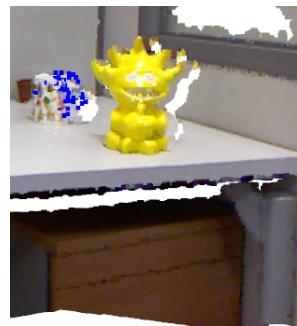


Figura 4.19: Resultado con valores c).

El tiempo en obtener las normales no varía en gran medida con los diferentes valores probados, es aproximadamente de **0.002 segundos** para todos los casos.

4. **Cálculo de puntos clave:** En este método hay 4 parámetros con posibilidad de modificación para obtener buenos resultados en la obtención de *keypoints*:

- a) **salient_radius:** Radio de vecindad esférico al punto, cuyo valor por defecto es 0.0 [21], y se ha modificado para obtener resultados satisfactorios a 0.005, lo que es equivalente a 5 milímetros.

PARTE 4. EXPERIMENTACIÓN

Como se puede observar, este valor es mayor que el tamaño del *voxel*, que posteriormente se trata de la distancia entre puntos, pues si fuera igual o menor no habría una vecindad de puntos alrededor del punto seleccionado.

A continuación, se muestra cómo la variación de este parámetro, reduciendo y aumentando el radio de vecindad, afecta en caso de la taza, pues este paso ha sido clave para la obtención correcta de su transformación:

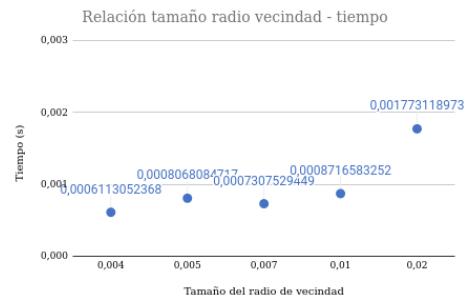
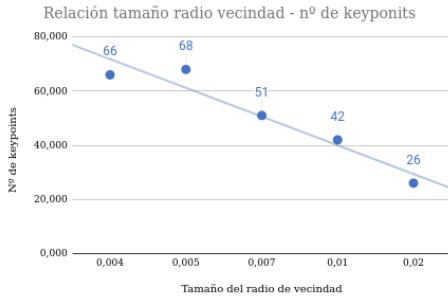


Figura 4.20: Gráfica comparativa entre el parámetro `salient_radius` y el número de *keypoints* obtenidos de la **taza**.

Figura 4.21: Gráfica comparativa entre el parámetro `salient_radius` y el tiempo usado en calcular los *keypoints* de la **taza**.

En la figura 4.20 se puede ver como el número de *keypoints* obtenidos sigue una tendencia lineal con respecto al tamaño del radio de vecindad. Conforme se va haciendo más grande, menos puntos son considerados. Esto es debido a que la superficie considerada como vecindad para un punto no es lo suficientemente representativa para describirlo, falseando su resultado como sucedía con el cálculo de las normales.

Así, comparado con el tiempo que tarda cada uno de los radios, se ha llegado a la conclusión de que el mejor compromiso precisión-rapidez viene dado por el radio **0.005**, es decir, 5 milímetros de radio de vecindad, es muy local y permite obtener el máximo número de *keypoints* en la taza, además de tener un tiempo admisible, ya que el crecimiento permanece estable y aumenta a partir de radios muy elevados.

- b) **non_max_radius**: Tal y como se explicó anteriormente, este radio sirve como umbral para eliminar aquellos *keypoints* que resultan redundantes o presentan ruido, de manera que si son demasiado débiles como para no superar ese umbral, se ven suprimidos (por eso nos quedamos con máximos locales, y suprimimos los que no son máximos).

Este valor de umbral tiene valor por defecto 0.0, pero se ha ido modificando para ver su repercusión en los resultados:

PARTE 4. EXPERIMENTACIÓN

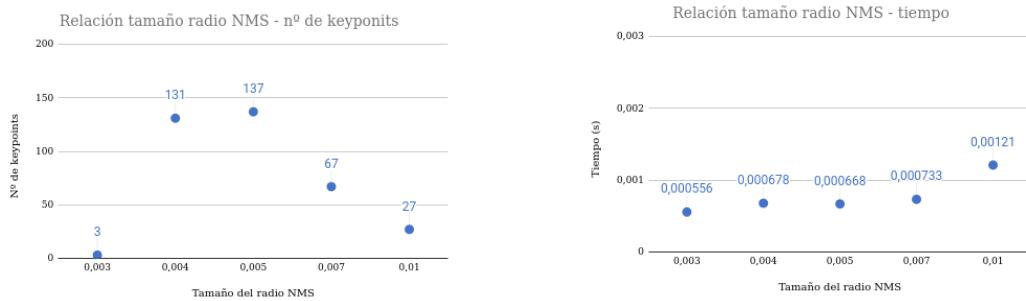


Figura 4.22: Gráfica comparativa entre el parámetro `non_max_radius` y el número de *keypoints* obtenidos de la **taza**. Figura 4.23: Gráfica comparativa entre el parámetro `non_max_radius` y el tiempo usado en calcular los *keypoints* de la **taza**.

En este caso, para el parámetro `non_max_radius` el número de *keypoints* obtenidos será mayor para tamaños pequeños de este parámetro. Esto es debido a que, cuando el radio es muy grande es susceptible a encontrar más *keypoints* irrelevantes, mientras que si el radio es muy pequeño no puede obtener suficientes *keypoints* como para discernir.

Por lo tanto, intentando obtener el mejor número de *keypoints* para todos los objetos, comprobando que se calculen en un tiempo asequible, de forma empírica se obtuvo un valor intermedio entre **0.5** y **0.7** que permitía representar, ya no sólo la taza, si no los 4 objetos de forma idónea: se fijó un valor **0.5** milímetros, con el que la diferencia de tiempos es inferior a milisegundos.

- c) **gamma_21**: Al igual que los parámetros anteriores, ya se explicó el concepto de estos ratios, que permiten ver cuál es la relación entre los autovalores calculados a partir de la matriz de covarianza de cada uno de los puntos estudiados.

Es importante destacar que, como los autovalores están ordenados de mayor a menor, el segundo nunca será mayor que el primero, como máximo será igual, y por ende el valor de ratio variará entre 0 y 1.

Si se obtiene un valor de 1 significa que ambos autovalores son iguales, y un número pequeño indica que son diferentes, que es lo realmente interesante para detectar *keypoints*, pues se quiere que haya mucha dispersión entre ejes y, de esa forma, obtener un punto con características especiales, que esté en una zona no monótona.

Por lo tanto, tanto en este ratio como en el siguiente, se tratará de obtener el compromiso de los mejores ratios, que no impacta que sean muy distintos, sino que sean parecidos y lo más pequeños posible.

PARTE 4. EXPERIMENTACIÓN

Este ratio tiene un valor por defecto de 0.975, pero como ya se ha comentado, es un número muy cercano a 1, por lo que se han realizado diversas pruebas:

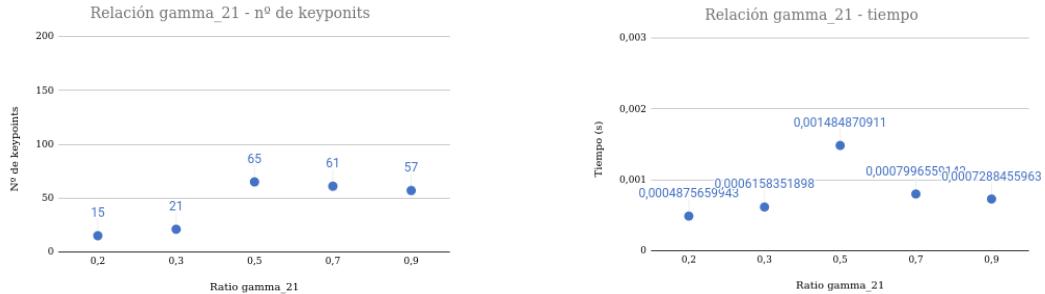


Figura 4.24: Gráfica comparativa entre el parámetro **gamma_21** y el número de *keypoints* obtenidos de la **taza**.
 Figura 4.25: Gráfica comparativa entre el parámetro **gamma_21** y el tiempo usado en calcular los *keypoints* de la **taza**.

Como se puede observar, en el caso particular de la taza, a partir de cierto valor de ratio se obtiene un mayor número de *keypoints*, por lo que ya da una idea de cómo de parecidos son esos ejes.

Sin embargo, con este valor se obtiene un tiempo de ejecución más elevado al tener un mayor número de *keypoints*, por lo que se debe buscar el compromiso para que todos los objetos obtengan un número de *keypoints* suficiente para poder describir el objeto, y no se ponga en juego un elevado aumento del coste temporal.

Por lo tanto, comprobando por medio de la experimentación qué rango de valores alrededor de **0.5** permitían obtener este compromiso se optó por utilizar un ratio de **0.45**, que permite representar todos los objetos de forma fiable y el tiempo empleado es menor que el de **0.5**.

- d) **gamma_32**: Como ya se ha explicado en el apartado anterior, este ratio permite comprobar si un punto es interesante, está fuera de una zona plana, pero en este caso comparando distintos ejes de coordenadas.

Al igual que sucedía con el ratio anterior, tiene un valor por defecto 0.975, por lo que se ha comprobado cómo afecta su reducción:

PARTE 4. EXPERIMENTACIÓN

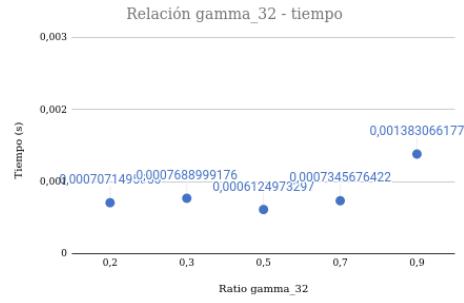
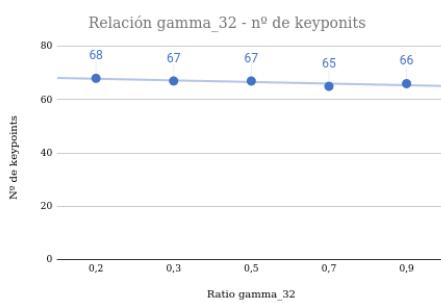


Figura 4.26: Gráfica comparativa entre el parámetro `gamma_32` y el número de *keypoints* obtenidos de la **taza**. Figura 4.27: Gráfica comparativa entre el parámetro `gamma_32` y el tiempo usado en calcular los *keypoints* de la **taza**.

En este caso, como en la gráfica 4.26 se ve que no hay demasiada variación entre el número de *keypoints* obtenidos, se puede decir que no hay demasiada dispersión entre los ejes de los autovalores 3 y 2.

Por lo tanto, la selección de este parámetro fue determinado, en mayor medida, por el tiempo de ejecución. Así pues, el valor seleccionado fue **0.5**, que permite mantener un número de *keypoints* representativo para la taza (y posteriormente también para el resto de objetos) y es el valor con el menor consumo de tiempo en la gráfica 4.27.

5. Cálculo de descriptores: El método utilizado para extracción de características en Open3D es FPFH, ya explicado en 5. Por lo tanto, los parámetros a modificar son los pertenecientes al KDTree que se realiza para obtener las características a partir de las normales calculadas que son, el radio de búsqueda `radius_feature` y el máximo de vecinos más cercanos que se pueden obtener, `max_nn`.

Por defecto, estos parámetros tienen un valor de radio de búsqueda de `voxel_size * 5` milímetros, y un máximo de 100 vecinos.

Tal y como sucedía con los *keypoints*, el valor del radio de búsqueda no debe ser igual o menor que el tamaño de `voxel_size`, porque si no no encontraría ninguna vecindad. Por lo tanto, las modificaciones del parámetro de `radius_feature` se harán como múltiplos de este valor.

A continuación, se muestran una serie de gráficas para ejemplificar cómo afecta el aumento y descenso de los dos parámetros de la función, comparándolos con el tiempo empleado para obtener los descriptores en el caso del primer objeto, la hucha, y de la escena general:

PARTE 4. EXPERIMENTACIÓN

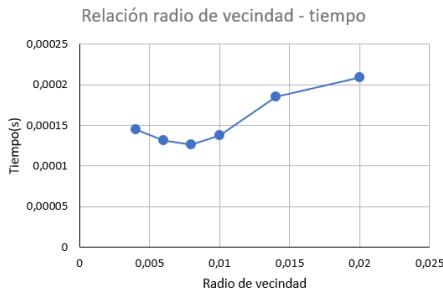


Figura 4.28: Gráfica comparativa entre el parámetro `radius_feature` y el tiempo usado en calcular los descriptores de la **hucha**.

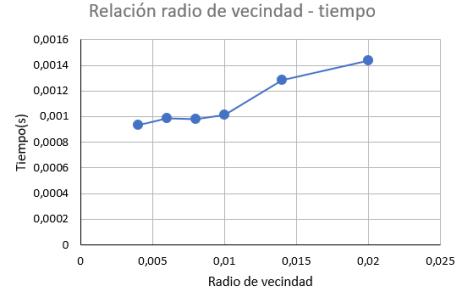


Figura 4.29: Gráfica comparativa entre el parámetro `radius_feature` y el tiempo usado en calcular los descriptores de la **escena**.

Se puede observar que ambas relaciones tienen una tendencia casi lineal, como era de esperar, pues a mayor radio de búsqueda, más puntos deberán ser considerados como vecinos y hace que el coste computacional se vea incrementado.

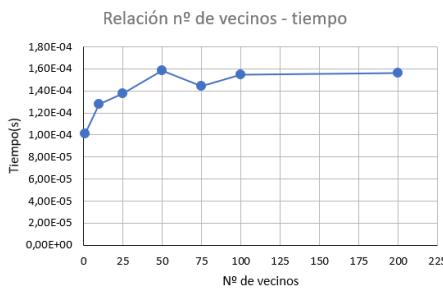


Figura 4.30: Gráfica comparativa entre el parámetro `max_nn` y el tiempo usado en calcular los descriptores de la **hucha**.

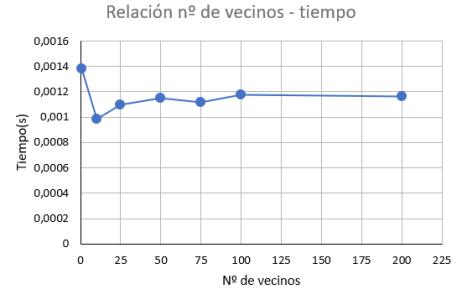


Figura 4.31: Gráfica comparativa entre el parámetro `max_nn` y el tiempo usado en calcular los descriptores de la **escena**.

En cambio, en las gráficas 4.30 y 4.31, el tiempo estimado tiende a estabilizarse a partir de cierto número de vecinos. Esto es debido a que, en ambos parámetros de experimentación, el otro valor permanecía constante. Es decir, en este caso, el parámetro `radius_feature` permanece constante con un valor de `voxel_size * 5` milímetros, mientras que se va modificando el parámetro `max_nn`.

Como se está construyendo una estructura de KDTree para encontrar los ‘n’ vecinos más cercanos, pero el radio de vecindad se mantiene constante, por mucho que se intente encontrar más vecinos, al no haber suficientes puntos dentro de ese entorno de vecindad, es imposible que incremente ese número.

Sin embargo, otra métrica que se debe tener en cuenta a la hora de confirmar si una combinación de parámetros es mejor que otra, es mediante los errores que se obtienen en cada uno de los objetos (explicación del cálculo del error en 4.2).

Como se han utilizado los tiempos para el cálculo de descriptores del objeto *piggybank* (figuras 4.28 y 4.30), se usará también su error para la interpretación de resultados, así como, de los dos objetos más conflictivos: la taza y el PLC.

Radio	Error Hucha	Error PLC	Error Taza
voxel_size*2	0,0026	0,00113	0,0019
voxel_size*3	0,00077	0,000108	0,00033
voxel_size*4	$4,46 \times 10^{-4}$	0,000107	0,00055
voxel_size*5	$5,25 \times 10^{-8}$	$6,91 \times 10^{-5}$	0,00023
voxel_size*7	$5,30 \times 10^{-8}$	$5,31 \times 10^{-16}$	0,00067
voxel_size*10	$4,17 \times 10^{-8}$	0,00542	0,0037

Cuadro 4.3: Errores para cada una de las modificaciones de `radius_feature`.

Como se puede observar, el mejor compromiso entre los tres objetos se obtiene cuando se multiplica por un factor **5** el tamaño del *voxel*, pues en el siguiente caso se consigue reducir mucho el error del PLC, pero a costa de volver a obtener una tendencia a la alta en el error de la taza. Además, es importante tener en cuenta que para cada uno de los casos se ha utilizado un valor de `max_nn` de 100, que es el valor por defecto.

Así, se fija este valor del parámetro `radius_feature` como **voxel_size*5**, para modificar, a continuación, el parámetro `max_nn`:

Nº de vecinos	Error Hucha	Error PLC	Error Taza
1	0,0051	0,00046	0,00068
10	$5,16 \times 10^{-8}$	$6,78 \times 10^{-5}$	0,00040
25	$1,88 \times 10^{-5}$	0,00050	$6,70 \times 10^{-5}$
50	$5,16 \times 10^{-8}$	$6,78 \times 10^{-5}$	$6,77 \times 10^{-5}$
75	$5,19 \times 10^{-8}$	0,00049	$6,53 \times 10^{-5}$
100	$4,06 \times 10^{-8}$	$3,58 \times 10^{-5}$	$6,77 \times 10^{-5}$
200	$5,14 \times 10^{-8}$	0,00048	0,0017

Cuadro 4.4: Errores para cada una de las modificaciones de `max_nn`.

En este caso, la mejor combinación viene dada por una cantidad máxima de vecinos es **50**, intentando alcanzar el mínimo error posible en los tres objetos.

Así pues, con los parámetros ajustados se obtiene un tiempo más que aceptable, en una escala de milisegundos.

6. **Cálculo de emparejamientos:** En este caso, se van a exponer los parámetros disponibles a modificar para obtener los mejores resultados en relación con el error obtenido para el objeto *piggybank* o taza y el tiempo empleado para su obtención.

Asimismo, es importante destacar que se compara con el error y no con el *fitness* que ya proporciona el resultado de RANSAC debido a que, este parámetro no significa el error métrico como distancia entre puntos, sino la relación de emparejamientos como resultado de calcular el número de *inliers* entre el número de puntos de la nube origen, que en este caso serán los *keypoints*.

Por otro lado, como RANSAC intenta ajustar iterativamente con diferentes muestras aleatorias puede ser que, un modelo sea más óptimo que otro pero su *fitness* sea menor porque se deja un único punto fuera a diferencia de otro que puede tener mejor *fitness* pero se ajusta peor al resultado deseado, por lo que no es una medida realmente fiable.

Una vez comentados los puntos más importantes a tener en cuenta a la hora de la evaluación del método, se prosigue a llevarla a cabo:

- a) **max_correspondence_distance:** Se corresponde con el umbral de distancia que se utilizará para considerar un *inlier*, es decir, de si se trata un buen emparejamiento o no. Por defecto, la función utiliza el parámetro `distance_threshold`, que es el parámetro `voxel_size` multiplicado por un escalar (con valor de 1.5 por defecto).

En las siguientes gráficas se muestra cómo afecta el aumento de este factor escalar:

PARTE 4. EXPERIMENTACIÓN

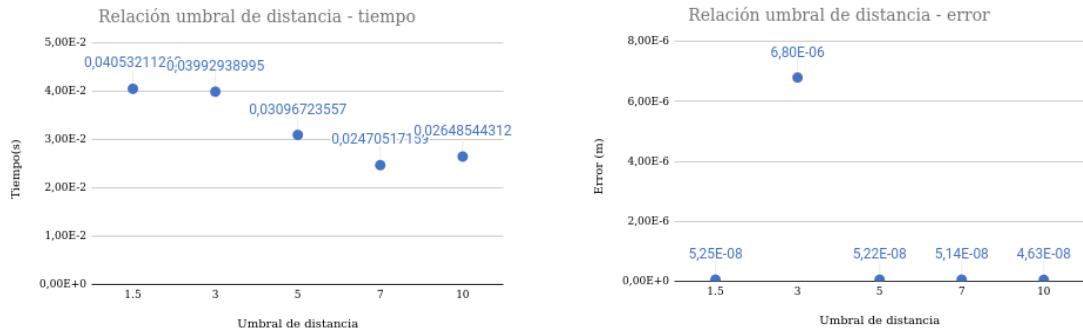


Figura 4.32: Gráfica comparativa entre el parámetro `distance_threshold` y el tiempo usado en calcular la transformación de `piggybank`. Figura 4.33: Gráfica comparativa entre el parámetro `distance_threshold` y el error obtenido al calcular la transformación de `piggybank`.

A diferencia de las gráficas obtenidas en los anteriores apartados, donde se podía observar claramente una tendencia lineal, logarítmica, etc., en este caso es más difícil visualizar dicha tendencia.

El problema surge de la ALEATORIEDAD de RANSAC, pues como se comentó en apartado 6, se van a coger cierta cantidad de puntos al azar para realizar emparejamientos (primeros *inliers*) y comprobar cuál es el modelo que más se ajusta, recalculando iterativamente este modelo de forma aleatoria.

Por tanto, la obtención de resultados puede variar entre ejecuciones del mismo código, y se puede ver claramente en el pico que se produce con un valor **3** de umbral en la gráfica 4.33.

Para solventar este inconveniente, se ha contemplado realizar la media de 3 ejecuciones del mismo código para cada uno de los datos representados para que así sea lo más representativo posible.

Una vez realizada esta aclaración, ya se puede razonar que el mejor compromiso entre error y tiempo se obtiene con el valor de umbral **1.5**, es decir, el valor por defecto.

¿Por qué este si hay valores de error similares con menor tiempo de ejecución?

Porque eso significaría que este objeto en concreto sería capaz de hacerlo perfectamente, pero perdería la capacidad de generalización para el resto.

Además, es importante recalcar que se trata del umbral admisible que deben cumplir los puntos para poder ser considerados un buen emparejamiento por lo que, a medida que se

PARTE 4. EXPERIMENTACIÓN

aumenta este umbral, significa que es más admisible y puede llegar a considerar emparejamientos erróneos como válidos en aquellos objetos con superficies más complejas como puede ser la taza.

- b) **estimation_method:** Por defecto se utiliza un método de estimación de la **transformación punto a punto**, pero también puede cambiarse para que sea punto a plano o incluso utilizar ICP.

Este parámetro **NO** se modificó porque, tal y como se explicó en el apartado de Desarrollo, el modelo punto a punto se diferencia con el punto a plano con respecto a que este último converge antes pero, la diferencia en el resultado es inapreciable.

Y, como se comenta en profundidad en el apartado 7, no tiene sentido utilizar un método como ICP porque no va a mejorar en ningún sentido la transformación.

- c) **ransac_n:** Por defecto se inicializará el modelo de RANSAC con 3 muestras de correspondencias aleatorias, pero para comprobar que este sea el mejor valor para esta práctica, se estudian varios valores:

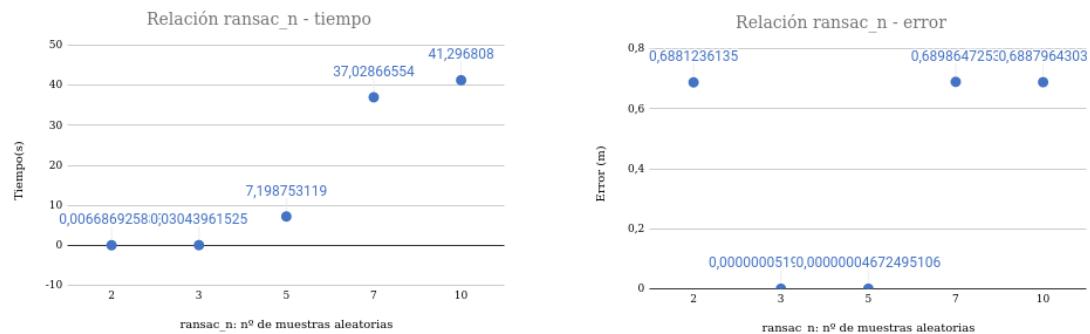


Figura 4.34: Gráfica comparativa entre el parámetro **ransac_n** y el tiempo usado en calcular la transformación de *piggybank*. Figura 4.35: Gráfica comparativa entre el parámetro **ransac_n** y el error obtenido al calcular la transformación de *piggybank*.

En este caso, sí que se puede ver cierta tendencia lineal en la gráfica 4.34, y en los errores de 4.35 se puede estimar claramente qué valor para este parámetro va a reducir indiscutiblemente los factores de error y tiempo.

De esta manera, el mejor compromiso entre precisión y rapidez se muestra en un valor de **3** muestras aleatorias como *inliers* iniciales.

PARTE 4. EXPERIMENTACIÓN

Este resultado es muy coherente ya que, la mejor forma de representar un plano es mediante 3 puntos, y se está tratando con los distintos planos o superficies del objeto, así como sus normales.

- d) **checkers**: Para comprobar que un emparejamiento es correcto, se utilizó un criterio basado en que la distancia entre los puntos no puede ser mayor a `distance_threshold`, es decir, que no se supere el umbral utilizado para considerarlo *inlier*.

Y también, se empleó la comprobación de que la similitud entre las distancias de los puntos sea la misma en ambos sentidos de emparejamiento de las nubes. Este parámetro se denomina, `similarity_threshold`, y debe tener valores inferiores a 1 al ser un porcentaje de similitud de las correspondencias en ambos sentidos de las nubes de puntos.

Dado que, ya se explicó la repercusión que tiene el parámetro `distance_threshold` en las gráficas 4.32 y 4.33 del apartado a), solo se estudia el parámetro `similarity_threshold`:

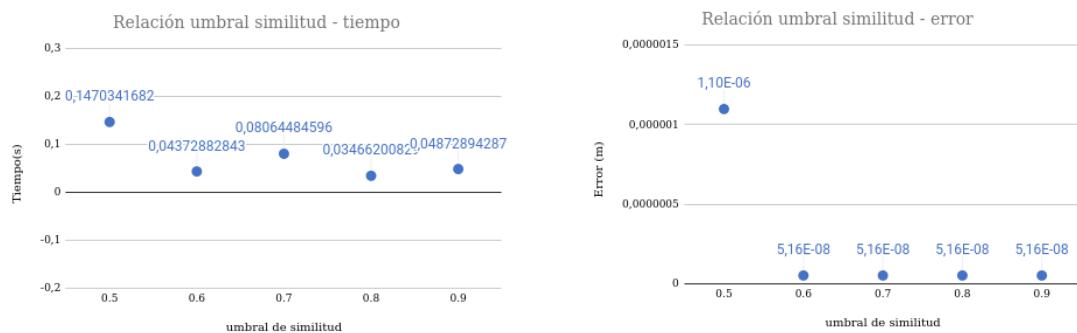


Figura 4.36: Gráfica comparativa entre el parámetro `similarity_threshold` y el tiempo usado en la ejecución de *piggybank*. Figura 4.37: Gráfica comparativa entre el parámetro `similarity_threshold` y el error obtenido al calcular la transformación de *piggybank*.

En la gráfica 4.36 se observa prácticamente una tendencia lineal, mientras que en la gráfica 4.37 cabe destacar la gran diferencia que se obtiene al aumentar más de 0.5 el umbral de similitud.

Al igual que sucedía con el umbral de distancia, aunque funcione bien para este objeto en concreto, no es buena opción intentar limitar el parámetro para obtener el mínimo tiempo de ejecución, como es el caso del valor **0.6**, pues para otros objetos puede funcionar mucho peor.

La diferencia se ve clara entre la hucha y la taza: mientras que la hucha es, en cierto modo, simétrica (por lo que las correspondencias en ambos sentidos son más susceptibles

PARTE 4. EXPERIMENTACIÓN

a ser correctas, como se ha observado en las gráficas) y de mayor volumen, mientras que la estructura de la taza es mucho más dispar, y por ello hay que agudizar el parámetro de similitud, y se dejará el valor por defecto de **0.9**, como puede observarse en las siguientes gráficas:

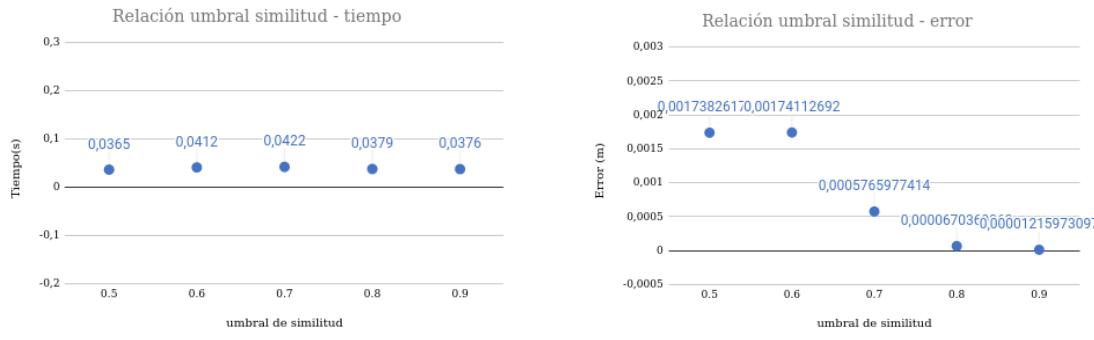


Figura 4.38: Gráfica comparativa entre el parámetro **similarity_threshold** y el tiempo usado en la transformación de la **taza**. Figura 4.39: Gráfica comparativa entre el parámetro **similarity_threshold** y el error obtenido al calcular la transformación de la **taza**.

Si se estudia el tiempo en 4.36, se puede ver que la diferencia de utilizar valores entre 0.6 y 0.9 es de apenas 5 milisegundos por lo que, no se compromete demasiado el coste temporal y esto supone una gran mejora a la hora de obtener errores pequeños para el resto de objetos.

- e) **criterio:** En este apartado se puede modificar el número máximo de iteraciones que se permite que RANSAC utilice para ajustar el modelo, y el grado de creencia para el criterio de parada; si es menor que 1, significa que podrá terminar anticipadamente sin tener que realizar todas las iteraciones, y a medida que se reduce, antes acaba.

En este caso, en vez de realizar los experimentos en base al objeto *piggybank*, se han hecho utilizando la taza, pues estos parámetros han resultado decisivos para poder obtener un resultado correcto y se ve de forma mucho más drástica su repercusión.

En primer lugar, se muestra el resultado temporal y el error tras haber modificado el número de iteraciones:

PARTE 4. EXPERIMENTACIÓN

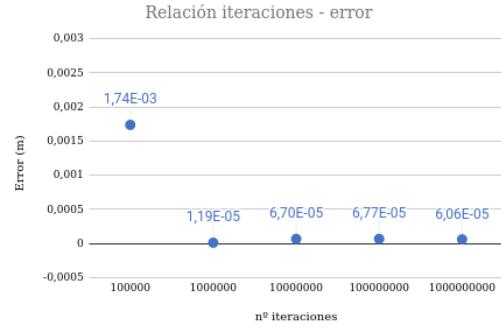
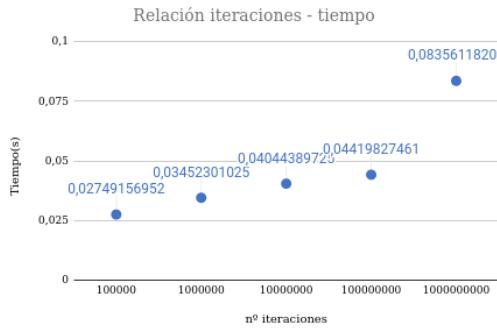


Figura 4.40: Gráfica comparativa entre el parámetro `max_iterations` y el tiempo usado en calcular la transformación de la taza.

Figura 4.41: Gráfica comparativa entre el parámetro `max_iterations` y el error obtenido al calcular la transformación de la taza.

En las gráficas 4.40 y 4.41, ya se puede ver el gran crecimiento en el tiempo que supone aumentar el número de iteraciones, y cómo el error consigue reducirse hasta estabilizarse según este parámetro aumenta. La coherencia entre ambos valores es directa: a mayor número de iteraciones que permiten probar nuevos modelos aleatorios para quedarse con el que mayor número de *inliers* tenga, significa que más probabilidades hay de encontrar el modelo más ajustado, y por ello se reduce el error.

En este caso SÍ es de gran importancia saber escoger cuál es el mejor compromiso, porque en el momento en que se aumentan demasiado las iteraciones, el tiempo se dispara. Se ha decidido utilizar el valor de 10000000 iteraciones, es decir, **10 millones de iteraciones**, pues es capaz de obtener un modelo óptimo y, por tanto, su error es muy pequeño, como se ve en la gráfica 4.41.

En un primer momento parece un número excesivo, pero gracias al siguiente parámetro a evaluar (criterio de terminación), se comprueba porqué, a pesar de eso, el tiempo que tarda el algoritmo en ejecutarse sigue siendo en tiempo real.

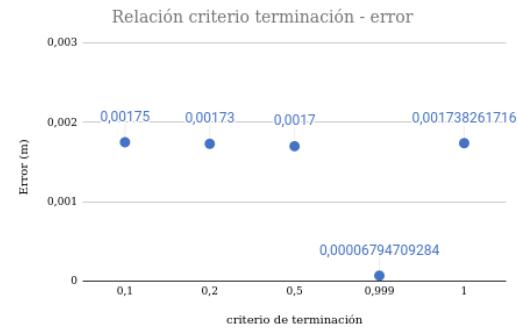
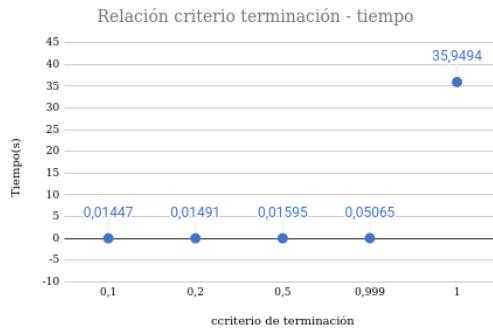


Figura 4.42: Gráfica comparativa entre el parámetro `confidence` y el tiempo usado en calcular la transformación de la taza.

Figura 4.43: Gráfica comparativa entre el parámetro `confidence` y el error obtenido al calcular la transformación de la taza.

Este parámetro es muy relevante dentro del algoritmo de RANSAC ya que, se encarga de realizar la optimización de este. Tal y como ya se había explicado en el apartado 6, es capaz de realizar una parada anticipada y quedarse con el resultado antes de realizar el máximo de iteraciones.

En caso de que este tenga un valor de 1.0, significará que no es necesario parar y realizará todas las iteraciones, por eso el tiempo se dispara en la gráfica 4.42.

Tal y como se explica en [20], mediante una fórmula matemática se puede establecer el número de iteraciones necesaria en función del parámetro *fitness* ya explicado, y el grado de creencia, que es este parámetro ajustado en las gráficas 4.42 y 4.43.

Por lo tanto, con mayor grado de confianza, el algoritmo podrá terminar con menos iteraciones, mientras que con menos confianza, la ejecución será más lenta, como se observa en 4.42.

En el caso particular de la taza, puede pensarse que un mayor grado de creencia significa que el algoritmo terminará antes, y al hacer menos iteraciones puede que no haya encontrado el modelo más ajustado. Sin embargo, al haber un número de iteraciones elevado, aumenta la probabilidad de que en una de ellas se obtenga el modelo óptimo, por lo que se utiliza el valor de creencia de **0.999**, que es el valor por defecto.

7. **Refinamiento de la transformación:** Tal y como se explicó en apartado 7, dado que, las nubes de puntos de los objetos segmentados se obtuvieron directamente de la nube de puntos de la escena, en el momento en el que se calcula la transformación correcta mediante RANSAC, esta ya es lo suficientemente buena y, por tanto, utilizar el algoritmo ICP no surte ningún efecto.



Figura 4.44: Resultado de transformación de PLC usando refinamiento con ICP.



Figura 4.45: Resultado de transformación de PLC sin refinar.

A simple vista no parece haber ninguna diferencia, pero la mejor forma de comprobarlo es mediante la métrica de error de la transformación (de la cual se hablará posteriormente). Para el resultado de la figura 4.44, se obtuvo un error de $3,83 \times 10^{-5}$ metros, mientras que en la figura 4.45 el error fue de $5,47 \times 10^{-5}$ metros.

Así, al tratarse de números tan pequeños, la diferencia entre ellos es despreciable, por lo tanto, utilizar este paso en el *pipeline* no supone una gran mejora.

4.2. Resultados finales de la experimentación

Tras estos experimentos, se pasa a analizar el resultado final y comprobar mediante cierta métrica el error que se comete en el reconocimiento de objetos, así como el tiempo necesario para ejecutarlo.

Así pues, una vez obtenida la transformación final, se calcula el error. Para ello, se siguen los siguientes pasos:

1. Se transforma la escena original en un árbol, con la función:
`tree = o3d.geometry.KDTreeFlann(<nube_de_puntos>).`
2. Seguidamente, se obtienen los puntos más cercanos a la escena de cada uno de los puntos de los objetos, con la función: `[k, idx, d] = tree.search_knn_vector_3d(<punto>, 1).`

k es el número de puntos más cercanos que se buscan, *idx*, son los índices de los puntos y *d* es la distancia al punto.

De esta manera, se hace la media con la distancia de todos los puntos y se muestran los resultados expuestos en la tabla 4.5.

Objeto	Error
Hucha	$5,11 \times 10^{-8}$
Planta	$3,71 \times 10^{-8}$
Taza	$6,70 \times 10^{-5}$
PLC	$5,31 \times 10^{-16}$

Cuadro 4.5: Errores finales de la alineación de los objetos en la escena.

Además, el tiempo de ejecución total, es decir, el tiempo transcurrido tras realizar todos los métodos anteriormente mencionados para los objetos y la escena, es de 1.843 segundos.

Por otro lado, cabe mencionar que, algunos resultados de tiempo y error, sobre todo para el objeto de la taza, pueden cambiar debido a la aleatoriedad de RANSAC, de la que ya se ha hablado.



Figura 4.46: Resultado de la hucha.

PARTE 4. EXPERIMENTACIÓN



Figura 4.47: Resultado de la planta.



Figura 4.48: Resultado de la taza.

PARTE 4. EXPERIMENTACIÓN



Figura 4.49: Resultado del plc.

Finalmente, con los resultados de los errores, el tiempo de ejecución y la valoración visual a partir de las figuras 4.46, 4.47, 4.48 y 4.49, se ha llegado a la conclusión de que son unos resultados muy satisfactorios.

Parte 5

Conclusiones

Durante todo el proceso de este ejercicio práctico se ha conseguido entender cómo implementar un *pipeline* tradicional para reconocimiento de objetos 3D, entendiendo cada uno de los pasos que lo conforman, así como las posibles dificultades que pueden suceder en cada uno de ellos.

Se ha observado su eficacia en la identificación de objetos de una escena proporcionada, sin embargo, es importante destacar una serie de desventajas que se deben tener en cuenta a la hora de escoger este algoritmo.

Uno de los principales problemas es que es **muy costoso computacionalmente**, pues el propio *pipeline* ya integra algoritmos que para obtener un buen resultado necesitan una cantidad elevada de iteraciones, lo cual supone un gran retardo en la ejecución, así como el tener que repetir todo el proceso para cada uno de los objetos (y utilizar la escena, que es la que más puntos contendrá, en cada uno de los objetos para, por ejemplo, realizar los emparejamientos).

De igual forma, una gran desventaja es la **escasez de generalización**, debido a que no sirve para reconocer cualquier tipo de objeto, debe estar presente tanto en la nube de puntos de la escena como del objeto segmentado (por ejemplo, no sirve para detectar cualquier tipo de hucha). Esto es debido a que muchos parámetros se deben ajustar para ese tipo de objeto en concreto.

Además, algo que suele suceder con gran frecuencia es que se necesita segmentar objetos de forma automática en el mundo real, pero uno de los principales problemas es que las nubes de puntos captadas suelen venir de distintos sensores y/o cámaras, por lo que son ligeramente diferentes (por ejemplo, habiendo más objetos alrededor del que se quiere reconocer y con los que se pueda confundir, o que los puntos no concuerden perfectamente) y esto conlleva errores del propio *hardware*.

PARTE 5. CONCLUSIONES

La forma más extendida hoy en día de solucionar todos estos problemas es pasando a utilizar algoritmos de **Deep Learning**, como los ya comentados en el apartado del estado del arte, y que incrementan notablemente el número de aciertos, pero estudiar este tipo de algoritmos tradicionales es esencial para entender el funcionamiento y comportamiento a bajo nivel y, posteriormente, poder extrapolarlo a las redes neuronales.

Parte 6

Referencias

- [1] **Scale-hierarchical 3D object recognition in cluttered scenes**, Prabin Bariya, Ko Nishino, IEEE:
https://ieeexplore.ieee.org/abstract/document/5539774?casa_token=zUbdJphljVgAA4AAA:xKWXWd5IXtTvFAB6nhL8x022Ef56cNrk_TN7vVOZ8KilhzpKFZqQW0_iUSwwy2WxPdYpSIxpRC
- [2] **Using Spin Images for Efficient Object Recognition in Cluttered 3D Scenes**, Andrew E. Johnson y Martial Hebert, Members, IEEE:
<https://www.cs.jhu.edu/~misha/Papers/Johnson99.pdf>
- [3] **Structural Indexing: Efficient 3-D Object Recognition**, Fridtjof Stein y Gérard Medioni, Members, IEEE:
<https://graphics.stanford.edu/~smr/ICP/comparison/stein-medioni-reg-pami92.pdf>
- [4] **Review of multi-view 3D object recognition methods based on deep learning**, Shaohua Qi, Xin Ning, Guowei Yang, Liping Zhang, Peng Long, Weiwei Cai, Weijun Li, Version of Record 18 July 2021:
https://www.sciencedirect.com/science/article/pii/S0141938221000639?casa_token=xdoj4yUM5gcAAAAA:38wIabKjJFTfPHMTtq0JZ8PX1mHrGwb13NcMquuaetUyfU2Dgfn08KMJZSFf02zb2MK1R1rZE#b0085
- [5] **Sliced voxel representations with LSTM and CNN for 3D shape recognition**, Ryo Miyagi y Masaki Aono, IEEE:
<https://ieeexplore.ieee.org/document/8282044>
- [6] **Escape from Cells: Deep Kd-Networks for the Recognition of 3D Point Cloud Models**, Roman Klokov, Victor Lempitsky:

PARTE 6. REFERENCIAS

https://openaccess.thecvf.com/content_ICCV_2017/papers/Klokov_Escape_From_Cells_ICCV_2017_paper.pdf

- [7] **RotationNet: Joint Object Categorization and Pose Estimation Using Multi-views from Unsupervised Viewpoints**, Asako Kanezaki, Yasuyuki Matsushita y Yoshifumi Nishida:

https://openaccess.thecvf.com/content_cvpr_2018/CameraReady/1282.pdf

- [8] Tutorial de Open3D para la Segmentación de Planos:

<http://www.open3d.org/docs/latest/tutorial/Basic/pointcloud.html#Plane-segmentation>

- [9] Tutorial de Open3D para el filtrado de las nubes de puntos:

<http://www.open3d.org/docs/latest/tutorial/Basic/pointcloud.html#Voxel-downsampling>

- [10] Definición y explicación del método `uniform_down_sample` de Open3D: http://www.open3d.org/docs/0.6.0/python_api/open3d.geometry.uniform_down_sample.html

- [11] Tutorial de Open3D para la extracción de puntos característicos:

http://www.open3d.org/docs/latest/tutorial/geometry/iss_keypoint_detector.html

- [12] **Intrinsic shape signatures: A shape descriptor for 3D object recognition**, Yu Zhong, 2009:

<https://ieeexplore.ieee.org/document/5457637>

- [13] Tutorial de Open3D para cálculo de descriptores, transformaciones y refinamiento de matrices de transformación: http://www.open3d.org/docs/latest/tutorial/pipelines/global_registration.html

- [14] `open3d.registration.CorrespondenceCheckerBasedOnEdgeLength`: http://www.open3d.org/docs/latest/python_api/open3d.registration.CorrespondenceCheckerBasedOnEdgeLength.html

- [15] `open3d.pipelines.registration.RANSACConvergenceCriteria`: http://www.open3d.org/docs/release/python_api/open3d.pipelines.registration.RANSACConvergenceCriteria.html

- [16] ICP Registration: http://www.open3d.org/docs/latest/tutorial/Basic/icp_registration.html

- [17] `open3d.t.pipelines.registration.RegistrationResult` http://www.open3d.org/docs/release/python_api/open3d.t.pipelines.registration.RegistrationResult.html

PARTE 6. REFERENCIAS

- [18] Root-mean-square deviation https://en.wikipedia.org/wiki/Root-mean-square_deviation
- [19] Fast Point Feature Histograms (FPFH) descriptors https://pcl.readthedocs.io/projects/tutorials/en/latest/fpfh_estimation.html
- [20] Explicación del criterio de terminación de RANSAC: http://www.open3d.org/docs/release/python_api/open3d.pipelines.registration.RANSACConvergenceCriteria.html
- [21] Valores por defecto de los parámetros del algoritmo ISS: http://www.open3d.org/docs/latest/cpp_api/namespacopen3d_1_1geometry_1_1keypoint.html