Universidad de Valladolid

ESCUELA DE INGENIERÍAS INDUSTRIALES

UNIVERSIDAD DE VALLADOLID
ESCUELA DE INGENIERIAS INDUSTRIALES
Grado en Ingeniería Electrónica Industrial y Automática

Desarrollo de un sistema de agarre de objetos en un robot mediante visión artificial

Autor:
García Portela, Alba

Tutores:
Duque Domingo, Jaime
Ingeniería de Sistemas y Automática

Gómez García-Bermejo, Jaime
Ingeniería de Sistemas y Automática

Valladolid, Junio de 2025.

Resumen

A lo largo de este trabajo se ha realizado el desarrollo de un sistema de detección de agarres de objetos para un robot colaborativo. Para ello se han realizado dos enfoques complementarios: un método tradicional basado en la reconstrucción geométrica del objeto mediante prismas y la generación de agarres basados en algoritmos matemáticos y geométricos; y un método que implementa el algoritmo de Grasp Pose Detection (GPD) para detectar agarres sobre objetos con formas y orientaciones más complejas. Para el segundo enfoque, se han implementado dos métodos de comunicación: SSH/SFTP y una arquitectura basada en un servidor web, ambas capaces de ejecutar el algoritmo GPD sobre un servidor remoto.

Palabras clave

Visión artificial, Detección de agarres, Nube de puntos, GPD, Comunicación remota.

Abstract

Throughout this work, a grasp detection system for a collaborative robot has been developed. Two complementary approaches have been implemented: a traditional method based on the geometric reconstruction of the object using prisms and the generation of grasps via mathematical and geometric algorithms; and a method that employs the Grasp Pose Detection (GPD) algorithm to detect grasps on objects with more complex shapes and orientations. For the second approach, two communication methods were implemented: SSH/SFTP and a web-server-based architecture, both capable of executing the GPD algorithm on a remote server.

Keywords

Computer vision, Grasp Detection, Pointcloud, GPD, Remote communication

Índice

Índice de ilustraciones:

1. Introducción y objetivos

La capacidad de un robot para manipular objetos de forma autónoma depende en gran medida de su percepción visual.

Tanto en la industria como en entornos colaborativos, la integración de la industria 4.0 y los sistemas robotizados para la elaboración de múltiples tareas es cada vez más extensa, convirtiéndose en una de las ramas de investigación y desarrollo más importantes. De la mano de la automatización de tareas va la visión artificial, resultando esencial para aumentar la escalabilidad y flexibilidad de la mayoría de los sistemas robotizados.

Una de las ramas de la robótica más estudiadas es el agarre de objetos. En este proyecto, se aborda el desarrollo de una aplicación que detecte automáticamente cómo y dónde un robot puede agarrar un objeto seleccionado por el usuario en tiempo real.

La complejidad del sistema reside en su capacidad para detectar agarres sobre objetos completamente desconocidos, sin necesidad de disponer de modelos tridimensionales previos ni entrenar redes neuronales específicas que limiten la gama de objetos que se puedan agarrar.

Otro aspecto destacado del proyecto es la capacidad de reconocer agarres de objetos vistos desde una única perspectiva, realizando una única toma de la escena, sin necesidad de emplear algoritmos que generen un modelo completo del objeto.

2. Marco Teórico de la Visión Artificial

La visión artificial es una rama de la inteligencia artificial y de la informática que pretende que las máquinas aprendan a interpretar y entender información visual del mundo real de manera similar a cómo lo hace el ojo humano. (IBM, Computer vision, 2025)

Utilizando cámaras y algoritmos, los sistemas de visión procesan imágenes y vídeos para extraer características relevantes que les ayuden a tomar decisiones automáticas. Esta tecnología permite automatizar tareas que requieren interpretación visual, como la detección y clasificación de objetos.

La visión artificial tiene aplicaciones en múltiples campos como la medicina y la seguridad, pero tiene especial importancia en la industria y la robótica. En este último, se convierte en una herramienta clave para dotar a los robots de la capacidad de percibir su entorno, pudiendo de esta manera identificar y clasificar objetos, calcular sus posiciones relativas e interactuar con ellos.

Gracias a los avances en el procesamiento de imágenes y el aprendizaje automático, los sistemas de visión artificial han alcanzado niveles de precisión y eficacia incluso superiores a los del ser humano en determinadas tareas. (IBM, Computer vision, 2025)

En el contexto de este trabajo, la visión artificial es el pilar fundamental para la creación de un sistema robótico de agarre de objetos, el cual sea capaz de reconocer su entorno, analizar un objeto aleatorio seleccionado y seleccionar un agarre óptimo para el mismo.

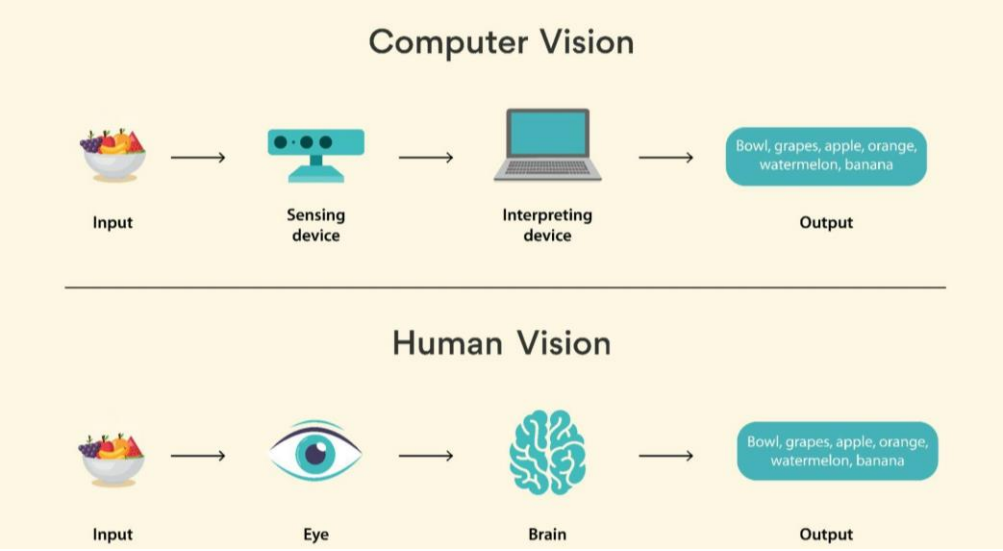

Ilustración 1. Comparación de la visión artificial con la visión humana (Turing, 2025)

2.1. Historia de la visión artificial

Los orígenes de la visión artificial se remontan a finales de los años 50, donde varios neurofisiólogos trataron de analizar la respuesta del cerebro de un gato ante distintos estímulos visuales. Descubrieron que respondía primero a bordes y líneas gruesas, lo cual significaría que el procesamiento de imágenes debía comenzar con las formas más simples. (HackerNoon, 2019)

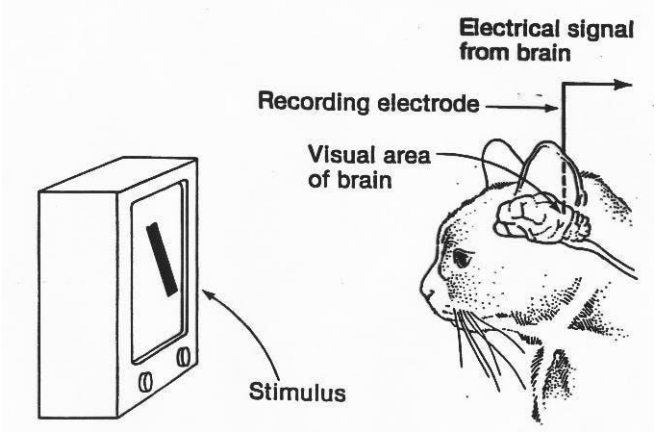

Ilustración 2. Experimento de Hubel y Wiesel, 1959 (HackerNoon, 2019)

Lawrence Roberts, a quién se considera padre de la visión artificial, publicó en 1963 un artículo titulado "Percepción mecánica de sólidos tridimensionales", donde analiza la extracción de información 3D sobre objetos sólidos a partir de imágenes 2D.

En 1966, el MIT impulsó el "Summer Vision Project", una ambiciosa idea de desarrollar un sistema que pudiera segmentar el fondo de una imagen y extraer objetos de esta automáticamente, en un verano. Aunque pronto se dieron cuenta de que las dimensiones del desafío al que estaban enfrentándose eran mucho mayores.

Durante las décadas de 1970 y 1980 se crearon los primeros algoritmos de procesamiento de imágenes, como la detección de bordes, esquinas y contornos. Técnicas que fueron fundamentales para la aplicación de la visión artificial en la industria, como el algoritmo *Canny*, el cual se sigue utilizando hoy en día.

Es en esta época cuando Kunihiko Fukushima, construye la red *Neocognitron*, la cual incluía varias capas convolucionales en una red neuronal.

En los 90, con el aumento de la capacidad de procesamiento de los ordenadores, la investigación en el campo crece exponencialmente. Se desarrollan algoritmos como SIFT y SURF para el reconocimiento de objetos a través de características locales. (HackerNoon, 2019)

En los 2000, los estudios se centraron en el reconocimiento de objetos y comenzaron las primeras aplicaciones de reconocimiento facial en tiempo real,

como el algoritmo de *Viola-Jones*. Además, a lo largo de esta década surge la estandarización de etiquetado de los datos visuales.

A medida que aumentaba la capacidad de procesamiento de los ordenadores y la disponibilidad de grandes cantidades de datos, surgieron nuevas técnicas dentro del aprendizaje automático capaces de abordar problemas más complejos. Entre ellas destaca el *deep learning* (aprendizaje profundo), un enfoque basado en redes neuronales con múltiples capas que permite analizar grandes conjuntos de datos, revolucionando campos como la visión artificial.

El inicio del ***Deep Learning*** lo marcó *AlexNet* cuando en 2012 ganó el desafío *ImageNet,* con un error de clasificación del 16% frente al 26% de los métodos tradicionales, demostrando el poder de las redes neuronales convolucionales. Impulsando aplicaciones como el reconocimiento facial, la conducción autónoma, la robótica avanzada y la medicina asistida por imágenes. (IBM, Neural networks, 2025)

Hoy en día, la visión artificial integra análisis 2D, 3D y procesamiento de video en tiempo real a través de sistemas que utilizan cámaras de alta velocidad y algoritmos de inteligencia artificial.

2.2. Visión 2D

La visión 2D se basa en el procesamiento y análisis de imágenes bidimensionales. Una imagen digital es un conjunto de valores numéricos, ordenados por filas y columnas en forma de matriz. Para pasar de una imagen continua a una digital se realiza un muestreo y cuantización, obteniendo una matriz compuesta por píxeles. A cada píxel se le asigna un valor cuantitativo si se trata de una imagen en blanco y negro o un vector de 3 valores (R, G, B) si se trata de una imagen a color.

La visión 2D emplea técnicas de procesamiento digital para obtener características útiles de las imágenes o videos. El procesamiento de una imagen 2D se basa en los siguientes pasos:

1. **Adquisición de la imagen**

   Se deben elegir las características apropiadas del sensor antes de adquirir las imágenes objetivo. Estas características dependen de la aplicación, se controlan parámetros clave como la resolución, sensibilidad y tiempo de exposición para la elección de la cámara adecuada.

2. **Preprocesamiento**

   Tras la captura de la imagen, se realiza un preprocesamiento de esta para mejorar su calidad antes de analizarla. Existen distintas técnicas que van desde el filtrado, como el *Filtro Gaussiano* o el *Filtro de la*

mediana, dependiendo del ruido que se quiera eliminar, la mejora del contraste, con una *Ecualización del histograma, h*asta la detección de bordes con algoritmos como *Canny o Sobel.* (IBM, Computer vision, 2025)

3. **Transformaciones geométricas**

   Se realizan distintas transformaciones geométricas, desde las más sencillas de rotación, escalado y traslación para corregir por ejemplo la perspectiva, hasta la detección de líneas, círculos o formas geométricas con herramientas como la *Transformada de Hough*

4. **Extracción de características**

   Una vez preprocesada y transformada la imagen, el sistema extrae características relevantes para identificar objetos en la imagen. Se utilizan técnicas como descriptores locales como *SIFT o SURF,* algoritmos de segmentación [1] como *SAM,* o distintas herramientas matemáticas con *NumPy.* Los algoritmos dependerán del objetivo del sistema.

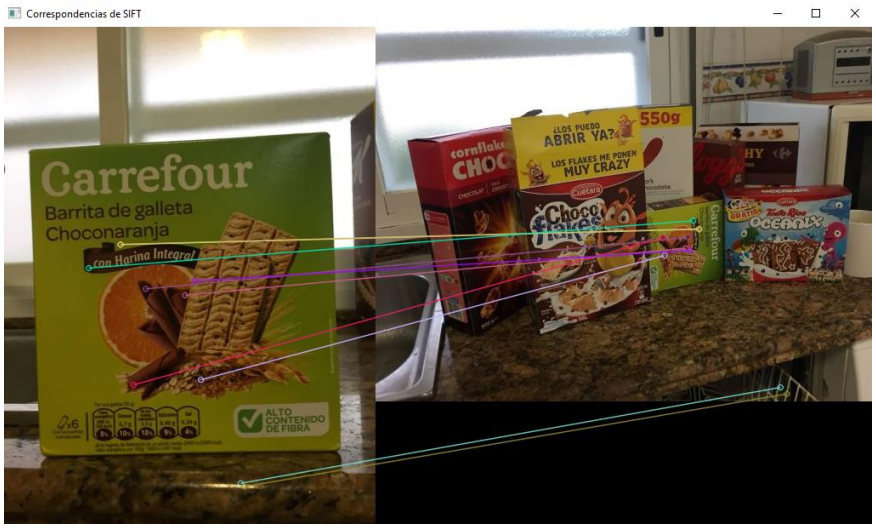

Ilustración 3. Algoritmo de detección de características SIFT (Elaboración propia)

La visión 2D tiene una gran presencia en la industria actual ya que es especialmente eficaz en entornos controlados. En este campo se programan cámaras inteligentes para detectar objetos conocidos, características locales dentro de estos, controles de calidad, etc. (Emergent Vision Technologies, 2025)

[1] *Segmentación:* proceso de asilar el área de interés de una escena.

2.3. Visión 3D

La visión 3D surge para suplir las limitaciones de la visión 2D cuando se requiere información de la profundidad de los objetos. Al introducir una tercera dimensión, esta permite obtener información espacial de los objetos, de la cual podemos extraer datos como la orientación, el volumen o la distancia de objetos a un sistema de referencia y utilizarlos para aplicaciones reales. (Tekvisa, 2022)

Existen distintas tecnologías para la extracción de imágenes o videos en tres dimensiones. Las más utilizadas se clasifican en:

- **Cámaras de tiempo de vuelo (TOF):** combinan una cámara RGB como las utilizadas en la visión bidimensional con un sensor de profundidad, el cual mide el tiempo que tarda un LED de alta potencia en regresar al sensor del dispositivo. Se denominan cámaras RGB-D.
- **Visión estéreo:** se utilizan dos cámaras colocadas en distintas posiciones para obtener distintos puntos de vista del objeto y calcular la representación 3D mediante triangulación.
- **Triangulación láser:** consiste en proyectar una línea laser sobre el objeto y capturarla con una lente a una distancia y ángulo conocido, obteniendo así el perfil del objeto. Es el sistema más utilizado en la industria para construir nubes de puntos debido a su alta precisión.
- **Luz estructurada**: se utiliza un proyector que ilumina con un patrón el objeto capturado y una o varias cámaras desde diferentes ángulos observan la deformación de la proyección sobre el objeto.

Estas tecnologías son ampliamente utilizadas en visión artificial 3D en entornos industriales. (Tekvisa, 2022) (BCNVision, 2025)

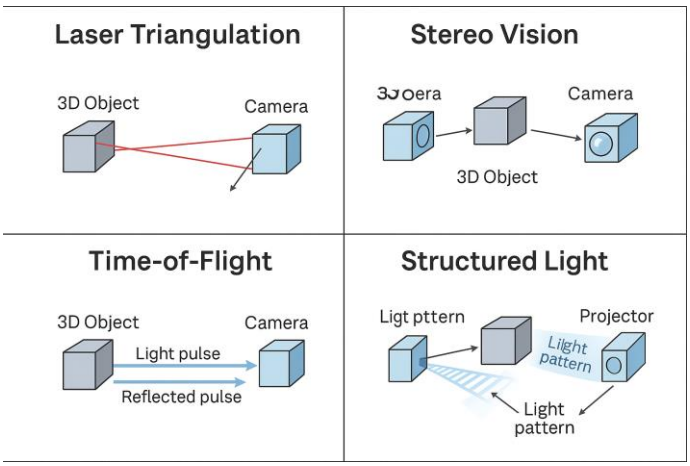

Ilustración 4. Principales tecnologías de extracción de imágenes 3D (ChatGPT, 2025)

La representación 3D de un objeto o entorno se conoce como **nube de puntos**, donde cada punto del espacio está compuesto por un vector de 3 coordenadas (x, y, z) que indican su posición, además de características adicionales como el color o la intensidad.

Al igual que en la visión 2D, se realiza un preprocesamiento de las nubes de puntos, realizando filtrados, para eliminar *outliers* o reducir la densidad de los puntos para aligerar el procesamiento.

Dentro del procesamiento de las nubes se pueden realizar segmentaciones, para aislar las regiones de interés, con algoritmos como *RANSAC* para identificar planos. O calcular en cada punto un vector normal para definir la orientación de las superficies, detectar esquinas o planear agarres. A partir de los puntos y sus normales, se puede realizar la reconstrucción 3D de la superficie.

Un paso crucial en la visión 3D es la alineación de la imagen 2D con el mapa de profundidad, o en su caso, la alineación de varias nubes de puntos si se realizan fotografías desde distintas perspectivas.

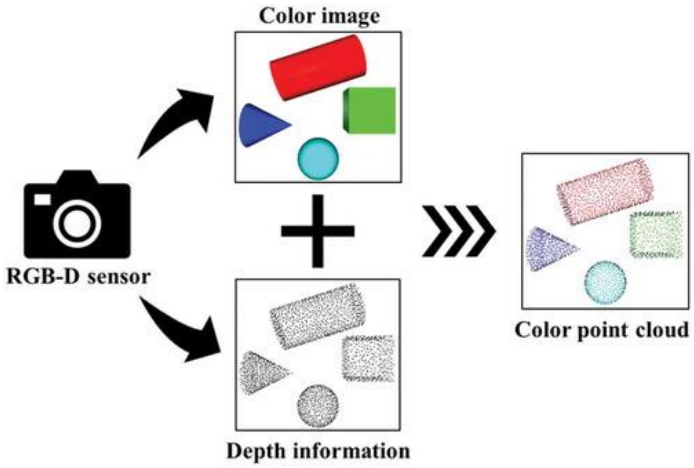

Ilustración 5. Proceso de captura RGB-D a nube de puntos coloreada (Wang, 2021)

El procesamiento de las nubes de puntos depende de la utilidad que se le quiera dar al sistema de visión, cabe destacar su importancia en tareas robóticas de agarre, ya que permite calcular el tamaño, orientación y agarre óptimo de objetos desconocidos.

2.4. Visión en la robótica

La integración de la visión artificial en la robótica transforma sistemas automáticos en sistemas capaces de percibir e interpretar su entorno, un paso crucial en el avance de la robótica colaborativa.

La robótica es una disciplina que combina mecánica, electrónica e informática para construir sistemas capaces de realizar tareas de forma autónoma. Dentro de esta área, la robótica colaborativa permite que los robots y humanos trabajen en entornos compartidos. Dentro de este campo, la capacidad visual de los robots se vuelve más importante ya que permite detectar la presencia humana, prever trayectorias y evitar colisiones. (Universal Robots, 2025)

En sistemas de agarres robóticos, la visión artificial permite localizar objetos, determinar sus dimensiones y orientación, evaluar distintos puntos de agarre y trazar y simular las trayectorias adecuadas para su agarre. Según la complejidad del sistema, puede utilizarse un sistema de visión 2D si los objetos se encuentran definidos sobre una superficie o requerir de un sistema 3D.

Dentro de la robótica, la visión desempeña distintos roles importantes:

- **Identificación de objetos:** reconocimiento de objetos, basado en características visuales.
- **Localización:** estimación de la posición y orientación de objetos o características para planificar o modificar trayectorias acorde a ellas.
- **Seguimiento en tiempo real:** permiten la actualización de la posición de un objeto en un sistema en movimiento, pudiendo ejecutar agarres en movimiento.
- **Trazabilidad y control de calidad:** en procesos industriales, permiten tener un seguimiento de las piezas manipuladas a la vez que se realizan controles de calidad de las mismas.

Un claro ejemplo de la aplicación de la visión artificial en la robótica son los procesos *pick & place*, donde los robots pueden detectar los objetos, calcular su posición, ejecutar una trayectoria acorde a la misma, analizar el objeto y colocarlo en la posición deseada de una manera rápida y eficaz.

Ilustración 6. Pick and Place con sistema de visión (Shin, 2024)

2.5. Redes neuronales

Una red neuronal es un modelo de *machine learning* que toma decisiones simulando la forma en que trabajan las neuronas del cerebro humano. Las redes neuronales constan de varias capas de nodos: una capa de entrada, una o varias capas ocultas y una capa de salida. Cada nodo se conecta a los demás con su propia ponderación y umbral, el cual al ser superado se activa, enviando datos a la siguiente capa de la red. (IBM, Neural networks, 2025)

Las redes neuronales convolucionales CNN, han supuesto una revolución en el procesamiento de imágenes y visión artificial. Estas redes, están diseñadas para trabajar con datos visuales ya que emplean filtros que recorren la imagen para detectar características.

Se aprovechan de principios del álgebra lineal como la multiplicación de matrices para identificar patrones dentro de una imagen. A medida que las imágenes atraviesan las distintas capas y se detectan características como bordes, texturas y formas, se construyen representaciones que permiten identificar objetos, reconocer rostros, clasificar escenas, entre otras tareas.

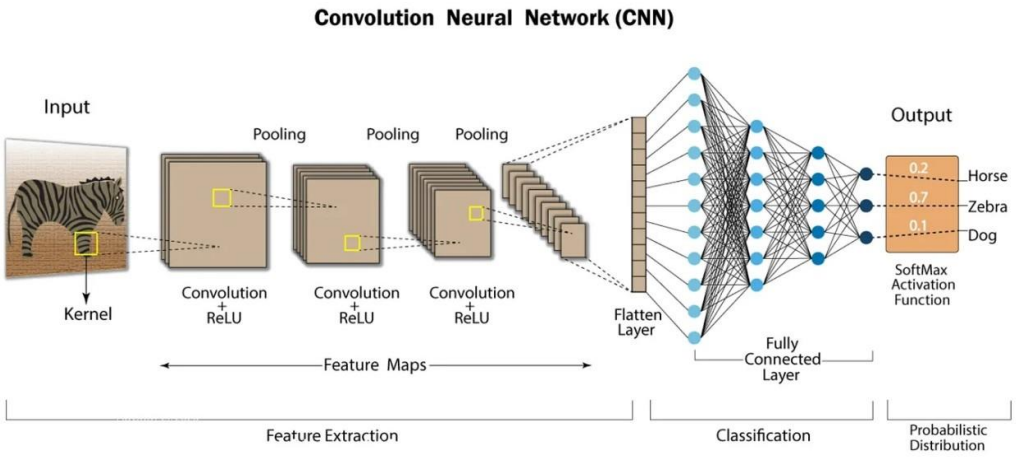

Ilustración 7. Funcionamiento de una Red Neuronal Convolucional (Haque, 2023)

Las redes neuronales requieren de un entrenamiento previo, donde se utilizan grandes conjuntos de datos etiquetados y a través de muchas iteraciones, se ajustan los pesos internos de la red para minimizar el error de predicción.

Actualmente, las redes neuronales se utilizan en gran variedad de aplicaciones dentro de la industria, especialmente en la robótica y visión artificial. Su capacidad de aprender directamente de los datos sin necesidad de diseñar manualmente algoritmos específicos hace que aumenten su valor, sobre todo en entornos complejos y dinámicos, donde no se pueden abordar los problemas con técnicas tradicionales.

3. La visión artificial en la robótica de agarre

El desarrollo de sistemas de agarre dentro de la robótica asistida por visión artificial es un área de investigación en auge dada su importancia para la automatización. La capacidad de un robot de identificar y manipular objetos de manera autónoma depende directamente de su capacidad de percibir su entorno visualmente.

A lo largo del tiempo, este campo ha evolucionado desde métodos geométricos clásicos hasta sistemas basados en aprendizaje profundo, cubriendo las limitaciones que los métodos más tradicionales no podían superar.

3.1. Métodos tradicionales de agarre guiado por visión

Antes del auge del aprendizaje profundo, la investigación en estrategias de agarre robótico se basaba en enfoques analíticos y heurísticos, necesitando información exacta de la geometría y características del objeto y dependiendo de la precisión de los sensores. Estos métodos se pueden clasificar en dos grandes grupos: basados en características geométricas y basados en sensores 3D con reglas heurísticas.

A pesar de considerarse métodos clásicos, no se han quedado obsoletos y en la actualidad siguen siendo de gran utilidad en entornos industriales controlados y en la interacción con objetos conocidos.

3.1.1. Métodos basados en características geométricas

Los métodos que se basan en la extracción de las características geométricas relevantes del objeto que se desea agarrar se fundamentaban en la utilización del modelo CAD (Computer-Aided Design) del objeto. Estos sistemas requerían que el objeto estuviera previamente modelado para poder detectar su pose[2] (posición y orientación) mediante algoritmos matemáticos.

Algunas de las técnicas más comunes para la identificación del objeto a partir de imágenes, como **SIFT** (Scale-Invariant Feature Transform) o **SURF** (Speeded-Up Robust Features), se basaban en la extracción de características locales. (Lowe, 2004) (Bay, 2008)

Una vez identificada la pose del objeto en el espacio, el sistema generaba una configuración de agarre basada en criterios como el centrado de masas, la estabilidad, los puntos de contacto óptimos, etc. Este enfoque usa el concepto *wrench space*[3] para verificar la estabilidad del agarre. (Bicchi & Kumar, 2000)

[2] *Pose:* posición y orientación de un objeto
[3] *Wrench space (espacio de fuerzas y momentos):* espacio matemático de 6 dimensiones que combina fuerzas (Fx, Fy, Fz) y torques (Tx, Ty, Tz) utilizado para analizar la capacidad de un agarre robótico para resistir perturbaciones externas.

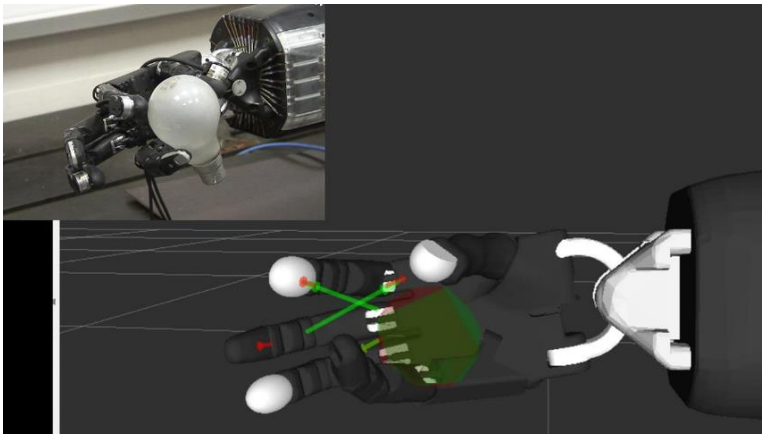

Ilustración 8. Grasp Wrench Space (Bimbo, 2016)

Estos métodos eran eficaces en entornos estructurados, donde la variabilidad de los objetos y su entorno es mínima, como líneas de montaje industriales. Así es que su principal limitación radica en la falta de flexibilidad ante objetos nuevos y oclusiones.

3.1.2. Métodos heurísticos con sensores 3D

Con la aparición de sensores de profundidad como Microsoft Kinect (Zhang, 2012) y sensores RGB-D (que combinan una imagen en color con información de profundidad), se redujo la dependencia de los modelos CAD ante la posibilidad de estimar la geometría tridimensional a través de la información visual. Gracias a esto, se desarrollaron métodos que analizaban la nube de puntos o la imagen de profundidad para localizar puntos de agarre. Estos métodos se basaban en reglas heurísticas como:

- Buscar superficies planas horizontales sobre las que se encuentran los objetos
- Detectar bordes o cavidades en la nube de puntos
- Calcular las normales de la superficie y segmentar por zonas potencialmente útiles
- Detectar superficies planas de los objetos

La información de profundidad de la imagen permitió emplear técnicas novedosas como la segmentación basada en la profundidad para identificar objetos y posteriormente calcular vectores normales a la superficie que permitan guiar el efector final del robot.

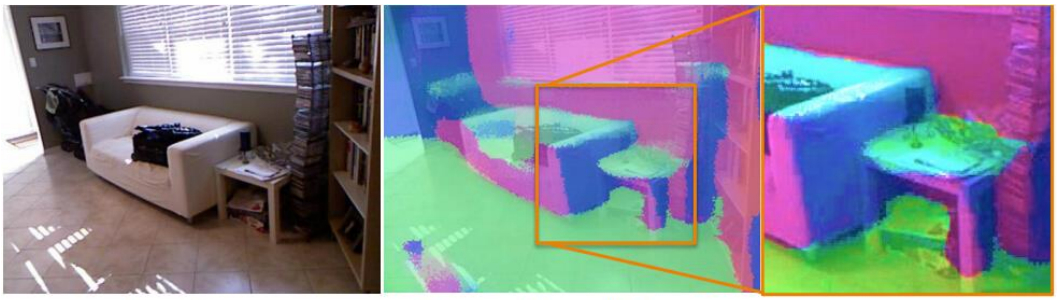

Ilustración 9. Cálculo de normales a la superficie (Wang X, 2016)

Otras propuestas utilizaban histogramas de orientaciones de superficie (Histograms of Oriented Gradients o HOG aplicados a datos 3D) para estimar la región en la que se obtendría el mejor agarre. (Hsiao, 2010)

Estas técnicas combatían las limitaciones anteriores puesto que no requieren de información previa del objeto, lo cual les permite trabajar con objetos desconocidos, ya que las técnicas se basaban en aspectos generales. Sin embargo, la precisión de los sistemas era menor y eran más sensibles a aspectos físicos como la calidad de la nube de puntos, el ruido en la imagen, las oclusiones u objetos superpuestos, etc.

3.2. Enfoques basados en aprendizaje profundo

El auge del *deep learning* produjo un cambio en la forma en que se abordan los problemas de agarre dentro de la robótica de manipulación. A diferencia de los métodos tradicionales anteriormente mencionados, que requerían modelos 3D exactos de los objetos o se basaban en reglas físicas y heurísticas, los métodos basados en el aprendizaje profundo permiten extraer información útil directamente de los datos obtenidos de los sensores. A través de imágenes RGB-D o nubes de puntos se pueden predecir poses de agarre viables, incluso con imágenes parciales, objetos desconocidos o geometrías complejas.

El deep learning resulta especialmente útil en entornos variables, donde los métodos tradicionales fallan. A su vez, se trata de métodos más complejos, normalmente con mayor coste computacional y necesidad de un preentrenamiento.

Dentro de este campo, surgen dos líneas de investigación: detectores de agarre basados en aprendizaje supervisado sobre nubes de puntos, como **GPD (Grasp Pose Detection)** y sistemas masivamente entrenados, más recientes como **GraspNet.**

3.2.1. GPD (Grasp Pose Detection)

Uno de los primeros métodos que incluían aprendizaje profundo para la predicción de agarres en nubes de puntos es **Grasp Pose Detection (GPD)** propuesto por Gualtier, Ten Pas, Saenko y Platt en 2016. Este método se centra en detectar poses de agarre de 6 grados de libertad (6-DoF) a partir de una nube de puntos, sin necesidad de un preprocesamiento de la nube, segmentación previa de los objetos ni sus modelos CAD. (Marcus Gualtieri, 2016) (Andreas ten Pas, 2017)

Ilustración 10. GPD ejemplo de uso (Nowosad, 2018)

El proceso que realiza el GPD se clasifica en dos etapas principales:

1. **Muestreo de los candidatos de agarre:** se generan múltiples candidatos a lo largo de toda la nube de puntos tridimensional. Para ello se extraen subconjuntos locales, creando pequeñas regiones 3D que pueden ser voxelizadas (es decir, divididas en pequeños volúmenes cúbicos llamados *voxels,* que representan el espacio en tres dimensiones). Se analizan estas regiones donde un agarre podría ser posible físicamente, basándose en la densidad y distribución de los puntos que hay dentro de cada una.
2. **Clasificación de agarres:** una red neuronal convolucional evalúa cada candidato de agarre y determina si es válido o no. Esta clasificación se basa en características geométricas de la nube de puntos proyectada en el sistema de referencia del gripper[4]. Además, pasa por distintos filtros, algunos configurables como la dirección del agarre o la apertura necesaria del efector.

[4] *Gripper:* efector final de un robot, diseñado para manipular objetos, generalmente compuesto por una pinza que se abre y cierra.

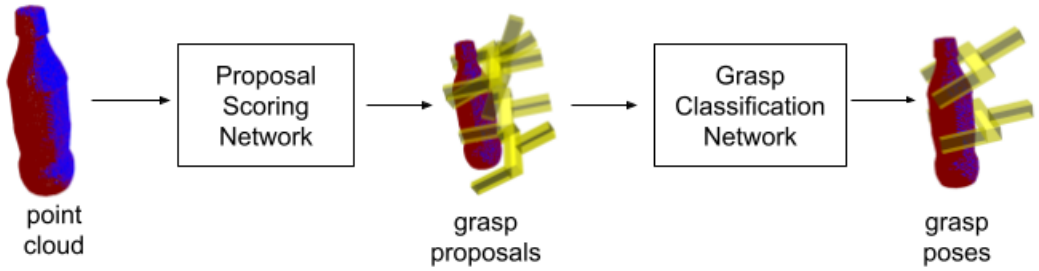

Ilustración 11. Esquema de funcionamiento de GPD (Ten Pas, A., Keil, C., & Platt, R. , 2021)

Una de las principales ventajas de GPD es que permite operar directamente sobre nubes de puntos obtenidas con los sensores RGB-D, sin necesidad de segmentar cada objeto. Esto lo hace especialmente útil en entornos desordenados o con objetos parcialmente visibles. Esto se debe a que el algoritmo funciona extrayendo regiones locales de la nube de puntos, sin necesidad de conocer de antemano a que objeto pertenecen, y generando múltiples candidatos a lo largo de toda la escena. Es decir, el sistema no necesita conocer donde empieza o termina cada objeto para generar un agarre válido para el mismo.

Sin embargo, en la práctica, como veremos a continuación, si se desean generar agarres para un objeto específico dentro de una escena desordenada y evitar que se generen agarres en el entorno, es necesaria una presegmentación. Con esta operación, extraemos la nube de puntos del objeto del entorno, asegurando que los agarres son generados únicamente para el objeto seleccionado. Lo cual, a su vez, supone una ventaja en cuanto al coste computacional que requiere analizar una escena desordenada frente a un único objeto segmentado.

Esta diferenciación, otorga una mayor flexibilidad al algoritmo, ya que permite su utilización en modo "denso", generando agarres sobre toda la escena, lo cual puede ser útil para operaciones como *bin picking*[5], o en modo más dirigido, indicando el objeto que se desea agarrar y utilizando la nube segmentada del mismo.

Otra de las principales ventajas del GPD es que permite generar múltiples poses de agarre para un objeto. En la segunda etapa del algoritmo, se filtran y clasifican los candidatos de agarre y se ordenan según una puntuación. A través de un parámetro, se puede seleccionar el número de agarres que se quieren obtener, siendo estos los que obtengan mayor puntuación en el análisis. Gracias a esta posibilidad de obtener varios agarres sobre un mismo objeto, facilita el posterior estudio de esto y la selección del más adecuado en

[5] *Bin picking:* técnica de robótica que consiste en identificar y recoger objetos desordenados dentro de un contenedor mediante visión artificial y un brazo robótico.

función de la cinemática del robot o la planificación de la trayectoria para evitar colisiones.

Además, mientras se realiza el proceso de análisis de la escena y clasificación de los agarres, GPD permite la visualización de cada uno de estos pasos. Pudiendo obtener escenas desde todos los candidatos generados inicialmente, hasta los finalmente seleccionados, pasando por varias etapas de filtrado, como la dirección, la apertura de la garra, etc. Esta visualización se realiza en formato nube de puntos, permitiendo al usuario navegar por la imagen, para observar el objeto y los agarres desde la perspectiva que desee. La elección de qué se quiere visualizar o no se realiza a través de los parámetros de entrada del algoritmo.

No obstante, presenta varias limitaciones, en primer lugar, implica un alto coste computacional, debido al número elevado de candidatos a evaluar, sobre todo en escenas desordenadas. En segundo lugar, la calidad de los agarres depende en alta medida de la calidad de la nube de entrada, siendo muy influyente en ella la resolución, el ruido y las oclusiones, lo cual puede generar falsas detecciones o agarres imposibles por falta de información.

A pesar de sus limitaciones, el GPD es una solución eficaz y versátil, dada su flexibilidad en entornos no estructurados y su sencillez de ser aplicado en escenarios reales. Para que este funcione correctamente, necesita de una red neuronal entrenada, con un conjunto de datos *(dataset)* que incluye múltiples ejemplos de éxito y fracaso, lo cual le permite distinguir entre configuraciones válidas y no. Esto limita su escalabilidad ya que la creación de los *datasets* requiere de un gran tiempo y esfuerzo. No obstante, la necesidad de un entrenamiento específico dependerá de la aplicación y la tasa de éxito necesaria.

3.2.2. GraspNet

Años más tarde del éxito del GPD, en búsqueda de mejorar la escalabilidad y generalización de los modelos de predicción de agarres, los investigadores de Tsinghua University y Microsoft Research Asia desarrollan **GraspNet.** (Gou, et al., 2021)

A diferencia de GPD, GraspNet consta de una potente arquitectura de red neuronal entrenada *end-to-end* para predecir configuraciones de agarre 6-DoF desde imágenes RGB-D, sin necesidad de muestrear candidatos manualmente ni segmentar previamente los objetos. (Gou, et al., 2021)

El éxito de esta tecnología reside en que el entrenamiento de la red se basa en un dataset sin precedentes: *GraspNet-1Billion*, el cual contiene más de mil millones de poses de agarres simulados sobre 97280 imágenes RGB-D.

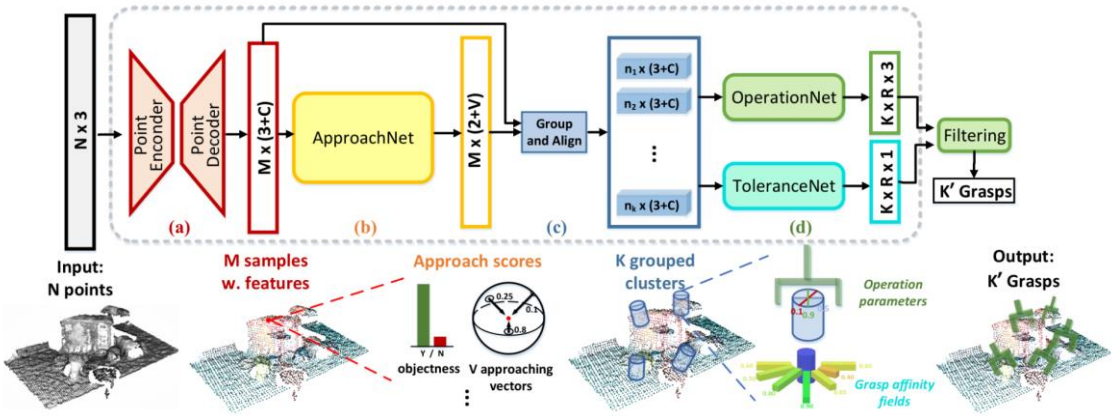

Ilustración 12: Funcionamiento general de la red GraspNet (Fang, Wang, Gou, & Lu, 2020)

El funcionamiento de la red GraspNet se divide en 4 etapas:

a) Entrada – Codificación de la nube de puntos

La entrada a la red es una nube de N puntos 3D[6] **[N x 3]**. Para extraer las características relevantes de la nube se utiliza una red tipo *encoder-decoder*[7]. El resultado de esta etapa es M puntos con C características por punto, es decir, se obtiene una nube reducida en puntos, pero rica en información relevante, en forma de matriz: **[M x (3+C)].**

b) Predicción de vectores de aproximación – ApproachNet

En esta etapa se utiliza una red *ApproachNet*, la cual toma como entrada la matriz anterior [M x (3+C)] (M puntos con sus coordenadas y características) y clasifica cada punto en agarrable o no y desde qué dirección. Generando, para cada punto 2 valores de salida, que indican si es agarrable o no y V posibles vectores de aproximación, asociando a cada uno de ellos una puntuación, obteniendo una matriz: **[M x (2+V)].**

c) Agrupación en regiones cilíndricas

Los puntos candidatos M se agrupan en regiones cilíndricas, alineadas con sus vectores de aproximación. Dentro de cada cilindro se incluyen los puntos 3D vecinos que pertenecen a esa área, permitiendo considerar la geometría local alrededor de cada punto. La salida de esta etapa son K regiones cilíndricas de puntos en torno al candidato de agarre.

[6] *Puntos 3D*: cada punto tiene 3 coordenadas: x, y, z.
[7] *Red encoder-decoder* (codificador-decodificador): transforma los datos complejos en una representación comprimida de los mismos que luego se expande para generar una salida útil.

d) Predicción del agarre – OperationNet y ToleranceNet

El objetivo de esta etapa es predecir para cada punto de agarre candidato cómo debe ser realizado el agarre (posición y orientación exacta) y qué tan robusto es el agarre. Para ello usa dos redes neuronales que se ejecutan en paralelo:

- **OperationNet:** predice los parámetros de la operación de agarre, es decir cómo debe moverse el gripper. Define para cada grupo cilíndrico K el ángulo de rotación, la apertura de la pinza, el offset en el eje z. **[K x R x 3]**
- **ToleranceNet:** define para cada grupo cilíndrico K cuán robusto es el agarre dado, si se produce una pequeña desviación ¿sigue siendo válido? **[K x R x 1]**

Finalmente, se filtran los agarres según la información obtenida anteriormente y se selecciona un grupo de los agarres con mayor puntuación. (Fang, Wang, Gou, & Lu, 2020)

GraspNet representa un avance significativo respecto a los métodos anteriores gracias a su capacidad para generalizar en entornos no estructurados y manejar objetos no vistos durante el entrenamiento (Fang, Wang, Gou, & Lu, 2020).

Al emplear el dataset **GraspNet-1Billion** se elimina la necesidad de entrenar modelos específicos para cada escenario y realizar un muestreo manual. Esto reduce el coste computacional en inferencia, aunque el entrenamiento inicial demanda muchos recursos dado el tamaño del dataset.

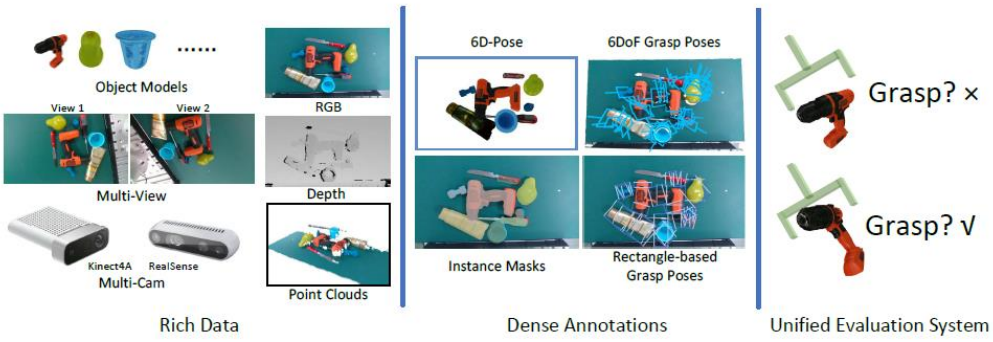

Ilustración 13. GraspNet-1Billion (Fang, Wang, Gou, & Lu, 2020)

Gracias a su arquitectura *end-to-end* basada en *PointNet++* se permite procesar toda la nube de entrada sin segmentación previa, logrando tasas de éxito del 82%. Otro de los grandes avances es el empleo de *ToleranceNet* que permite evaluar la estabilidad de los agarres.

Al igual que GPD, su rendimiento depende en gran medida de la calidad de la nube de puntos de entrada y por tanto de la calidad del sensor y el ruido u oclusiones que pueda haber en la escena.

La principal limitación que presenta este modelo es el preprocesamiento del dataset, el cual requiere de GPUs con gran potencia y grandes tiempos de entrenamiento, esto limita su accesibilidad para pequeños equipos de investigación. Por este motivo, se descartó su uso en este proyecto, optándose por adaptar a nuestra necesidad el modelo GPD.

4. Herramientas empleadas

El objetivo de este trabajo es desarrollar un sistema de agarre de objetos en un robot mediante visión artificial. Para ello, se ha requerido la integración de diversas herramientas, tanto de hardware como de software. Estas herramientas permiten la adquisición de información del entorno, el procesamiento de estos datos visuales y la ejecución de acciones de manipulación robótica.

A continuación, se detallan los componentes utilizados, divididos en dos categorías principales: hardware, correspondiente a los elementos físicos empleados, y software, que abarca las bibliotecas, entornos y algoritmos utilizados para el desarrollo del sistema.

4.1. Equipamiento físico

El sistema desarrollado ha sido propuesto para la integración con un brazo robótico **Kinova Gen-3** de 7 grados de libertad. Para la adquisición de imágenes contamos con una **cámara Intel RealSense** tipo **RGB-D,** la cual será la herramienta central para desarrollar un sistema de visión artificial asociado a ella. Además, contamos con una pinza **Robotiq 2F-85** utilizada como efector final para realizar agarres adaptativos, a la cual adaptaremos el desarrollo del sistema.

4.1.1. Cámara Intel RealSense D415

La cámara Intel RealSense D415 es un sensor de visión 3D que permite obtener información de profundidad mediante la tecnología de estéreo activo. Tiene las siguientes características técnicas:

- Tipo de tecnología: Estéreo activo con proyector infrarrojo (IR)
- Resolución RGB: 1920x1080
- Resolución de profundidad: 1280 x 720 píxeles
- Rango operativo ideal: De 0.5m a 3m
- Precisión de profundidad: Menos del 2% a una distancia de 2m
- Dimensiones: 99mm x 23 mm x 20mm
- Conectividad: USB 3.1 Gen 1

Ilustración 14. Cámara Intel RealSense D415 (Intel Corporation, 2025)

Esta cámara se ha empleado para capturar tanto la imagen RGB como el mapa de profundidad del entorno, permitiendo detectar y localizar objetos y sus dimensiones en el espacio tridimensional. (Intel Corporation, 2025)

4.1.2. Robot Kinova Gen3

El Kinova Gen3 es un brazo robótico colaborativo de última generación que dispone de 7 grados de libertad, lo cual le otorga gran flexibilidad y capacidad para ejecutar tareas de manipulación complejas. Consta de las siguientes características técnicas:

- Grados de libertad: 7 DoF
- Carga útil máxima: 4kg
- Peso del brazo: 8.2kg
- Alcance máximo: 902mm
- Velocidad máxima: 50cm/s
- Temperatura de funcionamiento: -30 a 35ºC
- Sensores: torque, posición, corriente, voltaje, temperatura, acelerómetro y giroscopio
- Tecnologías soportadas: MATLAB, ROS, ROS2, C++, Python, Gazebo

Ilustración 15. Robot Kinova Gen3 (ROS Components, 2025)

Su compatibilidad con ROS 2 y su arquitectura modular lo convierten en una herramienta ideal para el desarrollo de aplicaciones de robótica avanzada, como el agarre autónomo de objetos mediante visión artificial. (Kinova Robotics, 2018)

4.1.3. Pinza Robotiq 2F-85

La pinza Robotiq 2F-85 es un efector final adaptativo diseñado para tareas de agarre en robótica industrial y colaborativa. Se ha integrado en el brazo Kinova Gen3 para manipular objetos de distintas formas y tamaños. Tiene las siguientes características técnicas:

- Apertura máxima: 85mm
- Número de dedos: 2
- Fuerza de agarre ajustable: 20N a 235N
- Carga útil: Hasta 5kg
- Peso de la herramienta: 1kg
- Velocidad de cierre: 20 a 150 mm/s
- Temperatura de funcionamiento: -10 a 50 ºC

Ilustración 16. Pinza Robotiq 2F-85 (VICOSYSTEMS, 2025)

Esta herramienta permite realizar agarres precisos y seguros, permitiendo agarres tanto envolventes como de pellizco con una velocidad y fuerza programables. (Robotiq, 2025)

4.2. Herramientas de software

El desarrollo de un sistema de visión requiere el uso de varias herramientas de software. Desde un entorno donde se pueda realizar la programación del sistema de manera intuitiva y que nos facilite la integración de las librerías y simulaciones necesarias, tanto para la adquisición de datos a través de los dispositivos hardware, como su interpretación, procesamiento y visualización, hasta distintas herramientas o protocolos de comunicación para integrar y automatizar el sistema.

A continuación, se indican las principales herramientas de software que se han empleado para desarrollar la aplicación de visión.

4.2.1. Entorno de programación

El principal desarrollo del sistema se realiza en **Python**, un lenguaje de programación de alto nivel que dispone de una extensa colección de bibliotecas y módulos adaptables a múltiples aplicaciones. Fue creado en 1991 por Guido van Rossum y gracias a su sintaxis clara y legible, ha tenido una continua evolución, convirtiéndose en uno de los lenguajes de programación más utilizados en la actualidad.

El desarrollo del sistema se ha llevado a cabo íntegramente en el entorno de desarrollo integrado **Spyder,** incluido dentro de **Anaconda Navigator.** Spyder (*Scientific Python Development Environment*) es un entorno ampliamente utilizado en el ámbito científico por su enfoque orientado a análisis de datos, visualización y computación numérica. (Spyder Developers, n.d.) (Anaconda Inc., 2025)

Este entorno está especialmente diseñado para realizar programación en Python y ofrece una interfaz muy completa, que combina varias herramientas clave en una sola aplicación:

- Un **editor de código** con resaltado de sintaxis, autocompletado, análisis estático y navegación por bloques, lo cual facilita el desarrollo modular y la realización de un código estructurado y con menor probabilidad de errores.
- Una **consola interactiva IPython,** que permite ejecutar instrucciones en tiempo real e imprimir distintos mensajes fácilmente, lo que facilita la interpretación y depuración del código.
- Un **explorador de variables,** útil para depurar estructuras complejas, observando sus valores en memoria durante la ejecución
- Un sistema de **paneles gráficos** para visualizar salidas como gráficos, imágenes, visualizaciones 3D, etc.

A lo largo del desarrollo de este proyecto, Spyder ha resultado especialmente útil para controlar y supervisar el flujo de trabajo del sistema, facilitando la prueba individual de módulos y la integración entre ellos. La capacidad de ejecución modular y eficiencia de la recepción de mensajes de depuración en tiempo real ha facilitado el desarrollo del trabajo y la detección de errores de manera rápida y localizada.

Al estar integrado dentro de Anaconda, Spyder permite gestionar los entornos virtuales, instalar librerías y paquetes requeridos como *Open3D, Flask, Paramiko o scipy* y mantener una configuración controlada y replicable.

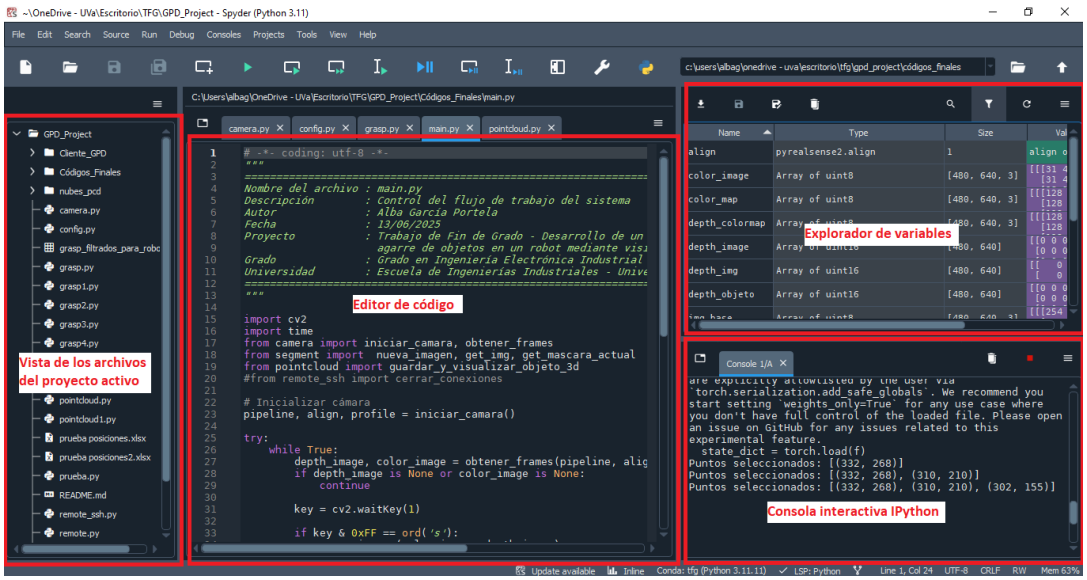

Ilustración 17. Entorno Spyder (Elaboración propia)

4.2.2. Adquisición de datos

Para la adquisición de datos de la cámara se empleará la librería ***pyrealsense2***, el *wrapper* oficial en Python del SDK de Intel RealSense llamada, que permite controlar, configurar y acceder a los datos de la cámara en tiempo real.

A través de esta librería, se configuran los parámetros de adquisición de la cámara, tanto de la imagen en color como de la imagen en profundidad, y se sincroniza la captura de ambos. Este paso de **alineación** de los datos de profundidad con la imagen RGB es esencial para el posterior procesamiento de estas.

Gracias a *pyrealsense2* es posible obtener matrices *NumPy* con los datos RGB-D, que se utilizarán para construir nubes de puntos precisas sobre las que se puedan realizar procesos de segmentación y análisis.

Hay que tener en cuenta los ejes cartesianos en los que trabaja esta librería:

- **Eje X:** eje horizontal, positivo hacia la derecha desde la cámara
- **Eje Y:** eje vertical, positivo hacia abajo desde la cámara
- **Eje Z:** eje de profundidad, positivo hacia delante de la cámara

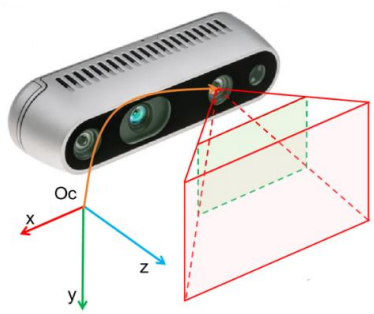

Ilustración 18. Sistema de referencia RealSense (Intel Corporation, 2025)

4.2.3. Visualización

Para el procesamiento y visualización de las imágenes 2D y escenas 3D se han utilizado dos bibliotecas de código abierto, diseñadas para facilitar el tratamiento de información visual.

OpenCV (Open Source Computer Vision Library) es una biblioteca de visión artificial especializada en el procesamiento de imágenes y vídeos en tiempo real. En este trabajo, se emplea el módulo **cv2** de OpenCV para tareas clave:

- Visualización de las imágenes RGB obtenidas de la cámara
- Visualización en 2D del mapa de profundidad obtenido (en escala de colores)
- Interacción con el usuario, permitiéndole la selección de puntos directamente sobre la imagen mediante eventos del ratón
- Dibujar información relevante sobre las imágenes reales
- Guardado de imágenes

Gracias a esta librería, podemos visualizar información real de la cámara y de los procesos que se están realizando, lo cual es esencial para comprobar su correcto funcionamiento. (OpenCV, 2025)

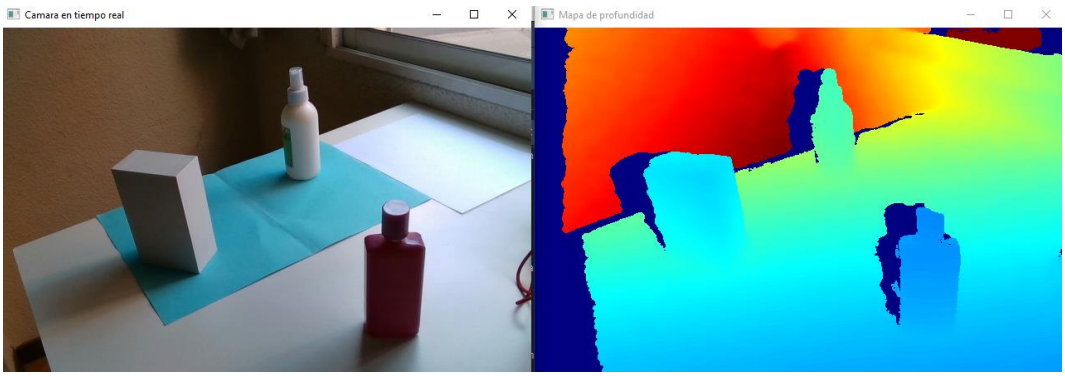

Ilustración 19. Imagen RGB y mapa de profundidad en tiempo real (Elaboración propia)

Por otro lado, Open3D es una biblioteca especializada en el procesamiento y visualización de datos 3D, resultando particularmente útil para trabajar con nubes de puntos. En este trabajo, se emplea para tareas como:

- Creación de nube de puntos que contenga el objeto a analizar
- Visualización nubes de puntos
- Interacción con el usuario, permitiendo a este navegar a través de las nubes de puntos para analizar la información desde distintas perspectivas de una manera sencilla e intuitiva.
- Dibujar información relevante sobre la nube de puntos del objeto
- Guardado de nubes de puntos

La combinación de estas herramientas ha resultado fundamental para unir los procesos en la imagen 2D con la escena en 3D. De esta manera, podemos extraer información relevante de la imagen RGB y extrapolarla a la nube de puntos o viceversa.

Ilustración 20. Nube de puntos de la escena (Elaboración propia)

Ilustración 21. Navegación por la nube de puntos (Elaboración propia)

4.2.4. Segmentación de objetos

Uno de los pasos fundamentales para estimar puntos de agarre de un objeto es aislarlo del entorno. Para realizar esta tarea se ha empleado **SAM (Segment Anything Model),** un modelo de segmentación de imágenes desarrollado por *Meta AI*, y en este caso, utilizando una versión adaptada por *Ultralytics***,** dada su sencilla integración en Python.

A diferencia de los modelos de segmentación de imágenes tradicionales, que requieren gran cantidad de datos y un entrenamiento específico para cada tarea, SAM ha sido entrenado con millones de imágenes y máscaras, por lo que está preparado para segmentar cualquier objeto en una imagen sin necesidad de entrenamiento adicional.

En este trabajo, el modelo SAM se emplea para segmentar el objeto que el usuario desea agarrar, mediante la selección de uno o varios puntos sobre la imagen RGB de la cámara. El modelo nos proporciona una máscara binaria del objeto, representando los píxeles que nos resultan de interés.

Gracias a esta máscara, es posible extraer el objeto de la imagen RGB y de la nube de puntos. De esta manera, podemos centrar análisis de una manera más sencilla, exclusivamente en el objeto de interés, sin que el entorno interfiera. Por tanto, no es necesario un control del entorno, ya que la presencia de otros objetos o ruido en la escena no afectará al procesamiento.

Ilustración 22. Ejemplo interactivo de Segment Anything (Kirillov, 2025)

Ilustración 23. Ejemplo interactivo de Segment Anything (Kirillov, 2025)

4.2.5. Procesamiento

Las herramientas utilizadas para realizar el procesamiento asociado a la detección de agarres enfrascan varias librerías y funciones de programación. A continuación, se indican las dos herramientas fundamentales para el procesamiento de imágenes de la aplicación desarrollada.

4.2.5.1. NumPy

NumPy (Numerical Python) es una de las bibliotecas fundamentales para la programación en Python. Está orientada al trabajo con arrays [8] multidimensionales y al cálculo numérico eficiente, lo cual resulta esencial en aplicaciones donde se manejan grandes cantidades de datos de manera estructurada, como es el caso del procesamiento de imágenes como en este proyecto. (Harris, 2020) (Van der Walt, 2011)

NumPy proporciona una estructura de datos principal llamada **ndarray** (N-dimensional array), que permite representar vectores, matrices y tensores de cualquier dimensión. Ndarray es una clase especializada que almacena datos en bloques contiguos de memoria y permite realizar operaciones numéricas de forma vectorizada. Gracias a este formato, se pueden realizar cálculos sobre todo el conjunto de datos sin necesidad de usar bucles de Python. (Developers, 2025)

Las operaciones vectorizadas son mucho más rápidas y eficientes que los bucles tradicionales ya que están optimizadas internamente. Esto representa una gran ventaja en el rendimiento de los programas cuando se trabaja con grandes volúmenes de datos como imágenes y nubes de puntos. Y es crucial

[8] *Array (matriz):* estructura de datos que permite almacenar múltiples valores del mismo tipo en una sola variable, organizados de forma contigua en la memoria. Pueden tener una o más dimensiones y permiten realizar operaciones matemáticas vectorizadas de manera eficiente.

en aplicaciones en tiempo real, además de facilitar la estructura y legibilidad del código.
En este proyecto, la biblioteca NumPy se ha empleado para llevar a cabo diversas tareas como:

- Manipulación de datos 2D y 3D

  Los datos recogidos por el sensor RGB-D (imágenes a color y datos de profundidad) se han representado mediante *array*s de *Numpy*.

- Cálculos vectoriales y algebra lineal

  NumPy cuenta con múltiples herramientas de algebra lineal, como normalización de vectores, resolución de ecuaciones lineales, productos vectoriales o escalares, etc. En este proyecto han sido esenciales para operaciones como el cálculo de normales a una superficie definida por un plano, determinar los puntos de intersección entre planos o transformar puntos entre distintos sistemas de coordenadas mediante operaciones matriciales de traslación y rotación.

- Cálculo de dimensiones y análisis geométrico

  Una parte relevante del proyecto reside en estimar las dimensiones del objeto a coger, para ello se emplean herramientas de NumPy que permiten calcular distancias entre dos puntos del espacio, encontrar los límites máximos y mínimos de puntos en direcciones determinadas o calcular la norma de un vector.

- Máscaras y filtrado de datos

  Otro uso importante de la biblioteca es la aplicación de **máscaras booleanas** para determinar qué pixeles cumplen ciertas condiciones. Por ejemplo, se pueden aplicar máscaras para eliminar píxeles que se encuentran fuera de un rango de profundidad específico, útil para eliminar el fondo o ruidos. O para seleccionar qué puntos pertenecen al objeto y cuáles al entorno, utilizando una máscara generada con SAM en la imagen RGB y aplicando posteriormente esa máscara a los datos de la nube de puntos.

- Compatibilidad con otras bibliotecas

  NumPy resulta especialmente útil cuando se trabaja con múltiples bibliotecas ya que actúa como “lenguaje común”. Esta interoperabilidad facilita la integración de los distintos módulos evitando problemas de incompatibilidades.

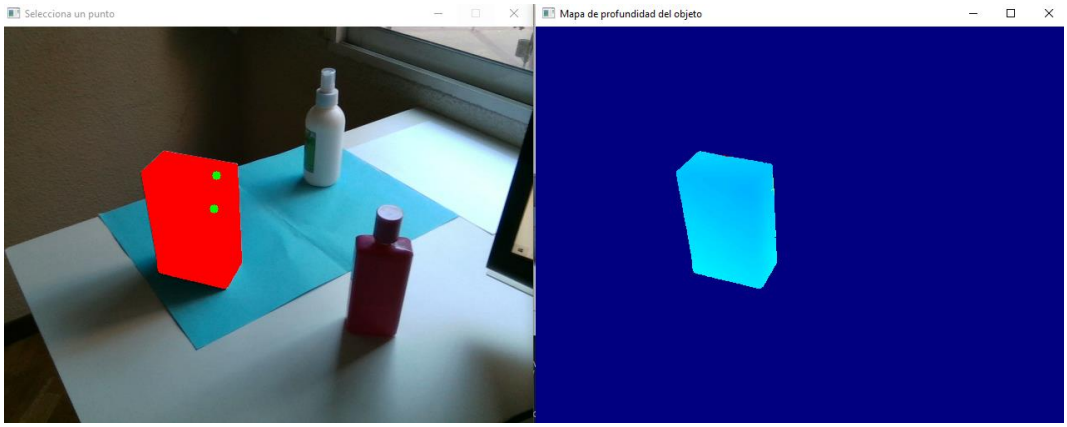

Ilustración 24. Aplicación de máscaras booleanas (Elaboración propia)

4.2.5.2. GPD

GPD (Grasp Pose Detection) es una herramienta de código abierto desarrollada en C++, diseñada para detectar automáticamente posiciones de agarre tridimensionales sobre objetos representados por nubes de puntos. (Ten Pas A. G., 2017)

En el presente proyecto, se ha empleado esta herramienta como motor principal de detección de agarres, ya que los modelos que emplean un procesamiento con herramientas estáticas presentaban graves limitaciones ante objetos no conocidos.

El funcionamiento de GPD se basa en un ejecutable que recibe como entrada un archivo .pcd con la nube de puntos del objeto y un archivo de configuración que define los parámetros del modelo. Su salida original consiste en la impresión por consola de los agarres detectados, a través de una garra virtual representada por una estructura de agarre definida por una posición 3D, tres vectores ortonormales (axis, binormal y approach) y un valor de apertura. (Ten Pas A. , 2019)

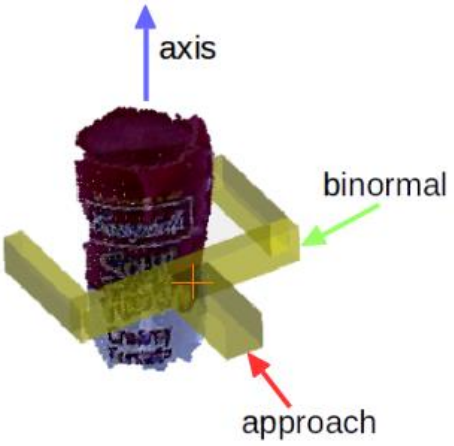

Ilustración 25. Grasp pose (Ten Pas A. , 2019)

Para integrarlo dentro del flujo de trabajo automatizado, el modelo GPD se ha tratado como una aplicación externa, compilada en un entorno Linux, que se ejecuta desde Python a través de llamadas a subprocesos. Esta herramienta se ha configurado para recibir como entrada la nube de puntos del objeto segmentado y devolver como resultados un fichero con los agarres detectados.

GPD se ha instalado y compilado en el servidor remoto *Inception,* lo que permite mantener el cliente local ligero y aprovechar los recursos computacionales del servidor. Por otro lado, esto supone la necesidad de desarrollar un sistema robusto de comunicaciones en tiempo real con el servidor para poder ejecutar el proceso de manera externa y automatizada totalmente.

Entre las ventajas de la elección de GPD frente a alternativas más complejas como GraspNet destacan su sencillez de despliegue, independencia de redes neuronales y compatibilidad directa con las nubes de puntos generadas por cámaras tipo RGB-D como la RealSense. Además, la flexibilidad del archivo de configuración permite adaptar el comportamiento del modelo a diferentes objetos y entornos robóticos no controlados sin necesidad de realizar un reentrenamiento específico de la red neuronal.

4.2.6. Comunicaciones

En el desarrollo del proyecto, uno de los aspectos clave ha sido establecer una comunicación robusta entre el sistema local de adquisición y procesamiento de imágenes y un servidor remoto destinado a ejecutar el algoritmo de detección de agarres GPD.

Esta comunicación ha permitido aprovechar los recursos computacionales del servidor *Inception,* donde se ha instalado y configurado el modelo GPD.

Esta configuración también ofrece una mayor portabilidad del sistema, permitiendo que el cliente local pueda ejecutarse desde cualquier equipo sin necesitar un hardware específico. Además, posibilita la conexión simultanea de varios clientes, lo que permite emplear el modelo de detección de agarres en distintos sistemas robóticos de características similares.

Se han implementado dos métodos diferentes de conexión al servidor: inicialmente, se creó un sistema que a través del protocolo SFTP realiza la transferencia de archivos y la ejecución remota del modelo, y posteriormente, se ha desarrollado un servidor web específicamente para recibir peticiones, ejecutar el algoritmo y devolver los resultados de manera más eficiente, segura y portable.

A continuación, se describen las herramientas empleadas en cada uno de estos métodos de comunicación, así como las tecnologías utilizadas para acceder al servidor remoto y la ejecución de subprocesos en su entorno.

4.2.6.1. Servidor remoto *Inception*

El servidor remoto *Inception* ha sido el entorno seleccionado para la instalación y ejecución del modelo GPD, debido a su mayor capacidad computacional en comparación con el equipo local disponible. Se trata de un sistema operativo **Linux,** lo cual ha permitido mayor flexibilidad en la instalación de dependencias, automatización de procesos y ejecución de scripts desde la línea de comandos.

El acceso al servidor se realiza mediante una red privada virtual (VPN) proporcionada por la universidad, que permite establecer una conexión segura externa. Una vez conectada a la red de la universidad, se puede acceder al entorno gráfico de *Inception* servidor utilizando la herramienta Escritorio Remoto de Windows**,** lo cual facilita la instalación del sistema y su supervisión, así como la gestión de archivos y verificación del algoritmo.

Al centralizar la ejecución del algoritmo GPD en un único servidor, se facilita la actualización del modelo y la gestión de recursos, así como la posibilidad de atender a múltiples solicitudes.

4.2.6.2. Comunicación mediante FTP (SFTP + SSH)

En la primera versión del sistema, se estableció comunicación con el servidor *Inception* mediante el protocolo SFTP (*Secure File Transfer Protocol*), el cual permite la transferencia segura de archivos a través de una conexión SSH[9]. Este sistema nos ofrece una solución fácil de implementar gracias a su compatibilidad con entornos Linux, permitiendo establecer una conexión remota desde Python sin necesidad de configurar servicios adicionales en el servidor.

En la parte cliente, la base de la comunicación se establece con la librería **paramiko,** una de las herramientas más utilizadas para realizar conexiones SSH y transferencias SFTP desde scripts de Python**.** Paramiko permite realizar, entre otras, según su documentación oficial (Paramiko Developers, 2023) las siguientes operaciones:

- Establecer sesiones SSH seguras autenticadas mediante usuario y contraseña o mediante una clave pública
- Transferir archivos entre el cliente local y el servidor remoto utilizando SFTP
- Ejecutar comandos remotos en el servidor Linux directamente desde Python
- Supervisar la salida y los errores generados por los procesos ejecutados remotamente
- Automatizar tareas de gestión remota como crear carpetas o eliminar archivos.

[9] SSH (Secure Shell): protocolo de red criptográfico para operar servicios de red de forma segura sobre una red.

En conclusión, esta herramienta permite automatizar el flujo completo de trabajo completamente desde el cliente local, enviando la nube de puntos del objeto, ejecutando en remoto el modelo GPD y recibiendo los resultados para su posterior procesamiento.

Esta implementación, se complementó con el uso del módulo **os,** que forma parte de la biblioteca estándar de Python y proporciona herramientas para interactuar mediante un script dentro del propio sistema operativo local. (Python Software Foundation, 2024) Las funcionalidades más relevantes de esta herramienta son:

- Gestión de rutas y nombres de archivos
- Creación y eliminación de carpetas y archivos temporales
- Lectura y escritura de ficheros locales utilizados como entrada o salida del sistema
- Verificar la existencia de rutas y archivos

Estas herramientas han resultado clave tanto para la programación de la parte cliente como de la parte servidor. Donde se desarrolla un script de Python para gestionar el flujo del servidor y lanzar el algoritmo GPD.

Una de las bibliotecas en las que se apoya este script es **sys,** que proporciona funciones y variables utilizadas para manipular aspectos del entorno de ejecución del sistema como:

- Lectura de argumentos por la línea de comandos, en este caso la ruta al archivo .pcd recibido por el cliente
- Validar la correcta introducción de parámetros antes de ejecutar el algoritmo
- Finalizar la ejecución del script en caso de error, informando al usuario con distintos mensajes de error personalizados

  (Python Software Foundation, 2024)

Por otra parte, se ha empleado la biblioteca **subprocess** para la ejecución del modelo de detección de agarres en el entorno Linux del servidor. (Python Software Foundation, 2024) Esta herramienta permite lanzar procesos externos desde Python y controlar completamente su ejecución, permite:

- Ejecutar el modelo GPD con los parámetros correspondientes
- Definir el directorio de trabajo desde el cual ejecutar el modelo
- Capturar la salida del proceso, mostrando los mensajes y resultados generados por el modelo
- Detectar y mostrar posibles errores durante la ejecución

Gracias a estas herramientas, se permitió desarrollar una solución funcional, donde el cliente realiza el preprocesamiento y envía datos al servidor, que ejecuta el modelo GPD y devuelve los resultados correspondientes.

4.2.6.3. Comunicación mediante servidor web

Tras validar el sistema de comunicación por SFTP, se optó por desarrollar una segunda opción de comunicación basada en un servidor web con el objetivo de simplificar el entorno, aumentar la escalabilidad y la seguridad y el control del servidor.

Esta solución elimina la conexión SSH directa y la transferencia de archivos mediante SFTP, sustituyéndolos por una comunicación cliente-servidor mediante peticiones HTTP[10].

El servidor web fue implementado en *Inception* utilizando la biblioteca **Flask.** Esta herramienta permite construir una interfaz HTTP que recibe solicitudes POST [11] con los datos necesarios para ejecutar el modelo GPD. (Flask Documentation Team, 2024) Las principales ventajas de Flask en este contexto son:

- Facilidad de creación de rutas que actúan como puntos de entrada para distintos tipos de solicitud (/procesar_pcd).
- Integración directa con scripts en Python, permitiendo ejecutar procesos y manejar archivos.
- Uso de utilidades propias como **send_file** y **jsonify** para devolver.
- Ligereza y compatibilidad con sistemas Linux.

En el servidor, se utiliza también la biblioteca **tempfile,** que permite crear directorios temporales de forma segura para almacenar la ruta a la nube de puntos enviada por el cliente. (Python Software Foundation, 2025) Una vez guardado el archivo, se lanza el script mencionado anteriormente que utiliza el módulo **subprocess** para ejecutar el GPD. También se siguen utilizando las librerías **os** y **sys** mencionadas anteriormente para la gestión de rutas.

Por otro lado, se simplifica el lado cliente, el envío de la nube de puntos y la recepción de resultados se realiza mediante la biblioteca **request,** que nos ofrece distintas funcionalidades:

- Establecer peticiones POST hacia el servidor Flask con los archivos necesarios
- Interpretar la respuesta del servidor, descargando los archivos generados como el csv con los agarres detectados.

[10] HTTP (HyperText Transfer Protocol): protocolo de comunicación que permite la transferencia de datos entre un cliente y un servidor. Es la base de comunicación en la web.
[11] POST: uno de los métodos del protocolo HTTP. Se utiliza para enviar datos desde el cliente al servidor como archivos o estructuras JSON (JavaScript Object Notation).

Gracias a esta arquitectura basada en peticiones HTTP, se eliminan las conexiones persistentes y se reduce la complejidad del servicio anterior. Al tratarse de un servicio web, el dueño del servidor decide cuando tenerlo activo y los clientes simplemente se deben conectar a él, sin tener que compartir las contraseñas del servidor, mejorando la seguridad del sistema. Además, se elimina también la necesidad de conectarse a la red del servidor mediante una VPN, lo que simplifica el uso del sistema desde ubicaciones externas. Finalmente, esta solución representa un avance en la escalabilidad del proyecto ya que se pueden programar distintas rutas de entrada con diferentes solicitudes para diversos procesos. (Reitz, 2023)

5. Desarrollo del sistema de visión

Tras realizar un estudio del estado de la tecnología de los sistemas de visión artificial para detección de agarres para guiado robótico. En el presente TFG se comenzó por un intento de aplicar métodos más tradicionales para realizar la aplicación propuesta.

Esta decisión ha sido fruto de observar las posibilidades computacionales del equipo que se disponía. Ya que los métodos tradicionales habitualmente requieren de menor coste computacional al prescindir de redes neuronales, para las cuales se necesita de características electrónicas más potentes como una unidad de procesamiento gráfico (GPU).

Tras estudiar, diseñar y desarrollar un método de detección de agarres mediante búsqueda de un patrón geométrico y aplicarlo, fuimos conscientes de las limitaciones que presentaba. Este sistema podía ser utilizado en entornos controlados, con objetos muy similares, pero a la hora de escalarlo a objetos desconocidos, más complejos, con distintas formas y parcialmente visibles el sistema requería el estudio de cada posibilidad o tipo de objeto.

Por este motivo, se comenzó una investigación de distintos métodos de agarre, más complejos y que contaban con mayor adaptabilidad y escalabilidad ante objetos desconocidos, entre ellos se encontraba GraspNet.

GraspNet se consideró como una opción muy atractiva por su capacidad de generar múltiples poses de agarre en escenas reales con gran variabilidad. Prometía resultados prometedores en escenarios complejos y con objetos de formas complejas. (Fang, Wang, Gou, & Lu, 2020). Sin embargo, a medida que se profundizó en su implementación, surgieron ciertas limitaciones que dificultaban su integración en este proyecto.

En primer lugar, GraspNet requiere una gran infraestructura computacional, incluyendo una GPU de alto rendimiento, lo cual excedía las capacidades del equipo disponible. Además, su integración implicaba preparación previa más compleja y distaba en gran medida del trabajo realizado hasta el momento, incluyendo formatos de entrada específicos y posibles conflictos entre librerías.

Ante estas dificultades, se optó por estudiar la posibilidad de emplear **GPD (Grasp Pose Detection),** un sistema más ligero y modular que se ajustaba a las condiciones del proyecto. GPD permite generar poses de agarre directamente a partir de nubes de puntos, sin necesidad de un proceso externo de procesamiento. (Andreas ten Pas, 2017)

No obstante, GPD también requería de ciertas características computacionales para su correcto funcionamiento, por lo que se decidió instalar en un servidor remoto y realizar allí el procesamiento que requiere generar las poses de

agarre. Además, su diseño facilitaba este proceso y no presentó grandes problemas de instalación ni comunicación.

Se ha diseñado un programa que realiza un preprocesamiento de las imágenes obtenidas de la cámara, dando libertad al usuario para decidir el objeto que se desea agarrar y pudiendo ejecutar la aplicación desarrollada de manera tradicional o mediante el modelo de detección de agarres GPD, con comunicación SSH y SFTP o utilizando el servidor web.

De manera local, se realiza la interacción con el usuario, la adquisición de datos, el preprocesamiento de estos, incluyendo la segmentación del objeto elegido, la visualización de las distintas operaciones y el procesamiento de los agarres generados. Por otro lado, en el servidor remoto se lleva a cabo la ejecución del módulo GPD, encargado de generar las poses de agarre y cargar los resultados en un fichero extraíble.

5.1. Arquitectura del sistema desarrollado

Con el objetivo de mejorar la legibilidad, mantenibilidad y eficiencia del sistema desarrollado se ha optado por dividir el código en distintos módulos funcionales. Esta arquitectura modular permite una organización más clara del flujo de trabajo, facilitando su implementación, futuras modificaciones y reutilización de componentes comunes.

Además, algunos de estos módulos, como los encargados de adquisición de imágenes de la cámara o la segmentación de los objetos son utilizados, tanto en el enfoque tradicional como en la implementación con GPD, lo que evita el uso de funciones duplicadas. A continuación, se muestra un diagrama de flujo del sistema desarrollado y cómo se relacionan los diferentes módulos entre sí.

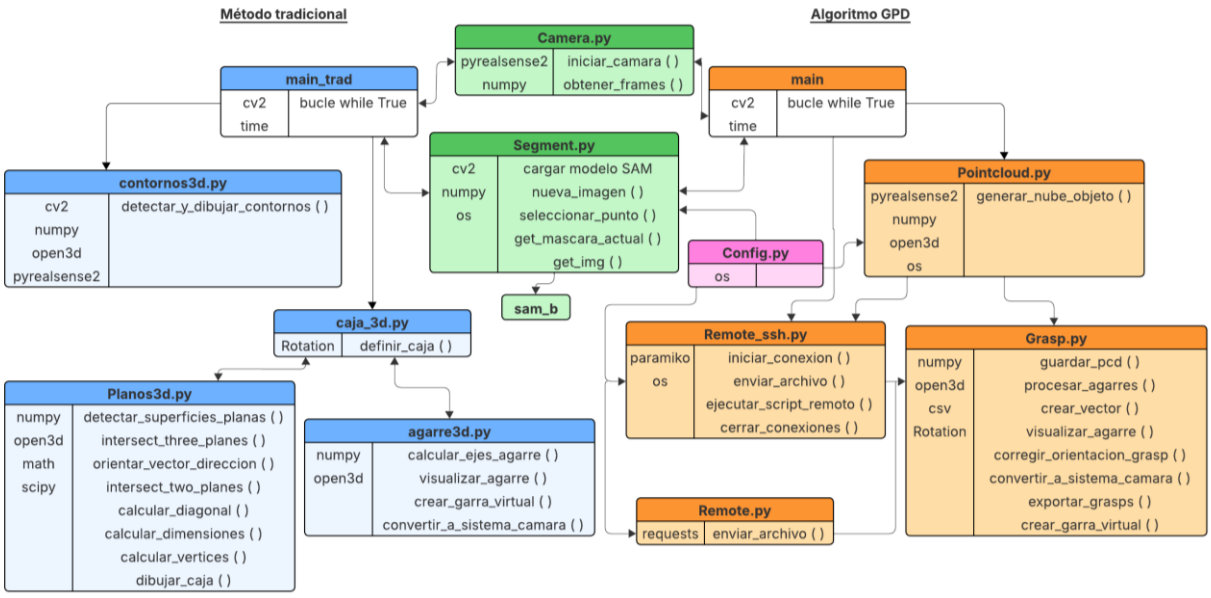

Ilustración 26. Esquema de la arquitectura de la aplicación desarrollada. (Elaboración propia)

Los sistemas desarrollados se desglosan en varios módulos. Los cuales podemos clasificar en tres grupos, según el sistema en el que se emplean. A continuación, se describen los módulos y su propósito principal

- **Módulos compartidos por los enfoques tradicional y GPD**
  - Camera.py: encargado de la configuración de la cámara y adquisición de imágenes y mapas de profundidad en tiempo real
  - Segment.py: realiza la segmentación del objeto mediante el modelo SAM y genera una máscara binaria del mismo.
- **Módulos particulares del sistema tradicional**

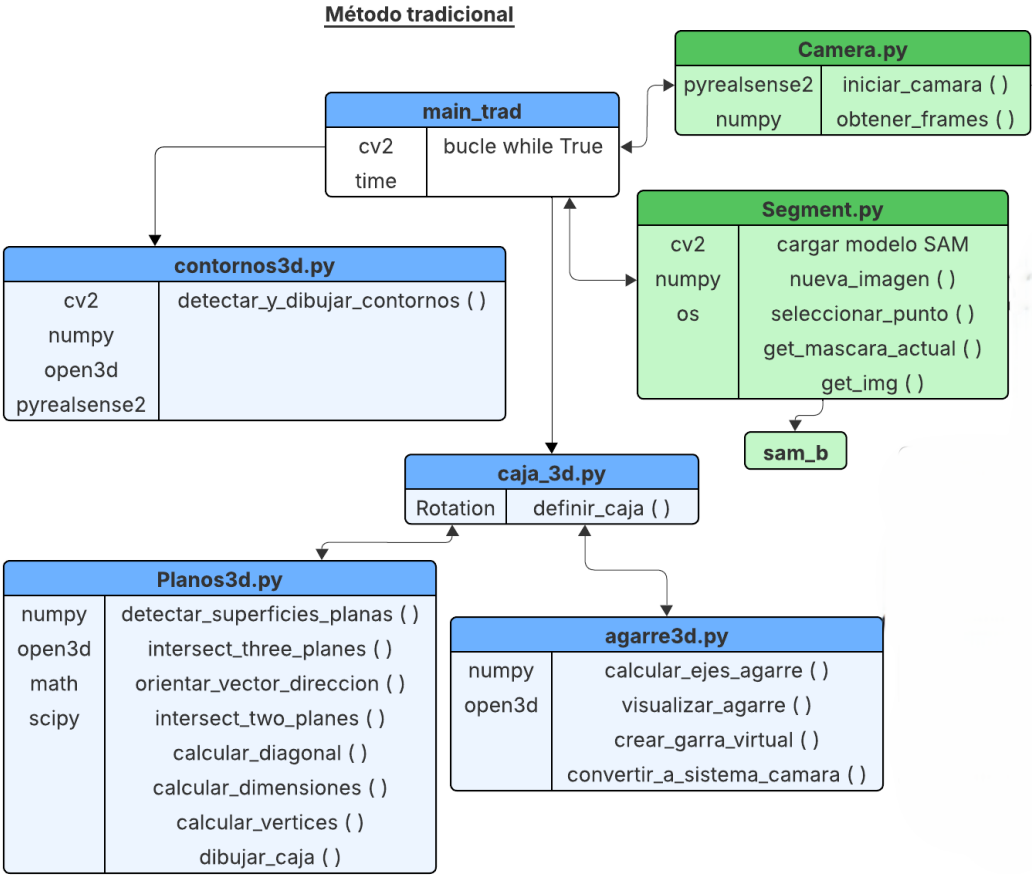

Ilustración 27. Esquema módulos sistema tradicional (Elaboración propia)

  - Contornos3d.py: detecta y dibuja los contornos del objeto sobre la nube de puntos segmentada
  - Caja3d.py: ajusta una caja tridimensional alrededor del objeto, basada en las características geométricas encontradas.
  - Planos3d.py: conjunto de funciones para detectar superficies planas en el objeto 3D y encontrar sus características geométricas
  - Agarre3d.py: calcula y visualiza un agarre para la caja virtual encontrada.
  - Main_tradicional.py: control principal del flujo de ejecución para el sistema tradicional.

- Módulos particulares del sistema GPD

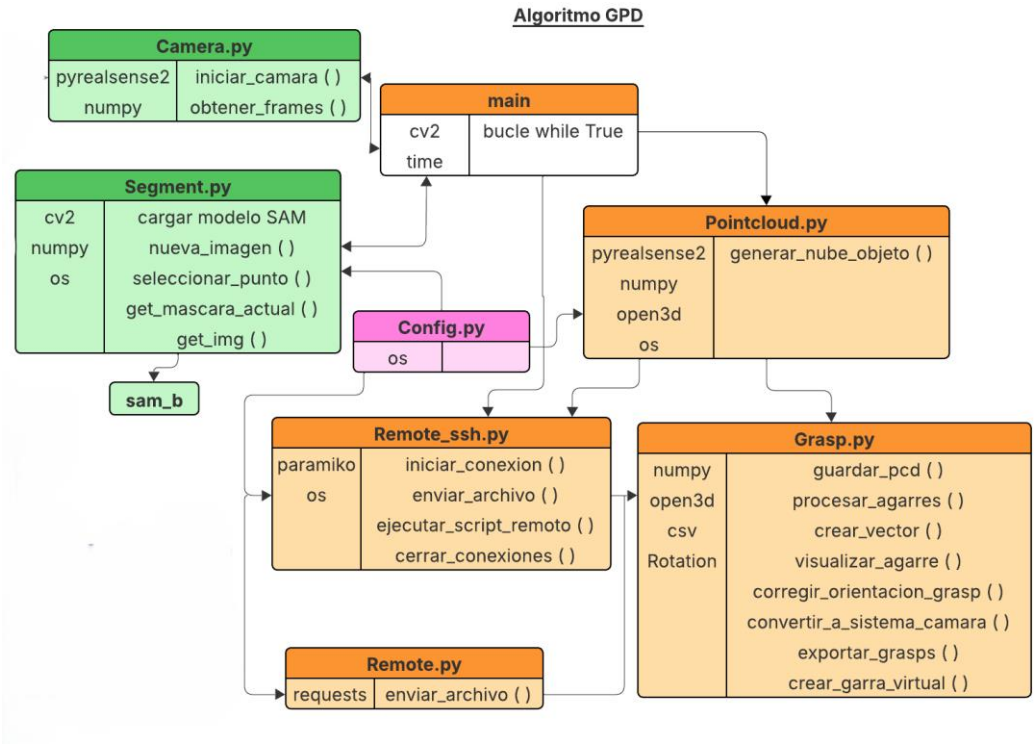

Ilustración 28. Esquema de módulos del sistema GPD (Elaboración propia)

- Config.py: define las rutas locales y del servidor y los parámetros necesarios para su configuración
- Pointcloud.py: genera y guarda la nube de puntos del objeto
- Remote.py: gestiona la conexión con el servidor remoto a través de HTTP y la transmisión y recepción de datos
- Remote_ssh.py: módulo alternativo a remote.py que ofrece una conexión basada en SSH y SFTP
- Grasp.py: interpreta, procesa y visualiza los agarres generados por GPD
- Main_GPD.py: control principal del flujo de ejecución del sistema basado en GPD

Dentro de los módulos del sistema basado en GPD, encontramos, en el servidor remoto los siguientes archivos:

- Servicio_web.py: programa que genera el servicio web, configurado para recibir solicitudes HTTP
- Script_gpd.py: programa que se encarga de lanzar el detector de agarres con la nube de puntos del objeto recibida

- **Archivos auxiliares del sistema**
  - README.md: contiene instrucciones de uso y dependencias del sistema
  - Requirements.txt: lista de dependencias necesarias para ejecutar los módulos
  - Sam_b.pt: archivo de pesos del modelo SAM utilizado para la segmentación, presente e imprescindible en ambos entornos.

Nota: Los fragmentos de código mostrados a continuación se distinguen por su formato: aquellos con fondo negro corresponden a programas ejecutados en el equipo local, mientras que los que presentan fondo blanco se refieren a programas ejecutados en el servidor remoto.

5.2. Preprocesamiento: Adquisición de imágenes y segmentación

Tanto el enfoque tradicional como el basado en GPD comparten una etapa inicial de preprocesamiento compuesta por dos módulos: **camera.py** y **segment.py.** Estos módulos permiten capturar la información visual de la escena y elegir y aislar el objeto de interés.

5.2.1. Camera.py: captura y alineación de imágenes RGB-D

A la hora de diseñar un sistema de agarre de objetos lo primero con lo que se debe contar es con un sistema de adquisición de imágenes en tiempo real del objeto que se desea agarrar y su entorno.

En este proyecto, como hardware de adquisición se establece una cámara RGB-D de Intel RealSense. En este módulo, basado en la librería *pyrealsense2,* permite una adquisición continua de imágenes en tiempo real y la sincronización automática con el mapa de profundidad asociado.

El módulo nos ofrece dos funciones principales:

Inicialización de la cámara:

La **función** ***iniciar_camara()*** se encarga de configurar y poner en marcha el pipeline[12] de RealSense. En este proceso de definen los siguientes parámetros:

- Resolución de salida: se establece una resolución de 640x480 píxeles tanto en la imagen de color como en el mapa de profundidad.

  Se establecen resoluciones menores a las que ofrece la cámara para reducir el consumo del procesamiento posterior, acelerar la visualización y evitar cuellos de botella.

[12] *Pipeline:* componente que configura y gestiona automáticamente la cámara, permitiendo capturar imágenes de color y profundidad de forma sincronizada y accesible para su procesamiento.

- Frecuencia de muestreo: se establecen 30 fotogramas por segundo (FPS) para recibir una visualización fluida e interacción en tiempo real suficiente.
- Formatos:
  - Stream de profundidad: z16 (16 bits por píxel)
  - Stream de color: bgr8 (8 bits por canal en formato BGR)

  Estos formatos han sido elegidos para asegurar la compatibilidad con la biblioteca OpenCV y Open3d que se emplean en el procesamiento de las imágenes capturadas.

Una vez configurada la cámara, el *pipeline* se inicia y se crea un objeto *align*, este será el responsable de alinear el mapa de profundidad con la imagen de color. Esta etapa es esencial ya que, dado el hardware de la cámara, los sensores RGB y de profundidad están ligeramente desplazados. El alineamiento corrige esta desviación y garantiza que cada píxel de la imagen RGB tenga su correcto valor de profundidad. Además, el objeto *profile* guarda la información activa de cómo está configurado el *pipeline* y permite acceder a detalles del sensor.

```
def iniciar_camara():
    pipeline = rs.pipeline()
    config = rs.config()
    config.enable_stream(rs.stream.depth, 640, 480, rs.format.z16, 30)
    config.enable_stream(rs.stream.color, 640, 480, rs.format.bgr8, 30)
    profile = pipeline.start(config)
    align = rs.align(rs.stream.color)
    return pipeline, align, profile
```

Ilustración 29. Módulo camera.py: Función iniciar_camara() (Elaboración propia)

Captura de imágenes

La **función *obtener_frames()*** tiene como propósito capturar un conjunto de *frames* de la cámara, alinearlos y devolver dos imágenes, una de color y otra de profundidad en formato *array* de *Numpy*.

Esta función recibe el pipeline y el objeto de alineación de la cámara, establecidos al inicio del bucle y obtiene *frames* en tiempo real, alineados y listos para ser procesados por otros módulos.

```
def obtener_frames(pipeline, align):
    frames = pipeline.wait_for_frames()
    aligned = align.process(frames)
    depth = aligned.get_depth_frame()
    color = aligned.get_color_frame()
    if not depth or not color:
        return None, None
    return np.asanyarray(depth.get_data()), np.asanyarray(color.get_data())
```

Ilustración 30. Módulo camera.py: Función obtener_frames() (Elaboración propia)

La importancia de este módulo reside en su capacidad de reutilización, permitiendo emplear estas funciones para obtener imágenes de manera

sencilla desde cualquier flujo de trabajo y en el momento que sea necesario. Al encapsular la configuración de los parámetros de la cámara en este punto, aseguramos la estabilidad del sistema de adquisición evitando que otros módulos o subprocesos los modifiquen, lo que podría provocar incompatibilidades o desalineamientos entre las imágenes de profundidad y color.

Además, se estandariza el retorno de imágenes, como *arrays* de *Numpy*, para poder ser manipuladas con herramientas estándar como OpenCV, lo que simplifica tareas posteriores de visualización, segmentación o reconstrucción 3D.

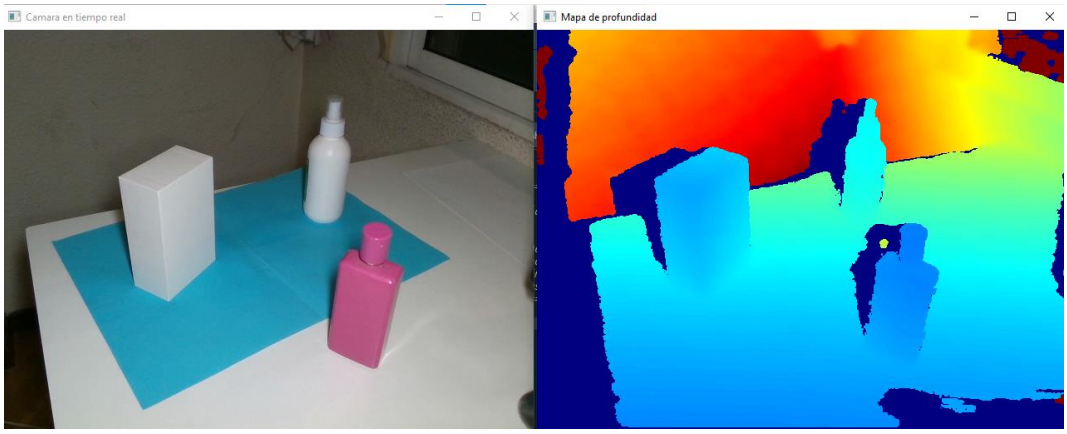

Ilustración 31. Imagen RGB y mapa de profundidad obtenido de la cámara en tiempo real (Elaboración propia)

5.2.2. Segment.py: Segmentación del objeto

El módulo **segment.py** se encarga de realizar la segmentación interactiva del objeto que desea agarrar el usuario, utilizando el modelo **SAM,** mencionado anteriormente en el apartado "herramientas".

Este archivo incluye tanto la carga del modelo SAM como la interacción con el usuario, basada en la imagen a color capturada, permitiendo la selección precisa del objeto a través de una interfaz que se actualiza con los movimientos del usuario, basada en OpenCV.

Al iniciar el script, se verifica la existencia del archivo de pesos preentrenados **sam_b.pt.** En este caso, se ha decidido utilizar la versión **vit_b** del modelo, se trata de la versión "base", la más ligera pero que ofrece buena relación entre rendimiento y velocidad. Este modelo se pasa a la clase *SamPredictor***,** creando un predictor interactivo que nos permitirá generar máscaras a partir de la imagen y puntos seleccionados.

Una vez cargado el modelo, se establecen ciertas variables globales para almacenar los estados de la segmentación, lo que permitirá la interacción con otros módulos de manera sencilla.

```
# Verificar modelo
if not os.path.exists(SAM_MODEL_PATH):
    raise FileNotFoundError(f"Modelo SAM no encontrado en {SAM_MODEL_PATH}")

sam = sam_model_registry["vit_b"](checkpoint=SAM_MODEL_PATH)
predictor = SamPredictor(sam)

# Variables globales para interfaz gráfica
puntos_seleccionados = []
mascara_actual = None
imagen_base = None
depth_image_s = None
```

Ilustración 32. Módulo segment.py: Verificación modelo SAM y variables globales (Elaboración propia)

Este módulo tiene dos funciones principales:

Nueva_imagen():

Al pulsar la tecla “s” en la ventana principal (main.py) se llama a esta función, la cual recibe la imagen a color y su mapa de profundidad actual alineados y los guarda para realizar la segmentación del objeto sobre estos. De esta manera, realizamos una “captura” para la interacción con el usuario.

En esta función, guardamos las imágenes en variables globales, limpiamos la lista de puntos seleccionados por el usuario, lo cual permite la reutilización del módulo en el mismo ciclo de trabajo. Inicializamos el predictor con la imagen capturada y creamos una ventana con OpenCV a la que asignamos la función de *callback* que nos permitirá gestionar los clics del usuario.

```
def nueva_imagen(imagen, depth_image):
    global imagen_base, puntos_seleccionados, mascara_actual, depth_image_s
    imagen_base = imagen.copy()
    depth_image_s = depth_image
    puntos_seleccionados.clear()
    mascara_actual = None

    predictor.set_image(imagen_base)
    cv2.namedWindow("Selecciona un punto")
    cv2.setMouseCallback("Selecciona un punto", seleccionar_punto)
    cv2.imshow("Selecciona un punto", imagen_base)
```

Ilustración 33. Módulo segment.py: Función nueva_imagen() (Elaboración propia)

La decisión de tomar una captura estática de la imagen para segmentar los objetos y no realizar la segmentación en tiempo real se tomó para reducir el tiempo de procesamiento. De esta manera, el predictor se lanza una única vez y tiene que procesar solo una imagen, si se quiere realizar la segmentación en tiempo real se requiere un equipo con muchas mayores capacidades computacionales. Cuando se intentó realizar esto en el equipo actual, el predictor no llegaba a procesar la imagen a tiempo antes de recibir un nuevo *frame* de la cámara.

Seleccionar_punto():

Esta es la función de *callback* asociada a la ventana de selección de puntos del objeto. Se activa cuando el usuario hace clic izquierdo sobre la imagen, y almacena las coordenadas del punto en la lista de puntos seleccionados. Estos puntos se establecen como positivos para indicar al modelo de segmentación que forman parte del objeto que se quiere aislar.

Cada vez que el usuario selecciona un punto, se activa el predictor y crea una máscara de segmentación basada en ese punto y los anteriores. Guarda la máscara creada y modifica la imagen original, pintando de ojo el objeto segmentado y en verde los puntos seleccionados.

De esta manera, se establece una interacción con el usuario, permitiéndole a este visualizar en todo momento la segmentación del objeto que ha realizado, pudiendo seleccionar otro punto si el objeto se encuentra incompleto o modificar la selección si se ha producido algún error.

```python
def seleccionar_punto(event, x, y, flags, param):
    global puntos_seleccionados, mascara_actual, imagen_base

    if event == cv2.EVENT_LBUTTONDOWN:
        puntos_seleccionados.append((x, y))
        print(f"Puntos seleccionados: {puntos_seleccionados}")

        puntos_np = np.array(puntos_seleccionados)
        etiquetas = np.ones(len(puntos_np))

        try:
            mascaras, _, _ = predictor.predict(
                point_coords=puntos_np,
                point_labels=etiquetas,
                multimask_output=False
            )
            mascara_actual = mascaras[0]

        except Exception as e:
            print(f"Error al predecir máscara: {e}")
            return

        imagen_vis = imagen_base.copy()
        imagen_vis[mascara_actual] = [0, 0, 255]
        for p in puntos_seleccionados:
            cv2.circle(imagen_vis, p, 5, (0, 255, 0), -1)

        cv2.imshow("Selecciona un punto", imagen_vis)
```

Ilustración 34.Módulo segment.py: Selección de puntos y segmentación del objeto (Elaboración propia)

En este módulo, además, se han incluido dos pequeñas funciones que sirven de retorno de la imagen que ha sido capturada y de la máscara procesada por SAM. Estas funciones se establecen para facilitar el acceso a estos objetos desde otros módulos del programa en los que se puedan necesitar.

```python
def get_mascara_actual():
    return mascara_actual

def get_img():
    return imagen_base, depth_image_s
```

Ilustración 35. Módulo segment.py: Funciones retorno imagen y macara actuales (Elaboración propia)

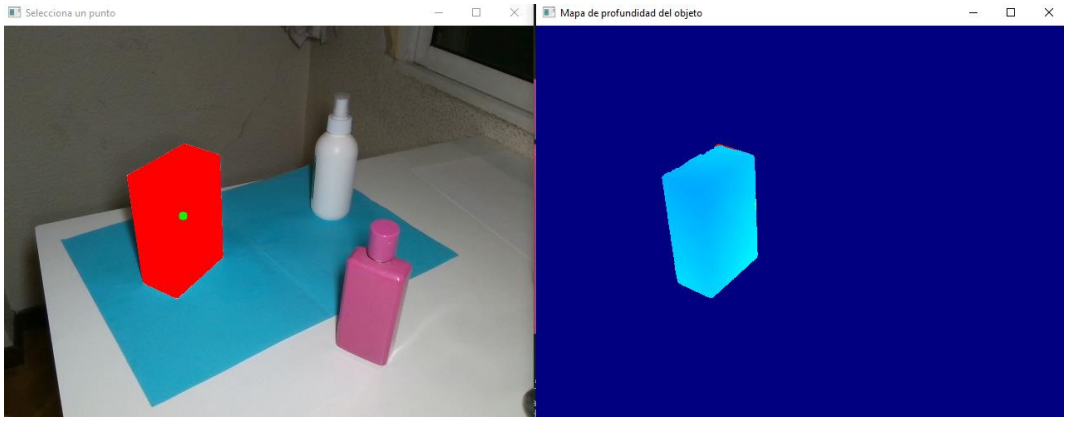

Ilustración 36. Selección y segmentación de objeto a partir de un punto (RGB + mapa de profundidad) (Elaboración propia)

5.3. Flujo de trabajo principal

El flujo de trabajo es controlado por la aplicación **main.py.** Esta función define cómo y cuándo realizar la adquisición de imágenes, segmentación y procesamiento 3D del objeto.

Al ejecutar el script, automáticamente se inicia la cámara, utilizando la función anteriormente descrita para ello. A partir de ese momento, el script se basa en un bucle de manera que en cada interacción se capturan y muestran imágenes de color y profundidad alineadas.

Si el usuario presiona “s”, se activa el módulo de segmentación, capturando el *frame* actual y permitiendo al usuario la segmentación del objeto que desee dentro de la imagen RGB. Además, con la máscara generada sobre el objeto en la imagen a color se realiza la segmentación del mapa de profundidad del propio objeto.

Una vez el usuario se encuentra conforme con la segmentación realizada, si presiona la tecla “g”, se guarda la imagen del objeto segmentado y se lanza el proceso de reconstrucción 3D del objeto mediante la generación de una nube de puntos basada en la máscara generada previamente.

Para guardar las imágenes sin sobrescribirlas se emplea la librería **time,** que nos permite extraer la fecha en que se ejecuta el programa, incluyendo la hora y el minuto. De esta manera, podemos conservar todas las imágenes y nubes de punto de manera unitaria, y poder recurrir a ellas si fuera necesario.

Tras realizar todo el proceso de generación de agarres del objeto, la cámara vuelve a mostrarnos imágenes en tiempo real, teniendo la capacidad de volver a seleccionar un *frame* y generar agarres para otro objeto de la escena, o utilizar incluso la imagen capturada en un inicio.

Se permite la salida del bucle en cualquier momento pulsando “q” o ESC, cerrando todas las conexiones y recursos utilizados.

```python
# Inicializar cámara
pipeline, align, profile = iniciar_camara()

try:
    while True:
        # Capturar imagenes de color y profundidad en tiempo real
        depth_image, color_image = obtener_frames(pipeline, align)
        if depth_image is None or color_image is None:
            continue

        key = cv2.waitKey(1)

        # Capturar escena al pulsar 's' y segmentar objeto
        if key & 0xFF == ord('s'):
            nueva_imagen(color_image, depth_image)

        # Obtener máscara actual y visualización de profundidad del objeto segmentado
        mask = get_mascara_actual()
        color_img, depth_img = get_img()

        if mask is not None and depth_img is not None:
            depth_objeto = depth_img.copy()
            depth_objeto[~mask] = 0
            color_map = cv2.applyColorMap(cv2.convertScaleAbs(depth_objeto, alpha=0.15), cv2.COLORMAP_JET)
            cv2.imshow("Mapa de profundidad del objeto", color_map)

        # Mostrar cámara y mapa de profundidad en tiempo real
        cv2.imshow("Camara en tiempo real", color_image)
        depth_colormap = cv2.applyColorMap(cv2.convertScaleAbs(depth_image, alpha=0.15), cv2.COLORMAP_JET)
        cv2.imshow("Mapa de profundidad", depth_colormap)

        # Generar y procesar nube de puntos del objeto segmentado al pulsar 'g'
        if key & 0xFF == ord('g'):
            mask = get_mascara_actual()
            if mask is not None:
                objeto = cv2.bitwise_and(color_img, color_img, mask=mask.astype('uint8'))
                timestamp = time.strftime("%m%d_%H%M")
                cv2.imwrite(f"nubes_pcd/foto_{timestamp}.png", objeto)
                print(f"Imagen del objeto guardada como foto_{timestamp}.")
                generar_nube_objeto(mask, depth_img, color_img, profile, timestamp)

        # Salir del programa al pulsar 'q' o ESC
        if key & 0xFF == ord('q') or key == 27:
            break

finally:
    #cerrar_conexiones()
    pipeline.stop()
    cv2.destroyAllWindows()
```

Ilustración 37. Módulo main.py: Control del flujo de trabajo (Elaboración propia)

El flujo de trabajo mostrado en Ilustración 37, se ha implementado tanto para el desarrollo de un sistema de visión tradicional, basado en parámetros geométricos como para la implementación de aprendizaje profundo con GPD. Para utilizar el método tradicional debe ejecutarse el módulo **main_trad.py,** si por lo contrario se quiere seleccionar el detector GPD debe ejecutarse el módulo **main.py.**

5.4. Método tradicional

En el desarrollo de un método tradicional de detección de agarres se han programado varias funciones, distribuidas en 4 módulos de trabajo en Python, de manera que seguir el método y ampliarlo pueda resultar sencillo.

El objetivo de este método consiste en reconstruir una forma geométrica virtual que se ajuste al volumen y orientación del objeto que se desea agarrar. De esta manera, se preestablecen posiciones de agarre óptimas para cada tipo de objeto virtual y se adaptan al objeto real.

Este método particularmente ha sido diseñado para objetos cuya forma se aproxima a prismas rectangulares, es decir, cuerpos con caras planas y ortogonales entre sí. El objetivo del sistema es detectar las superficies del objeto y construir a partir de ellas una caja virtual ajustada sobre la que se realiza el análisis de agarres.

El método de detección de agarres utilizado ha sido el siguiente:

- Tras realizar un preprocesamiento del objeto con SAM y obtener una máscara binaria del mismo se estudian sus contornos en 2D y se proyectan en una nube de puntos que nos muestra el objeto en tres dimensiones
- Sobre la nube de puntos del objeto obtenida de la operación anterior se detectan los tres planos principales (visibles), suponiendo estos como las caras del objeto
- Posteriormente se calculan las aristas del objeto, calculando la intersección de los tres planos principales.
- A su vez, se calculan las diagonales del objeto para determinar sus dimensiones.
- Finalmente, con las aristas y dimensiones calculadas se crea una caja virtual, sobre la cual, se detectan y visualizan agarres viables, teniendo en cuenta el tamaño del gripper que poseemos.

5.4.1. Detección de los contornos del objeto

Para obtener una representación más precisa del objeto segmentado, se ha desarrollado una función que permite identificar sus contornos visibles y proyectarlos posteriormente en el espacio tridimensional. Esta función se ha implementado en un módulo llamado **contornos3d.py** y recibe como argumentos esenciales la imagen RGB, el mapa de profundidad, la máscara binaria del objeto y los parámetros de calibración de la cámara.

En primer lugar, se aplica la máscara binaria sobre la imagen RGB para aislar completamente el objeto del entorno. Posteriormente, se convierte la imagen resultante a escala de grises para aplicar el detector de bordes de *Canny.* Los contornos encontrados se dibujan sobre la imagen original y sobre el mapa de profundidad mediante la función *applyColorMap* de OpenCV, de esta manera podemos observarlos sobre el objeto para comprobar su correcta detección.

```python
def detectar_y_dibujar_contornos(imagen_rgb, depth_image, mascara, profile):
    if mascara is None:
        print("Máscara no válida para contornos.")
        return

    # Aplicar máscara
    mask_u8 = mascara.astype(np.uint8)
    img_obj = cv2.bitwise_and(imagen_rgb, imagen_rgb, mask=mask_u8)

    # Canny
    gris = cv2.cvtColor(img_obj, cv2.COLOR_BGR2GRAY)
    bordes = cv2.Canny(gris, 100, 200)
    if np.sum(bordes) == 0:
        print("No se detectaron bordes.")
        return

    # Dibujar contornos
    contornos, _ = cv2.findContours(bordes, cv2.RETR_LIST, cv2.CHAIN_APPROX_SIMPLE)
    img_cont = imagen_rgb.copy()
    cv2.drawContours(img_cont, contornos, -1, (0, 255, 0), 2)
    cmap = cv2.applyColorMap(cv2.convertScaleAbs(depth_image, alpha=alfa), cv2.COLORMAP_JET)
    cmap_cont = cmap.copy()
    cv2.drawContours(cmap_cont, contornos, -1, (0, 255, 0), 2)
    cv2.imshow("Contornos en RGB", img_cont)
    cv2.imshow("Contornos en Mapa de Profundidad", cmap_cont)
```

Ilustración 38. Módulo contornos3d.py: Detección de contornos 2D. (Elaboración propia)

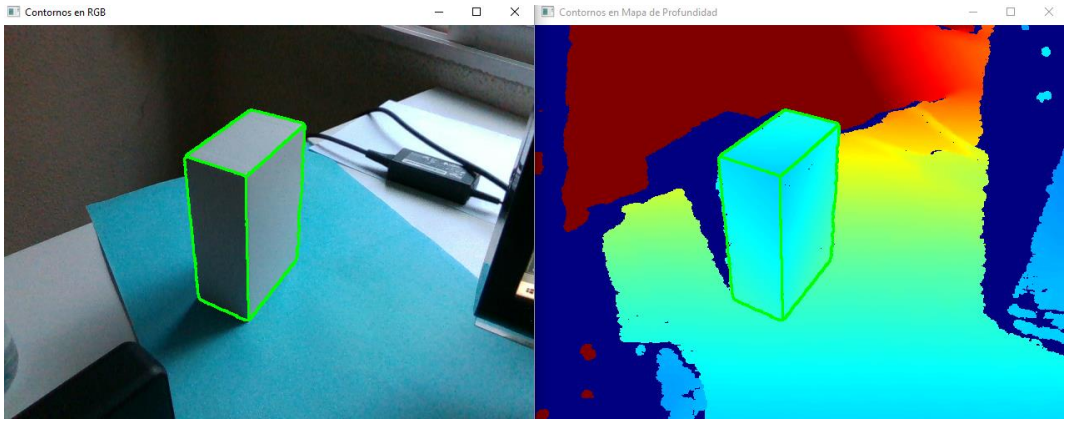

Ilustración 39. Contornos detectados en 2D (RGB + mapa de profundidad). (Elaboración propia)

A continuación, se realiza la reconstrucción 3D del objeto. Para ello, se extrae la información de intrínsecos y escala de profundidad de la cámara. A través de estos parámetros podemos proyectar píxeles con profundidad real desde 2D a coordenadas 3D en el sistema de referencia de la cámara.

Para realizar esta operación se crea un bucle que recorre todos los píxeles del mapa de profundidad y los que se encuentran dentro de la máscara binaria son proyectados y guardados en un array para su posterior procesamiento. Además, durante esta operación se distingue entre los puntos que pertenecen al interior del objeto y los que pertenecen al borde de este, de manera que obtenemos dos grupos de puntos proyectados. Asimismo, se crea un tercer array con los colores de los puntos encontrados, para realizar la representación 3D del objeto a color.

```
# 3D
intrin = profile.get_stream(rs.stream.depth).as_video_stream_profile().get_intrinsics()
scale = profile.get_device().first_depth_sensor().get_depth_scale()

points, colors, edge_points = [], [], []
for v in range(depth_image.shape[0]):
    for u in range(depth_image.shape[1]):
        if not (mascara[v, u] or bordes[v, u]):
            continue
        d = depth_image[v, u]
        if d == 0:
            continue
        pt = rs.rs2_deproject_pixel_to_point(intrin, [u, v], d * scale)
        pt[1] *= -1; pt[2] *= -1
        if bordes[v, u]: edge_points.append(pt)
        else:
            points.append(pt)
            colors.append(imagen_rgb[v, u][::-1] / 255.0)

points = np.array(points)
colors = np.array(colors)
edge_points = np.array(edge_points)
```

Ilustración 40. Módulo contornos3d.py: Creación de arrays para las nubes de puntos. (Elaboración propia)

El siguiente paso de esta función es un filtrado de puntos, para ello se calcula la distancia media de los puntos del objeto a la cámara y se eliminan los puntos cuya distancia al "centro del objeto" supera cierto umbral. Gracias a esta etapa podemos eliminar valores atípicos que puedan incluir ruido en la imagen y posteriores problemas de detección.

```
dist = np.linalg.norm(points, axis=1)
d_avg = np.mean(dist)
umbral = 0.3
in_range = lambda d: (d > (1-umbral) * d_avg) & (d < (1+umbral) * d_avg)
pf, cf = points[in_range(dist)], colors[in_range(dist)]
edgef = edge_points[in_range(np.linalg.norm(edge_points, axis=1))]

pcd = o3d.geometry.PointCloud(points=o3d.utility.Vector3dVector(pf))
pcd.colors = o3d.utility.Vector3dVector(cf)
edge_pcd = o3d.geometry.PointCloud(points=o3d.utility.Vector3dVector(edgef))
edge_pcd.paint_uniform_color([1, 0, 0])
o3d.visualization.draw_geometries([pcd, edge_pcd], window_name="Objeto 3D con Bordes")

return pcd
```

Ilustración 41. Módulo contornos3d.py: Filtrado de puntos y creación de la nube. (Elaboración propia)

Finalmente, se construyen dos nubes de puntos en Open3D:

- Una nube que representa el cuerpo del objeto, con los colores obtenidos de la imagen RGB original
- Otra nube que representa únicamente los puntos del contorno, visualizados en rojo.

Ambas nubes son mostradas simultáneamente a través de la herramienta *draw_geometries* de Open3D, permitiendo al usuario analizar la geometría del objeto segmentado y sus contornos. Además, este visor nos permite navegar por el objeto, pudiendo obtener distintas perspectivas de este, siempre con la información obtenida de la cámara.

Ilustración 42. Objeto 3D con bordes. (Elaboración propia)

5.4.2. Reconstrucción del objeto virtual

Una vez obtenida la nube de puntos que representa el objeto segmentado, el siguiente paso es reconstruir una representación simplificada del objeto mediante una caja virtual. Esta caja se define a partir de las tres superficies principales visibles y se adapta al volumen y orientación del objeto de trabajo, facilitando la búsqueda de agarres válidos para el manipulador robótico.

Este objeto virtual nos permitirá obtener un agarre óptimo de una manera sencilla mediante herramientas matemáticas, ya que trabajar únicamente con la nube de puntos del objeto real no es una opción válida para este tipo de sistemas ya que las nubes se encuentran altamente incompletas al tomarse desde una única perspectiva. Para realizar la orientación de un agarre válido para el objeto sin crear una forma aproximada virtual sería necesario el modelo 3D del objeto o la captura de varias tomas del mismo y la superposición de estas para conseguir una nube de puntos completa de este.

Este proceso se implementa a través del módulo **caja_3d.py,** el cual se apoya en un conjunto de funciones definidas en **planos3d.py.** El proceso que se sigue para definir esta caja virtual es el siguiente:

- Detección de los planos principales del objeto:

  Se segmentan sucesivamente tres planos sobre la nube de puntos. Para ello se utiliza la función *detectar_superficies_planas(),* la cual emplea el algoritmo *RANSAC* para detectar superficies planas en la nube de

puntos. Para cada plano detectado se especifica un color y se extraen los coeficientes del plano en forma de ecuación: **Ax + By +Cz + D = 0.**

El funcionamiento de RANSAC se basa en una ejecución iterativa, donde en cada operación se selecciona un conjunto mínimo de tres puntos que definen un plano. A partir de ese plano se calcula cuántos puntos de la nube se ajustan a él con cierta tolerancia previamente definida. A los puntos que se encuentran dentro del plano establecido se denomina *inliers* y definen una posible superficie plana.

El proceso se repite un número establecido de iteraciones, y al finalizar, se selecciona el plano con mayor número de *inliers* como plano más representativo de la superficie. El resultado ofrece dos subconjuntos de puntos: los que pertenecen al plano detectado ***(inliers)*** y los que se encuentran fuera de este ***(outlier).***

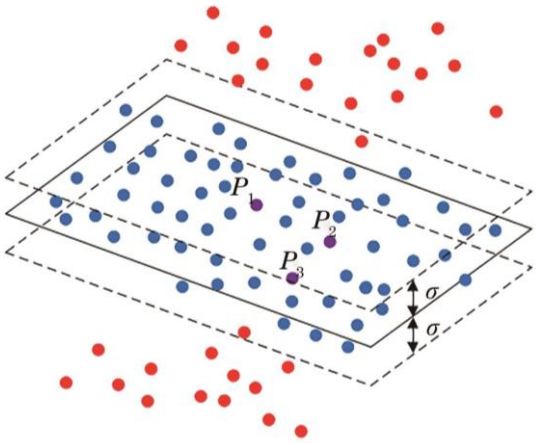

Ilustración 43. Representación del algoritmo RANSAC (Zuo, 2023)

En el flujo de trabajo de nuestra aplicación deseamos encontrar las tres superficies planas más representativas, para ello primero ejecutamos RANSAC con toda la nube del objeto, de manera que encontramos la superficie plana con mayor número de puntos. En la siguiente iteración, ejecutamos el algoritmo sobre los *outliers* de la detección anterior. De esta manera, garantizamos no encontrar el mismo plano en varias iteraciones.

```
def detectar_superficies_planas(pcd, color, distance_threshold=0.002, ransac_n=3, num_iterations=1000):
    try:
        plane_model, inliers = pcd.segment_plane(distance_threshold, ransac_n, num_iterations)
    except RuntimeError as e:
        print("No se pudo segmentar plano:", e)
        return None, pcd, None
    inlier_cloud = pcd.select_by_index(inliers)
    outlier_cloud = pcd.select_by_index(inliers, invert=True)
    inlier_cloud.paint_uniform_color(color)
    outlier_cloud.paint_uniform_color([0.5, 0.5, 0.5])
    print(f"Ecuación del plano: {plane_model[0]}x + {plane_model[1]}y + {plane_model[2]}z + {plane_model[3]} = 0")
    return inlier_cloud, outlier_cloud, plane_model
```

Ilustración 44. Módulo planos3d.py: Función detectar_superficies_planas. (Elaboración propia)

Tras la detección de los tres planos, se visualizan en una nube de puntos para comprobar que se han detectado correctamente.

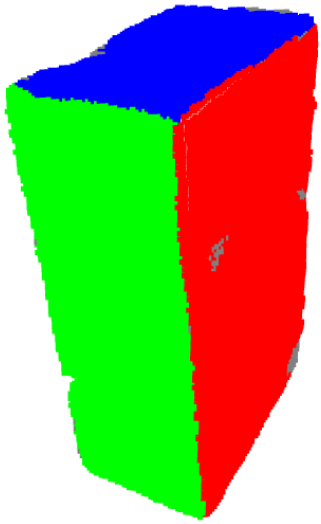

Ilustración 45. Superficies planas detectadas sobre el objeto 3D. (Elaboración propia)

- Cálculo del vértice común de la intersección:

  Una vez se tienen las ecuaciones de los tres planos, el siguiente paso es definir el vértice sobre el cual se cruzan. Para ello se ha diseñado una función que calcula la intersección entre tres planos resolviendo un sistema de ecuaciones lineales. Este punto sirve para posicionar la caja tridimensional, la función que lo define es **intersect_three_planes()**

```
def intersect_three_planes(p1, p2, p3):
    A = np.array([p1[:3], p2[:3], p3[:3]])
    b = np.array([-p1[3], -p2[3], -p3[3]])
    try:
        pt = np.linalg.solve(A, b)
        return pt
    except np.linalg.LinAlgError:
        print("Los planos no se intersectan en un punto.")
        return None
```

Ilustración 46. Módulo planos3d.py: Función intersect_three_planes(). (Elaboración propia)

- Determinación de las direcciones principales

  A partir de cada par de planos, se calcula una línea de intersección. Estas líneas definen las direcciones ortogonales del objeto, es decir, los ejes que orientan la caja (aristas visibles), para ello se define una función similar a la anterior, pero esta vez calculando la intersección entre dos planos. Esta función se denomina ***intersect_two_planes.***

  Además, los ejes encontrados deben orientarse de forma cuya dirección sea hacia la cámara, para evitar conflictos con la orientación posterior del objeto virtual. Para ello se utiliza una función secundaria que orienta

el vector de dirección apoyándose en el vértice de corte de los tres planos calculado anteriormente, ***orientar_vector_direccion().***

```python
def orientar_vector_direccion(vec, ref_pt, cam_pt):
    hacia_camara = cam_pt - ref_pt
    return -vec if np.dot(vec, hacia_camara) > 0 else vec

def intersect_two_planes(p1, p2, p_ref, cam_pt):
    n1 = np.array(p1[:3])
    n2 = np.array(p2[:3])
    d = np.cross(n1, n2)
    if np.linalg.norm(d) == 0:
        print("Los planos son paralelos o coincidentes.")
        return None
    d = orientar_vector_direccion(d, p_ref, cam_pt)
    return p_ref, d
```

Ilustración 47. Módulo planos3d.py: Funciones orientar_vector_direccion e intersect_two_planes. (Elaboración propia)

- Estimación de las dimensiones del objeto

  El siguiente paso en el programa es calcular las dimensiones del objeto, para ello se definen dos funciones:

  - La función **calcular_diagonal()** navega por los puntos de cada plano definido, calculando la distancia entre ellos hasta encontrar la diagonal máxima de cada cara del objeto.
  - Posteriormente, la función **calcular_dimensiones(),** extrae los datos de ancho, alto y largo del objeto a partir de las diagonales encontradas. Esta función utiliza la librería matemática de Python ***math***.

```python
def calcular_diagonal(plano):
    puntos = np.asarray(plano.points)
    if len(puntos) == 0:
        return 0.0
    dist = squareform(pdist(puntos))
    return np.max(dist)

def calcular_dimensiones(d1, d2, d3):
    s2 = (d1**2 + d2**2 + d3**2) / 2
    a = math.sqrt(max(s2 - d2**2, 0))
    b = math.sqrt(max(s2 - d3**2, 0))
    c = math.sqrt(max(s2 - d1**2, 0))
    return a, b, c
```

Ilustración 48. Módulo planos3d.py: Funciones calcular_diagonal y calcular_direcciones. (Elaboración propia)

- Cálculo de los vértices de la caja virtual

  Utilizando el punto de intersección, las tres direcciones principales encontradas y las dimensiones de la caja se construyen ocho vértices para representar la caja tridimensional virtual a través de la función **calcular_vertices().**

```python
def calcular_vertices(p0, r1, r2, r3, w, h, l):
    if None in (r1, r2, r3):
        return np.zeros((8, 3))
    v1 = r1[1] / np.linalg.norm(r1[1]) * w
    v2 = r2[1] / np.linalg.norm(r2[1]) * h
    v3 = r3[1] / np.linalg.norm(r3[1]) * l
    return np.array([
        p0,
        p0 + v1,
        p0 + v2,
        p0 + v3,
        p0 + v1 + v2,
        p0 + v1 + v3,
        p0 + v2 + v3,
        p0 + v1 + v2 + v3
    ])
```

Ilustración 49. Módulo planos3d.py: Función calcular_vertices. (Elaboración propia)

- Visualización de la caja virtual

  Para dar al usuario la opción de ver el resultado de la caja virtual generada, se crea la función **dibujar_caja(),** que se encarga de generar un objeto *LineSet* en *Open3D* que une los vértices encontrados y construir así una estructura que se representa junto a la nube de puntos original.

```python
def dibujar_caja(vertices):
    line_set = o3d.geometry.LineSet()
    line_set.points = o3d.utility.Vector3dVector(vertices)
    line_set.lines = o3d.utility.Vector2iVector([
        [0, 1], [1, 4], [4, 2], [2, 0],
        [3, 5], [5, 7], [7, 6], [6, 3],
        [0, 3], [1, 5], [2, 6], [4, 7]
    ])
    line_set.colors = o3d.utility.Vector3dVector([[1, 0, 0]] * 12)
    return line_set
```

Ilustración 50. Módulo planos3d.py: Función dibujar_caja. (Elaboración propia)

Una vez finalizado el flujo de trabajo de definición del objeto virtual, este se muestra junto con la nube de puntos del objeto original segmentado, para que el usuario pueda comprobar la coincidencia de ambos.

En las siguientes ilustraciones se puede comprobar la adaptación del objeto virtual creado, representado por líneas rojas, frente a la nube incompleta de puntos del objeto real. Este proceso es una manera sencilla de obtener una reconstrucción 3D del objeto completo, sin necesidad de realizar varias tomas del mismo ni de utilizar algoritmos complejos para ello.

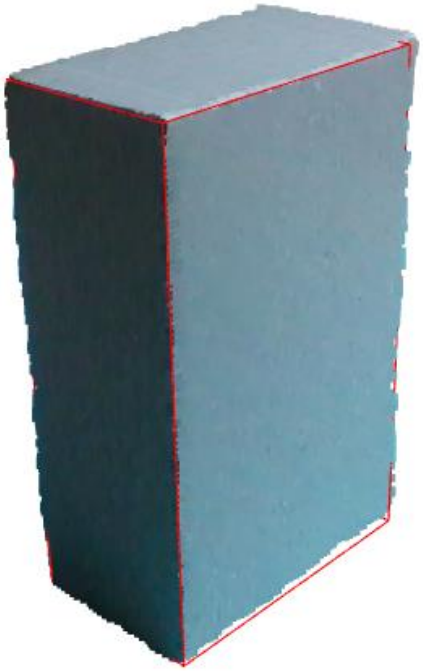

Ilustración 51. Caja virtual sobre nube de puntos del objeto, perspectiva 1. (Elaboración propia)

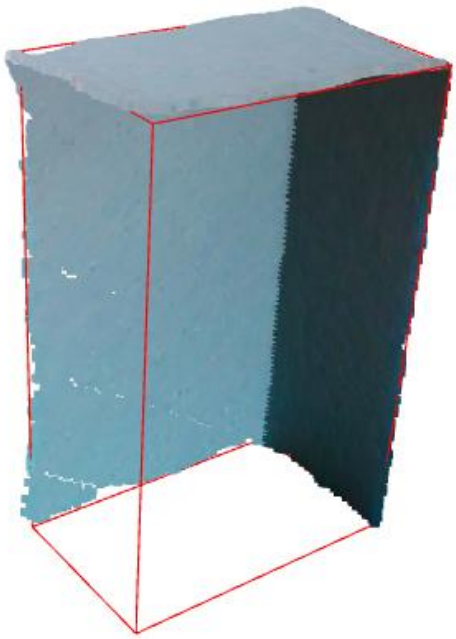

Ilustración 52. Caja virtual sobre nube de puntos del objeto, perspectiva 2. (Elaboración propia)

El flujo de trabajo de creación del objeto se controla con el módulo **caja_3d.py** que utiliza las funciones mencionadas anteriormente para realizar cada una de las etapas necesarias para crear el objeto virtual y visualizarlo.

```
def definir_caja(pcd):

    plano1, resto, m1 = detectar_superficies_planas(pcd, [1, 0, 0])
    plano2, resto, m2 = detectar_superficies_planas(resto, [0, 1, 0])
    plano3, resto, m3 = detectar_superficies_planas(resto, [0, 0, 1])

    if m1 is None or m2 is None or m3 is None:
        print("No se detectaron 3 planos válidos")
        return

    o3d.visualization.draw_geometries([plano1, plano2, plano3, resto], window_name="Superficies Planas")

    cam_pt = np.array([0, 0, 0])
    inter_pt = intersect_three_planes(m1, m2, m3)
    if inter_pt is None:
        return

    l1 = intersect_two_planes(m1, m2, inter_pt, cam_pt)
    l2 = intersect_two_planes(m1, m3, inter_pt, cam_pt)
    l3 = intersect_two_planes(m2, m3, inter_pt, cam_pt)

    d1, d2, d3 = calcular_diagonal(plano1), calcular_diagonal(plano2), calcular_diagonal(plano3)
    ancho, alto, largo = calcular_dimensiones(d1, d2, d3)
    print(f"ancho:", ancho, "  alto:", alto, "largo:", largo)
    vertices = calcular_vertices(inter_pt, l1, l2, l3, ancho, alto, largo)
    caja = dibujar_caja(vertices)
    o3d.visualization.draw_geometries([pcd, caja])
```

Ilustración 53. Módulo caja_3d.py: Función definir_caja. (Elaboración propia)

5.4.3. Generación y visualización de agarres

El módulo **agarre3d.py** se integra como etapa final del sistema tradicional de detección de agarres. Este se encarga de estimar, a partir de la caja virtual creada con el módulo anterior, una posible configuración de agarre válida para el robot.

Este método utiliza un planteamiento geométrico, basándose en las dimensiones calculadas y la orientación espacial del objeto para definir un único agarre. Consta de tres funciones principales:

Calcular_ejes_agarre()

Esta función recibe como entrada los vértices de la caja virtual y sus dimensiones (ancho, alto y largo) y en función de estos se encarga de analizar las posibles orientaciones del Grasp cumpliendo con su máxima apertura.

Primero, divide las seis caras que definen la caja en tres grupos, simulando estos los tres ejes principales (X, Y, Z), añadiendo a cada grupo una de las dimensiones de la caja. Esta agrupación se utiliza para realizar un filtrado de las posibles orientaciones del Grasp.

Para que un agarre sea considerado válido, la dimensión a lo largo del eje *binormal* (eje de apertura) debe ser menor que la apertura máxima del gripper, definida como *MAX_APERTURA*. En caso de que ninguna de las tres orientaciones posibles cumpla este requisito, se descarta el agarre.

```python
def calcular_ejes_agarre(vertices, ancho, alto, largo):
    if vertices.shape != (8, 3):
        return None

    cam_pt = np.array([0, 0, 0])

    # Asociar dimensiones correctamente basadas en planos3d.py
    dimensiones = {
        'x': {  # ancho (v1)
            'dim': ancho,
            'caras': (vertices[[0, 1, 3, 5]], vertices[[2, 4, 6, 7]]),
            'vertices_binormal': (0, 1),
        },
        'y': {  # alto (v2)
            'dim': alto,
            'caras': (vertices[[0, 2, 3, 6]], vertices[[1, 4, 5, 7]]),
            'vertices_binormal': (0, 2),
        },
        'z': {  # largo (v3)
            'dim': largo,
            'caras': (vertices[[0, 3, 1, 5]], vertices[[2, 6, 4, 7]]),
            'vertices_binormal': (0, 3),
        },
    }

    # Filtrar por dimensiones menores a la apertura
    opciones_validas = [(k, v) for k, v in dimensiones.items() if v['dim'] <= MAX_APERTURA]
    if not opciones_validas:
        print("Ninguna orientación permite agarre con apertura ≤ 85mm")
        return None
```

Ilustración 54. Módulo agarre3d.py: Agrupación de dimensiones y filtrado (Elaboración propia)

En el caso de que existan varias opciones válidas (varias caras por donde se pueda agarrar el objeto), se selecciona la que corresponda a la mayor

dimensión, para favorecer un agarre más firme y con mayor sujeción. Acto seguido se evalúa la distancia de las dos caras a la cámara, para seleccionar el agarre en la cara más próxima.

La dirección de aproximación del agarre (*approach*) se define como el vector que conecta la cara cercana con la lejana y se normaliza para obtener una orientación unitaria. El punto de agarre (position) se define como un punto en el centro de la cara seleccionada, ligeramente desplazado (1cm) hacia fuera de la caja, a lo largo del eje *approach*, para evitar colisiones directas con el objeto.

La dirección que define el eje de apertura de la garra (*binormal)* se define como el vector entre los dos vértices que forman la cara de la caja virtual y distan la dimensión elegida como válida para el agarre. Siendo este ortogonal al ya definido *approach.*

Por último, se calcula el eje de cierre del gripper (*axis)* como producto vectorial de los dos anteriores. Y nos aseguramos de que apunte hacia la dirección correcta para facilitar el acceso con la garra, invirtiendo su valor si la coordenada Y es negativa.

Esta función devuelve un conjunto de datos que definen el agarre.

```
# Elegir menor dimensión válida como eje de agarre (binormal)
eje_binormal, datos = max(opciones_validas, key=lambda x: x[1]['dim'])
cara1, cara2 = datos['caras']

# Centros de las caras
c1, c2 = np.mean(cara1, axis=0), np.mean(cara2, axis=0)

cara_cercana = c1 if np.linalg.norm(c1 - cam_pt) < np.linalg.norm(c2 - cam_pt) else c2
cara_lejana = c2 if cara_cercana is c1 else c1

# approach apunta DESDE la cara cercana hacia la lejana
approach = (cara_lejana - cara_cercana)
approach /= np.linalg.norm(approach)

# position = centro de cara cercana + 10mm fuera
position = cara_cercana - 0.01 * approach

# binormal = vector entre dos vértices definidos por la dimensión de agarre
vi, vj = datos['vertices_binormal']
binormal = vertices[vj] - vertices[vi]
binormal -= np.dot(binormal, approach) * approach  # proyectar fuera de approach
binormal /= np.linalg.norm(binormal)

# axis = cierre del gripper
axis = np.cross(binormal, approach)
axis /= np.linalg.norm(axis)

# aseguramos que axis apunte hacia arriba si su componente Y es negativa
if axis[1] < 0:
    axis = -axis

return {
    "position": position,
    "axis": axis,
    "approach": approach,
    "binormal": binormal,
    "eje": eje_binormal,
    "dimension": datos['dim']
}
```

Ilustración 55. Módulo agarre3d.py: Determinación de la posición y orientación del gripper. (Elaboración propia)

Visualización del agarre

El módulo incluye una función llamada **visualizar_agarre** que genera una escena en Open3D donde se representa:

- La nube de puntos del objeto
- La caja virtual ajustada al objeto
- La dirección del agarre representada por una flecha por cada vector de dirección: *approach* (rojo), *axis* (azul) y *binormal* (verde)
- Una pequeña esfera que marca la posición central del agarre
- Una **garra virtual** tridimensional, generada con las dimensiones de la pinza Robotiq 2F-85, orientada según los vectores anteriores.

```
def visualizar_agarre(pcd, caja, agarre):
    geometries = [pcd, caja]

    garra = crear_garra_virtual(agarre["position"], agarre["binormal"], agarre["approach"])

    for nombre, vector, color in zip(["axis", "approach", "binormal"],
                                     [agarre["axis"], agarre["approach"], agarre["binormal"]],
                                     [[0, 0, 1], [1, 0, 0], [0, 1, 0]]):
        arrow = o3d.geometry.TriangleMesh.create_arrow(
            cylinder_radius=0.002, cone_radius=0.004,
            cylinder_height=0.05, cone_height=0.02)
        arrow.paint_uniform_color(color)
        v = vector / np.linalg.norm(vector)

        z = np.array([0, 0, 1])
        if np.allclose(v, z):
            R = np.eye(3)
        elif np.allclose(v, -z):
            R = o3d.geometry.get_rotation_matrix_from_axis_angle(np.pi * np.array([1, 0, 0]))
        else:
            axis_rot = np.cross(z, v)
            angle = np.arccos(np.clip(np.dot(z, v), -1.0, 1.0))
            R = o3d.geometry.get_rotation_matrix_from_axis_angle(angle * axis_rot / np.linalg.norm(axis_rot))

        arrow.rotate(R, center=np.zeros(3))
        arrow.translate(agarre["position"])
        geometries.append(arrow)

    esfera = o3d.geometry.TriangleMesh.create_sphere(radius=0.002).translate(agarre["position"])
    esfera.paint_uniform_color([1, 1, 0])
    geometries.append(esfera)

    o3d.visualization.draw_geometries(list(garra) + geometries)
```

Ilustración 56. Módulo agarre3d.py: Función visualizar_agarre. (Elaboración propia)

Esta ventana interactiva permite validar el agarre generado y observar cómo se adapta la garra al tamaño y orientación del objeto. Resulta muy útil durante la etapa de depuración y pruebas del sistema, ya que permite ajustar el algoritmo y obtener una visualización realista sin necesidad de incorporar el sistema con el brazo robótico en el laboratorio.

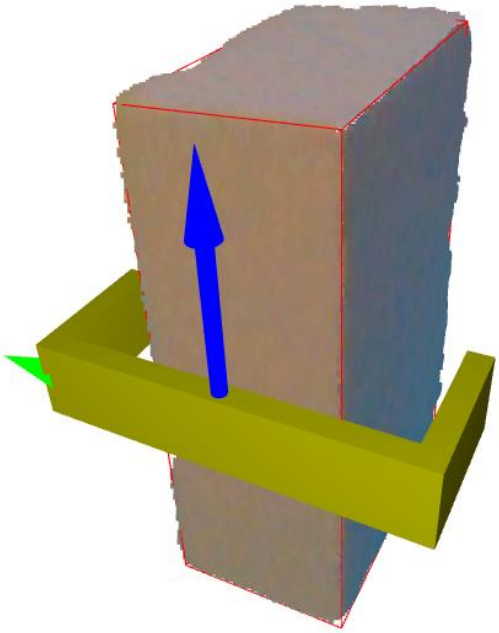

Ilustración 57.Visualización del agarre creado para el objeto 3D, perspectiva 1. (Elaboración propia)

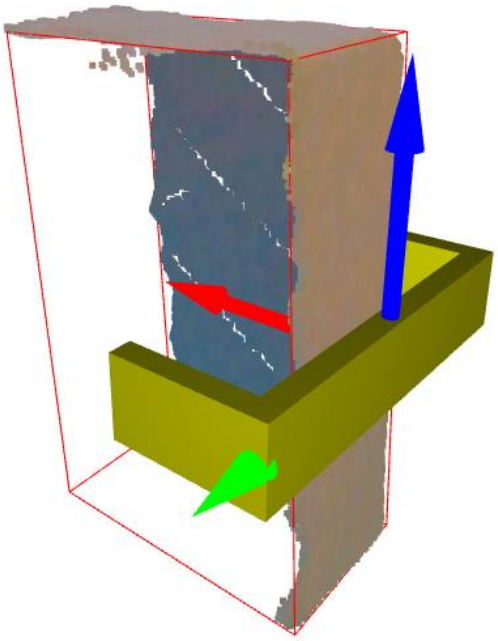

Ilustración 58. Visualización del agarre creado para el objeto 3D, perspectiva 2. (Elaboración propia)

Conversión al sistema de la cámara

El agarre generado, se encuentra referenciado a la posición de la cámara, pero con el sistema de referencia de Open3D, cuyos ejes "Y" y "Z" tienen la dirección opuesta al sistema de referencia de la cámara. Por este motivo, es necesario realizar una conversión al sistema de referencia de Intel RealSense antes de calcular los parámetros asociados al agarre que se ejecutarán en el brazo robótico.

Para ello se crea una pequeña función que invierte los valores de dirección de los datos del agarre generado.

```
def convertir_a_sistema_camara(pos, axis, binorm, appr):
    """
    Convierte los vectores de grasp del sistema Open3D al sistema de la cámara (RealSense).
    """
    def revertir(vec):
        v = vec.copy()
        v[1] = -v[1]
        v[2] = -v[2]
        return v

    return revertir(pos), revertir(axis), revertir(binorm), revertir(appr)
```

Ilustración 59. Módulo agarre3d.py: Función convertir_a_sistema_camara. (Elaboración propia)

Una vez convertidos los agarres al sistema de referencia de la cámara, se calculan los ángulos de Euler (xyz) y el cuaternión asociado, empleando las funciones de la librería *scipy.spatial.transform* y se imprime la información por

pantalla. Este cálculo se realiza en el módulo caja_3d.py para facilitar el flujo de trabajo.

```
agarre = calcular_ejes_agarre(vertices, ancho, alto, largo)
if agarre:
    print("Agarre:", agarre)

    pos_cam, axis_cam, binorm_cam, appr_cam = convertir_a_sistema_camara(
        agarre["position"], agarre["axis"], agarre["binormal"], agarre["approach"])

    # Rotación para visualización y consola
    rot = R.from_matrix(np.column_stack((binorm_cam, axis_cam, appr_cam)))
    euler = rot.as_euler('xyz', degrees=True)
    quat = rot.as_quat()

    print(f"  Posición (Open3D): {pos_cam}")
    print(f"  Ángulos Euler (Open3D): {euler}")
    print(f"  Cuaternión: {quat}")

    visualizar_agarre(pcd, caja, agarre)
```

Ilustración 60. Módulo caja_3d.py: Cálculo, visualización y extracción del agarre generado. (Elaboración propia)

```
Puntos seleccionados: [(272, 186)]
Imagen del objeto guardada.
Ecuación del plano: 0.8732354813180978x + -0.25759120748821046y + 0.41365029190357844z + 0.2168041364610552 = 0
Ecuación del plano: -0.4601099315172326x + -0.5224273582653604y + 0.7178917092814795z + 0.3079394182683854 = 0
Ecuación del plano: -0.025114791447918016x + 0.7939512129792182y + 0.6074625244896641z + 0.19580321140182316 = 0
ancho: 0.0847848313056114   alto: 0.14172571078569068 largo: 0.05065052284585997
Agarre: {'position': array([-0.06212334,  0.00658254, -0.45005907]), 'axis': array([-0.02511479,  0.79395121,  0.60746252]),
'approach': array([ 0.48504471,  0.54101339, -0.68704886]), 'binormal': array([-0.87412864,  0.2773914 , -0.39868928]),
'eje': 'z', 'dimension': 0.05065052284585997}
  Posición (Open3D): [-0.06212334 -0.00658254  0.45005907]
  Ángulos Euler (Open3D): [ -41.48189333  -23.49626443 -162.39395787]
  Cuaternión: [ 0.24123314 -0.31349982  0.91585052 -0.06886402]
```

Ilustración 61. Salida por la consola, información del agarre generado. (Elaboración propia)

Este sistema de detección de agarres, aunque limitado frente a sistemas basados en aprendizaje profundo, ofrece una solución funcional, rápida y comprensible para entornos con objetos controlados y que puedan ser representados por ortoedros.

5.5. GPD

Una vez realizado el preprocesamiento, que concluye con la segmentación del objeto de interés, se inicia el flujo de trabajo específico basado en el modelo GPD (Grasp Pose Detection). El objetivo de este flujo de trabajo será reconstruir la nube de puntos del objeto y ejecutar sobre esta el algoritmo GPD, el cual nos generará los mejores agarres, para después extraerlos y procesarlos para adecuarlos a nuestro entorno.

A continuación, se explicará el procesamiento que se lleva a cabo en la nube de puntos, la comunicación con el servidor donde se ha instalado GPD, qué procesamiento lleva a cabo este algoritmo para detectar agarres y cómo se interpretan estos posteriormente.

5.5.1. Config.py: Módulo de configuración

El módulo **config.py** se ha creado para centralizar todos los parámetros de configuración necesarios para la ejecución del sistema, permitiendo al usuario

actualizar rutas, credenciales y direcciones IP sin necesidad de alterar el código funcional de los distintos módulos.

Este enfoque facilita la portabilidad del sistema, así como su mantenimiento y la separación entre la lógica del procesamiento y las variables del entorno.

El archivo se encuentra dividido en tres bloques principales, según la función que desempeñan las variables definidas.

1. **Parámetros generales**

- SAM_MODEL_PATH

  Define la ruta local al modelo de segmentación *Segment Anything (SAM),* que será cargado por el módulo **segment.py** para realizar la extracción del objeto del entorno.

  Valor por defecto: "sam_b.pt"

- LOCAL_PCD_DIR

  Ruta a la carpeta local donde se almacenan las nubes de puntos generadas por el sistema. En el momento de la carga del módulo se verifica la existencia de la carpeta seleccionada con la función **os.makedirs()** y en caso de que esta no esté definida se crea.

2. **Configuración para la comunicación mediante servidor web**

- SERVIDOR_URL

  Dirección IP del servidor donde se encuentra desplegado el servicio web. Esta dirección IP es utilizada por el módulo **remote.py** para enviar las peticiones HTTP (POST) que contienen la nube de puntos a procesar.
  En este caso, el sistema escucha el puerto 80, este puerto es el escuchado por defecto, por lo que no es necesario especificarlo en la dirección.
  En caso de modificar el servicio web e instalarlo escuchando otro puerto deberá incluirse esta información junto a la IP del servidor.

3. **Configuración para la comunicación mediante SSH/SFTP**

   Este bloque contiene los parámetros necesarios para establecer una conexión remota segura mediante el protocolo SSH y realizar la transferencia de ficheros mediante SFTP. Es utilizado por el módulo **remote_ssh.py,** por lo que si este módulo se encuentra inactivo no es necesario establecer estos parámetros.

- HOST, PORT, USERNAME, PASSWORD:

  Con estas cuatro variables se establecen las credenciales al servidor remoto *Inception.*

  En el parámetro HOST debemos incluir la dirección IP del servidor

El parámetro PORT nos indica el puerto, que toma el valor estándar 22

En los parámetros USERNAME y PASSWORD debemos incluir las credenciales de la sesión de *Inception* donde hemos instalado el modelo de detección de agarres. La contraseña se encuentra comentada para evitar la exposición del servidor.

- REMOTE_PCD_PATH

  Ruta a la carpeta del servidor donde se almacenará la nube de puntos enviada desde el cliente. Este directorio debe ser accesible desde el usuario remoto y coincidir con el entorno de trabajo del modelo GPD

- REMOTE_CSV_PATH

  Ruta del servidor en la que el modelo GPD genera el archivo de resultados **grasp_filtered.csv** tras la ejecución.

  Esta ruta es consultada por el cliente para recuperar el archivo tras la detección de los agarres.

- REMOTE_SCRIPT_COMMAND

  Comando que se ejecuta de forma remota para lanzar el script de Python instalado en el servidor que ejecuta el modelo de detección de agarres.

  Este comando, que debe encontrarse en lenguaje Linux (lenguaje del servidor) consta de varias partes:

  - **cd ... :** El comando cambia el directorio del servidor al de trabajo donde se encuentra el script a ejecutar
  - **DISPLAY=:11.0 :** Define una variable del entorno, necesaria si se utilizan visualizaciones gráficas o entornos de ejecución virtual. (ej. Plots que nos ofrece el modelo GPD para visualizar las etapas del detector de agarres)
  - **Python3 script_remoto {{archivo_remoto}}:** comando que lanza el script de Python incluyendo como argumento la ruta al archivo .pcd generado

    El uso de la sintaxis **{{archivo_remoto}}** permite formatear dinámicamente el comando desde el módulo de comunicación para actualizar el nombre del archivo enviado en cada ejecución.

```
# ---------- PARÁMETROS GENERALES ------------------------
# Ruta local del modelo de segmentacion
SAM_MODEL_PATH = "sam_b.pt"

# Carpeta local para el guardado de imágenes y nubes de puntos
LOCAL_PCD_DIR = "nubes_pcd"
os.makedirs(LOCAL_PCD_DIR, exist_ok=True)


# ---------- COMUNICACIÓN SERVIDOR WEB --------------------
SERVIDOR_URL = 'http://157.88.201.8'  # IP del servidor (PUERTO = 80)


# ---------- COMUNICACIÓN SSH/SFTP ------------------------
# Datos del servidor remoto
HOST = '157.88.201.8'
PORT = 22
USERNAME = 'alba'
PASSWORD = '******' #-> Añadir contraseña para utilizar el módulo

# Rutas remotas
REMOTE_PCD_PATH = '/home/alba/GPD/nubes_pcd/'
REMOTE_CSV_PATH = '/home/alba/GPD/gpd/build/'

# Comando remoto a ejecutar
REMOTE_SCRIPT_COMMAND = (
    f"cd /home/alba/GPD/gpd && "
    f"DISPLAY=:11.0 python3 script_gpd.py {{archivo_remoto}}"
)
```

Ilustración 62. Módulo config.py. (Elaboración propia)

Este módulo permite adaptar fácilmente el comportamiento del sistema a distintos entornos, tanto locales como remotos y facilita la transición entre los dos modelos de comunicación (HTTP y SSH) sin necesidad de modificar el código de los módulos principales.

5.5.2. Pointcloud.py: Procesamiento de la nube de puntos

Este módulo se encarga de agrupar la imagen RGB del objeto con su mapa de profundidad para crear una representación tridimensional del objeto segmentado en forma de nube de puntos (.pcd). Este formato es esencial para que el sistema GPD pueda analizar la geometría del objeto y calcular posibles puntos de agarre.

De esta tarea se encarga la función **generar_nube_objeto(),** que recibe como entradas la máscara binaria de segmentación, las imágenes de profundidad y color y los parámetros de la cámara (*profile*).

En primer lugar, se extraen los parámetros de calibración necesarios desde el objeto *profile*. En este caso, es necesaria la definición de las características internas del sensor de profundidad *depth_intrin,* y la escala de profundidad *depth_scale.* Gracias a estos parámetros, podemos convertir los valores de profundidad de unidades internas a metros.

A continuación, se filtran los píxeles de la imagen de profundidad, tomando como referencia la máscara binaria del objeto segmentado. Para cada píxel válido, se calcula su espacio tridimensional con la función

rs2_deproject_pixel_to_point que nos ofrece la biblioteca **pyrealsense2** y transforma las coordenadas de una imagen 2D y el valor de profundidad en coordenadas 3D reales.

Cada punto 3D se asocia con su color correspondiente del canal RGB, para formar una nube de puntos con los colores reales. Al generar los puntos, se invierten las coordenadas de los ejes "Y" y "Z" para adaptarse al sistema de referencia de *Open3D,* que tiene ambos ejes contrarios a los de *RealSense.* Al finalizar la aplicación, se debe revertir esta operación para volver al sistema de referencia de la cámara.

Una vez dimensionados todos los píxeles del objeto, se calcula la distancia media a la cámara y se filtran los puntos que se encuentran más alejados de la media cierto umbral. De esta manera se eliminan puntos que no pertenezcan al objeto e introduzcan ruido en la nube, estos puntos pueden ser del fondo que no han sido bien detectados en la segmentación o cuyo valor de profundidad no era correcto.

Gracias a la librería Open3D, podemos crear una nube de puntos y asignar a esta los puntos filtrados del objeto y sus respectivos colores. Una vez creada la nube, tenemos la posibilidad de visualizarla con la misma librería. Además, se guarda en una carpeta local con un el mismo nombre único que recibió la imagen a color del objeto que guardamos en el bucle principal, así podemos asociar fácilmente la imagen con la nube de puntos para posteriores investigaciones.

Una vez guardada la nube de puntos en formato .pcd, esta se almacena en una variable del módulo **grasp.py** y se llama al módulo **remote.py** para que envíe la nube al servidor y ejecute sobre ella el algoritmo GPD.

```python
def generar_nube_objeto(mascara: np.ndarray, depth_image: np.ndarray, color_image: np.ndarray,
                        profile: rs.pipeline_profile, nombre_base: str) -> None:
    depth_intrin = profile.get_stream(rs.stream.depth).as_video_stream_profile().get_intrinsics()
    depth_scale = profile.get_device().first_depth_sensor().get_depth_scale()

    depth_obj = np.where(mascara, depth_image, 0)
    coords = np.column_stack(np.where(depth_obj > 0))

    points = []
    colors = []
    for v, u in coords:
        depth = depth_obj[v, u]
        point = rs.rs2_deproject_pixel_to_point(depth_intrin, [u, v], depth * depth_scale)
        point[1] = -point[1]
        point[2] = -point[2]
        points.append(point)
        colors.append(color_image[v, u][::-1] / 255.0)

    points = np.array(points)
    colors = np.array(colors)

    distancias = np.linalg.norm(points, axis=1)
    media = np.mean(distancias)
    umbral = 0.3
    valids = (distancias > (1-umbral) * media) & (distancias < (1+umbral) * media)
    points = points[valids]
    colors = colors[valids]

    pcd = o3d.geometry.PointCloud()
    pcd.points = o3d.utility.Vector3dVector(points)
    pcd.colors = o3d.utility.Vector3dVector(colors)

    pcd_path = os.path.join(LOCAL_PCD_DIR, f"nube_{nombre_base}.pcd")
    o3d.io.write_point_cloud(pcd_path, pcd)
    print(f"Nube de puntos guardada como {pcd_path}")

    o3d.visualization.draw_geometries([pcd], window_name="Objeto Seleccionado en 3D")

    guardar_pcd(pcd)
    enviar_archivo(pcd_path)
```

Ilustración 63. Módulo pointcloud.py: Función generar_nube_objeto. (Elaboración propia)

A través de la nube de puntos generada, podemos comprobar la adquisición del objeto, visualizándolo desde distintas perspectivas. Se puede observar cómo se trata de una nube de puntos notablemente incompleta, aquí reside la complejidad del sistema de detección. Se debe intentar obtener una fotografía donde se visualice el mayor número de características importantes del objeto.

Ilustración 64.
Nube de puntos del objeto, perspectiva 1
(Elaboración propia)

Ilustración 65.
Nube de puntos del objeto, perspectiva 2
(Elaboración propia)

Ilustración 66.
Nube de puntos del objeto, perspectiva 3
(Elaboración propia)

5.5.3. Módulo de comunicación

El módulo remote.py es el encargado de realizar la interacción entre el sistema local donde se realiza la adquisición del objeto y el servidor remoto donde se encuentra instalado el modelo de detección de agarres GPD.

Su objetivo principal es enviar de manera fiable la nube de puntos generada, ejecutar el modelo GPD en remoto y recuperar los resultados de agarre generados al flujo de trabajo local.

Durante el desarrollo de las comunicaciones se han implementado dos versiones de este módulo, con el fin de aumentar la flexibilidad de la aplicación, según las restricciones de la red y la expansión del sistema.

1. **Remote_ssh.py** nos ofrece una conexión cifrada SSH que utiliza el protocolo SFTP para la trasferencia de archivos y la ejecución remota de comandos
2. **Remote.py** se trata del cliente de un servidor web que se ha implementado en el servidor para ofrecer comunicación mediante peticiones HTTP

Por defecto, se ha configurado **remote.py** como módulo de comunicación para trabajar con el servidor web, que debe estar activo en *Inception* ya que ofrece mayor simplicidad y seguridad, evitando la necesidad de gestionar credenciales del sistema operativo y la necesidad de conexión mediante VPN.

No obstante, se ha mantenido accesible el módulo de comunicación SSH/SFTP, activarlo es tan sencillo como descomentar las líneas de código *"#from remote_ssh import cerrar_conexiones"* y *"#cerrar_conexiones()"* de **main.py** y comentar *"from remote import enviar_archivo"* y descomentar *"#from remote_ssh import enviar_archivo"* en **pointcloud.py.**

```
import numpy as np
import open3d as o3d
import pyrealsense2 as rs
import os
from remote import enviar_archivo
#from remote_ssh import enviar_archivo
from config import LOCAL_PCD_DIR
from grasp import guardar_pcd
```

Ilustración 67. Librerias del módulo pointcloud.py (Elaboración propia)

A continuación, se explican ambos módulos de comunicación.

5.5.3.1. Comunicación SSH/SFTP

El módulo **remote_ssh.py** implementa la comunicación SSH/SFTP para conectar con el servidor, para ello se han definido varias funciones que realizan distintas partes del diálogo, entre ellas el inicio de la conexión, una función para

enviar la nube de puntos al servidor y otra última para recibir el resultado del detector de agarres y finalmente una para cerrar las conexiones al finalizar el diálogo.

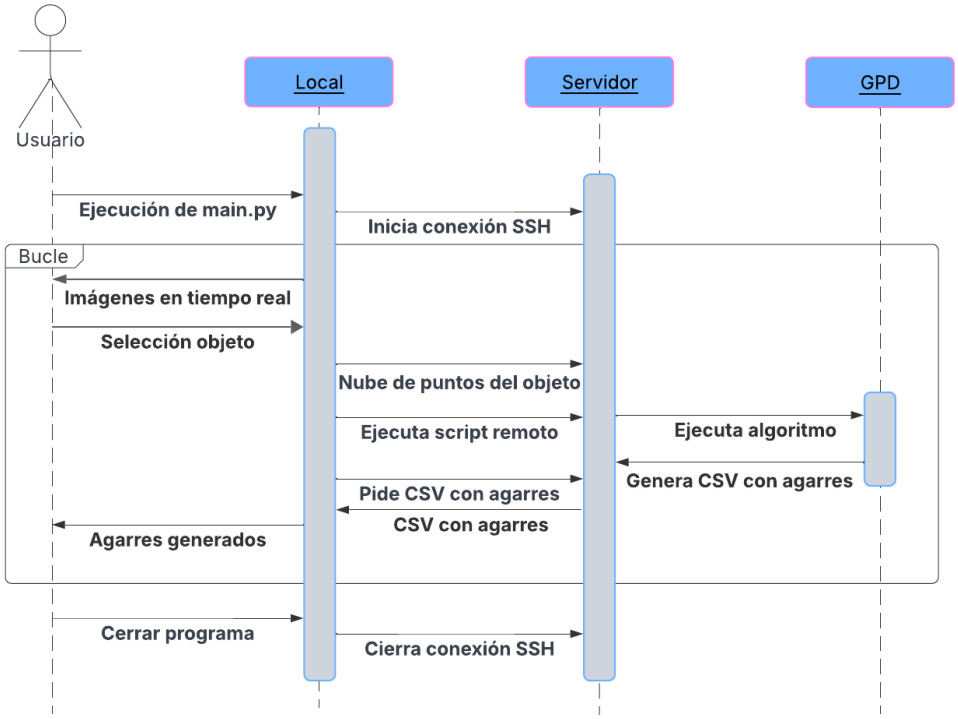

Ilustración 68. Diagrama de secuencia comunicación SSH/SFTP (Elaboración propia)

Para ello se utiliza la librería **paramiko** que nos ofrece conexiones seguras vía SSH y transferencia de archivos con SFTP. Al iniciar el script y cargar este módulo, se establece la conexión con el servidor a través de la función **iniciar_conexion().** Los parámetros necesarios para conectar con el servidor se encuentran en el módulo **config.py,** del cual se cargan el HOST, Puerto, usuario y contraseña.

La función mencionada devuelve tres objetos:

- Ssh: objeto que permite la ejecución de comandos remotos
- Sftp: objeto que permite enviar y recibir archivos
- Transport: canal de comunicación

```
def iniciar_conexion():
    transport = paramiko.Transport((HOST, PORT))
    transport.connect(username=USERNAME, password=PASSWORD)
    sftp = paramiko.SFTPClient.from_transport(transport)
    ssh = paramiko.SSHClient()
    ssh.set_missing_host_key_policy(paramiko.AutoAddPolicy())
    ssh.connect(HOST, port=PORT, username=USERNAME, password=PASSWORD)
    return ssh, sftp, transport
```

Ilustración 69. Módulo remote_ssh.py: Función iniciar_conexion. (Elaboración propia)

La siguiente función que nos encontramos dentro de este módulo es la que recibe la llamada desde **pointcloud.py,** junto con la ruta al archivo local de la nube de puntos generada. Esta función es **enviar_archivo()** y es la encargada de cargar la nube de puntos del objeto en el servidor.

```
def enviar_archivo(archivo_local: str) -> None:
    try:
        remoto = os.path.join(REMOTE_PCD_PATH, os.path.basename(archivo_local))
        sftp.put(archivo_local, remoto)
        print("Archivo enviado correctamente.")
        ejecutar_script_remoto(remoto)
    except Exception as e:
        print(f"Error al enviar archivo: {e}")
```

Ilustración 70. Módulo remote_ssh.py: Función enviar_archivo. (Elaboración propia)

Gracias a la conexión establecida anteriormente, esta función utiliza el objeto **sftp** para copiar el archivo desde la ruta del dispositivo local que se le indica a una ruta preestablecida en el servidor, utilizando el protocolo SFTP (SSH File Transfer Protocol).

Una vez realizado el envío de manera correcta, se ejecuta la función **ejecutar_script_remoto(),** cuyo objetivo es invocar al modelo GPD, indicándole la nube que debe procesar y esperar la respuesta del algoritmo. La ejecución del script se realiza a través del objeto **ssh** y una vez completada la ejecución del script remoto, se procede a descargar el archivo con los agarres generados de manera similar al envío de la nube de puntos. A través del protocolo SFTP se copia el archivo csv remoto a un directorio local.

```
def ejecutar_script_remoto(archivo_remoto: str) -> None:
    try:
        comando = REMOTE_SCRIPT_COMMAND.format(archivo_remoto=archivo_remoto)
        stdin, stdout, stderr = ssh.exec_command(comando)

        print("Salida del script:")
        print(stdout.read().decode())
        errores = stderr.read().decode()
        if errores:
            print("Errores del script:", errores)

        archivo_csv_remoto = os.path.join(REMOTE_CSV_PATH, 'grasps_filtered.csv')
        archivo_csv_local = "grasps_recibidos.csv"
        sftp.get(archivo_csv_remoto, archivo_csv_local)
        print(f"CSV descargado: {archivo_csv_local}")
        procesar_agarres(archivo_csv_local)

    except Exception as e:
        print(f"Error al ejecutar script remoto: {e}")
```

Ilustración 71. Módulo remote_ssh.py: Función ejecutar_script_remoto. (Elaboración propia)

Finalmente, para evitar conexiones abiertas que puedan provocar conflictos, se establece una función **cerrar_conexiones()** que cierra completamente los canales SSH, SFTP y de transporte.

```
def cerrar_conexiones():
    sftp.close()
    ssh.close()
    transport.close()
```

Ilustración 72. Módulo remote_ssh.py: Función cerrar_conexiones. (Elaboración propia)

5.5.3.2. *Comunicación HTTP Cliente*

El módulo **remote.py** implementa el cliente HTTP que permite la comunicación con un servidor web configurado en el sistema *Inception*, este es el encargado de ejecutar el modelo GPD y devolver los resultados al sistema local.

El servicio web se ha establecido para que funcione mediante un sistema de peticiones, donde el cliente le manda una petición de ejecución del GPD con la nube de puntos del objeto segmentado y el servidor le devuelve un fichero csv con los agarres detectados.

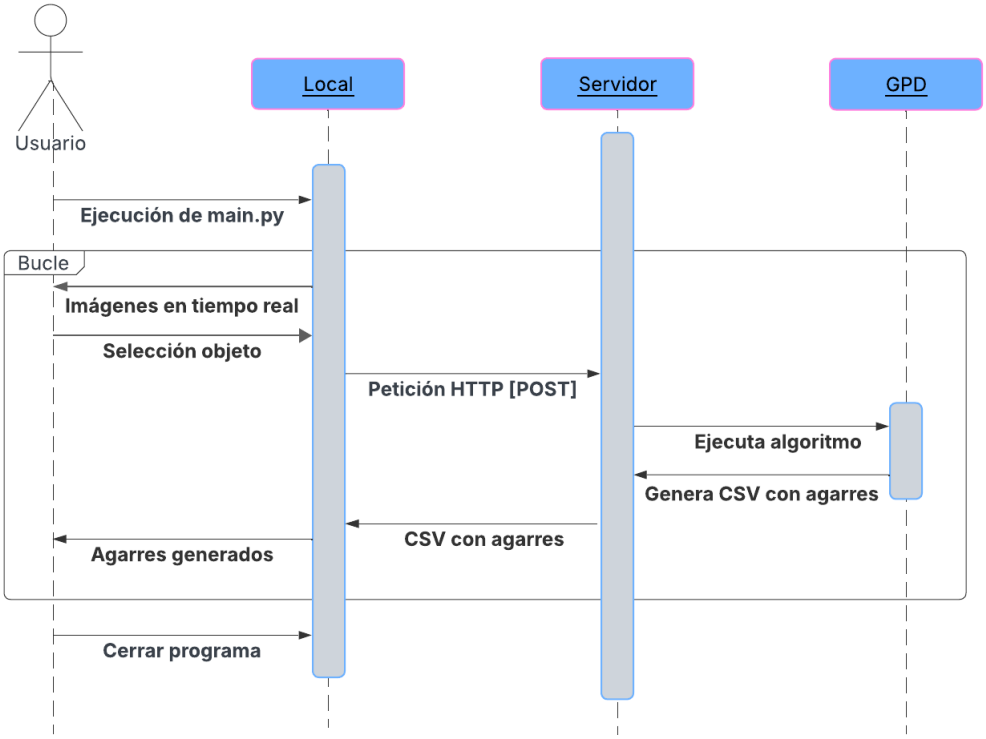

Ilustración 73. Diagrama de secuencia comunicación servidor web (Elaboración propia)

Para implementar este módulo se ha utilizado la biblioteca ***requests***, la cual permite enviar archivos al servidor remoto a través de una solicitud HTTP POST y recibir como respuesta un archivo generado por el modelo.

La función principal del módulo es **enviar_archivo( )**, la cual se encarga de abrir la nube de puntos generada en el sistema local, empaquetarla y enviarla al servidor mediante una petición POST a la ruta "/procesar_pcd" con la nube como fichero adjunto. Esta ruta está previamente definida en el servidor.

Una vez procesada la petición, se comprueba que el servidor responde con un código de éxito (*status_code* == 200) y la función descarga automáticamente

el archivo *Grasp_recibidos.csv* generado por el modelo y lo almacena en el directorio local.

Finalmente, se manda el nombre del archivo almacenado a la función *procesar_agarres()* del módulo **grasp.py** para procesar los agarres que se han recibido del servidor.

En caso de que la respuesta del servidor indique un error, ya sea por fallo en el archivo recibido o por errores internos en el script ejecutado en el servidor, se imprime un mensaje de error que incluye el estado del servicio web.

```python
def enviar_archivo(archivo_local: str) -> None:
    try:
        with open(archivo_local, 'rb') as f:
            files = {'file': f}
            print(f"Enviando {archivo_local} al servidor...")
            respuesta = requests.post(f"{SERVIDOR_URL}/procesar_pcd", files=files)

        if respuesta.status_code == 200:
            output_csv = "grasps_recibidos.csv"
            with open(output_csv, 'wb') as f:
                f.write(respuesta.content)
            print(f"CSV recibido y guardado como {output_csv}")
            procesar_agarres(output_csv)
        else:
            print(f"Error del servidor: {respuesta.status_code}")
            print(respuesta.text)

    except Exception as e:
        print(f"Error en la comunicación HTTP: {e}")
```

Ilustración 74. Módulo remote.py (Elaboración propia)

Al igual que en el módulo de comunicación anterior, los datos del servidor se extraen del fichero **config.py,** en este caso, únicamente se necesita establecer la dirección IP del servidor. Particularmente, no es necesario especificar el puerto de entrada del servicio porque se ha establecido en el puerto por defecto (puerto 80), pero en caso de cambiar el servicio web a otro, este se debe especificar en el archivo de configuración, además de permitir las conexiones remotas a ese puerto.

5.5.3.3. Comunicación HTTP Servidor

Para realizar la comunicación entre el cliente local y el servidor remoto se ha desarrollado un servicio web utilizando **Flask,** escrito en Python y ejecutado en el servidor *Inception.* Esta aplicación actúa como interfaz HTTP que recibe la nube de puntos generada por el sistema local, ejecuta el modelo GPD sobre ella, devolviendo al cliente un archivo CSV con los datos de los agarres resultantes.

El servicio establecido contiene una única ruta definida */procesar_pcd,* configurada para recibir solicitudes HTTP tipo POST con un archivo **.pcd** como cuerpo de la petición.

Se ha desarrollado una estructura modular, utilizando varias bibliotecas estándar de Python:

- **Flask:** gestiona las peticiones HTTP entrantes y envía respuestas
- **Tempfile:** para crear directorios temporales donde alojar los archivos .pcd durante la ejecución
- **Os:** para manejar rutas y verificar la existencia de los archivos de resultados generados
- **Subprocess:** para invocar el script de Python que lanza el ejecutable del modelo GPD
- **Jsonify y send_file:** para estructurar las respuestas del cliente, en caso de error serán mensajes JSON y en caso de éxito un archivo CSV

La lógica del servidor se resume en los siguientes pasos:

1. Recepción del archivo:

   Al recibir una solicitud POST en */procesar_pcd,* el servidor verifica que se haya enviado un archivo en el campo *file.* En caso contrario devuelve un error 400.

2. Almacenamiento temporal:

   Si el archivo se ha recibido correctamente, se guarda en un directorio temporal generado por tempfile.TemporaryDirectory(), así se asegura que los datos se gestionen de manera aislada y se eliminen al finalizar la ejecución, evitando la sobrecarga del servidor.

3. Ejecución del modelo GPD:

   El archivo temporal se pasa como argumento al script de Python que se encarga de ejecutar el detector de agarres. En caso de error en la ejecución de este, devuelve una respuesta con código de error 500.

4. Comprobación del resultado:

   Una vez finalizado correctamente el procesamiento, se verifica la existencia del archivo *Grasp_filtered.csv* en la ruta esperada. Si el archivo no existe se devuelve un mensaje de error personalizado indicando que el CSV no se generó.

5. Envío del archivo al cliente:

   Si el archivo existe, se envía como respuesta al cliente utilizando *send_file,* manteniendo el nombre del archivo que descargará el cliente.

```python
from flask import Flask, request, send_file, jsonify
import subprocess
import os
import tempfile

app = Flask(__name__)

RUTA_GPD = '/home/alba/GPD/gpd/build'
RUTA_SCRIPT = '/home/alba/GPD/gpd/script_gpd.py'

@app.route('/procesar_pcd', methods=['POST'])

def procesar_pcd():

    if 'file' not in request.files:
        return jsonify({'error': 'No se recibió archivo .pcd'}), 400

    file = request.files['file']

    if file.filename == '':
        return jsonify({'error': 'Archivo sin nombre'}), 400

    with tempfile.TemporaryDirectory() as tmpdir:

        pcd_path = os.path.join(tmpdir, file.filename)
        file.save(pcd_path)
        comando = [
            'python3',
            RUTA_SCRIPT,
            pcd_path
        ]
        try:
            resultado = subprocess.run(comando, capture_output=True, text=True, check=True)

        except subprocess.CalledProcessError as e:
            return jsonify({
                'error': 'Error al ejecutar script',
                'stdout': e.stdout,
                'stderr': e.stderr
            }), 500

        csv_path = os.path.join(RUTA_GPD, 'grasps_filtered.csv')

        if not os.path.exists(csv_path):
            return jsonify({'error': 'No se encontró el CSV generado'}), 500

        return send_file(csv_path, mimetype='text/csv', as_attachment=True, download_name='grasps_filtered.csv')


if __name__ == '__main__':

    app.run(host='0.0.0.0', port=8080)
```

Ilustración 75. Servidor web (Elaboración propia)

El servicio está diseñado para ejecutarse sobre cualquier entorno Linux con Python3 instalado. Al lanzar el script del servidor este se mantiene escuchando el puerto 80 y queda accesible desde cualquier equipo externo.

5.5.4. Script de Python en el Servidor

Para facilitar el uso automático del sistema GPD se ha diseñado un script de Python que se ejecuta en el servidor donde se encuentra instalado el modelo y es el encargado de gestionar la comunicación en el lado servidor y lanzar el modelo de predicción de agarres.

Este script se nutre de las librerías *subprocess*, *os* y *sys*. Y consta de dos funciones principales, una que se encarga de obtener la ruta donde se encuentra la nube del objeto segmentada y otra que se encarga de ejecutar el detector de agarres.

La primera de las funciones obtiene la ruta de la nube de puntos desde la entrada de argumentos, por tanto, al ejecutar el script desde el cliente local debemos indicar la ruta donde hemos copiado el archivo pcd generado. En caso de no encontrarse el archivo especificado el detector no se ejecuta y se generará un código de error.

```
import subprocess
import os
import sys

RUTA_GPD_BUILD = '/home/alba/GPD/gpd/build'
RUTA_CONFIG = '/home/alba/GPD/gpd/cfg/eigen_params.cfg'
RUTA_DETECT_BIN = os.path.join(RUTA_GPD_BUILD, 'detect_grasps')

def obtener_ruta_nube():
    """
    Obtiene la ruta de la nube de puntos desde argumentos.
    """
    if len(sys.argv) < 2:
        print("Error: No se proporcionó la ruta de la nube de puntos como argumento.")
        sys.exit(1)

    ruta = sys.argv[1]
    if os.path.exists(ruta):
        return ruta
    else:
        print(f"Error: No se encontró el archivo en la ruta {ruta}.")
        sys.exit(1)
```

Ilustración 76. Script_gpd.py, primera parte. (Elaboración propia)

La otra de las funciones se encarga de, una vez obtenida la ruta de la nube de puntos, ejecutar el detector de agarres. Para ello, debe indicar el fichero donde se encuentran los parámetros configurados y la nube de puntos del objeto. Esta función realiza su principal función con la librería *subprocess,* a través de la cual no solo ejecuta el comando ordenado, sino que también se mantiene a la escucha de los resultados o errores que se produzcan en la ejecución.

```
def ejecutar_detector():
    """
    Función para ejecutar el detector de grasps.
    """
    try:
        print("\nEjecutando el detector de agarres...")

        ruta_nube = obtener_ruta_nube()

        comando = [RUTA_DETECT_BIN, RUTA_CONFIG, ruta_nube]

        resultado = subprocess.run(comando, cwd=RUTA_GPD_BUILD, capture_output=True, text=True)

        print("Salida del comando:")
        print(resultado.stdout)
        if resultado.stderr:
            print("Errores del comando:")
            print(resultado.stderr)

    except Exception as e:
        print(f"Error al ejecutar el detector: {e}")


def main():
    ejecutar_detector()


if __name__ == "__main__":
    main()
```

Ilustración 77. Script_gpd.py, segunda parte. (Elaboración propia)

5.5.5. Arquitectura técnica de GPD

GPD es un sistema modular que detecta posibles agarres en nubes de puntos tridimensionales para pinzas de dos dedos. Su procesamiento se divide en tres etapas principales:

1. Generación de candidatos de agarre: se generan múltiples poses de agarre potencialmente viables a partir de la nube de puntos
2. Clasificación mediante CNN: cada candidato generado se evalúa con una red neuronal convolucional entrenada para distinguir entre agarres viables y no viables
3. Filtrado y agrupamiento de los agarres: los agarres clasificados como viables se filtran y agrupan según su similitud geométrica.

5.5.5.1. Punto de entrada al sistema GPD

El archivo **detect_grasps.cpp** actúa como el ejecutable principal del sistema GPD. Su función es gestionar el proceso de detección de agarres, desde la lectura de parámetros y carga de datos hasta la llamada al detector. Su estructura se divide en las siguientes etapas:

1. Lectura de argumentos:

Al inicio del programa se comprueban que se han pasado todos los argumentos necesarios para la ejecución:

- CONFIG_FILE: archivo de configuración con los parámetros seleccionados **.cfg**
- PCD_FILE: archivo que contiene la nube de puntos **.pcd**
- [NORMALS_FILE]: archivo opcional con las normales de la nube precalculadas en formato **.csv**

2. Comprobación de archivos:

El sistema utiliza una función *checkFileExists* para verificar que los archivos proporcionados existen en las rutas especificadas.

3. Carga y lectura del archivo de configuración:

Se extraen los valores definidos por el usuario en el archivo de configuración. Estos incluyen propiedades de la cámara y pinza y opciones de preprocesamiento y filtrado de candidatos. Para ello se emplea una clase específica llamada *ConfigFile*. El archivo de configuración se explicará en profundidad más adelante.

4. Definición del punto de vista de la cámara

Se construye una matriz con el punto de vista desde el que se observa la nube de puntos. De esta manera se generan correctamente las imágenes virtuales de la pinza.

5. Carga de la nube de puntos

Se emplea una clase para cargar nubes de puntos desde archivos tipo **pcd** llamada *Cloud,* proporcionándole el punto de vista definido anteriormente.

6. Carga opcional de las normales

Si se ha proporcionado un archivo con normales calculadas se asignan a la nube. Esto puede resultar interesante si se desea utilizar un método más avanzado para el cálculo de normales o filtrarlas antes de procesar los agarres.

7. Inicialización del detector

Se crea una instancia de la clase *GraspDetector* con el archivo de configuración seleccionado. Esta clase contiene la lógica para detectar posibles agarres.

8. Preprocesamiento de la nube

A través de la función *preprocessPointCloud* del detector se realiza un preprocesamiento de la nube. Este preprocesamiento incluye pasos como:

- Voxelización (reducción de la resolución para facilitar el procesamiento)
- Filtrado por espacio de trabajo (para evitar agarres inalcanzables)
- Cálculo de normales (en caso de no ser proporcionadas)

9. Corrección de normales

El fichero tiene configurada una excepción que especifica que si el objeto está centrado en el origen deben invertirse las direcciones de todas las normales. Esta acción es importante ya que las normales calculadas pueden encontrarse orientadas hacia dentro del objeto, lo que produciría errores en la generación de imágenes para la CNN.

10. Detección de agarres

Finalmente, se llama a la función *detectGrasps* del detector que realiza la detección de poses de agarre válidas. A continuación, se explicará cómo se realiza este proceso.

5.5.5.2. Parámetros de configuración

Una de las principales funcionalidades que resultan ventajosas del modelo GPD es la posibilidad de adaptar el generador de agarres a las necesidades actuales del sistema en que se quiere introducir. Para ello se establecen varios archivos tipo **.cfg,** donde podemos modificar algunos de los parámetros del detector sin tener que modificar el código o la red neuronal ni recompilar el algoritmo.

En este caso, se establece como fichero de configuración el llamado **eigen_params.cfg**. Este archivo se divide en los siguientes campos:

Geometría y red neuronal:

En esta primera parte se especifican las rutas a los tres archivos fundamentales:

- Hand_geometry_filename: contiene la descripción de la geometría de la pinza
- Image_geometry_filename: define la geometría de la imagen utilizada como entrada a la red neuronal.
- Weights_file: directorio que contiene los pesos entrenados de la red neuronal.

En este caso la geometría de la pinza viene dada por el fichero **hand_geometry.cfg,** que contiene los siguientes detalles:

- finger_width: ancho de cada dedo de la pinza
- hand_outer_diameter: diámetro exterior de la pinza, equivale a la máxima apertura más el doble de ancho de los dedos
- hand_depth: profundidad de la pinza, es decir, longitud de los dedos
- hand_height: altura de la pinza, perpendicular al eje de apertura
- init_bite: penetración mínima en el objeto para considerarse agarre válido.

La definición de estos valores es fundamental tanto para la generación de candidatos válidos como para su evaluación, ya que nos determinarán que configuraciones son físicamente viables para el robot, adaptándose a la pinza.

Por otro lado, el archivo **image_geometry_15channels.cfg** nos indica la geometría del volumen 3D que se recorta alrededor de cada punto muestreado y cómo se convierte en una representación en formato imagen que se usa como entrada a la red neuronal. Este fichero contiene los siguientes parámetros editables:

- volume_width: ancho del volumen 3D extraído para cada muestra
- volume_depth: profundidad del volumen
- volume_height: altura del volumen
- image_size: resolución de la imagen cuadrada resultante
- image_num_channels: número de canales de la imagen

  Dependiendo del número de canales[13] elegido se obtendrá mayor o menor información para discriminar los agarres:

  - 3 canales: imagen proyectada en tres planos ortogonales XY, YZ, ZX
  - 12 canales: puntos, normales y curvaturas proyectados en cada plano
  - 15 canales: contiene los 12 canales anteriores y añade histograma de profundidad y mapa binario de colisión de los dedos.

[13] Canales: indica el número de imágenes que se crean por cada muestra, representando cada una de ellas una característica local.

Para definir estos parámetros se deben tener en cuenta los tamaños típicos de los objetos que se van a manipular y las dimensiones del gripper, además de la precisión que se desea tener y el coste computacional que esta puede tener.

Procesamiento de la nube de puntos:

Para el procesamiento de la nube de puntos podemos modificar los siguientes parámetros:

- Voxelize: activa la reducción de densidad mediante voxelización
- Voxel_size: define el tamaño del voxel. A más pequeño mayor detalle y tiempo de procesamiento
- Remove_outliers: si se activa se eliminan puntos aislados. Útil para eliminar ruido.
- Workspace: define el espacio de trabajo del robot, en formato cubo centrado en el origen de la nube de puntos
- Camera_position: define la posición de la cámara
- Sample_above_plane: si se activa descarta muestras que se encuentren en el plano de trabajo. Útil para evitar que se produzcan agarres en la mesa de trabajo.

Generación de candidatos de agarre:

Aquí se puede modificar qué se tiene en cuenta para la generación de candidatos. De esta manera podemos controlar la precisión y el coste computacional que necesitemos según el caso.

- num_samples: número de puntos desde los cuales se generarán candidatos
- num_threads: hilos de CPU usados en paralelo
- nn_radius: radio de vecindad para estimar el marco de referencia local
- num_orientations: número de orientaciones de la mano que se prueban por muestra
- num_finger_placements: número de posiciones de los dedos de la pinza que se prueban por orientación (desplazamiento lateral)
- hand_axes: eje sobre el que rota la vecindad
    - 0 = approach
    - 1 = binormal
    - 2 = axis
- deepen_hand: si se activa empuja la mano hacia delante en el objeto
- friction_coeff: ángulo de fricción en grados para determinar si un agarre es antipodal[14]
- min_viable: número mínimo de puntos necesarios a cada lado de la pinza para considerar el agarre válido

[14] Agarre antipodal: agarre en el cual los dedos aplican fuerzas enfrentadas sobre el objeto de forma que la fricción impide el deslizamiento.

Filtrado de candidatos:

En este grupo nos encontramos dos tipos de filtros que deben pasar los candidatos y podemos modificar: según apertura y dimensiones y según dirección de aproximación:

- Parámetros geométricos:
  - Min_aperture: mínima apertura del gripper
  - Max_aperture: máxima apertura del gripper
  - Workspace_grasps: zona en la que se aceptan candidatos (debe ser igual o menor a *workspace)*
- Parámetros de dirección de aproximación:
  - Filter_approach_direction: si lo activamos se filtrarán los candidatos según su orientación
  - Direction: vector de dirección preferida para la aproximación
  - Thresh_rad: umbral angular en radianes sobre la dirección.

Agrupamiento de candidatos:

Se definen si se desean agrupar los agarres similares con el parámetro **min_inliners,** que define el número mínimo de candidatos por grupo.

Selección de candidatos:

Con el parámetro **num_selected** elegimos el número de agarres finales que deseamos generar. Siempre se seleccionarán los agarres que obtengan mayor puntuación, en orden.

Visualización:

Por último, se permite al usuario elegir la visualización de las etapas que desea realizar. Activando o desactivando los siguientes parámetros se generarán estas visualizaciones o no. En caso de activarlos, se deben cerrar las ventanas una vez visualizadas para que el proceso continue, en caso contrario no es necesaria la interacción del usuario con el servidor.

Opciones de visualización:

- plot_normals: permite visualizar las normales del objeto
- plot_samples: visualiza las muestras generadas
- plot_candidates: muestra los candidatos iniciales
- plot_filtered_candidates: muestra los candidatos filtrados
- plot_valid_grasps: muestra los candidatos identificados como válidos
- plot_clustered_grasps: muestra los agarres después del agrupamiento
- plot_selected_grasps: muestra los agarres seleccionados, siendo estos la salida final del proceso.

5.5.5.3. *Generación de candidatos de agarre*

La clase **GraspDetector** es el núcleo del sistema GPD, es el encargado de detectar posibles posiciones de agarre sobre la nube del objeto. Como mencionamos en la explicación del ejecutable **detect_grasps,** este fichero hacia una llamada a **grasp_detector** para crear el detector de agarres basado en esta clase. A continuación, se explica cómo funciona este detector.

El archivo **grasp_detector.cpp** es el responsable de sostener el flujo de trabajo necesario para generar, evaluar y filtrar los candidatos de agarre. Tiene una estructura modular y llama a varios subprocesos para realizar algunas partes específicas del sistema.

En primer lugar, el módulo se encarga de inicializar el detector de agarres y cargar los parámetros definidos por el usuario, activando o desactivando las opciones de visualización.

El **generador de candidatos** se inicializa con la clase **CandidatesGenerator,** a la que se le proporcionan dos estructuras clave:

- Generator_params: contiene los parámetros relacionados con el procesamiento de la nube de puntos (voxelización, número de muestras, espacio de trabajo, etc)
- Hand_search_params: define las condiciones de búsqueda de los candidatos de agarre (orientaciones, colocación de los dedos, fricción, etc)

El clasificador de agarres se inicializa mediante la clase **Classifier,** especificando los archivos del modelo y los pesos de la red neuronal que se usará. Este componente será el encargado de evaluar cada candidato generado.

También se crea un objeto **ImageGenerator,** cuya función será convertir los candidatos generados en imágenes para ser clasificados por la red.

Además, se inicializan los parámetros de filtrado, dividiéndose en dos grupos: según *workspace* y ancho del *gripper* por un lado, y según la dirección de aproximación si se encuentra activa por otro.

La función **GraspDetector::detectGrasps** es la responsable de ejecutar el flujo de detección sobre la nube de puntos, para organizar la detección se siguen las siguientes etapas:

1. Chequeo inicial, verificando que la nube no se encuentre vacía
2. Visualización de muestras y normales si se ha habilitado
3. Generación de candidatos, a través de la función **generateGraspCandidateSets,** que devuelve un conjunto de

objetos **HandSet,** donde cada objeto representa múltiples posiciones de agarre posible en una región

4. Filtrado de candidatos, según las dos etapas establecidas anteriormente, y si se encuentra habilitado, se visualizan los agarres válidos
5. Generación de imágenes para la clasificación: *ImageGenerator* genera imágenes 2D para cada candidato
6. Clasificación de las imágenes y asignación de puntuaciones a cada candidato
7. Selección de los mejores N agarres, según su puntuación, a través de la función **selectGrasps.**
8. Clustering, si se encuentra activado, se agrupan los agarres similares.
9. Ordenación final y retorno: se devuelve el vector final **Hand** con los mejores agarres detectados

Además, en esta parte del procesamiento, se ha incluido una modificación que nos permite ejecutar nuestra aplicación: una vez filtrados los agarres, se convierten los objetos de tipo *unique_ptr,* a objetos regulares tipo *Hand* y se llama a una función que guarda la lista de agarres seleccionados en un archivo tipo **csv.** Este archivo es el que rescataremos posteriormente desde nuestra aplicación y contendrá los agarres que ejecutaremos en el brazo robótico.

```
// GUARDAR EN UN FICHERO LOS AGARRES FILTRADOS
if (!hands.empty()) {

    // Convertir unique_ptr a objetos Hand normales para usar writeHandsToFile
    printf("--------FUNCION FICHERO ----");
    std::vector<candidate::Hand> hands_to_save;

    for (const auto& hand_ptr : hands) {

      hands_to_save.push_back(*hand_ptr);

    }

    // Llamar a la función de guardado

    hands_to_save[0].writeHandsToFile("grasps_filtered.csv", hands_to_save);

    std::cout << "Saved filtered grasps to grasps_filtered.csv" << std::endl;

}else {
  printf("ELSE");
}
```

Ilustración 78. GPD: Llamada a la función writeHandsToFile() (Elaboración propia)

```
void Hand::writeHandsToFile(const std::string &filename,
                            const std::vector<Hand> &hands) const {
  std::ofstream myfile;
  myfile.open(filename.c_str());

  //Encabezado
  myfile << "X,Y,Z,Axis_X,Axis_Y,Axis_Z,Binormal_X,Binormal_Y,Binormal_Z,Approach_X,Approach_Y,Approach_Z,GraspWidth\n";

  for (int i = 0; i < hands.size(); i++) {
    std::cout << "Hand " << i << std::endl;
    print();

    myfile << vectorToString(hands[i].getPosition())
           << vectorToString(hands[i].getAxis())
           << vectorToString(hands[i].getBinormal())
           << vectorToString(hands[i].getApproach())
           << std::to_string(hands[i].getGraspWidth()) << "\n";
  }

  myfile.close();
}
```

Ilustración 79. GPD: Función writeHandsToFile (Elaboración propia)

5.5.5.4. Red neuronal

La pieza fundamental para el procesamiento de los agarres en GPD es la red neuronal profunda y entrenada para evaluar la calidad de los candidatos de agarre generados por el modelo.

Una vez que se ha preprocesado la nube de puntos y se han generado un conjunto de candidatos de agarre, el siguiente paso del sistema es determinar cuáles de estos candidatos representan agarres que se consideran válidos. Esta es la principal función de la red neuronal, recibe como entrada una representación 3D local del entorno alrededor de cada agarre y produce como salida una puntuación de viabilidad.

Esta puntuación se considera una medida de confianza para indicar cómo de bueno es un agarre o cuánto de probable es que resulte exitoso en la realidad. Posteriormente el sistema selecciona los candidatos de agarre con mayor puntuación como resultado final.

La red neuronal que utiliza GPD se basa en una arquitectura **LeNet,** una red convolucional sencilla y efectiva. Está compuesta por:

- Capas convolucionales para extraer las características locales de la representación de cada agarre
- Capas de agrupamiento para reducir la dimensionalidad y captar patrones en el espacio
- Capas totalmente conectadas para realizar la clasificación final

Gracias a esta arquitectura el sistema puede procesar múltiples candidatos en un tiempo razonable sin requerir excesiva carga computacional.

La red que se utiliza se encuentra entrenada offline mediante simulaciones o datos reales anotados, donde se sabe si un agarre es exitoso o no. Un buen entrenamiento es esencial para obtener una red capaz de generalizar nuevas escenas.

5.5.6. Procesamiento, filtrado y visualización de los agarres

Este módulo constituye la última etapa del flujo de trabajo, encargándose de procesar los agarres generados por el modelo GPD para adaptarlos a las necesidades del brazo robótico.

Este módulo recibe como entrada el archivo *Grasp_filtered.csv* generado por el servidor remoto y se han definido varias funciones para realizar un posprocesamiento de los agarres, filtrando aquellos que puedan resultar conflictivos, adaptarlos a los vectores del sistema y a una posición ergonómica para la garra del robot y mostrando gráficamente los resultados al usuario para que verifique los agarres antes de ejecutarlos.

El flujo principal se gestiona a través de la función **procesar_agarres(),** la cual realiza las siguientes operaciones:

1. Lectura del archivo CSV

Esta función, lee el archivo línea por línea, extrayendo los parámetros geométricos que definen los agarres, **posición, orientación** (3 vectores ortonormales: *axis*, *binormal* y *approach*) **y ancho estimado** para el agarre (*width*).

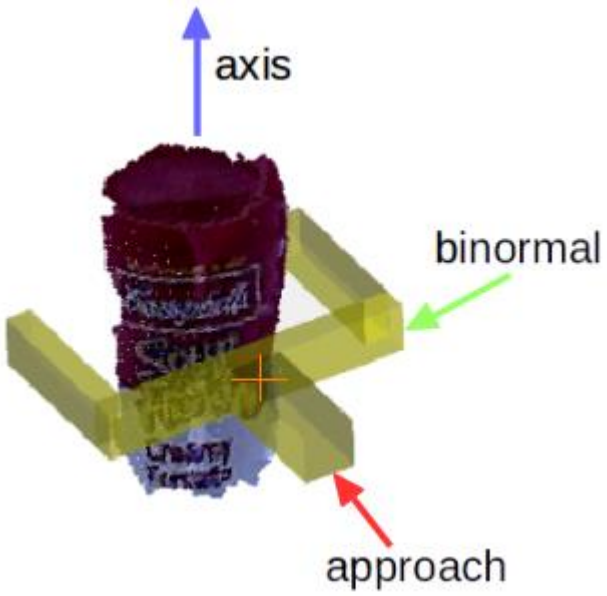

Ilustración 80. Geometría de los agarres generados por GPD. (Ten Pas A. , 2019)

El primer campo de cada agarre nos indica su **posición en el espacio,** con coordenadas X, Y, Z, en metros. Los siguientes parámetros (*axis*, *binormal* y *approach*) son tres vectores que representan la orientación de la pinza a través de los ejes principales del agarre como se muestra en la figura anterior. Por último, nos indica el ancho del agarre sugerido.

```python
def procesar_agarres(csv_path: str) -> None:
    print("\nCargando agarres desde:", csv_path)
    centroide = np.mean(np.asarray(pcd_obj.points), axis=0)

    try:

        with open(csv_path, newline='') as csvfile:
            reader = csv.reader(csvfile)
            next(reader)
            grasp_filtrados = []


            for idx, row in enumerate(reader, 1):
                valores = [float(x) for x in row]
                pos = np.array(valores[:3])
                axis = np.array(valores[3:6])
                binorm = np.array(valores[6:9])
                appr = np.array(valores[9:12])
                width = valores[12]
```

Ilustración 81. Módulo grasp.py: Lectura de parámetros del CSV. (Elaboración propia)

2. Clasificación del tipo de agarre

Tras realiza un estudio de los datos obtenidos de los agarres, se llegó a la conclusión que se podía determinar el tipo del agarre fácilmente en función de la componente "Y" del vector "*approach*". Entendiendo el tipo de agarre como **lateral** o **superior.**

Esta clasificación permite aplicar criterios de filtrado distintos según el tipo de orientación del agarre, para asegurar al máximo que el agarre que vamos a ejecutar se considera válido.

Además, se añade esta información al fichero con los agarres finales extrapolados al sistema de referencia de la cámara para facilitar al usuario la incorporación con el robot, dar la posibilidad de elegir el tipo de agarre que se necesita para la operación y evitar posibles conflictos.

3. Filtrado de agarres

Continuando con el estudio de los componentes obtenidos del detector de agarres, se decidió realizar un filtrado manual, adicional al que realiza el propio sistema, para garantizar la seguridad del agarre. Esta operación resulta realmente útil dada la complicidad que se añade al sistema el hecho de detectar agarres sobre objetos que han sido capturados desde solo una vista, por lo que su composición 3D se encuentra gravemente incompleta.

En este caso, se ha decidido descartar agarres que:

- Estén orientados hacia la cámara (componente Z de approach positiva)
- No cumplen con ciertas condiciones geométricas adaptadas al manipulador, como son ciertos rangos de orientaciones que tras el estudio resultaban conflictivas
- apuntan hacia fuera del objeto: se calcula el centroide del objeto y se realiza el producto escalar entre approach y el vector que apunta al

centroide desde la posición de agarre, pudiéndose controlar el umbral del ángulo que se desea obtener.

```
grasp_type = 'lateral' if appr[1] > 0 else 'superior'

# ------------------- FILTRADO DE LOS AGARRES -------------------
# Descartar agarres cuyo vector de aproximación apunta hacia la cámara
if  appr[2] > 0:
    continue
# Descartar agarres laterales con orientación conflictiva
elif grasp_type=='lateral' and (abs(axis[0]) > 0.2 or abs(binorm[0]) < 0.45 ):
    continue
# Descartar agarres superiores con orientación conflictiva
elif grasp_type=='superior' and (abs(appr[0]) > 0.2 or abs(axis[1]) < 0.4 ) :
    continue

# Filtrar agarres cuyo vector de aproximación no apunta hacia el centro del objeto
vector_hacia_centro = centroide - pos
vector_hacia_centro /= np.linalg.norm(vector_hacia_centro)
appr_norm = appr / np.linalg.norm(appr)
cos_angle = np.dot(appr_norm, vector_hacia_centro)

if cos_angle < -0.5:
    continue
```

Ilustración 82. Módulo grasp.py: Filtrado de los agarres. (Elaboración propia)

4. Corrección visual

Se ha realizado una función específica, cuya funcionalidad es corregir la orientación de los vectores para adaptarlos al manipulador. Tras realizar varias pruebas, agarres que resultaban muy positivos, se convertían en agarres conflictivos con el brazo robótico ya que el sistema GPD situaba la garra de manera que tuviera que realizar giros de 180 grados innecesarios.

Al encontrar esta problemática, se decidió optar por corregir estos agarres en vez de descartarlos. De manera que se evalúa la componente Y del eje *axis* y si esta es negativa, se invierte el vector por completo. De esta manera quedaría en la posición más ergonómica desde el punto de vista de la garra.

Para asegurar la ortonormalidad de los vectores, se recalcula el eje *binormal* en función de los dos anteriores.

```
def corregir_orientacion_grasp(pos, axis, binorm, appr):

    axis_corr = axis.copy()
    appr_corr = appr.copy()

    if axis_corr[1] < 0:
        axis_corr = -axis_corr

    binorm_corr = np.cross(appr_corr, axis_corr)

    return pos, axis_corr, binorm_corr, appr_corr
```

Ilustración 83. Módulo grasp.py: Función corregir_orientacion_grasp. (Elaboración propia)

5. Conversión al sistema de coordenadas de la cámara RealSense

Como se comentó en el módulo **pointcloud.py,** el sistema de coordenadas adquisición (RealSense) no coincide con el sistema de creación de las nubes de puntos (Open3D), resultando opuestas las direcciones de sus ejes Y y Z. Para la correcta creación de la nube de puntos y el funcionamiento óptimo del detector de agarres, se decidió invertir estos ejes y utilizar el sistema de

referencia de Open3D para el procesamiento de las nubes de puntos. Por tanto, los agarres que devuelve el detector se encuentran referenciados a este sistema.

Sin embargo, para interpolar las posiciones y orientaciones obtenidas al sistema del robot, se realiza la transformación entre el sistema de referencia de la cámara y la base del robot. Por tanto, se crea la necesidad de obtener los agarres en el sistema de referencia de la cámara.

Para resolver este conflicto, se ha creado una función dentro del módulo que revierte las coordenadas del sistema de referencia de Open3D al de RealSense de nuevo.

```python
def convertir_a_sistema_camara(pos, axis, binorm, appr):
    def revertir(vec):
        v = vec.copy()
        v[1] = -v[1]
        v[2] = -v[2]
        return v

    return revertir(pos), revertir(axis), revertir(binorm), revertir(appr)
```

Ilustración 84. Módulo graps.py: Función convertir_a_sistema_camara. (Elaboración propia)

6. Cálculo de orientación del Grasp

A partir de los vectores *binormal, axis y approach* ya convertidos, se construye una **matriz de rotación 3x3** que representa la orientación del sistema de referencia del Grasp. Con esta matriz se calculan:

- Los **ángulos de Euler** en grados, en orden **xyz** (roll, pitch, yaw), requeridos por el sistema de control robótico
- El **cuaternión de rotación,** que también puede ser utilizado por el robot para definir la orientación del efector.

Esta conversión se realiza con la clase *Rotation* de *scipy.spactial.transform,* lo que permite representar de forma compacta y precisa cualquier orientación 3D.

Estos formatos de orientación son los que posteriormente utiliza el sistema robótico real para planificar y ejecutar el movimiento. De este modo, el sistema no solo identifica posibles puntos de agarre, sino que entrega directamente al robot las instrucciones necesarias para realizar el movimiento de forma precisa.

```python
rot = R.from_matrix(np.column_stack((binorm_cam, axis_cam, appr_cam)))
euler = rot.as_euler('xyz', degrees=True)
quat = rot.as_quat()

grasp_final = {
    "type": grasp_type,
    "pos": pos_cam,
    "axis": axis_cam,
    "binorm": binorm_cam,
    "appr": appr_cam,
    "width": width,
    "euler": euler,
    "quat": quat
}
```

Ilustración 85. Módulo grasp.py: Cálculo de Euler y cuaternión (Elaboración propia)

7. Visualización de los agarres

Además de la generación de los datos necesarios para ejecutar el agarre, se han diseñado varias funciones que permiten obtener una representación visual del agarre que se propone.

De esta forma, el usuario, de manera muy intuitiva puede seleccionar el agarre que mejor le convenga dependiendo de la pieza que se esté tratando, el entorno, posibles conflictos a nivel robótico, etc. Además, permite al desarrollador o integrador del sistema evaluar los agarres del sistema antes de ejecutarlos, pudiendo interpretar errores no detectados anteriormente.

Para cada agarre válido se genera una escena visual en tres dimensiones que contiene:

- La nube de puntos del objeto segmentado
- Una garra virtual compuesta por dos dedos y una barra de unión, adaptada a las dimensiones de la garra que disponemos para la aplicación.
- Tres flechas 3D que representan los ejes *axis, binormal y approach,* con distintos colores representativos, de manera que queda perfectamente definida la orientación de la garra.

```
def crear_vector(vector, origin, color=[1, 0, 0]):
        arrow = o3d.geometry.TriangleMesh.create_arrow(
            cylinder_radius=0.002, cone_radius=0.004,
            cylinder_height=0.05, cone_height=0.02)
        arrow.paint_uniform_color(color)

        base_vector = np.array([0, 0, 1])

        axis = np.cross(base_vector, vector)
        if np.linalg.norm(axis) > 1e-6:
            axis /= np.linalg.norm(axis)
            cos_angle = np.dot(base_vector, vector)
            angle = np.arccos(np.clip(cos_angle, -1.0, 1.0))
            rotation = R.from_rotvec(axis * angle)
            rotation_matrix = rotation.as_matrix()
        else:
            rotation_matrix = np.eye(3)

        transform = np.eye(4)
        transform[:3, :3] = rotation_matrix
        transform[:3, 3] = origin
        arrow.transform(transform)

        return arrow
```

Ilustración 86. Módulo grasp.py: Función crear_vector. (Elaboración propia)

```python
def crear_garra_virtual(pos, binorm, appr, width=0.098, altura=0.022, grosor=0.0065, finger_length=0.038, color=[1, 1, 0]):

    grasp_meshes = []

    # Centro de cada dedo respecto al grasp
    offset = (width / 2) * binorm
    altura_offset = (finger_length / 2) * appr

    # Eje ortogonal al grasp
    eje_ortogonal = np.cross(binorm, appr)
    rot_matrix = np.column_stack((binorm, appr, eje_ortogonal))

    for signo in [-1, 1]:
        finger = o3d.geometry.TriangleMesh.create_box(width=grosor, height=finger_length, depth=altura)
        finger.paint_uniform_color(color)
        finger.compute_vertex_normals()

        finger.translate(-finger.get_center())  # Centrar la caja
        transform = np.eye(4)
        transform[:3, :3] = rot_matrix
        transform[:3, 3] = pos + signo * offset + altura_offset

        finger.transform(transform)
        grasp_meshes.append(finger)

    # Barra unificadora entre los dedos (en el plano base de los dedos)
    barra = o3d.geometry.TriangleMesh.create_box(width=width + grosor, height=grosor, depth=altura)
    barra.paint_uniform_color(color)
    barra.compute_vertex_normals()

    barra.translate(-barra.get_center())
    transform_barra = np.eye(4)
    transform_barra[:3, :3] = rot_matrix
    transform_barra[:3, 3] = pos

    barra.transform(transform_barra)
    grasp_meshes.append(barra)

    return grasp_meshes
```

Ilustración 87. Módulo grasp.py: Función crear_garra_virtual (Elaboración propia)

La escena se lanza sobre una ventana interactiva de Open3D a la vez que se imprimen por consola los datos del agarre, incluyendo en ambas, un identificador numérico para evitar la confusión entre los distintos agarres generados. Además, esta ventana al ser interactiva facilita el movimiento del usuario a través de ella para comprobar desde distintas perspectivas, el agarre que se va a ejecutar posteriormente.

8. Exportación

Una vez encontrados agarres óptimos, se exportan de nuevo a un archivo tipo CSV, incluyendo posición, orientación, ancho, ángulos de Euler asociados y cuaternión. Esta salida facilita la integración del sistema con el movimiento del brazo robótico de manera automatizada.

```python
def exportar_grasps(grasp_filtrados, output_path):
    with open(output_path, mode='w', newline='') as file:
        writer = csv.writer(file)
        writer.writerow([
            'type', 'x', 'y', 'z',
            'axis_x', 'axis_y', 'axis_z',
            'binorm_x', 'binorm_y', 'binorm_z',
            'appr_x', 'appr_y', 'appr_z',
            'width',
            'euler_x', 'euler_y', 'euler_z',
            'quat_x', 'quat_y', 'quat_z', 'quat_w'
        ])

        for g in grasp_filtrados:
            fila = (
                [g['type']] +
                list(g['pos']) +
                list(g['axis']) +
                list(g['binorm']) +
                list(g['appr']) +
                [g['width']] +
                list(g['euler']) +
                list(g['quat'])
            )
            writer.writerow(fila)
```

Ilustración 88. Módulo grasp.py: Función exportar_grasps. (Elaboración propia)

En las siguientes ilustraciones encontramos dos ejemplos de agarres generados sobre nuestro objeto en tres dimensiones.

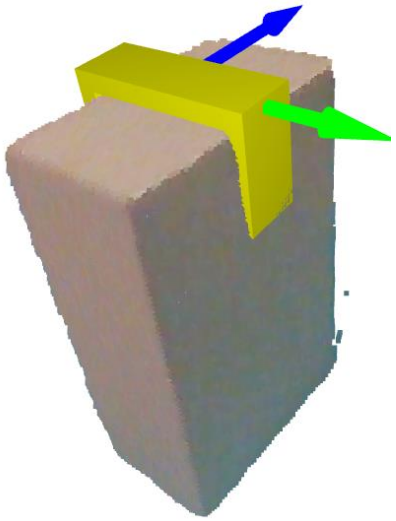

Ilustración 89. Agarre superior generado (Elaboración propia)

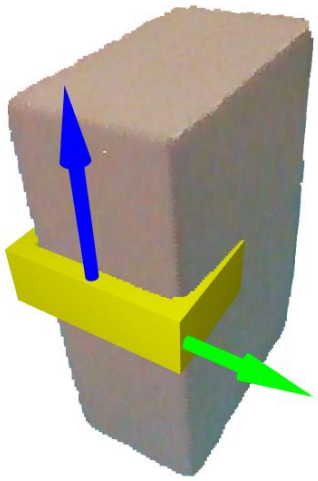

Ilustración 90. Agarre lateral generado (Elaboración propia)

6. Resultados

En este capítulo se presentan los resultados obtenidos tras la implementación de ambos enfoques de detección de agarres desarrollados: el método tradicional basado en geometría y el sistema automático basado en el modelo GPD. Se incluyen imágenes explicativas de cada etapa del proceso, así como una comparativa cualitativa entre ambos métodos, destacando sus principales ventajas e inconvenientes.

6.1. Resultados del sistema GPD

Para validar el correcto funcionamiento del modelo GPD y entender su proceso interno se han activado las distintas visualizaciones que el sistema ofrece mediante el fichero de configuración. Estas visualizaciones permiten observar las diferentes fases por las que pasa la nube de puntos antes de generar los agarres finales.

Esta visualización ha resultado de alta utilidad en el estudio inicial del detector de agarres y en su integración para la aplicación que se deseaba desarrollar, ya que permite entender las etapas del proceso que se lleva a cabo en el algoritmo y cómo estas se pueden adaptar al agarre deseado, siendo muchas de ellas configurables de manera sencilla en el propio fichero de configuración.

Para poder visualizar estas salidas debemos estar conectados mediante la aplicación de escritorio remoto de Windows al servidor, ya que las ventanas interactivas que nos mostrarán el objeto y el proceso de selección de agarres se visualizarán en el servidor.

En las siguientes imágenes podremos observar el flujo completo del sistema:

1. Normales del objeto (*plot_normals)*
2. Muestras generadas sobre la nube de puntos (*plot_samples)*
3. Candidatos iniciales de agarre (*plot_candidates)*
4. Candidatos filtrados por aproximación y apertura del gripper (*plot_filtered_candidates)*
5. Candidatos válidos para el agarre (*plot_valid_grasps)*
6. Candidatos de agarre agrupados (*plot_clustered_grasps)*
7. Agarres seleccionados (*plot_selected_grasps)*

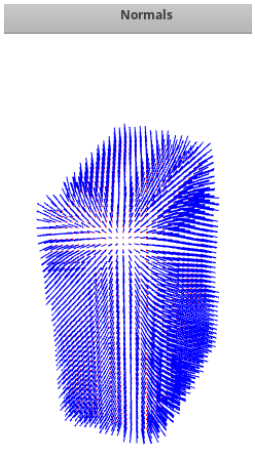

Ilustración 91. GPD Plots: Normals (Elaboración propia)

Ilustración 92. GPD Plots: Samples. (Elaboración propia)

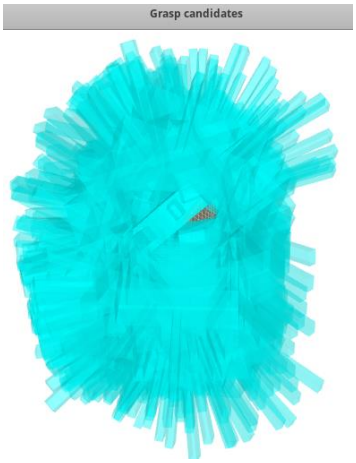

Ilustración 93. GPD Plots: Grasp candidates. (Elaboración propia)

Ilustración 94. GPD Plots: Filtered Grasps (Aperture, Workspace). (Elaboración propia)

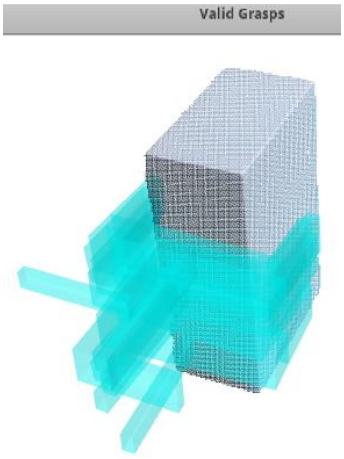

Ilustración 95. GPD Plots: Valid Grasps. (Elaboración propia)

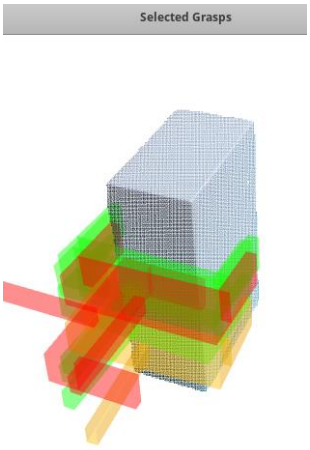

Ilustración 96. GPD Plots: Selected Grasps. (Elaboración propia)

La secuencia anterior representa el proceso modular que realiza el algoritmo interno de GPD para generar agarres óptimos. En función de los parámetros establecidos podemos realizar más o menos visualizaciones de las etapas, así como activar o desactivar etapas como el filtrado por dirección o la agrupación de los agarres y el número de agarres seleccionados finales que deseamos obtener. La versatilidad del sistema reside en la posibilidad de modificar estas visualizaciones de manera sencilla e instantánea, sin necesidad de cambiar configuraciones del código ni de volver a compilar el algoritmo.

Para integrar la aplicación desarrollada como un sistema completamente automático, estos *plots* se deben mantener desactivados, ya que si no el sistema espera la interacción del usuario con ellos para continuar el proceso.

Para comprobar el funcionamiento del sistema desarrollado se han realizado varias pruebas con diferentes objetos de forma y tamaño variable, sin que ninguno de ellos hubiera sido visto por el modelo durante su entrenamiento. A continuación, se muestran varias pruebas realizadas, siguiendo las distintas etapas del algoritmo, desde la captura de la imagen y segmentación del objeto hasta observar varios de los agarres generados por el detector, después de ser filtrados y adaptados para su uso directo por el brazo robótico.

Ejemplo 1:

Captura de la imagen y elección del objeto de interés:

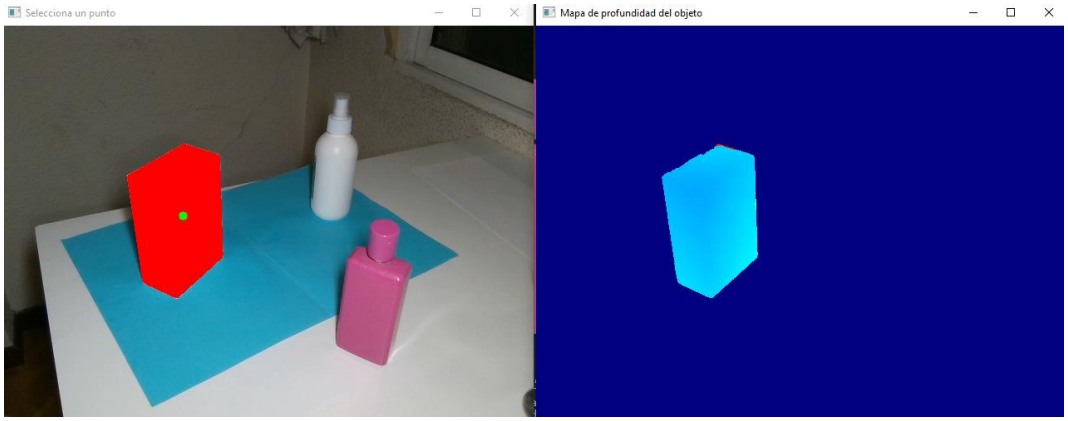

Ilustración 97. GPD. Ejemplo 1: Segmentación del objeto de interés. (Elaboración propia)

Agarres generados automáticamente:

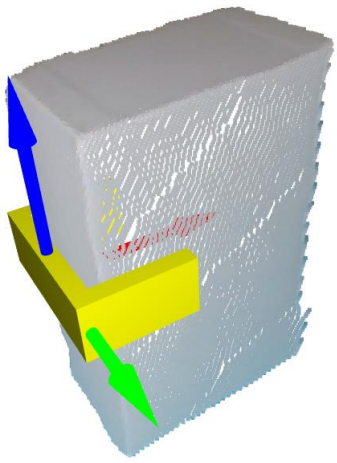

Ilustración 98. GPD. Ejemplo1: Agarre 1 generado (Elaboración propia)

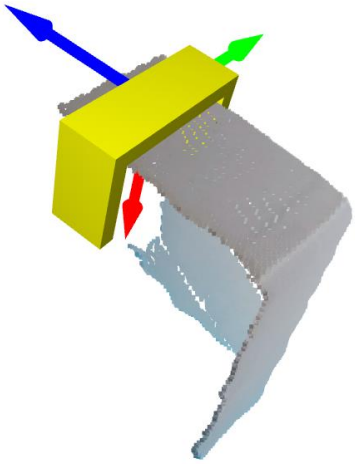

Ilustración 99. GPD. Ejemplo1: Agarre 2 generado (Elaboración propia)

Ejemplo 2:

Captura de la imagen y elección del objeto de interés:

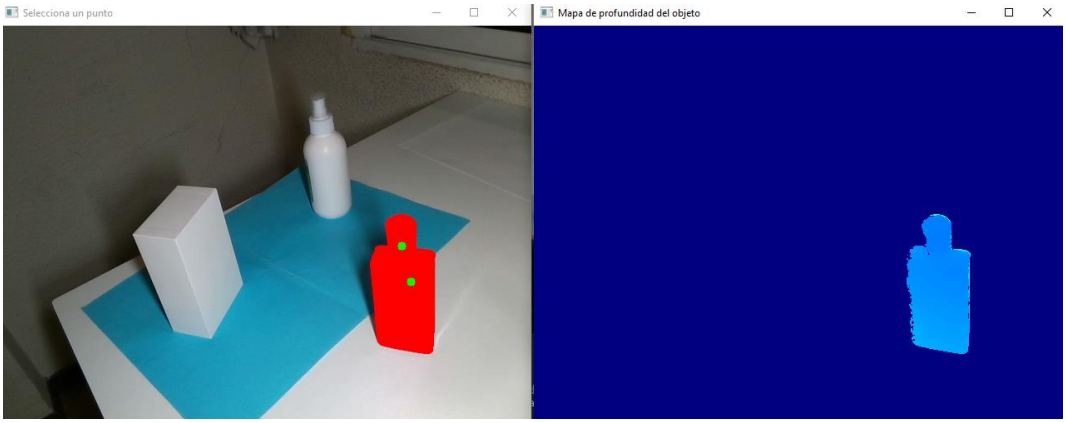

Ilustración 100. GPD. Ejemplo 2: Segmentación del objeto de interés. (Elaboración propia)

Agarres generados automáticamente:

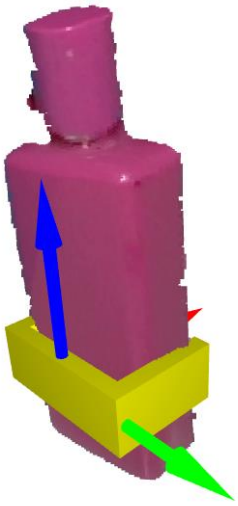

Ilustración 101. GPD. Ejemplo2: Agarre 1 generado (Elaboración propia)

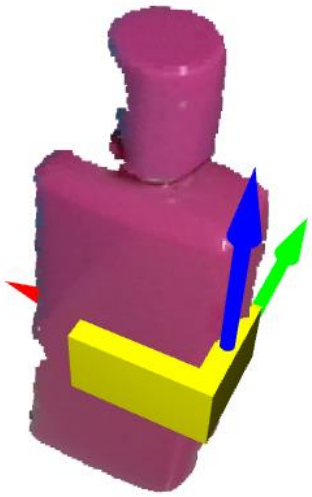

Ilustración 102. GPD. Ejemplo2: Agarre 2 generado (Elaboración propia)

Ejemplo 3:

Captura de la imagen y elección del objeto de interés:

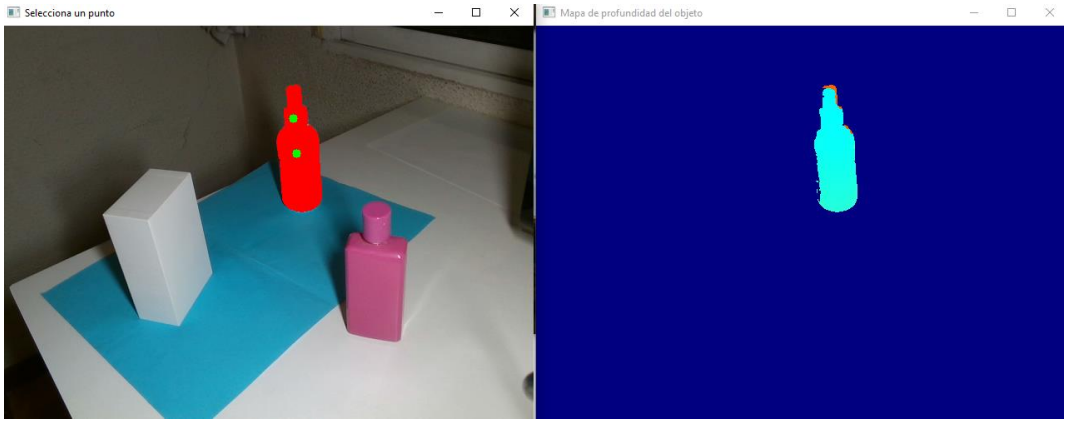

Ilustración 103. GPD. Ejemplo 3: Segmentación del objeto de interés. (Elaboración propia)

Agarres generados automáticamente:

Ilustración 104. GPD. Ejemplo3: Agarre 1 generado (Elaboración propia)

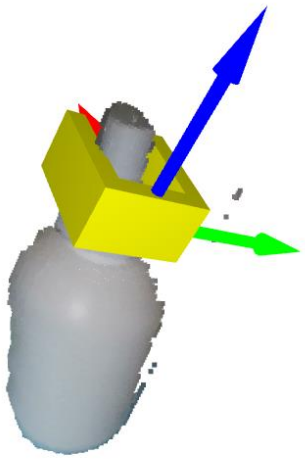

Ilustración 105. GPD. Ejemplo3: Agarre 2 generado (Elaboración propia)

Ejemplo 4:

Captura de la imagen y elección del objeto de interés:

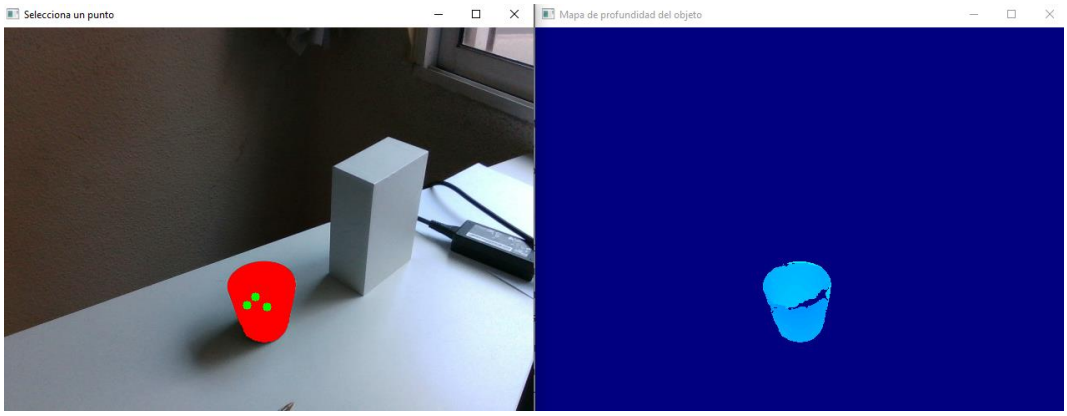

Ilustración 106. GPD. Ejemplo 4: Segmentación del objeto de interés. (Elaboración propia)

Agarres generados automáticamente:

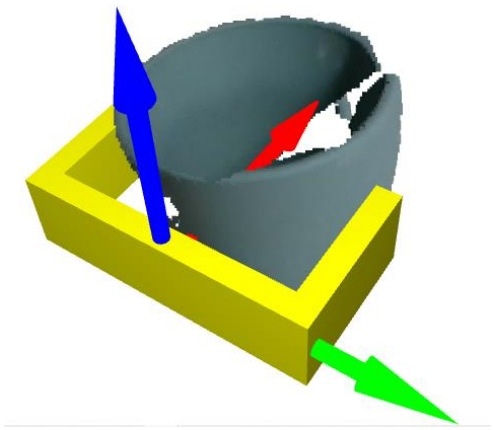

Ilustración 107. GPD. Ejemplo4: Agarre 1 generado (Elaboración propia)

Ilustración 108. GPD. Ejemplo4: Agarre 2 generado (Elaboración propia)

En la consola IPython que nos ofrece *Spyder* podemos observar los comentarios que nos indican algunas de las etapas del algoritmo, además de la información de cada uno de los agarres generados.

```
Puntos seleccionados: [(314, 228)]
Puntos seleccionados: [(314, 228), (337, 166)]
Imagen del objeto guardada como foto_0619_2109.
Nube de puntos guardada como nubes_pcd\nube_0619_2109.pcd
pcd guardada
Enviando nubes_pcd\nube_0619_2109.pcd al servidor...
CSV recibido y guardado como grasps_recibidos.csv

Cargando agarres desde: grasps_recibidos.csv

Grasp 1:
  Tipo de agarre: lateral
  Posición: [-0.019452  0.002128  0.44043 ]
  Ángulos Euler: [-177.95215765   -3.12503746    8.57302267]
  Cuaternión: [ 0.99663637  0.07519015  0.02585193 -0.019851  ]
  Ancho de agarre: 0.057439

Grasp 2:
  Tipo de agarre: lateral
  Posición: [-0.02243   0.022144  0.459379]
  Ángulos Euler: [-177.26402454   -4.92873863   13.13796205]
  Cuaternión: [ 0.99211576  0.11528074  0.03997503 -0.02861251]
  Ancho de agarre: 0.059691

Grasp 3:
  Tipo de agarre: lateral
  Posición: [-0.022426  0.006374  0.44689 ]
  Ángulos Euler: [-178.7642644    -1.80248125    5.08220004]
  Cuaternión: [ 0.99882749  0.04449741  0.01523454 -0.011469  ]
  Ancho de agarre: 0.058095

Grasp 4:
  Tipo de agarre: lateral
  Posición: [-0.019113  0.040468  0.472258]
  Ángulos Euler: [-178.43605241   -3.37226522    9.24537026]
  Cuaternión: [ 0.99619031  0.08095144  0.02822644 -0.0159685 ]
  Ancho de agarre: 0.058934

Grasp 5:
  Tipo de agarre: lateral
  Posición: [-0.019316  0.012547  0.450138]
  Ángulos Euler: [-178.33441163   -2.64789971    7.36531367]
  Cuaternión: [ 0.99754174  0.06454144  0.02212178 -0.0159845 ]
  Ancho de agarre: 0.057935
Grasp finales guardados en grasp_filtrados.csv
```

Ilustración 109. Consola IPython, procesamiento y resultados (Elaboración propia)

6.2. Resultados del método tradicional

Para estudiar las posibilidades de la aplicación desarrollada mediante el método tradicional de detección de agarres se han realizado varias pruebas con objetos cuadrados, de distintos tamaños y en distintas posiciones. Sobre ellos se calculan los contornos y se ajusta una forma geométrica simplificada, en este caso, una caja virtual, que servirá de modelo para detectar posibles agarres de acuerdo a la garra física de la que disponemos.

A continuación, se muestran distintas pruebas realizadas, en las que podemos comprobar que el sistema detecta correctamente un agarre válido y ajustado al brazo robótico del que se dispone.

Este método resulta robusto frente a objetos que se consideren ortoedros y podamos visualizar correctamente tres de sus caras principales, independientemente de la orientación de la que estos dispongan. Nos ofrece un algoritmo simplificado, que no requiere de un alto coste computacional ni de la necesidad de ejecución en servidores externos, con las consiguientes comunicaciones y problemática que puedan llevar a cabo.

El sistema se basa en operaciones matemáticas, acorde a las medidas encontradas del objeto, sin necesidad de tener su modelo CAD ni varias perspectivas del mismo. Sin embargo, presenta limitaciones de escalabilidad ya que no sería útil para objetos con distintas formas más complejas, por lo que para aquellos casos deberíamos emplear el sistema anterior, con modelos de aprendizaje profundo.

A continuación, se muestran varios ejemplos con objetos en distintas posiciones en los que se puede comprobar el proceso de detección del sistema y el agarre final generado.

Ejemplo 1:

Detección de contornos 2D:

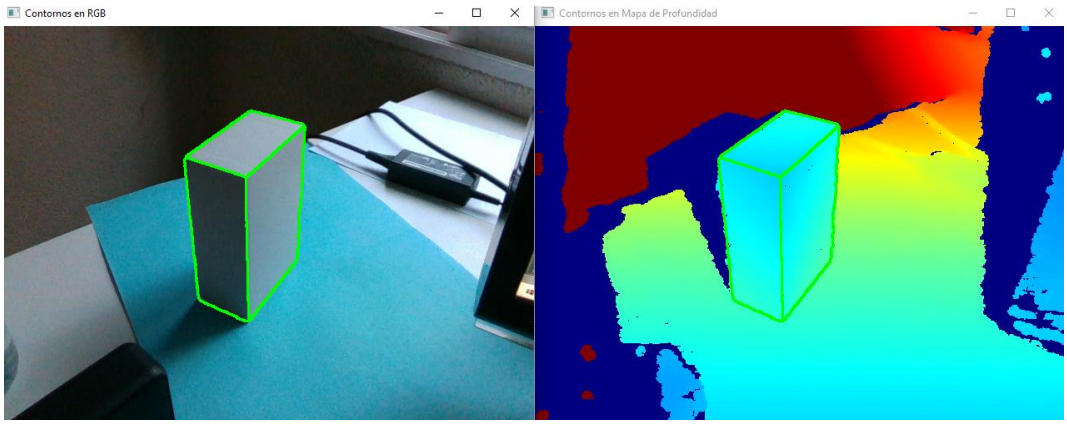

Ilustración 110. Sistema tradicional: Ejemplo 1: Detección de contornos 2D (Elaboración propia)

Contornos 3D, detección de superficies planas y creación de caja virtual

Ilustración 111. Sistema tradicional: Ejemplo 1: Contornos 3D (Elaboración propia)

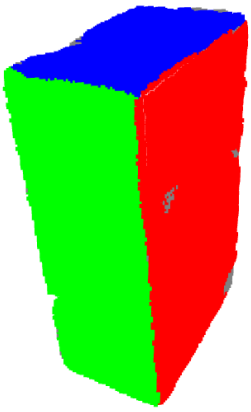

Ilustración 112. Sistema tradicional: Ejemplo 1: Detección de superficies planas (Elaboración propia)

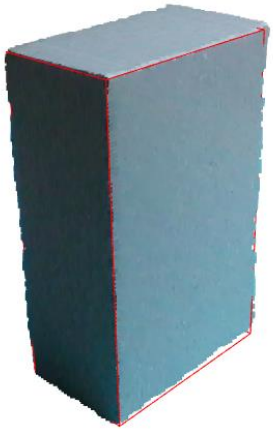

Ilustración 113. Sistema tradicional: Ejemplo 1: Caja virtual 3D (Elaboración propia)

Agarre generado

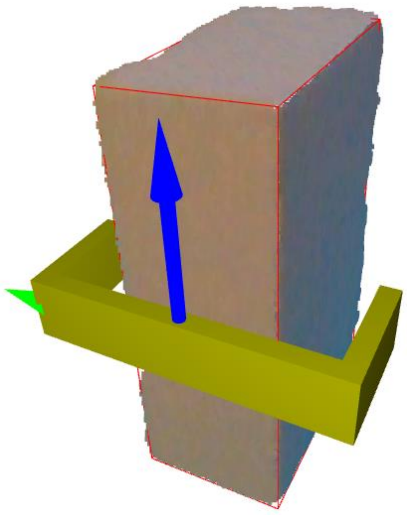

Ilustración 114. Sistema tradicional: Ejemplo 1: Agarre generado, perspectiva 1 (Elaboración propia)

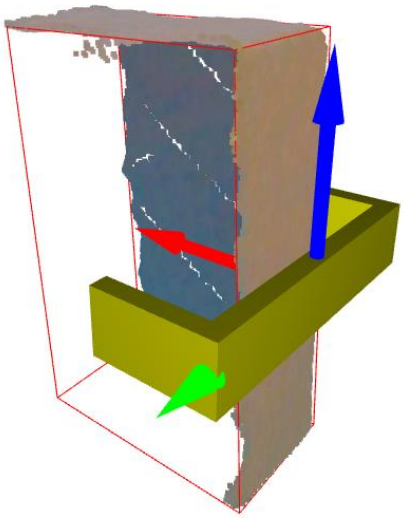

Ilustración 115. Sistema tradicional: Ejemplo 1: Agarre generado, perspectiva 2 (Elaboración propia)

Ejemplo 2:

Detección de contornos 2D:

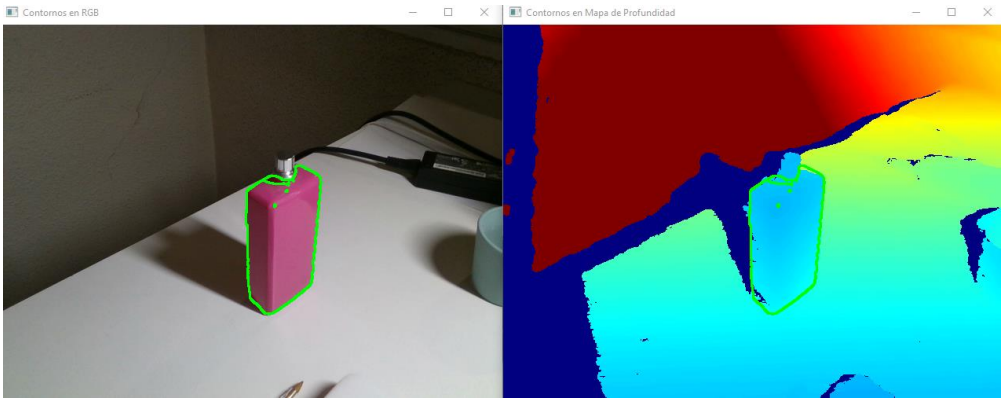

Ilustración 116. Sistema tradicional: Ejemplo 2: Detección de contornos 2D (Elaboración propia)

Contornos 3D, detección de superficies planas y creación de caja virtual

Ilustración 117. Sistema tradicional: Ejemplo 2: Contornos 3D (Elaboración propia)

Ilustración 118. Sistema tradicional: Ejemplo 2: Detección de superficies planas (Elaboración propia)

Ilustración 119. Sistema tradicional: Ejemplo 2: Caja virtual 3D (Elaboración propia)

Agarre generado

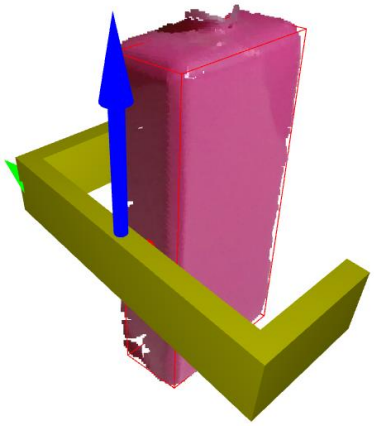

Ilustración 120. Sistema tradicional: Ejemplo 2: Agarre generado, perspectiva 1 (Elaboración propia)

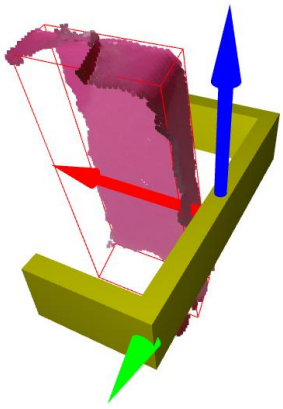

Ilustración 121. Sistema tradicional: Ejemplo 2: Agarre generado, perspectiva 2 (Elaboración propia)

Como se observa, los agarres generados para los distintos objetos se ajustan a la posición y orientación del mismo. El punto clave de este sistema es la correcta adaptación de la caja virtual al objeto real, ya que en caso de una detección errónea de las superficies planas principales el objeto virtual no se adaptaría a la realidad y el agarre, al generarse sobre la caja virtual no sería válido para el objeto.

6.3. Comparativa entre enfoques

A continuación, se realiza un pequeño comparativo de los dos sistemas creados, contemplando las ventajas y limitaciones de ambos, tanto funcionales como computacionalmente, mediante la siguiente tabla.

Criterio	GPD	Método tradicional
Generalización	**Alta** Permite trabajar con objetos desconocidos de distintas formas y tamaños	**Baja** Únicamente permite trabajar con objetos regulares y formas cuadradas
Coste computacional	**Alto** Necesita de un sistema de comunicaciones y ejecución en un servidor remoto	**Bajo** Procesamiento local sin grandes requerimientos hardware
Intervención del usuario	**Mínima** Elección del objeto y comprobación del agarre generado	**Mínima** Elección del objeto y comprobación del agarre generado
Calidad de los agarres	**Variable** Requiere de una comprobación humana posterior y un posible filtrado	**Media** Genera agarres controlados, pero poco flexibles
Robustez frente a oclusiones	**Moderada**	**Limitada**
Complejidad del algoritmo	**Alta**	**Media**
Ejecución	**Local + Servidor**	**Local**
Escalabilidad	**Alta**	**Limitada a formas simples**

7. Conclusiones

En este apartado, se muestran conclusiones generales de ambos sistemas desarrollados, así como futuras líneas de trabajo posibles.

- **Eficiencia dual de enfoques:**
  El método tradicional demostró ser más fiable en objetos regulares, sin embargo, su aplicación queda limitada a geometrías prismáticas y entornos más controlados. Por otra parte, GPD ofrece un sistema de detección con mayor robustez y capacidad de generalización, gestionando geométricas complejas y desconocidas con facilidad.
- **Automatización y comprobación visual:**
  A pesar de que ambos sistemas ofrecen resultados válidos, no resultan lo suficientemente robustos como para ser totalmente automatizados, ya que pueden generar agarres inesperados que produzcan colisiones o posiciones imposibles en el brazo robótico. Por tanto, es necesaria la comprobación visual por parte del usuario antes de ejecutar el agarre detectado.
- **Importancia del filtrado:**
  En el sistema que emplea el algoritmo GPD, uno de los puntos clave para el aumento de la tasa de éxito de los agarres es el post-procesado de los mismos, realizando un filtrado que descarta posiciones conflictivas y reorienta agarres para que resulten ergonómicos para el robot. No obstante, un filtrado exhaustivo requiere un entorno más controlado, por lo que se debe tener en cuenta el equilibrio entre la flexibilidad y la robustez que se requieran para la aplicación.
- **Comunicaciones**
  La comunicación remota con el servidor permitió aprovechar los recursos computacionales del mismo sin sacrificar la portabilidad ni la seguridad, además de generar un entorno más seguro y controlado donde se pueda monitorizar el algoritmo.

Futuras líneas de trabajo:

A partir del análisis realizado y las limitaciones detectadas durante el desarrollo del proyecto, se proponen las siguientes líneas de trabajo como continuación del sistema implementado:

- Desarrollo de algoritmos geométricos especializados:
  Se propone implementar métodos tradicionales adaptados a nuevas formas características, como objetos cilíndricos, esféricos o adaptables a distintas formas geométricas conocidas. Permitiría obtener un sistema sencillo y robusto frente a las principales formas conocidas.
- Entrenamiento personalizado del modelo GPD:
  Se podría mejorar la tasa de agarres válidos reentrenando la red neuronal de GPD con un conjunto de datos adaptado a los objetos más comunes que se quieran agarrar, incluyendo iluminaciones variables.

De esta forma se mejoraría la precisión del sistema y quizás la automatización total del mismo.

- Integración automática con el brazo robótico:
  La siguiente línea de trabajo clara a seguir es un módulo que permita la conexión automática entre los resultados de agarre generados por el sistema de detección y el planificador del brazo robótico. Este módulo debería incluir tanto la calibración entre la cámara y la base del robot, como la conversión de los datos de salida del sistema de visión al formato requerido por el planificador para su ejecución.
- Exploración de arquitecturas más modernas como GraspNet:
  Como futura línea de trabajo se propone estudiar la integración de redes como GraspNet, empleando para ello mayores recursos computacionales y rediseñando ciertos módulos del sistema.

Bibliografía

Anaconda Inc. (20 de Abril de 2025). *Anaconda Navigator*. Obtenido de https://www.anaconda.com/

Andreas ten Pas, M. G. (2017). Grasp Pose Detection in Point Clouds. *The International Journal of Robotics Research*, 1455–1473. doi:10.1177/0278364917735594

Bay, H. T. (2008). Speeded-up robust features (SURF). *Computer Vision and Image Understanding*, 346–359.

BCNVision. (19 de Mayo de 2025). *Tecnologías 3D para la industria*. Obtenido de https://bcnvision.es/blog-vision-artificial/tecnologias-3d-para-la-industria/

Bicchi, A., & Kumar, V. (2000). Robotic grasping and contact: A review. *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, (págs. 348–353).

Bimbo, J. (Marzo de 2016). *Touch based object pose estimation for robotic grasping*. Obtenido de https://www.researchgate.net/figure/Grasp-quality-according-to-the-Grasp-Wrench-Space-metric_fig46_324412506

ChatGPT. (2025).

Developers, N. (8 de Junio de 2025). *NumPy documentation.* Obtenido de https://numpy.org/doc/

Elaboración propia. (2024).

Emergent Vision Technologies. (10 de Junio de 2025). *Evolution of machine vision*. Obtenido de https://emergentvisiontec.com/es/tech-portal/evolution-of-machine-vision/

Fang, H.-S., Wang, C., Gou, M., & Lu, C. (2020). GraspNet-1Billion: A Large-Scale Benchmark for General Object Grasping. *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, (págs. 11444-11453). doi:10.1109/CVPR42600.2020.01146

Flask Documentation Team. (2024). *Flask Documentation*. Obtenido de https://flask.palletsprojects.com/en/stable/

Gou, M., Fang, H., Zhu, Z., Yi, S., Wang, C., & Lu, C. (2021). RGB Matters: Learning 7-DoF grasp poses on monocular RGB-D images. *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*

(pp. 13 459–13 466). IEEE. doi:https://doi.org/10.1109/ICRA48506.2021.9561409

GraspNet Team (MVIG, S. (15 de Mayo de 2025). *GitHub*. Obtenido de Graspnet-baseline: baseline model for GraspNet-1Billion dataset: https://github.com/graspnet/graspnet-baseline

HackerNoon. (26 de febrero de 2019). *Una breve historia de la visión artificial (y las redes neuronales convolucionales)*. Obtenido de https://hackernoon.com/lang/es/una-breve-historia-de-la-vision-por-computadora-y-las-redes-neuronales-convolucionales-8fe8aacc79f3

Haque, K. (3 de Abril de 2023). *What is Convolutional Neural Network — CNN (Deep Learning).* Obtenido de LinkedIn: https://www.linkedin.com/pulse/what-convolutional-neural-network-cnn-deep-learning-nafiz-shahriar/

Harris, C. R. (2020). Array programming with NumPy. *Nature*, 357–362.

Hsiao, K. K.-P. (2010). Grasping POMDPs. *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, (págs. 4685–4692). IEEE.

IBM. (5 de Junio de 2025). *Computer vision*. Obtenido de https://www.ibm.com/es-es/topics/computer-vision

IBM. (19 de Junio de 2025). *Neural networks*. Obtenido de https://www.ibm.com/es-es/think/topics/neural-networks

IBM. (4 de Mayo de 2025). *The history of computer vision*. Obtenido de https://www.ibm.com/es-es/think/topics/computer-vision

Intel Corporation. (25 de Mayo de 2025). *Intel® RealSense™ Depth Camera D415. RS Components*. Obtenido de https://es.rs-online.com/web/p/camaras-de-profundidad/1720980

Intel Corporation. (18 de Junio de 2025). *Intel RealSense Depth Camera D415*. Obtenido de https://www.intelrealsense.com/depth-camera-d415/

Kinova Robotics. (2018). Obtenido de https://www.kinovarobotics.com/product/gen3-robots

Kirillov, A. M. (1 de Junio de 2025). *Segment Anything. Meta AI.* Obtenido de https://segment-anything.com/

Lowe, D. G. (2004). Distinctive image features from scale-invariant keypoints. *International journal of computer vision*, 91–110.

Marcus Gualtieri, A. t. (2016). High Precision Grasp Pose Detection in Dense Clutter. *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, (págs. 598–605). doi:10.1109/IROS.2016.7759134

Nowosad, J. (2018). *Global dataset of gridded population and GDP*. Obtenido de https://www2.ccs.neu.edu/research/helpinghands/code/gpd/

OpenCV. (8 de Junio de 2025). *Open Source Computer Vision Library*. Obtenido de OpenCV: https://opencv.org

Paramiko Developers. (2023). *Paramiko Documentation*. Obtenido de https://docs.paramiko.org/en/stable/

Python Software Foundation. (2024). *sys — System-specific parameters and functions.* Obtenido de https://docs.python.org/3/library/sys.html

Python Software Foundation. (2024). *os — Miscellaneous operating system interfaces.* Obtenido de https://docs.python.org/3/library/os.html

Python Software Foundation. (2024). *subprocess — Subprocess management.* Obtenido de https://docs.python.org/3/library/subprocess.html

Python Software Foundation. (15 de Mayo de 2025). *tempfile — Generate temporary files and directories.* Obtenido de https://docs.python.org/3/library/tempfile.html

Reitz, K. &. (2023). *Requests: HTTP for Humans.* Obtenido de https://docs.python-requests.org/en/latest/

Robotiq. (15 de Junio de 2025). Obtenido de https://robotiq.com/es/productos/pinzas-adaptables#Two-Finger-Gripper

ROS Components. (17 de Mayo de 2025). Obtenido de https://www.roscomponents.com/es/producto/gen3-es/

Shin, N. (31 de Octubre de 2024). *5 Benefits of using 3D Machine Vision for your Pick and Place Robots.* . Obtenido de Zivid Blog: https://blog.zivid.com/5-benefits-of-using-3d-machine-vision-for-your-pick-and-place-robots

Spyder Developers. (n.d.). *Spyder IDE*. Obtenido de https://www.spyder-ide.org/

Tekvisa. (23 de Mayo de 2022). *Tecnologías de visión artificial 3D*. Obtenido de https://tekvisa.com/tecnologias-de-vision-artificial-3d/

Ten Pas, A. (2019). *GPD: Grasp Pose Detection*. Obtenido de GitHub: https://github.com/atenpas/gpd

Ten Pas, A. G. (2017). Grasp Pose Detection in Point Clouds. *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)* (págs. 2103-2110). Singapur: IEEE.

Ten Pas, A., Keil, C., & Platt, R. . (2021). *Efficient and accurate candidate generation for grasp pose detection in SE*. Obtenido de https://atenpas.github.io/psn/

Turing. (19 de Abril de 2025). Obtenido de https://www.turing.com/kb/all-you-need-to-know-about-computer-vision

Universal Robots. (28 de Abril de 2025). *Visión artificial en robots*. Obtenido de https://www.universal-robots.com/es/blog/vision-artificial-en-robots/

Van der Walt, S. C. (2011). The NumPy array: a structure for efficient numerical computation. Computing in Science & Engineering. 22-33.

VICOSYSTEMS. (20 de Abril de 2025). Obtenido de https://vicosystems.com/productos/pinza-2f-85/

Wang X, G. A. (2016). *Generative Image Modeling using Style and Structure Adversarial Networks*. Obtenido de https://www.cs.cmu.edu/~xiaolonw/publication1.html

Wang, Z. X. (2021). Obtenido de https://www.tandfonline.com/doi/full/10.1080/0951192X.2021.1946853

Zhang, Z. (2012). Microsoft Kinect Sensor and Its Effect. *IEEE MultiMedia*, 19(2), 4–10. doi:https://doi.org/10.1109/MMUL.2012.24

Zuo, Y. R. (2023). *Rapid Removal Algorithm of Road Surface Point Cloud Based on LP-RANSAC Algorithm*. Obtenido de https://www.researching.cn/articles/0J18a38bef91b25e95