UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN
DEPARTAMENTO DE INGENIERÍA ELÉCTRICA

ALGORITMO PARA MANIPULACIÓN DE OBJETOS EN UN ROBOT PR2

MEMORIA PARA OPTAR AL TÍTULO DE INGENIERO CIVIL EN COMPUTACIÓN E INGENIERO CIVIL ELÉCTRICO

IAN ALON FRANCISCO YON YON

PROFESOR GUÍA:
PABLO GUERRERO PÉREZ

MIEMBROS DE LA COMISIÓN:
JAVIER RUIZ DEL SOLAR
JOCELYN SIMMONDS WAGEMANN

SANTIAGO DE CHILE
2016

Resumen

Uno de los desafíos importantes para la Robótica, es la capacidad del robot de manipular objetos de su entorno, ya sea para transportarlos u operarlos de alguna manera. Si bien esta capacidad está prácticamente resuelta en ambientes controlados, es un problema abierto en el caso de robots autónomos y ambientes no controlados, dado que la forma de los objetos, sus características físicas y las cualidades del efector del robot no están acotadas. Como segundo requisito, se busca además que las soluciones sean robustas y funcionen en tiempo real para aumentar las aplicaciones reales de la robótica.

Una de las partes centrales de un algoritmo que permita manipular objetos es la detección de puntos de agarre. Esto corresponde a calcular los puntos del objeto por donde un robot debe tomarlo para que este no se caiga. Existen varios algoritmos que intentan dar solución a esta problemática pero solo funcionan para ciertas familias de objetos y en muchos casos toma demasiado tiempo realizar el cálculo.

En esta memoria se implementó un algoritmo de manipulación de objetos basado en un método del estado del arte. El algoritmo permite manipular objetos en tiempos razonables y no esta restringido a una familia específica de objetos, aunque los objetos manipulables requieren de cierta simetría axial.

El algoritmo se implementó en C++ en un robot PR2, un robot especialmente diseñado para investigación, usando Robot Operating System (ROS) como framework de desarrollo, lo que permitirá que este algoritmo sea usado fácilmente por otros equipos de investigación y robots en diferentes partes del mundo.

El algoritmo implementado consta de una etapa de filtrado y segmentación de una nube de puntos, la determinación de los puntos de agarre, muestreo de poses de agarre, descarte de éstas por diferentes criterios, la asignación de puntaje a los agarres y finalmente la ejecución del mejor agarre seleccionado.

Los experimentos muestran que el algoritmo permite tomar objetos en simulación y en un robot PR2 real.

A mi madre *Nolda* que siempre me apoya y soporta.

If a machine is expected to be infallible, it cannot also be intelligent. - Alan Turing

Agradecimientos

No sería nada sin mi madre que me ha acompañado, guiado y entendido; me enseñó a conocer el mundo con una mirada distinta de la común, y se mantuvo firme en su afán porque descubriera la música y a aquellos que me rodeaban, luchando todos los años porque no me faltara nada a pesar de las dificultades. Gracias por tu amor.

Gracias a mi familia: tíos, primos y abuelos que me han acompañado siempre aportando su granito de experiencia, enseñanza y amor. Gracias por su apoyo y compañía.

Gracias a Pablo Guerrero por ayudarme a encontrar este camino y estar siempre dispuesto a hablar a pesar de las decepciones. Gracias también a los profesores Javier Ruiz del Solar y Jocelyn Simmonds por su tiempo, buena disposición e invaluables consejos y correcciones.

Gracias a los que me conocieron cuando mi camino aún no era ingeniería, por los buenos momentos, la música, las juntas, las risas, y ese crecimiento juntos que siempre recordaré. Emiliano, Ricardo, Alonso, Gary, SW extendido y Team 509, Mati, Camilo, Franco, Juanito, Natalia. A aquellos que me iniciaron en la robótica, todos mis compañeros y profesor del laboratorio de robótica de Instituto Nacional, donde sudamos tanto, perdimos tantas competencias para al fin ganar una. Gracias también a los profesores que desde el colegio creyeron en mi y a esos compañeros que se sentaron conmigo a estudiar y consiguieron que me gustaran las matemáticas, sin su aliento la historia habría sido otra.

Gracias a aquellos que iniciamos juntos esta etapa y vivimos muchos de los momentos más divertidos de la vida, mis amigos de inducción, sección y muchos años más, Llama, Nadia, Sobi, Lina, Tomi, Lucho, Basi, Robi, Vale, Gordo. A aquel de las ideas locas y proyectos, Feña, y a todos los que conocí después y este último tiempo en la memoria, Cristóbal y Max.

Gracias a mis compañeros eléctricos por aceptarme y prestarme una mano aunque no me integrara tanto en la comunidad y a los DCC y la gente de la Salita por su infinita distracción, compañía y aventuras en las JCC y Encuentro Linux.

Gracias a aquellos en Kung Fu y a mis grandes amigos del coro, pues fueron la recarga de energía para la semana, un descubrimiento, una pasión y una amistad que valoro mucho. Gracias especiales a Gonzalo, Begoña, Oscar, Andrés, Iván, Ari, Sofía, Pablo P., Edo y Angie.

Finalmente muchas gracias a mi Ivy que me ha alentado, acompañado, ayudado y amado buena parte de este proceso. Nada sería lo mismo sin ti; no habría tanta luz por las mañanas ni tanto brillo por las noches. Muchas gracias.

Tabla de contenido

Índice de tablas

Índice de figuras

Índice de algoritmos

Capítulo 1

Introducción

1.1. Antecedentes generales y fundamentación

Una de las principales habilidades que se espera de los robots es su capacidad de interactuar con el medio, problema que está prácticamente resuelto para casos específicos y controlados, como se puede apreciar en las líneas de producción de automóviles o similares. Sin embargo si relajamos las restricciones y damos mayor capacidad de decisión a los robots aparecen muchos vacíos en diferentes áreas que impiden la masificación de la robótica de forma comercial en áreas menos controladas. Por esto el siguiente gran paso en la robótica es contar con robots autónomos robustos, funcionales y que puedan responder en tiempo real a los requerimientos que se les hace.

La existencia de robots autónomos robustos en ambientes no controlados tiene muchas aplicaciones, desde asistentes y traslado de objetos en industrias y oficinas hasta robots de cuidado de salud para enfermos o ancianos, lo que significa una reducción de costos en el cuidado y una mejor calidad de vida para estas personas. En estos escenarios es fundamental poder tomar objetos de variadas formas para transporte o uso, por lo que esta habilidad es indispensable para estas aplicaciones.

Los esfuerzos de investigación en el mundo han probado que la manipulación de objetos es un problema difícil en su forma general; a lo largo de los años se han probado diferentes mecanismos de interacción con los objetos: brazos de múltiples ejes, diferentes efectores (o grippers), que es la parte del brazo robótico que está en contacto con los objetos, considerando algunos de tipo pinza, tipo mano, efectores que adaptan su forma al objeto, entre otros, sin embargo muchos de estos esfuerzos solo sirven para problemas específicos o no cumplen con los niveles de éxito esperados, por lo que la búsqueda de un método de manipulación general se mantiene.

1.1.1. Grasping robótico

Para realizar grasping se ha de considerar los objetos que se manipularan, en que ambientes, y que se hará con ellos. Esta información determinará principalmente el tipo de efector a usar y los sensores para percibir el entorno y los objetos de interés.

Como este mundo lo hemos construido para que pueda ser usado por humanos, es natural pensar que imitar nuestra forma de manipular objetos es una buena respuesta desde el punto de vista mecánico, ya que permitiría manipular los objetos que los humanos pueden manipular. Debido a esto muchos de los esfuerzos se han centrado en manipuladores que imitan la mano humana o representan una simplificación de ésta, como es el caso de las pinzas que usa el robot PR2 (que se usará en este trabajo).

La visión es uno de los sentidos más usados por los humanos y por tanto es nuevamente natural pensar en sensores visuales para obtener información del mundo, aunque es difícil en ocasiones obtener información útil desde estos sensores, por lo que se han desarrollado otros sensores especializados que ayudan a determinar distancias de manera exacta.

Para poder manipular un objeto se debe tener información de su posición y orientación (pose), los sensores RGBD (como el sensor Microsoft Kinect) son los que han tenido más éxito últimamente debido a su bajo costo y buenos resultados. Estos sensores entregan una imagen con color y profundidad (ver Figura 1.1), con la que se construye un modelo interno del entorno que se usa para realizar el grasping (manipulación del objeto).

Los sensores utilizados (y por tanto los modelos construidos) tienen incertidumbre y pueden dar lecturas erróneas que causen errores en múltiples etapas del grasping, haciendo este problema muy difícil de abordar. Las principales etapas de la manipulación de un objeto (o grasping) se explican en las siguiente secciones.

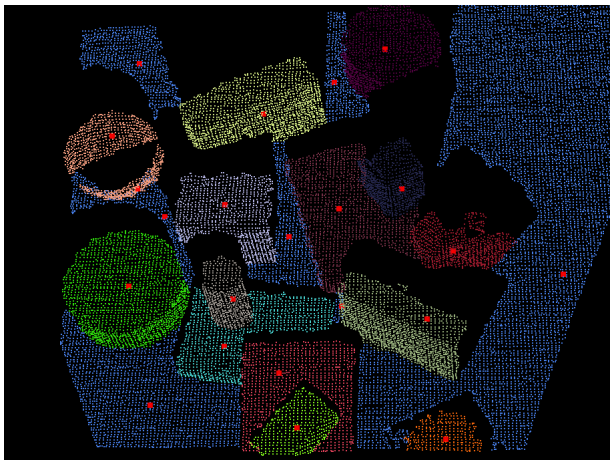


Figura 1.1: Ejemplo de nube de puntos y segmentación de objetos usando Kinect y procesándola con un algoritmo[1]de caminata aleatoria. En rojo se muestran los puntos iniciales del algoritmo.

[1] `http://www.pointclouds.org/blog/tocs/alexandrov/index.php`

1.1.2. Etapas del grasping

Se presentan las 5 etapas principales para manipular un objeto a partir de una nube de puntos.

1.1.2.1. Segmentación de la escena

Consiste en dividir una imagen (o nube de puntos) en grupos de pixeles (o puntos) con diferente significado con el fin de tener una representación más fácil de analizar. Para esto se le asigna una etiqueta a cada pixel de la imagen, de forma que aquellos que comparten etiqueta tienen características comunes entre si. En este caso se procesa una nube de puntos de la escena y se asignan etiquetas diferentes a puntos pertenecientes a distintos objetos de interés, así como a aquellos que corresponden a otros elementos no interesantes como la pared o suelo, que se etiquetan como fondo.

La segmentación tiene una gran variabilidad pues, dependiendo de lo que se quiera segmentar es necesario tener algoritmos específicos para obtener buenos resultados. En la figura 1.2 se muestra un ejemplo de segmentación, donde se asignan etiquetas diferentes a las personas y al avión.

Figura 1.2: Segmentación de personas y un avión con porcentajes de error

1.1.2.2. Reconocimiento de objetos

Corresponde a la tarea de detectar e identificar objetos de interés en una imagen, además de su ubicación espacial. Esta es una tarea difícil, dadas las diferentes condiciones físicas de los objetos en la escena:

- Pueden visualizarse desde diferentes puntos de vista
- Pueden estar expuestos a diferentes condiciones de iluminación
- Pueden ser de diferente tamaño del objeto de referencia
- Pueden estar ocluidos

Además los objetos no tienen una única representación física para cumplir una funcionalidad, por lo que el solicitar interactuar con lo que funcionalmente es una lámpara puede tener que lidiar con diferentes tipos de lámparas completamente diferentes en formas y tamaños como se ve en la Figura 1.3.

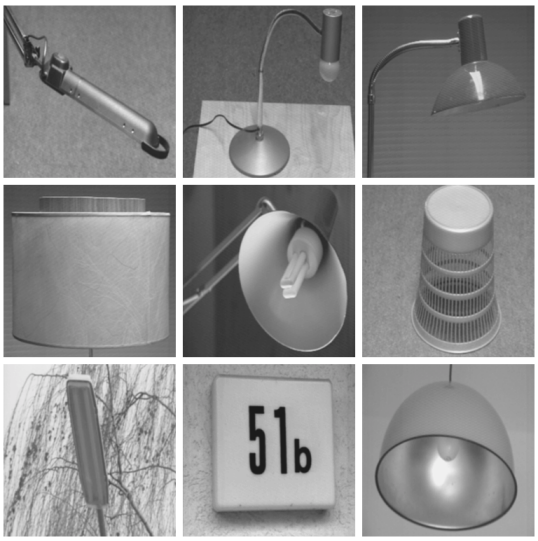


Figura 1.3: Diferentes tipos de lámparas

1.1.2.3. Cálculo de puntos de agarre

Consiste en encontrar los mejores puntos de la superficie de un objeto desde los cuales el efector del brazo robótico lo pueda tomar. En otras palabras, encontrar un punto desde el que se pueda tomar el objeto minimizando el riesgo de caídas y permitiendo ejercer la función de éste o la razón por la que se quiere manipular. En el caso de un vaso puede ser un par de puntos en la superficie de su cara cilíndrica o en una taza podría ser el asa, pero esta elección depende también de las características del efector, puesto que un efector tipo pinza probablemente no conseguirá un agarre estable en el asa.

1.1.2.4. Planificación de trayectoria

Consiste en encontrar un camino libre y conveniente para el brazo desde su pose inicial a la final, cerca del objeto a tomar. El camino debe considerar las restricciones de movimiento del brazo y los obstáculos que pueda haber en el camino, como mesas, otros objetos, personas, etc. Existen varias formas de solucionar este problema, siendo las más usadas los algoritmos basados en muestras (tipo Montecarlo[2]) y los basados en primitivas, que encadenan movimientos posibles incrementalmente.

[2]que pueden dar la respuesta correcta o respuesta erróneas (con probabilidad baja).

1.1.2.5. Cinemática inversa

Consiste en el cálculo de las posiciones de los motores del brazo [3], tales que éste alcance cierta configuración o pose en su efector. En general existe más de una solución pues hay varias configuraciones que llevan a la misma posición final. Existen varios algoritmos de solución tanto probabilísticos como analíticos, pero estos últimos, aunque más rápidos, pueden no encontrar soluciones en algunos casos.

1.1.3. Características de equipos

Para permitir que múltiples robots puedan realizar tareas similares, se hace necesaria la utilización de un framework que permita reutilizar algoritmos entre robots. Uno de los más utilizados es ROS (Robot Operating System) (Sección 2.2.1).

Una de las librerías más utilizadas para manipulación junto con el framework ROS es MoveIt! (Sección 2.2.3), la que permite calcular trayectorias de un brazo, cálculo de cinemática inversa, detección de colisiones y movimiento del brazo.

En el laboratorio de robótica del Departamento de Ciencias de la Computación (RyCh) se cuenta con un robot PR2, el que está diseñado para investigación e integrado con varios sensores, siendo el más relevante una cámara Microsoft Kinect y 2 brazos con pinzas como efectores.

1.2. Motivación

Las múltiples complicaciones de la manipulación de objetos y la necesidad de autonomía por parte de los robots obliga a encontrar un algoritmo que se comporte de manera aceptable para una amplia gama de objetos dentro de las capacidades mecánicas del robot, su rango de acción y además funcione en tiempo real. Para esto se plantea acotar el espectro y generar comportamientos autónomos considerando ambientes de casa u oficina, ya que éstos representan una gran variabilidad de objetos manipulables por personas con una sola mano, a diferencia de ambientes industriales donde se puede necesitar maquinaria especial, herramientas, o mecanismos de alta potencia para la manipulación efectiva. Se espera poder extender a futuro el algoritmo a más escenarios de interés, para los casos de objetos manipulables por personas con una sola mano.

La implementación de un manipulador es útil al robot PR2 disponible en RyCh dado que se necesita una base eficiente y robusta para hacer investigación de alto nivel, y así tener un competidor serio para posibles competencias robóticas en esta materia.

Basar el algoritmo implementado en un método del estado del arte sirve de apoyo a la comunidad nacional e internacional en robótica dado que disponibiliza de forma abierta un algoritmo que permite abstraer la etapa de manipulación para ciertas situaciones con lo que

se podrían construir mejores soluciones en la industria con menos esfuerzo, o usarlo como punto de partida para aplicaciones específicas.

1.3. Objetivos del proyecto

1.3.1. Objetivo general

Implementar un algoritmo de agarre robótico, en un robot PR2 simulado, basado en un método del estado del arte que permita realizar manipulación en tiempo real en un ambiente de casa u oficina sin considerar una familia específica de objetos a priori.

1.3.2. Objetivos específicos

1. Actualizar el software del PR2 a una versión soportada de ROS e integrar una librería para facilitar tareas de manipulación.
2. Analizar las etapas del grasping robótico.
3. Evaluar algoritmos de cálculo de puntos de agarre
4. Diseñar e implementar un algoritmo eficiente para manipulación de objetos.
5. Ajustar parámetros del algoritmo para lograr que el robot PR2 en simulación manipule objetos no vistos anteriormente en diferentes circunstancias con un tiempo de cálculo cercano a 1 segundo.
6. Evaluar desempeño del algoritmo implementado.
7. Presentar el algoritmo siguiendo las normas de la comunidad de ROS, de forma extensible y clara para permitir su modificación a futuro y como una herramienta útil a nuevas investigaciones.

1.4. Estructura de la memoria

En este primer capítulo se explican los conceptos generales más importantes y se exponen los objetivos generales y específicos. En el capítulo 2 se hace una revisión de todos los antecedentes teóricos necesarios para comprender el trabajo realizado, así como los principales frameworks y librerías utilizadas. En el capítulo 3 se detallan las características de la implementación del algoritmo y sus parámetros, así como las características del software creado. En el capítulo 4 se muestran y analizan los resultados obtenidos sobre datos grabados y de las pruebas en simulación. Finalmente se muestran las conclusiones de la memoria y el trabajo futuro que podría realizarse.

Capítulo 2

Contextualización

El presente capítulo tiene por objetivo ubicar al lector en el entorno en el cual se desarrolla este trabajo de título, entregando los antecedentes previos y necesarios para su contextualización. Se describirán los conceptos principales asociados al tema del trabajo de título y se mostrará el estado del arte para cada tópico.

En primer lugar, en la sección 2.1, se indican las características generales del proceso de manipulación robótica de objetos para sistemas autónomos. Se dará una explicación de los componentes de software y hardware importantes en la sección 2.2. En la sección 2.3 se detalla cada una de las etapas de la manipulación robótica de objetos. Se hace una revisión de varios algoritmos de cálculo de puntos de agarre en la sección 2.4. Finalmente se habla de los aportes de este trabajo.

2.1. Manipulación robótica de objetos

En la actualidad existen robots autónomos capaces de manipular un gran número de objetos (Ver Figura 2.1). Sin embargo, la mayoría de los sistemas que funcionan razonablemente requieren de un modelo 3D del objeto, el cual no siempre está disponible, o solo pueden manipular objetos de ciertas categorías, disminuyendo considerablemente su precisión al tratar con objetos desconocidos. Los algoritmos que tratan con objetos desconocidos deben incrementar su tiempo de procesamiento para poder manipular el objeto, lo que es poco deseable para interacciones en tiempo real. Los robots autónomos comerciales de la actualidad, en tareas de ordenamiento de objetos, recogida y descarga, toman hasta 8 veces más tiempo del que le tomaría a un humano realizar la misma tarea, lo que los hace inútiles para reemplazar a un humano y tediosos para relacionarse con personas en interacciones cotidianas.

Se revisarán los algoritmos más importantes de las diversas etapas en la sección 2.3.

Figura 2.1: Escena real de grasping [10]

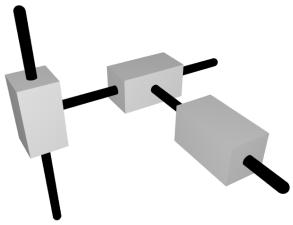
2.1.1. Dificultades

El grasping robótico es un problema complejo debido a la gran variabilidad respecto del robot. Existen diferentes arquitecturas para los brazos robóticos en búsqueda de aumentar los grados de libertad, osea el número de movimientos independientes posibles del efector, o suavizar ciertos movimientos:

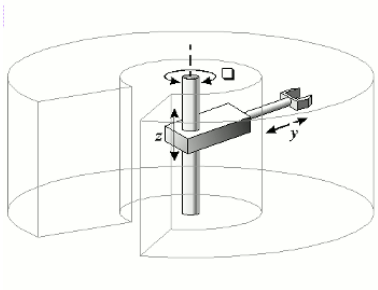
- Naturaleza de los efectores
  - Pinzas
  - Antropomórficos
  - Sujeción: Ventosas, electro-imanes, efectores deformables, etc.
- Naturaleza del brazo (ver Figura 2.2)
  - Cartesiano: Se compone generalmente de tres articulaciones, cuyos ejes son coincidentes con los ejes cartesianos.
  - Cilíndrico: Los ejes de este brazo forman un sistema de coordenadas cilíndricas.
  - Esférico: Los ejes del brazo forman un sistema polar de coordenadas.
  - SCARA: Es un robot que tiene dos articulaciones rotatorias paralelas para proporcionar elasticidad en un plano.
  - Articulado: Es un robot cuyo brazo tiene como mínimo tres articulaciones rotatorias.
  - Paralelo: Es un robot cuyos brazos tienen articulaciones prismáticas o rotatorias concurrentes.

Y la variabilidad relativa a la situación de grasping que se enfrenta:

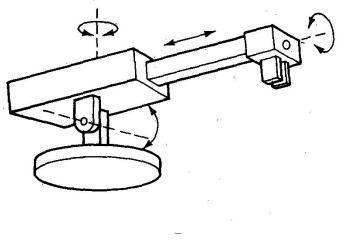
- Naturaleza de los objetos



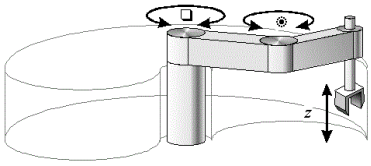
(a) Cartesiano



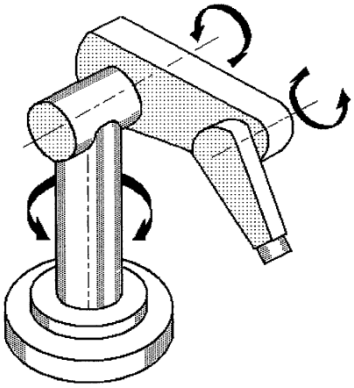
(b) Cilíndrico



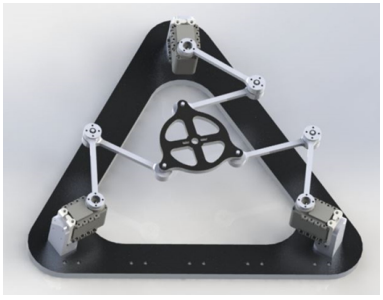
(c) Esférico



(d) SCARA



(e) Articulado



(f) Paralelo

Figura 2.2: Tipos de brazos robóticos

  - Objetos comunes: tazas, botellas, cajas, pelotas
  - Objetos de difícil manipulación: libros, cajas grandes, platos, contenedores con líquidos.
  - Objetos deformables: mochilas, ropa, comida, granos.
  - Objetos amorfos o no comunes: arte, diseños extravagantes, etc.
  - Objetos vivos/móviles: plantas, animales, mecanismos móviles, objetos en cintas transportadoras, objetos voladores
- Naturaleza de la escena
  - Iluminación
  - Disposición espacial de los objetos
  - Distancia relativa del robot
  - Objeto de interés
  - Tarea encomendada
  - Obstáculos móviles

A estas consideraciones hay que agregar que existe incertidumbre en el sensado, por los que estos errores pueden causar problemas en múltiples etapas del grasping, haciendo este problema muy difícil de abordar de forma general. Existen algoritmos que resuelven el problema de grasping para clases específicas de objetos, sin embargo hace falta un esquema que pueda enfrentarse a situaciones desconocidas, y los algoritmos actuales en el área aún son mejorables.

2.2. Componentes de software y hardware

Para poder implementar un algoritmo de manipulación robótica es necesario hacer muchas más cosas que el algoritmo de manipulación, por lo que se usan muchos frameworks que ayudan a controlar el robot y sus diferentes elementos. A continuación se explican los frameworks utilizados, librerías y la arquitectura del robot usado.

2.2.1. ROS

ROS es un meta-sistema operativo[1] de código libre mantenido por la Open Source Robotics Foundation (OSRF), que permite el desarrollo de software para robots. Su principal virtud es el hecho de que favorece la integración de nuevos robots y la re-utilización del software creado para un robot específico en otros robots distintos. ROS funciona sobre sistemas Unix, soporta oficialmente programación en C++, Python y Lisp; y cuenta con una gran cantidad de robots disponibles, como el Willow Garage Turtlebot, Willow Garage PR2, Lego NXT, Aldebaran Nao, IRobot Roomba, Kobuki, entre otros.

[1] Debe correr sobre un sistema operativo normal

ROS será usado en conjunto con el robot PR2 de Willow Garage para implementar y probar los algoritmos que se desarrollarán.

2.2.1.1. Conceptos

En general ROS funciona como una arquitectura cliente-servidor (Ver Figura 2.3). A las aplicaciones de ROS se les llama nodos, y los canales de comunicación se conocen como tópicos. Los nodos se suscriben a tópicos publicados por otros nodos, y de esta forma los nodos que están publicando en el tópico le envían la información relevante a los nodos suscritos.

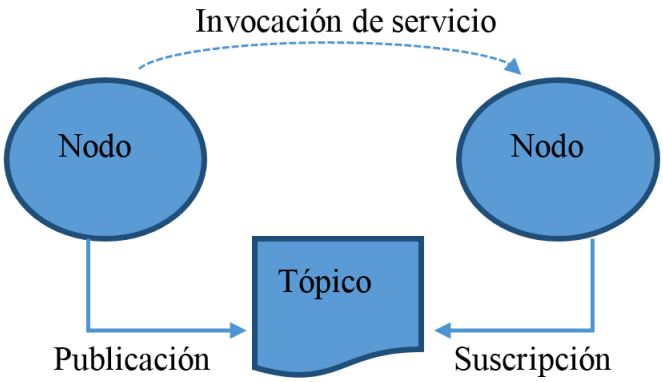


Figura 2.3: Diagrama de funcionamiento de ROS

Nodo Es un programa ejecutable en ROS, dedicado a una función específica, como interactuar con actuadores o sensores, procesar datos, entre otros. Para comunicarse con otros componentes del sistema hace uso del sistema de mensajes de ROS.

Mensaje Es una estructura de datos definida estáticamente que sirve para enviar información entre nodos. Se declaran en archivos de texto simple con un formato determinado y las herramientas de compilación generan las clases para poder ser usados en los nodos.

Tópico Permite que los nodos se comuniquen con mensajes, suscribiéndose o publicando en un tópico dado. Se tiene una relación n a n entre nodos y tópicos pudiendo un nodo suscribir y publicar simultáneamente en cuantos tópicos quiera.

Servicio Es una forma de comunicación síncrona entre nodos que usa mensajes diferentes para la petición y la respuesta. La relación entre nodos y servicios es *1* a n, muchos nodos pueden llamar a un servicio pero éste solo responde al que hizo la petición.

Paquete y Meta-paquete Un nodo o conjunto de nodos, sus servicios, mensajes definidos y otros archivos se conocen como paquete, y un conjunto de paquetes como un meta-paquete.

Servidor de parámetros: ROS provee un servidor para guardar parámetros que permite que distintos nodos puedan compartir información, mantener un estado por defecto para el nodo o cambiar parámetros internos sin tener que recompilar. Para acceder a estos parámetros se puede usar la herramienta `rosparam` en línea de comandos, las librerías clientes en Python y C++ o la librería y herramienta `dynamic_reconfigure` (ver párrafo en la sección 2.2.1.2).

La herramienta `rosparam` tiene un costo asociado para aplicaciones en tiempo real, dado que las consultas y respuestas de este se entregan usando el sistema de mensajes de ROS por red, lo que podría producir una pausa en la ejecución esperando la respuesta por red. Este impacto se vería en condiciones de alta pérdida de paquetes y con llamadas a `rosparam` dentro de un ciclo pues para cada llamada habría que esperar la respuesta.

2.2.1.2. Herramientas

dynamic_reconfigure: `dynamic_reconfigure`[2] es una herramienta que usa una función de callback en el código de usuario para atender las modificaciones en los parámetros del nodo, permitiendo modificar variable globales directamente. Es una forma eficiente de modificar parámetros, ya que solamente se llama al Servidor de parámetros cuando se modifica el parámetro y no en cada ciclo del programa.

`dynamic_reconfigure` además provee una interfaz gráfica, en la que se pueden modificar los parámetros. Para esto cuenta con un archivo de configuración donde se pueden separar los parámetros en grupos y se definen valores mínimos, máximos, por defecto y tipo de datos para cada uno. Los grupos de parámetros se puede definir como visibles, invisibles o colapsables, de forma de no mostrar parámetros cuando no son necesarios.

Roslaunch: `roslaunch`[3] es una herramienta que permite la ejecución de múltiples nodos, definiendo parámetros y opciones con una sola llamada. Hace uso de archivos XML o YAML (lanzadores) de extensión *'.launch'* para definir nodos a correr o incluso otros archivos *launch*. Esto permite modularizar la ejecución de un paquete.

Visualización con rviz: `rviz`[4] permite la visualización de datos de sensores, modelos de movimiento de robots, y otros datos mediante la suscripción a los tópicos correspondientes y la selección de plugins adecuados al dato seleccionado. Además permite interactuar con los robots dándoles instrucciones sobre planificación de trayectorias, movimiento, entre otros; usando los plugins para cada caso. Existe un plugin que funciona con MoveIt! para el PR2.

rosbag y rqt_bag: Son herramientas que permiten guardar la información publicada en diferentes tópicos en archivos, permitiendo la reproducción de éstos a futuro. Esto permite

[2] `http://wiki.ros.org/dynamic_reconfigure`
[3] `http://wiki.ros.org/roslaunch`
[4] `http://wiki.ros.org/rviz`

en un sistema separar el proceso de adquisición de datos del de procesamiento. `rosbag`[5] es la versión de línea de comandos de `rqt_bag`[6], que tiene interfaz gráfica.

En el caso particular de la Kinect, no se puede grabar la nube de puntos en el formato de trabajo (`pcl::PointCloud<pcl::PointXYZ>`), porque es un formato descomprimido, y la alta cantidad de datos recibidos saturaría la red y el disco. En cambio se guarda su versión en bruto comprimida y luego se procesa usando el nodo *openni*, que la descomprime. Para ello es necesario iniciar *openni* con un *id* de dispositivo inválido, de forma de que tome la información publicada por `rosbag` en vez de esperar por una Kinect. *openni* publica en los mismos tópicos que la Kinect (*/camera/depth/points*), haciendo transparente, para el resto del sistema la existencia de una Kinect real o datos grabados.

Gazebo: Gazebo[7] es un simulador de robots que se integra perfectamente con ROS, aunque funciona independiente de este también. Permite simular entornos con objetos genéricos o diseñados, para ver el comportamiento de algoritmos antes de probarlos en robots reales. Existen paquetes de integración con ROS para el robot PR2 que simulan los controladores del robot siguiendo las interfaces de ROS.

2.2.1.3. actionlib

Es un paquete de ROS que presenta una forma estándar de comunicarse con tareas de largo plazo que sean cancelables. Trabaja definiendo un cliente y un servidor que se comunican en función de metas a cumplir solicitadas por el cliente, y funciones de callback y resultado entregado por el servidor. El servidor mantiene una máquina de estados para cada acción solicitada y el cliente intenta copiar el estado del servidor para informar al usuario el estado de su solicitud. Internamente estas entidades realizan la comunicación estándar de ROS de tópicos. La especificación de su comunicación se hace a través de una estructura particular de mensajes de ROS llamadas Acciones, donde se define la estructura de datos de la meta, los callbacks y el resultado.

2.2.1.4. Sistemas de coordenadas: *tf*

La librería de transformaciones entre sistemas de coordenadas es llamada *tf*. Esta librería almacena un árbol de transformaciones geométricas entre los sistemas de coordenadas de cada frame del robot. Este árbol permite transformar entre los sistemas de coordenadas de cualquier frame y solicitar transformaciones en el pasado o futuro (extrapolación) de configurarse.

Para el PR2 el árbol (ver Figura 2.4) parte con el frame */odom_combined* que representa el mundo y luego *base_footprint* marca el centro del robot entre las ruedas (a la altura del piso).

[5] `http://wiki.ros.org/rosbag`
[6] `http://wiki.ros.org/rqt_bag`
[7] `http://gazebosim.org/`

Para obtener transformaciones, éstas deben buscarse en el árbol en el tiempo específico en que fue solicitada, o la más cercana de estar disponible, por lo que hay un costo asociado importante. Deben evitarse llamadas masivas a *tf*, como por ejemplo pedir transformaciones para todos los puntos de una nube por separado, en vez de transformar la nube completa con su transformación asociada.

Los sistemas de coordenadas usados por ROS son Z arriba, X adelante e Y a la izquierda[8], excepto para aquellos frames con el sufijo *optical_frame*, los que corresponden a cámaras, puesto que el estándar en estos es Z en la dirección de la cámara, X abajo e Y izquierda.

Representación de transformaciones Se usan un tipo de dato `tf::Vector3` para la traslación y `tf::Quaternion` para la rotación. Los Quaterniones son un sistema de vectores que extienden los números complejos y tienen la particularidad de que pueden representar rotaciones de forma compacta y eficiente. Esta representación difiere de la usada mayormente en PCL, donde se usa la librería Eigen con las estructuras `Eigen::Vector3f` y `Eigen::Quaternionf`.

2.2.2. Point Cloud Library (PCL)

PCL [59] es una librería que facilita el trabajo con nubes de puntos, pues contiene varios métodos para manipularlas, y algoritmos del estado del arte para tareas comunes como segmentación, filtrado, visualización, reconocimiento, descriptores, etc. La librería tiene licencia BSD de 3 cláusulas, es código abierto y funciona independiente de otros frameworks o hardware.

2.2.3. MoveIt!

MoveIt! [69] es una librería de manipulación móvil que hace uso de algoritmos del estado del arte en planificación de movimiento, manipulación, percepción 3D, cinemática, control y navegación. Su diseño modular y extensible permite integrar fácilmente nuevos algoritmos y expone una interfaz de fácil uso para el desarrollo de soluciones robóticas. Como es de código abierto se puede consultar, extender y desarrollar con facilidad. Esta librería funciona en conjunto con el framework ROS desde su versión Hydro con Ubuntu 12.04. MoveIt! realiza la planificación de movimiento, cálculo de colisiones y ejecución final del grasping.

Figura 2.4: Figura en siguiente página. Arbol *tf* para el PR2. Rojo: La raíz del árbol. Naranjo: Frames pertenecientes al brazo derecho (Simétrico para el izquierdo). Verde: Frames pertenecientes a la Kinect. Violeta: Camino desde la Kinect y brazo a la raíz. Gris: Frame del punto medio entre los dedos del gripper (Idealmente el punto central del agarre). Café: Frame del efector final desde el punto de vista de MoveIt!; los agarres se calculan en función de este punto.

[8] En `rviz` es azul, rojo y verde respectivamente. Esto sirve como referencia para las figuras mostradas en `rviz` más adelante


odom_combined
base_footprint
base_link
bl_caster_rotation_link
torso_lift_link
fl_caster_rotation_link
fr_caster_rotation_link
br_caster_rotation_link
base_laser_link
base_below_link
torso_lift_motor_screw_link
l_torso_lift_side_plate_link
r_shoulder_pan_link
fl_caster_l_wheel_link
fr_caster_l_wheel_link
br_caster_r_wheel_link
imu_link
l_shoulder_pan_link
fl_caster_r_wheel_link
fr_caster_r_wheel_link
br_caster_l_wheel_link
r_torso_lift_side_plate_link
l_shoulder_lift_link
r_elbow_flex_link
r_forearm_roll_link
head_pan_link
r_shoulder_lift_link
r_forearm_link
r_forearm_cam_frame
laser_tilt_mount_link
r_upper_arm_roll_link
r_upper_arm_link
r_wrist_flex_link
r_forearm_cam_optical_frame
laser_tilt_link
l_upper_arm_roll_link
r_wrist_roll_link
bl_caster_l_wheel_link
l_upper_arm_link
r_gripper_palm_link
bl_caster_r_wheel_link
l_elbow_flex_link
r_gripper_motor_accelerometer_link
r_gripper_led_frame
r_gripper_l_finger_link
head_tilt_link
l_forearm_roll_link
r_gripper_tool_frame
r_gripper_motor_slider_link
r_gripper_r_finger_link
head_plate_frame
l_forearm_cam_frame
l_forearm_link
r_gripper_l_finger_tip_link
r_gripper_motor_screw_link
r_gripper_r_finger_tip_link
l_forearm_cam_optical_frame
l_wrist_flex_link
projector_wg6802418_frame
r_gripper_l_finger_tip_frame
sensor_mount_link
l_wrist_roll_link
projector_wg6802418_child_frame
l_gripper_palm_link
head_mount_link
l_gripper_motor_accelerometer_link
l_gripper_led_frame
l_gripper_l_finger_link
head_mount_kinect_ir_link
l_gripper_tool_frame
l_gripper_motor_slider_link
l_gripper_r_finger_link
head_mount_prosilica_link
high_def_frame
double_stereo_link
l_gripper_motor_screw_link
l_gripper_r_finger_tip_link
head_mount_prosilica_optical_frame
narrow_stereo_link
l_gripper_l_finger_tip_link
l_gripper_l_finger_tip_frame
high_def_optical_frame
wide_stereo_link
wide_stereo_optical_frame
narrow_stereo_l_stereo_camera_frame
head_mount_kinect_ir_optical_frame
narrow_stereo_optical_frame
wide_stereo_l_stereo_camera_frame
head_mount_kinect_rgb_link
narrow_stereo_r_stereo_camera_frame
wide_stereo_l_stereo_camera_optical_frame
head_mount_kinect_rgb_optical_fram
narrow_stereo_r_stereo_camera_optical_frame
wide_stereo_r_stereo_camera_frame
narrow_stereo_l_stereo_camera_optical_frame
wide_stereo_r_stereo_camera_optical_frame

Como MoveIt! tiene poco tiempo de desarrollo muchas de sus capacidades no han sido bien probadas en todos los robots que soporta. La tabla 2.1 presenta el nivel de estabilidad de los diferentes componentes de la librería en sus paquetes comunes. Los paquetes específicos de MoveIt! para el PR2 están en estado Alfa.

Tabla 2.1: Estado de MoveIt!

Componente	Estado
Cinemática (Cadena-Serial)	Beta
Cinemática (Articulaciones planares)	Alfa
Cinemática (Robots paralelos)	Alfa
Chequeo de colisiones (FCL)	Beta
Chequeo de colisiones (Basado en esferas)	Alfa
Planificación de movimiento (Objetivo de las articulaciones)	Beta
Planificación de movimiento (Objetivo de la pose)	Beta
Planificación de movimiento (Restricciones cinemáticas)	Beta
Planificación de movimiento (trayectorias deseadas del efector)	Alfa
Planificación de movimiento (Re-planificación rápida)	Beta
Planificación de trayectoria (OMPL)	Beta
Filtrado de trayectorias (Iterativo)	Beta
Navegación	Alfa
Navegación 3D	No soportado
Auto-filtrado 3D (Nube de puntos)	Beta
Auto-filtrado 3D (Mapa de profundidad)	Beta
Mapa de ocupancia 3D (*Octomap*)	Beta
Escena de planificación	Beta
Plugin `rviz`	Beta
Evaluación comparativa	Alfa
Evaluación comparativa (GUI)	Alfa
Análisis del espacio de trabajo	Beta
Recoger y depositar	Alfa
Manipulación doble brazo	Alfa
Asistente MoveIt!	Beta
Robot PR2	Alfa

2.2.3.1. Arquitectura y conceptos

El principal nodo de la librería, `move_group`, recibe desde el servidor de parámetros los archivos de configuración, URDF (Universal Robot Definition File) y SRDF (Semantic Robot Definition File), que tienen información sobre la geometría del robot, además de materiales, mallas 3D para colisiones, uniones, articulaciones e información semántica. Se comunica con los sensores, actuadores y nodos de estado del robot por medio de tópicos y acciones, e interactúa con diferentes interfaces que han de proveerle acciones y condiciones del entorno, como objetos unidos al robot, espacios de colisión, entre otros. La arquitectura general y algunas de las llamadas más importantes aparecen en el la Figura 2.5. Un ejemplo del funcionamiento de MoveIt! y algunos de sus módulos está en la Figura 2.6.

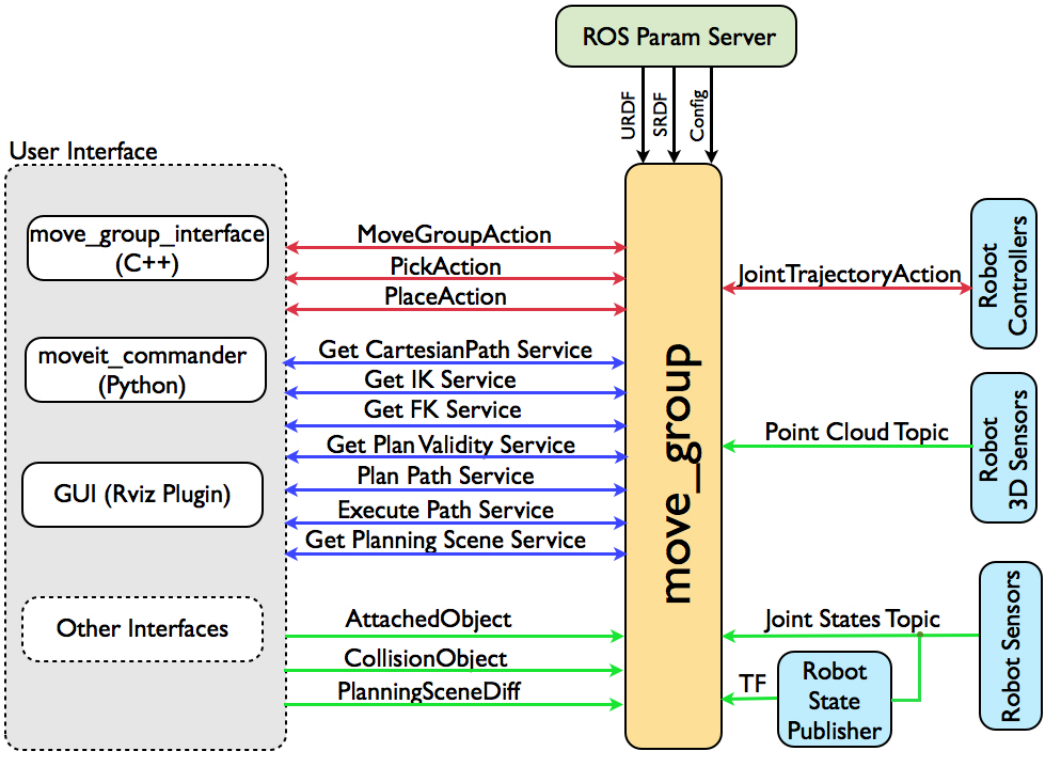


Figura 2.5: Diagrama[9]de alto nivel del nodo `move_group`. El usuario puede usar cualquiera de las interfaces del bloque *User interface* o crear la suya. `move_group` lee URDF, SRDF y archivos de configuración desde el servidor de parámetros, los que deben estar disponibles al momento de iniciar `move_group`. Rojo: Acciones de `actionlib`; la interfaz puede fijar objetivos a `move_group` y este puede fijar objetivos a los controladores del robot; en ambos casos se puede obtener feedback y resultado de la acción. Azul: servicios disponibles; las interfaces pueden hacer llamadas a servicios para obtener información o ejecutar trayectorias y recibir respuesta de sus llamadas. Verde: tópicos disponibles; las interfaces pueden modificar la escena mediante mensajes en tópicos, mientras que `move_group` lee los tópicos publicados por los sensores del robot (RGBD y motores) y por *Robot State Publisher* que publica las transformaciones mantenidas por *tf*. Los tópicos leídos por `move_group` son publicados por nodos del robot.

Escena de planificación Se usa para representar el mundo y el estado del robot. MoveIt! se suscribe al tópico */joint_states* para estar actualizado del estado de cada articulación en todo momento. También se suscribe a *tf* [10] con lo que tiene acceso a las transformaciones de coordenadas necesarias para mover el robot.

La información del entorno la recibe pre-procesada por el *Monitor de geometría del mundo*, que construye una representación 3D del entorno desde los sensores e información entregada por el usuario. La representación corresponde a un *Octomap*, una representación probabilista de ocupación de las celdas de una discretización del entorno.

[9] `http://moveit.ros.org/documentation/concepts/`

[10] *tf* tiene las matrices de transformación entre cada elemento del robot, así es posible trasformar una posición dada por la Kinect respecto del brazo, dada la pose de este.

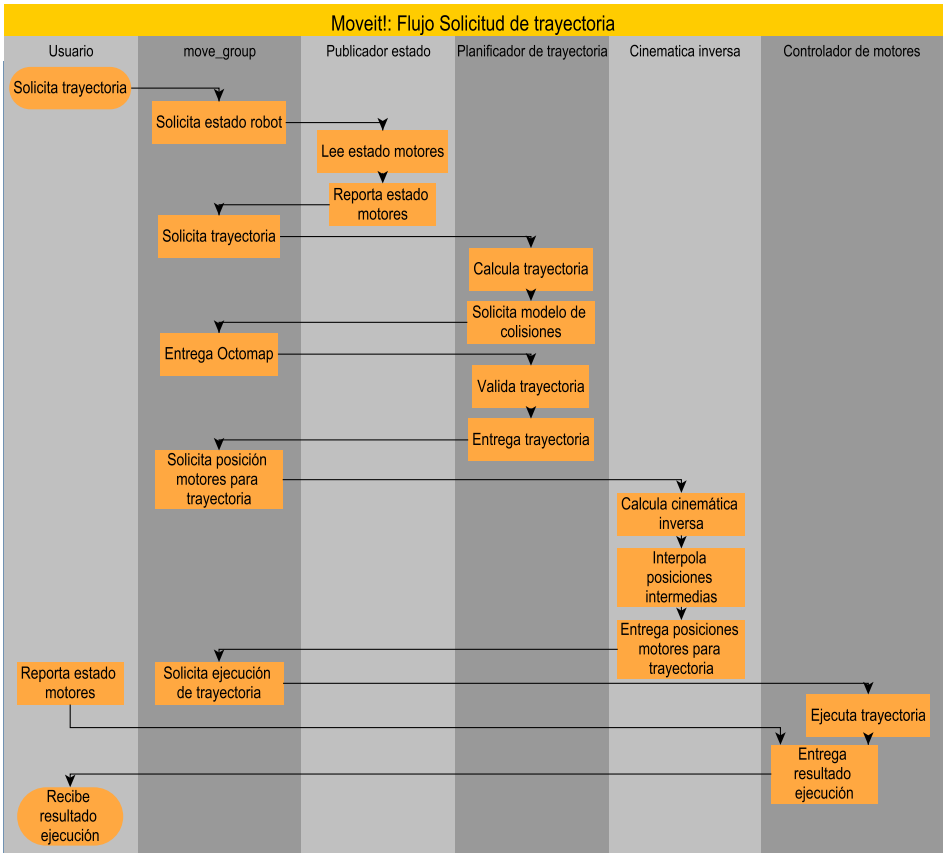


Figura 2.6: Ejemplo de solicitud de trayectoria en MoveIt!. El diagrama está simplificado para hacerse solo una idea general del proceso.

Planificador de movimiento Recibe una solicitud de movimiento que puede ser mover el brazo en espacio de articulaciones o una nueva pose del efector final; donde de ser necesario se puede informar que un objeto está unido al brazo para considerarlo en la planificación. Su respuesta es una trayectoria que lleva a cumplir el objetivo.

Verificación de colisiones Las colisiones son filtradas automáticamente por MoveIt!. Se soportan colisiones con mallas 3D, formas primitivas, y el *Octomap* de ocupación. Esta etapa es la más pesada del procesamiento, llegando a ocupar 90 % de los recursos.

2.2.4. PR2

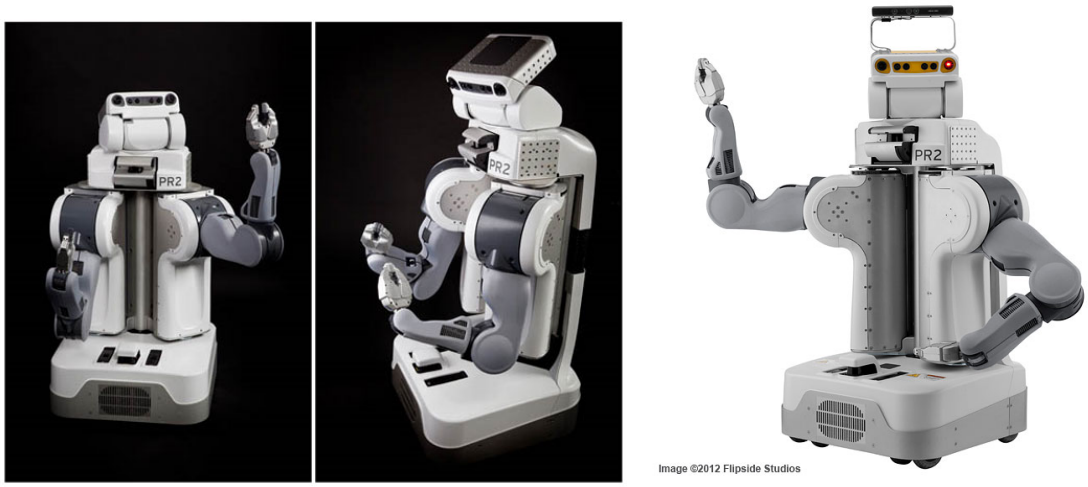
PR2 es un robot (ver Figura 2.7) con fines de investigación creado por la empresa Willow Garage que funciona con ROS. Cuenta con una base móvil para desplazarse, un torso que se desplaza hacia arriba para modificar su altura, 2 brazos de 7 grados de libertad cada uno, con pinzas como efectores. Sus principales sensores son: una cámara en cada brazo, un láser central para medir profundidad, uno en la base para navegación, una Kinect como sensor

RGBD, 2 cámaras estéreo de corto y largo alcance y un proyector de texturas[11].

Existe un gran número de aplicaciones creadas para el PR2, así como también existen muchos trabajos científicos en el área de grasping que utilizan este robot y que pueden servir como base de lo que se ha hecho y no en esta área. A pesar de este contexto, analizando los resultados de los trabajos revisados y viendo sus resultados en video, se ha mantenido como constante una lentitud en el procesamiento y ejecución del grasping [46] [11].

El Departamento de Ingeniería Civil Eléctrica (DIE) en conjunto con el Departamento de Ingeniería Civil en Computación (DCC) cuenta con un robot PR2 comprado con el segundo concurso de equipamiento científico y tecnológico mediano FONDEQUIP, proyecto EQM130098.

El robot es usado por ambos departamentos, por lo que se dispone de medio tiempo para utilizarlo.



(a) Robot PR2 Original (pre-2010) **(b)** Robot PR2 con Kinect (post-2010)

Figura 2.7: Robot PR2

2.2.4.1. BaseStation

Es un computador especialmente dedicado al robot, que permite acceder a el como administrador con facilidad, además de mantener una VPN, realizar funciones de respaldo de logs y permitir la re-instalación o actualización del robot. Este computador requiere pasar por un proceso conocido como *Branding*, en el que se modifican archivos en este y en el robot relativos a la configuración de red, nombres de dominio, rutas estáticas y configuración de la VPN.

[11]Proyecta una textura infrarroja (grilla o puntos con un diseño predefinido sobre los objetos. Esto mejora considerablemente el desempeño de las cámaras estero instaladas en el PR2 sobre objetos sin textura o reflectantes.

2.2.4.2. Arquitectura interna

El robot cuenta con 2 computadores funcionando como maestro y esclavo, llamados C1 y C2 respectivamente. Cada computador tiene dos procesadores Quad Core Intel Xeon 17 (8 cores), 24Gb de memoria RAM DDR3, 500GB de almacenamiento interno, un disco extraible en caliente de 1.5TB y vienen con una versión modificada de Ubuntu 10.04 (Lucid Lynx) y ROS Groovy (Anterior a Hydro) instalado. Estos computadores tienen una placa madre diseñada especialmente para el PR2, por lo que drivers y configuraciones son específicas al robot. C2 arranca por red desde C1 y C1 mantiene el proceso *roscore*, fundamental para la arquitectura distribuida de ROS, puesto que se encarga de comunicar a los nodos con otros nodos a partir de sus tópicos. C1 tiene buena parte de la carga de trabajo además de una Tarjeta de video Nvidia Quadro, mientras que C2 se encarga principalmente de los nodos del láser superior e inferior, así como la conectividad bluetooth del robot, necesaria para controlarlo con el joystick.

Cada uno de los elementos del robot, actuadores, sensores, baterías, sistemas de control, entre otros, se conectan por red a switches internos, generándose una compleja red interna que ROS debe administrar.

Como medidas de seguridad cuenta con 2 interruptores que cortan la corriente de los motores, uno en la parte trasera del robot y otro en forma de control remoto inalámbrico, que funciona de tal forma que si el robot pierde la conectividad con el control (por distancia, baterías o mal funcionamiento), los motores se apagan.

2.2.4.3. Arquitectura de red

La arquitectura de red del robot es bastante compleja como se aprecia en la figura 2.8. El router WIFI es un *"Linksys WRT610N Simultaneous Dual-N Band Wireless"*, modificado con el firmware *"DD-WRT v24-sp2 (09/30/09) big"*. El router WIFI se conecta en modo cliente a la red del edificio, permitiendo la salida a esta al robot.

El robot mantiene VPN con la BaseStation a través del WIFI del edificio, por lo que para contactarlo se debe pasar primero por el basestation que rutea a C1 o C2 según corresponda. En caso de necesitar hacer operaciones de mantenimiento se puede conectar un cable al puerto ethernet de servicio o conectarse por WIFI a una red propia para este fin.

2.3. Etapas del grasping robótico

A continuación se ahondará en las etapas del grasping robótico.

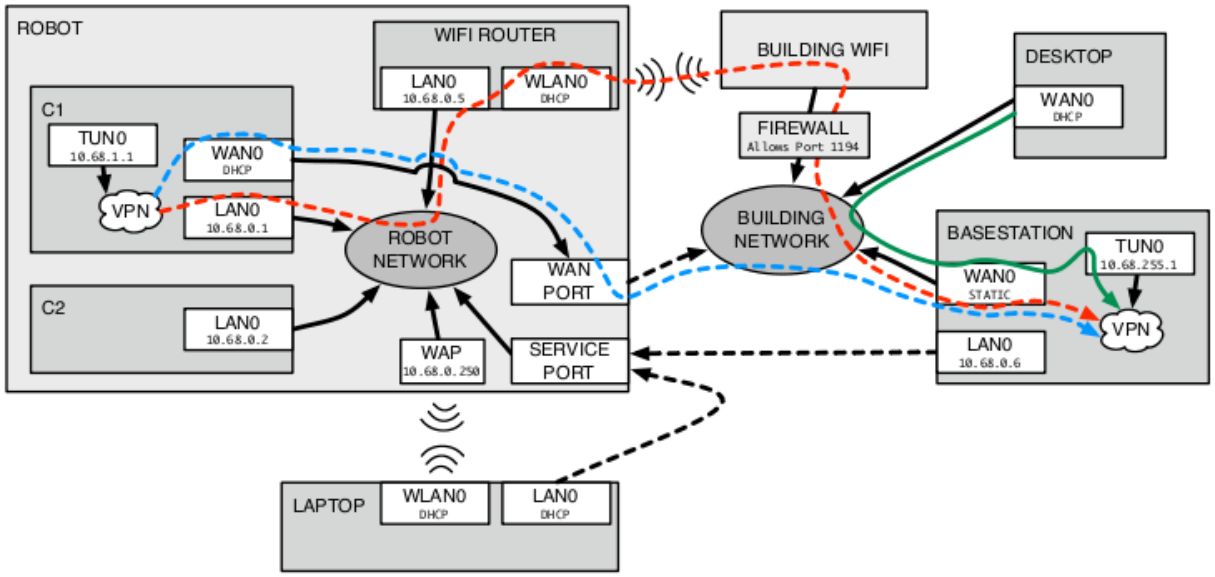


Figura 2.8: Diagrama de red del robot PR2. Las líneas punteadas pueden o no estar conectadas. Service port y WAP en el robot son las conexiones de servicio. El robot puede estar conectado a la red del edificio por cable en el puerto WAN, en cuyo caso la BaseStation rutea la VPN por este puerto. En caso de estar conectado al WIFI (y no al puerto WAN) del edificio por WLAN0, la BaseStation rutea la VPN por este medio. Un laptop en la red del edificio, para llegar al robot, se rutea por la línea verde hasta el BaseStation y luego por la roja o celeste dependiendo de la que esté conectada.

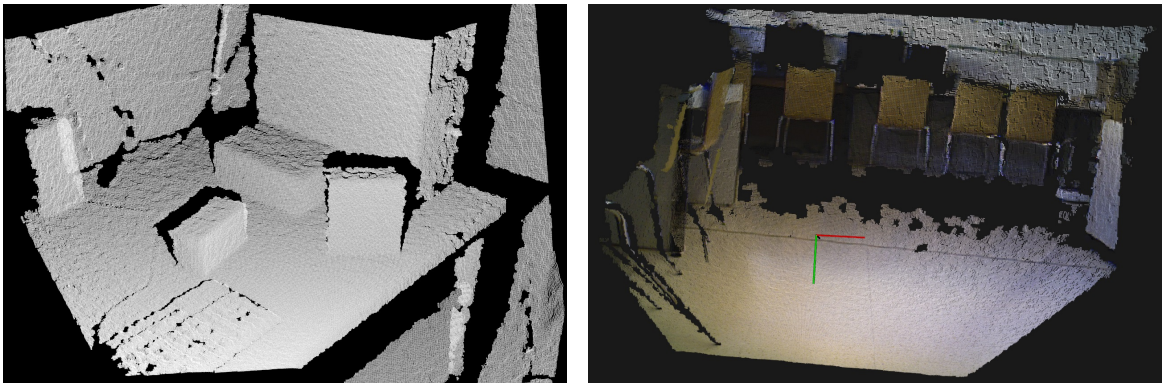
2.3.1. Segmentación de la escena

El procesamiento a realizar para etiquetar los componentes de la escena en objetos diferentes y otros elementos de interés depende casi completamente del sensor con el que se trabajará. A continuación se exponen algunas de las alternativas y luego se verán algunos de los algoritmos disponibles.

2.3.1.1. Sensores para segmentación de objetos

Para realizar esta tarea en la literatura se han usado cámaras web [64], cámaras estéreo [62], entre otros sensores, pero el que ha tenido mayor éxito es el Microsoft Kinect, usado desde su lanzamiento en 2010 por la mayoría de los investigadores.

El sensor Microsoft Kinect es un sensor tipo RGBD (en inglés: RGB + Depth), similar a las cámaras estéreo pero con un funcionamiento diferente, que entrega una imagen de la escena asignándole a cada pixel un valor de profundidad que indica su distancia al lente del sensor, por lo que teniendo la ubicación espacial de la Kinect, se tiene la de cada pixel. Esta información se conoce como nube de puntos, aunque una nube de puntos, para ser tal, solo requiere una posición en el espacio de un conjunto de puntos, sin considerar la información de color dada por la cámara de la Kinect. Tiene una resolución de 640x480 y funciona adecuadamente en interiores, sin mucha interferencia infrarroja, para elementos entre 80cm y 4m. Un ejemplo se ve en la Figura 2.9.



(a) Información de profundidad.

(b) Información de profundidad y color (RGBD).

Figura 2.9: Nubes de puntos entregadas por la Kinect. La nube es bastante densa, por lo que puede parecer una superficie, pero está formada por puntos solamente

2.3.1.2. Procesamiento de escena

El etiquetado de una escena puede hacerse con diferentes algoritmos dependiendo de la información que interese segmentar.

Técnicas de segmentación para diferentes casos:

Aplicación de un umbral (Binarización): Se convierte una imagen a escala de grises y se aplica un umbral para convertir el pixel a blanco o negro. Esto puede ayudar a separar elementos de una imagen con alto contraste. Se pueden manipular los histogramas para mejorar el contraste previo a la binarización.

Detección de bordes: Se pueden detectar los bordes de un objeto de interés para saber su posición y su forma. Existen múltiples algoritmos como operadores de Sobel [65], Prewitt [52], Roberts [58] o Canny [9].

Detección de líneas: Si la escena tiene cierta estructura (como un pasillo), se pueden detectar líneas y así solo considerar zonas de interés (como el suelo del pasillo). Un ejemplo es la transformada de Hough [15] para detectar lineas o curvas.

Detección de blobs: detectar zonas con alguna propiedad constante (blobs) puede ser un buen indicador de que el blob corresponde a un objeto (o un segmento de uno). Los algoritmos puede variar mucho dependiendo de la o las propiedades que se mantendrán constantes.

Métodos de clustering: Clustering es agrupar elementos similares generalmente por distancia en un dominio dado (distancia espacial, vector descriptor, etc.). El algoritmo más famoso es K-Means (ver Figura 2.10) y su versión difusa, Fuzzy C-Means, que requieren que se especifique el número de grupos que se quieren. En vez de usar el espacio de color se puede usar el de distancia en la nube de puntos y definir un número máximo de puntos en un cluster, así se evita especificar el número de clusters deseados;

este algoritmo se llama *Euclidean Clustering*.

Métodos de crecimiento de región: Se basan en que los pixeles vecinos en una región son similares en cierta propiedad (suavidad superficial por ejemplo) y se comienzan a agregar a una región común.

Métodos que usan clasificadores estadísticos: Se pueden usar descriptores para detectar objetos y usar clasificadores como redes neuronales u otros para determinar si un blob corresponde a un objeto de interés.

La segmentación de escena para detección de objetos en el campo del grasping puede ser difícil, pues simplemente analizando una muestra RGBD (Figura 2.9b) de la escena, como la proveniente de la Kinect, no es directo reconocer diferentes objetos del fondo (pared, suelo, mesa, muebles, etc.), ya que un conocimiento más profundo de las funcionalidades de cada elemento, de su interrelación y de las posibilidades de interacción con cada uno de ellos, correspondería a un conocimiento previo o a métodos más avanzados. Adicionalmente, como usualmente la muestra RGBD viene desde un solo punto de vista, es difícil decidir, para objetos que se obstruyen, si estos corresponden al mismo objeto o no.

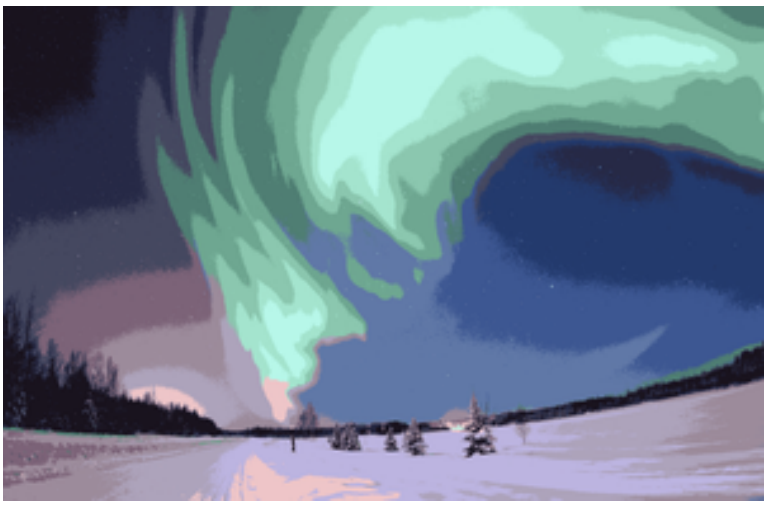


Figura 2.10: Imagen segmentada por colores luego de aplicar K-means con 16 grupos

2.3.1.3. Segmentación eficiente de planos

Una etapa importante al interactuar con el entorno es la segmentación de planos, la que puede ser útil para la navegación del robot, o para encontrar superficies de apoyo de objetos que se desea manipular. Para hacer esto de forma eficiente se pueden usar **imágenes integrales**, que corresponden a una representación de la imagen, donde cada pixel es reemplazado con la suma del rectángulo desde el primer pixel hasta ése. La ventaja de esta técnica es que para calcular la suma de los pixeles en cualquier rectángulo de la imagen, basta con sumar y restar por las 4 esquinas en vez de sumar todos los pixeles internos del rectángulo. Esto tiene un efecto directo en el orden de magnitud de los algoritmos, de cálculo de normales, dándoles tiempo constante una vez calculada la imagen integral.

2.3.2. Reconocimiento de objetos

El reconocimiento de objetos se torna aún más específico que la segmentación, puesto que los objetos tienen propiedades y configuraciones diferentes, por lo que en general hay que tener un algoritmo específico para cada objeto que se quiera reconocer.

Si se puede proveer un ejemplo visual del mismo objeto que se quiere reconocer, ya sea imagen o modelo 3D, basta con usar algoritmos que usen descriptores locales (puntos y regiones de interés).

En este caso se busca un método de manipulación relativamente general, por lo que no se realizará reconocimiento de un objeto o clase en particular, y solo detección de objetos (existencia y posición), aunque en el futuro puede ser de gran utilidad reconocer objetos para realizar tareas como traer determinado objeto o realizar una acción con dicho objeto.

2.3.3. Cálculo de puntos de agarre

Calcular puntos de agarre consiste específicamente en encontrar la pose del efector en 6D al entrar en contacto con el objeto, es decir su posición espacial y ángulo en ese momento.

Estos algoritmos se dividen en 2 familias principalmente: los que usan información de un modelo 3D completo del objeto a manipular y los que no.

2.3.3.1. Uso de un modelo 3D

Estos algoritmos asumen la existencia de un modelo 3D del objeto que se quiere manipular. A partir del modelo 3D, calculan una serie de propiedades del objeto, como sus normales más importantes, y en ocasiones también su centro de masa.

El desempeño de estos algoritmos es suficientemente bueno, consiguiendo resultados aceptables con bajas tasas de error. Sin embargo, en el caso de robots autónomos, no se cuenta con dicho modelo del objeto a manipular, lo que hace a estos algoritmos imprácticos en ambientes no controlados.

Un algoritmo que es un híbrido entre uso de modelos e información parcial es el presentado por Kehoe en [28], donde se entrena de forma offline puntos de agarre para diferentes objetos en una base de datos, de los que se cuenta con un modelo 3D completo. Luego, cuando el robot opera, se usa la información de la imagen y el servicio de reconocimiento de objetos de Google para reconocer el objeto en la base de datos. Además, se usa la información de profundidad de la cámara RGBD para estimar la pose del objeto comparándola con el modelo 3D de la base de datos. Una vez realizado este proceso, se utiliza el mejor punto de agarre computado para ese objeto, y los resultados de esta ejecución se reportan a la base de datos para mejorar las tasas de éxito de cada punto de agarre. La Figura 2.11 muestra el flujo del algoritmo.

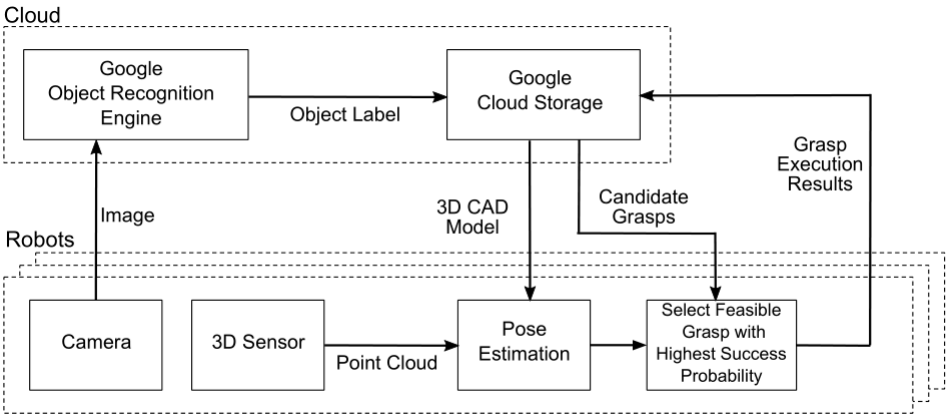


Figura 2.11: Diagrama de funcionamiento

2.3.3.2. Uso de información parcial

Estos algoritmos calculan los mejores lugares para manipular un objeto a partir de la información que puede sensar, típicamente con cámaras RGBD.

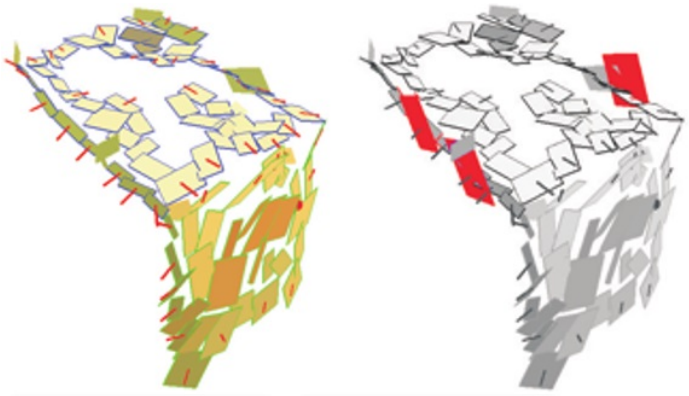
El algoritmo presentado por Kootstra en [31], hace uso de la información de los bordes y superficies de un objeto, medidos con una cámara estéreo. Se separan las formas de manipular un objeto en aquellas derivadas de características de superficie y aquellas derivadas de características de borde, generándose acciones de manipulación elementales (EGA). Las EGA corresponden a movimientos pre-definidos, como por ejemplo tomar al objeto por el borde, tomarlo desde arriba, insertar el efector en un orificio del objeto, etc.; estos movimientos son parametrizados por las características calculadas para bordes o superficies. Se exige que las formas de manipular generadas no colisionen con el objeto u otros objetos, con lo que se filtran varias y que cumplan ciertas propiedades respecto de las normales de contacto y ángulo de agarre. La Figura 2.12 muestra un punto de agarre calculado y la escena de la que proviene.

En [62], se calculan los mejores puntos para agarrar un objeto a partir de imágenes de éste. Se usa aprendizaje supervisado con imágenes sintéticas para enseñarle los mejores lugares. A continuación se utiliza un modelo probabilístico para triangular la posición 3D del punto de agarre a partir de un par de imágenes, y estimar la probabilidad de todos los puntos cercanos a los rayos de la triangulación de forma de escoger el que tiene mayor probabilidad de ser un punto de agarre. El resultado obtenido en un ambiente de cocina se ve en la Figura 2.13.

Deep Learning es una familia de técnicas que intenta modelar abstracciones de alto nivel en datos usando múltiples capas de procesamiento. Generalmente se usan redes neuronales y aprendizaje no supervisado pero depende del problema que se quiere resolver.

En [37] se entrega una imagen RGBD a un sistema de *Deep Learning* que usa 2 redes neuronales en etapas secuenciales: en la primera se aprenden (extraen) características de forma no supervisada, y en la segunda se usan las características aprendidas en una red neuronal en cascada (varias redes secuenciales con la misma arquitectura aunque diferente tamaño donde la salida de una se pasa a la siguiente) entrenada en forma supervisada. Se calculan los

(a) Escena original en estéreo con ruido de texturas en el fondo. El algoritmo es robusto a la presencia de texturas en la escena.



(b) Izquierda: Representación de texturas (rectángulos de colores) y bordes (normales en rojo) de las características calculadas sobre el objeto. Derecha: Puntos de agarre se muestran en rojo para un efector de 3 pinzas.

Figura 2.12: Imagen original y puntos de agarre calculados por el algoritmo.

Figura 2.13: Predicción de puntos de agarre para varios objetos en la imagen.

rectángulos de agarre para un manipulador tipo pinza, donde un rectángulo de agarre incluye información de la orientación y tamaño de la pinza además del lugar. La primera etapa de la cascada tiene menos características para permitir eliminar rápidamente configuraciones con poca probabilidad de ser puntos de agarre y la segunda tiene características más intensivas que permiten distinguir pequeñas diferencias entre las configuraciones ya filtradas. La Figura 2.14 muestra la etapa supervisada del sistema y la Figura 2.15 el resultado para varios objetos.

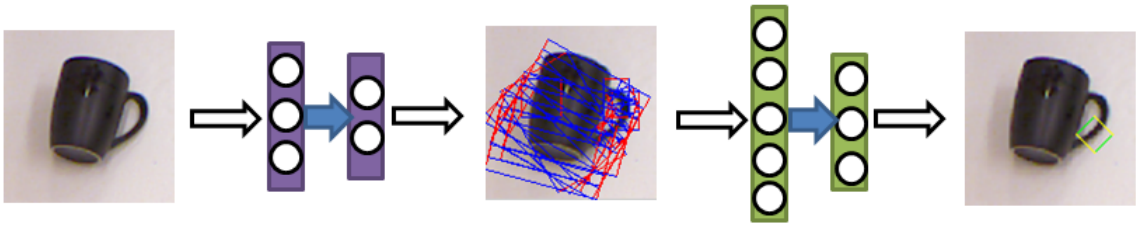


Figura 2.14: Etapa supervisada del sistema de *Deep Learning*: Cascada de redes neuronales. La primera red recibe las características, calculadas en la primera etapa, de la imagen RGBD y entrega un número de posibles rectángulos de agarre. La segunda red filtra los rectángulos y determina el de mayor probabilidad de dar un agarre exitoso.

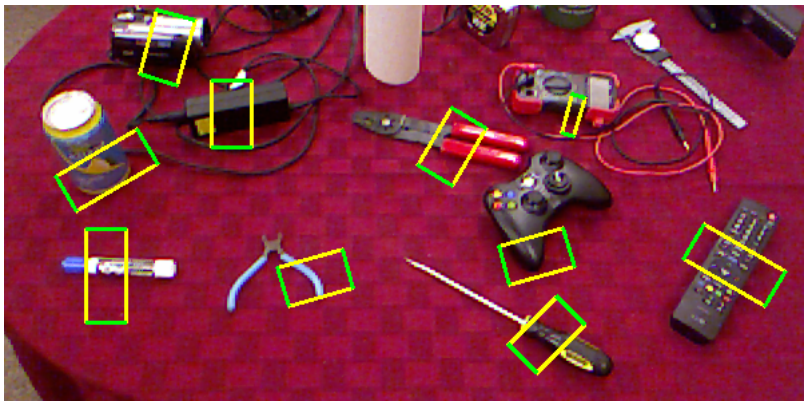


Figura 2.15: Puntos de agarre para un efector tipo pinza [37]

2.3.4. Planificación de trayectoria

Se usan las diferentes familias dependiendo del problema particular que se quiere resolver. La librería de movimiento usada, MoveIt! (ver sección 2.2.3), es flexible y modular, por lo que permite trabajar con cualquier algoritmo de planificación de trayectorias con la apropiada interfaz. Dos librerías que cumplen con esta interfaz y son ampliamente usadas son **SBPL** [39] y **OMPL** [71]; ambas son independientes de ROS y agnósticas al hardware que modelan. La primera usa algoritmos basados en primitivas y la segunda basados en muestras, es por tanto fácil usar diferentes algoritmos dependiendo del problema particular que se quiere resolver. Se presenta a continuación una descripción detallada de cada familia.

2.3.4.1. Algoritmos basados en primitivas

Consisten en movimientos básicos que se encadenan para formar movimientos más complejos. Se realiza una búsqueda en cada estado entre un subconjunto de movimientos posibles para el robot considerando sus restricciones y los obstáculos, lo que permite construir el grafo incrementalmente o pre-calcularlo dependiendo de las necesidades del problema. La librería SBPL implementa varios de estos algoritmos, siendo los más relevantes los que se presentan en las siguientes secciones.

- Ventajas
  - El grafo generado es pequeño
  - Todos los caminos encontrados son factibles
  - Incorpora naturalmente las restricciones del robot u otras adicionales que sean de interés
- Desventajas
  - Dependiendo de la discretización elegida puede que sea imposible muestrear ciertos caminos válidos. Esto hace concluir falsamente infactibilidad para un problema factible.

2.3.4.1.1 Anytime Repairing A* (ARA*) [40]

A* expande sus estados basándose en la minimización g+h, donde g es el costo conocido de llegar a un nodo y h es una heurística de cuánto cuesta llegar a la meta desde dicho nodo. A* ponderado minimiza $g + \varepsilon h$ con $\varepsilon > 1$, garantizando que:

$$costo(solución) \leq \varepsilon \cdot costo(solución\ óptima)$$

La idea de ARA* es correr A* varias veces disminuyendo ε hasta que se acabe el tiempo, y re-usar los valores de los estados que no cambian. La Figura 2.16 muestra una ejecución de ARA* comparándolo con A* ponderado, donde se ve que A* ponderado debe repetir la búsqueda en cada caso pero ARA* solo expande los estados no visitados aumentando considerablemente la eficiencia.

2.3.4.1.2 Anytime D* [42]

Es similar a ARA* pero funciona sobre entornos dinámicos. Permite re-planificar eficientemente frente a cambios en el camino. Cuando vuelve a planificar puede incrementar o disminuir ε.

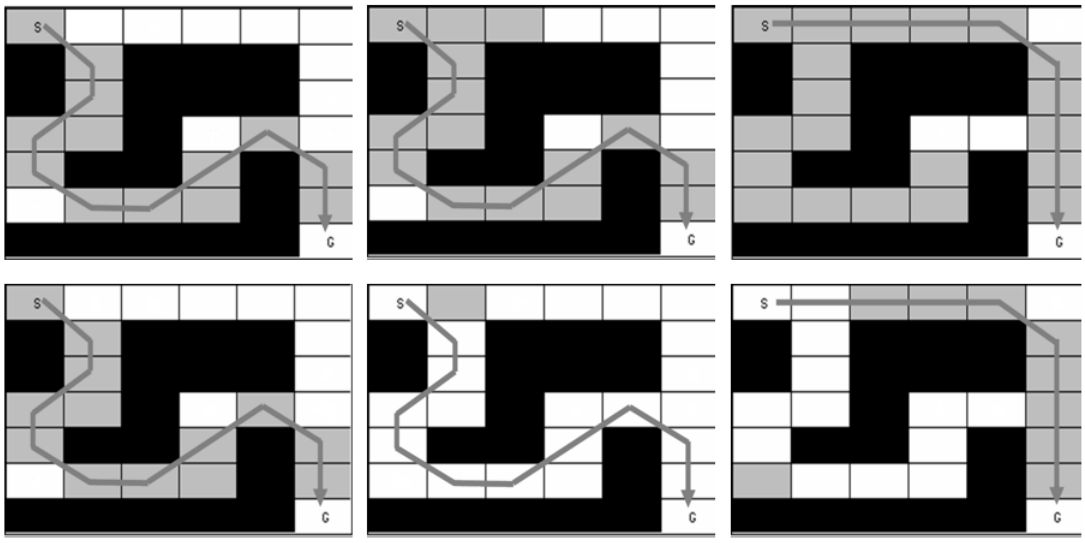


Figura 2.16: Ejecución del algoritmo para encontrar una trayectoria de S a C. De izquierda a derecha: valores decrecientes de ε. Los cuadros negros son obstáculos, los blancos es camino inexplorado y los grises representan el camino explorado. Arriba: serie de A* ponderado. Abajo: ARA*

2.3.4.1.3 R* [41]

Es la versión aleatorizada de A*. Muestrea caminos intermedios para llegar a la meta. Si un camino está tomando demasiado tiempo, lo evita y sigue con los más sencillos. Sirve para trabajar en entornos complejos y grandes.

2.3.4.2. Algoritmos basados en muestras

Algoritmos que eligen puntos aleatorios del espacio y los van agregando a la trayectoria actual si están libres de obstáculos. Los algoritmos usados en la librería OMPL se basan principalmente en Probabilistic roadmaps [27] y Rapid Exploring Random Trees [34], sin embargo hay muchos más que no se relacionan directamente con ellos. A continuación se presentan brevemente los más importantes.

2.3.4.2.1 Probabilistic RoadMaps (PRM) [27]

Se construye un camino con puntos de control válidos (espacio sin obstáculos) que aproxima la conectividad del espacio de estados. Los puntos de control cercanos se conectan con una discretización de movimientos válidos intermedios. Finalmente se usa un algoritmo como A* para buscar un camino en la red creada.

Variantes:

- **PRM* [26]:** En vez de elegir a mano un número de vecinos más cercanos a unir en la red, se eligen el número automáticamente basándose en parámetros de ocupación del

espacio de estados, esto quiere decir que el número de vecinos es una función de la razón de espacio libre (sin obstáculos) al espacio total. Esto garantiza soluciones óptimas.

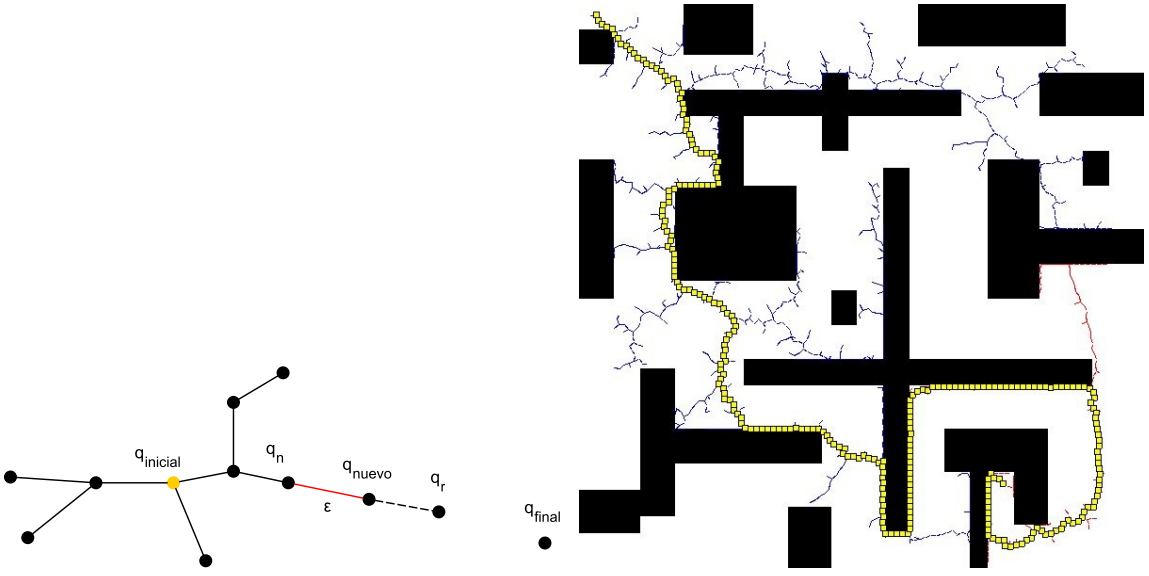
- **LazyPRM/LazyPRM* [6]** : Se comprueban perezosamente las colisiones con el fin de obtener soluciones probables más rápidamente. Esto quiere decir que se encuentra un camino, inicialmente línea recta, y luego este se descarta si presenta colisiones, y se tratan de encontrar un camino que una los segmentos separados por las colisiones.

2.3.4.2.2 SPARS [14] y SPARS2 [13]

Construye iterativamente la red de PRM, pero decrece en el tiempo la probabilidad de agregar nodos a la red, lo que garantiza soluciones cercanas a la óptima. La segunda versión obtiene mejores soluciones pero es más lenta que la primera.

2.3.4.2.3 Rapidly-exploring Random Trees (RRT) [34]

La idea es construir incrementalmente un árbol que determine los caminos posibles. Para agregar un nodo nuevo, se muestrea un estado al azar q_r y se busca el estado q_n en el árbol que está más cercano al estado muestreado. Luego se expande el árbol un delta ε, creando un nuevo estado q_{nuevo} que está más cerca de q_r, hasta que se alcance un estado q_{final}. Finalmente q_{final} se agrega al árbol de exploración. La figura 2.17a ejemplifica este proceso. Como es un algoritmo aleatorizado se pueden obtener diferentes soluciones como se ve en la Figura 2.17b.



(a) Diagrama ejecución RRT.

(b) RRT encontrando camino entre 2 puntos. Caminos encontrados para 2 ejecuciones diferentes de RRT son diferentes debido a la aleatoriedad del algoritmo

Figura 2.17: Diagrama y ejemplo RRT

Variantes:

- **RRT Connect [33] y Parallel RRT (pRRT):** Usa 2 árboles en paralelo, uno en el inicio y otro en el final e intenta unirlos. La versión paralela es una implementación multiproceso.

- **Lazy RRT [5]:** Al acercarse al nuevo estado q_m, no se chequean las colisiones. Cuando se encuentra un camino se verifica su validez y remueven segmentos inválidos, volviendo a la búsqueda de antes.

- **RRT* [26]:** Garantiza converger a una solución óptima y el tiempo de cálculo (TC) está acotado por $\alpha \cdot TC(RRT)$

- **Lower Bound Tree RRT (LBTRRT) [60]:** Mantiene un grafo y un árbol sobre el espacio de estados. La solución cumple $solución(LBTRRT) \leq \alpha \cdot óptimo(RRT)$. Provee mejores soluciones que RRT y más rápidas que RRT*

- **Transition-based RRT (T-RRT) [25]:** Considera un mapa de costos que permite considerar obstáculos difusos

2.3.4.2.4 Expansive Space Trees (EST) [23]

Se comienza con 2 árboles: uno en el estado inicial y el otro en el estado final y se define una grilla sobre una proyección del espacio de estados (Disminución de dimensionalidad). Con esto intenta detectar las áreas menos exploradas siguiendo 2 etapas:

Expansión

- $w(x)$ = Número de nodos en el árbol que yacen en la vecindad del nodo x
- Elegir un nodo con probabilidad $\frac{1}{w(x)}$
- Muestrear la vecindad del nodo y agregar un nodo si el camino está conectado a la raíz.

Conexión

- Para cada nodo en el árbol inicial, buscar un nodo en el árbol del objetivo.
- Si la distancia entre los nodos es menor que cierta constante, unirlos y terminar.

2.3.4.2.5 Single-query Bi-directional Lazy collision checking planner (SBL) y su versión paralela (pSBL) [61]

Se tienen 2 árboles para la exploración, similar a un EST con una distancia decreciente de vecindad. La exploración se guía con una grilla de estados previamente visitados, donde las casillas menos llenas tienen más probabilidades de ser seleccionadas para exploración. Conexión de estados de diferentes árboles es intentada si caen en la misma celda de la grilla.

2.3.4.2.6 Kinematic Planning by Interior-Exterior Cell Exploration (KPIECE) [70]

Algoritmo basado en árboles. Realiza una discretización de varios niveles de diferente resolución para guiar la exploración del espacio continuo, instanciando dinámicamente la grilla en una proyección del espacio. Se intentan explorar las celdas de la frontera, es decir, con menos de 2 vecinos (se usa vecindad 4: los vecinos son los de los lados y arriba y abajo), y para esto se van eligiendo las celdas de niveles superiores, y se va bajando hasta llegar a las de nivel más bajo. Funciona mejor que RRT, EST y Path-Directed Subdivision Trees (PDST) en varios casos.

Las variaciones más importantes son Bi-directional KPIECE (BKPIECE) que usa 2 árboles de exploración en los estados iniciales y finales, y Lazy Bi-directional KPIECE (LBKPIECE) que agrega un chequeo de colisiones perezoso. Este último es el usado por defecto en MoveIt!.

2.3.4.2.7 Search Tree with Resolution Independent Density Estimation (STRIDE) [17]

Usa Geometric Near-neighbor Access Tree una estructura de datos recursiva para optimizar la búsqueda y detectar las áreas menos exploradas. Su funcionamiento es similar a EST pero no requiere una proyección del espacio ni una grilla dado que la estructura se adapta al espacio de estados.

Está diseñado para trabajar en entornos de alta dimensionalidad, donde supera ampliamente a los otros métodos.

2.3.4.3. Aplicación a brazos robóticos

Se tomaron los datos de un experimento disponibles en la plataforma de evaluación de algoritmos de planificación de trayectorias, PlanerArena[12], donde se dispone de una interfaz web para evaluar algoritmos usando datos disponibles o cargar los propios además de que las pruebas realizadas son independientes de hardware y framework, por lo que no se usa ROS. El experimento consiste en una cadena kinematica de 20 grados de libertad (modelo complejo de un brazo robótico) que está en un tubo curvado y debe salir doblándose sobre si misma y evitando colisiones para extenderse fuera en el espacio abierto. La plataforma realizó 20 ejecuciones para cada algoritmo de la librería OMPL, para promediar resultados. Los detalles del equipo usado y los parámetros de los algoritmos graficados se pueden ver en el Anexo A.1. De estas pruebas se determinó que los más apropiados para planificación de trayectorias son STRIDE, KPIECE y EST, puesto que obtuvieron los mejores tiempos de planificación para un brazo simulado de 20 grados de libertad como se muestra en la Figura 2.18, donde se muestran los mejores algoritmos.

[12] `http://plannerarena.org/`

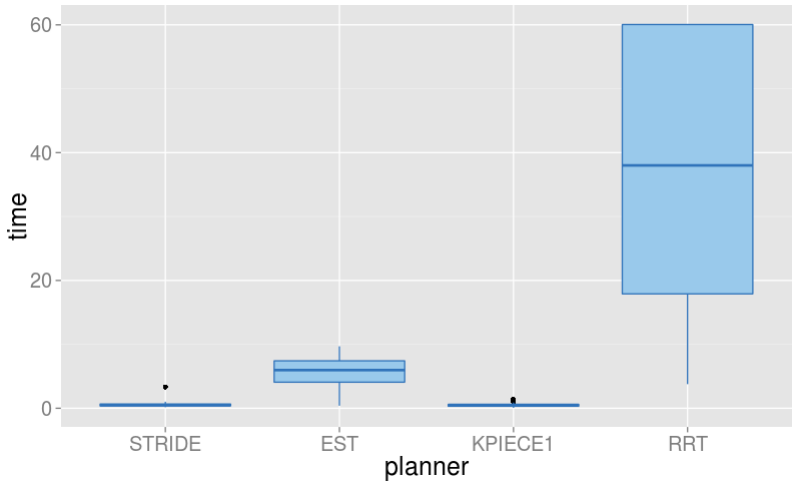


Figura 2.18: Mejores tiempos de cálculo de trayectorias en la librería OMPL para un brazo de 20 grados de libertad

2.3.5. Cinemática inversa

Existen muchas técnicas para resolver la cinemática inversa. A continuación se revisarán las más usadas en la librería MoveIt! y ROS.

2.3.5.1. Método algebraico (analítico)

Consiste en resolver las ecuaciones del problema (una por cada grado de libertad). A medida que aumentan los grados de libertad, se vuelve sustancialmente más complejo resolver las ecuaciones debido a la no linealidad de éstas. Está garantizado que hasta 6 grados de libertad el sistema es resoluble. Sin embargo, el método no asegura encontrar la solución. En ROS se usa el módulo IKFast[13], miembro del framework robótico OpenRave[14], por defecto para generar el algoritmo de solución apropiado al problema y luego resolver apropiadamente de forma algebraica. Este método obtiene soluciones del orden de $4[\mu s]$, sin embargo para algunos mecanismos de 6 grados de libertad, el algoritmo no encuentra una solución.

2.3.5.2. Método iterativo: Inversión Jacobiana

Consiste en resolver la ecuación que relaciona el movimiento cartesiano del efector con los movimientos de los ángulos de cada articulación, mediante la inversión de la matriz jacobiana:

$$\dot{X} = J(\theta)\dot{\theta} \rightarrow \dot{\theta} = J^{-1}(\theta)\dot{X}$$

[13] `http://openrave.org/docs/0.8.0/openravepy/ikfast/`
[14] `http://openrave.org/`

Donde $\dot{X}$ representa la velocidad lineal y rotacional del efector; y $\dot{\theta}$ representa la velocidad rotacional de cada articulación. La matriz jacobiana es obtenida columna a columna de las matrices de transformación para cada articulación. El proceso de inversión calcula las soluciones para posiciones del efector cada vez más cercanas a la posición final deseada, actualizando las relaciones de transformación. Cuando se llega a una distancia suficientemente cercana o se supera el máximo de iteraciones se detiene la búsqueda.

Este método tiene como principal problema el tiempo que toma invertir la matriz, lo que hace que rápidamente sea infactible en tiempo real al aumentar los grados de libertad. Además si la dimensión en el plano cartesiano es diferente de la dimensión de ángulos en las articulaciones, la matriz no es cuadrada, por lo que es necesario calcular una pseudo-inversa, lo que provoca errores al tratar con variaciones muy grandes en el eje cartesiano.

Lo positivo es que los movimientos generados son intuitivos y para pocas articulaciones es suficientemente rápido, por lo que este es el método usado en el PR2, a diferencia del que usa ROS por defecto.

2.4. Revisión algoritmos de detección de puntos de agarre

Se realizó una ardua revisión del estado del arte para encontrar el algoritmo más apropiado que cumpliera con las características requeridas. Para esto se dividieron los algoritmos en los tipos de objetos que puede manipular respecto de los que se usaron para entrenamiento, entre:

- Objetos conocidos: Son aquellos mismos con los que se entrenó el algoritmo, de haber entrenamiento o un set definido sobre el que se puede realizar la manipulación.
- Objetos familiares: El algoritmo funciona sobre objetos de la misma familia que aquellos usados para entrenamiento.
- Objetos desconocidos: Los objetos pueden ser de cualquier tipo y no tener relación alguna con los usados antes.

Además se consideró el tiempo que tomaban los algoritmos para entregar un resultado y el porcentaje de éxito alcanzado por éstos, con el fin de determinar su utilidad en aplicaciones en tiempo real.

Aunque los autores de los diferentes algoritmos usaron equipos diferentes para realizar sus pruebas, los equipos usados son del segmento consumidor (no se usaron supercomputadores), lo que significa que con equipos del año 2015 se pueden conseguir resultados similares o levemente mejores. Esto significa que no se puede hacer una comparación numérica de tiempos de cálculo y porcentaje de éxito (para tareas que dependen del tiempo disponible para el cálculo) entre los algoritmos; por tanto los datos reportados son interesantes cuando hay diferencias de órdenes de magnitud o para evaluar si un método podría ejecutarse en tiempo real o pseudo-real (cercano a 1 segundo).

Algunos algoritmos usan MATLAB[15] como plataforma de implementación y otros usan OpenRave, ROS u otros. Entre los que usan ROS, en su mayoría son versiones obsoletas que no incluyen a MoveIt! como plataforma de planificación de movimiento, por lo que la utilización de cualquiera de ellos en aplicaciones comerciales requeriría una re-implementación en Moveit!.

Los tiempos reportados corresponden al tiempo del algoritmo de cálculo de puntos de agarre, incluyendo segmentación, para objetos sobre una mesa. En la Tabla 2.2 se detallan los resultados reportados en el paper de cada algoritmo estudiado. La nomenclatura de la tabla es:

TO: Tipo de objetos. Los tipos son Conocidos (C), Familiares (F) y Desconocidos (D).

T: Tiempo

TR: Tiempo real

PE: Porcentaje de Éxito

TEC: Técnica

[15] `http://www.mathworks.com/products/matlab/`

Tabla 2.2: Algoritmos de detección de puntos de agarre

Nombre paper	TO	T [s]	TR	PE	TEC
Real-Time 3D Perception and Efficient Grasp Planning for Everyday Manipulation Tasks[a][68]	F	0,1588	Si	98,75	Primitivas de agarre y chequeo eficiente de colisiones
Combining active learning and reactive control for robot grasping[b][32]	C,D[c]	17[d]	No	>75[e]	Modelos de recompensas esperadas. Cota superior de confianza (UCB) y aprendizaje reforzado (RL) usando regresión de procesos gaussianos.
Deep Learning for Detecting Robotic Grasps[37]	F	13,5[f]	No	89	*Deep Learning*
A strategy for grasping unknown objects based on coplanarity and colour information[51]	D	-	NA	Var.	Información de color y planar (ECV)
Enabling grasping of unknown objects through a synergistic use of edge and surface information[31]	D	-	NA	Var.	Información de color y planar (ECV)
Learning to Grasp Novel Objects using Vision[63]	F,D	>1,2	Si	90(87,5)	Aprendizaje de imágenes supervisado
Robotic Grasping of Novel Objects[64]	F,D	>1,2	Si	90(87,5)	Aprendizaje de imágenes supervisado
Robotic Grasping of Novel Objects using Vision[62]	F,D	>1,2	Si	90(87,5)	Aprendizaje de imágenes supervisado
Cloth Grasp Point Detection based on Multiple-View Geometric Cues with Application to Robotic Towel Folding[g][44]	F	13,8	No	81	Detección de ángulos utilizando profundidad

Continúa en la siguiente página

[a] Objetos domésticos
[b] Todos los niveles de dimensionalidad
[c] Añadiendo tiempo de inicialización
[d] 0,65 en paralelo; 24 minutos de inicialización
[e] Recompensa del aprendizaje reforzado
[f] MATLAB
[g] Toallas

Tabla 2.2 – continuada desde la página anterior

Nombre paper	TO	T [s]	TR	PE	TEC
Real-Time Grasp Detection Using Convolutional Neural Networks[54]	F	0,0769	Si[a]	88	Regresión usando red neuronal convolucional
A 3D-grasp synthesis algorithm to grasp unknown objects based on graspable boundary and convex segments[1]	D	10	No	87,78	Extracción heurística de agarres basado en contornos, ángulos y propiedades geométricas
Image-Based Grasping Point Detection Using Boosted Histograms of Oriented Gradients[36]	D	-	-	66,56	Descriptor semi-local de histogramas de orientación de bordes. Utiliza *boosting*
Boosted Edge Orientation Histograms for Grasping Point Detection[35]	D	-	-	77,94	Descriptor semi-local de histogramas de orientación de bordes. Utiliza *boosting*
Grasp Planning via Decomposition Trees[19]	D	n^4	No	~20[b]	Reduce espacio de búsqueda planificando en un árbol de descomposición supercuadrática y simula usando modelo 3D
Efficient 3D object perception and grasp planning for mobile manipulation in domestic environments[c][67]	D	~0,040	Si	94	Muestrea puntos de agarre y clasifica dos posiciones distintas de agarre.
Grasp Planning Under Shape Uncertainty Using Gaussian Process Implicit Surfaces and Sequential Convex Programming[d][43]	D	96,43	No	86,87[e]	Resuelve el problema de optimización en espacios de posibles agarres
Grasping of Unknown Objects from a Table Top[56]	D	-	NA	85,71	Encuentra la superficie superior y el centro de masa de los objetos para encontrar puntos de agarre

Continúa en la siguiente página

[a] GPU
[b] Mejor de lo planeado en el modelo
[c] Objetos simples
[d] Incertidumbre en la forma de objetos
[e] Probabilidad de cierre forzado

Tabla 2.2 – continuada desde la página anterior

Nombre paper	TO	T [s]	TR	PE	TEC
Robotic Grasping of Unknown Objects[57]	D	4,34	No	85	Superficies planas superiores y centro de masa
Shape-Primitive Based Object Recognition and Grasping[a][49]	F	14,9	No	76	Mapa multi-resolución para descarte de agarres infactibles debido a colisiones
Implementing Robotic Grasping Tasks Using a Biological Approach[b][38]	F	-	Si	95	Redes neuronales en percepción y control. Aprendizaje reforzado en movimiento
HERB:a home exploring robotic butler[66]	C	0,25	Si	91	*Scale-invariant feature transform* (SIFT) y estructura para movimiento
Learning Robot Grasping from 3-D Images with Markov Random Fields[8]	F	56,3[c]	No	73,3	Entrena campos aleatorios de Markov
Robotic grasping of unmodeled objects using time-of-flight range data and finger torque information[45]	D	0,0042	Si	81,35	Distribución de puntos gaussianos
Learning grasping affordances from local visual descriptors[47]	F	-	NA	66,6	Descriptores y aprendizaje de la experimentación
Learning a Real Time Grasping Strategy[24]	F	0,010	Si	85	Modelo de mezcla de gaussianas entrenado en datos sintéticos
Learning of grasp selection based on shape-templates[21]	F	5-30	No	87	Comparación de profundidades con datos aprendidos
High-level Reasoning and Low-level Learning for Grasping: A Probabilistic Logic Pipeline[2]	F	0,0275	Si	77,8	*SVM* en nuevos descriptores
Learning Grasp Affordance Densities[12]	D	-	NA	61	Exploración autónoma para agarrar y soltar objetos

Continúa en la siguiente página

[a]Objetos compuestos de figuras geométricas simples
[b]Sólo probado con una naranja y una lata
[c]Variabilidad de 27,9

Tabla 2.2 – continuada desde la página anterior

Nombre paper	TO	T [s]	TR	PE	TEC
Learning grasping points with shape context[4]	F	-	-	86,31	Aprendizaje supervisado *SVM* usando descriptores de forma
Learning object, grasping and manipulation activities using hierarchical HMMs[50]	-[a]	-	Si	-	Modelo de Markov oculto en multiniveles jerárquicos. Descomposición de agarres en grupos de primitivas de acción
Learning RGB-D descriptors of garment parts for informed robot grasping[b][53]	D	1	Si	Var.	Bolsas de palabras visuales y data 3D
Learning to grasp from point clouds[48]	F	-	NA	49	Descriptores y SVM. Aprendizaje por experimentación.
Learning to Manipulate Unknown Objects in Clutter by Reinforcement[c][7]	D	0,89	Si	~92.86[d]	Aprendizaje reforzado para acciones de pre-agarre. Clustering espectral para segmentación de objetos.
Object detection methods for robot grasping: Experimental assessment and tuning[55]	D	-	Si	82,85	Ajuste de plano a superficie superior y agarre por el centro
Probabilistic Models of Object Geometry for Grasp Planning[e][18]	C	-	NA	94,28	Utiliza criterio *NGUYEN* para encontrar agarres estables sobre contornos de objetos

[a] Sólo predicción de primitivas de acción y actividades
[b] Ropa
[c] Ambientes agrupados
[d] Recompensa
[e] Objetos deformables

2.4.1. Aporte del presente trabajo

Esta memoria permitirá a RyCh contar con un robot de categoría mundial que pueda usar un algoritmo de cálculo de puntos de agarre para tomar objetos, sin ceñirse a una categoria especifica de estos y en un tiempo razonable, al señalarle el objeto de interés. Esto abrirá las puertas para nuevas investigaciones en el área de manipulación robótica y la futura participación en competencias. Adicionalmente se hace disponible de forma libre un buen algoritmo de manipulación robótica para que toda la comunidad pueda usarlo.

Capítulo 3

Implementación

La implementación del sistema hecha en esta memoria, se basa en la descripción dada por Stueckler et al. en [68], donde se definen las etapas utilizadas y optimizaciones hechas como se expone en la sección 3.1; se intentó replicar las etapas y algoritmos allí usados. Luego la sección 3.2, habla del contexto específico donde se espera que funcione el algoritmo. La sección 3.3, explica las etapas previas necesarias para implementar el algoritmo. Finalmente la sección 3.4 explica la implementación realizada.

3.1. Algoritmo base de manipulación de objetos

El algoritmo de detección de puntos de agarre que mejor se desempeñó de los analizados en la Sección 2.4, debido a tener la mejor tasa de éxito y tener un aceptable tiempo de procesamiento tal que puede usarse en tiempo real, corresponde al de Stueckler et al.[68]: *Real-Time 3D Perception and Efficient Grasp Planning for Everyday Manipulation Tasks*.

Este algoritmo se usó como base para la implementación realizada en este trabajo, por lo que su descripción engloba las etapas necesarias y la forma de procesar los datos. Todos los parámetros usados en las siguientes secciones fueron ajustados en los experimentos realizados en [68], por lo que en la implementación que se realizó en este trabajo, se modificaron de acuerdo a las restricciones del robot PR2.

A continuación se explicarán los detalles de cada una de las etapas en dicho algoritmo.

3.1.1. Segmentación de escena

Es necesario encontrar el plano de la mesa para poder segmentar los objetos que están sobre ésta. Para este cálculo se hace uso de RANSAC [16] sobre la nube de puntos de la escena, considerando además de la distancia de los puntos al modelo, la distancia angular entre normales de los puntos y la normal del modelo de plano, de forma de restringir la

inclinación del plano a valores cercanos a la horizontal. Para evitar errores, es necesario considerar una vecindad para calcular la normal (ver Figura 3.1), y de forma que el cálculo sea eficiente, se usan imágenes integrales para calcular las normales.

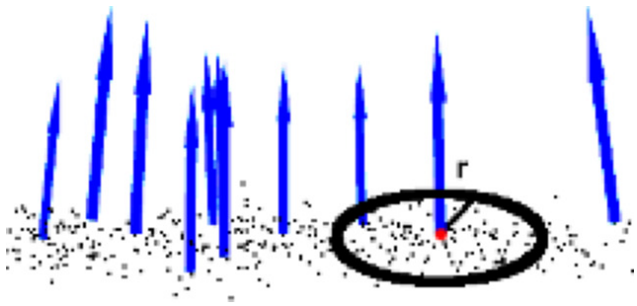


Figura 3.1: Normales calculadas usando radio.

Las restricciones de normales cercanas a la vertical que estén en una región de interés, al alcance del robot, ayuda a reducir considerablemente el tamaño del problema. La Figura 3.2 muestra varios planos en una escena sin restricciones; si se tomara el plano más grande es muy probable que se considere la pared o el suelo.

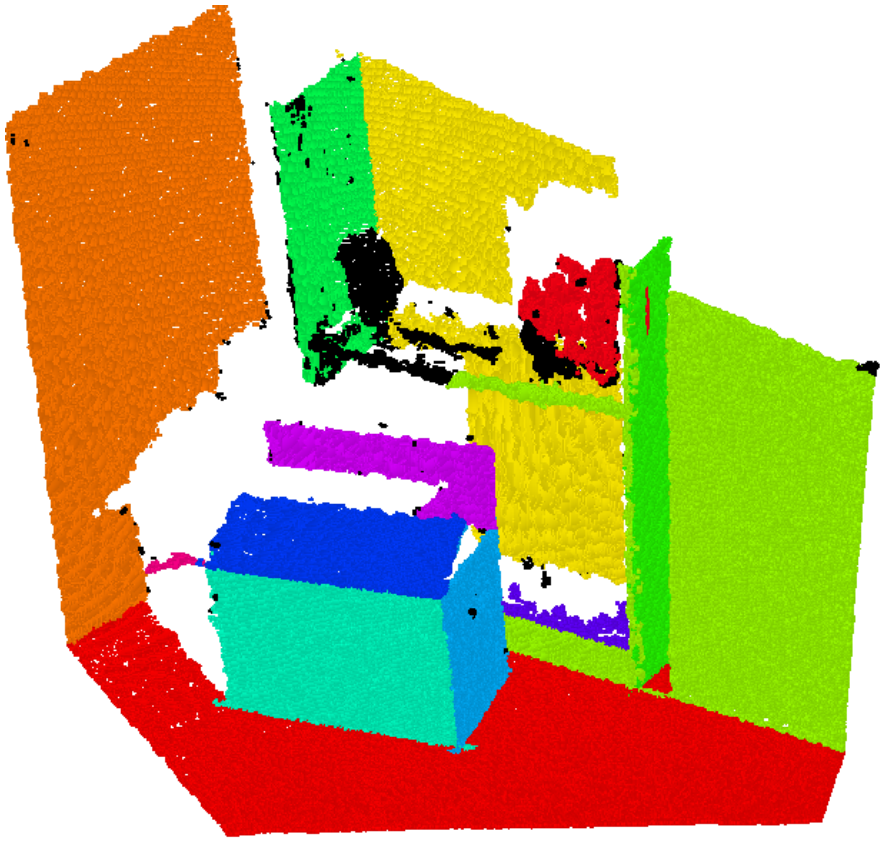
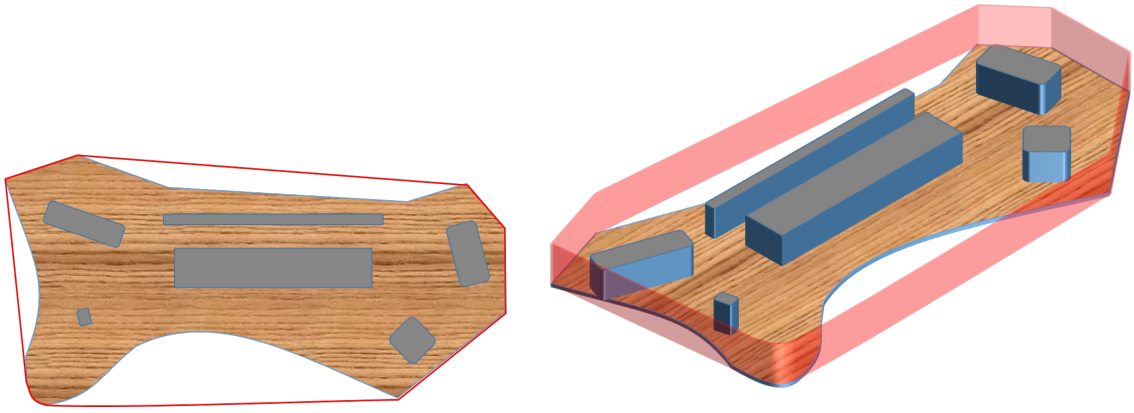


Figura 3.2: Ejemplo de segmentación de planos sin restricciones.

El siguiente paso es extraer los puntos sobre el plano encontrado y quedarse con los objetos cuya proyección yace dentro de la envoltura convexa del plano de soporte. Para clusterizar se asume que los objetos están separados por una pequeña distancia y se mantiene actualizado un rastreador multi-objeto. Esta es una de las simplificaciones del algoritmo, en situaciones

reales no siempre se cumple pero se sacrifican esas situaciones con el fin de tener un algoritmo rápido. La Figura 3.3 muestra un ejemplo de este proceso. Son parámetros del algoritmo la altura mínima y máxima del prisma con el que se extraen los objetos.



(a) Vista superior de la mesa **(b)** Vista en 3D

Figura 3.3: Envoltura convexa de una mesa con objetos. Rojo: envoltura convexa. Gris: objetos

3.1.2. Detección de puntos de agarre

Los agarres que se calcularán se basan en el supuesto de que el objeto tiene cierta simetría, dado que se usan los ejes principales de este para muestrear agarres. Los ejes principales son las 2 direcciones de máxima varianza en la nube de puntos del objeto y si el objeto es simétrico son los ejes de simetría del mismo.

Se definen 2 tipos de agarres posibles, horizontales y verticales. Para elegir el punto de agarre considerando la geometría del objeto se proyecta la nube de puntos de este último en el plano horizontal, y se calculan ejes principales de la distribución de puntos, la altura de la nube, centroide y el Bounding box alineado con los ejes principales. Con estos datos se realiza un muestreo de posibles agarres a distancias uniformes, considerando una posición de pre-manipulación a 10 cm del objeto y la posición de manipulación final en línea recta desde la anterior. La Figura 3.4 muestra un objeto y sus ejes principales proyectados, a partir de esta información se construye el Bounding box del tamaño de la nube con centro en los ejes principales a la altura media de la nube.

3.1.2.1. Agarre horizontal

El efector está paralelo al plano. Se muestrean agarres en una elipse en el plano horizontal a intervalos regulares de ángulos, manteniéndose a mínimo 10 cm del Bounding box. La altura del agarre es la mitad de la altura del gripper más 3 cm para seguridad. Esto permite darle mayor estabilidad al agarre en caso de pasar a llevar el objeto.

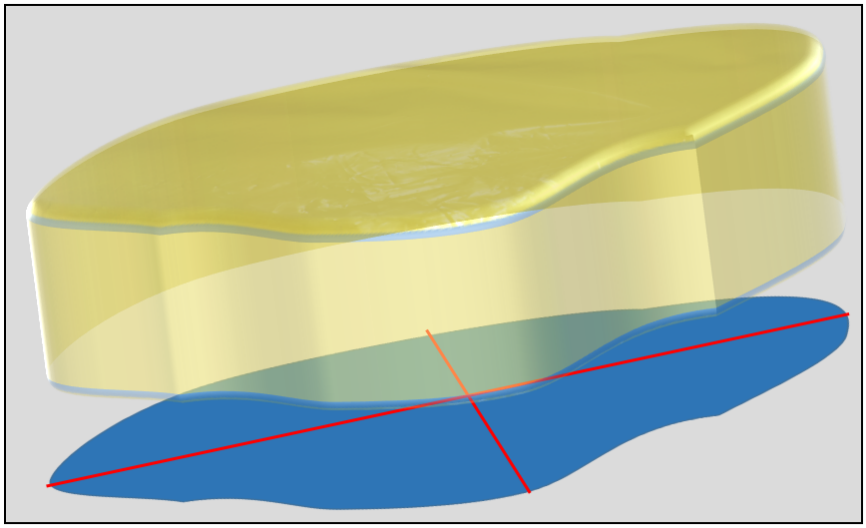


Figura 3.4: Objeto con su proyección y ejes principales en el plano de la mesa.

3.1.2.2. Agarre vertical

El efector está en 45° hacia el objeto manteniéndose paralelo al plano. Se muestrean agarres equidistántemente por ambos ejes principales a través del centro del Bounding box y se define la pose de pre-manipulación a 10 cm de la altura del objeto.

3.1.3. Filtrado de candidatos

Se considera un conjunto de criterios para filtrar los agarres muestreados:

- Ancho del agarre: Se rechazan agarres donde el ancho ortogonal del objeto sea mayor a la apertura del gripper
- Altura del objeto: Se rechazan agarres laterales si la altura del objeto esta por debajo de un umbral.
- Alcance: Se filtran aquellos agarres que estén fuera del alcance de los brazos.
- Se filtran los agarres que presentan colisiones.

Para filtrar las colisiones se usan condiciones geométricas simples. Se proyectan los puntos en el plano horizontal y se considera un círculo al rededor del hombro, usando el largo de la parte superior del brazo como radio r_u; un cono en dirección opuesta a la de agarre, del largo del antebrazo r_f y ángulo entre la dirección de agarre y el hombro α; un círculo del diámetro máximo del gripper r_g en la posición de agarre y se extiende el cono hacia el centro del robot para considerar el movimiento inicial. Esta figura representa el área de búsqueda de colisiones. La figura 3.5 muestra el área que debe estar libre de obstáculos.

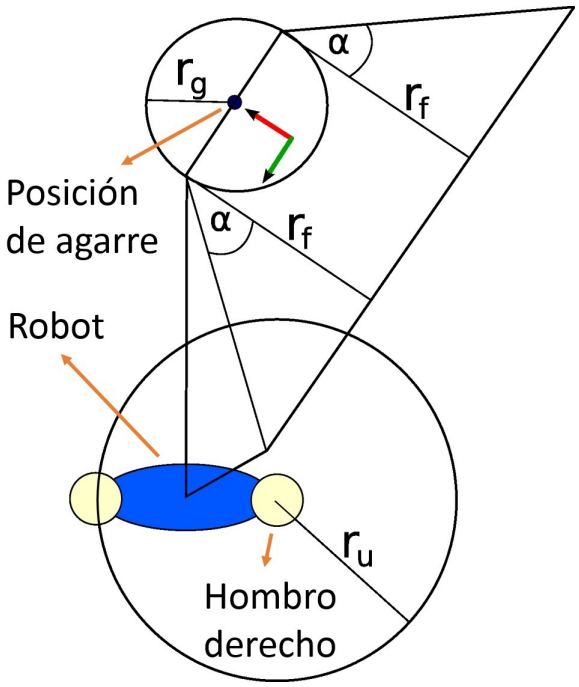


Figura 3.5: Diagrama restricciones geométricas para el cálculo de colisiones. Azul: cuerpo del robot visto desde arriba. Beige: corresponde al hombro izquierdo y derecho del robot, con el círculo de radio r_u centrado en el hombro derecho

3.1.4. Puntuación de agarres

Los agarres que pasaron la etapa de filtrado es necesario puntuarlos para decidir cual es el mejor. Se usan los siguientes criterios:

- Distancia al centro del objeto: Se favorecen los que están cerca del centro.
- Ancho del agarre: Se favorecen aquellos con un ancho de 8 cm.
- Orientación: Se prefieren aquellos con un ángulo α pequeño entre la línea hacia el hombro y la dirección del agarre.
- Distancia al robot: Se favorecen aquellos con menor distancia al hombro.

La puntuación se realiza por separado para agarres horizontales y verticales, y entre los ganadores se elije uno considerando la relación de su altura con su largo en el plano horizontal, dando una pequeña ventaja a los agarres horizontales dado que son más rápidos.

3.1.5. Experimentos

En los experimentos se usó una cámara de resolución 160x120, lo que dió $16Hz$, mientras que la Kinect usada en este trabajo tiene una resolución de 640x480, con lo que Stueckler obtuvo $6Hz$, por lo que se espera que los tiempos que se obtengan sean mayores.

Los tiempos obtenidos por Stueckler para cada etapa se muestran en la tabla 3.1. Si consideramos además el tiempo de adquisición de datos, y el de ejecución del agarre, se

obtienen *15s* para un agarre lateral y *25s* para uno vertical.

Tabla 3.1: Tiempos de procesamiento

Etapa	Tiempo promedio [ms]	Desviación estándar
Estimación de normales	7.2	2.4
Segmentación de escena	11.9	1.4
Clustering de objetos	41.6	1.5
Cálculo de puntos de agarre	98.1	9.1
Total	**158.8**	

3.1.5.1. Objetos usados

Se usaron objetos comunes de un ambiente de oficina como un plátano, caja de filtros de café, caja de té, taza, chicle, pañuelos, ropa y lápiz.

3.1.5.2. Resultados de agarres

Para cada objeto del set se hicieron 10 pruebas con diferentes orientaciones, obteniéndose éxito en todas las pruebas excepto un agarre superior para los pañuelos desechables, dando una tasa de éxito de 98.75 %.

3.2. Consideraciones preliminares

Se tuvieron en consideración simplificaciones, restricciones y alcances del trabajo en la implementación realizada.

3.2.1. Simplificaciones

- Dado que se quiere enfrentar solamente el problema del grasping, para todas las tareas adicionales necesarias, se utilizarán los algoritmos por defecto disponibles en ROS.
- Se ignorará el proceso de acercamiento a la zona de manipulación, comenzando con el robot posicionado frente a los objetos a manipular.
- No se requiere de trasladar el objeto para considerarlo un grasping válido, por lo que basta con levantarlo durante algún tiempo.
- Por la naturaleza del algoritmo se espera que los objetos manipulables presenten cierto grado de simetría respecto de sus ejes principales.

3.2.2. Restricciones y alcances

- Se considerarán casos con objetos estáticos solamente.
- Solo se considerarán objetos que puedan encontrarse en una casa u oficina.
- Solo se usarán objetos que puedan manipularse con una mano.

3.3. Trabajo de preparación

Se realizaron tareas necesarias antes de proceder con la implementación del algoritmo: se instalaron librerías necesarias en el robot y se realizó una segmentación simplificada para familiarizarse con las librerías y determinar su comportamiento.

3.3.1. Instalación de software

Para permitir la ejecución de librerías modernas de manipulación y movimiento, junto con contar con un entorno más estable de trabajo fue necesario actualizar la versión de ROS del robot de Groovy a Hydro, luego de lo cual se pudo instalar MoveIt!. Hydro es la nueva versión estable del robot PR2 desde febrero 2015 y viene a cambiar y mejorar muchas cosas que no funcionaban bien en Groovy. Los cambios más importantes entre estas versiones son:

- Los nodos *pr2_kinematics* y *pr2_arm_kinematics_constraint_aware* de cinemática inversa y el stack *pr2_arm_navigation* han sido descontinuados e integrados dentro de la librería MoveIt!. No toda la funcionalidad provista por estos paquetes ha sido cubierta y la forma de abordar los problemas es diferente, por lo que tanto la lógica de los algoritmos como el código ha cambiado.
- El Dashboard, herramienta que permite visualizar el estado del robot, mensajes de error, estado de sistemas y carga de la batería ha sido portado al nuevo estándar, ahora llamado *rqt_pr2_dashboard*.
- El simulador Gazebo, que antes formaba parte de ROS, ahora es un paquete independiente y debe ser instalado por el sistema de paquetes del sistema operativo. Se dispone del paquete *gazebo_ros_pkgs* para hacer de interfaz entre ROS y Gazebo. Se necesita una versión específica de Gazebo para cada versión de ROS. En Hydro es Gazebo 1.9, siendo que la actual es 6.1, por lo que la versión de Gazebo disponible presenta algunos errores y la falta de importantes características, además de funcionar muy lento debido a la alta carga de la simulación.
- La librería de trabajo con nubes de puntos PCL también fue separada completamente de ROS, de forma que ya no depende de este ni de sus tipos. Se ha dispuesto del paquete *pcl_conversions* para hacer conversiones entre mensajes de ROS y los tipos de datos de PCL. Este cambio requiere varias modificaciones en el código fuente, por lo que código antiguo deja de funcionar al actualizar a Hydro.
- La librería *tf* se ha deprecado en función de la librería *tf2*. Se mantiene compatibilidad entre ambas y una clara forma de hacer la transición, por lo que se mantiene el código

que usa *tf* por ahora.

Debido a la arquitectura interna del robot para actualizarlo es necesario que arranque por red desde la BaseStation, que lo proveerá con una versión modificada de Ubuntu 12.04 . Como el arranque por red falló al intentar la actualización, se necesitó acceder[1] a C1 y C2 para conectar pantallas y teclados, además de modificar el script de instalación en python, pues tenía comandos que fallaban. Tras esto se pudo continuar con la reinstalación[2].

Para probar que la instalación funcione correctamente se ejecutaron los nodos de teleoperación con joystick y se usó la interfaz *Moveit Commander* para mover el brazo del robot a posiciones aleatorias, ambas con éxito.

3.3.2. Segmentación con datos simulados simples

Para comenzar el trabajo y familiarizarse con las herramientas a utilizar se implementaron versiones simplificadas de las etapas de procesamiento para probar el funcionamiento básico de las mismas. Se usó un pequeño conjunto de datos generados a mano que permitiera apreciar el procesamiento. PCL tiene varias funciones que permiten realizar gran parte del trabajo que se quiere hacer si se usan de forma correcta. La implementación fue hecha en C++ dado que se requiere de gran velocidad de procesamiento.

Se implementó RANSAC clásico para segmentar un plano perfecto de 1000 puntos con 30 % de outliers de forma satisfactoria (ver Figura 3.6a). Luego se repitió el proceso pero agregando las normales a la superficie como entrada a la segmentación. Esto también tuvo éxito (ver Figura 3.6b).

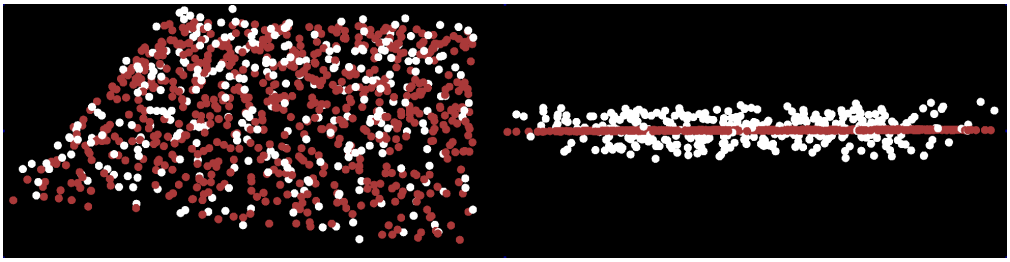
Luego de tener el plano se deben proyectar los puntos sobre éste, calcular la envoltura convexa y extraer todos los puntos que quedan dentro de un prisma definido por esta hasta cierta altura sobre el plano (la que será nuestra mesa). Estas etapas se realizaron utilizando funciones que PCL tiene para cada caso, también con resultados favorables.

3.4. Implementación del algoritmo

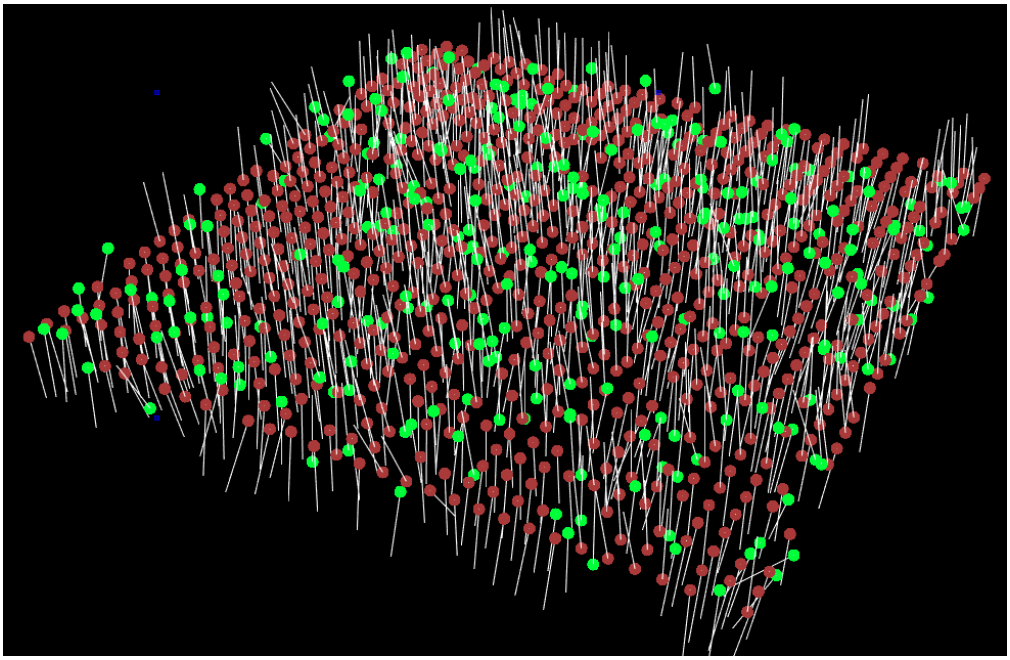
En esta sección se detallará la implementación realizada, separándola en sus etapas más importantes.

[1]Intrucciones de acceso con fotos: `https://rych.dcc.uchile.cl/doku.php?id=documentacion:pr2:abrir-el-robot`

[2]Instrucciones detalladas de reinstalación realizada: `https://rych.dcc.uchile.cl/doku.php?id=documentacion:pr2:reinstalacion-pr2`



(a) Rojo: plano encontrado. Blanco: outliers



(b) Rojo: plano encontrado. Blanco: normales. Verde: outliers.

Figura 3.6: Prueba de segmentación de plano con RANSAC, usando datos sintéticos casi perfectos.

3.4.1. Segmentación

Una función de callback en un thread secundario guarda la nube de puntos cada vez que es enviada por el sensor y el procesamiento de datos se hace en el thread principal, puesto que realizarlo en el callback bloquea las llamadas de ROS y puede tener efectos indeseados. Este procesamiento se explica en el algoritmo 1.

Algoritmo 1 Algoritmo de segmentación.

Entrada: Datos del sensor. Nube recibida en tópico */camera/depth/points*
Salida: Vector de clusters de objetos

1: Cortar nube de puntos
2: Cálculo de normales con imágenes integrales
3: Ajustar plano mediante normales
4: Proyectar puntos del plano sobre el mismo
5: Calcular envoltura convexa de la mesa
6: Filtrar puntos sobre la envoltura convexa
7: Realizar clustering de objetos

3.4.1.1. Intentos fallidos

Como en la nube hay mucha información inútil (como pared o piso) que está muy lejos para que el robot actúe sobre esta, se intentó cortar la nube de puntos usando umbrales de distancia (Filtro Passthrough de PCL), pero eso rompía la organización [1] de la nube de puntos, y se requiere que la nube esté organizada para poder usar imágenes integrales.

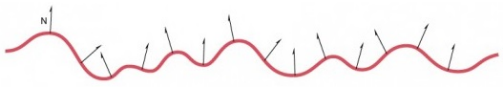
Los datos reales de una Kinect presentan ruido que no es fácil de modelar a partir de datos simulados. Al usar datos reales la etapa de segmentación de plano dejó de funcionar, puesto que las normales presentan una gran variabilidad entre elementos vecinos lo que imposibilita aproximar un plano con ellas (ver Figura 3.7). Esto motivó el suavizado de la nube usando un filtro Voxelgrid (sub-muestreo) y suavizado gaussiano (implementación manual). Ambos filtros fueron descartados porque el primero rompía la organización y el segundo tomada demasiado tiempo (˜10s).

Se consideró utilizar métodos que rompieran la organización, para luego reconstruirla con algún algoritmo. Sin embargo se necesitan algoritmos específicos para cada método usado y estos tienen una alta complejidad computacional, por lo que se perderían todos los beneficios de tiempo de usar una nube organizada.

3.4.1.2. Etapas del algoritmo

Todos los parámetros usados en la segmentación son modificables usando `dynamic_reconfigure`.

[1] Una nube de puntos es organizada cuando cada pixel de profundidad se le puede asociar un pixel de la imagen y por tanto 2 pixeles contiguos en la nube también lo están en la imagen. Se puede aprovechar esta propiedad para hacer algunos cálculos de forma eficiente.



(a) Alta definición produce normales disparejas

(b) Normales deseadas en precencia de ruido

Figura 3.7: Normales ruidosas

Etapa 1: Cortar nube Corresponde a cortar los bordes de la imagen asociada a la nube de puntos para disminuir el tamaño del problema (esto elimina los puntos de la nube asociados a los segmentos cortados de la imagen). Esto se hace aprovechando el conocimiento previo de que la mesa ha de estar centrada en la imagen y los bordes corresponden a muro, suelo y otros elementos irrelevantes. En un caso de aplicación real es necesario que el algoritmo de navegación posicione correctamente al robot antes de iniciar la manipulación.

Parámetros:

- `scale` Determina el tamaño de la ventana final
- `xTranslate` **e** `yTranslate` Determina la posición de la ventana final

Etapa 2: Cálculo de normales Realiza el cálculo de normales usando imágenes integrales. Como usar una vecindad mayor no tiene mayor influencia en el costo del algoritmo se usa el método más costoso:*Matriz de covarianza*, donde se crean 9 imágenes integrales para calcular un punto. La utilización de los métodos *Promedio 3D de gradiente*, *Promedio de cambio en profundidad* y *Gradiente simple 3D* dieron malos resultados por lo que se descartaron.

Parámetros:

- `normalEstimationMethod` Método de estimación. Se usa matriz de covarianza como principal.
- `maxDepthChangeFactor`[2] Máximo cambio aceptable en profundidad. Es un factor que multiplica a la profundidad en z de un punto (eje de la cámara) al compararla con la profundidad de sus vecinos derecho y abajo. Si la diferencia de profundidad entre puntos adyacentes es mayor que el doble de este valor, los puntos se consideran como NaN (diferencia de profundidad cero).
- `useDepthDependentSmoothing` Determina si se usará la profundidad en el suavizado de la vecindad.

[2] `https://github.com/PointCloudLibrary/pcl/blob/master/features/include/pcl/features/impl/integral_image_normal.hpp#L745`

- `normalSmoothingSize` Tamaño de la vecindad de suavizado. Ayuda a evitar outliers en las normales.

Etapa 3 Ajuste de plano Usa las normales calculadas para encontrar un plano usando RANSAC. En esta etapa podemos aprovecharnos de las restricciones del problema y dejar afuera inmediatamente todas las normales que estén muy alejadas del área de trabajo del robot, aquellas que no representen un plano horizontal y aquellas que estén muy abajo o muy arriba para representar una mesa. PCL tiene clases de modelo diferentes para cada restricción que se quiera imponer sobre el plano, y se seleccionan con un enum. Se utilizó `SACMODEL_NORMAL_PARALLEL_PLANE` que permite usar normales, definir un eje y un ángulo ε de forma de asegurar que las normales del plano estén a lo más ε grados del eje, y definir una distancia desde el origen con su respectivo umbral de aceptación. Debido a un error en la librería no se puede acceder al valor del umbral y por tanto no se puede usar la distancia al origen. Para definir el eje en que debe estar el plano (eje normal a la mesa), se crea un vector apuntando en el eje Z, lo que equivale al `/base_footprint`, y luego se transforma usando *tf* al frame de la Kinect (`head_mount_kinect_ir_optical_frame`). Para estar seguros de que el plano que se obtiene es válido se sigue adelante solamente si el plano calculado está entre 60 cm y 90 cm desde el suelo y se verifica que su normal este a 15°de la vertical.

Parámetros:

- `normalDistanceWeight` Peso de la distancia entre normales en la decisión. Equilibra ajustar la distancia al modelo y el ángulo de las normales. Se eligió 0.1.
- `maxIterations` Número de iteraciones de RANSAC antes de terminar.
- `distanceThreshold` Umbral de distancia para considerar un dato dentro del modelo.
- `optimizeCoefficients` Determina si, luego de encontrar un modelo, se optimizan los coeficientes usando solo los inliers del modelo.
- `probability` Probabilidad de elegir un punto libre de outliers en la etapa de muestreo.
- `sampleMaxDistance` Máxima distancia al muestrear puntos.
- `planeX`, `planeY`, `planeZ` Definición del plano normal al modelo a encontrar.
- `epsAngle` Máxima diferencia posible entre el modelo y un eje.

Etapa 6 Filtrado sobre la mesa Se encarga de conservar todos los puntos contenidos dentro de la envoltura convexa de la mesa, o sea aquellos que están sobre la mesa. Se determina un prisma con una altura determinada experimentalmente, tal que los puntos que estén dentro del prisma son los que finalmente se conservan. La altura del prisma está determinada por las restricciones físicas del robot en cuanto a la carga que puede manipular y la estabilidad de un agarre de un objeto demasiado largo con una sola mano. Para utilizar la envoltura convexa de PCL es necesario agregar el punto inicial al final de forma de tener una envoltura

cerrada.

Parámetros:

- `minHeight` y `maxHeight` Coordenadas de altura mínima y máxima del prisma en la mesa. Se extraerán todos los puntos que queden dentro.

Etapa 7 Clustering La última etapa de la segmentación, corresponde al clustering de los puntos restantes, el que se realiza comparando la distancia entre dos puntos y aceptándolos como miembros de un cluster si están dentro de un rango y si el cluster no ha superado el máximo de puntos. Luego se descartan los clusters que tengan pocos puntos puesto que corresponden a ruido u objetos no manipulables. Como en esta etapa solo se cuenta con los puntos que son partes de objetos, y se ha restringido la distancia mínima entre ellos, el clustering es directo, y la salida es un vector de nubes de puntos donde cada elemento corresponde a la nube de un objeto sobre la mesa.

Parámetros:

- `clusterTolerance` Distancia máxima para que un punto pertenezca al cluster.
- `minClusterSize` y `maxClusterSize` Límites de número de puntos en un cluster. Se descartan clusters fuera de los límites.

3.4.1.3. Visualización

La gran cantidad de normales generadas hace imposible visualizarlas en `rviz`, puesto que este crea un objeto 3D con su respectivo modelo para cada una, lo que satura el procesador y la memoria al tratarse de más de 10.000. Es necesario un visualizador más versátil y eficiente, por lo se utiliza el visualizador de PCL, que permite mucha flexibilidad para manipular la visualización y además permite fácilmente ver las normales calculadas, modificar el número a mostrar, y su tamaño.

ROS y el visualizador deben llamar a una función (`spinOnce`) de forma periódica para atender sus eventos, por lo que se ejecuta el visualizador en un thread aparte. Es necesario compartir entre los threads las nubes de puntos que se quieren visualizar, por lo que el proceso principal notifica al visualizador cuando ha actualizado los datos, modificando un flag y liberando un mutex para que este proceda a visualizar la información a partir de un puntero a la clase que contiene los datos. Esto evita bloqueos de la interfaz del visualizador o de la segmentación.

Restricciones de implementación Debido a restricciones de la clase `PCLVisualizer`, que se basa en la librería `VTK`, no se pueden llamar a funciones de visualización desde diferentes

threads para el mismo objeto[3], y tampoco se pueden llamar a estas funciones para objetos diferentes, por lo que todas las instancias de esta clase deben ejecutarse en el mismo thread. Otra consideración es que los objetos de la clase deben crearse en el thread donde se ejecutan, puesto que es necesario destruir el objeto para cerrar la ventana, de lo contrario la ventana se congela al presionar el botón de cerrar, debido a un error en la librería `VTK`. Ambas situaciones no están clarificadas en la documentación disponible, por lo que corresponde a conclusiones experimentales y fruto de hilos de discusión en el foro de PCL.

3.4.2. Detección

Se ordenan por tamaño los clusters de objetos de la etapa de segmentación y se elige uno (configurable usando `dynamic_reconfigure`). El algoritmo 2 detalla el proceso.

Algoritmo 2 Algoritmo de detección.

Entrada: Nube de puntos de objeto
Salida: Punto de agarre elegido
1: Cálculo de Bounding box orientado con los ejes principales
2: Muestreo de puntos de agarre laterales y superiores
3: Filtrado de puntos de agarre infactibles
4: Puntuación de puntos de agarre
5: Selección del mejor punto de agarre

3.4.2.1. Etapa 1 Cálculo de Bounding box

Se requiere que el Bounding box esté alineado con la mesa, así que para forzar esto se calcula el Bounding box en 2D de la proyección del objeto sobre la mesa y luego se obtiene la altura del objeto.

El algoritmo 3 detalla el cálculo del Bounding box 2D. Es necesario forzar que los vectores propios tengan cierta orientación debido a que no son únicos y queremos que el sistema de coordenadas que se obtenga sea siempre el mismo para ese objeto. Usando esta transformación se transforma la nube 3D del mismo y se calcula la altura, con lo que se tienen las propiedades finales del Bounding box. Finalmente se publica la transformación del sistema de coordenadas del objeto, centrado en el centro del Bounding box, a la librería *tf*; esto permite que para transformar puntos de un sistema de coordenadas a otro solo basta llamar a *tf* especificando el frame del objeto.

3.4.2.2. Etapa 2 Muestreo de puntos de agarre

Para ejecutar un agarre se llama al cliente `actionlib` con un mensaje `moveit_msgs::Grasp`. Este cliente comunica el objetivo a un servidor iniciado por MoveIt!, el que realiza la acción.

[3] `http://docs.pointclouds.org/trunk/classpcl_1_1visualization_1_1_p_c_l_visualizer.html#details`

Algoritmo 3 Algoritmo de cálculo de Bounding box 2D.

Entrada: Proyección de objeto sobre la mesa (N)

Salida: Bounding box alineado con la mesa y ejes principales {respecto de sistema de coordenadas de la Kinect}

```
1: C = calcularCentroide(N)
2: M = calcularMatrizCovarianza(N, C)
3: V = calcularVectoresPropios(M)
4: V = forzarOrientaciónPositiva(V)
5: tamaño, posición, orientación = calcular(V, C, N)
```

Los elementos fundamentales del mensaje son:

- `pre_grasp_posture` Posición de las pinzas antes de tomar el objeto. Deben configurarse los valores de todos las articulaciones que se quieren mover.

- `grasp_posture` Posición de las pinzas al tomar el objeto. Deben configurarse los valores de todos las articulaciones que se quieren mover.

- `grasp_pose` Pose del efector final al tomar el objeto.

- `pre_grasp_approach` Dirección de acercamiento a la posición del agarre y distancia a la que se ubica la posición de pre-agarre

- `post_grasp_retreat` Dirección de alejamiento de la posición del agarre y distancia a la que se ubica la posición de post-agarre

- `allowed_touch_objects` Objetos que se permite tocar en el proceso, desde el agarre a la posición de post-agarre

El efector final es `r_wrist_roll_link`, que corresponde a la muñeca, no al punto entre las pinzas (`r_gripper_tool_frame`) donde el objeto hará contacto efectivamente, por lo que es necesario desplazar todos los puntos de agarre 18 [cm], la separación entre ambos, en la dirección opuesta al movimiento (ver Figura 3.8).

Las formas de realizar el muestreo son diferentes para agarres laterales y superiores, por lo que en la clase `GraspSampler` se tienen métodos diferentes que hacen los cálculos correspondientes.

Nota: En la documentación se señala que la articulación activa de las pinzas se llama `r_gripper_joint` y que representa un desplazamiento horizontal, sin embargo el tutorial y ejemplos disponibles para mover las pinzas no funcionan o presentan movimientos impredecibles, por lo que se optó por obtener la lista de articulaciones activas del grupo `right_gripper` y moverlas todas[4].

[4]Después se descubrió que la articulación activa se llama `r_gripper_motor_screw_joint`, sin embargo no se encontraron referencias a esto en la documentación más que en la implementación misma del controlador.

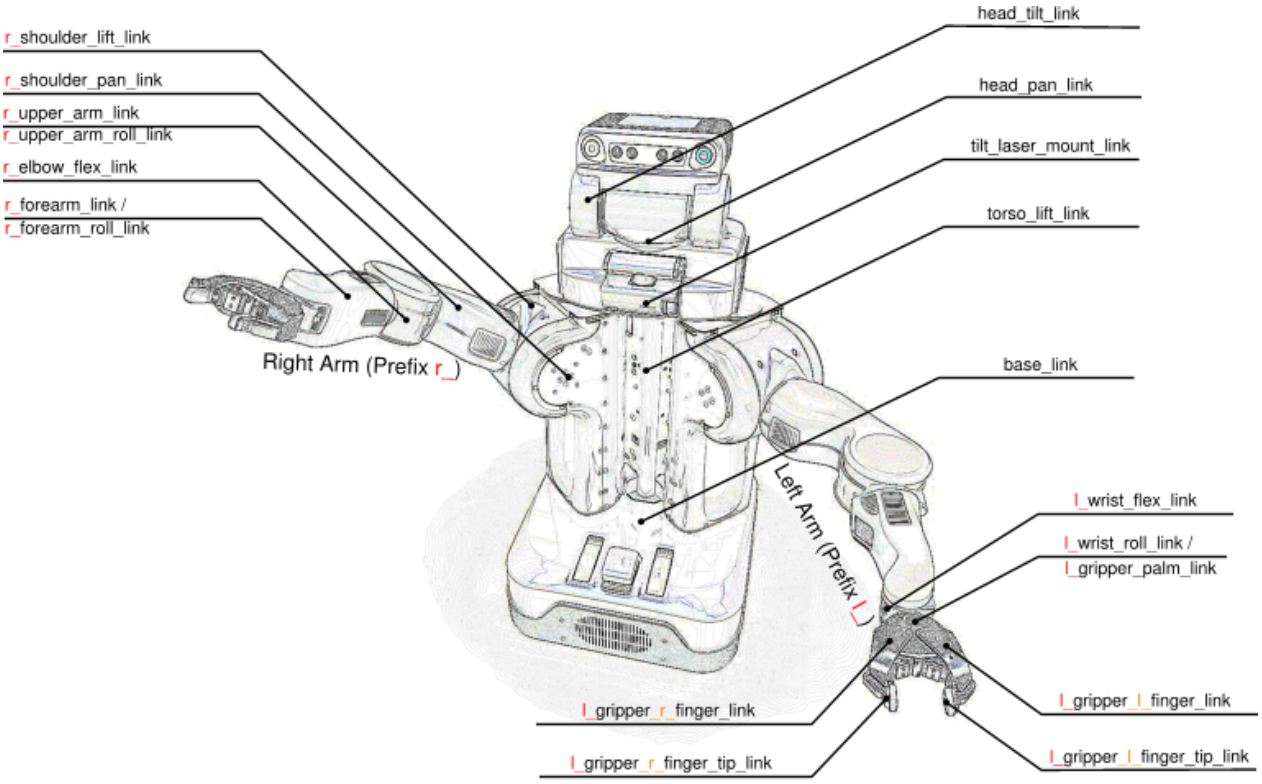


Figura 3.8: Frames del Robot PR2

Agarres laterales Se definen en una elipse alineada con la mesa alrededor del objeto, donde el gripper apunta al centro del objeto. Al muestrear en intervalos angulares regulares se tiene un exceso de puntos en el diámetro mayor de la elipse y muy pocos en el diámetro menor, por lo que para muestras bien distribuidas se requeriría un gran número de puntos que volverían lentas las siguientes etapas.

No hay una solución analítica a la ecuación de la elipse para muestrear uniformemente, por lo que se implementó una solución numérica que aproxima el muestreo uniforme dando pequeños *pasos* sobre la elipse. La clase `EllipseOperations` realiza el cálculo y sigue la parte básica del patrón de diseño *Iterator*, ya que permite consultar si quedan más puntos por muestrear (por defecto se muestrean 50) y obtener el siguiente. El algoritmo está basado en una respuesta[5] en el foro *StackOverFlow*.

Agarres superiores Corresponden a un muestreo de los ejes principales del Bounding box, un poco más arriba del centro y apuntando hacia el centro del objeto. Por defecto se muestrean 50 puntos repartidos entre ambos ejes proporcionalmente a su tamaño.

[5] `http://stackoverflow.com/a/20510150`

3.4.2.3. Etapa 3 Filtrado de puntos de agarre

Para filtrar puntos de agarre, se usa MoveIt!, aprovechándose de que sus métodos de planificación de movimiento detectan colisiones usando la librería FCL (Flexible collision library).

Configuración de la escena Es necesario configurar MoveIt! para que se permitan colisiones entre las pinzas del robot y el objeto a manipular, así como entre el objeto ya tomado por el robot y la mesa, de lo contrario al intentar tomarlo se reportarían colisiones entre las pinzas y el punto de contacto en el objeto, y al intentar levantarlo se reportarían colisiones del objeto con la mesa[6].

Se agregan 2 objetos de colisión a la escena de planificación de MoveIt!: La mesa, representada por una caja desde el suelo a la altura de la misma, para evitar trayectorias que pasen debajo de esta; y el objeto a tomar, representado por su Bounding box. Para crear el objeto de colisión de la mesa se calcula su Bounding box en 2D usando el método expuesto en el algoritmo 3, pero se ignora la rotación, debido a que los ejes principales de la mesa son sus diagonales y se necesita que esté alineada con los ejes del robot.

Al agregar objetos de colisión a la escena, MoveIt! borra los vóxeles del *Octomap* que entran en colisión con estos objetos. Ver Figura 3.9.

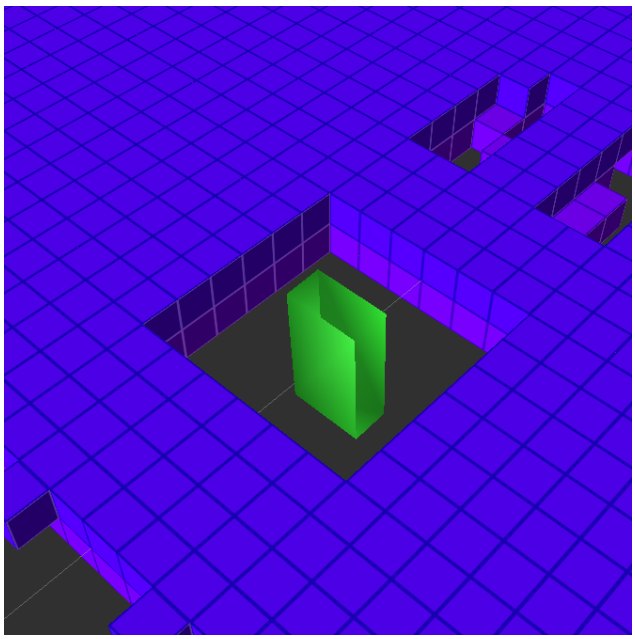


Figura 3.9: *Octomap* con objeto de colisión. Se borró *Octomap* más allá del que estaba en colisión para mostrar el comportamiento. Azul (Morado): *Octomap*. Verde: Objeto de colisión

Planificación del movimiento Para ejecutar un agarre `MoveGroup`, la interfaz principal de MoveIt!, provee una función llamada `pick`, que ejecuta el agarre definido en el mensaje

[6]Al vincular un objeto a un *link* del robot, MoveIt! verifica colisiones para el objeto vinculado también.

recibido, sin embargo esta interfaz tiene varias deficiencias:

- No provee una opción para solamente planificar el movimiento, sino que lo ejecuta.
- Ignora algunos parámetros configurados en el grupo activo, como tiempo de planificación.
- No determina un tiempo de espera para la acción, por lo que cualquier error en las etapas inferiores deja esperando indefinidamente.

El protocolo de comunicación de `actionlib` funciona sobre `TCP`, sin embargo para poder continuar la ejecución de una acción en situaciones de pérdida de paquetes y no esperar a la comunicación por red, `actionlib` agrega un protocolo no-confiable[7] encima de `TCP`. Esto funciona bien para el caso del feedback de la acción pero para el resultado puede presentar problemas dado que el servidor solo envía una vez el resultado y no se confirma la comunicación, por lo que si el mensaje que contiene el resultado se pierde, nunca se obtendrá información sobre esto. Este comportamiento (esperado por los desarrolladores de `actionlib`) produce que la función `pick` espere indefinidamente por una respuesta (lo que sucede cada cierto tiempo de forma imprevisible) forzando a cerrar manualmente el programa.

Por estas razones fue necesario replicar, desde el código fuente de `MoveGroup`, la infraestructura necesaria para esta funcionalidad. Se implementó el ciclo de vida de un cliente `actionlib`, donde se pueden configurar opciones adicionales. Debido a la falla antes mencionada, el servidor reporta a veces que hay una acción ejecutándose de forma indefinida, por lo que es necesario cancelar todas las acciones en curso manualmente cada vez que se quiere hacer una solicitud.

Como `MoveGroup` no presenta una interfaz para obtener el valor de algunos de sus parámetros, se tuvo que manejar de forma externa también el ciclo de vida de éstos, y configurarlos en los momentos apropiados.

Se cambió el algoritmo de planificación por defecto LBKPIECE, por RRT Connect debido a que este último es más rápido y no se necesita realizar tareas de planificación demasiado complejas. Se hicieron pruebas llevando el brazo desde una posición arriba a la derecha hasta la mesa (posición listo para agarrar a posible agarre lateral) donde tomó en promedio 0.008 [s] contra 0.006 [s] de LBKPIECE. Este cambio aunque pequeño es importante puesto que se hacen muchas solicitudes de planificación para filtrar colisiones.

Pre-filtrado Adicionalmente se realiza un pre-filtrado para evitar generar muestras si el objeto es muy bajo, y por tanto impide tomarlo.

3.4.2.4. Etapa 4 Puntuación de puntos de agarre

Se ordenan 4 listas de los agarres factibles, ordenándolas de menor a mayor por:

- Distancia al centro del objeto

[7] con pérdida de información

- Distancia al hombro del robot (`r_shoulder_pan_link`)
- Ángulo entre la línea del hombro al centro del objeto y la orientación del agarre
- Ancho del agarre cercano a un valor ajustado

Para los 3 primeros puntos se hace una comparación sencilla usando la función `std::sort` y un comparador para cada caso. El algoritmo 4 muestra la implementación del cuarto caso:

Algoritmo 4 Algoritmo de cálculo del ancho de un agarre.

Entrada: Pose del agarre (P), Nube del objeto (N)
Salida: Ancho del agarre (A)
1: $P = planoDelAgarre(P)$
2: $V = puntosEnVecindad(P)$
3: $V_{rotado} = rotarHastaEjes(V)$
4: $MAX, MIN = calcular(V_{rotado})$ {calcular máximo y mínimo en plano del agarre}
5: $A = MAX - MIN$

Teniendo las cuatro listas, el puntaje de un agarre se calcula como la suma de la posición de ese agarre en cada lista. Luego el mejor agarre será el que obtenga menor puntaje. Finalmente se ordena la lista de puntajes de menor a mayor y se normaliza para poder comparar ambas listas aunque tengan diferente número de muestras.

El cuarto criterio de puntuación no fue implementado actualmente y se deja como trabajo futuro, debido a que es necesario ajustar el ancho deseado del agarre considerando las características de las pinzas con las que cuenta el PR2, y la efectividad estadística del valor elegido respecto de su influencia en el éxito de los agarres.

3.4.2.5. Etapa 5 Selección del mejor punto de agarre

Si solo hay agarres laterales o superiores disponibles, se elije el primer elemento de la lista disponible. Si hay disponibles ambos tipos de agarres y la diferencia porcentual de puntajes entre el mejor de los laterales y el mejor de los superiores es menor al 5 %, se elige el agarre lateral debido a que es más rápido. De lo contrario se elige el mejor.

3.4.3. Ejecución del agarre

Para ejecutar un agarre se llama al cliente `actionlib` con el mejor agarre evaluado, señalando un tiempo para esperar su ejecución (10 [s]).

Se presentaron problemas al ejecutar la trayectoria debido a la implementación de los controladores del brazo y gripper del PR2, y a la implementación del cliente de `actionlib` en MoveIt!:

- El comportamiento esperado de los controladores es que frente a la imposibilidad de cumplir con la trayectoria debido a restricciones de tiempo, fuerza, posición u otras, se

anuncia el movimiento como abortado pero no se le ordena a los motores que se detengan en un intento por aproximarse lo más posible a la meta deseada. Este comportamiento puede ser problemático dado que requiere detener manualmente los motores enviando otra meta frente al aborto de una trayectoria, pero esta orden tiene el costo en tiempo asociado de al menos 4 comunicaciones por red, por lo que en situaciones delicadas podría significar que el robot rompa algún objeto o no detenga sus motores a tiempo antes de sufrir daño por chocar con algo. Parece haber una sobre restricción en la parametrización de la trayectoria hecha por el cliente debido a que con alta probabilidad se abortan movimientos al ejecutar trayectorias válidas cerca de objetos.

- Tanto Moveit! como el controlador calculan de forma diferente el tiempo esperado aproximado de ejecución de la trayectoria (distinto del tiempo de planificación señalado arriba), de forma de abortar la misma si luego de transcurrido ese tiempo no se ha terminado la ejecución por que se asume que ocurrió un problema que no fue detectado. Si bien en teoría el tiempo calculado por MoveIt! siempre debiera ser mayor, en ocasiones la trayectoria se aborta porque MoveIt! estima un tiempo menor del que le toma al controlador ejecutar la acción. Esto parece deberse a un error de implementación de MoveIt!, sin embargo no existe una forma clara de obtener el tiempo estimado calculado por el controlador de forma de usar el mismo valor en ambos paquetes.

Debido a estos errores, se abortan las trayectorias planeadas con alta probabilidad señalando una "Falla en la ejecución hecha por el controlador". A pesar de abortarse, a veces sí se culmina exitosamente el movimiento, lo que obliga a implementar una interfaz interactiva donde se puede solicitar volver a realizar el movimiento si este presenta error o volver a realizar el proceso desde el principio.

3.4.4. Detalles del software

Los diferentes módulos del software y frameworks utilizados se comunican en diferentes etapas del procesamiento como muestra la Figura 3.10.

Todo el proyecto[8] está en el `namespace bachelors_final_project` y se definieron además los `namespaces segmentation`, `detection` y `visualization` dentro del principal, de esta forma se separan los 3 principales módulos.

Se ha realizado un diseño buscando modularidad y eficiencia. Las secciones algorítmicas cuentan con el método `doProcessing` que especifica las etapas necesarias, haciendo fácil generalizar el comportamiento implementando el patrón de diseño *Template method*.

Para la segmentación la clase `CloudSegmentator` se encarga de recibir el callback de la nube de puntos y procesarla con funciones para las diferentes etapas. La mayoría de las variables son miembros públicos de la clase para poder acceder a ellas desde el visualizador. Se realiza una medición de tiempo para todos los métodos y chequeo de errores en las etapas que impidan continuar la ejecución.

Para la detección la clase `GraspPointDetector` procesa un objeto para encontrar su punto

[8] `https://github.com/ianyon/bachelors-project`

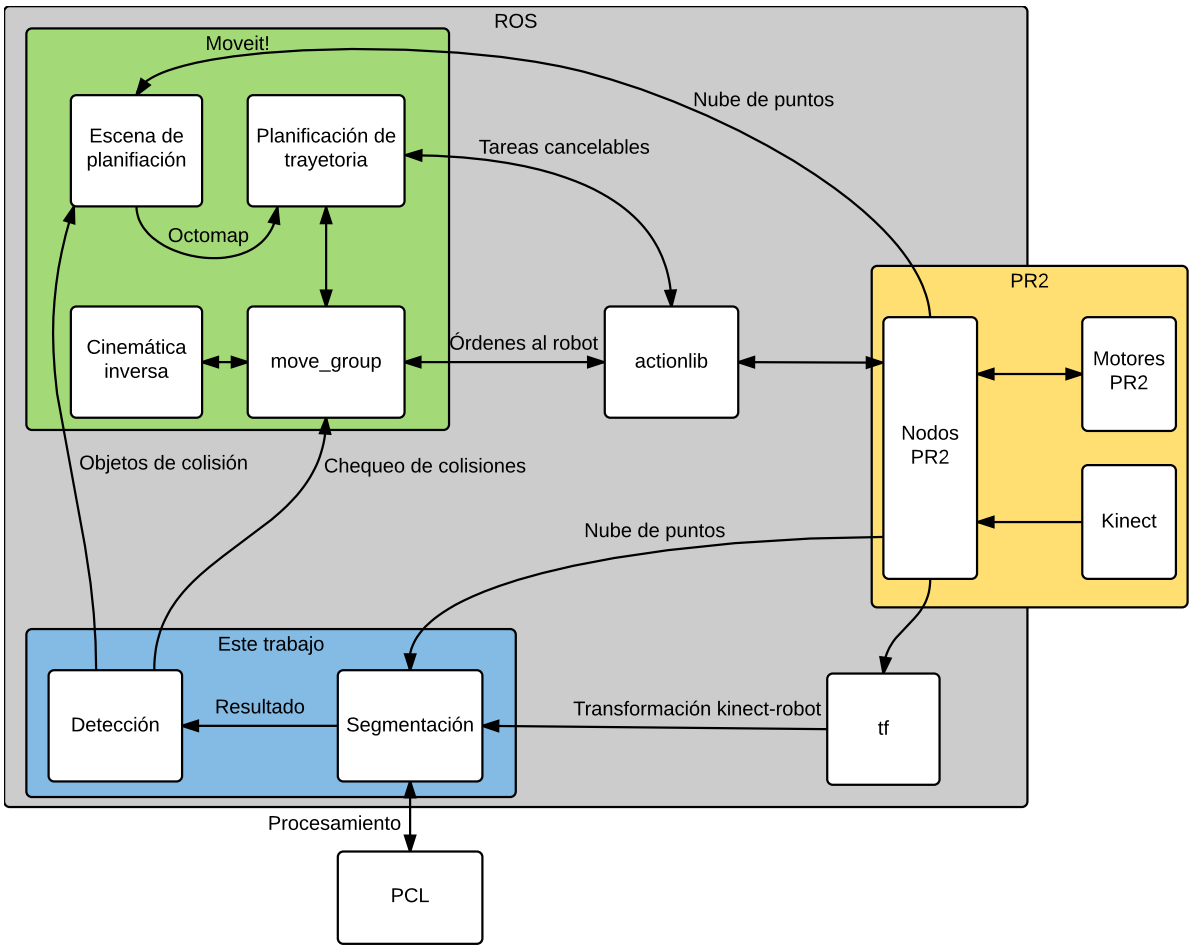


Figura 3.10: Resumen de componentes del software.

de agarre. Las etapas usadas están encapsuladas en clases por lo que es fácil extenderlo a patrón de diseño *Strategy*.

Debido a un error en la herramienta `dynamic_reconfigure`, fue necesario descargar su código y compilarlo para utilizarla.

La figura 3.11 muestra un esquema de las principales clases y sus principales métodos.

3.4.4.1. Tamaño y archivos utilitarios

Se usó el programa `cloc` para calcular el tamaño del software y se muestra en la tabla 3.2.

Los archivos YAML, XML y algunos otros corresponden a archivos *launch* y parámetros para los nodos que fueron modificados. Se crearon algunos scripts para tareas específicas:

`print_joints.sh` Imprime las posiciones de todos las articulaciones de los brazos del PR2

`change_model_position.py` Cambia la posición en *gazebo* de los brazos y otras articulaciones especificadas desactivando los controladores (para permitir mover las articulaciones)

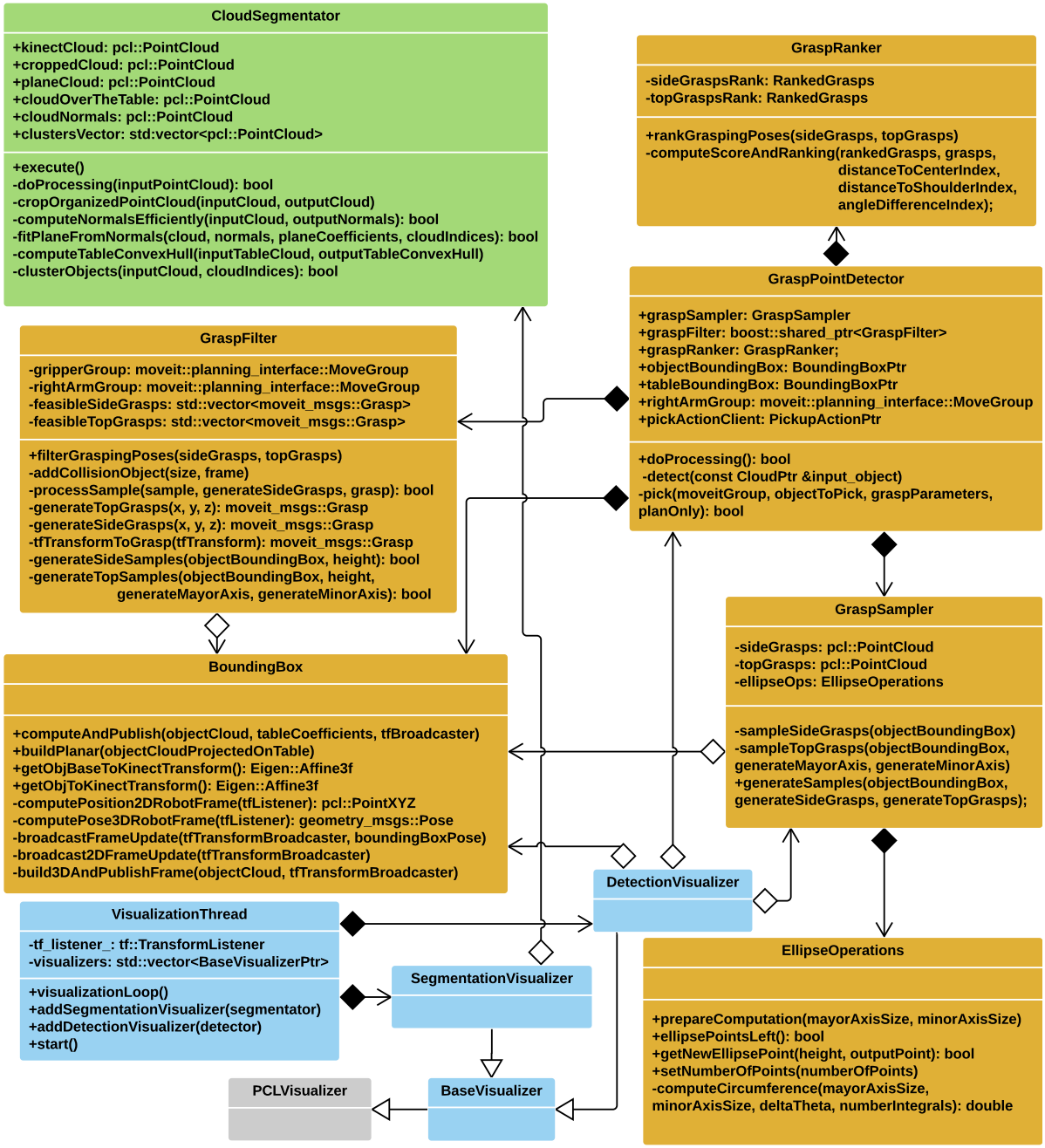


Figura 3.11: Diagrama de clases del software. Verde: módulo `segmentation`. Naranja: módulo `detection`. Celeste: módulo `visualization`.

Tabla 3.2: Líneas de programa por lenguaje

Lenguaje	Número de archivos	Líneas en blanco	Comentarios	Código
C++	17	502	278	2207
C/C++ Header	16	377	101	754
Bourne Shell	10	58	146	300
Python	1	40	30	137
XML	1	11	17	28
YAML	2	0	0	12
SUM:	47	988	572	3438

y luego reactivandolos. Deja la cabeza mirando hacia una mesa hipotética enfrente, el brazo izquierdo plegado (*tuck arm*) y el derecho levantado a un costado listo para tareas de manipulación.

`timed_roslaunch.sh` Permite ejecutar un nodo dentro de un archivo launch, con cierto retraso configurable.

3.4.4.2. Ambiente de desarrollo

Debido al uso indiscriminado de boost, Eigen, PCL, entre otras; todas librerías con fuerte uso de templates de C++, los tiempos de compilación se volvieron poco prácticos para el desarrollo, tomando en promedio 2 minutos para la compilación y llegando a veces a los 10 minutos!. Es por esto que se utilizaron diferentes herramientas para acelerar el proceso:

`gold` Es una mejora sobre el linker clásico (*ld*), desarrollado por Ian Lance Taylor y un pequeño equipo de Google.

`ninja` Es un sistema de construcción que reemplaza a make.

`clang` Es un compilador parte del proyecto llvm.

`ccache` Es un cache que evita recompilar archivos ya compilados cuando cambia solo una parte no dependiente del programa.

`cotire` Pequeño programa para *cmake* que habilita el uso de headers pre-compilados para las librerías del sistema, y unidad de compilación unitaria, que junta el código en una sola unidad de compilación.

Usando todas estas herramientas se obtienen tiempos promedio de 30 segundos por compilación.

Capítulo 4

Resultados y análisis

En este capítulo se muestran los resultados obtenidos. Primero se describe el ambiente de simulación utilizado en la sección 4.1, luego se muestran los resultados relativos a la segmentación, detección y ejecución; en la secciones siguientes. Finalmente en la sección 4.5 se muestran resultados de la ejecución en el robot PR2 real.

4.1. Ambiente de simulación

Para poder hacer pruebas que incluyan las etapas posteriores del algoritmo se diseñó una escena en gazebo que sirva como plataforma de pruebas. La figura 4.1 muestra la escena diseñada y el robot en su posición inicial como se ve en gazebo. Se creó un archivo launch que instancie el robot en gazebo, inicie sus controladores y sensores, inicie MoveIt!, `rviz`, y configure parámetros para cada nodo.

Para modificar la posición inicial del robot, se creó el archivo `change_model_positions.py` que deja al robot en una posición tal que tenga buena vista de la mesa y el brazo derecho listo para manipular. Para hacer esto es necesario apagar los controladores antes de mover una articulación y encenderlos luego, o de lo contrario estos impiden el movimiento de las articulaciones del robot.

4.1.1. Elementos del ambiente

Se colocaron varios objetos en posiciones aleatorias sobre la mesa. Hay objetos que el robot no puede manipular por ser muy anchos, altos o estar fuera de su alcance, y otros cuya manipulación no debiera ser un problema.

Figura 4.1: Escena de pruebas diseñada. El robot PR2 al fondo con los brazos hacia adelante. En la mesa varios objetos: 2 latas, una llave Stillson, un vaso, una pelota y un cilindro.

4.1.2. Configuración del ambiente

4.1.2.1. Gazebo

Gazebo corre considerablemente lento debido al proceso de simulación dinámica de físicas y a la ejecución de múltiples controladores y sensores simulados. En ocasiones el factor de velocidad, que mide la velocidad de la simulación respecto del tiempo real, baja de $0,2$, lo que significa que en vez de enviar nubes de puntos cada segundo (configuración por defecto), éstas se reciben cada 20 [s] o más.

Para acelerar la simulación se realizaron los siguientes cambios:

- Se modificó el parámetro `max_step_size` que controla el tamaño del paso de la simulación y se definió como 0.0035, con lo que se triplica la velocidad de simulación y se obtienen razonables físicas. Como ejemplo si se dejara en 0.004 se puede apreciar que el robot comienza a vibrar y al mover algo las cosas saltan de su lugar.
- Al cerrar gzclient (la interfaz gráfica de gazebo) se obtiene una mejora del 15 %.
- Se desactivó el cálculo de sombras en la simulación.
- Se modificaron los archivos que cargan los controladores y sensores del PR2 para que solo se cargaran los controladores de brazos, pinzas y cabeza y como sensor solo funcionara la Kinect.

Problemas Al ejecutar la simulación en un computador con una tarjeta de video integrada para notebooks Intel HD Graphics año 2009, la nube de puntos que se obtiene está saturada a 1 metro como se muestra en la figura 4.2. No se encontró información relativa a este error, ni formas de solucionarlo. Al cambiar a una tarjeta de video de escritorio Radeon HD 7790 año 2013 el error se solucionó. Otra situación es que al mover la escena en gazebo se producen extraños saltos en los movimientos de traslación y rotación, como si se resistiera al movimiento, lo que en ocasiones produce un *Segmentation Fault*. En todos los casos al cerrar gazebo se produce un *Segmentation Fault*.

Nota: En julio de 2015 se terminó el soporte para gazebo 1.9 que es la versión necesaria al usar ROS Hydro. ROS Indigo usa gazebo 2.2 y ROS Jade usa la versión 5 (la actual es la 6).

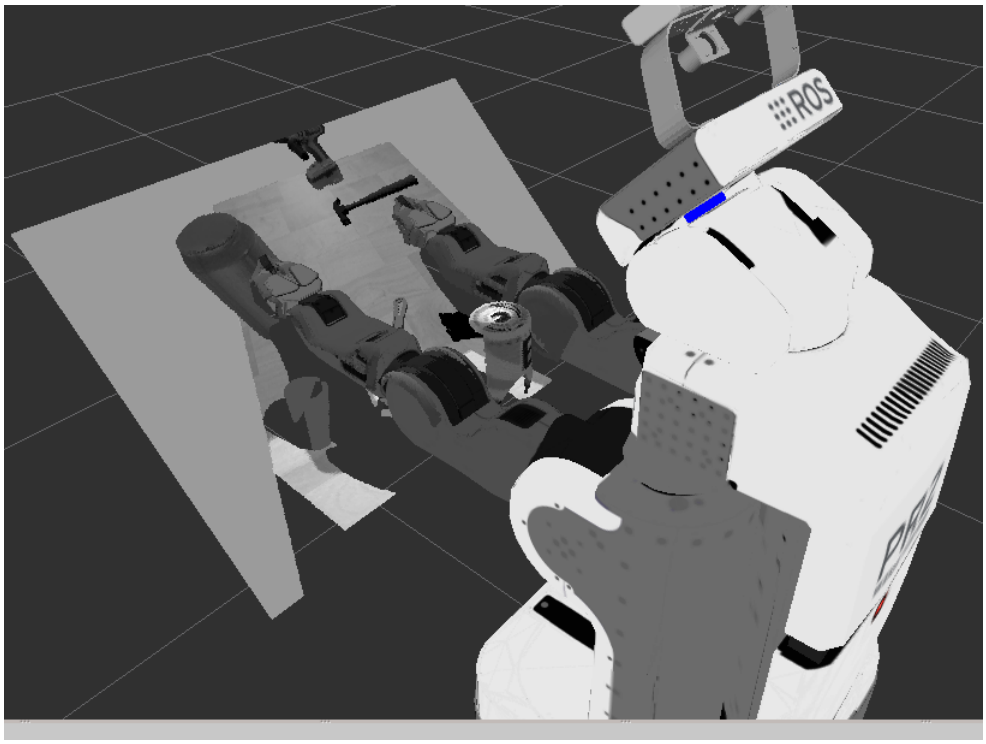


Figura 4.2: Nube errónea saturada. Solo el borde de la mesa, el vaso y una lata alcanzan a ser percibidos con profundidad, lo demás se percibe con profundidad constante en el espacio vacío haciendo inútil la nube de puntos.

4.1.2.2. MoveIt!

Resolución del *Octomap* Se modificó la resolución del *Octomap* generado por MoveIt! debido a que se perdían muchos detalles de los objetos y podrían producirse colisiones no detectadas por el sistema. Sin embargo, esto dejó en evidencia que el *Octomap* no estaba alineado con la nube de puntos del sensor y por tanto habían vóxeles de colisión en espacio vacío a un costado de los objetos y secciones de los objetos sin vóxeles que los marquen como colisión.

Al agregar objetos de colisión a la escena, estos borraban la mayor parte del objeto del *Octomap*, pero los vóxeles desalineados (ver figura 4.3) en ocasiones quedaban suficientemente fuera del objeto como para no borrarse, pero lo bastante cerca para marcar colisión. Esto impide cualquier manipulación sobre el objeto dado que en todos los casos se descartan debido a colisiones. En algunos casos habían vóxeles del *Octomap* que no se borraban aún cuando estos se encontraban completamente dentro del objeto de colisión[1]. Esto se debe a que el *Octomap* pasa por un proceso de *Ray Tracing* donde se agregan/borran los vóxeles que encuentran colisiones con los rayos, y la implementación del borrado de vóxeles dentro de un volumen es costosa computacionalmente basándose en la implementación actual, por lo que se han implementado soluciones imperfectas de forma temporal mientras se encuentra una mejor alternativa[2].

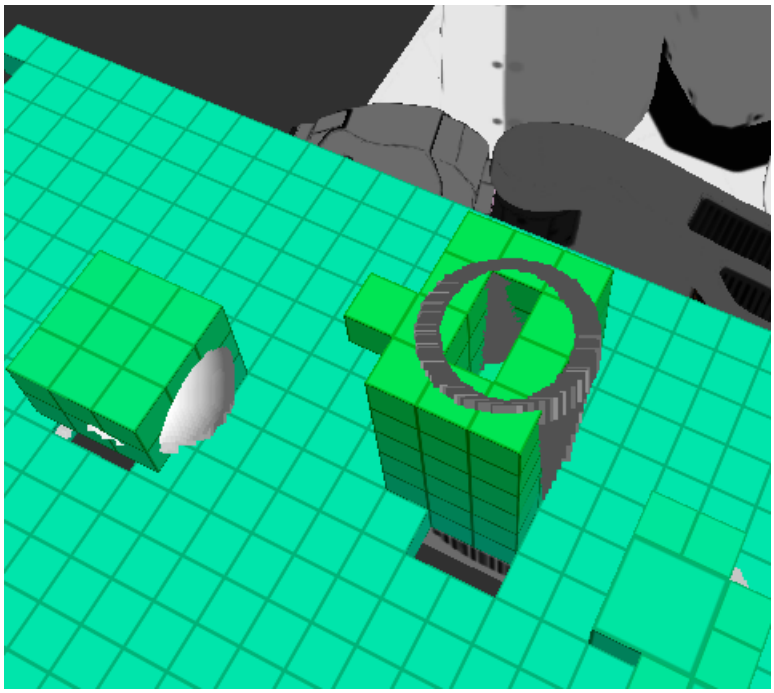


Figura 4.3: *Octomap* desalineado con nube de puntos. Verde: *Octomap*. Gris y Blanco (dentro del *Octomap*): Nube de puntos sensada por la Kinect. No deberían haber segmentos de la nube de puntos fuera del *Octomap* ni cubos del *Octomap* pertenecientes a un objeto que no estén en contacto con su nube de puntos (No debería verse la nube de puntos), sin embargo se ve que el costado derecho de ésta sale del *Octomap* que la representa.

En este caso se disminuyó la resolución del *Octomap*, con lo que disminuyó considerablemente la probabilidad de que ocurran estos problemas, sin embargo el desalineamiento del *Octomap* con los objetos no pudo solucionarse, y es un problema que aunque permite realizar agarres, disminuye la probabilidad de éxito de estos.

Padding de los objetos de colisión Al agregar un objeto de colisión a la escena se comprobó que no solo se borraban los vóxeles del *Octomap* dentro del objeto, sino que muchísimos más en un radio de 10 [cm]. Para solucionarlo se configuró el parámetro `padding_offset` en $0,02$ (original $0,1$).

[1] `https://groups.google.com/d/msg/moveit-users/T2feBu5uNos/5KshOEfqoRYJ`
[2] `https://github.com/ros-planning/moveit_ros/issues/315`

4.2. Segmentación

Primero se muestran los resultados de pruebas preliminares para validar las primeras etapas del algoritmo, y luego se realiza un ajuste de parámetros y optimización para mejorar los resultados obtenidos.

4.2.1. Resultados preliminares

Se creó un dataset de datos reales grabados con `rosbag` desde una Kinect para probar los algoritmos implementados. Se grabó 1 minuto de datos que incluyen segmentos de piso y muro, una silla usada como mesa y 3 objetos sobre ésta, dejando espacios entre los objetos.

Para reproducir los datos `rosbag` debe publicar la información almacenada además del tópico */clock*, lo que indica que se está ejecutando una simulación (esto publica el tiempo almacenado en los mensajes de forma de hacer creer al sistema que recién están siendo recibidos). El programa principal debe subscribirse a */camera/depth/points* y por tanto la función de callback recibirá los datos simulados a 30 Hz (La velocidad a la que fueron grabados originalmente).

En la implementación inicial se tienen problemas para segmentar los datos reales de la Kinect debido al ruido presente en éstos y la dificultad de la escena elegida. El algoritmo no encontraba planos en la nube, y tomaba mucho tiempo (˜15 [s]) por lo que se le comenzaron a agregar etapas de pre-procesamiento que ayudan a disminuir el tiempo y obtener una segmentación de plano. Usando la etapa de suavizado gaussiano y la de cortado se redujo el tiempo a menos de 1 segundo pero no se obtuvieron buenos resultados de segmentación usando imágenes integrales.

Para poder avanzar en la implementación se realizó la segmentación sin usar imágenes integrales, con lo que pudo segmentar el plano, a costa de un promedio de 1.5 segundos por nube. Si consideramos que falta agregar el algoritmo de detección de puntos de agarre encima de éste, el tiempo final crece bastante y por tanto esta solución sería más lenta de lo deseable.

La figura 4.4 muestra la segmentación parcial del dataset. Los bordes sin normales se deben a un alto valor de `normalSmoothingSize`. Debido al alto nivel de ruido, se debe aumentar el `distanceThreshold`, lo que provoca que la superficie de los objetos también sea considerada como parte del plano. Sin embargo el centro del plano está más abajo, por lo que igualmente se segmentará buena parte de los objetos como puntos sobre la mesa. Los espacios vacíos en medio de la nube corresponden a las partes que la Kinect no ve al ser obstruidas por los objetos. Al final se ve un plano que se sube y corresponde al respaldo de la silla.

Se revisó el código fuente de la clase `SACSegmentation` de PCL para entender mejor los parámetros y se descubrió que varios de estos no estaban siendo usados por el algoritmo debido a que no se había seleccionado el flag necesario para esto. Al corregir el error y volver a ajustar los parámetros se pudo segmentar usando imágenes integrales.

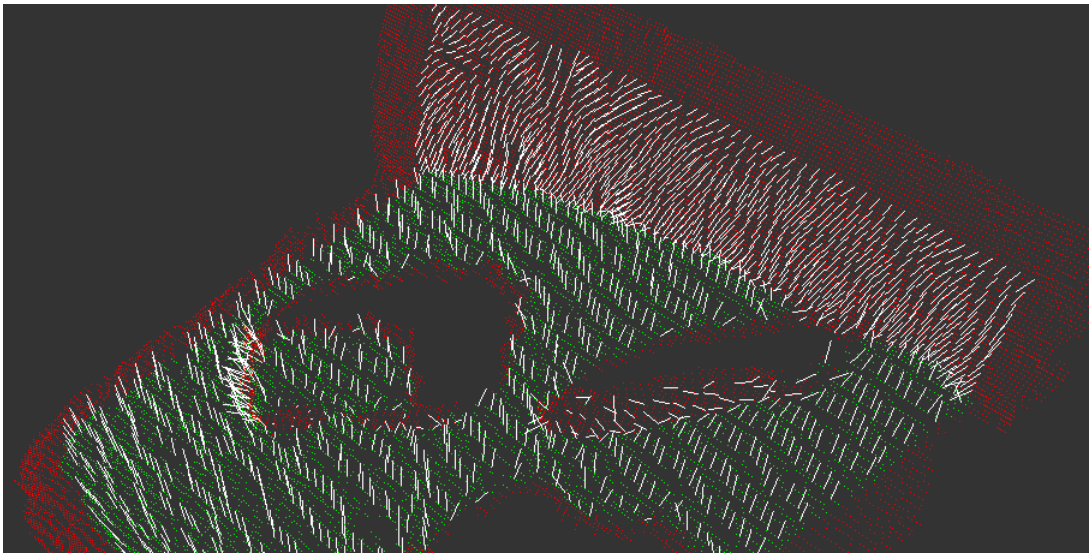


Figura 4.4: Plano segmentado en silla con 2 objetos y normales. En rojo los puntos de la nube, en verde los puntos pertenecientes al plano.

4.2.2. Ajuste de parámetros

Cada etapa del algoritmo de segmentación tiene una gran cantidad de parámetros que ajustar para tener resultados buenos. Para evitar re-compilar cada vez que se quiere probar algún cambio se implementó un servidor de `dynamic_reconfigure` que llama a una función de callback cada vez que se hace un cambio en el servidor de parámetros para este nodo. Este método permite ver en tiempo real el efecto de los parámetros en el algoritmo.

Los parámetros más relevantes ajustados son:

- Cálculo eficiente de normales

  `normalSmoothingSize:` Al aumentar el radio de suavizado las normales se aproximan al promedio, evitándose normales extrañas en las partes con textura, pero se pierde la información de los bordes y las zonas cercanas a puntos NaN, donde no se puede calcular el suavizado por existir NaN y se omiten esas normales.
- Ajuste de plano: Modificar los parámetros `sampleMaxDistance` y `probability` solo empeoró el desempeño y aumentó el tiempo de cálculo.

  `normalDistanceWeight:` Se le da mucha importancia a las normales en el ajuste del plano puesto que es importante que el plano encontrado sea una superficie horizontal y no una pared u otros planos distractores

  `epsAngle:` Al relajar esta restricción a 20°, se hace más probable encontrar planos en situaciones complejas y aún permite encontrar una superficie que contenga objetos
- Clustering

  `minClusterSize:` Valores demasiado pequeños del tamaño mínimo hacen aparecer clusters que solo contienen ruido o imperfecciones, y demasiado grandes borran pequeños objetos manipulables

  `maxClusterSize:` Valores demasiado pequeños hacen desaparecer objetos potencialmente manipulables a pesar de su tamaño, puesto que no dice nada de su forma

o peso; además en ciertos casos se da que un mismo objeto se divide en varios clusters debido a que no se permite que estos crezcan más.

Para entender mejor el algoritmo se realizó un profiling de tiempo de ejecución de cada etapa por separado, donde quedó en evidencia que las etapas que más influían en el tiempo eran el ajuste del plano y el clustering, usando un 90 % del tiempo de procesamiento. El clustering es el de mayor variabilidad al aumentar el número de puntos a procesar. Esto llevó a buscar formas de optimizar estas etapas:

4.2.2.1. Optimización del clustering

Se descubrió que aunque los tutoriales construyen un `KDTree` para realizar la búsqueda en el clustering, se podía optimizar construyendo un `OrganizedNeighbor` que hace búsquedas eficientes en nubes organizadas, con esto se bajó un orden de magnitud el tiempo del clustering. Además, leyendo el código fuente de la clase `EuclideanClusteringExtraction`, se descubrió que se ignora el objeto de búsqueda provisto externamente y construye uno nuevo internamente dependiendo del tipo de nube provisto, por lo que al dejar de construir el objeto externamente se redujo a la mitad el tiempo de cálculo.

4.2.2.2. Optimización del ajuste del plano

Dejando `epsAngle` y `distanceThreshold` fijos y moviendo el número de iteraciones, `normalDistanceWeight`, se consiguió bajar el tiempo de cálculo sin modificar el resultado. Probablemente un análisis más detallado podría hacer que baje aún más.

4.2.2.3. Resultados del ajuste realizado

Con el nuevo ajuste e implementación se eliminó la etapa de suavizado porque empeoraba los resultados y era muy lento el cálculo. La etapa de cortado se dejó pero no es necesario activarla puesto que ahora la segmentación es mucho más rápida y considera la nube completa. Los parámetros finales usados se pueden ver en los anexos A.2. Un ejemplo de segmentación se muestra en la figura 4.5. El tiempo final de segmentación está en la Tabla 4.1.

Tabla 4.1: Tiempo de procesamiento de la segmentación

Etapa	Tiempo promedio [ms]
Cortado de nube (si está activado)	30
Normales usando imágenes integrales	70
Ajuste del plano	200
Envoltura convexa del plano	15
Clustering	300
Total segmentación	615

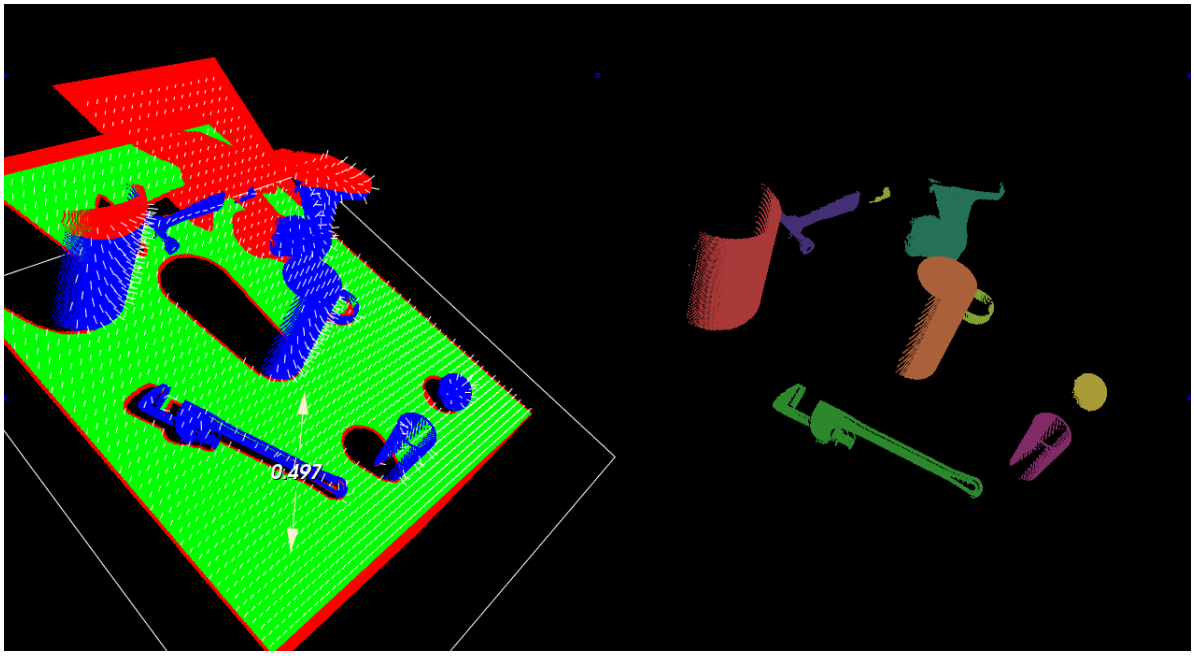


Figura 4.5: Resultado segmentación. A la izquierda: Nube original en rojo, puntos pertenecientes al plano en verde, plano encontrado como un cuadrado blanco al rededor de los puntos, puntos sobre el plano en azul. A la derecha clusters encontrados. Los objetos que aparecen en los clusters son (de izquierda a derecha): Lata gigante, martillo (arriba, tiene la base del mango como otro cluster en amarillo), llave Stillson, lata normal (al medio), taladro (arriba en calipso; está cortada su parte superior), cilindro (amarillo), vaso y pelota.

4.2.2.4. Factores que hacen fallar a la segmentación

Algunas de las posibles fuentes de error en la segmentación son:

- Presencia de objetos cóncavos, donde el objeto se obstruye a sí mismo, generando una discontinuidad que supera el máximo aceptado y por tanto obteniéndose más de un cluster por objeto.
- Cuando un objeto supera la altura máxima en la etapa de filtrado sobre la mesa; se obtendrá un cluster que contenga al objeto hasta su altura máxima, ignorando el resto del objeto que colinda con el cluster (ver figura 4.5).
- Presencia de superficies cóncavas. Si hay una mesa con concavidades, alguna silla u otro elemento que entre en la envoltura convexa de la mesa será considerado como objeto sobre la mesa.
- Obstrucciones entre objetos. En la figura 4.5 se puede apreciar como el taladro obstruye al martillo, lo que provoca que partes de su mango se separen en la nube por una distancia mayor a la aceptada en la etapa de clustering y por tanto sean detectados como objetos diferentes.

4.2.2.5. Posibles mejoras

El sistema implementado desaprovecha toda la información de color en la escena, con lo que se podría mejorar la segmentación para determinar nubes alejadas que corresponden al

mismo objeto. Para los objetos segmentados de forma incompleta (parte superior excluida), haciendo un chequeo por componentes conexas en la nube de puntos se puede evitar agregar clusters de objetos incompletos, puesto que incluso si se llegase a manipular la parte visible, no se pueden calcular colisiones para lo restante. Últimamente ha aparecido un segmentador multi-planos en PCL que podría suplir estas falencias y por tanto sería interesante de integrar.

Uno de los casos más comunes de fallas en la segmentación es la obstrucción causada por otros objetos. Además de lo ya planteado, se podría solucionar este problema integrando información semántica sobre la escena que permita determinar que corresponden al mismo objeto, o resolver el problema de *registration* de varias nubes de puntos mientras el robot se acerca a la escena, para contar con más puntos de vista que completen la información faltante, como en el martillo de la figura 4.5. Para esto también pueden aprovecharse las cámaras en los brazos del PR2, de forma de agregar información a medida que estos se mueven por la escena.

4.3. Detección

Se exponen los resultados de las etapas de cálculo de Bounding box, muestreo de puntos de agarre, filtrado y puntuación de estos, para finalmente comparar los tiempos de ejecución de las etapas.

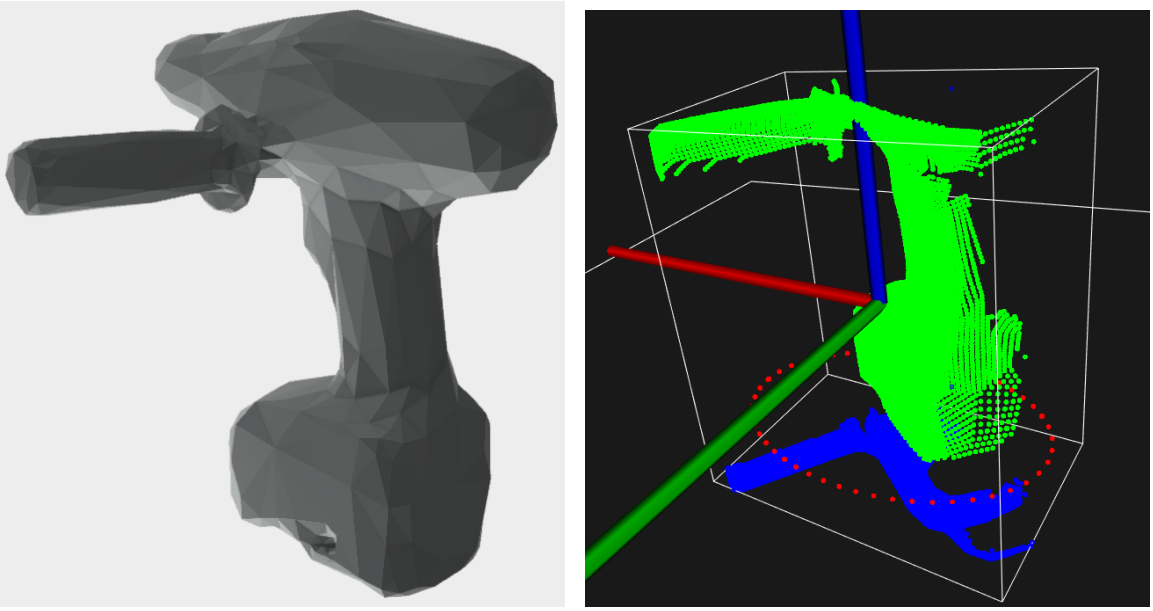
4.3.1. Bounding box

Se implementó un visualizador de PCL para poder testear el funcionamiento del cálculo de Bounding box, las traslaciones y rotaciones de coordenadas, usando como base la nube de puntos original de la Kinect.

En la figura 4.6b se ve como el objeto original está encerrado por su Bounding box y se dibujan los ejes de su propio sistema de coordenadas encima. El Bounding box es más alto que la nube de puntos debido a que va desde la mesa al punto más alto del cluster, y al extraer los puntos sobre la mesa se define una distancia mínima de extracción de forma de no tomar fragmentos de la propia mesa.

Esta representación como Bounding box del objeto es útil para objetos relativamente simétricos y uniformes, pero falla completamente en otros casos. En la figura 4.6b se ve como el Bounding box no está alineado con las partes simétricas del taladro, donde se esperaría encontrar agarres factibles.

Una posibilidad para mejorar la representación de objetos no simétricos es definir Bounding boxes por segmentos dentro del objeto. El cálculo del Bounding box es eficiente y mejora aún más su desempeño al disminuir el tamaño de la nube a usar, debido a que la parte más costosa tiene que ver con el cálculo de vectores propios usando la matriz de covarianza. Al tener un objeto definido como un conjunto de Bounding boxes que representan las componentes simétricas del mismo, se pueden definir mejores puntos de agarre sobre éstas. Para que



(a) Modelo original del taladro. La parte superior fue cortada en la segmentación.

(b) Bounding box para objeto asimétrico en coordenadas de Kinect (verde) con agarres laterales (rojo). Proyección sobre la mesa del objeto en azul.

Figura 4.6: Cálculo de Bounding box sobre un taladro

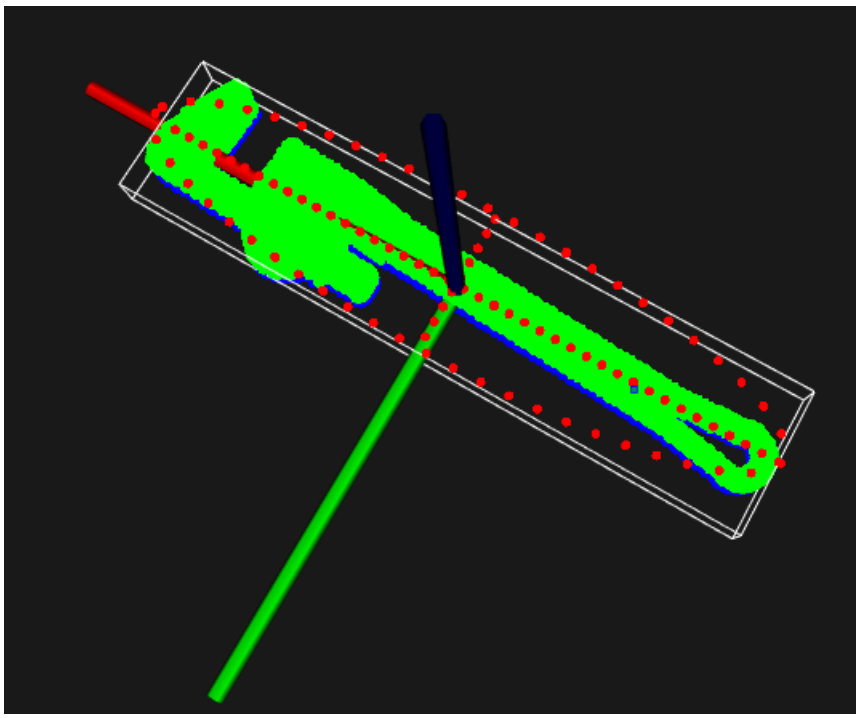


Figura 4.7: Bounding box para objeto relativamente simétrico con agarres laterales y superiores.

esta idea tenga éxito es necesario definir un algoritmo eficiente de cálculo de componentes simétricas a partir de una nube de puntos.

4.3.2. Muestreo de puntos de agarres

En la figura 4.7 se ven los puntos rojos representando agarres muestreados. Una elipse alrededor del objeto para los laterales y en los ejes principales para los superiores. Estos también pueden verse en `rviz`, donde los cubos rojos en la figura 4.8 marcan los puntos de agarre muestreados.

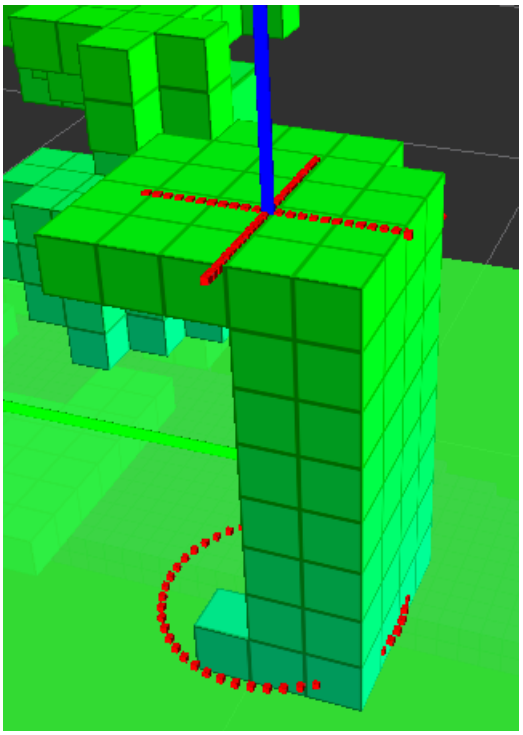


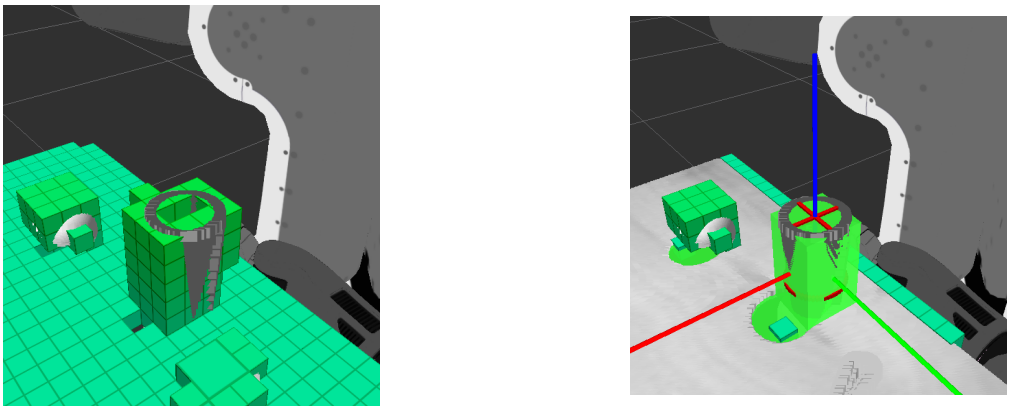
Figura 4.8: Puntos de agarre sobre *Octomap* de un objeto.

Los agarres superiores debieron ser modificados de su especificación inicial debido a la representación del objeto de colisión como una caja. Como no se tiene información de la forma específica del objeto desde el punto de vista de colisiones, al muestrear agarres superiores paralelos a Z en un punto de los ejes principales, se puede topar con superficies curvas que harían resbalar el agarre. Por esta razón y como solución temporal se cambió la orientación del agarre para que X apunte al centro del Bounding box, mientras que la posición se mantiene en los ejes principales. Esto limita el número de objetos a manipular y por tanto es importante que se solucione en un trabajo futuro.

4.3.3. Filtrado y puntuación de puntos de agarre

Los objetos de colisión se representan como cajas como se muestra en la figura 4.9. Esta representación aunque útil, es muy inexacta y no permite calcular adecuadamente colisiones

para objetos que tienen formas menos uniformes.



(a) *Octomap* representando el objeto

(b) Modelo de colisión del objeto

Figura 4.9: Un objeto y su modelo de colisión

Una forma de mejorar las limitaciones que provoca el uso de cajas como objetos de colisión es definirlos como una malla, para esto es necesario usar la nube de puntos del objeto y calcular los triángulos que encierran al objeto usando estos puntos. Con esta información se puede construir un mensaje `shape_msgs::Mesh` y usarlo para construir un nuevo objeto de colisión mucho más preciso.

Luego de buscar un plan para cada punto de agarre, se muestran solo los factibles como en la figura 4.10. Se ve que se excluyen algunos puntos que podrían ser factibles, debido a que se dio poco tiempo de planificación de trayectorias, por lo tanto si no encontró una trayectoria en los primeros intentos, simplemente se marca como infactible.

Debido a las características del robot y de las escenas objetivo, muchos puntos de agarre laterales se filtran, puesto que entran en colisión con otros objetos, con el robot mismo, con la mesa, o están fuera del alcance del brazo ejecutor. Para mejorar esto, se podría ajustar el ángulo del punto de agarre para que en vez de ser paralelo a la mesa, tenga cierta inclinación de forma que se pueda acceder desde arriba.

4.3.3.1. Puntuación de agarres

La puntuación de los agarres se realiza tal cual se describe en [68], a excepción del criterio de ancho del objeto en la zona del agarre. Si bien esta puntuación presenta buenos resultados en el sentido que los agarres más cercanos al brazo y aquellos con ángulos menos extraños reciben mejores puntajes, existe mucho espacio para mejorar el sistema de puntuación, considerando otros criterios como el centro de masa y el material del objeto que se quiere tomar.

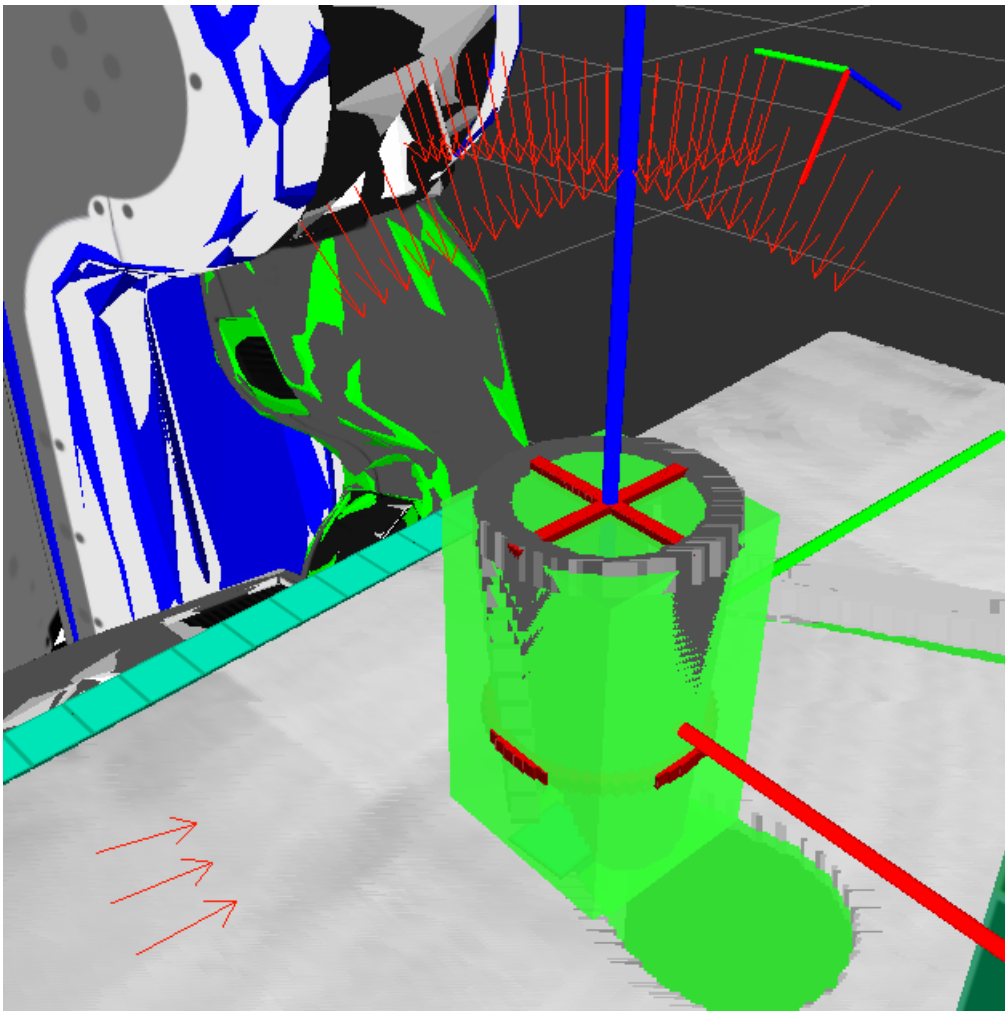


Figura 4.10: Puntos de agarre factibles representados por flechas rojas.

4.3.4. Tiempo de ejecución

Se analiza el tiempo por etapas (Tabla 4.2), y es evidente que el filtrado es la etapa más costosa del algoritmo, con un tiempo promedio de 4,3 [s], sin embargo existe espacio para optimización de la misma. Se hace evidente que con la implementación actual no se puede ejecutar el algoritmo en tiempo real, y debido a los cálculos necesarios para detectar colisiones, es la parte más desafiante para optimizar.

Tabla 4.2: Tiempo de procesamiento de la detección

Etapa	Tiempo promedio [ms]
Muestreo de agarre (incl. Bounding box)	15
Filtrado de agarres	4300
Puntuación de agarres	10
Total detección	4325

Se plantean las siguientes optimizaciones que a futuro podrían acelerar el proceso:

- En este momento se hacen todas las solicitudes de planificación de trayectorias por separado, lo que representa una gran sobrecarga dado que deben pasar por la máquina de estados del cliente `actionlib`, transmitirse por red por el sistema de tópicos de ROS y luego pasar por la máquina de estados del servidor de `actionlib`. Una forma eficiente de hacer esta consulta sería mandar todas las solicitudes en un paquete, aunque esta solución requeriría extensiones al servidor existente.
- Limitar el área de trabajo dentro del alcance del robot. Con esto se pueden descartar inmediatamente puntos fuera de este espacio.
- El algoritmo base estipula descartar agarres infactibles usando restricciones geométricas de la escena, de forma de solo chequear colisiones en volúmenes específicos, que corresponden a un cono en la posición del agarre y otro desde el hombro.
- Se pueden aprovechar las restricciones geométricas del brazo para descartar puntos que requieran ángulos imposibles. Por ejemplo si un punto está del lado contrario del brazo que está ejecutando, se puede verificar que su posición de pre-agarre sea alcanzable haciendo back-tracking eficientemente de la posición necesaria.

4.4. Ejecución

A continuación se detallan los resultados derivados de la ejecución del algoritmo, tales como los objetos usados y comportamiento en diversos casos.

4.4.1. Objetos de prueba

Se detallan los objetos utilizados en las pruebas y el resultado esperado de ellos:

- Vaso: manipulable con agarres superiores y laterales
- Pelota: manipulable con agarres superiores y laterales
- Cilindro: manipulable con agarres superiores
- Taladro: no manipulable. El objeto no es simétrico y sobrepasa las dimensiones máximas.
- Martillo: manipulable con agarres superiores
- Llave mecánica: no manipulable (muy bajo)
- Lata de cerveza: no manipulable (no cabe en el gripper por un par de mm)
- Lata de bebida gigante: no manipulable. El objeto es más grande que el ancho máximo del gripper.

La detección de objetos no manipulables demuestra la capacidad del sistema de evitar intentos de manipulación que van a fracasar con seguridad, debido a que no cumplen las restricciones impuestas en un comienzo.

4.4.2. Resultados de los agarres

El algoritmo filtra de forma efectiva las colisiones con otros objetos y no intenta manipulación de los objetos clasificados como no manipulables. Para aquellos que solo admiten un tipo de agarre, solo se obtienen muestras para el agarre admitido, demostrando la capacidad de filtrar agarres imposibles.

Falta un chequeo de ancho del agarre. En ocasiones el algoritmo intenta tomar el martillo con un agarre muestreado por su eje corto, lo que es imposible dado que el martillo es muy largo. Este tipo de comportamiento se produce debido a la representación de las colisiones como cajas y se espera que desaparezca al mejorar la representación.

Al ejecutar los puntos de agarre calculados, lo primero que se hace evidente es la dificultad que tienen los agarres superiores al tratar con objetos de geometría cilíndrica en el plano X-Y, como son pelotas, vasos, etc. Solo el punto de agarre superior correspondiente al centro del Bounding box tiene probabilidades de éxito, pero sigue siendo un agarre inestable, dado que al menor resbalamiento o error en el grasping se aumenta la diferencia angular entre la dirección del agarre y la normal de la superficie en el punto de contacto. Para sobrellevar este problema se propone determinar específicamente si se está en uno de esos casos (usando RANSAC, por ejemplo) para evitar muestrear agarres superiores.

La figura 4.11 muestra la ejecución de un agarre sobre el objeto vaso. Se presentan agarres factibles laterales y superiores y se elije uno lateral como era de esperar, dado que está más cerca del hombro y es el de menor ángulo contra el hombro. También se grabó un video[1] con el proceso de retirada de la mesa planificado.

El algoritmo tuvo problemas ejecutando agarres sobre la pelota, debido a que al menor toque esta rodaba. Los errores se deben al desalineamiento del *Octomap* con la realidad y

[1] `https://drive.google.com/open?id=0B4eddLuGahfNcXVJR2oyNGhVY00`

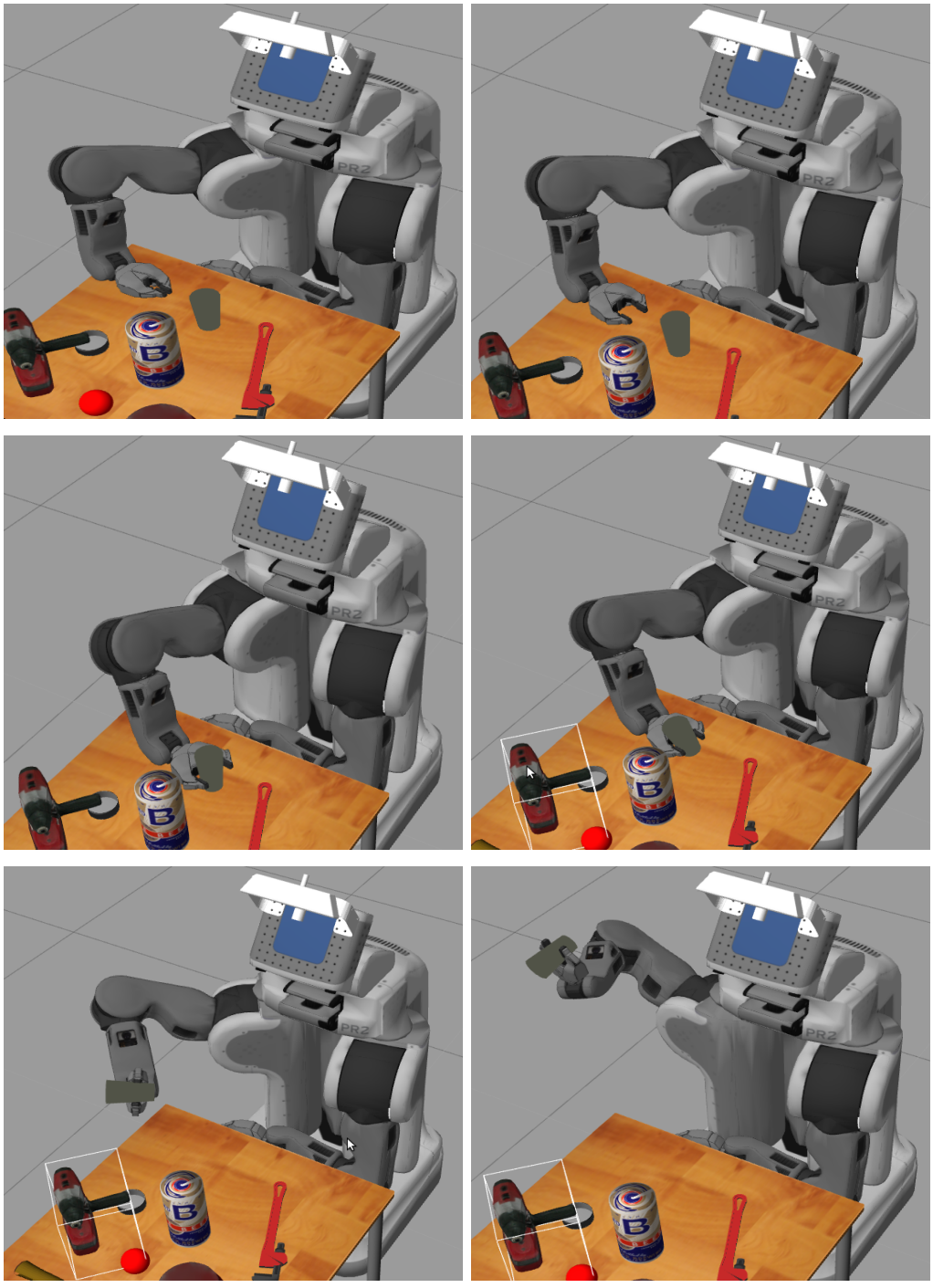


Figura 4.11: Agarre de un vaso

a la necesidad de un ajuste de la profundidad del agarre. Es necesario que este parámetro dependa de la superficie real del objeto y no solo del Bounding box como sucede ahora.

4.5. Resultados en el robot real

Se hicieron algunas pruebas en el robot PR2 real para determinar la validez del algoritmos fuera de ambientes simulados. La Figura 4.12 muestra el escenario de pruebas, que corresponde a una mesa rectangular con 2 cubos y un paralelepípedo. La mesa presenta en sus bordes una protuberancia que sirve para poner a prueba el algoritmo.

(a) Vista lateral. **(b)** Vista desde la perspectiva del robot.

Figura 4.12: Escena de pruebas con el robot PR2 real. Una mesa con 3 objetos.

4.5.1. Segmentación

La segmentación se presenta más desafiante que lo experimentado en simulación debido al ruido del sensor, la existencia de varios objetos en el fondo y las variaciones de iluminación dadas por el movimiento de personas en la habitación, sin embargo gracias a las restricciones impuestas, el algoritmo pudo encontrar exitosamente el plano de interés usando sus normales.

Fue necesario aumentar el parámetro `minHeight` de la envoltura convexa sobre la mesa, debido a que de lo contrario se consideraban los bordes de la mesa como objetos irregulares; y en algunos casos cuando el gripper izquierdo del robot yacía a una altura similar a la de la mesa, se segmentaba el gripper como parte de esta con un objeto encima, lo que es fácil de solucionar posicionando los brazos fuera del campo visual.

A pesar de poder segmentar correctamente, hay una perdida de robustez, pues se comprobó que al haber puntos en la nube a la misma altura de la mesa pero que corresponden a objetos del fondo, estos son segmentados como parte de la mesa. Esto no es un problema para las etapas siguientes dado que esa zona de la mesa que no existe esta fuera del alcance del robot y no se usa, pero esos puntos alteran el plano encontrado agregándoles cierta inclinación indeseable, lo que si puede afectar los objetos que se encuentran sobre la mesa. Es necesario

agregar un filtro de continuidad para los planos encontrados en RANSAC para solucionar este problema.

El otro problema es que el ángulo de incidencia de la Kinect sobre una superficie y el ángulo de la Kinect respecto del robot influyen en el cálculo de profundidad, haciendo difícil (incluso para un ser humano) determinar si la nube de puntos es un plano o no. El primer caso tiene que ver con fenómenos de la luz y la cantidad de luz reflejada por la superficie; el segundo caso tiene que ver con errores del calculo de las transformaciones debido al juego de los motores que sostienen la Kinect al estar inclinada. Estas situaciones en algunos casos impiden segmentar adecuadamente superficies planas, sobre todo si estas reflejan medianamente la luz, como el caso de una madera cubierta con melamina.

4.5.2. Detección

En ocasiones el algoritmo invierte los ejes del Bounding box del objeto de interés, posiblemente debido al ruido encontrado en el cluster que lo representa, por lo que es necesario agregar una restricción más fuerte a los vectores propios que asegure su orientación.

Fue necesario aumentar la altura de muestreo de los agarres laterales, debido a que las protuberancias del borde de la mesa entraban en colisión con el brazo, eliminando todos los agarres laterales.

El Bounding box de la mesa se ve fuertemente afectado por el ruido pues en ocasiones no representa la forma de la misma, sin embargo sigue cumpliendo su propósito, pues mantiene alejado el brazo de trayectorias que pasen por debajo de ésta (una zona que no se puede sensar), y siempre hay una cara de esta en contacto con el objeto de interés.

4.5.3. Ejecución en el robot PR2

Con las modificaciones señaladas fue posible ejecutar agarres sobre paralelepípedos y una botella, sin embargo el brazo no levanta el objeto debido a que encuentra una colisión entre el objeto de colisión que está en la mesa y aquel fijado al brazo del robot (ambos son el mismo objeto). Esto puede deberse a que el mensaje que elimina el objeto de colisión del mundo se pierde, provocando que este nunca sea eliminado, y por tanto entrando en colisión con el objeto de colisión fijado al brazo del robot en cuanto este es agregado a la escena. Este error se produce en la etapa de ejecución solamente, debido a que la trayectoria planeada no presenta colisiones, por lo que no se descarta que esté asociado a un error en la sincronización del `actionlib` que controla el movimiento del brazo con la escena de planificación. Se implementó una solución temporal a este problema: luego de agarrado el objeto, se desactiva el chequeo de colisiones con el *Octomap*, se mueve el brazo a la posición inicial y finalmente se reactiva el chequeo de colisiones.

Para el caso del agarre de la botella, es necesario ajustar el parámetro de esfuerzo del agarre (en la especificación del agarre entregada a MoveIt!) para evitar que el robot apriete

demasiado fuerte. Las pruebas realizadas mostraron que este parámetro tiene un comportamiento poco previsible, dado que al variar su valor no se apreciaron cambios importantes en la fuerza ejercida en el agarre. Así mismo el valor de movimiento del gripper para la posición con el objeto tomado presenta un comportamiento no lineal, dado que asignando el valor 0 a la articulación, cierra el gripper, el valor 1 lo abre y los valores intermedios lo abren una cantidad mucho menor a la proporcional entre ambos extremos, con un salto a completamente abierto cerca de 0,9.

Un ejemplo de agarre se puede ver en la Figura 4.13 y en la dirección `https://goo.gl/photos/HqeRmgAsDanz5CeZA` hay videos de otras pruebas realizadas tanto exitosas como fallidas (en cuyo caso se desactivan los motores para evitar daños al robot). En el caso de las pruebas fallidas, el agarre muestreado es correcto pero hay un corrimiento en la posición y se presume que se debe al desalineamiento del *Octomap* con la nube de puntos explicado antes.

Figura 4.13: Robot PR2 Agarrando un paralelepípedo.

Conclusión

Se implementó un algoritmo de manipulación robótica de objetos en C++ para ser usado con un robot PR2. El algoritmo cuenta con diversas etapas: segmentación de la escena, muestreo de puntos de agarre, filtrado de puntos infactibles y puntuación de agarres.

En la primera parte de este trabajo se expusieron las necesidades, dificultades, desafíos y etapas de la manipulación robótica de objetos; se detallaron principales conceptos a usar, así como la plataforma de desarrollo, librerías y el robot PR2; se realizó una extensa revisión del estado del arte de la planificación de trayectorias y la detección de puntos de agarre; y se detalló el funcionamiento de un algoritmo elegido como base para este trabajo.

Para la planificación de trayectorias se revisaron los algoritmos presentes en 2 de las principales librerías del tema, SBPL y OMPL. Estos algoritmos se basan en búsqueda usando primitivas y muestras aleatorias respectivamente; y tienen aplicaciones en ambientes estáticos, dinámicos, y con incertezas.

Para la detección de puntos de agarre se revisaron 35 algoritmos entre los más significativos de los últimos 10 años, con énfasis en los más recientes, y se clasificaron según los tipos de objetos que pueden manipular, tiempo de cálculo necesario, principal técnica usada, tasa de éxito obtenida y una recomendación personal sobre el tipo de problemas a los que aplicarlo.

Las principales librerías y plataformas utilizadas son ROS como plataforma de interfaz con el robot, PCL como librería de nubes de puntos y MoveIt! como librería de planificación de trayectorias.

La segmentación tiene las restricciones asociadas a una cámara Kinect para su funcionamiento: escenas interiores y materiales no reflectantes. Además de las restricciones del sensor también se requiere que los objetos tengan cierta separación mínima entre ellos. Para una amplia gama de situaciones que cumplen estas características es posible segmentar los objetos disponibles sobre la mesa de forma eficiente. La implementación realizada comprueba que con este algoritmo es posible realizar una segmentación robusta de la escena bajo las condiciones supuestas y considerando que por la naturaleza del tamaño de los objetos que se pueden manipular con una sola mano y la distancia requerida entre ellos se pueden esperar bajas obstrucciones.

La etapa de filtrado por colisiones que requiere de más tiempo del deseado y presenta la inconveniencia de que se descartan muchos agarres en la práctica factibles, debido a que el modelo de colisión de los objetos es una caja. Tanto para esta etapa como para las de

muestreo y puntuación de agarres se plantea una serie de posibles mejoras, como trabajos futuros, que ayudarían a ampliar los tipos de objetos manipulables y mejorar la eficiencia del algoritmo.

Las principales fallas del método están dadas por objetos que no cumplen los requisitos de simetría y tamaño. La falta de un método robusto para detectar dichos objetos automáticamente es la principal causa de fracaso en los agarres intentados.

Dentro de los objetos que cumplen las restricciones impuestas, aquellos cilíndricos representan un desafío para los agarres superiores y denotan la necesidad de tener puntos de agarre adaptables a estas situaciones.

Los tiempos reportados para este algoritmo están basados en su ejecución sin optimizaciones del compilador, por lo que se espera que activándolas se mejore su desempeño al pasar a un esquema de producción.

Se demostró experimentalmente que el algoritmo se puede usar tanto en ambientes simulados como en un robot PR2 real, previo ajuste de algunos parámetros.

Trabajos futuros

El principal trabajo futuro que se desprende de esta memoria es la validación estadística de la efectividad del método, probando con un amplio conjunto de objetos, así como realizar más pruebas en el robot PR2 real. Un conjunto de objetos tentativos a probar y su resultado esperado es:

- Taza plástica: manipulable con agarres superiores y laterales
- Caja de cereales: manipulable con agarres superiores y laterales
- Billetera: manipulable con agarres superiores
- Libro de pie: manipulable con agarres superiores
- Plátano: manipulable con agarres superiores
- Manzana: manipulable con agarres superiores y laterales

Además algunos de los otros trabajos futuros relevantes son:

- Generar un método automático de detección de los objetos que no cumplen con las propiedades de simetría y tamaño
- Generar un detector que determine cuando un objeto presenta geometría cilíndrica para evitar sus agarres superiores, o modificar la generación de puntos de agarre para que se adapten a ciertos tipos de objeto, por ejemplo cambiando el punto de agarre desde la punta de las pinzas a la zona central, para objetos redondos.
- Implementar Bounding boxes por sub-segmentos simétricos en objetos no simétricos.
- Incluir propiedades del punto de contacto en la puntuación de agarres.
- Agregar información de color y de componentes conexas a la segmentación.

- Agregar etapa de estimación de ruido del sensor, para ajustar los parámetros de la segmentación automáticamente frente a condiciones cambiantes.
- Reducir los tiempos más relevantes como:
  - Filtrado de agarres por colisiones usando solicitudes en bloque
  - Filtrado de agarres implementando restricciones geométricas para los puntos.
  - Ejecución del agarre: se puede implementar cálculo de puntos de agarre en paralelo del movimiento del brazo a una posición cercana al objeto de interés. Esto podría dar un movimiento fluido del robot en los mejores casos.

Notas finales sobre la plataforma

En el trabajo realizado se han usado varios de los módulos de ROS (`actionlib`, controladores, arquitectura de mensajes, *tf*, `dynamic_reconfigure`), y varias librerías y software importantes comunmente usadas en conjunto (MoveIt!, PCL, Gazebo, Eigen), y se ha lidiado con numerosas falencias y errores en varios de ellos. Si bien ROS por si solo funciona bastante bien para problemas puntuales y especificos tipo demostración, al encadenar comportamientos y procesamientos más complejos que usen elementos de múltiples librerías se exponen rápidamente los errores en las librerías usadas volviendo inestable el sistema. Si bien no se puede exigir un software libre de errores, la arquitectura del sistema debiera asegurar un manejo apropiado de estos de forma recuperable, así como un mayor control sobre el estado del robot y el estado de los nodos en ejecución. La arquitectura actual de ROS hace difícil depurar errores en multiples nodos o librerías y abundan las perdidas de mensajes por la naturaleza de los protocolos de comunicación usados, sin una forma de forzar una comunicación confiable.

La complejidad de un robot como el PR2 hace que muchas cosas puedan fallar desde el punto de vista de hardware, y si se agrega una plataforma de software compleja como ROS, se presentan en ocaciones comportamientos azarosos y errores no recuperables que obligan a un reinicio de todo el sistema.

En mi opinion hace falta mayor cohesión al sistema para que los diferentes módulos trabajen mejor juntos, esto sin perder la naturaleza modular del diseño, y faltan aún años de trabajo y perfeccionamiento para que la arquitectura sea más confiable y facil de usar. Es difícil construir soluciones de alto nivel sobre hardware de alta complejidad si se tiene una base de software inestable.

Bibliografía

[1] RajeshKanna Ala y col. "A 3D-grasp synthesis algorithm to grasp unknown objects based on graspable boundary and convex segments". En: *Information Sciences* 295 (2015), págs. 91-106.

[2] Laura Antanas y col. "High-level Reasoning and Low-level Learning for Grasping: A Probabilistic Logic Pipeline". En: *arXiv preprint arXiv:1411.1108* (2014).

[3] Lukas Barinka y Roman Berka. "Inverse kinematics-basic methods". En: *Web.<http://www.cescg.org/CESCG-2002/LBarinka/paper.pdf>* (2002).

[4] Jeannette Bohg y Danica Kragic. "Learning grasping points with shape context". En: *Robotics and Autonomous Systems* 58.4 (2010), págs. 362-377.

[5] Robert Bohlin y Lydia E Kavraki. "A Randomized Approach to Robot Path Planning Based on Lazy Evaluation". En: *COMBINATORIAL OPTIMIZATION-DORDRECHT-* 9.1 (2001), págs. 221-249.

[6] Robert Bohlin y Lydia E Kavraki. "Path planning using lazy PRM". En: *Robotics and Automation, 2000. Proceedings. ICRA'00. IEEE International Conference on*. Vol. 1. IEEE. 2000, págs. 521-528.

[7] Abdeslam Boularias, James Andrew Bagnell y Anthony Stentz. "Learning to Manipulate Unknown Objects in Clutter by Reinforcement". En: *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence, January 25-30, 2015, Austin, Texas, USA*. 2015, págs. 1336-1342. URL: http://www.aaai.org/ocs/index.php/AAAI/AAAI15/paper/view/9360.

[8] Abdeslam Boularias, Oliver Kroemer y Jan Peters. "Learning robot grasping from 3-d images with markov random fields". En: *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE. 2011, págs. 1548-1553.

[9] John Canny. "A computational approach to edge detection". En: *Pattern Analysis and Machine Intelligence, IEEE Transactions on* 6 (1986), págs. 679-698.

[10] Benjamin J Cohen y col. "Planning for manipulation with adaptive motion primitives". En: *IEEE International Conference on Robotics and Automation (ICRA)*. IEEE. 2011, págs. 5478-5485.

[11] Anthony Cowley y col. "Perception and motion planning for pick-and-place of dynamic objects". En: *Intelligent Robots and Systems (IROS), 2013 IEEE/RSJ International Conference on*. IEEE. 2013, págs. 816-823.

[12] Renaud Detry y col. "Learning grasp affordance densities". En: *Paladyn, Journal of Behavioral Robotics* 2.1 (2011), págs. 1-17.

[13] Andrew Dobson y Kostas E Bekris. "Improving sparse roadmap spanners". En: *Robotics and Automation (ICRA), 2013 IEEE International Conference on*. IEEE. 2013, págs. 4106-4111.

[14] Andrew Dobson, Athanasios Krontiris y Kostas E Bekris. "Sparse roadmap spanners". En: *Algorithmic Foundations of Robotics X*. Springer, 2013, págs. 279-296.

[15] Richard O Duda y Peter E Hart. "Use of the Hough transformation to detect lines and curves in pictures". En: *Communications of the ACM* 15.1 (1972), págs. 11-15.

[16] Martin A Fischler y Robert C Bolles. "Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography". En: *Communications of the ACM* 24.6 (1981), págs. 381-395.

[17] Bryant Gipson, Maciej Moll y Lydia E Kavraki. "Resolution independent density estimation for motion planning in high-dimensional spaces". En: *Robotics and Automation (ICRA), 2013 IEEE International Conference on*. IEEE. 2013, págs. 2437-2443.

[18] Jared Glover, Daniela Rus y Nicholas Roy. "Probabilistic models of object geometry for grasp planning". En: *The International Journal of Robotics Research* 28.8 (ago. de 2009), págs. 999-1019.

[19] Corey Goldfeder y col. "Grasp planning via decomposition trees". En: *Robotics and Automation, 2007 IEEE International Conference on*. IEEE. 2007, págs. 4679-4684.

[20] Peter E Hart, Nils J Nilsson y Bertram Raphael. "A formal basis for the heuristic determination of minimum cost paths". En: *IEEE Transactions on Systems Science and Cybernetics* 4.2 (1968), págs. 100-107.

[21] Alexander Herzog y col. "Learning of grasp selection based on shape-templates". En: *Autonomous Robots* 36.1-2 (2014), págs. 51-65.

[22] Todd Hester y Peter Stone. "TEXPLORE: real-time sample-efficient reinforcement learning for robots". En: *Machine Learning* 90.3 (2013), págs. 385-429.

[23] David Hsu, Jean-Claude Latombe y Rajeev Motwani. "Path planning in expansive configuration spaces". En: *Robotics and Automation, 1997. Proceedings., 1997 IEEE International Conference on*. Vol. 3. IEEE. 1997, págs. 2719-2726.

[24] Bo Huang y col. "Learning a real time grasping strategy". En: *IEEE International Conference on Robotics and Automation (ICRA)*. IEEE. 2013, págs. 593-600.

[25] Léonard Jaillet, Juan Cortés y Thierry Siméon. "Sampling-based path planning on configuration-space costmaps". En: *Robotics, IEEE Transactions on* 26.4 (2010), págs. 635-646.

[26] Sertac Karaman y Emilio Frazzoli. "Sampling-based algorithms for optimal motion planning". En: *The International Journal of Robotics Research* 30.7 (2011), págs. 846-894.

[27] Lydia E Kavraki y col. "Probabilistic roadmaps for path planning in high-dimensional configuration spaces". En: *IEEE Transactions on Robotics and Automation* 12.4 (1996), págs. 566-580.

[28] Ben Kehoe y col. "Cloud-based robot grasping with the google object recognition engine". En: *IEEE International Conference on Robotics and Automation (ICRA)*. IEEE. 2013, págs. 4263-4270.

[29] Pradeep Khosla y Richard Volpe. "Superquadric artificial potentials for obstacle avoidance and approach". En: *Proceedings., 1988 IEEE International Conference on Robotics and Automation, 1988*. IEEE. 1988, págs. 1778-1784.

[30] Jin-Oh Kim y Pradeep K Khosla. "Real-time obstacle avoidance using harmonic potential functions". En: *Robotics and Automation, IEEE Transactions on* 8.3 (1992), págs. 338-349.

[31] Gert Kootstra y col. "Enabling Grasping of Unknown Objects Through a Synergistic Use of Edge and Surface Information". En: *Int. J. Rob. Res.* 31.10 (sep. de 2012), págs. 1190-1213. ISSN: 0278-3649. DOI: 10.1177/0278364912452621. URL: http://dx.doi.org/10.1177/0278364912452621.

[32] OB Kroemer y col. "Combining active learning and reactive control for robot grasping". En: *Robotics and Autonomous Systems* 58.9 (2010), págs. 1105-1116.

[33] James J Kuffner y Steven M LaValle. "RRT-connect: An efficient approach to single-query path planning". En: *Robotics and Automation, 2000. Proceedings. ICRA'00. IEEE International Conference on*. Vol. 2. IEEE. 2000, págs. 995-1001.

[34] Steven M LaValle. *Rapidly-Exploring Random Trees A New Tool for Path Planning*. Inf. téc. Computer Science Dept., Iowa State University, oct. de 1998.

[35] Leonidas Lefakis y col. "Boosted Edge Orientation Histograms for Grasping Point Detection". En: *20th International Conference on Pattern Recognition (ICPR)*. IEEE. 2010, págs. 4072-4076.

[36] Leonidas Lefakis y col. "Image-Based grasping point detection using boosted histograms of oriented gradients". En: *Image Analysis and Recognition*. Springer, 2010, págs. 200-209.

[37] Ian Lenz, Honglak Lee y Ashutosh Saxena. "Deep learning for detecting robotic grasps". En: *arXiv preprint arXiv:1301.3592* (2013).

[38] Fabio Leoni y col. "Implementing robotic grasping tasks using a biological approach". En: *Proceedings. IEEE International Conference on Robotics and Automation*. Vol. 3. IEEE. 1998, págs. 2274-2280.

[39] Maxim Likhachev. *SBPL, Search Based Planning Library*. 2009. URL: `http://www.sbpl.net` (visitado 25-11-2015).

[40] Maxim Likhachev, Geoffrey J Gordon y Sebastian Thrun. "ARA*: Anytime A* with provable bounds on sub-optimality". En: *Advances in Neural Information Processing Systems*. 2003, None.

[41] Maxim Likhachev y Anthony Stentz. "R* search". En: *Lab Papers (GRASP)* (2008), pág. 23.

[42] Maxim Likhachev y col. "Anytime Dynamic A*: An Anytime, Replanning Algorithm." En: *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*. Jun. de 2005, págs. 262-271.

[43] Jeffrey Mahler y col. "GP-GPIS-OPT: Grasp Planning With Shape Uncertainty Using Gaussian Process Implicit Surfaces and Sequential Convex Programming". En: (2015).

[44] Jeremy Maitin-Shepard y col. "Cloth grasp point detection based on multiple-view geometric cues with application to robotic towel folding". En: *IEEE International Conference on Robotics and Automation (ICRA)*. IEEE. 2010, págs. 2308-2315.

[45] Alexis Maldonado, Ulrich Klank y Michael Beetz. "Robotic grasping of unmodeled objects using time-of-flight range data and finger torque information". En: *Intelligent Robots and Systems (IROS), 2010 IEEE/RSJ International Conference on*. IEEE. 2010, págs. 2586-2591.

[46] Ashok Menon, Benjamin Cohen y Maxim Likhachev. "Motion planning for smooth pickup of moving objects". En: *Robotics and Automation (ICRA), 2014 IEEE International Conference on*. IEEE. 2014, págs. 453-460.

[47] Luis Montesano y Manuel Lopes. "Learning grasping affordances from local visual descriptors". En: *Development and Learning, 2009. ICDL 2009. IEEE 8th International Conference on*. IEEE. 2009, págs. 1-6.

[48] Plinio Moreno, Jonas Hornstein y Jose Santos-Victor. *Learning to grasp from point clouds*. Inf. téc. Technical report, Vislab-TR001/2011. Dept. of Electrical y Computers Eng., Instituto Superior Tecnico, 2011.

[49] Matthias Nieuwenhuisen y col. "Shape-primitive based object recognition and grasping". En: *Robotics; Proceedings of ROBOTIK 2012; 7th German Conference on*. VDE. 2012, págs. 1-5.

[50] Mitesh Patel y col. "Learning object, grasping and manipulation activities using hierarchical HMMs". En: *Autonomous Robots* 37.3 (2014), págs. 317-331.

[51] Mila Popović y col. "A strategy for grasping unknown objects based on co-planarity and colour information". En: *Robotics and Autonomous Systems* 58.5 (2010), págs. 551-565.

[52] Judith MS Prewitt y Mortimer L Mendelsohn. "The analysis of cell images". En: *Ann. NY Acad. Sci* 128.3 (1966), págs. 1035-1053.

[53] Arnau Ramisa y col. "Learning RGB-D descriptors of garment parts for informed robot grasping". En: *Engineering Applications of Artificial Intelligence* 35 (2014), págs. 246-258.

[54] Joseph Redmon y Anelia Angelova. "Real-Time Grasp Detection Using Convolutional Neural Networks". En: *arXiv preprint arXiv:1412.3128* (2014).

[55] D Riafio y col. "Object detection methods for robot grasping: Experimental assessment and tuning". En: *Artificial Intelligence Research and Development: Proceedings of the 15th International Conference of the Catalan Association for Artificial Intelligence*. Vol. 248. IOS Press. 2012, pág. 123.

[56] Mario Richtsfeld y Markus Vincze. "Grasping of Unknown Objects from a Table Top". En: *Workshop on Vision in Action: Efficient strategies for cognitive agents in complex environments*. Markus Vincze and Danica Kragic and Darius Burschka and Antonis Argyros. Marseille, France, oct. de 2008. URL: `https://hal.inria.fr/inria-00325794`.

[57] Mario Richtsfeld y Markus Vincze. *Robotic Grasping of Unknown Objects*. INTECH Open Access Publisher, 2009.

[58] Lawrence Gilman Roberts. "Machine perception of three-dimensional solids". Tesis doct. Massachusetts Institute of Technology, 1963.

[59] Radu Bogdan Rusu y Steve Cousins. "3D is here: Point Cloud Library (PCL)". En: *IEEE International Conference on Robotics and Automation (ICRA)*. Shanghai, China, mayo de 2011.

[60] Oren Salzman y Dan Halperin. "Asymptotically near-optimal RRT for fast, high-quality, motion planning". En: *Robotics and Automation (ICRA), 2014 IEEE International Conference on*. IEEE. 2014, págs. 4680-4685.

[61] Gildardo Sánchez y Jean-Claude Latombe. "A single-query bi-directional probabilistic roadmap planner with lazy collision checking". En: *Robotics Research*. Springer, 2003, págs. 403-417.

[62] Ashutosh Saxena, Justin Driemeyer y Andrew Y Ng. "Robotic grasping of novel objects using vision". En: *The International Journal of Robotics Research* 27.2 (2008), págs. 157-173.

[63] Ashutosh Saxena y col. "Learning to grasp novel objects using vision". En: *Experimental Robotics*. Springer. 2008, págs. 33-42.

[64] Ashutosh Saxena y col. "Robotic grasping of novel objects". En: *Advances in neural information processing systems*. 2006, págs. 1209-1216.

[65] Irwin Sobel y Gary Feldman. "A 3x3 isotropic gradient operator for image processing". En: (1968).

[66] Siddhartha S Srinivasa y col. "HERB: a home exploring robotic butler". En: *Autonomous Robots* 28.1 (2010), págs. 5-20.

[67] Jörg Stückler y col. “Efficient 3D object perception and grasp planning for mobile manipulation in domestic environments”. En: *Robotics and Autonomous Systems* 61.10 (2013), págs. 1106-1115.

[68] Jörg Stückler y col. “Real-Time 3D Perception and Efficient Grasp Planning for Everyday Manipulation Tasks.” En: *European Conference on Mobile Robots (ECMR)*. 2011, págs. 177-182.

[69] Ioan A. Sucan y Sachin Chitta. *MoveIt!* 2014. URL: `http://moveit.ros.org` (visitado 25-08-2015).

[70] Ioan A Şucan y Lydia E Kavraki. “Kinodynamic motion planning by interior-exterior cell exploration”. En: *Algorithmic Foundation of Robotics VIII*. Springer, 2010, págs. 449-464.

[71] Ioan A. Şucan, Mark Moll y Lydia E. Kavraki. “The Open Motion Planning Library”. En: *IEEE Robotics & Automation Magazine* 19.4 (dic. de 2012). `http://ompl.kavrakilab.org`, págs. 72-82. DOI: `10.1109/MRA.2012.2205651`.

Apéndice A

Anexos

A.1. Especificación experimentos para grasping de algoritmos en OMPL

Datos de configuración de algoritmos de planificación de trayectorias usados en pruebas y especificaciones del equipo de pruebas.

A.1.1. Parámetros algoritmos de planificación de trayectorias

- EST
  - `goal_bias` = 0.050000000000000003
  - `longest_valid_segment_fraction` = 0.01
  - `projection.cellsize.0` = 23.6519126892
  - `projection.cellsize.1` = 23.646852111800001
  - `projection.cellsize.2` = 23.6463912963765
  - `range` = 164.15498006390021
  - `valid_segment_count_factor` = 1
- KPIECE1
  - `border_fraction` = 0.90000000000000002
  - `failed_expansion_score_factor` = 0.5
  - `goal_bias` = 0.050000000000000003
  - `longest_valid_segment_fraction` = 0.01
  - `min_valid_path_fraction` = 0.20000000000000001
  - `projection.cellsize.0` = 23.6519126892
  - `projection.cellsize.1` = 23.646852111800001
  - `projection.cellsize.2` = 23.6463912963765

  - range = 164.15498006390021
  - valid_segment_count_factor = 1
- RRT
  - goal_bias = 0.050000000000000003
  - longest_valid_segment_fraction = 0.01
  - projection.cellsize.0 = 23.6519126892
  - projection.cellsize.1 = 23.646852111800001
  - projection.cellsize.2 = 23.6463912963765
  - range = 164.15498006390021
  - valid_segment_count_factor = 1
- STRIDE
  - degree = 16
  - estimated_dimension = 6
  - goal_bias = 0.050000000000000003
  - longest_valid_segment_fraction = 0.01
  - max_degree = 18
  - max_pts_per_leaf = 6
  - min_degree = 12
  - min_valid_path_fraction = 0.20000000000000001
  - projection.cellsize.0 = 23.6519126892
  - projection.cellsize.1 = 23.646852111800001
  - projection.cellsize.2 = 23.6463912963765
  - range = 164.15498006390021
  - use_projected_distance = 0
  - valid_segment_count_factor = 1

A.1.2. Características del equipo utilizado

- Versión OMPL: 0.15.0
- Architecture: x86_64
- CPU op-mode(s): 32-bit, 64-bit
- Byte Order: Little Endian
- CPU(s): 16
- On-line CPU(s) list: 0-15
- Thread(s) per core: 2
- Core(s) per socket: 4
- Socket(s): 2
- NUMA node(s): 2
- Vendor ID: GenuineIntel

- CPU family: 6
- Model: 44
- Stepping: 2
- CPU MHz: 2000.000
- BogoMIPS: 4799.90
- Virtualization: VT-x
- L1d cache: 32K
- L1i cache: 32K
- L2 cache: 256K
- L3 cache: 12288K
- NUMA node0 CPU(s): 0-3,8-11
- NUMA node1 CPU(s): 4-7,12-15

A.2. Parámetros segmentación

Estos son los parámetros usados en la segmentación.

- Cálculo de normales
  `normalEstimationMethod` Matriz de covarianza
  `maxDepthChangeFactor` 0.0025
  `useDepthDependentSmoothing` falso
  `normalSmoothingSize` 12
- Ajuste de plano
  `normalDistanceWeight` 0.1
  `maxIterations` 50
  `distanceThreshold` 0.1
  `optimizeCoefficients` verdadero
  `probability` 0.99
  `sampleMaxDistance` 0.0
  `planeX`, `planeY`, `planeZ` [0.0, 0.0, 1.0]
  `epsAngle` 20°
- Filtrado sobre la mesa
  `minHeight` y `maxHeight` [0.005, 0.25]