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

MANIPULACIÓN Y POSICIONAMIENTO DE OBJETOS DESCONOCIDOS POR PARTE DE UN ROBOT AUTÓNOMO

MEMORIA PARA OPTAR AL TÍTULO DE INGENIERO CIVIL EN COMPUTACIÓN

MAXIMILIANO ANDRÉS CASTRO DÍAZ

PROFESOR GUÍA:
PABLO GUERRERO PÉREZ

MIEMBROS DE LA COMISIÓN:
JOHAN FABRY
ÉRIC TANTER

SANTIAGO DE CHILE
2016

RESUMEN DE LA MEMORIA PARA OPTAR AL TÍTULO DE INGENIERO CIVIL EN COMPUTACIÓN
POR: MAXIMILIANO CASTRO DÍAZ
FECHA: MAYO 2016
PROFESOR GUÍA: PABLO GUERRERO

MANIPULACIÓN Y POSICIONAMIENTO DE OBJETOS DESCONOCIDOS POR PARTE DE UN ROBOT AUTÓNOMO

Para investigación en el área de robótica autónoma, el Laboratorio RyCh (Robótica y Computación en Chile) del Departamento de Ciencias de la Computación de la Universidad de Chile dispone del robot PR2, que provee de una plataforma completa para implementación y testeo de algoritmos orientados a solucionar problemas de este ámbito. Una de las materias altamente estudiadas globalmente es el de la capacidad de manipulación de los robots en ambientes cotidianos, con mínima o nula intervención humana sobre las acciones del robot en el proceso.

El presente trabajo de memoria aborda el problema de "*placing"* (posicionamiento) autónomo de objetos, para entregar así al robot PR2 una funcionalidad hasta ahora ausente y de alta utilidad. Para lograr este cometido, se llevó a cabo en una primera etapa un proceso de investigación sobre el tema y las herramientas a utilizar, para luego pasar a las etapas de diseño, implementación y testeo de una solución computacional que resuelva este problema.

Específicamente, en las primeras etapas del trabajo, se investigó acerca de la constitución lógica y física del robot, sobre sus alcances y limitaciones y estándares para el trabajo con esta plataforma. Paralelamente, se trabajó sobre el robot a modo de corroboración de los conceptos aprendidos y se destinó una gran parte del tiempo a realizar indagaciones acerca de trabajos similares realizados sobre este y otros sistemas robóticos autónomos. A continuación se analizó a fondo el problema a resolver, pudiendo así identificar sus posibles aristas y acotar el trabajo realizable con los recursos disponibles, llegando a la definición exacta de lo que se busca desarrollar: Un algoritmo de posicionamiento de objetos desconocidos en superficies planas para el robot PR2.

Se prosiguió con el diseño, implementación y testeo de los cinco módulos principales de la solución: *Modelamiento del objeto*, *Cálculo de Pose Estable*, *Búsqueda de Superficie*, *Desplazamiento hacia Superficie* y *Posicionamiento* cuyas subcomponentes funcionales formaron la estructura total del software, dando origen a mecanismos de control motriz del robot y a algoritmos específicos para escaneo tridimensional de objetos en una tenaza del robot, detección de superficie de apoyo estable óptima para objetos de diversas geometrías, búsqueda autónoma de una superficie adecuadas para posicionamiento basada en segmentación y traducción de pose estable del objeto a posiciones de la tenaza que lo sostiene, entre otros. Se creó también un módulo preliminar: *Preparación del* Robot, para llevar al PR2 al estado inicial de posicionamiento de objeto.

El desarrollo de estas componentes se realizó de forma independiente, testeando y ajustando parámetros de cada algoritmo. La unificación de todas estas partes dio origen finalmente al programa completo de "*placing*", que resultó en el posicionamiento exitoso de objetos de diversa complejidad geométrica. Los casos fallidos se deben casi exclusivamente a la mala elección del espacio en la superficie utilizada.

Se ofrece al final del documento detalles y análisis sobre su aplicación en diversos casos de estudio, mostrando sus fortalezas en la estimación de estabilidad, y debilidades en elección de superficies de posicionamiento. Mejoras al programa son propuestas luego en la sección Trabajo Futuro. El trabajo permitió entender mejor el modo de desarrollo actual para la robótica y en este proceso sentó las bases para la entrega de un sistema funcional.

A mi mamá, Diana, mi cable a tierra e inagotable fuente de amor.

Agradecimientos

Muchas gracias a Ian Yon por su gran paciencia y buena disposición para compartir su experiencia conmigo.

A Jennifer Buehler por su tolerancia y destreza invertidas para echar a andar una de las componentes fundamentales de este trabajo.

A los amigos músicos con los que tuve el placer de compartir alguna juerga instrumental de esas que llenan el alma.

Gracias en general a todos los que aportaron compañía, buen humor, contención y apoyo a toda costa: mi madre, hermana y familiares, los viejones computines, compañeros del colegio… Con ustedes todo esto toma sentido.

Tabla de Contenido

Índice de Ilustraciones

Índice de Algoritmos

Capítulo 1

Introducción

Contexto

La Robótica

La robótica es un área de investigación muy activa en la actualidad, congregando a personas de todo el mundo a participar del mejoramiento de los robots y su comportamiento. Esto tiene sentido si se toma en cuenta los enormes beneficios que estas máquinas han traído a la humanidad, al aumentar considerablemente la *seguridad*, *productividad*, *eficiencia energética* y *precisión* en procesos industriales de fabricación de diversos insumos, y en la asistencia en general al humano en diversos ámbitos, ya sea complejos o cotidianos.

Desde su comienzo hasta el día de hoy, ha evolucionado notablemente. Las últimas generaciones de robots son capaces de tomar decisiones cada vez más intrincadas basadas en sus mecanismos de percepción del entorno e ingeniosos algoritmos diseñados de tal manera de disminuir gradualmente la dependencia de un ser humano para lograr ejecutar una tarea objetivo, aumentando su *autonomía*.

Se puede ver entonces que la robótica es un excelente ejemplo de cómo la computación puede tener un efecto directo sobre el mundo real.

Robótica Autónoma

Un robot autónomo es una máquina capaz de realizar tareas con un alto grado de autonomía, sin control humano explícito. Ejemplos de este concepto van desde el robot *Roomba* (Ilustración 1) hasta robots de exploración espacial.

Ilustración 1: Robot Roomba

Para que un robot sea considerado como completamente autónomo debe poder realizar las siguientes tareas por sí mismo:

- Obtener información de su ambiente.
- Trabajar de manera independiente por un tiempo considerable.
- Desplazarse en su ambiente de trabajo e interactuar con él.
- Evitar situaciones peligrosas para sí mismo, para las personas y para el entorno, cuando este no sea su objetivo.

No es difícil comenzar a imaginar cómo estas características pueden llegar a ser de utilidad en diversos ambientes donde los humanos no somos capaces de trabajar, ya sea por razones de seguridad, capacidad o limitada destreza.

Motivación

Es sabido que en el presente se hace constante uso de una amplia variedad de máquinas para asistirnos en múltiples labores, habiendo logrado así mejorar nuestras tecnologías para apoyar procesos de producción y tareas cotidianas. Sin embargo el poder que supone el dotar de cierto grado de autonomía a estas máquinas es enorme.

Por ejemplo, en la exploración espacial, es conveniente enviar robots no tripulados, pero es también difícil mantener constantemente el control del robot de manera fluida, dadas las inherentes limitaciones de nuestros sistemas de comunicaciones y las largas distancias que estas máquinas deben recorrer. Por esto, se investigan y se implementan maneras de aumentar la autonomía de éstas de manera de transparentar y automatizar cada vez más la ejecución de acciones y tomas de decisiones "simples", como la navegación en ambientes impredecibles desde el punto de vista del robot. Otra capacidad sumamente relevante es la de auto mantención, tarea que en otra época sería imposible sin la asistencia de una o más personas.

La variedad de materiales y tecnologías disponibles para el diseño e implementación de robots permite la creación de máquinas capaces de trabajar en condiciones de otra manera inalcanzables para nosotros como especie. Ante esta realidad es posible imaginar entonces por qué la robótica autónoma significa un área de estudio de altísimo interés.

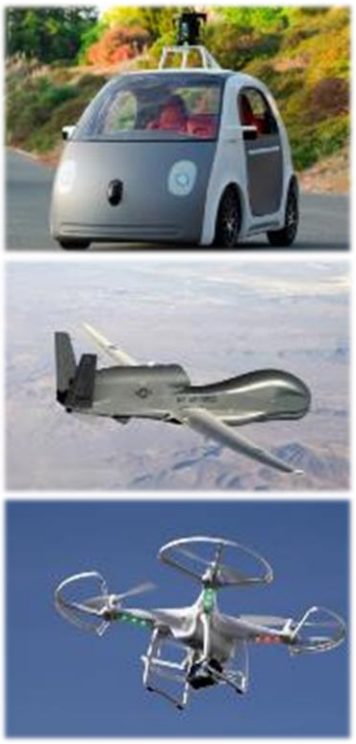


Ilustración 2: Ejemplos de navegación autónoma

Estado Actual

- *Navegación y aeronáutica*

Los automóviles hoy en día constituyen máquinas cada vez más automatizadas, siendo capaces de obtener información de variables internas (temperatura, niveles, navegación, estados) y externas (colisiones, temperatura, emisiones) para el apoyo a la comodidad y seguridad al momento de conducir. Lo mismo ocurre con aeronaves militares y comerciales.

Además de esta semi-automatización, existen ya prototipos altamente funcionales de automóviles y aeronaves (Ilustración 2) no tripulados que, haciendo uso de algoritmos de *machine learning* y nociones de *big data*, logran perfeccionar cada vez más su raciocinio en ambientes de tráfico automovilístico regular.

Ilustración 3: Brazo robótico clasificador de items

- *Industria*

Muchos procesos industriales dependen de máquinas con capacidad de toma de decisiones. Debido al grado de incertidumbre del entorno de trabajo de estas máquinas (por ejemplo, la

posición exacta de un ítem puede variar) es que la robótica autónoma tiene lugar en esta área. Un ejemplo se puede ver en la Ilustración 3.

- *Medicina*

Existen procedimientos en el área de la medicina que requieren de gran precisión o son, de manera no excluyente, de alta complejidad. Para esto, la asistencia de robots autónomos ha sido integrada de forma paulatina a medida que la robótica autónoma evoluciona.

Además se ha investigado y logrado avances importantes en la asistencia a personas con impedimentos motores. Con la ayuda de robots autónomos estos individuos han logrado realizar tareas que antes eran imposibles de concretar.

- *Uso doméstico*

También ha sido de interés el ambiente cotidiano como entorno de aplicación de este tipo de máquinas. Un buen ejemplo de esto es el robot *Roomba*, mencionado previamente, que es un robot autónomo capaz de navegar por el espacio del hogar y mantener limpio el suelo y alfombras sin asistencia.

- *Investigación*

Existen plataformas completas diseñadas con el fin de permitir al investigador probar nuevas ideas, estudiarlas y evaluar su desempeño, eliminando la necesidad de invertir energías cubriendo todo aspecto necesario para la construcción de un robot. Específicamente en el desarrollo de algoritmos para robots autónomos, permiten concentrar los esfuerzos en el desarrollo y pruebas de éstos directamente en un robot que garantiza de antemano su completa funcionalidad.

Manipulación

Un problema que ha captado el interés de los investigadores es el de otorgar a un robot la capacidad de interactuar, de forma autónoma, con objetos en un ambiente cotidiano, principalmente dentro de una vivienda. Este interés nace de la importancia que se observa en las aplicaciones de este concepto, que cubren desde la mera comodidad hasta la asistencia a personas con alta incapacidad motora y/o cognitiva, para quienes una máquina de estas características significa un aumento considerable de su independencia.

El problema mencionado puede separarse en dos aspectos centrales: la capacidad de *tomar ("grasp")* y *dejar ("place")* ese objeto en alguna configuración conveniente. Es este último punto, el de posicionamiento, el que concentra los esfuerzos de este trabajo de título.

Contexto de la Universidad

El Departamento de Ciencias de la Computación de la Universidad de Chile (DCC) tiene actualmente a disposición de los estudiantes y profesores el robot PR2, ideal para la investigación de software para robots. Sin embargo, hasta el momento, no consta con un algoritmo de placing integrado. Más detalles sobre esta máquina y el software correspondiente se explicitarán más adelante en este documento.

El problema de "*Placing*"

Lo que para la mayoría de nosotros los humanos es totalmente trivial en la cotidianeidad, para un robot puede ser altamente complejo. El *placing*, o posicionamiento de objetos, cumple sin duda con esta característica.

De manera totalmente intuitiva para nosotros, al llevar un objeto desde la mano a una superficie, somos capaces de rápidamente identificar ***cómo*** y ***hacia dónde*** movernos para desplazarlo efectivamente y ***cómo posicionarlo*** de manera estable. Para dar a un robot la capacidad de realizar estas mismas tareas por sí mismo debemos hacer uso de sus capacidades de percepción e interacción con su entorno. El Diagrama 1 muestra a grandes rasgos el procedimiento involucrado, desde que el robot toma el objeto, hasta que lo deja.

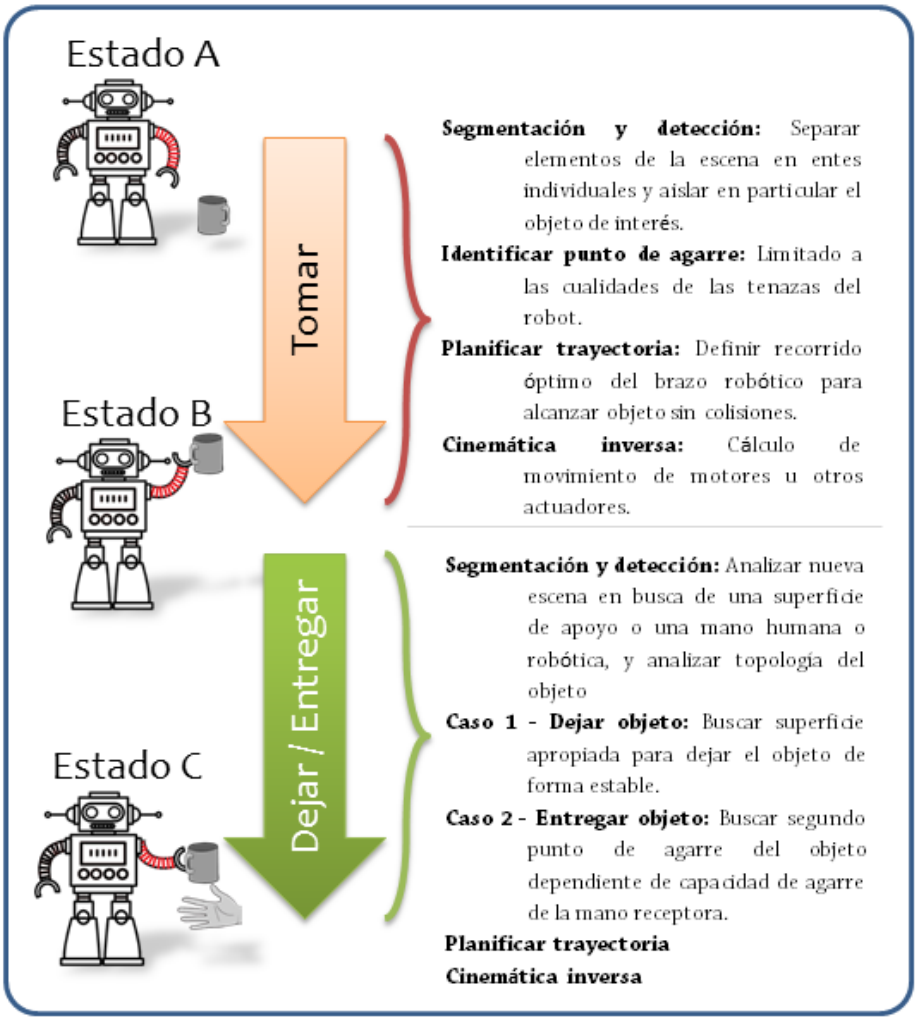


Diagrama 1: Entendiendo los problemas de grasping *y* placing

Objetivos

General

Desarrollar e implementar una metodología de posicionamiento estable de objetos en una superficie, por parte de un robot móvil autónomo, que considere la disposición geométrica tanto de dicha superficie de apoyo como la del objeto a posicionar, el cual no necesariamente es conocido con antelación por el robot.

Específico

- Desarrollar un método de generación de una representación digital tridimensional de un objeto desconocido, a partir de datos adquiridos por el robot.
- Desarrollar un algoritmo de determinación de zonas de apoyo del objeto que aseguren un buen alineamiento entre éste y la superficie, así como cumplimiento de condiciones de equilibrio mecánico.
- Implementar un método de detección de superficie de destino a partir de datos sensados.
- Desarrollar una estrategia de aproximación de la base móvil del robot a la superficie detectada.
- Construir un mecanismo para indicar al robot exactamente cómo posicionar el objeto en la superficie.
- Enfocar el desarrollo del software para su compatibilidad con la arquitectura del sistema ROS y para su uso en el robot PR2.

El trabajo realizado

Una primera fase relevante consistió en limitar a través de supuestos convenientes la extensión de la gran cantidad de variables implicadas en el problema de *placing* (abordadas con más detalle en el Capítulo 3 del presente documento) de manera de acotar el problema.

La solución propuesta separa las actividades que debe efectuar el robot en cinco acciones principales:

1. **Modelamiento del objeto:** En esta etapa el robot observa el objeto y elabora una representación computacional del mismo.
2. **Detección de pose estable:** A partir del modelo obtenido del objeto, se calcula su posición de apoyo más estable sobre un plano.
3. **Búsqueda de superficie adecuada:** En este punto el robot ya tiene una noción de las dimensiones del objeto, por lo que puede buscar una superficie adecuada para su apoyo. Para esto el robot debe observar su entorno, analizar la información obtenida y decidir qué superficie conviene utilizar como área de trabajo.
4. **Desplazamiento hacia superficie:** Tras encontrar el área de trabajo tentativa, el robot debe evaluar cómo desplazarse hacia ella y hacerlo.
5. **Posicionar el objeto:** Teniendo una representación de la superficie de trabajo y la mejor forma de posicionar el objeto manipulado, es necesario determinar la pose exacta a la que el robot debe llevar sus tenazas para asegurar un posicionamiento estable, recorriendo además una trayectoria libre de obstáculos.

Cada una de ellas fue implementada y evaluada de manera independiente para finalmente ser sintetizadas en un solo algoritmo de *placing* capaz de obtener una razonable proporción de resultados exitosos de posicionamiento estable.

Capítulo 2

Marco Teórico

Conceptos importantes

Antes de poder entrar en detalles teóricos se aconseja al lector no familiarizado con esta temática poner atención a los conceptos relevantes que se describen a continuación y son utilizados a lo largo del documento.

a) Métodos y algoritmos

Segmentación

Es el proceso de dividir un conjunto de elementos en subconjuntos llamados *segmentos*. En el contexto de este trabajo los elementos corresponden a puntos en el espacio tridimensional. La Ilustración 4 resalta el resultado de segmentar una nube de puntos para obtener de ella dos subconjuntos con datos que mejor representan a un plano, bajo ciertas restricciones específicas, encontrando en este caso particular dos coincidencias: El conjunto de puntos del suelo (en color magenta), y el de la mesa (en color celeste).

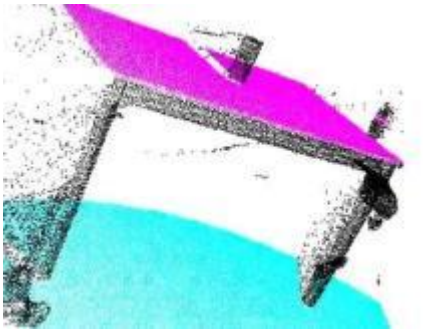


Ilustración 4: Segmentación en escena de mesa con objeto

Algoritmos RANSAC y PROSAC

RANSAC (Random Sample Consensus) es un método iterativo diseñado para estimar parámetros de un modelo matemático desde un conjunto de datos. Por cada iteración se toma como punto de referencia inicial un elemento seleccionado aleatoriamente. El resultado es no-determinista en el sentido de que se entrega una respuesta con cierta probabilidad de ser correcta, la que aumenta a medida que se permiten más iteraciones.

PROSAC (Progressive Sample Consensus) es una aproximación similar pero esta vez ya no tomando aleatoriamente la muestra inicial de cada iteración, sino que priorizando el orden de la selección de estos datos en base a algún criterio.

b) Conceptos geométricos

Normal

Cuando se utilice este concepto, se hará refiriéndose al *vector normal a una superficie*.

Centroide

El centroide de un conjunto de puntos es el punto resultante de la suma vectorial de todas las posiciones de los puntos, divida escalarmente por la cantidad de puntos. El resultado representa su centro geométrico.

Cuaternión (o cuaternio)

Definido por una 4-tupla de valores, es una manera de representar la operación de *rotación* en tres dimensiones relativa a un sistema de referencia. Su uso se justifica por la simplificación que significa tanto en almacenamiento como en facilidad de uso en un contexto computacional, y es la representación nativamente utilizada en las herramientas de software utilizadas para este trabajo. La definición formal de este concepto no es trivial, pero para efectos de este trabajo, basta con saber lo anterior.

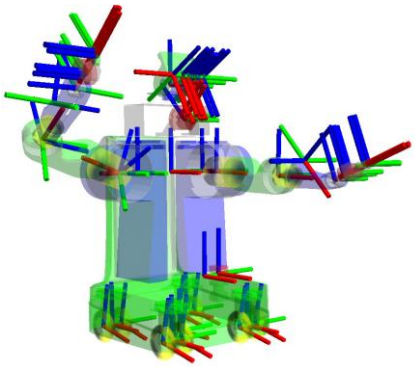


Ilustración 5: Orígenes de sistemas de referencia de los distintos frames de un robot PR2

Rotaciones RPY (Roll, Pitch, Yaw)

Notación para definir una rotación en el espacio. Existen diferentes formas de interpretar estas rotaciones. Para este trabajo, *Roll* corresponderá a una rotación en torno al eje X, *Pitch* al eje Y y *Yaw* al eje Z.

Voxel

Análogo a un pixel en una imagen, un *voxel* representa una unidad de volumen del espacio discretizado.

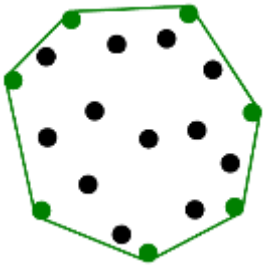


Ilustración 6: Envoltura convexa en un conjunto de datos de 2 dimensiones

Minimum Bounding Box

Una nube de puntos tridimensional finita puede ser siempre confinada dentro de una o más *cajas* tridimensionales. El concepto de *Minimum Bounding Box* (de ahora en adelante, *Bounding Box*) de un objeto hace referencia a la caja de *menor volumen* que lo inscribe completamente.

Frame

Cuando se trabaja con coordenadas espaciales para un sistema particular, es necesario definir un sistema de referencia primero. En un robot existen muchas partes que pueden ser usadas como el origen de un sistema de coordenadas. Un *frame* (Ilustración 5) es exactamente un sistema de referencia fijo a alguna parte del robot o del escenario mismo. Distintos sensores ocupan distintos frames por encontrarse fijos a distintas partes de la máquina. Es el S.O. del robot quien se ocupa de mantener la relación entre todos los frames.

Envoltura Convexa

La envoltura (o envolvente) convexa de un conjunto de puntos es un concepto matemático que puede ser definido informalmente, como el cuerpo *convexo* más pequeño que contiene a todos los puntos del conjunto.

En la Ilustración 6 se grafica el concepto de envoltura convexa en un conjunto de puntos en el espacio bidimensional.

Pose

Es un concepto utilizado para definir una *posición* y *orientación* particular en el espacio, relativo a algún sistema de referencia. La Ilustración 7 muestra la aplicación de la traslación y rotación necesarias a un objeto para llevarlo desde el origen del sistema a su pose final.

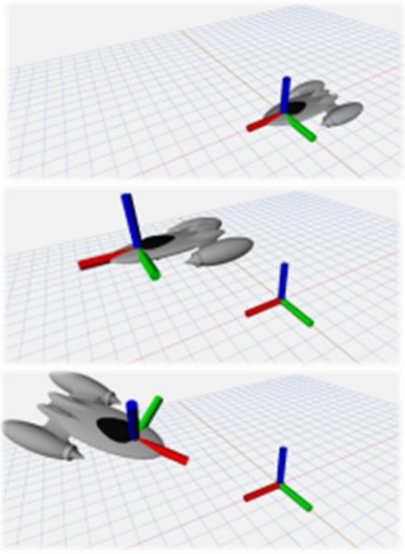


Ilustración 7: Pose final de un objeto, respecto a frame de referencia

c) Relativos a robots

Actuador

Define a todo dispositivo capaz de transformar energía en la activación de un proceso. Ejemplos de actuadores son motores, luces, pistones hidráulicos, solenoides.

Gripper

Denominación en idioma inglés de las tenazas de un robot. El término proviene del verbo "to grip" que significa aferrar, agarrar, tomar.

Efector

Parte de un robot relacionada con actuadores, capaz de realizar una acción específica. Ejemplos de ellos son: *gripper*, ruedas, brazo, luces.

DOF (Degrees of Freedom)

Este término hace referencia a los grados de libertad que posee un robot o alguna de sus partes. Cada eje de movimiento de un actuador es considerado un grado de libertad.

Link

Unidad sólida del robot que une dos articulaciones adyacentes. Un ejemplo es el antebrazo del PR2, que une la articulación flectora del codo con la flectora de la muñeca.

Joint

Provee articulación entre links.

Odometría

Estimación de pose en el entorno de un móvil, usando la información sobre el movimiento de los efectores con los que se desplace, como ruedas o piernas. Cuando se hable de un *frame* o *sistema odométrico*, se estará hablando del frame de referencia del mundo exterior, interpretado por el robot según su pose actual comparada con la de inicio.

d) Relativos a planificación de trayectorias

La forma en que se representa el mundo exterior a ser considerado en el planeamiento de trayectorias en el presente trabajo hace necesario conocer los siguientes conceptos:

Octomap

Es la forma elemental de representar al mundo visualizado como un potencial campo de objetos con los que se puede colisionar. Se mantiene un mapa de voxels actualizado de puntos considerados como potenciales puntos a esquivar, junto con las partes propias del robot, cuando se planea trayectorias de movimiento para los grippers.

Collision Object

Otra manera de representar una región en el espacio que el robot debe esquivar al moverse. Un *collision object* se puede generar programáticamente utilizando primitivas geométricas (esferas, cubos, cilindros, etcétera), planos y/o mallas de polígonos. En el planeamiento de trayectorias, se ignora todos los voxels del octomap encerrados por el collision object y alrededor de él.

Attached Collision Object

Hace referencia a un Collision Object ya no fijo a un punto del espacio, sino que a uno de los links del robot, junto al cual se moverá. Por ejemplo, un objeto tomado por una mano robótica se moverá junto al efector que lo sostiene, caso en el que es conveniente representar este objeto como un Attached Collision Object, acoplado al efector.

e) Otros

Inlier y outlier

En el contexto de este trabajo, dado un conjunto de puntos en el espacio tridimensional, y una característica analizada, todo punto del conjunto que cumpla con dicha característica es un *inlier*. El conjunto restante define a todos los *outliers*. Por ejemplo, si se considera todos los puntos que forman parte de un volumen esférico en el espacio, se tendría para una nube de puntos dada que los elementos ubicados al interior de dicho volumen serían inliers del modelo esférico, mientras que el resto de los puntos serían outliers.

El problema de Placing en la literatura

Respecto a la manipulación robótica autónoma, el interés de los investigadores se ha enfocado mayormente a resolver el problema de *grasping*, que es sin duda un desafío notable y altamente relevante en este ámbito, pero si se quiere ser capaz de levantar un objeto, probablemente se quiera también ser capaz de dejarlo, por lo que el *placing* es igualmente importante y no necesariamente constituye una lógica exactamente opuesta a la del *grasping* como podría uno imaginar.

Distintos grupos de personas han dedicado su tiempo a investigar el problema del *placing*. Cada uno de ellos se ha centrado en diferentes escenarios. Además, existen investigadores que intentan optimizar y enriquecer procesos intermedios necesarios para esta tarea.

A continuación se entrega una reseña sobre algunos de los trabajos presentes en la literatura asociados al problema del *placing*, organizada según sus principales enfoques:

Segmentación

Holz et al. [5] abordan el problema de segmentación usando imágenes RGB-D como las que maneja el *Kinect* del robot *PR2*, mientras que Mishra et al. [6] se enfocan en la mejora de este proceso en términos de eficiencia, orientándose a un desempeño de tiempo real. Tomando en cuenta que los objetos muchas veces pueden estar relacionados entre ellos, por ejemplo, unos sobre otros, Silberman et al. [8] estudian la factibilidad de segmentar la escena con esto en consideración. Como complemento, en el trabajo de Carson et al. [7] (Ilustración 8) se estudia el proceso de segmentación a partir de imágenes bidimensionales, más simples de obtener con las tecnologías a nuestro alcance. En el presente estudio se utiliza una metodología similar al trabajo de Holz et al., efectuando segmentación sobre nubes de puntos RGB-D usando el método *RANSAC*.

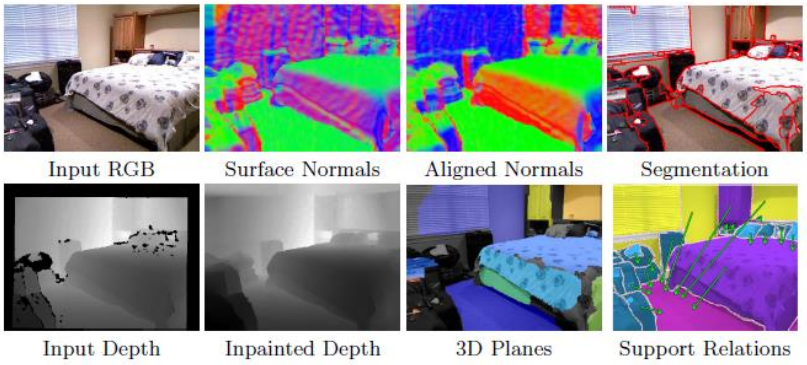


Ilustración 8: Etapas de segmentación plana."BlobWorld: Inage Segmentation Using Expectation-Maximization and its Application to Image Quering", N. Silberman et al (2002).

Análisis conceptual

Desde el punto de vista conceptual del proceso de *placing*, se estudia en el trabajo de Chan et al. [9] el proceso de manipulación de objetos desde una perspectiva psicológica humana, mientras que en el de Pandey et al. [10] se complementa esta idea, considerando variables más complejas que tan solo la estabilidad final del objeto. Todo esto es de alta utilidad para el diseño conceptual de este tema de memoria.

Placing como tal

Los estudios de Edsinger et al. [12], [13], entregan un análisis genérico de la evolución de las capacidades de manipulación robótica, y una visión realista de los alcances esperados en esta área. En "Manipulation in Human Environments" implementan algoritmos de posicionamiento para su robot humanoide Domo (Ilustración 9) orientado a entornos domésticos, para asistencia a personas con limitada movilidad. El enfoque de su investigación se centra más en la usabilidad del robot que en el perfeccionamiento del placing. La determinación de una posee estable para el objeto en este estudio se hace de forma simple, usando un control relajado del brazo para aprovechar la dinámica natural de los objetos a posicionarse de forma estable sobre una superficie plana. Este enfoque resulta ser muy simplista si se consideran escenarios más complejos.

En el trabajo de Stueckler et al. [11] se propone mejoras en la eficiencia en el planeamiento de agarre de objetos. Respecto al uso de las tenazas, Harada et al. [16] exponen sobre la utilización de ambos brazos en el proceso. Una interesante perspectiva sobre el *placing* es propuesta por Holladay et al. [14], donde se introduce el concepto de *Inverse Motion Planning*, que considera más a fondo las limitaciones de las tenazas del robot, planteando que en ciertas ocasiones al posicionar obstáculos, incluidas sus propias tenazas, de forma inteligente se puede ampliar el espectro de objetos manipulables y de poses finales. Otro notable estudio por Cowley et al. [15] expone acerca de las implicaciones de implementar algoritmos de manipulación para objetos en movimiento.

Un factor relevante en el estudio de las capacidades de manipulación de objetos por parte de robots en ambientes cotidianos es el nivel de caos de la escena en cuestión. En el trabajo de Schuster et al. [17] se aborda este problema orientando sus esfuerzos a la detección del nivel de desorden de una superficie con objetos, para finalmente encontrar un lugar útil de posicionamiento. Emeli et al. [18] se diseña un planeador de trayectorias para tareas de manipulación ejecutables por el robot PR2, orientado a modelar lo mejor posible el mundo observado, considerando específicamente casos de alto desorden. En su trabajo habilitan al robot para identificar y "empujar" elementos de la superficie para hacer espacio para el objeto manipulado.

Ilustración 9: Extracto de posicionamiento ejecutado por robot Domo. "Manipulation in Human Environments", Edsinger et-al (2006).

Objetos desconocidos

Para la determinación de posición final, el robot necesita tener alguna representación del objeto manipulado. Jiang et al. [19] hacen uso de técnicas de *machine-learning*, específicamente aprendizaje supervisado, para la manipulación de objetos nunca antes vistos por el robot, y determinación del lugar y pose óptimos para ubicarlo. La máquina intenta posicionar el objeto según sus conocimientos previos. A medida que entrena y es retroalimentado con respecto a la calidad de su trabajo, se aumenta la probabilidad de éxito para futuros posicionamientos de objetos desconocidos. Si además se posee una representación tridimensional detallada, la probabilidad de éxito es mayor.

Ilustración 10: Robot El-E asistiendo a paciente con ALS. "Hand It Over Or Set It Down: A User Study of Object Delivery with an Assistive Mobile Manipulator", Sang Choi et-al (2009).

Este estudio define un acercamiento interesante a este problema, y se considera para futuras mejoras del algoritmo desarrollado en este trabajo.

Aplicaciones específicas

Nguyen et al. [20] y Choi et al. [21] orientan sus estudios a la aplicación de la robótica en la medicina, específicamente pensando en pacientes con discapacidades motoras, similar a los estudios

de Edsinger et al. [12], [13]. Su robot *El-E (*Ilustración 10*)* es capaz de recibir indicaciones con puntero láser y otorgar feedback hablado, y de entregar objetos a los pacientes de forma directa o indirecta. En el intento por mejorar la interacción humano-robot para asistencia, Estrela et al. [22] implementan un algoritmo de detección de lenguaje de manos a partir de capturas RGB-D, y Ramírez-Hernández et al. [23] desarrollan un sistema de reconocimiento de manos humanas para reforzar los comandos de voz con gestos naturales.

El concepto de manipulación por parte de un único robot es refinado en el trabajo de Edsinger et al. [24], donde se estudian las implicancias de implementar algoritmos que den la capacidad de entregar objetos directamente a otro robot para generar una cadena de trabajo conjunto. Kehoe et al. [25] hablan de la misma idea, pero usando plataformas basadas en computación en la nube, de modo de generar una red colaborativa de robots donde puedan compartir sus aprendizajes independientes y coordinarse.

Herramientas a Utilizar

Para trabajar en la resolución del problema se dispone de las herramientas de software y hardware detalladas a continuación.

a) Software

a. ROS Hydro [1]

El proyecto de código abierto *ROS* (Robot Operating System), creado y mantenido por *Willow Garage*, consiste en una plataforma gratuita de software completa destinada al desarrollo de programas computacionales para robots, pensada para correr en sistemas operativos *Linux*. Es actualmente un gran estándar en la investigación en robótica, y posee una enorme comunidad de colaboradores a nivel mundial. La versión más reciente es *ROS Jade Turtle*, liberada el 23 de mayo de 2015. La versión con la que se trabajará es *ROS Hydro Medusa*, liberada el 4 de septiembre de 2013, que es la que el robot ejecuta actualmente. Ejemplos de robots que funcionan con este sistema se pueden ver en la Ilustración 11.

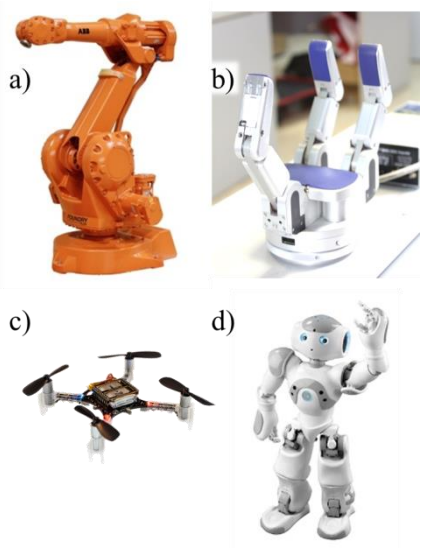


Ilustración 11: Robots que funcionan con ROS. a) es el ABB IRB 2400, b) el manipulador BarretHand, c) el quadcopter Crazyflize, y d) el robot Nao.

El sistema está diseñado con una arquitectura de **nodos**, cada uno con labores especializadas y capacidad de comunicarse entre ellos. En conjunto, estos nodos otorgan al robot capacidad de **sensar** e **interactuar** con el mundo.

Este diseño modular permite controlar uno o más robots desde una red de nodos distribuida en uno o más computadores conectados a una misma red IP. Otra gran característica del diseño de ROS es la reusabilidad del software desarrollado, al poder integrarse en diferentes robots sin mayores modificaciones.

Terminología de ROS

Principalmente, los nodos de ROS se comunican utilizando un modelo *Cliente-Servidor*. Los nodos comunican estructuras de datos llamadas ***mensajes*** a través de canales de comunicación denominados ***tópicos***. Un nodo ***publica*** mensajes en un determinado tópico, y otros pueden consumirlos ***suscribiéndose*** a dicho tópico (Diagrama 2). Existen nodos destinados a otorgar

servicios a todo nodo que los requiera. Un conjunto de Nodos conforman un ***paquete***, y un conjunto de Paquetes relacionados, un ***stack***.

Este sistema incluye numerosas librerías para acceder a las distintas funcionalidades de bajo y alto nivel que el robot ofrece, y permite su uso a través de programación en los lenguajes C++ y Python, y también por comandos de consola.

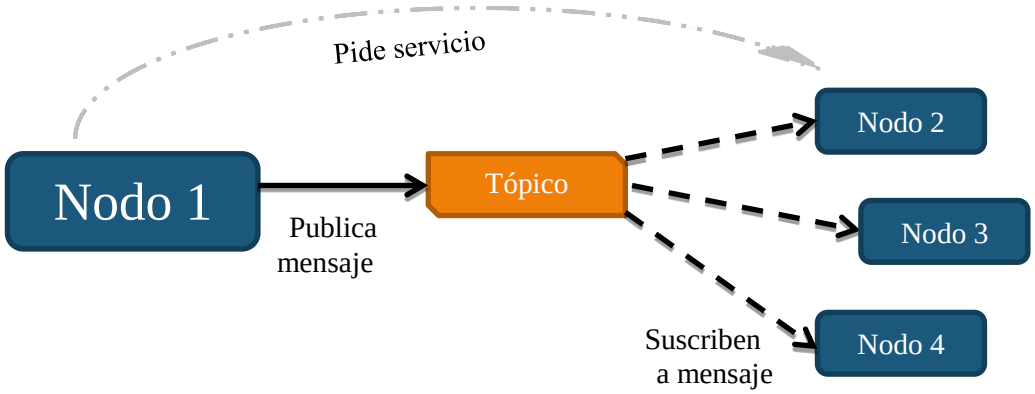


Diagrama 2: Esquema sencillo arquitectura ROS

Cada uno de los distintos tópicos presentes en el ambiente ROS aceptan un y sólo un tipo de mensajes para recibir y enviar información. Esto tiene sentido si se piensa en el contexto semántico del diálogo entre nodos.

Un ejemplo concreto es el caso del sensor Kinect. Dentro de todos los datos que éste adquiere del ambiente, se encuentra la nube de puntos que representa el entorno tridimensional observado en un instante de tiempo dado. El nodo que controla estos datos *publica* esta nube de puntos al tópico `"/head_mount_kinect/depth_registered/points"` que acepta mensajes de tipo `"sensor_msgs/PointCloud2"` que contiene la estructura interna necesaria para representar estos datos. Un nodo puede consumir la información adquirida por el sensor Kinect mediante una suscripción al mismo tópico, y leyendo la estructura de cada mensaje recibido.

Para el ajuste de parámetros diversos de distintos nodos, ROS implementa dentro de su arquitectura un *servidor de parámetros* donde se pueden realizar consultas y ajustes mientras el sistema se ejecuta, sin necesidad de editar y volver a compilar código fuente cada vez.

Para inicializar grandes árboles de procesos en ROS, existen los archivos `".launch"`, que, con directivas expresadas con lenguaje de marcado, puede iniciar múltiples nodos, configurar parámetros desde archivos e inicializar más archivos `.launch` en cascada de forma asíncrona.

b. PCL [2]

La librería de código abierto *PCL* (Point Cloud Library) (ejemplo de uso en Ilustración 12) es un completo conjunto de herramientas computacionales para procesamiento de imágenes y nubes de puntos. Provee implementaciones de algoritmos estándar de procesamiento de nubes de puntos, y

oros propios de la librería, diseñados para aplicación de ciertos filtros, estimación de características, reconstrucción de superficies, segmentación, procesamiento por GPU, entre otros. Además, está integrada con la libraría ***Eigen*** diseñada principalmente para trabajo con elementos de álgebra lineal en el espacio, para lo que provee de tipos de datos y algoritmos ad-hoc.

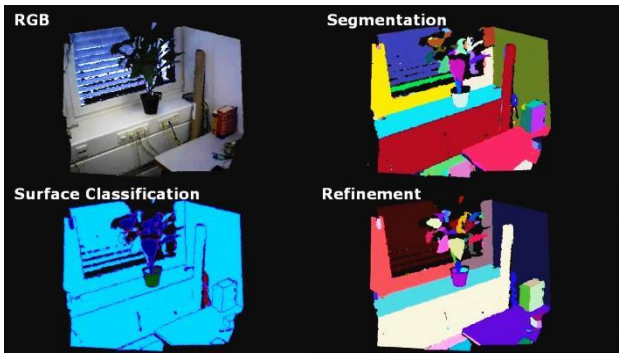


Ilustración 12: Distintas etapas de un caso particular de procesamiento de una escena tridimensional usando librería PCL.

Su implementación es totalmente integrable dentro de ROS, al ser un proyecto inicialmente diseñado para este sistema.

PCL además provee el ***PCL Visualizer*** que otorga un set de herramientas avanzadas específicamente diseñadas para visualización de datos de la librería, altamente configurables.

c. Rviz

El software Rviz es una plataforma de visualización de capturas de sensores integrados en una instancia del sistema ROS. Con esta herramienta es posible visualizar los datos obtenidos por los distintos sensores disponibles, en otras palabras, ver el mundo desde la perspectiva del robot. Se observa en la Ilustración 13 la visualización entregada por Rviz de los datos obtenidos por un robot en plena tarea de generación de mapas de navegación.

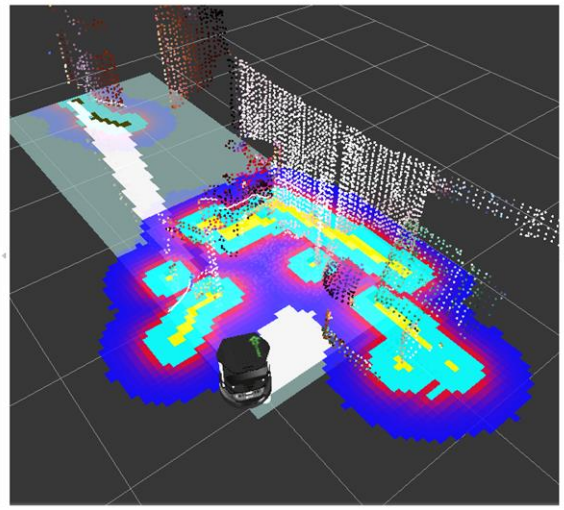


Ilustración 13: Visualización de Rviz del progreso del robot TurtleBot en la elaboración autónoma de mapas de navegación.

d. MoveIt! [3]

MoveIt! es una librería de código abierto ampliamente usada en el mundo de la robótica autónoma. Fue desarrollada para manipulación de objetos, ofreciendo herramientas de planeamiento de trayectorias, percepción 3D, cinemática, control, navegación y reconocimiento de objetos, entre otras. También es completamente integrable dentro de ROS por ser creada en un inicio, al igual que PCL, especialmente para este sistema.

Para su funcionamiento, MoveIt inicializa todo un contexto dentro de ROS que permite acceso a sus funcionalidades de planeamiento de trayectorias y configuraciones a través de distintas interfaces. Existe por ejemplo un complemento para Rviz que permite controlar los brazos del robot mediante controles visuales de alto nivel. Para el presente trabajo se utilizó la interfaz existente para su acceso a través del lenguaje C++: La "`planning_interface`".

Internamente, el contexto de la librería mantiene actualizados tres modelos del mundo, descritos en la Tabla 1, utilizados para considerar y evitar colisiones en el cálculo de trayectorias.

Modelo	Descripción	Utilidad
Octomap	Modela activamente todo punto espacial adquirido por los sensores disponibles como un vóxel que describe una región de potencial colisión.	Entrega a la librería una noción sobre el ambiente que rodea al robot, para ser considerado en el planeamiento de trayectorias.
Modelo del Robot	Descripción interna de todos los links del robot, actualizada constantemente.	Otorga una noción de la región en el espacio ocupada por cada parte del robot en todo momento, y de las limitaciones de sus actuadores. Todo punto dentro del modelo geométrico es excluido del octomap.
Collision Objects	Modelos geométricos personalizados agregados por el usuario.	Define una región volumétrica o plana en el mundo que MoveIt considera como un sólido colisionable. Todo punto dentro y suficientemente cerca de esta región se excluye del octomap.
Attached Collision Objects	Collision Objects unidos a alguno de los links de un robot.	Misma funcionalidad de los Collision Objects, pero se mantendrán fijos respecto al link al que se encuentran unidos.

Tabla 1: Componentes internas de MoveIt para detección de colisiones

Es relevante conocer también que MoveIt dispone de distintos *Planners* (planificadores de trayectorias) que usan distintas estrategias para resolver el problema de cinemática inversa que implica cada movimiento del brazo. Más información sobre Planners pueden encontrarse en el Anexo A.

e. Gazebo [4]

Gazebo es una plataforma capaz de emular en un entorno virtual escenarios realistas, desde el punto de vista de la física. A partir de modelos de robots y objetos, permite crear escenarios simulados para luego calcular en tiempo real de ROS las interacciones físicas entre los elementos presentes. Con esto se hace posible probar y depurar rápidamente algoritmos, diseñar robots y definir escenas complejas, sin tener que trabajar directamente con el robot real, con lo cual se puede acelerar el desarrollo. Un ejemplo de uso se observa en la Ilustración 14.

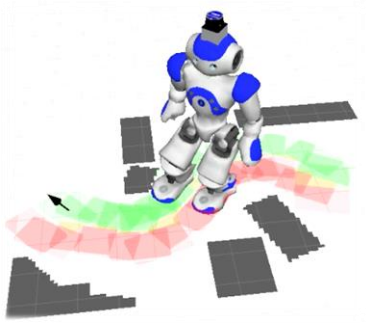


Ilustración 14: Robot Nao virtual ejecutando tareas de navegación en entorno simulado por Gazebo.

f. TF

TF es una librería que forma parte de la suite de funcionalidades integradas en el sistema operativo ROS. Su principal funcionalidad es la de proveer mecanismos transparentes de conversión entre diferentes *frames* del robot. Muchas veces se quiere expresar una misma posición en el espacio, en términos de distintos puntos de referencia, en distintos momentos, por ejemplo, se querría poder expresar los puntos adquiridos desde la Kinect (cuyas posiciones son almacenadas respecto al punto de referencia del lente receptor del aparato), pero relativos al frame de la base del robot, para lo que se debe tener una noción completa de la posición relativa entre estos dos puntos de referencia, para poder así tener una transformación tridimensional entre ambos sistemas.

Esta librería es capaz de hacer esto manteniendo una estructura de árbol actualizada que relaciona todos los distintos puntos de referencia y sus transformaciones relativas.

g. *Git*

Se hizo un control de versiones del software desarrollado usando la plataforma Git. Además, se utilizó un repositorio alojado en Github.

h. *Strata Design CX 6.0*

Para la generación de modelos de prueba preliminares, se utilizó este software de modelamiento 3D orientado a diseño gráfico. Posee una amplia librería de objetos pre-modelados, y la capacidad de exportar estos datos en diversos formatos, lo que permitió obtener nubes de puntos para testeo y corroboración de conceptos.

b) *Hardware*

a. *El PR2*

Como ya se dijo, existen diferentes plataformas de hardware y software diseñadas específicamente para la investigación en robótica autónoma. El robot *PR2 (*Ilustración 15*)* es una de ellas. Fue desarrollado por *Willow Garage* con la intención de aportar a la exploración de software para robots quitando la necesidad de preocuparse por las particularidades electromecánicas del funcionamiento de la máquina, garantizando de antemano robustez en sus funciones.

Esta plataforma ha sido escogida para el presente trabajo de memoria de título por encontrarse disponible en el Departamento de Ciencias de la Computación para su uso, y ser ideal en particular para el desarrollo de éste.

Ilustración 15: El robot PR2

Características

El PR2[1] es un robot de 1.33 m a 1.64 m de altura (variable) que consta principalmente de dos brazos de unos 80 cm de longitud, con 7 DOF y provistos cada uno de un *gripper* en su extremo para manipulación de objetos; una base móvil, una cabeza con capacidad para mirar hacia abajo, arriba y ambos lados, y un conjunto de sensores para visión y percepción del entorno en general.

Dentro del grupo de sensores se puede encontrar un *Microsoft Kinect*, una cámara stereo de rango amplio y una de rango corto, láser móvil de percepción de profundidad, láser planar estático, sensores de presión y acelerómetros en cada gripper, entre los más relevantes.

Como hardware de procesamiento el robot cuenta con dos computadores, cada uno equipado con un dos procesadores Quad-Core i7 Xeon, 24 GB de memoria RAM y 2TB de almacenamiento persistente.

Este robot tiene estrictos requerimientos de energía y cuidado, por lo que su uso significa pasar previamente por un sencillo pero obligatorio proceso de capacitación. Además se recomienda trabajar acompañado de otra persona capacitada.

[1] Más especificaciones técnicas del robot en: https://www.willowgarage.com/pages/pr2/specs.

El sensor Kinect

Para este proyecto, es este sensor el principal utilizado como nexo entre el mundo observable y el robot. El Kinect es un conjunto de sensores capaz de adquirir información de color, profundidad, inclinación y sonido. La Tabla 2 muestra algunas de sus propiedades. El simulador Gazebo es capaz de ejecutar una versión virtual de este sensor.

Característica	Valor
Ángulo de visión	**43° vertical, 57° horizontal**
Tasa de refresco	**30 FPS (canal RGB-D)**
Distancias adquisición en modo "Near"	**0.5m – 3m**
Distancias adquisición en modo normal	**0.8m – 4m**
Resolución cámara	**1280x960**

Tabla 2: Características Kinect

b. Computador con sistema Linux

La naturaleza de sistema distribuido de ROS permite que el trabajo se realice en un computador separado de los internos al robot. Al momento del desarrollo de este trabajo se contó con un Laptop *Lenovo ideapad S410p*[2]. ROS *Hydro* está diseñado para funcionar de manera óptima en el sistema operativo *Ubuntu 12.04*[3], por lo que se creó en este equipo una partición dedicada para este sistema, y destinada únicamente para trabajar con ROS.

[2] Especificaciones del equipo disponibles en: http://laptops.specout.com/l/1982/IdeaPad-S410p.
[3] Detalles del S.O. en: https://en.wikipedia.org/wiki/Ubuntu_%28operating_system%29.

Capítulo 3

Especificación del Problema

Tomar y no soltar

No hace falta tener mucha imaginación para entender que un escenario de manipulación de objetos donde se es capaz de tomar objetos, pero no dejarlos jamás, constituye rápidamente una actividad de poco sentido. Podría considerarse que, según el objetivo buscado, podría ser útil tener la capacidad de *grasp* únicamente. Un ejemplo posible es el de entregar un objeto a una persona. El robot no debe hacer más que tomar el objeto con su gripper, y esperar a que la persona lo retire, delegando el resto de responsabilidad en el humano. Sin embargo incluso en un escenario como el descrito, es necesario tener una noción de entrega de objetos, por ejemplo, muy probablemente sea necesario tener una noción de la posición de la persona y de criterios a tomar en cuenta para efectivamente poder soltar el objeto cuando la persona haya tomado el objeto de forma segura. Nos encontramos entonces ante la conclusión de que con seguridad se necesita algún algoritmo de *placing* al manipular objetos.

Particularmente en el caso del robot PR2 existen actualmente mecanismos integrados previamente en el sistema dentro de la librería *Move-It* para efectuar *grasping* y *placing*, que son útiles para situaciones de alta simpleza. Sin embargo no son suficientes para abordar la inmensa variabilidad que puede presentar un entorno real de trabajo, como se detallará más adelante, además de existir problemas de dependencias de módulos de ROS deprecados para la versión en uso. La literatura disponible muestra distintas formas de abordar el problema de placing pero en la investigación previa hecha no se encontró específicamente trabajos dedicados al modelamiento de un algoritmo de posicionamiento estable para objetos desconocidos. Se aborda en general el problema a partir de modelos previamente conocidos y a veces orientando los esfuerzos en situaciones que difieren del escenario considerado para este trabajo. Además, específicamente en el robot PR2 del *Laboratorio RyCh* no existe soluciones de placing más robustas que las ya integradas en Move-It, problema que, dicho todo lo anterior, justifica el presente trabajo.

Variables del *placing*

¿Qué elementos es necesario tener en cuenta cuando se posiciona un objeto? A continuación se explica de forma conceptual cuáles son las variables que se debe considerar en el caso de querer poner un elemento en algún lugar del espacio.

a) Características del lugar de posicionamiento

Existen, solamente en un ambiente doméstico, muchos posibles destinos finales para un objeto. Un lápiz puede ponerse en un lapicero, un plato en un posa platos o tal vez en una mesa, un huevo

en una bandeja para huevos, un patito de hule sobre el agua, una toalla en un tendedero, un libro puede ir debajo de otros tres libros (Ilustración 16). Además, en ciertos casos, se podría querer trabajar con una superficie de trabajo móvil, como una cinta transportadora o una persona caminando.

Ilustración 16: Ejemplos de posibles lugares de posicionamiento

Con estos ejemplos se puede tener una idea de la inmensa diversidad de escenarios posibles, cada uno dando origen a problemas muy diferentes y poco generalizables, por lo que resolver uno de ellos no implica directamente haber resuelto los demás.

b) Características del objeto

Esta es otra variable que define cómo debe abordarse el problema. Puede identificarse dentro de ella los siguientes parámetros:

- *Geometría*: La forma del objeto define en gran medida sus posiciones estables en el lugar de posicionamiento.
- *Tamaño*
- *Color:* ¿Cómo percibe el robot los objetos según los colores que éstos tengan? ¿Qué pasa con los objetos con transparencias?
- *Distribución de masa*: No todo objeto está constituido de una masa uniforme. Sin esta información es imposible determinar sus poses estables.
- *Restricciones de orientación*: En ciertas aplicaciones podría quererse mover un objeto sin girarlo aleatoriamente, como en el caso de una taza con café, donde no se desea derramar el líquido.
- *Fragilidad*: ¿Cuán delicado se debe y se puede ser según la fragilidad de un objeto?
- *Rigidez*: La capacidad de deformarse de algunos objetos puede resultar caótica al considerarla dentro de un algoritmo de manipulación.
- *Conocimiento previo*: Es posible que se tenga conocimiento previo sobre algunas de las anteriores variables, o bien, que se esté trabajando con un objeto totalmente desconocido para el robot. Esto determina en notable medida la manera de abordar la situación.

c) Movilidad

Aun sabiendo exactamente cómo posicionar el elemento manipulado, se debe tener en consideración las posibilidades de movilidad del robot en el escenario. Con cuánta facilidad puede **desplazarse** por el espacio y **mover sus extremidades** es de alta relevancia. En el contexto espacial que se está considerando, es indeseable modificar el escenario fuera de los parámetros planeados, por ejemplo botando objetos o estructuras, o rompiendo partes del entorno de trabajo o incluso del mismo robot por movimientos descuidados. Sin considerar este factor, se estaría ignorando la conexión del robot con su entorno real.

d) Capacidades

No es posible tener una visión completa del problema si no se toma en consideración las capacidades y limitaciones del robot mismo. Es totalmente necesario preguntarse, por ejemplo:

- *¿Qué sensores tengo disponibles para percibir mi entorno?*
- *¿Qué datos obtengo de ellos?*
- *¿Qué necesito para poder trabajar con estos datos?*
- *¿Es suficiente esta información para este problema?*
- *¿Qué tipo de objetos se puede manipular con este robot?*
- *¿Bajo qué condiciones se considera un área de trabajo como segura?*

Estas y más preguntas aclaran y acotan en gran medida qué seremos capaces e incapaces de hacer, y reduciremos el espacio de posibles formas de trabajar.

Acotando el problema

Es sabido que el tiempo es un recurso limitado, y particularmente en el caso de esta investigación, no se dispone del suficiente para cubrir a cabalidad todas sus aristas. En base a las *variables del placing* que se expone más atrás, se acota el problema de la siguiente manera:

a) Características del lugar de posicionamiento

Se considerará el caso de **superficies razonablemente planas** y de poca inclinación respecto al plano del suelo. Por supuesto debe ser alcanzable por el robot.

Si se toma en cuenta el amplio espacio de cobertura de los brazos del robot y su capacidad de extender el torso, se puede notar que se abordará una cantidad razonable de posibles escenarios de *placing*, muy similar al que posee una persona común. Esto, sumado a que nuestro algoritmo se orienta a trabajar en espacios cotidianos, muestra que las limitaciones impuestas en este punto no significan una gran reducción del total de posibles espacios de trabajo para el problema abordado.

b) Características del objeto

- *Geometría:* No existe una limitación particular de geometría, sin embargo, se decidió poner énfasis en lograr trabajar con objetos simples y de algún grado de simetría. Sin perjuicio de lo anterior, se puso a prueba el algoritmo con objetos de geometrías irregulares.
- *Tamaño:* El elemento que define este punto es el volumen de agarre de los *grippers*. Se considera objetos capaces de ser tomados por el robot, y que no ocupen un volumen de diámetro mayor a 40cm.
- *Color:* Se considerará objetos opacos, de cualquier color.
- *Distribución de masa:* Se prefiere elementos de densidad uniforme para una mejor estimación del centro de masa.
- *Orientación:* No se restringe la orientación.
- *Fragilidad y rigidez:* Por simplificación se optará por trabajar con objetos considerablemente rígidos y no frágiles.
- *Conocimiento previo:* El funcionamiento del algoritmo no requerirá conocimiento previo de la geometría del objeto ni de otro elemento relacionado a él.

c) Movilidad

Dado que el factor de *movilidad autónoma* no es el tema fuerte de esta investigación, se asumirá que se operará en un espacio **libre de obstáculos** e irregularidades que entorpezcan el desplazamiento del robot y de sus extremidades, a excepción de los objetos con los que se trabajará.

d) Capacidades

Este punto está definido por las especificaciones de hardware y software del robot PR2 y sus prestaciones.

Como conclusión, el problema a abordar se puede enunciar como la "*Manipulación autónoma y posicionamiento de objetos no conocidos sobre superficies planas, en un entorno cotidiano controlado*". Se exige el cumplimiento de la siguiente condición inicial:

El robot posee un objeto previamente tomado por uno de sus grippers.

Requisitos de Aceptación de Solución

Habiendo ya definido con exactitud el problema, se puede ahora determinar las pautas que permitan calificar a una solución como válida.

No transables

- Siempre que el robot tenga una superficie de posicionamiento aceptable en su entorno cercano, debe poder detectarla.
- El robot debe ser capaz de obtener una representación tridimensional del objeto a manipular.
- La pose estable determinada por el robot debe tener una tasa de éxito de un 70%.
- El robot debe posicionar objetos simples con al menos un 80% de éxito, y complejos con al menos un 50%.

Deseables

- Tras detectar una superficie de posicionamiento, el robot se desplaza quedando posicionado frente a ella de manera conveniente para efectuar el *placing*.
- La solución implementada debe poder ejecutarse en el robot real.

Capítulo 4

Descripción de la Solución

Trabajo preliminar

Previo a comenzar con la implementación de la solución del problema de *placing*, fue necesario preparar el camino.

a) Investigación

Durante todo el año, fue necesario dedicar tiempo al aprendizaje. Una importante parte se centró en la lectura de documentos relacionados con el problema de placing y de robótica en general, para entender los paradigmas actuales y aprender de las experiencias y alcances obtenidos por otros grupos de investigadores, tanto de logística como de implementación.

En otro ámbito, hubo que también estudiar constantemente sobre el correcto uso de las distintas herramientas necesarias para el trabajo, especificadas en el Capítulo 2. La confección de mapas conceptuales, resúmenes, perseverante ejercitación y participación activa en foros y listas de correos de sus comunidades de desarrollo y de usuarios en internet fueron clave para hacer uso efectivo de ellas, pasando de ser elementos desconocidos a ser parte del conjunto de instrumentos esenciales para todo el desarrollo.

b) Planificación

Además de lo anterior, se dedicó una porción del tiempo a la planificación de la estrategia que modelaría el resto del curso del trabajo.

Se optó por seguir una metodología de desarrollo incremental. Cada componente se implementaría en su forma mínima funcional, y se enriquecería sucesivamente hasta producir un resultado aceptable.

En este punto fue también necesario definir un punto de alta importancia para el resto del proceso: Hubo que escoger si desarrollar el algoritmo de *placing* directamente en el robot PR2 o inicialmente trabajar con el simulador Gazebo.

Las ***ventajas*** de usar el entorno visual Gazebo son principalmente:

- Al sólo necesitar un computador para funcionar, se tenía mayor flexibilidad en cuanto a dónde y cuándo trabajar, al contrario del robot PR2 real, que se encuentra sólo en la Universidad, y debe ser compartido con más estudiantes.
- El robot PR2 real es una máquina compleja que requiere de mucho cuidado en su uso.

Por otro lado, la ***desventaja*** principal es el hecho de que el robot PR2 es altamente superior en poder de cómputo a cualquier computador de gama media para uso doméstico, por lo que los tiempos de compilación y ejecución de los programas son mucho peores en el simulador Gazebo, que tiene que ejecutar versiones virtuales de todos los sensores y actuadores en uso, entre otras simulaciones.

Evaluando y sopesando estos pros y contras, se optó por realizar el trabajo inicialmente en el entorno virtual Gazebo, que promete portabilidad de los programas hacia el robot real.

c) Preparación

A continuación, se procedió a preparar el entorno de desarrollo. Este procedimiento se dividió en las siguientes actividades:

a. Instalación de sistema operativo Ubuntu 12.04

Existían dos opciones para hacer esto. La primera involucra la creación de una máquina virtual con el sistema. Esta opción fue descartada dadas las altas exigencias de procesamiento que tiene el funcionamiento de ROS junto a Gazebo, lo que dado el sobrecosto que significa para el sistema operativo ejecutar una máquina virtual, deprimiría en gran medida la calidad de la simulación, ralentizando el desarrollo.

Se optó entonces por crear una nueva partición en el disco duro primario del computador utilizado para este trabajo (especificado en el Capítulo 2) de manera de poder ejecutar en el hardware real disponible todo el sistema.

Una gran ventaja de utilizar un sistema operativo dedicado es la disminución de inclusión de posibles *errores sistemáticos* al evitar usar el sistema para otras actividades no relacionadas.

b. Instalación del sistema operativo ROS, Gazebo y librerías

Para esta etapa se siguió las instrucciones presentes en el sitio web del *Laboratorio RyCh*[4] junto con las indicaciones de los sitios web oficiales de cada librería.

En gran parte de los casos, éstos ofrecían al final de la instalación mecanismos de test para comprobación de una correcta instalación. Cuando este no fue el caso, se realizó de inmediato pequeños sencillos tests con este fin.

Ilustración 17: Entorno de trabajo inicial, simulado en Gazebo

En este paso se instaló el sistema operativo ROS, los *packages* específicos de ROS para el control del robot PR2, el simulador Gazebo con sus componentes para simulación del robot PR2, la librería MoveIt, las componentes de ésta para planeamiento de trayectorias para el robot PR2, y la librería PCL en la versión 1.7 que es la utilizada por el resto del equipo de desarrollo del laboratorio, y la que se puede descargar compilada directamente desde los repositorios de Ubuntu.

[4] Sitio web: https://rych.dcc.uchile.cl/

c. *Configuración del entorno*

Teniendo instalado todo el entorno de desarrollo para el robot PR2, fue necesario aplicar configuraciones, particularmente para el simulador Gazebo, para poder definir un escenario inicial con el que trabajar que represente las cualidades del entorno con el que se esperaba hacer pruebas en el robot real. En este punto, se configuró, mediante la utilización de los archivos de lenguaje de marcado `".launch"` definir la simulación de un mundo vacío, sólo con un robot PR2 funcional. Se agregaron distintas superficies y objetos según las pruebas del programa lo ameritaran. Un ejemplo de configuración inicial se ve en la Ilustración 17.

d. *Elección de herramientas de desarrollo*

La primera elección importante fue definir el lenguaje de programación con el que trabajar. El sistema ROS ofrece tres formas de interactuar con los nodos conectados:

i. Mediante comandos de consola
ii. Mediante librerías para lenguaje Python (descartada)
iii. Mediante librerías para lenguaje C++

La primera opción no se descartó completamente, pues siempre que se quiso en general interactuar rápida y directamente con el sistema, se utilizó esta forma.

La segunda opción parecía en un principio ser la más amigable dada la conocida característica del lenguaje *Python* de agilizar el proceso de programación por sus relajadas exigencias y sencilla sintaxis. No obstante, esta alternativa fue totalmente descartada tras investigar al respecto que el soporte para interactuar con ROS a través de C++ se encuentra en mejor estado y madurez.

Una vez elegido el lenguaje C++ como principal medio de comunicación con ROS, a continuación hubo que escoger qué IDE[5] o editor de texto usar para programar. Por comodidad y mayor experiencia, el alumno memorista escogió el editor de texto *Sublime Text*[6] como el software a utilizar para esta tarea.

Se realizó un control de versiones del software utilizando la plataforma de versionamiento *Git*.

e. *Aplicación de plugin para mejor simulación de agarre en Gazebo*

La versión 1.9 del simulador es relativamente antigua respecto a la versión 7.0 actual, donde existen implementadas ciertas optimizaciones para aplicaciones de manipulación, ausentes en distribuciones anteriores. Esto se hace notar al momento de realizar *grasping* en el ambiente simulado: Al aprisionar los distintos objetos testeados entre las tenazas del robot, se observaron comportamientos problemáticos inherentes a la simulación, donde el objeto oscilaba rápidamente y resbalaba del gripper, incluso en ocasiones saltando disparado muy lejos.

Jennifer Buehler, de la *Open Source Robotics Foundation (OSRF)* a la que pertenece el proyecto Gazebo, desarrolló un plugin[7] para la versión 1.9, capaz de corregir este comportamiento,

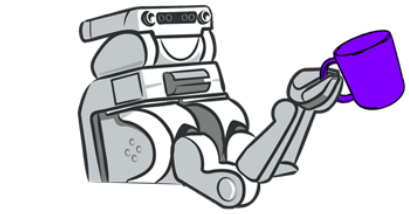
[5] "Integrated Development Environment"
[6] Sitio web oficial: http://www.sublimetext.com/
[7] Disponible en https://github.com/JenniferBuehler/gazebo-pkgs/tree/master/gazebo_grasp_plugin

y participó activamente en su mejoramiento[8] para lograr hacerlo funcionar en este proyecto en particular.

[8] Ejemplo en https://github.com/JenniferBuehler/gazebo-pkgs/issues/1

Descripción General de la Solución

Para comenzar a diseñar la solución, se abordó inicialmente el problema desde la intuición, organizando conceptualmente las etapas que llevarán al robot desde su estado inicial hasta un posicionamiento exitoso.



Estado inicial: El PR2 tiene suspendido el objeto con una de sus tenazas

Diagrama 3: Algoritmo general de placing

El Diagrama 3 grafica este flujo de procesos, destacándose en él cinco módulos principales: **"*Modelamiento del Objeto"***, ***"Cálculo de Pose Estable"***, ***"Búsqueda de Superficie"***, ***"Desplazamiento hacia Superficie"*** y ***"Posicionamiento"***. Ellos definen casi en su totalidad la implementación de la solución y son revisadas en mayor profundidad en la siguiente sección. Existe un módulo más: **"*Preparación del Robot*"**, donde se configura el sistema para dejar al robot en el estado inicial requerido. Dado que existen muchas formas de llegar al estado inicial requerido (algoritmos de grasping, entregar directamente objeto, recepción de objeto desde otro robot), este módulo no se considera como parte esencial del algoritmo.

En el contexto de lo analizado en el presente capítulo, por conveniencia el término "*módulo*" será estrictamente diferenciado del término "*componente*" o "*sub componente*", los cuales se refieren específicamente a las sub partes que constituyen cada uno de estos cinco módulos.

Descripción Detallada de Módulos

Las cinco partes principales de la solución general definen el comportamiento global del algoritmo de *placing*. Es en el proceso de creación de estos módulos en que el software principal realmente toma forma.

A continuación se analiza cada módulo, desde su diseño conceptual y el de sus sub componentes, hasta el proceso y resultado de su materialización en tareas programables. Al finalizar la construcción de cada uno de estos módulos, se construye paso a paso herramientas que serán utilizadas de manera transversal.

Es importante en este punto aclarar que lo que viene de esta sección comprende el proceso de creación de módulos hasta **antes de su integración** en el software completo, momento en que se realizó un cambio de arquitectura importante que es explicado en la sección "Integración de Módulos" dedicada para ello, presente al final de este capítulo.

Módulo 0: Preparación del Robot

Para poder empezar el programa, se exige que el robot posea previamente sujeto el objeto en una de sus tenazas. Además, *Willow Garage* recomienda subir el torso del robot al máximo cuando se esté realizando tareas de manipulación. Para hacer esto, se crea el presente módulo, que se constituye de las siguientes subcomponentes:

***Subcomponente n°1**: Inicialización*

Al preparar el entorno de trabajo, se instaló paquetes destinados a la inicialización del PR2, que se encargan, entre otras cosas, de activar los controladores de sus sensores y actuadores, crear una nueva instancia de ROS o unirse a una existente, cargar datos al servidor de parámetros de ROS, iniciar nodos, publicar y suscribirse a tópicos, y de opcionalmente inicializar el contexto MoveIt según se necesite, utilizando los archivos `".launch"`.

La configuración incluida por defecto es útil para instanciar al robot para tareas generales. Sin embargo, como se verá más adelante, se hizo necesario personalizar parámetros e inicializar componentes con configuraciones especiales, situación ante la que fue meritorio crear un conjunto de archivos `".launch"` pensados específicamente para el algoritmo de posicionamiento.

Finalmente, para subir el torso se indica en el archivo que se envíen mensajes al tópico "/torso_controller/command trajectory_msgs/JointTrajectory", al que está suscrito el controlador del torso para recibir comandos.

El resultado de este proceso fue la creación de una carpeta con copias de los archivos originales pero con modificaciones para referenciar a configuraciones particulares del programa. Un ejemplo se puede ver en la Ilustración 18, donde se habilitó la recepción de parámetros por consola para cambiar la inicialización, y se modificó una entrada que inicializaba el contexto MoveIt a partir de los datos de un archivo por defecto de la instalación.

```
<launch>
    <!-- declaro argumentos para usar dentro de las llamadas internas -->
    <param name="/use_sim_time" value="true" />
    <arg name="paused" default="true"/>
    <arg name="gazebo_models" default="/home/mexomagno/.gazebo/models"/>
    <!-- Argumentos a recibir por consola de la forma <launchfile.launch> arg:=valor -->
    <arg name="gzclient" default="true"/>
    <arg name="surface" default="mesita"/>          <!-- Para escoger superficie a usar -->
    <arg name="moveit" default="true"/>             <!-- Para iniciar con contexto moveit -->
    <arg name="rviz" default="false"/>              <!-- Para iniciar con RViz -->
    <arg name="quick" default="false"/>             <!-- Para correr mundo con parámetros rápidos -->
    <arg name="cube" default="false" />             <!-- Para spawnear cubo sobre la superficie -->
    <arg name="torso_up" default="true" />          <!-- Para subir el torso -->
                    .
                    .
                    .

    <!-- Cargar contexto moveit personalizado -->
    <!--<include file="$(find pr2_gazebo)/launch/fixed/moveit_demo.launch" if="$(arg moveit)">-->
    <include file="$(find mipr2_gazebo)/launch/fixed/moveit_demo_fixed.launch" if="$(arg moveit)">
                    .
                    .
                    .
</launch>
```

Ilustración 18: Extracto de archivo de inicialización del robot

***Subcomponente n°2**: Tele-operación de grippers*

Existe incluido en la instalación de los componentes ROS para el PR2 el módulo *Teleop* que provee de funcionalidades de tele-operación, permitiendo mover partes de robot remotamente desde el teclado, y en el caso de robot real, admite control remoto dese un joystick. Sin embargo, este control no incluye el desplazamiento de los brazos en el espacio. Esta funcionalidad es de especial urgencia cuando se trabaja con el robot simulado en Gazebo, donde ya no se le puede sencillamente entregar un objeto al robot directamente con la mano como se haría con el robot real. Por eso y dado que el problema de *grasping* no resulta trivial, se implementa un programa de tele-operación de grippers, que permite, al igual que *Teleop*, controlar remotamente su posición y orientación haciendo uso de MoveIt.

El programa permite controlar la posición de cada tenaza según el input del usuario, que puede ser cualquiera de los que se detallan en la Tabla 3.

Gracias a este programa se puede llevar ambos brazos a cualquier pose alcanzable del entorno, para poder realizar *grasping* manualmente y con la precisión que se desee.

Input	Función	Frame de referencia
q/a	Mover un paso en eje X	Odométrico
w/s	Mover un paso en eje Y	Odométrico
e/d	Mover un paso en eje Z	Odométrico
Q/A	Mover un paso en eje X	Gripper
W/S	Mover un paso en eje Y	Gripper
E/D	Mover un paso en eje Z	Gripper
i/k	*Roll* un paso en torno a X	Gripper
j/l	*Pitch* un paso en torno a Y	Gripper
1	Cambiar tamaño de paso lineal	--
2	Cambiar tamaño de paso angular	--

Tabla 3: Controles de tele-operación de grippers

Para apertura y cierre de grippers se envía directamente, desde la consola del sistema, mensajes al tópico `"/l_gripper_controller/gripper_action/goal"` para el izquierdo y `"/r_gripper_controller/gripper_action/goal"` para el derecho, que son escuchados por los controladores que ejecutan la petición. El mensaje ROS, de tipo `“prt_controllers_msgs/Pr2GripperCommandActionGoal”`, permite especificar la apertura y esfuerzo a ejercer.

Subcomponente n°3: Interacción con el Simulador Gazebo

Gazebo cuenta con la interfaz gráfica *gzclient* que provee de limitado pero intuitivo control sobre los objetos simulados. Convenientemente, el simulador inicializa ciertos servicios ROS destinados a acceder a información de la simulación, lo que otorga la base para generar un conjunto de scripts personalizados, encargados de acortar repetitivas llamadas a estos servicios necesarias cada vez que se inicia la simulación. Notar que sólo resultan útiles en el caso del entorno simulado y no cuando se trabaja con el robot real.

Los scripts y sus funciones son resumidos en la Tabla 4.

Nombre	Argumentos	Efecto
gz-spawn	(nombre del objeto[:alias opcional] (x y z)	Agrega un nuevo objeto al entorno en la posición especificada, relativa al frame odométrico. Si se ingresa alias, se le asigna como nombre.
gz-put	(nombre del objeto) (x y z)	Mueve un objeto existente a la posición especificada
gz-put-on-table	(nombre del objeto)	Posiciona un objeto existente sobre la superficie actual
gz-spawn-on-table	(nombre del objeto [:alias opcional] (x y z)	`gz-spawn` seguido de `gz-put-on-table`
gz-which-models	--	Retorna los nombres de los modelos actualmente cargados en la escena
gz-valid-models	--	Muestra los nombres de los modelos soportados para `gz-spawn`
gz-delete	(nombre del objeto)	Elimina un objeto de la escena, si existe

Tabla 4: Scripts para interacción con Gazebo

¿Para qué existe `”gz-valid-models”`? Gazebo 1.9 posee una librería de modelos de objetos. Algunos de ellos no poseen descripciones de inercia sin las cuales la simulación en esta versión muestra serios problemas. Este script se preocupa de listar sólo los nombres de los modelos que contienen correctamente configurada esta componente.

Módulo 1: Modelamiento del Objeto

En esta primera instancia se intenta obtener una representación digital del objeto manipulado, del que no necesariamente se conocen sus características físicas y mecánicas con anterioridad.

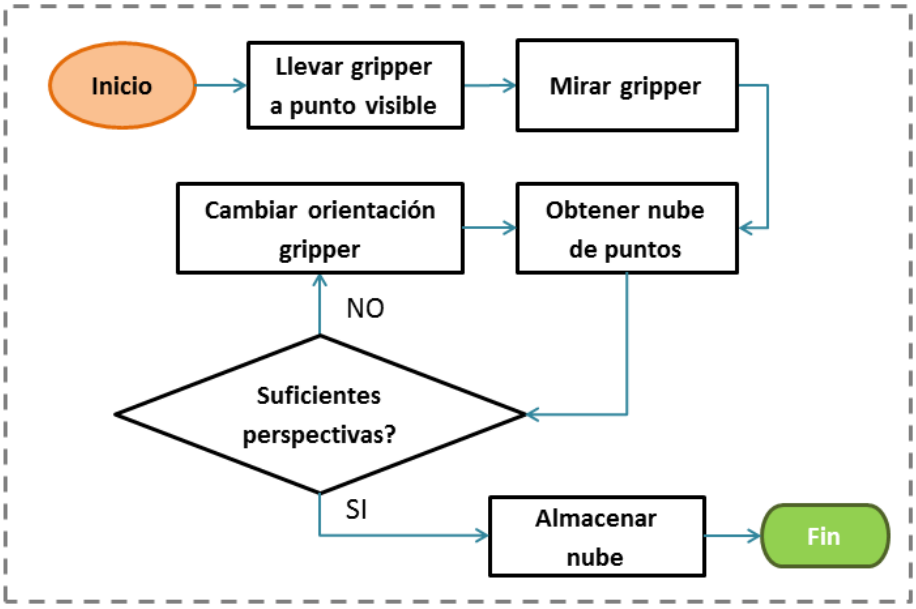


Diagrama 4: Flujo de procesos del Módulo 1

Inicialmente, por conveniencia, se lleva el brazo no utilizado a un costado del robot para evitar que su presencia entorpezca tanto el movimiento del otro brazo como el campo visual de adquisición. Luego, se configura al robot para una óptima observación del objeto, llevando el gripper activo al frente del sensor Kinect. Se procede después a obtener iterativamente capturas de nubes de puntos variando entre cada una de ellas la posición de la tenaza, de modo de poder guardar distintas perspectivas que permitan crear una imagen tridimensional lo más completa posible del objeto. Es conveniente guardar estos datos relativos al frame del gripper activo, y no respecto al frame del Kinect. Para esto, se pidió a TF en cada iteración la transformación necesaria para cambiar este sistema de referencia y convertir los datos de manera acorde. También es de utilidad aplicar un submuestreo del total de los puntos recibidos, para disminuir la carga computacional asociada al procesamiento de estos datos. Empíricamente se comprobó que con no más de un 2% del total de puntos adquiridos es posible generar una buena representación del objeto. Este submuestreo se hace usando el módulo *Filters* de PCL.

Cuando este proceso termina (ver Diagrama 4) y se guardaron suficientes puntos de vista, se acumula estas capturas en una sola nube de puntos que representa al objeto. El número exacto de perspectivas a capturar del objeto y sus respectivas poses son totalmente libres. En esta etapa de desarrollo se consideró cuatro vistas laterales, una superior y cuatro inferiores (ver Ilustración 19), esto último porque ahí es donde se ubica el gripper y, naturalmente, bloquea parte de la visibilidad.

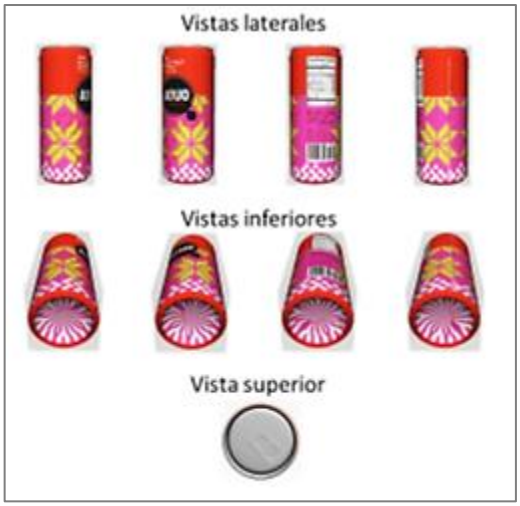


Ilustración 19: Esquema de perspectivas consideradas para modelamiento

***Subcomponente n°1**: Motricidad*

Para construir este primer módulo encargado del modelamiento del objeto, fue necesario crear dos funciones relevantes de motricidad:

Motricidad de cabeza

Para poder observar el entorno con la Kinect conectada a la cabeza del robot, éste debe poder moverla acorde. Para esto, se aprovechó la existencia del módulo *ActionLib* de ROS. Esta librería está diseñada para ser utilizada por todo par de nodos con roles de tipo cliente-servidor, donde el cliente necesite llevar un seguimiento del estado de la petición enviada al servidor.

El robot implementa un *ActionServer* que escucha instrucciones para ejecutar distintas acciones. Uno de los módulos internos se encarga de interpretar mensajes ROS dirigidos al control de la cabeza. Esta interfaz provee de retroalimentación sobre el estado de la petición y el avance de la tarea.Visto desde la perspectiva de ROS, el *ActionServer* inicializa nodos en el ambiente ROS, cada uno de ellos suscribiendo y publicando a varios tópicos. En particular, es el servicio `"/head_traj_controller/point_head_action"` el encargado de recibir peticiones de movimiento de cabeza, ejecutarlas e informar su estado.

El desarrollo de esta componente llevó a la creación del servicio ROS personalizado ***LookAt*** encargado tanto de transparentar el uso del servicio de *ActionLib*, como de otorgar funcionalidades de más alto nivel que las que admite la interfaz nativamente.

Motricidad de brazos

Es la librería MoveIt la encargada de planear y ejecutar toda la cinemática inversa implicada para llevar un gripper desde una posición a otra de manera segura. Utilizando la interfaz `"planning_interface"` se implementó, al igual que para el caso de la subcomponente de motricidad de cabeza, el servicio ROS ***MoveArm***, que administra las configuraciones de la interfaz y provee de utilidades de alto nivel. El funcionamiento de este servicio se implementó aparte de la componente de tele-operación de grippers, pues para el caso de uso de este módulo, el acceso se hace de forma programática.

El contexto MoveIt mantiene constantemente actualizado su modelo interno del mundo exterior, alimentando cada uno de los elementos descritos en la Tabla 1. La actualización del octomap supone un problema en este punto, pues, cuando el robot observa el objeto, los puntos de la nube que no pertenecen a partes del robot, incluidos los del objeto, son considerados como puntos de colisión y se agregan al octomap. Esto hace que el gripper quede atrapado por esta nube de voxels y sea muy difícil para el planeador de trayectorias lograr calcular alguna forma de sacarlo de ahí, como muestra la Ilustración 20 en (a).

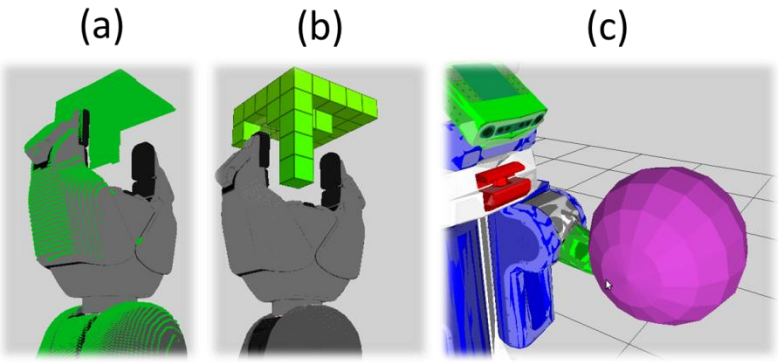


Ilustración 20: Gripper con cubo de madera. En (a), perspectiva de kinect. En (b), voxels generados por MoveIt. En (c), gripper cubierto por Collision Object Esférico.

Para resolver esta limitación, se ideó la creación de un Attached Collision Object esférico centrado en el punto de agarre del gripper activo, suficientemente grande para circunscribir al objeto manipulado y asociado al link principal del gripper, el `"gripper_palm_link"` ,como se grafica en la Ilustración 20 (c). Con esta solución, moveit no genera voxels que interrumpan un correcto desplazamiento del brazo robótico. En la práctica, MoveIt a veces no limpia inmediatamente los voxels encerrados en un Collision Object. Esto es un problema conocido[9], pero que aparece con poca frecuencia.

En la creación del servicio *MoveArm* se decide utilizar el Planner *"RTTConnect"*, elección que se puede justificar observando las métricas presentes en el Anexo A.

***Subcomponente n°2**: Percepción*

Generar un modelo del objeto observado aislado del resto de la escena involucra poder tener algún mecanismo de filtrado que permita quitar todo punto no perteneciente al objeto mismo, incluyendo a los que corresponden al gripper que lo sostiene. Para esto, se evaluaron las siguientes opciones:

1) ***Utilizar el paquete* Robot_self_filter*:*** Los desarrolladores de ROS incluyen en las versiones actuales del sistema el paquete *Robot_self_filter*, que cumple con la función de auto-filtrado del robot. Desafortunadamente, el soporte para este paquete quedó incompleto para ROS Hydro y prontamente fue deprecado.
2) ***Implementar un filtro propio*:** Se optó en una primera instancia por esta opción, que llevó a la creación de una versión rudimentaria del auto-filtro buscado. Sin embargo, los resultados

[9] Detalles del problema se encuentran en https://github.com/ros-planning/moveit_ros/issues/315

no fueron del todo satisfactorios y se pudo implementar la opción 3. Los resultados del software creado y deprecado para este punto se pueden encontrar en el Anexo B.

3) ***Utilizar auto-filtro de MoveIt (opción elegida)*:** Esta librería implementa internamente un auto-filtro del robot, capaz de quitar de las nubes de puntos crudas adquiridas desde el kinect todos los puntos suficientemente cerca de los links del robot. Esto se hace con el fin de poder generar el octomap de colisiones. El resultado de esta operación es la generación de una versión auto-filtrada de esta nube de puntos, que es convenientemente publicada por MoveIt a un tópico ROS al que cualquier nodo se puede suscribir. Basta entonces con crear un nodo que consuma datos desde este tópico, en vez del que publica directamente el sensor.

La opción escogida presentó en su desarrollo ciertos desafíos a superar:

Estrecha relación con octomap

Como ya se dijo, MoveIt genera esta nube filtrada para poder generar el octomap de colisiones. Cada vez que un punto aparece en esta nube, éste es considerado para la generación de voxels y es guardado como un punto de colisión en el mundo. El problema de este comportamiento es que el octomap no es actualizado inmediatamente según los datos recibidos desde la nube filtrada, sino que se mantienen por cierto tiempo, con lo que al querer observar los puntos pertenecientes al objeto manipulado se está al mismo tiempo entregando a MoveIt la información de que se está ante un espacio de abundantes sectores que hay que evitar pasar a llevar. Esto es solucionado por la componente n°1 de movilidad al implementar el servicio MoveArm con Attached Collision Objects.

Margen de error del auto-filtro

Para generar una nube auto-filtrada, MoveIt analiza las posiciones de cada link en el espacio y sus dimensiones, para luego restar del total de puntos todos aquellos que quedaran inscritos por estas regiones. Como el sensor Kinect no es perfecto, la librería se preocupa de definir un margen de error denominado *padding* que hace que se tome en cuenta las regiones ocupadas por cada link, más un volumen adicional a prueba de ruido en los datos. Este margen es ajustado por defecto a un valor de 10 cm que resulta en la remoción de una porción notable de puntos que corresponden al objeto de interés (Ilustración 21). Afortunadamente, este valor es ajustable a través del servidor de parámetros, junto con otras constantes relacionadas. En la Ilustración 22 se pueden ver los resultados de variar dichos ajustes hasta lograr un equilibrio entre filtrar correctamente las partes del robot y adquirir la mayor cantidad de puntos pertenecientes al objeto. El mejor valor empíricamente encontrado para este parámetro fue de 0.9 cm.

Ilustración 21: Resultado de usar auto-filtro de MoveIt con valores por defecto. A la izquierda, la situación simulada. Al medio, la nube de puntos cruda capturada por el Kinect. A la derecha, la versión auto-filtrada con padding de 10cm.

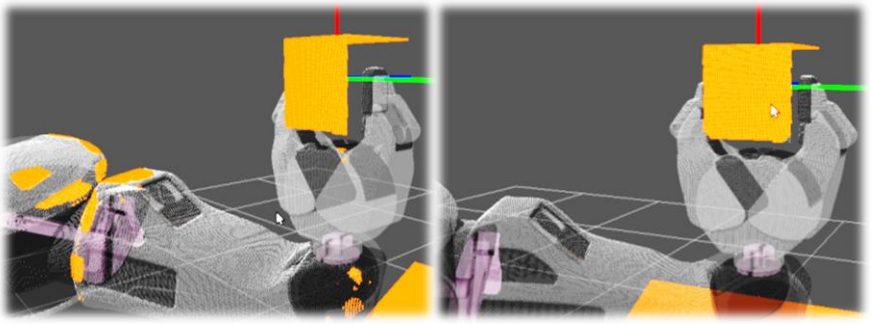


Ilustración 22: Variando ajustes de auto-filtro. A la izquierda, padding demasiado bajo provoca aparición de partes del brazo en la nube. A la derecha, el filtro está correctamente configurado.

Tan solo resta eliminar el resto de los puntos que con alta probabilidad no pertenecen al objeto. Para este trabajo se creó una primera versión del filtro encargado de este paso, que utiliza simples criterios de distancia de puntos hacia el gripper. Como se sabe que no se admitirán objetos más grandes que 40 cms, es seguro remover todo punto que sobrepase esta distancia, sin olvidar la naturaleza controlada del entorno de trabajo. Más adelante, en la sección de Trabajo Futuro, se discute sobre posibles refinamientos a esta fase de filtrado.

Todo el trabajo relativo a este primer módulo lo realiza el Algoritmo 1.

Algoritmo 1: Modelamiento de Objeto

```
INPUT:
        - Nube de puntos auto-filtradas de MoveIt
        - Gripper activo (saber si se usa el izquierdo o el derecho)
OUTPUT:
        - Nube de puntos con representación 3D del objeto
DEFINICIÓN:
        1.      Si el otro gripper estorba, mover a un lado (servicio MoveArm)
        2.      Mover gripper activo a pose de observación (servicio MoveArm)
        3.      Mirar hacia el gripper (servicio LookAt)
        4.      while (queden perspectivas por adquirir)
                 4.1.    Adquirir nube de puntos auto-filtrada desde tópico MoveIt
                 4.2.    Obtener transformación desde frame de Kinect a frame del gripper
                 4.3.    Eliminar puntos muy lejanos
                 4.4.    Submuestrear
                 4.5.    Transformar nube de puntos a frame del gripper
                 4.6.    Guardar nube
                 4.7.    Posicionar gripper en siguiente perspectiva
        5.      Unir capturas en una sola nube de puntos
        6.      return nube del objeto
```

Es importante observar que este algoritmo no considera en primera instancia el ruido de los datos adquiridos por el sensor real que traen asociado un grado de ruido. Esto puede afectar en el resultado final de la suma de nubes de puntos. En una etapa inicial donde se utiliza el simulador, esto no es un problema, pero al momento de trabajar con datos reales, se espera poder aplicar, específicamente entre los pasos 4.5 y 4.6 del Algoritmo 1, una etapa de refinación de los datos usando filtros de PCL existentes.

La arquitectura de software generada vista por ROS se representa en el Diagrama 5.

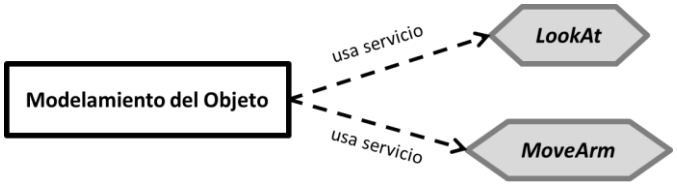


Diagrama 5: Perspectiva de ROS del módulo 1

Módulo 2: Cálculo de Pose Estable

Este es el módulo encargado de realizar la labor más importante en nuestro algoritmo de placing. Es aquí donde se obtiene la noción de cómo debiera posicionarse el objeto sujeto por la tenaza del robot. El Diagrama 6 muestra que el proceso es conceptualmente sencillo, pero cada uno de estos pasos requiere de suficiente cuidado en su diseño de modo de alcanzar soluciones razonables.

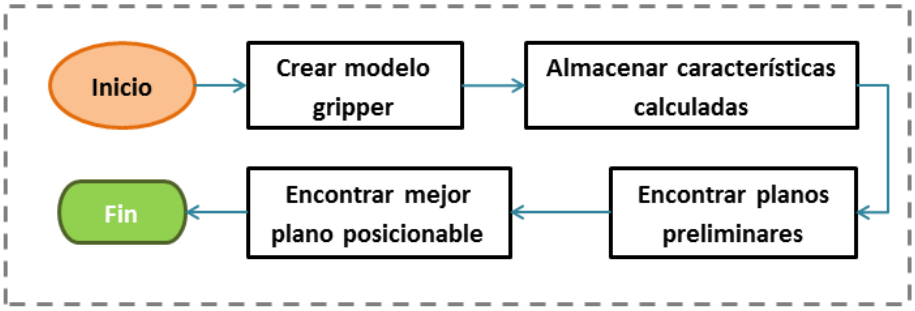


Diagrama 6: Flujo de procesos para el Módulo 2

En este punto, se posee una representación tridimensional aislada del objeto en estudio. En el proceso de cálculo de sus propiedades y de sus posibles poses estables, no se puede ignorar el hecho de que está siendo sujetado por el gripper, y es éste el que debe ser capaz de posicionarlo. Un objeto por sí solo puede tener muchas poses estables que pasan a ser imposibles de alcanzar por el gripper según su agarre, pues implicarían por ejemplo tener que ser capaz de atravesar la superficie de apoyo. Es por esto que se crea inicialmente un modelo aproximado de las dimensiones de la tenaza activa para ser incluida como variable a considerar al momento de evaluar la factibilidad de una pose estable.

Después el programa hace un análisis de la nube de puntos, de modo de poder extraer de ella información relevante al algoritmo, como su centro de masa, envoltura convexa, representación triangulada, bounding box, entre otras. Haciendo uso de estas características es que se puede pasar a la obtención de potenciales superficies de apoyo estable, que finalmente son reducidas según la factibilidad del gripper para poder de hecho llevar al objeto a estas poses, y son priorizadas además según el tamaño de área de apoyo, encontrando al final de este proceso el mejor plano de posicionamiento.

Aprovechando el procesamiento anterior, se refina la representación del objeto como un collision object en el entorno de colisiones de MoveIt. Ya no se envuelve simplemente al objeto en una bola que lo contenga, sino que ahora se crea una malla basada en la geometría real observada.

***Subcomponente n°1**: Modelamiento del gripper*

Empíricamente se observó las dimensiones ocupadas espacialmente por el gripper cuando está cerrado y abierto, pudiendo obtener así una aproximación de su constitución física. Por simplicidad se decidió representar al volumen del gripper como un conjunto de cuboides superpuestos, como se puede ver en la Ilustración 23.

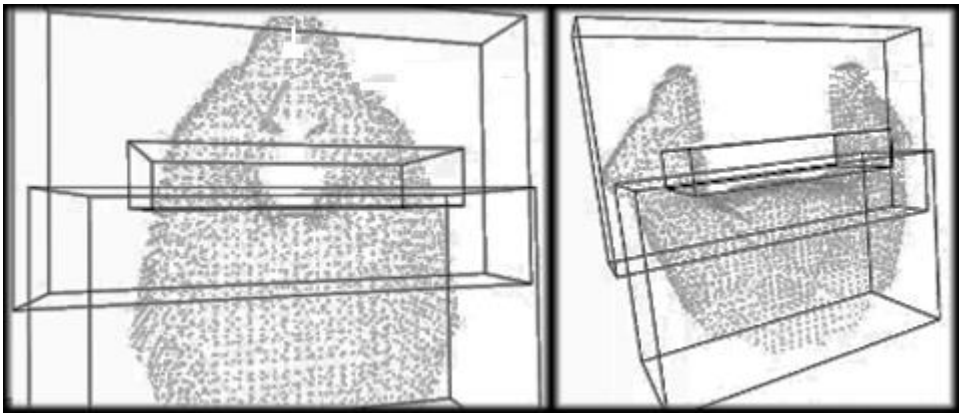


Ilustración 23: Visualización de cajas circunscritas al gripper

Este modelo es de importancia en la parte final de implementación de este módulo para la reducción del espacio de posibles posiciones estables encontradas.

***Subcomponente n°2**: Detección de superficie de apoyo*

En esta subcomponente se abordan en conjunto el resto de las tareas del módulo, logrando como resultado la implementación completa del algoritmo responsable de detectar zonas de apoyo estables para el objeto.

¿Cómo se analizó el problema?

En la primera gran parte de lo que sigue se analiza esta situación para un objeto aislado, es decir, olvidándose de la existencia del gripper y del resto del entorno en general. Al final del desarrollo se incluye este factor apropiadamente.

En primera instancia se debe construir una definición formal sobre qué determina que cierta porción de superficie de un objeto constituya ciertamente una superficie de apoyo estable, sobre la cual poder crear el software.

Basta un sencillo razonamiento para poder hacer la afirmación siguiente:

Definición 1*: Un conjunto de puntos de la superficie del objeto definen una* zona de apoyo estable *si se cumplen dos condiciones principales:*

Condición 1) Todos los puntos se ubican cerca de un mismo plano en el espacio

Condición 2) La proyección del centro de masa de los puntos sobre este plano queda encerrado por la envoltura convexa bidimensional de la proyección de los puntos sobre dicho plano.

Desde la intuición se puede rápidamente coincidir con la afirmación. Se analizará a continuación en detalle el raciocinio detrás de las dos condiciones enunciadas, de forma independiente, explicando cómo cada uno da origen a sus implementaciones computacionales.

Condición 1) "*Todos los puntos se ubican cerca de un mismo plano en el espacio*"

Ilustración 24: Objeto resultante de sustraer cilindro a una esfera.

Lo primero que se debe considerar es el hecho de que las características geométricas del objeto definen en gran medida cómo analizar este punto, en particular su **concavidad** o **convexidad**. Una idea simplista de búsqueda de superficie de apoyo plana puede ser sencillamente mirar el objeto en busca de la mayor superficie plana. Sin embargo esto está lejos de ser el acercamiento correcto.

Para esclarecer esta idea, considérese el escenario de la Ilustración 24, que muestra un cuerpo geométrico resultante de sustraer una porción cilíndrica a una esfera. En este caso, claramente la mayor (y única) zona plana en la superficie del objeto, se encuentra en la concavidad (resaltada en rojo), no obstante, ésta evidentemente no constituye una superficie de apoyo válida en una superficie plana, como una mesa o velador. La real zona de apoyo estable para este objeto es la región *tangente* a la figura, donde comienza la concavidad (malla transparente cuadriculada).

Este análisis otorga la base para poder comprender que lo que buscamos es **toda superficie tangente al objeto en tres o más puntos no colineales**. Desde esta aseveración extraemos que para esta etapa toda concavidad del objeto puede ser ignorada, y sólo resulta de interés trabajar con su componente convexa. En este punto se decide entonces utilizar la noción de e*nvoltura convexa* del objeto, explicada en el Capítulo 2, que ofrece una herramienta ideal por constituir exactamente la representación más cercana del objeto, sin sus concavidades.

Por simplicidad, considérese un espacio bidimensional. Proyéctese además un perfil de la figura anterior sobre este plano. El resultado es lo que se observa en la Ilustración 25 a la izquierda. A la derecha de la misma, se muestra la figura original en línea punteada y su envoltura convexa, donde la zona representante de la superficie de apoyo en este espacio se puede apreciar en su parte inferior en color naranja, que cumple claramente con ser tangente al objeto y se alinea perfectamente con una recta, como la representada en azul. Extendiendo este ejemplo al caso tridimensional de la Ilustración 24, se puede ver cómo su envoltura convexa definirá un plano en el agujero (en la figura, el plano cuadriculado), que puede ser alineado con un plano en el espacio.

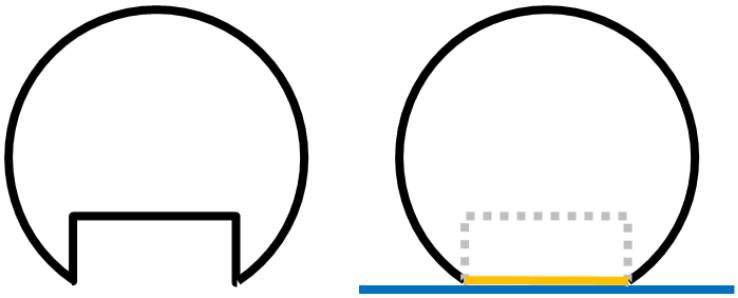


Ilustración 25: Perfil del objeto. A la izquierda, el perfil original. A la derecha, en naranja, su envoltura convexa y en azul una superficie plana de apoyo alineada con la cara plana de la envoltura.

Un caso más irregular se puede apreciar en la Ilustración 26, donde el mismo principio se cumple. Se aprecian en color naranja las líneas rectas que muestran sus potenciales rectas de apoyo.

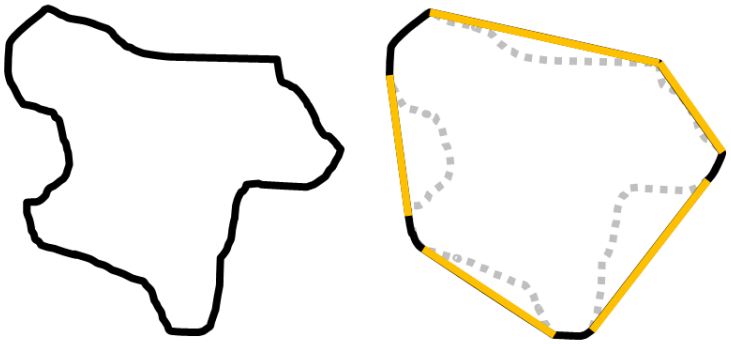


Ilustración 26: Objeto amorfo. A la izquierda, forma original. A la derecha, su envoltura convexa.

Obtención de la Envoltura Convexa

La librería PCL provee de un método para calcular la envoltura convexa de una nube de puntos tridimensionales, que retorna un objeto que contiene una estructura interna simple conformada por una lista con los triángulos que la constituyen, como se muestra en el Diagrama 7.

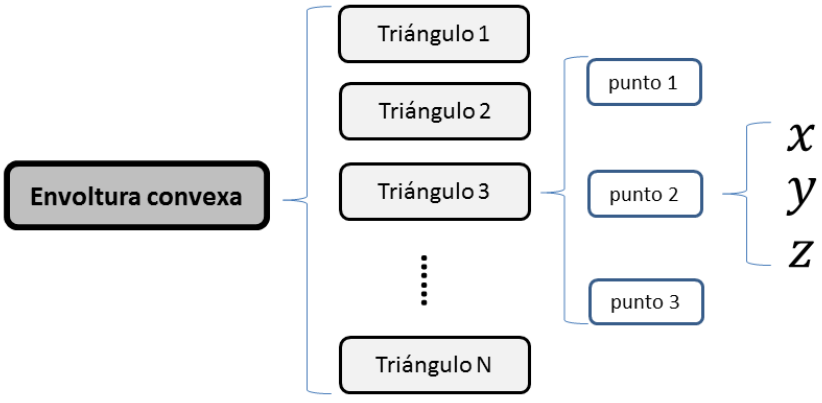


Diagrama 7: Representación interna de envoltura convexa de librería PCL.

Primera Recapitulación

Hasta aquí se ha definido el marco con el cual empezar a desarrollar el algoritmo de obtención de zonas de apoyo. Para comenzar, se debe atender a la siguiente interrogante:

¿Cómo se define una superficie plana, usando las estructuras de datos con las que se cuenta?

Para responder esto, se debe tener en cuenta que se trabajará con datos adquiridos desde el sensor Kinect, factor que, por la construcción de este dispositivo, potencialmente añade errores en los datos. Incluso en el caso ideal en que se tuviera la representación de un objeto 3D con una superficie perfectamente lisa, en la realidad su interpretación en nube de puntos puede poseer ciertas irregularidades. Es por esto que se debe admitir un grado de relajación.

Con esto y teniendo clara la estructura de la envoltura convexa con que se trabaja, se puede definir una superficie plana como sigue:

Definición 2*: En el presente contexto, una superficie plana se definirá como un conjunto de triángulos contiguos y de inclinaciones suficientemente similares.*

Para interpretar esto y llevarlo a algo tangible, se debe poner atención a cada parte del anterior enunciado:

a) Triángulos contiguos:

La estructura de datos que entrega la librería PCL no establece una relación entre los triángulos de la envoltura convexa, por lo que directamente no se tiene una noción de su contigüidad. Para resolver este punto se proponen dos acercamientos:

1) ***Creación de una nueva estructura de datos a partir de la original***, que relacione cada triángulo con sus vecinos directamente colindantes. Esto permitiría recorrer las vecindades de cada triángulo fácilmente.
2) ***Aprovechar la propiedad de convexidad (opción escogida)***. Se puede observar la siguiente propiedad:

Propiedad*: Considérese un objeto tridimensional convexo O, definido por el conjunto de triángulos T_O, todos con vectores normales correctamente orientados hacia afuera. Si $\hat{n}_t$ simboliza el vector normal del triángulo $t \in T_O$, entonces se tiene que:*

Sea un $t \in T_O$, y sea S_t^θ: $\{t' \in T_O \mid \acute{a}ngulo\ entre\ \hat{n}_t\ y\ \hat{n}_{t'} < \theta, \theta < \pi\}$ (claramente $S_t^\theta \subseteq T_O$). Entonces todos los triángulos de S_t^θ están conectados entre sí.

Para convencernos de esta propiedad, asumamos que la propiedad es falsa para un S_t^θ donde se tienen N grupos de triángulos no conectados. Si t_0 es un triángulo de un grupo y t_1 es uno del otro grupo, se tiene que el ángulo ϕ entre $\hat{n}_{t_o}$ y $\hat{n}_{t_1}$ cumple $\phi < \theta$. Por convexidad de O, todos los triángulos entre t_0 y t_1 poseen normales que varían suavemente entre $\hat{n}_{t_o}$ y $\hat{n}_{t_1}$, por lo que los grupos están conectados. Esta propiedad permite no tener que rehacer completamente la estructura de los triángulos como lo propone la alternativa 1), por lo que fue ésta la opción escogida.

b) Inclinaciones suficientemente similares

La noción de inclinación utilizada para los triángulos se hará tomando el vector normal a su plano, posicionado en su centroide (Ilustración 27). Con este concepto es posible, con conceptos simples de álgebra lineal, calcular cómo se diferencian angularmente las inclinaciones entre distintos triángulos en término de sus vectores normales. El único punto delicado restante para completar este acercamiento es la orientación de estos vectores que, dada la naturaleza desordenada de la estructura con la que se trabaja, no son necesariamente coherentes con el volumen que encierra la envoltura convexa. Esto es resuelto por el Algoritmo 2.

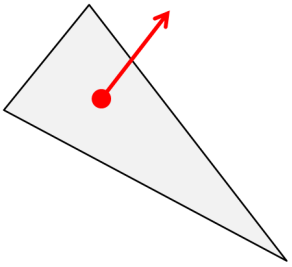


Ilustración 27: Vector normal de un triángulo, posicionado en su centroide.

Algoritmo 2: Reorientación de normales

```
INPUT:
        - Listado de triángulos de objeto convexo
OUTPUT:
        - Listado de normales para cada triángulo, orientada hacia fuera.
DEFINICIÓN:
        1.      Calcular centroide C de todo el objeto
        2.      Inicializar lista vacía de normales LN
        3.      for (cada triángulo en objeto):
                3.1.    Calcular Centroide CT del triángulo
                3.2.    Obtener vector V = C-CT
                3.3.    Calcular normal del triángulo N
                3.4.    if (ángulo entre N y V > PI/2):
                    3.4.1.              invertir N (N*=-1)
                3.5.    Añadir N a LN
        4.      return LN
```

El centroide C de un objeto convexo siempre estará dentro de él, por lo que el vector diferencia V entre C y el centroide CT de un triángulo invariablemente apunta hacia afuera del objeto. De esta manera, siempre se cumplirá que el ángulo θ entre el vector normal N del triángulo y V será menor a $\pi/2$ cuando esté bien orientada, y mayor en caso contrario.

Segunda Recapitulación

Hasta aquí se ha construido lo que se necesita para la obtención de zonas suficientemente planas. Este proceso es llevado a cabo por el Algoritmo 3. Aquí se introduce el concepto de ***parche***, que hace alusión a la naturaleza de la definición de "superficie plana" que se hizo previamente en el presente análisis, la que habla de conjunto de triángulos contiguos, una especie de "parche" de la envoltura convexa.

Algoritmo 3: Búsqueda de parches planos

```
INPUT:
        - Listado de triángulos de objeto convexo
        - Listado de normales de los triángulos, correctamente orientadas hacia afuera.
        - Angulo de relajación
OUTPUT:
        - Listado de parches planos, ordenados de mayor a menor área
        - Listado de áreas totales de parches, ordenadas de mayor a menor
DEFINICIÓN:
        1.      Crear listado vacío de parches P
        2.      Crear listado vacío de áreas A
        3.      for (triángulo T):
                3.1.    Crear parche para T
                3.2.    Añadir T a su parche
                3.3.    for (TR := el resto de los triángulos):
                    3.3.1.      Obtener ángulo entre normal propia y la de TR
                    3.3.2.      if (ángulo < ángulo de relajación):
                            3.3.2.1.    Añadir TR al parche
                3.4.    Añadir parche a P
                3.5.    Calcular área del parche y añadirla a A
        4.      Ordenar P según áreas
        5.      Ordenar A
        6.      return P
        7.      return A
```

El algoritmo busca para cada triángulo una vecindad de triángulos (un "*parche*") de similar inclinación, cuya diferencia no sea más grande que cierto ángulo de relajación. Se define un contenedor vacío P para todos los parches que se generarán. Paralelamente se aprovecha el ciclo de construcción de parches para almacenar además el área total del parche en un contenedor separado A. Para cada parche P_i se almacena su área total en A_i. Al final del proceso se ordena P y A usando como criterio el tamaño de área mayor. Esto para facilitar posteriormente la selección del mejor parche. En el punto 3.1 se ve que se crea un nuevo parche para cada triángulo T, sin verificar si este ya pertenece a otro parche. Este comportamiento se justifica por el hecho de que, si bien T puede pertenecer a otro parche, puede estar ubicado en una región cercana su perímetro. Es necesario considerar además el parche que considera a T como su centro. Más adelante en la sección Trabajo Futuro se discute algunas modificaciones a este algoritmo.

Con todo lo visto en esta parte, se ha podido resolver los desafíos planteados por el primer requerimiento surgido al concebir la Definición 1.

Condición 2) "*La proyección del centro de masa de los puntos sobre este plano queda encerrado por la envoltura convexa bidimensional de la proyección de los puntos sobre dicho plano*"

Esta afirmación es respaldada por la conocida definición mecánica de estabilidad de un objeto, que involucra las dimensiones de la base de apoyo del objeto y su centro de masa. En la Ilustración 28 se aprecia en el caso 1 un objeto posicionado de manera estable sobre una superficie. Al aplicar una fuerza, el objeto pasa al estado del caso 2 donde se ve que su centro de masa sigue al mismo lado de su pivote de apoyo. El caso 3 muestra que, dadas las fuerzas presentes, irremediablemente el bloque se volcará.

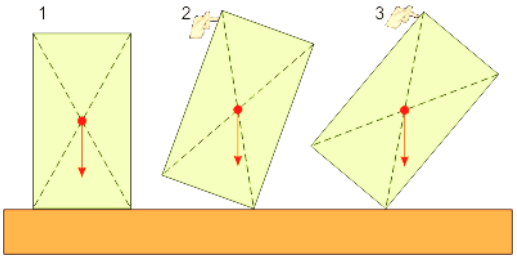


Ilustración 28: Centro de masa como descriptor de estabilidad.

El primer elemento relevante descrito en la afirmación

es el *Centro de Masa* de un objeto. La definición formal del centro de masa para un elemento discreto es como sigue:

Definición 3: *Cálculo de centro de masa para sistemas discretos*.

Sea un sistema de masa total $\boldsymbol{M}$*, compuesto por* $\boldsymbol{n}$ *partículas*

Sea $\vec{r_i}$ *la posición de una partícula del sistema y* $\boldsymbol{m_i}$ *su masa asociada.*

Se define el centro de masa $\vec{\boldsymbol{R}}$ *del sistema como*:

$$\vec{R} = \frac{1}{M}\sum_{1}^{n} m_i \vec{r_i}$$

Para intentar aplicar esta definición, observar que el sistema con el que se trabaja aquí es la nube de puntos que representa al objeto. Lo primero que hay que ver es que esta representación no considera las partículas internas del mismo, que son invisibles para el Kinect, sino que sólo implica a un subconjunto de puntos ubicados en su superficie. Con esto claro, se proponen los siguientes dos acercamientos:

a) ***Asignar a cada punto masas m iguales y aplicar la fórmula.*** Este acercamiento simplista puede entregar resultados útiles en algunos casos simples, donde su superficie es mayormente convexa y la adquisición de puntos los distribuye uniformemente sobre su superficie. Claramente este no será el caso para la mayoría de los escenarios. Se debe estudiar una nueva alternativa

b) ***Considerar puntos de peso ponderado según estructura triangulada (opción elegida).*** Una triangulación del objeto permite tener una medida del área ocupada por cada uno de estos en su superficie. Con esta medida se puede entonces realizar una ponderación del peso que debiese representar cada punto de la nube, eliminando finalmente el factor de heterogeneidad en la distribución de los puntos sobre la superficie.

Para obtener una triangulación del objeto, basta con utilizar el módulo *Surface* de la librería PCL que se encarga de esto. Esta nueva malla de polígonos no debe ser confundida con la que se ha mencionado hasta este punto, que correspondía a la representación de la envoltura convexa y no a la geometría real de todo el objeto. La implementación del punto b) se define en el Algoritmo 4.

Algoritmo 4: Cálculo de centro de masa estimado

```
INPUT:
        - Representación en nube de puntos del objeto
OUTPUT:
        - Posición de centro de masa
DEFINICIÓN:
        1.      Obtener triangulación del objeto
        2.      Inicializar vector CM = [0 ,0 ,0]
        3.      Inicializar áreas totales AT = 0
        4.      for (cada triángulo):
                 4.1.    Calcular área A del triángulo
                 4.2.    Calcular centroide C de triángulo
                 4.3.    Crear vector V = A*C
                 4.4.    Suma vectorial, CM = CM + V
                 4.5.    Acumular áreas, AT = AT + A
        5.      CM = CM/AT
        6.      return CM
```

Teniendo a disposición un mecanismo para obtener el centro de masas, se termina de cubrir los requerimientos de la Definición 1. Ahora se debe ver la siguiente parte, que tiene que ver con la proyección del centro de masas sobre el parche plano encontrado.

Se sabe por construcción que el parche se constituye potencialmente de un conjunto de triángulos contiguos de similar **pero no igual** inclinación. Es obligatorio entonces desarrollar una forma de proyectar el centro de masas sobre los parches candidatos. Para lograr esto, se plantean los siguientes conceptos:

"Aplanamiento" de parches:

Se hace necesario obtener una representación aproximada del plano formado por el parche en el caso de ángulos de relajación suficientemente pequeños. Para encontrar los parámetros del modelo plano que mejor se ajuste a los puntos que componen a cada parche, se utiliza una metodología centrada en un algoritmo de *segmentación* que, a través de la aplicación de una estrategia RANSAC, entrega como resultado los coeficientes del modelo encontrado, pudiendo así pasar de tener un parche levemente irregular a un plano tridimensional perfecto.

"Perímetro" de parches:

Se desea poder trazar sobre el plano encontrado un polígono que delimite el área representada por el parche imperfecto. Para esto, se decide realizar un proceso de dos etapas:

1. Se realiza una proyección de todos los puntos del parche sobre el plano encontrado en 1.
2. Se obtiene la *envoltura convexa bidimensional* de dichos puntos

El resultado de esto es exactamente el perímetro de la proyección de todo el parche sobre su plano aproximado.

El Algoritmo 5 recibe un parche y, con las estrategias ya explicadas, entrega su versión aplanada.

Algoritmo 5: Aplanamiento de parche

```
INPUT:
        - Parche de triángulos contiguos de similar inclinación
OUTPUT:
        - Perímetro plano del parche (puntos)
DEFINICIÓN:
        1.      Obtener conjunto P de puntos del parche
        2.      Obtener Bounding Box BB de P
        3.      Configurar algoritmo de segmentación para incluir inliers no más lejanos que la menor
        componente de BB
        4.      Aplicar segmentación para búsqueda de plano, algoritmo RANSAC
        5.      Proyectar puntos sobre plano y obtener puntos Q
        6.      Obtener envoltura convexa 2D de puntos, obteniendo perímetro R
        7.      return R
```

PCL, con su módulo *Segmentation*, permite la aplicación del algoritmo RANSAC que busca el plano. Eso sí, para su funcionamiento requiere de parámetro la distancia máxima a la que un punto puede estar del plano evaluado para ser elegido como un inlier, por lo que se necesita tener una noción de la extensión de los puntos del parche en el espacio. Es por esto que se comienza obteniendo el *Bounding Box* del conjunto de puntos *P*. Esto entrega una aproximación del parche en forma de una caja tridimensional. Si para encontrar el parche se escogió previamente un ángulo de relajación adecuado, entonces se tiene que la distancia máxima posible de un punto en *P* a su plano representado está dada por la menor dimensión del Bounding Box (Ilustración 29).

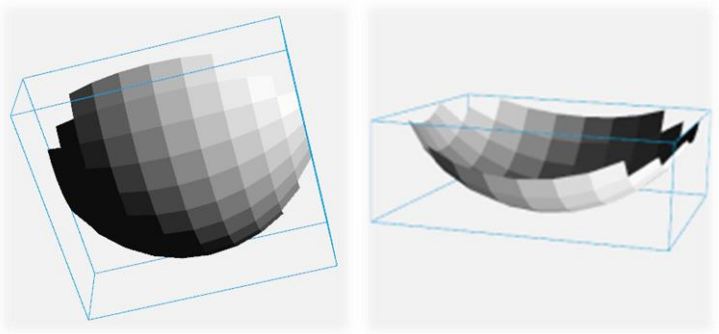


Ilustración 29: Bounding Box de un parche. Recreación de un parche construido con un ángulo de relajación notoriamente alto.

El algoritmo de ajuste de modelo plano a los puntos de *P* entrega los coeficientes del plano que representa, los que son entregados como parámetro a la funcionalidad de PCL encargada de proyectar puntos sobre planos.

Finalmente, teniendo una versión plana de todos los puntos del parche, se obtiene el perímetro usando la técnica de obtención de envoltura convexa para un conjunto bidimensional. La Ilustración 30 grafica la entrada y la salida de este algoritmo.

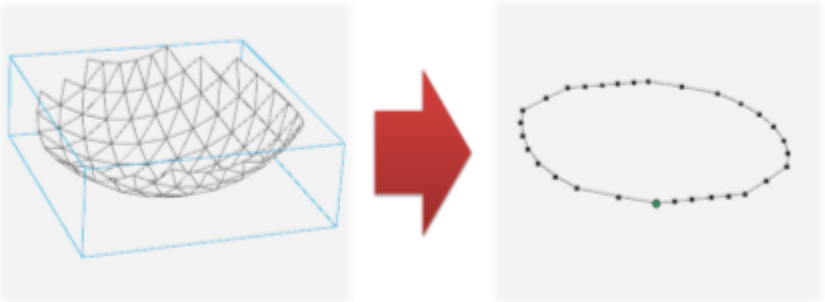


Ilustración 30: Recreación del antes y después de "aplanado" de parche.

Teniendo lo anterior se puede proceder a verificar si el centro de masa se proyecta dentro o fuera del perímetro definido por el parche. Este paso se hace de forma simple y rápida en dos pasos, primero obteniendo una proyección del centro de masa sobre el plano, y luego verificando si esta proyección es contenida por el perímetro, verificación que es capaz de hacer PCL.

Volviendo al robot

Se ha entregado soluciones para cubrir los dos requerimientos de la Definición 1 en la cual se basa la funcionalidad de detección de estabilidad. Se revisó paso a paso todas las consideraciones a tener para poder obtener un parche y verificar estabilidad para un objeto aislado. Ahora resta añadir una restricción más, relativa a la presencia del gripper. Es hora de definir cómo determinar que una pose estable es efectivamente posicionable por el gripper, tarea que es responsabilidad del Algoritmo 6, que utiliza la representación del gripper obtenida por la subcomponente 1 del presente módulo.

Algoritmo 6: Verificación de intersección entre gripper y plano

```
INPUT:
        - Nube de puntos G que representa al gripper
        - Coeficientes del plano a verificar (parche “aplanado”)
        - Un punto cualquiera P dentro del plano
OUTPUT:
        - true si intersecta o false en caso contrario
DEFINICIÓN:
        1.      Obtener vector normal N del plano
        2.      Inicializar lados (lado_a = lado_b = false)
        3.      for (Pg := punto en G):
                3.1.    Obtener vector V = P – Pg
                3.2.    Obtener ángulo α entre V y N
                3.3.    if (α < PI/2):
                    3.3.1.              lado_a = true
                3.4.    else:
                    3.4.1.              lado_b = true
                3.5.    if (lado_a and lado_b):
                    3.5.1.      return true // Hay puntos a ambos lados del plano.
                                           // SI intersecta.
        4.      return false // Nunca hubo puntos a ambos lados del plano. NO intersecta.
```

Se pide inicialmente un punto del parche plano, para acotar el plano infinito definido por los coeficientes. Luego, se recorre todos los puntos de la descripción del gripper. Si en algún punto dado se encuentran puntos a ambos lados del plano, entonces éste lo intersecta, por ende, el parche representa una superficie imposible de posicionar con el agarre actual.

Síntesis

Se ha construido toda la base para poder buscar un plano estable en el objeto. El Algoritmo 7 unifica todo este trabajo y entrega la pose que describe mejor al plano de posicionamiento encontrado.

Algoritmo 7: Búsqueda de pose estable

```
INPUT:
        - Nube de puntos O que representa objeto
        - Nube de puntos G que representa gripper
OUTPUT:
        - Pose ideal p de posicionamiento
DEFINICIÓN:
        1.      Obtener envoltura convexa de O
        2.      Obtener centro de masa CM de O (Algoritmo 4)
        3.      Obtener todos los parches (ordenados por área) planos de O, y sus respectivas áreas
        (Algoritmo 3)
        4.      for (cada parche):
                4.1.    Aplanar parche (Algoritmo 5)
                4.2.    Proyectar CM sobre parche plano
                4.3.    if    ( CM se proyecta dentro de parche and
                        G no es intersectado por parche plano (Algoritmo 6)):
                    4.3.1.              Inicializar nueva pose p
                    4.3.2.              Posicionar p en centroide de parche plano
                    4.3.3.              Orientar p según normal de parche plano
                    4.3.4.              Corregir normal de pose: Debe apuntar hacia dentro de G
                    4.3.5.              return p
        5.      return null // NO se encontró pose de placing
```

Dentro de este algoritmo se encapsula todo lo anteriormente expuesto sobre el funcionamiento interno de esta subcomponente. La reorientación de la normal hacia dentro del gripper se hace con la misma lógica utilizada para la orientación de normales del Algoritmo 2. Esto es necesario, pues la dirección de la normal debe apuntar justamente hacia donde apunte la f*uerza normal* aplicada por la mesa sobre el objeto. En la sección Trabajo Futuro se desarrolla una discusión relativa a este tema.

Tras la ejecución de esta rutina se obtiene una pose centrada en el frame del gripper, que señala cómo debiera alinearse después con la superficie de posicionado, de lo que se encarga el Módulo 5: Posicionamiento.

***Subcomponente n°3**: Representación de Objeto como Collision Object*

Luego de haber obtenido, entre otros datos, la envoltura convexa del objeto, se la puede aprovechar para representar de manera más exacta al objeto en el sistema de detección de colisiones de MoveIt, esto es, como un Collision Object.

Existen distintas formas de definir geométricamente un Collision Object (y Attached Collision Objects también). Se puede definir como conjuntos de primitivas geométricas (conos, cilindros, paralelepípedos, esferas), mallas de triángulos, planos, o una combinatoria de ellas. En el módulo 1, en la construcción del servicio *MoveArm*, para agregar una esfera se definió un Attached Collision Object basado en una primitiva esférica. Como a estas alturas se tiene detallada información sobre la geometría del objeto, se puede añadir a la escena un Collision Object basado en *malla de triángulos* que represente de la mejor manera posible su estructura externa para el planeamiento de trayectorias. La forma de especificar esta malla para este tipo de datos es distinta a la representación que entrega PCL. Por esto, lo primero que se hace es crear un método capaz de convertir desde uno al otro.

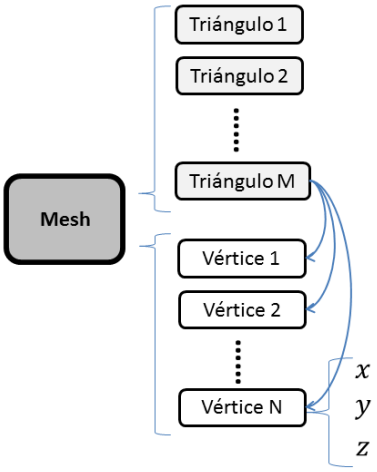


Diagrama 8: Estructura interna de un objeto Mesh

Definición de un Collision Object

Internamente, un Collision Object es un mensaje ROS del tipo `moveit_msgs/CollisionObject` que se compone de los elementos que se pueden observar en la Tabla 5. Para el uso que se le dará en este punto, sólo se debe añadir el *header*, *id* con un nombre único, operación ADD, *meshes[]* y *mesh_poses[]* relativas al frame entregado en el header.

Los *meshes[]* deben ser mensajes de tipo `moveit_msgs/Mesh`, cuya estructura interna es más simple, pues se trata de un arreglo de *vértices* con información de todos los puntos que componen a la malla, y otro arreglo de *triángulos*, cada uno compuesto de tres *vértices* referenciados en el arreglo anterior (ver Diagrama 8).

Miembros	Definición
header	Información de tiempo y frame de referencia
id	Nombre único del collision object
type	Usado cuando se conoce un modelo previo del objeto. No es nuestro caso.
primitives[]	Arreglo de primitivas geométricas.
primitive_poses[]	Arreglo de poses para las primitivas.
meshes[]	Arreglo de mallas de triángulos.
mesh_poses[]	Arreglo de poses para las mallas de triángulos.
planes[]	Arreglo de planos delimitadores.
plane_poses[]	Arreglo de poses para los planos delimitadores.
operation	Especifica si se agregará (ADD), adjuntará (APPEND) o eliminará (REMOVE) el Collision Object.

Tabla 5: Especificación de un mensaje moveit_msgs/CollisionObject

Existe mucha similitud con la estructura de los PolygonMesh de PCL, por lo que la creación de una malla para el Collision Object se hace relativamente fácil, siguiendo el Algoritmo 8.

Algoritmo 8: Creación de malla para Collision Object a partir de PolygonMesh de PCL

```
INPUT:
        - PolygonMesh PM en formato de PCL
OUTPUT:
        - Mesh MM de MoveIt
DEFINICIÓN:
        1.      Crear MM vacío
        2.      Obtener nube de puntos PC del PM
        3.      for (cada vértice v_pcl de PC):
                3.1.    Crear nuevo vértice MoveIt v_mit
                3.2.    Asignar coordenadas de v_pcl(x,y,z)
                3.3.    Agregar v_mit a vértices de MM
        4.      for (cada triángulo t_pcl de PM):
                4.1.    Crear nuevo triángulo MoveIt t_mit
                4.2.    Asignar índices de vértices en v_mit
                4.3.    Agregar t_mit a triángulos de MM
        5.      return MM
```

Usando esta malla y usando una pose en el origen del gripper, se modela correctamente el objeto manipulado como Collision Object para cálculo de trayectorias.

La implementación completa de este módulo se concentra en un nodo aislado de ROS.

Módulo 3: Búsqueda de superficie

Esta tercera fase consiste en observar el entorno en búsqueda de una superficie adecuada para el posicionamiento de un objeto. La lógica empleada se puede observar en el Diagrama 9.

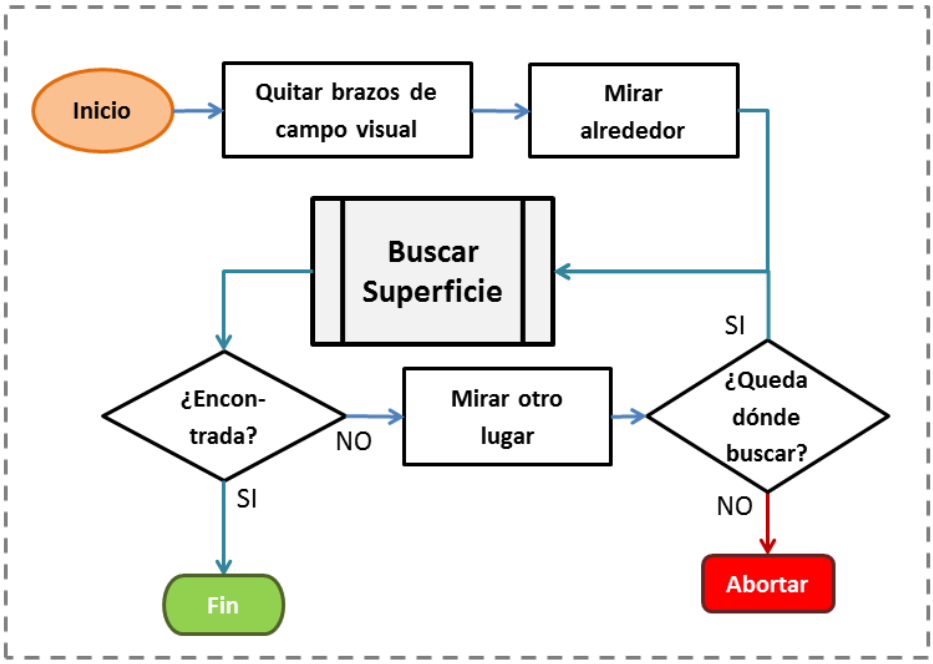


Diagrama 9: Proceso de búsqueda de superficie

En primer lugar, el robot quita del paso ambos brazos, tanto para disminuir la envergadura total del robot y no entorpecer el futuro movimiento por el escenario, como para evitar bloquear el campo de visión del sensor Kinect. A continuación el robot mira alrededor en búsqueda de superficies.

El resto del proceso itera en el intento de encontrar una superficie que cumpla los criterios ya conocidos. En cada iteración se variará la orientación de la cabeza de forma de cubrir a cabalidad el rango visual disponible. Cuando una superficie es encontrada, la etapa termina su ejecución y el robot queda listo para pasar a la etapa de desplazamiento del robot.

Naturalmente, se implementa un criterio de término en el caso de no encontrar la superficie. El criterio de cancelación de la ejecución es no haber encontrado una superficie tras buscar en todo el rango visual alcanzable por la Kinect, siempre desde la perspectiva del punto de partida.

Subcomponente n°1*: Motricidad*

Se hace uso de los servicios *MoveArm* y *LookAt* creados para el Módulo 1: Modelamiento del Objeto.

Subcomponente n°2*: Búsqueda de superficie*

Esta componente es la encargada de obtener y procesar la nube de puntos desde los datos capturados por el sensor Kinect. En términos de software, es aquí donde se implementa la lógica de todo el módulo, definida por el Algoritmo 9, y contiene el algoritmo principal para su funcionamiento. Del proceso de modelamiento y análisis del objeto manipulado, se obtuvo entre otros datos, el área que éste ocuparía ya posicionado en su pose estable, parámetro que es considerado en la búsqueda.

Algoritmo 9: Detección de superficie para posicionamiento de objeto

```
INPUT:
        - Nube de puntos del Kinect
OUTPUT:
        - Nube con puntos que componen a la superficie (inliers del modelo plano)
        - Centroide de la nube de puntos
        - Vector normal del plano aproximado de la superficie
DEFINICIÓN:
        1.      Obtener transformación desde frame del Kinect a frame odométrico
        2.      Submuestrear nube de entrada
        3.      while (hay suficientes puntos para procesar):
             3.1.               Segmentar input, usando algoritmo RANSAC para búsqueda de un plano
             3.2.               if (muy pocos inliers) or (no se encuentra el modelo):
                  3.2.1.        return //NO se encontró modelo
             3.3.               Calcular centroide y transformar a frame base
             3.4.               if (altura centroide es razonablemente alcanzable por el robot):
                  3.4.1.        Calcular vector normal de superficie
                  3.4.2.        Transformar normal a frame base
                  3.4.3.        if (inclinación de normal, respecto al suelo, es poca):
                        3.4.3.1. refinar X veces
                        3.4.3.2.        return inliers superficie
                        3.4.3.3.        return vector normal
                        3.4.3.4.        return centroide
             3.5.               Quitar inliers del input // El modelo encontrado no sirvió:
                                                         // Eliminarlo y seguir buscando
```

El sensor Kinect entrega una nube de puntos considerablemente densa, considerando su resolución de 1280x960 pixeles. Dependiendo de las operaciones que se deseen hacer sobre este conjunto de datos, su procesamiento puede resultar costoso, por lo que se opta por aplicar primero un submuestreo sobre la nube de entrada (ver Ilustración 31). Al igual que el caso del Módulo 1: Modelamiento del Objeto en la adquisición de puntos para modelamiento del objeto, se observó experimentalmente que con alrededor de un 2% de los puntos de entrada fue posible obtener nubes de puntos que representaran de forma suficientemente precisa el entorno.

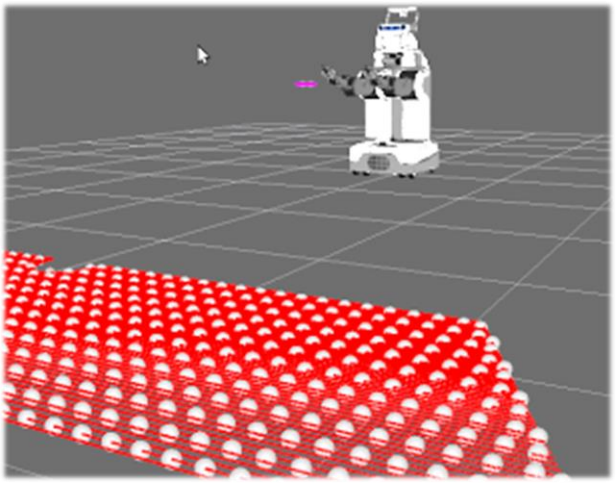


Ilustración 31: Submuestreo de nube de puntos del sensor Kinect. En este ejemplo se toma un punto cada 0.1 m. El resultado es la nube de puntos representados como esferas.

Los puntos recibidos están configurados en un espacio centrado en el punto de referencia del Kinect. En la Ilustración 32 se puede apreciar cómo se ve inicialmente la nube de puntos respecto a dicho frame, cuyo origen se encuentra graficado como tres ejes perpendiculares ubicados en el lente del sensor. Sobre la cabeza del robot se puede ver una grilla que representa al plano XY en este frame. En esta imagen en particular se observa el resultado de búsqueda del mayor conjunto de puntos que representan un plano. Es importante representar los puntos relativos al frame de referencia del suelo, desde donde se puede recién hacer un análisis de inclinación y altura de la superficie que tenga sentido, para lo que se usa el módulo TF de ROS.

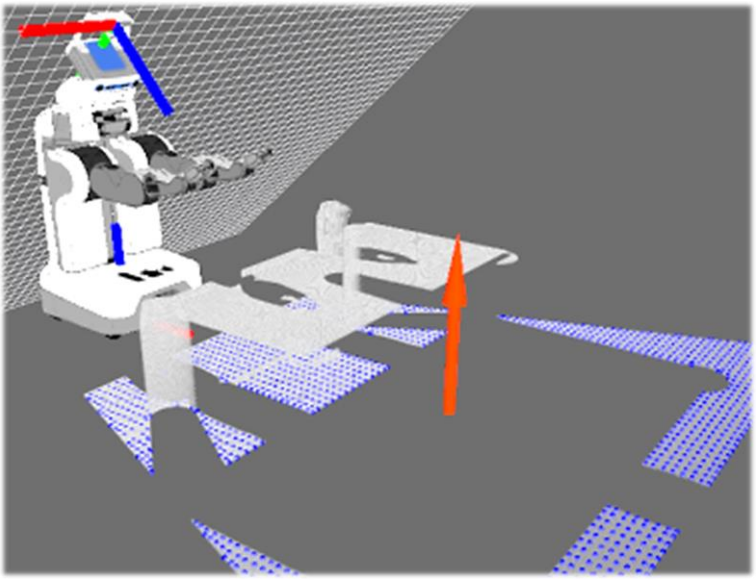


Ilustración 32: Nube de puntos desde perspectiva de sensor Kinect, ubicado en la cabeza del robot. La grilla flotante representa el plano XY del kinect, y las rectas roja y azul se ubican en el origen del sistema de coordenadas, representando los ejes X y Z respectivamente..

Para la detección de los puntos pertenecientes al plano buscado, se utiliza el módulo *Segmentation* de la librería PCL. El parámetro de segmentación *distance_threshold*, que fija la

distancia máxima a la que tiene que estar un punto del modelo buscado para ser considerado un inlier, fue ajustado experimentalmente, logrando determinar que con 1 cm de margen se obtienen buenos resultados en tiempos cortos (inferiores a 500 ms). En el caso de la búsqueda de superficie, no se observó grandes diferencias entre utilizar directamente la nube de puntos del Kinect y leer desde su versión auto-filtrada, generada por MoveIt. Pero ya que la geometría del cuerpo del robot no es relevante en este análisis, se escogió la segunda opción.

Los criterios de búsqueda de superficie consisten en su altura, inclinación respecto al suelo y área libre total. Sin embargo, esto puede no representar bien la factibilidad de poder efectivamente posicionar el objeto en esta superficie. Una discusión respecto a este tema se ofrece en la sección Trabajo Futuro.

Al momento de encontrar un candidato, el robot pasa a una etapa de "refinación" (indicada por 3.4.3.1 del Algoritmo 9: Detección de superficie para posicionamiento de objeto), donde apunta su cabeza al centroide de la nube, y vuelve a buscar una superficie, de altura similar a la anterior. Este proceso es repetido tantas veces se desee, y se ha configurado para repetirse 3 veces por defecto.

Subcomponente n°3*: Representación de superficie como Collision Object*

Si bien la superficie es actualmente representada como voxels del octomap, es conveniente incluir la geometría de la superficie como un collision object al mundo, puesto que entrega a MoveIt una noción mucho más exacta de sus dimensiones que el particionado de voxels, pudiendo así hacer un placing más preciso. Al igual que en la Subcomponente n°3 del módulo 2, se creará un nuevo Collision Object a partir de una triangulación del conjunto de puntos adquiridos, creando primero la malla de PCL con su módulo *Surface* y luego realizando la conversión de estos datos a un *Mesh* de MoveIt, para finalmente ser añadida al mapa.

El módulo generado se ve desde la arquitectura de ROS como se ve en el Diagrama 10.

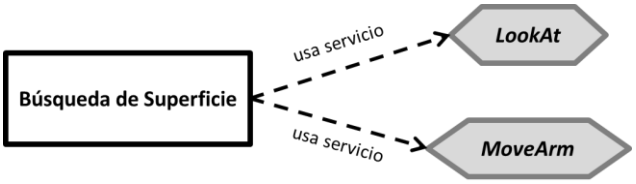


Diagrama 10: Perspectiva de ROS del módulo 3

Módulo 4: Desplazamiento hacia Superficie

Una vez definida la posición y composición de la zona de *placing*, el robot debe posicionarse cerca de la superficie y en una ubicación que lo habilite para el posicionamiento. La lógica de esta etapa es sencilla, y puede observarse en el Diagrama 11.

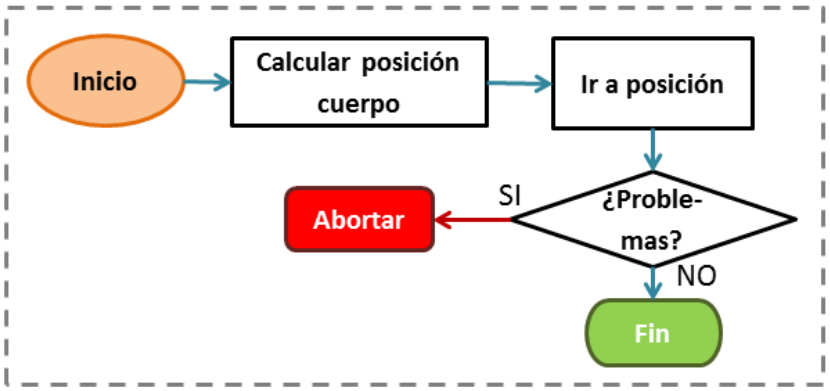


Diagrama 11: Proceso de acercamiento a superficie

El robot, a partir de la información de posición de la superficie obtenida previamente, calcula una pose adecuada de ubicación, frente al punto más cercano de la superficie, suficientemente distanciado del borde y orientado de forma de mirar hacia ella. El PR2 intenta a continuación posicionarse y orientarse según esta pose. En caso de detectarse algún tipo de problema grave, la ejecución se aborta.

***Subcomponente n°1**: Movilidad*

En el Capítulo 1 se mencionó al problema de movilidad en robótica autónoma como un desafío no trivial, que requiere ciertas consideraciones:

- Un entorno cotidiano puede ofrecer al robot obstáculos que entorpecen su desplazamiento, pudiendo incluso aparecer sorpresivamente objetos o personas en el camino, teniendo el robot que calcular un nuevo camino a recorrer.
- El problema del cálculo de la ruta de desplazamiento no puede ser pasado por alto, pues existen infinitas trayectorias posibles para llegar desde un punto a otro en un camino abierto, y se debe ser capaz también de poder detectar cuándo no existe realmente solución para llegar.
- Pueden darse casos donde no sólo es necesario saber la ruta a recorrer, sino que también la pose del robot durante la ejecución de esa ruta, por ejemplo, en casos el robot podría no poder pasar por un corredor orientado diagonalmente respecto al avance, pero sí desplazándose de lado.

Los dos primeros problemas han sido abordados por los desarrolladores del sistema operativo ROS, creando para ello el *Navigation Stack*. Lamentablemente, pese a la documentación ofrecida en el sitio web del sistema de cómo configurar y ejecutar las funcionalidades de planeamiento de trayectoria, no se pudo hacer uso de este *stack*, mayormente por inconsistencias entre la información oficial entregada y el comportamiento real del software distribuido.

Evaluando esta situación y en búsqueda de alternativas, se decidió implementar una versión básica de navegación “no inteligente”, sin detección de obstáculos ni cálculo de trayectorias, puesto

que estas capacidades no son estrictamente necesarias para lograr tareas de manipulación, y constituyen un problema distinto.

Para suplir esta necesidad, se creó el servicio ROS ***GoToPose***, encargado de mover al robot a una pose específica en su entorno. El programa hace uso de la componente *ActionLib* encargada de recibir requerimientos de desplazamiento de la base del robot. Admite distintos comandos para desplazar lineal y angularmente al robot activando las ruedas de su base. Haciendo uso de esta funcionalidad, se dotó al servicio *GoToPose* de métodos de alto nivel para moverse a una pose exacta en entorno.

Subcomponente n°2*: Cálculo de pose para el robot*

En términos de software es ésta la subcomponente que implementa la mayor parte de la lógica del módulo. El programa toma en cuenta las dimensiones del robot y posición de la superficie relativa a éste para una buena estimación de pose de manipulación, y hace llamados al servicio *GoToPose* para mover el robot. Como la nube de puntos ya se encuentra representada relativa al frame odométrico, que es según el cual el robot interpretará la pose hacia dónde ir, no es necesario obtener una transformación.

Algoritmo 10: Cálculo de pose para el robot

```
INPUT:
        - Nube con puntos de la superficie
        - Centroide superficie
OUTPUT:
        - Estado del requerimiento
DEFINICIÓN:
        1.      Buscar punto más cercano de la mesa al robot
        2.      Si el robot está suficientemente cerca, terminar
        3.      Calculo pose entre el robot y el punto cercano, suficientemente lejos para no
        chocar, y orientada hacia el centroide de la superficie
        4.      Desplazar robot (servicio GoToPose)
        5.      Mirar hacia la mesa (servicio LookAt)
```

El algoritmo es independiente de la pose inicial del robot relativa a la superficie. La pose final calculada para efectuar el placing es alcanzada gracias al servicio *GoToPose* que se encarga de rotar y desplazar el cuerpo completo controlando las ruedas de su base.

El punto más cercano de la mesa al robot se calcula utilizando el módulo *KdTree* de la librería PCL. El resto de la definición del Algoritmo 10 es prácticamente auto explicativa. Notar que al final se hace una llamada al servicio *LookAt*, creado para el Módulo 1: Modelamiento del Objeto, aquí utilizado para finalizar el proceso mirando la superficie. Este paso en la práctica no es estrictamente necesario pero tiene sentido para el módulo siguiente.

La comunicación entre los archivos ejecutables generados para su funcionamiento en ROS puede observarse en el Diagrama 12.

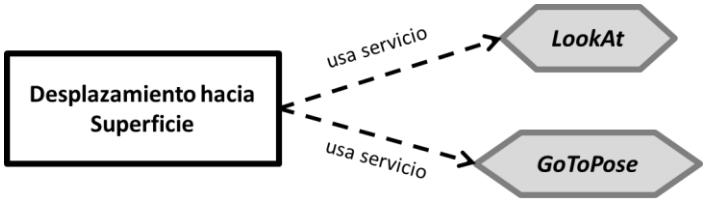


Diagrama 12: Perspectiva de ROS del módulo 4

Módulo 5: Posicionamiento

El módulo 1 digitaliza el objeto manipulado, dando paso al Módulo 2 para obtener su pose estable. Los módulos 3 y 4 se encargan de llevar al PR2 a una posición que posibilite el posicionamiento. El trabajo restante consiste en calcular cómo ubicar el gripper activo en el espacio para efectuar el "placing", y finalmente soltar el objeto. El Diagrama 13 muestra este proceso lógico.

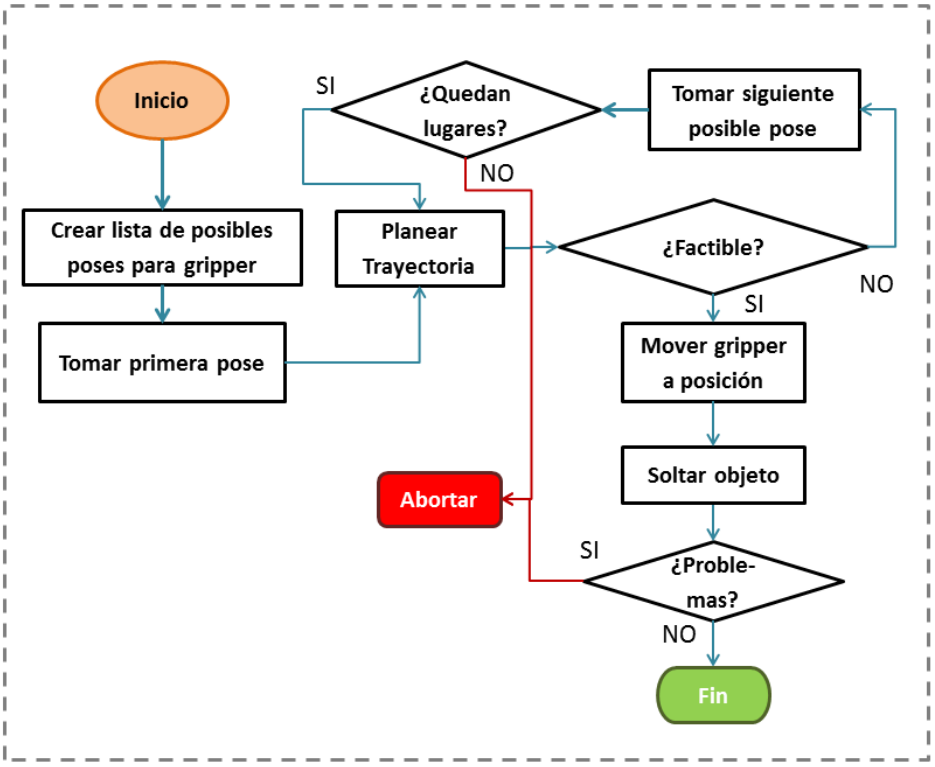


Diagrama 13: Proceso de posicionamiento de objeto

Primero se calcula las posibles formas de poner el gripper sobre la superficie para posicionar bien el objeto. Luego se verifica la factibilidad de ir hacia alguna de estas poses, labor que realiza MoveIt. Se prueba todas las poses posibles hasta encontrar una factible, o bien, agotar las posibilidades, caso en el que el programa aborta.

Si se encontró una pose factible para el gripper, se envía una petición a MoveIt para ejecutar la trayectoria. Finalmente se abre el gripper, el objeto cae y se quita el brazo activo de la escena.

***Subcomponente n°1**: Generar poses para gripper*

En general no existe una única solución al problema de encontrar una pose para el gripper que asegure la estabilidad del objeto, primero porque existen distintas posiciones posibles en la superficie donde colocarlo, y segundo porque en cada una de estas posiciones, pueden existir infinitas orientaciones además.

El Módulo 2 muestra exactamente dónde debe descansar éste para que se mantenga estable y no caiga. Pero esto no sirve de nada por sí solo, se necesita una forma de relacionar esto con la superficie encontrada.

Asumamos por ahora que se sabe exactamente dónde se lo quiere ubicar, y que todas las poses se evalúan respecto a un mismo punto de referencia arbitrario. En la Ilustración 33 se grafica el caso de un objeto cuya pose estable calculada es A y su pose asignada sobre la superficie es A'. En la parte inferior, se ve al gripper inicialmente ubicado en la pose B para sostener al objeto. A la derecha se observa que para poder poner al objeto en su pose A', el gripper debe asumir una pose B'. Se puede observar entonces que B' dependerá del estado de A'. Para encontrar esta relación, se realiza el siguiente análisis:

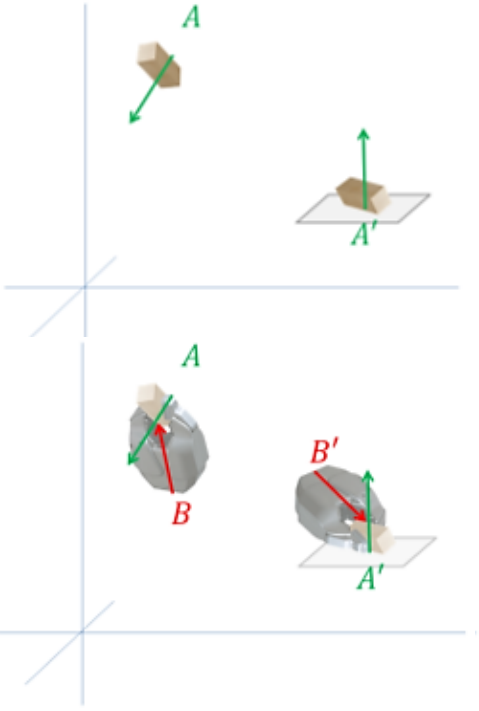


Ilustración 33: Desplazamiento de gripper relativo al objeto

Para representar la posición del gripper en función de la posición del objeto, es útil primero revisar cómo se usan los cuaterniones, para luego entender qué significan realmente las componentes internas de una pose.

Se dijo anteriormente que los cuaterniones se utilizan para operaciones de rotación. Si se desea rotar un vector $\vec{v}$ usando un cuaternión $\vec{q}$, se debe hacer $\vec{v}' = \vec{q}\vec{v}\vec{q}^{-1}$. Los detalles algebráicos de esta operación, como el tipo de multiplicación vectorial utilizada y definición de inverso de un cuaternión, no resultan relevantes al ser aplicados de forma transparente por la librería TF, que maneja este tipo de datos y ofrece mecanismos de aplicación de estas operaciones.

Un detalle importante es que para aplicar varias rotaciones sucesivas $q_1, q_2, q_3, \dots, q_n$ en ese orden, sobre un vector, basta con calcular el producto de todos los cuaterniones, **en el mismo orden en el que se quiere hacer las rotaciones**, al contrario del caso de matrices de rotación, donde el último factor es la primera rotación aplicada.

En ROS, una pose $P(\vec{p}, \vec{q})$ independiente del frame es un mensaje de tipo `geometry_msgs/Pose`, que internamente se compone de una componente de posición $\vec{p}(x, y, z)$, y una de orientación representada por un cuaternión $\vec{q}(x, y, z, w)$. La forma de interpretar esta información es:

Asumir un sistema de referencia ubicado en $P_0(\overrightarrow{p_0}, \overrightarrow{q_0})$. Se crea una pose centrada en el origen $\overrightarrow{p_0}(0,0,0)$ y orientada con respecto al eje X, $\overrightarrow{q_0}(0,0,0,1)$. Luego se aplica la rotación $\vec{q}$ sobre esta pose inicial, para luego trasladar el origen de la pose a la posición espacial dada por $\overrightarrow{p_0} + \vec{p}$.

Como se quiere expresar la pose B en términos de A, basta con reemplazar la pose inicial del sistema de referencia por la pose de A, que pasará a ser el sistema de referencias de B. Con esto, siempre que se tenga la definición de A se podrá encontrar la pose exacta de B, o sea, si sabemos la pose final del objeto, sabremos dónde debe estar el gripper.

Se debe cumplir:

$$q_B = q_A q_m$$

para algún q_m, cuyo valor se puede calcular haciendo:

$$q_B = q_A q_m \Leftrightarrow$$
$$q_A{}^{-1} q_B = q_A{}^{-1} q_A q_m \Leftrightarrow$$
$$q_A^{-1} q_B = (q_A^{-1} q_A) q_m \Leftrightarrow$$
$$q_A^{-1} q_B = q_m \blacksquare$$

La posición de B, p_B, en términos de A, estará dada por el vector de largo $\|p_A - p_B\|$ y orientado según un q_n aplicado sobre q_A. Para encontrar q_n sencillamente se convierte el vector posición p_B en una nueva pose P_{pB} centrado en el origen de A y orientado desde A hacia B. Cuando se aplique q_m a P_{pB} se obtendrá el vector que representa la nueva posición $p_{B'}$.

Sea cual sea la nueva pose que se proponga para el objeto, siempre podrá obtenerse la pose del gripper que lo llevará a esa configuración.

Rotaciones en torno a la pose estable

Para una cierta pose final del objeto, se puede calcular la pose relativa del gripper. Sin embargo esta no es la única pose válida, pues si se considera todas las posibles rotaciones en torno al eje definido por el vector normal a la superficie de apoyo de la envoltura convexa del objeto (equivalentemente, por la normal del plano de la superficie de posicionamiento), se sigue teniendo poses estables. La Ilustración 34 grafica esta idea mostrando distintas poses para el gripper, donde la pose final del objeto se representa como un vector perpendicular al plano de la imagen, de sentido que apunta hacia el lector.

Ilustración 34: Distintos ángulos válidos de roll para pose final del gripper

Estas distintas poses cambian radicalmente las probabilidades de éxito al planear una trayectoria para llegar a ellas, donde algunas pueden resultar muy "cómodas" para el robot, mientras otras resultan imposibles de alcanzar.

Para incluir este factor de rotación según el eje que define la pose final A', se incluye en la cadena de rotaciones una última operación, dada por un nuevo cuaternión $q_{roll}(\theta)$ que representa una rotación en torno al plano YZ del sistema de referencia. Anteriormente se tenía que la orientación del gripper estaba dada por:

$$q_B = q_A q_m$$

Ahora, para poder añadir el efecto de $q_{roll}(\theta)$ se redefinirá:

$$q_B = q_A(q_{roll}(\theta)) q_m$$

El significado de esta operación es: "Orientar pose según sistema de referencia definido por A, luego rotar según el eje X_A en θ grados; finalmente aplicar rotación final".

Recapitulación

Habiendo definido cómo obtener la pose final del gripper para el posicionamiento estable, se crea el Algoritmo 11, que es el encargado de generar un conjunto de poses a ser evaluadas por el planeador de trayectorias.

Dado que es preferible que este espacio de soluciones sea reducido, se toma dos medidas:

1. Considerar sólo puntos de la superficie realmente alcanzables por el brazo activo. Para esto, se considera sólo los puntos de la superficie cuya distancia al comienzo del brazo es menor a su largo total.
2. Considerar un subconjunto pequeño de ángulos para la definición de los distintos $q_{roll}(\theta)$. La cantidad de ángulos fue fijada, tras validaciones experimentales, en 10. Al aumentar este valor, no se obtuvo poses significativamente mejores. Se ofrece una discusión sobre este punto en la sección Trabajo Futuro.

Algoritmo 11: Generación de posibles poses finales para el gripper

```
INPUT:
        - Pose estable inicial del objeto PA(pA,qA)
        - Pose inicial del gripper PB(pB,qB)
        - Nube de puntos de la superficie PC
OUTPUT:
        - Arreglo de poses posibles Pout[]
DEFINICIÓN:
         1.     Transformar PA y PB a frame odométrico
         2.     Filtrar PC, quitar puntos no alcanzables por el brazo
         3.     Crear arreglo vacío de poses Pout[]
         4.     for (p_s := punto de PC):
                     4.1.       Calcular orientación qm=qA^-1 qB
                     4.2.       Calcular posición pm
                     4.3.       for (θ := ángulo de partición de rotaciones)
                             4.3.1.     Crear cuaternión de rotación q_roll(θ)
                             4.3.2.     Obtener nueva posición pm_roll y orientación qm_roll
                             4.3.3.     Crear nueva pose P_new(pm_roll,qm_roll)
                             4.3.4.     Añadir P_new a Pout[]
    5.  return Pout[]
```

***Subcomponente n°2**: Testeo de poses y posicionamiento*

Para cada pose generada, se pide al *planner* de MoveIt, a través del servicio *MoveArm* que evalúe su factibilidad. Cuando alguna de las poses del gripper es considerada como factible, se envía la orden de ejecutar la trayectoria correspondiente. A continuación se procede a abrir la tenaza, dejar que el objeto se apoye en la superficie de posicionamiento, y retirar el brazo de la escena. En este último paso se tiene el cuidado de retirarlo retrocediendo en la misma dirección de agarre del gripper, para no correr el riesgo de voltear el objeto (ver Ilustración 35). Se finaliza llevando el brazo activo, ahora desocupado, a un lado del robot. Este proceso lo ejecuta el Algoritmo 12.

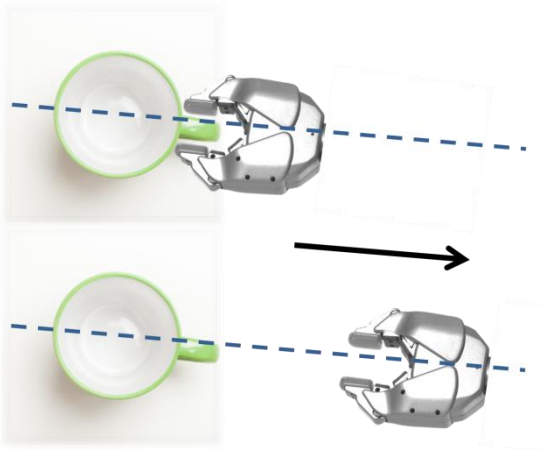


Ilustración 35: Retirada de gripper tras posicionamiento de objeto

Algoritmo 12: Testeo de poses y posicionamiento

```
INPUT:
        - Arreglo de poses posibles para gripper P[]
OUTPUT:
        - Estado de la operación
DEFINICIÓN:
         1.     for (P := pose en P[]):
                     1.1.       Intentar planear trayectoria
                     1.2.       if (plan exitoso):
                                1.2.1.     Llevar gripper a pose
                                1.2.2.     if (hay errores):
                                            1.2.2.1.       return false
                                1.2.3.     Retirar gripper
                                1.2.4.     Mover gripper a un lado
                                1.2.5.     return true
         2.     return false
```

La arquitectura básica de software en ROS generada queda como se muestra en el Diagrama 14.

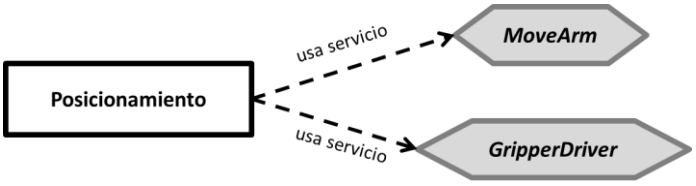


Diagrama 14: Perspectiva de ROS del módulo 5

Integración de Módulos

Habiendo diseñado, implementado y probado todos los módulos y sus subcomponentes correspondientes, llega el momento de unificar todo en un solo programa principal.

Hasta ahora la arquitectura total generada, en términos del sistema ROS consiste en lo graficado por el Diagrama 15.

Los servicios de ROS creados constituyeron en principio una solución elegante para la implementación de funcionalidades de movilidad y motricidad del robot, pero no ofrecen en este escenario específico una real ventaja por sobre el uso clásico de clases y objetos de C++, a lo que se suma el hecho de que en particular no se está haciendo uso de la naturaleza de sistema distribuido de ROS, produciendo un overhead innecesario en la transmisión de datos por la red.

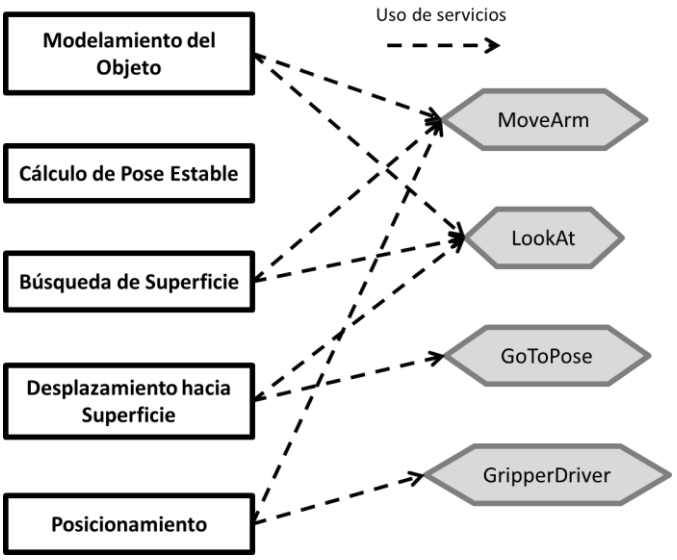


Diagrama 15: Perspectiva de ROS de todos los módulos

Es importante también el hecho de que cada módulo contiene una cantidad no despreciable de parámetros configurables para sus distintas etapas, que en este modelo deben ser configurados en archivos separados. Esto no es conveniente cuando ya se trabaja con la versión unificada del algoritmo de placing. Se ofrece un mayor análisis de estos puntos en la sección de Trabajo Futuro más adelante.

A lo anterior se añade que cada componente implementa métodos similares (operaciones sobre nubes de puntos, transformaciones espaciales, interacción con entorno de colisiones MoveIt) que perfectamente pueden ser unificados en un solo archivo.

Por todo esto se decidió realizar un proceso de reestructuración importante que culminó en una arquitectura más eficiente, menos redundante y mucho más comprensible. Esta reestructuración dio nacimiento a las siguientes componentes:

***Componente n°1**: RobotDriver*

Todas las componentes de software dedicadas al control de sensores y actuadores del robot para control del movimiento de extremidades y adquisición de datos, fueron congregadas dentro de la clase ***RobotDriver***. El programa principal inicializa una instancia *RobotDriver*, a través de la que se puede acceder a todas las funcionalidades previamente entregadas por los Servicios ROS diseñados..

La responsabilidad del funcionamiento correcto de *RobotDriver* recae en sus componentes internas que son las siguientes:

- **Clase *RobotBaseDriver***: Provee métodos de control de la base, para desplazamiento y giros. Reemplaza al servicio *GoToPose.*
- **Clase *RobotGripperDriver***: Engloba desde la apertura del gripper hasta posicionamiento en el espacio. Es esta clase la encargada de configurar e implementar los métodos de la librería MoveIt. Reemplaza al servicio *GripperDriver* y *MoveArm.*

- **Clase *RobotHeadDriver***: Implementa métodos para movimiento de cabeza, admitiendo especificación de comandos de alto nivel como rotar o inclinar cabeza en cierto ángulo y mirar hacia cierto punto del espacio relativo a algún frame dado. Reemplaza servicio *LookAt*.
- **Clase *RobotSensors***: Diseñada para encapsular la adquisición de datos desde sensores en el robot. Actualmente sólo se utiliza el Kinect, pero es extensible para añadir en un futuro otros sensores.

Componente n°2: Polymesh

En el Módulo 2 al obtener parámetros del objeto se debe hacer un procesamiento de la nube de puntos que lo representa, principalmente en su componente convexa, para poder obtener sus vectores normales, parches planos y sus áreas asociadas, centro de masa, entre otros. Para lograr el encapsulamiento de estas operaciones se crea la clase *Polymesh* que contiene los métodos para configurar, computar y acceder a estos cálculos.

Componente n°3: PlacedObject

Específicamente pensada en el modelamiento y obtención de características para el objeto manipulado. Agrupa todas las operaciones de adquisición del objeto y de obtención de características instanciando internamente un objeto *Polymesh*. Permite acceder a su pose estable y obtener la información necesaria para ser representado como Collision Object.

Componente n°4: PlacingSurface

Similar a PlacedObject, se crea esta componente para agrupar todas las operaciones relativas a la búsqueda de la superficie de apoyo, obtención de datos y representación en el mapa de MoveIt.

Componente n°5: Util

Esta es la clase que encapsula las utilidades generales necesarias para el programa. Se compone de cinco principales partes:

- Declaración de **parámetros** para todo el programa.
- Métodos simples de **conversión** entre unidades y tipos de datos.
- Encapsuladores de operaciones relativas a **transformaciones** (potenciadas por la librería TF). Permite obtener la matriz de transformación entre dos frames, y su aplicación a nubes de puntos, poses, cuaterniones, vectores y puntos.
- Creación y configuración de **Collision Objects** de MoveIt.
- **Utilidades** esenciales para el funcionamiento del programa, destinadas a simplificar la lectura y programación del programa principal.

Componente n°6: Placing

Programa principal que implementa todo el algoritmo de placing. Aquí se inicializan nodos ROS, se crea el RobotDriver para control del PR2 y se realiza la integración completa de los 4 módulos discutidos. Utiliza todas las declaraciones integradas en la clase Util.

Componente n°7: Viewer

Diseñada específicamente para visualización de datos de la librería PCL y de especial importancia para visualizar resultados independientes de ROS. El visualizador original de la librería

posee muchas posibles configuraciones y sobrecarga en gran medida sus métodos. La clase *Viewer* realiza internamente las configuraciones necesarias para ofrecer visualizaciones útiles para el software de posicionamiento.

En la sección Trabajo Futuro de este documento se encuentra una discusión sobre el diseño de estas componentes.

Estructura Final

El resultado de esta organización se puede resumir en el mapa de la Ilustración 36:

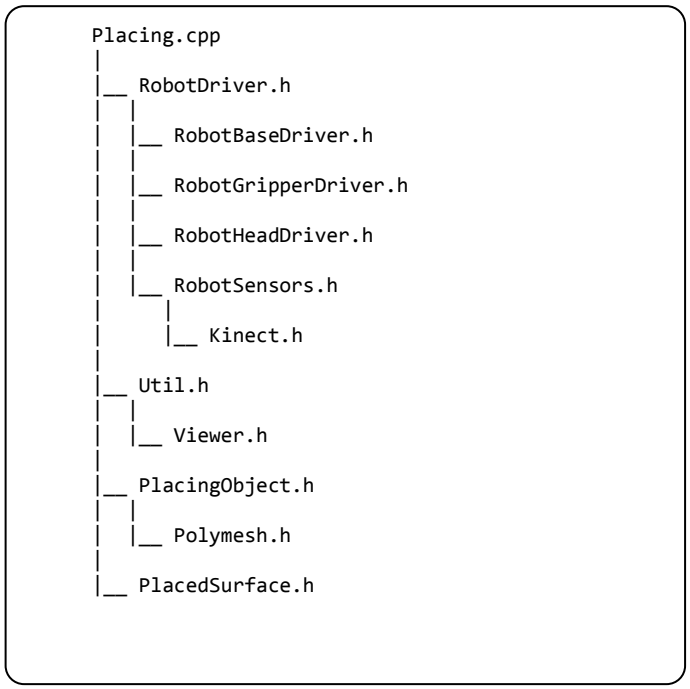


Ilustración 36: Mapa organizativo de archivos header de la estructura total integrada.

Utilización del programa

El código fuente completo del paquete ROS resultante puede encontrarse en el repositorio público ubicado en https://github.com/mexomagno/pr2_placing/tree/master/src/memoria2.

Para instalar el software se debe copiar la carpeta dentro del workspace de *catkin*[10], como se haría para cualquier paquete ROS. Luego de compilar y cargar el script bash generado por catkin, previo iniciar una instancia del robot PR2 (ya sea simulado o real) que cumpla con las precondiciones descritas en el Capítulo 3, se debe ejecutar el archivo binario *"place2"* y entregar como argumento el brazo que sostiene al objeto, entregando *"l"* o *"r"* según sea el gripper izquierdo o derecho respectivamente.

[10] Colección de macros creada para compilar paquetes ROS.

Ejemplo de ejecución en una consola bash:

```
$ rosrun memoria2 place2 l # objeto sostenido en gripper izquierdo.
```

El programa comenzará a ejecutar los módulos y entregará feedback por la consola.

El envío de una interrupción a través del comando Ctrl+C cancela la ejecución y destruye los objetos creados.

Capítulo 5

Resultados

El Capítulo 4 muestra la construcción detallada de cada uno de los cinco módulos de los que se compone todo el programa de posicionamiento. En el proceso, se crearon algoritmos para la resolución de problemas específicos de cada módulo. Lo que sigue del presente capítulo se dividirá en una primera parte, "*Resultados intermedios*" que muestran el desempeño de estos algoritmos, ordenados por el módulo al que pertenece, y en una segunda parte, "*Resultados Finales*", que grafica el comportamiento de la versión unificada.

En la página de información del repositorio que alberga el código de este proyecto, se puede acceder a videos que demuestran algunos ejemplos de funcionamiento.

Resultados Intermedios

Módulo 0: Preparación del Robot

Usando los scripts de inicialización, interacción con Gazebo y tele-operación de los grippers, se puede configurar el robot en casi cualquier situación inicial válida. En la Ilustración 37 se observa al robot virtual en Gazebo sosteniendo objetos simples, con agarres sencillos, y otros más complejos.

Ilustración 37: PR2 sujetando un cubo y un taladro, mostrando distintas posibilidades de agarre. Visto desde Gazebo.

Módulo 1: Modelamiento del Objeto

Motricidad

La subcomponente de motricidad de este módulo dio origen a los servicios *LookAt* y *MoveArm*. Resultados de los tests de implementación del primero se pueden ver en la Ilustración 38, y del segundo, en la Ilustración 39.

La Ilustración 40 muestra el movimiento de un brazo del robot, junto al Attached Collision Object esférico anclado al gripper activo.

Ilustración 38: PR2 moviendo la cabeza con servicio LookAt

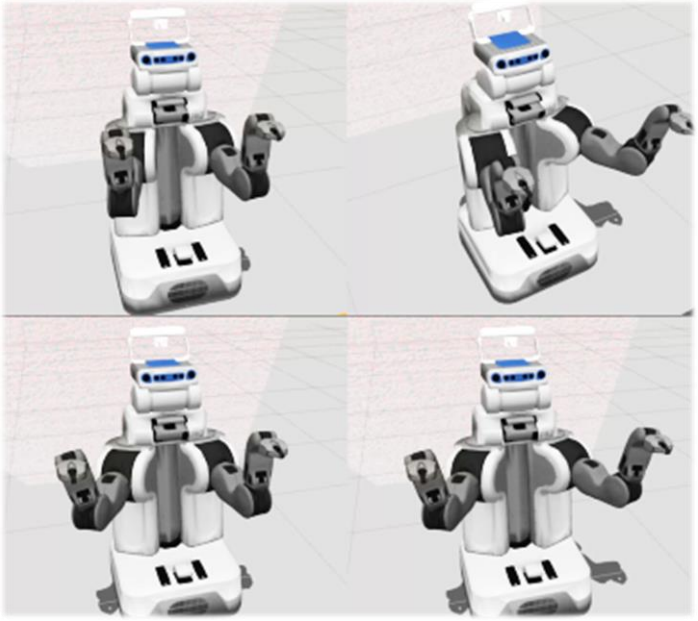


Ilustración 39: PR2 moviendo grippers con servicio MoveArm

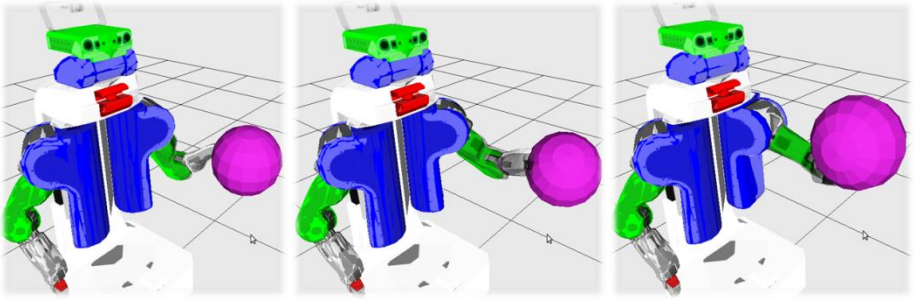


Ilustración 40: PR2 moviendo brazo con un Attached Collision Object esférico.

Adquisición

En una versión preliminar, se probó el algoritmo para capturar una imagen del gripper vacío y cerrado, que se logró leyendo directamente desde la nube de puntos que entrega el sensor Kinect en el tópico `"/head_mount_kinect/depth_registered/points"`. El resultado es el que se observa en la Ilustración 41, donde se ve una nube de puntos con la forma de un gripper. Los cuatro colores dan cuenta de las cuatro distintas perspectivas capturadas, aunque se trata de una misma nube de puntos.

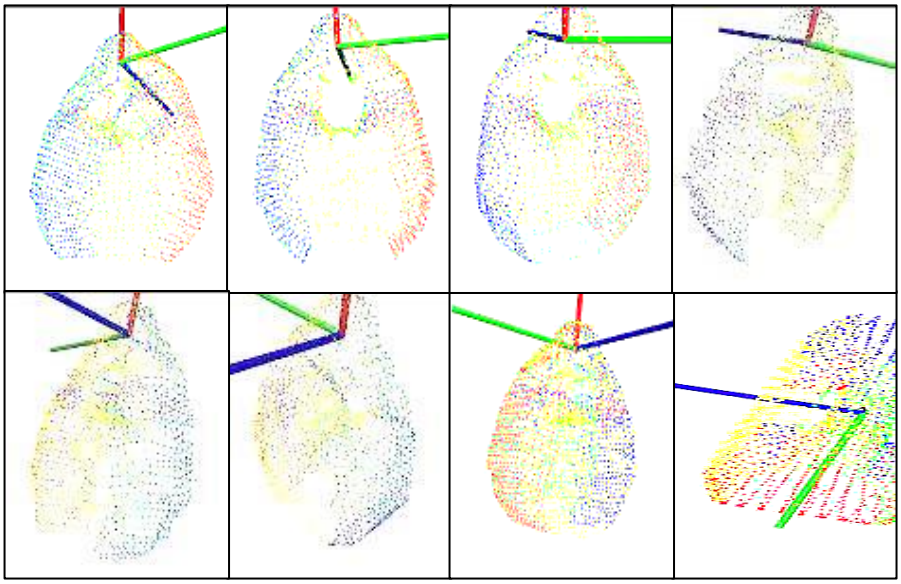


Ilustración 41: Gripper scaneado desde nube cruda de puntos de Kinect (sin auto-filtrado)

Se configura inicialmente el robot para que sostenga un objeto con el gripper izquierdo, como se grafica en la Ilustración 42. La Ilustración 43 muestra un proceso completo de modelamiento de un cubo, realizado por el Algoritmo 1. Se inicia la ejecución (a), y el robot procede a mover hacia un lado el brazo inactivo (b), para luego llevar el objeto a la pose de captura y apuntar el sensor Kinect directamente hacia él (c). Procede a capturar varias perspectivas (d, e, f) para obtener finalmente una representación tridimensional aislada (g).

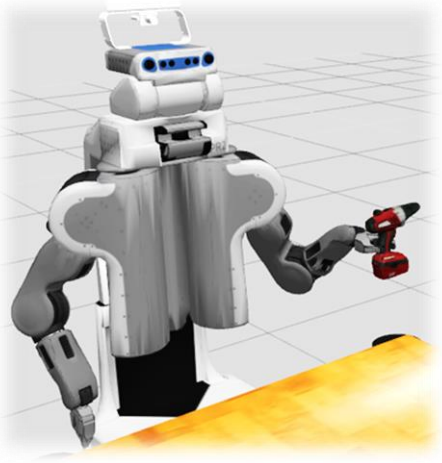


Ilustración 42: Configuración inicial del PR2 para prueba de digitalización de objeto

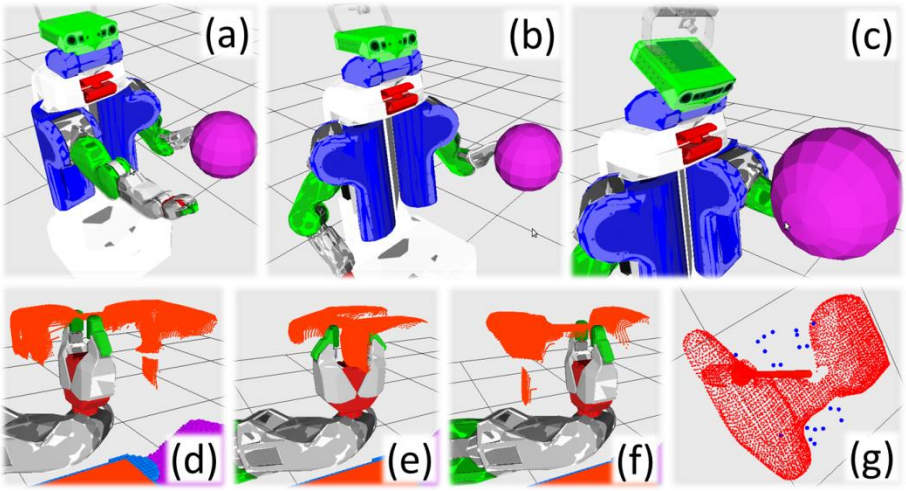


Ilustración 43: Proceso de obtención de modelo geométrico

Módulo 2: Cálculo de Pose Estable

Modelamiento del gripper

El modelamiento del gripper resulta en lo que muestra la Ilustración 44, que para fines de selección de parches no conflictivos, es suficiente.

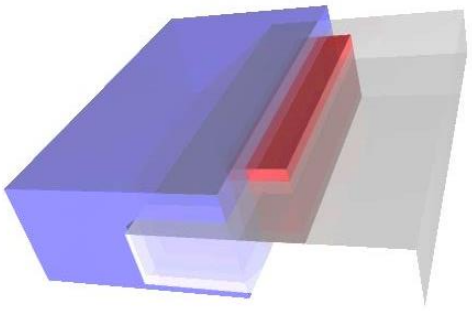


Ilustración 44: Representación del modelo de gripper basado en tres cuboides superpuestas.

Reorientación de normales

Al crear inicialmente el conjunto de vectores normales, se obtienen resultados incorrectamente orientados, como se puede ver en el óvalo de la parte inferior de la Ilustración 45, que muestra cómo las normales apuntan hacia el interior del volumen, mientras otras apuntan hacia el exterior. Al aplicar el Algoritmo 2, se obtienen normales correctamente orientadas (ver Ilustración 46).

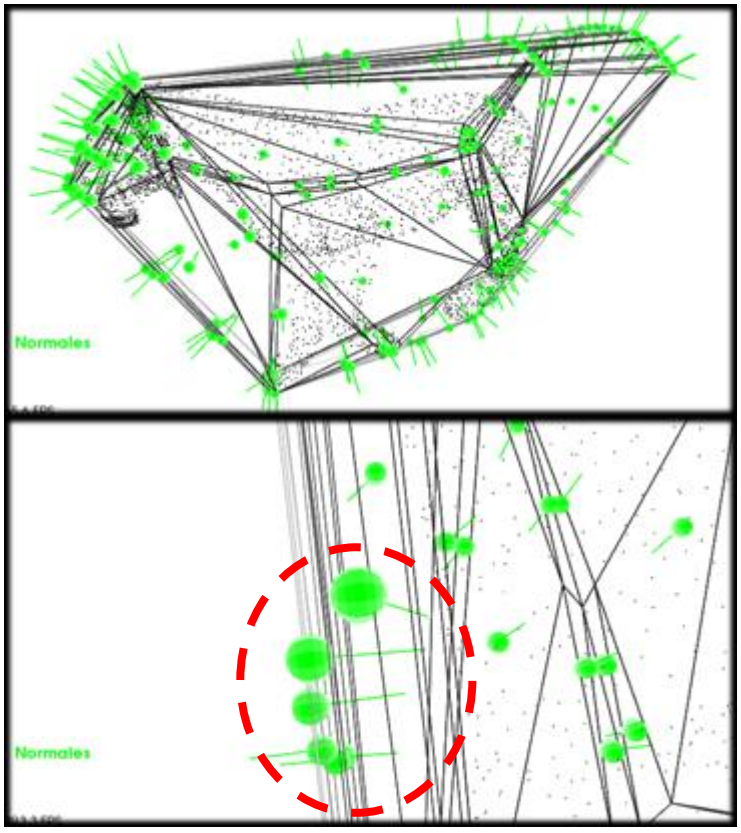


Ilustración 45: Envoltura convexa con normales incorrectamente orientadas..

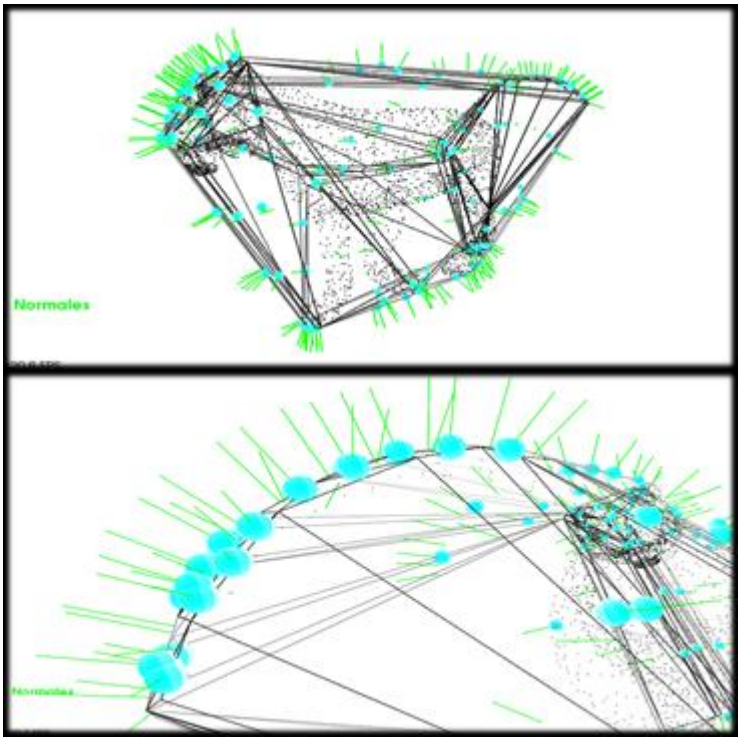


Ilustración 46: Normales correctamente orientadas

Objetos aislados

En primera instancia se realizó pruebas sobre nubes de puntos sin considerar el gripper. En la Ilustración 47 se muestra progresivamente el efecto del Algoritmo 7 sobre nubes de puntos de distintos objetos. De arriba a abajo, se grafica un cubo, una botella, un pato y un automóvil. Se puede ver de izquierda a derecha para cada caso los pasos de procesamiento para llegar desde la nube de puntos hasta el parche plano que definirá la pose estable. En la columna central se puede ver la notoria diferencia entre el centro de masas calculado con el Algoritmo 4 y el centroide de la nube, siendo el caso más notable el de la botella, que concentra más puntos en su parte superior, alejando notablemente su centroide del centro de equilibrio real.

La representación de nubes de puntos se generó a partir de objetos creados en el software *Strata Design 3D CX*, que permite exportar los modelos como una nube de puntos en formato `.obj`, cuya estructura interna extremadamente simple dio la posibilidad de crear un script conversor a formato `.pcd`, que es el admitido por PCL para importación de nubes de puntos.

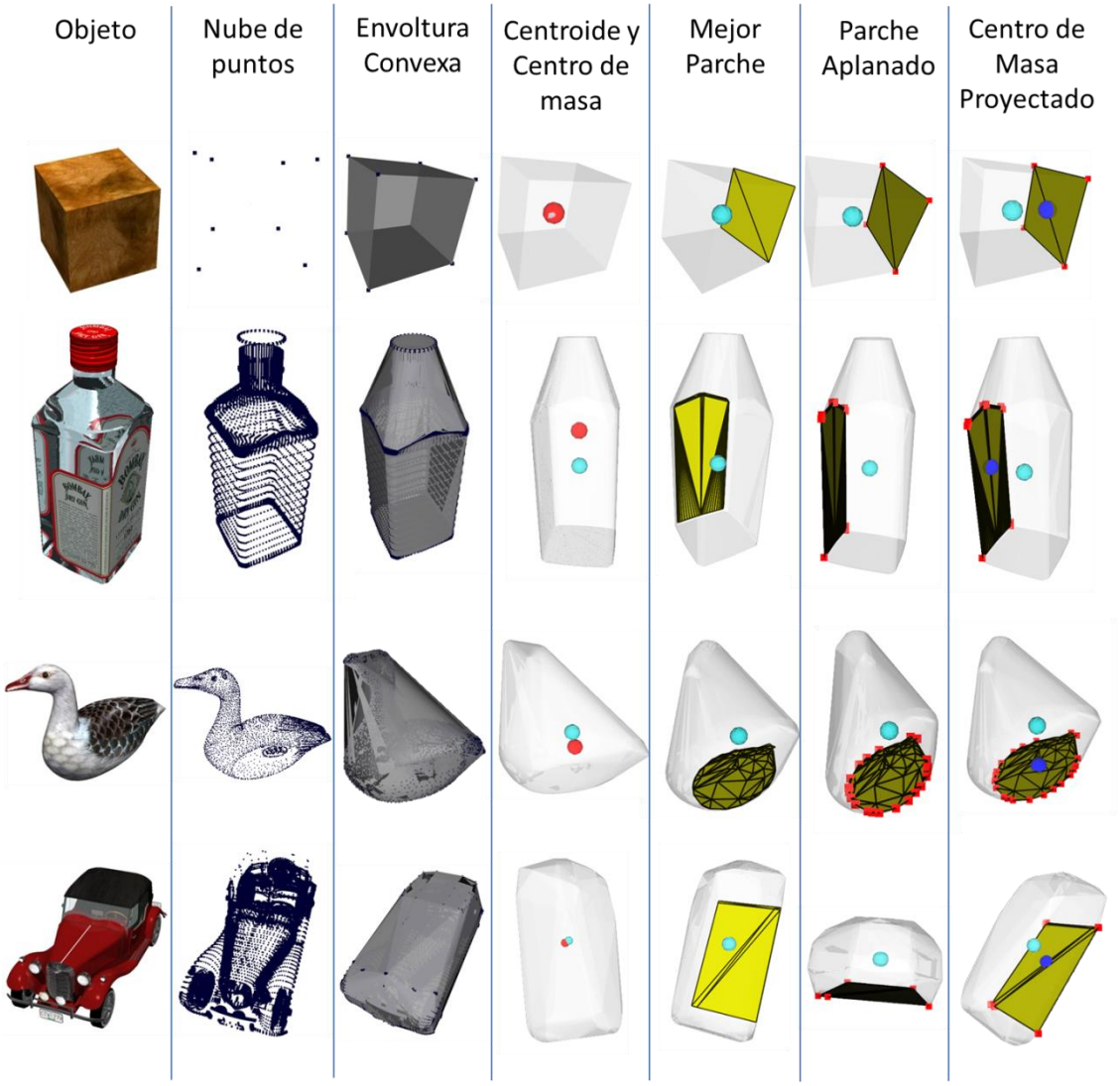


Ilustración 47: Sucesión de procesos aplicados para encontrar pose estable. Visualización con Strata Design y PCLViewer.

Objeto en gripper

Teniendo una versión digitalizada del objeto usando el Algoritmo 1, integrando la funcionalidad de búsqueda de pose estable en el programa principal y considerando el modelo del gripper, se obtiene lo que ilustra la Ilustración 48. A la izquierda, el caso de un cubo. A la derecha, un taladro inalámbrico, que representa un volumen más irregular.

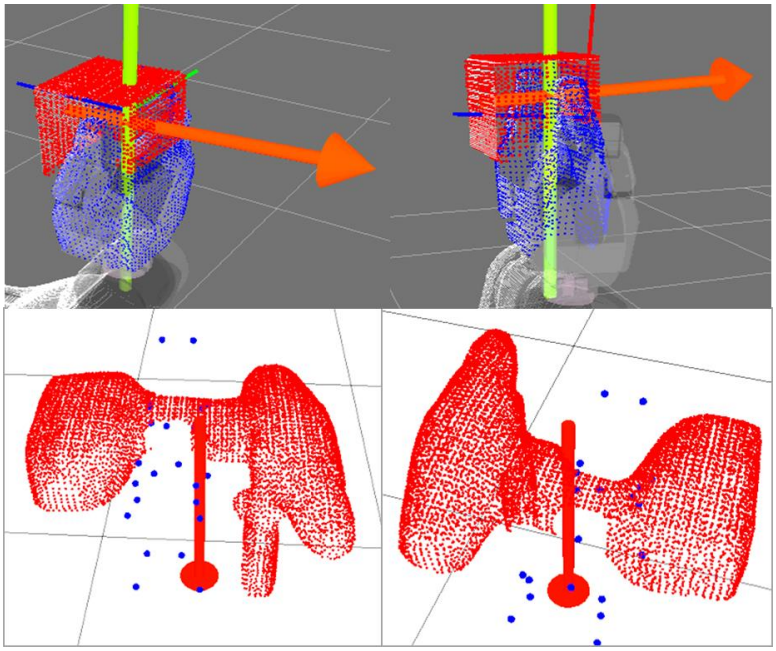


Ilustración 48: Pose estable detectada para un cubo (arriba) y para un taladro inalámbrico (abajo), para un agarre particular.

Agregando como Attached Collision Object

Aplicando el Algoritmo 8 sobre el modelo, se agrega el objeto al entorno de colisiones de MoveIt. La geometría resultante respeta las dimensiones reales del objeto manipulado. En la Ilustración 49 se ve el resultado para el taladro de la figura anterior, añadido como Attached Collision Object al gripper.

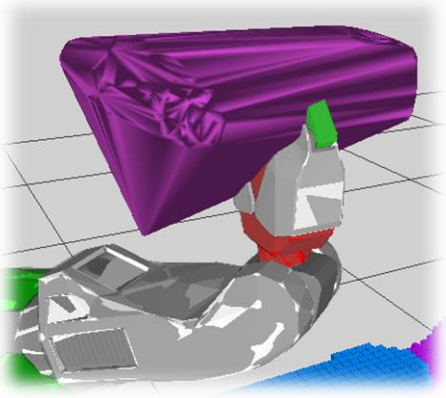


Ilustración 49: En color morado, el taladro representado como Collision Object en MoveIt.. En gris, rojo y verde, se ve la representación de las partes colisionables del brazo robótico.

Módulo 3: Búsqueda de superficie

La Ilustración 50 muestra el resultado de una etapa preliminar de ejecución del Algoritmo 9 de la subcomponente de búsqueda de superficie, donde el plano descartado (en azul) representando el suelo, se encuentra demasiado bajo, mientras que el escogido (en verde) está al alcance del robot.

El resultado final de la ejecución de este programa es una nube de puntos representantes de una superficie de posicionamiento adecuada para manipulación (Ilustración 51). El algoritmo fue capaz de detectar la superficie simulada incluso cuando ésta se encontraba a distancias lejanas (no más de 3.5 m), que son limitadas por el alcance del sensor Kinect, que por construcción es capaz de percibir puntos no más lejos de 4 m.

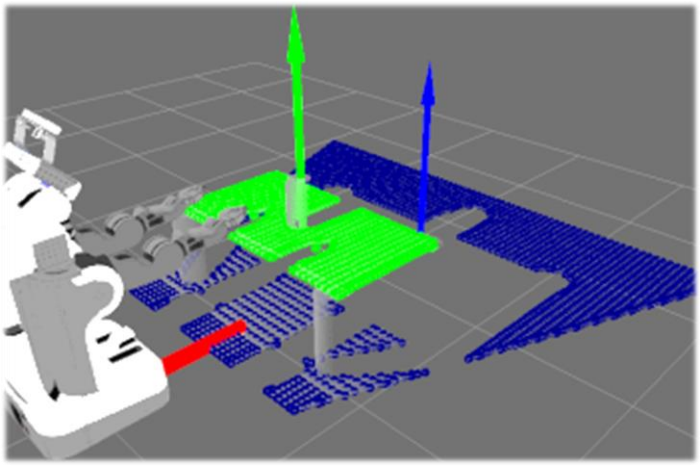


Ilustración 50: Planos encontrados por algoritmo de segmentación. En azul se muestran los puntos del plano encontrado en el suelo y su vector normal, y en verde el plano de una mesa junto a su vector normal.

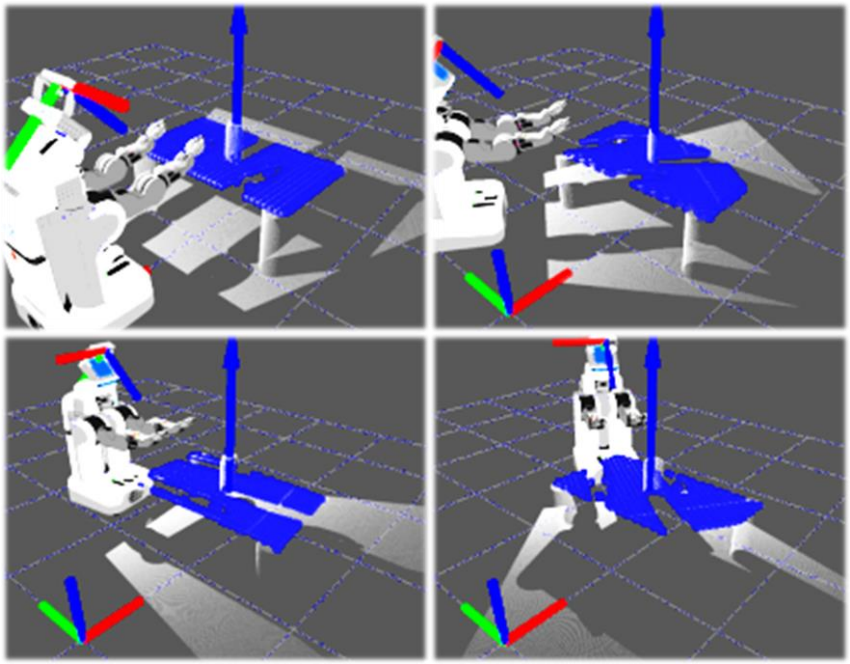


Ilustración 51: Búsqueda de superficies desde distintas ubicaciones. En azul se observa la nube de puntos que representa la superficie encontrada. Su vector normal se muestra como una flecha. En la esquina inferior izquierda de cada cuadro se muestra el origen del sistema de referencia odométrico.

En la Ilustración 52 se ve el resultado de agregarla la superficie como un Collision Object. Las irregularidades visibles corresponden a la forma de Rviz para visualizar mallas de triángulos, y no representa una rugosidad en la superficie.

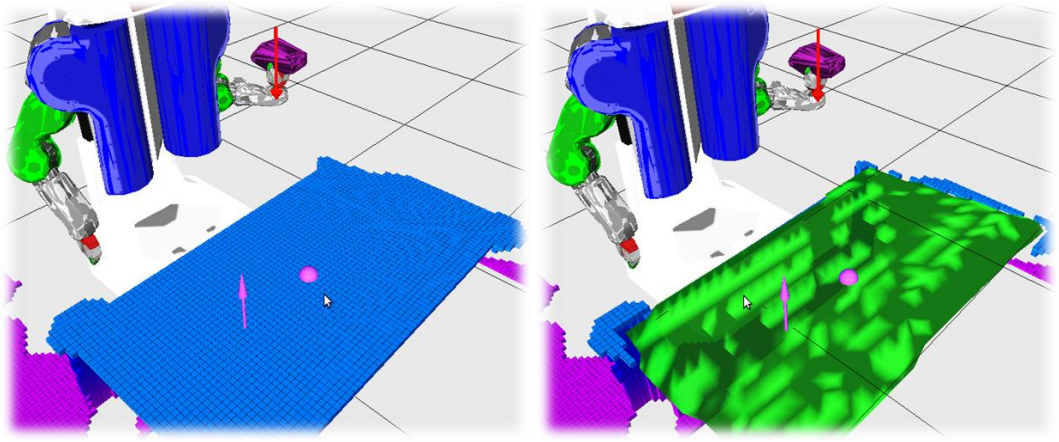


Ilustración 52: A la izquierda, vista de octomap antes de agregar superficie como Collision Object. A la derecha, en verde se ve superficie agregada. Las irregularidades son inherentes al dibujo de polígonos en Rviz y no a iregularidades reales en la geometría del plano.

Módulo 4: Desplazamiento hacia Superficie

La Ilustración 53 muestra un caso sencillo donde la superficie se encuentra cerca del robot (a no más de 1m). El robot calcula, ejecutando el Algoritmo 10, la pose final (representada por una flecha) que le indica cómo desplazarse y orientarse. Otro caso se puede ver en la Ilustración 54, donde la mesa se posicionó a aproximadamente 3m de distancia del punto de inicio del robot. El programa pudo encontrar la pose y consecuentemente la base del robot se desplazó hacia ella.

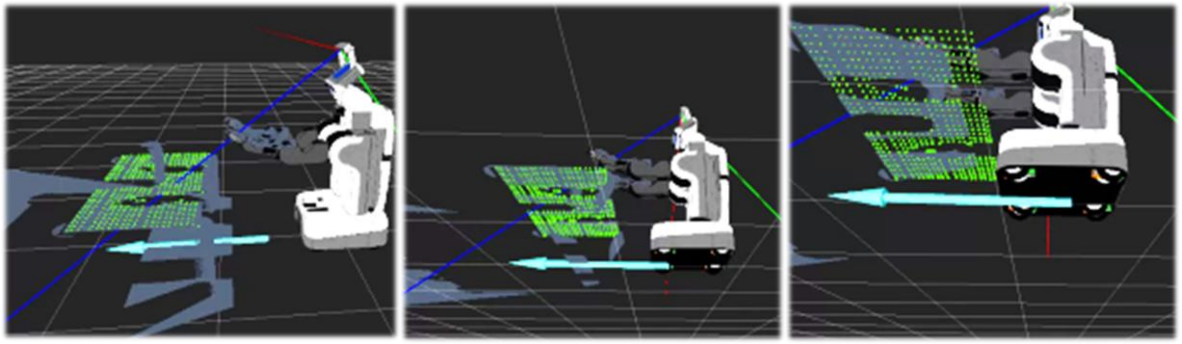


Ilustración 53: PR2 desplazándose hacia superficie encontrada. La pose final se representa por una flecha en color celeste.

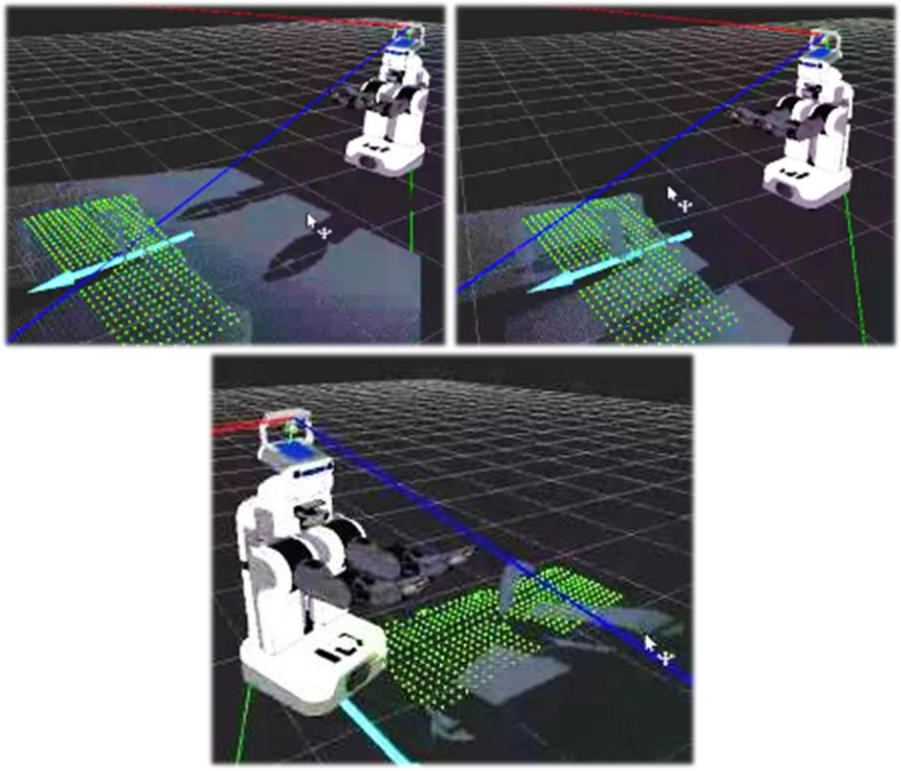


Ilustración 54: PR2 desplazándose hacia superficie encontrada, desde más de 3m.Pose final representada en flecha color celeste.

Módulo 5: Posicionamiento

El desempeño de este módulo engloba gran parte del rendimiento del algoritmo completo, por lo que su evaluación y discusión se realiza en la siguiente sección: Resultados Finales.

Resultados Finales

El programa posee tres variables principales interesantes de evaluar:

1. Desempeño en detección de distintas superficies
2. Desempeño en detección de superficie con objetos
3. Éxito de posicionamiento final

Se puede asumir con propiedad que existe un alto grado de independencia entre la componente de detección de superficies y la de determinación de pose estable, por lo que ambas pueden ser evaluadas por separado.

Detección de superficies

Para probar esta componente, se quitó la etapa inicial de detección de características del objeto, configurando el algoritmo para la búsqueda de superficies de al menos 100cm^2.

Cambiando posición

Para iniciar las pruebas, se comienza con un caso sencillo: Una mesa perfectamente plana, de generosa superficie y altura adecuada. En una primera instancia, se pone a prueba el robot situando la mesa frente a él a una distancia y orientación convenientes. Se observa en la Ilustración 55 la correcta detección. Al variar la ubicación de esta superficie, el algoritmo no tuvo mayores problemas, como se ve en la Ilustración 56.

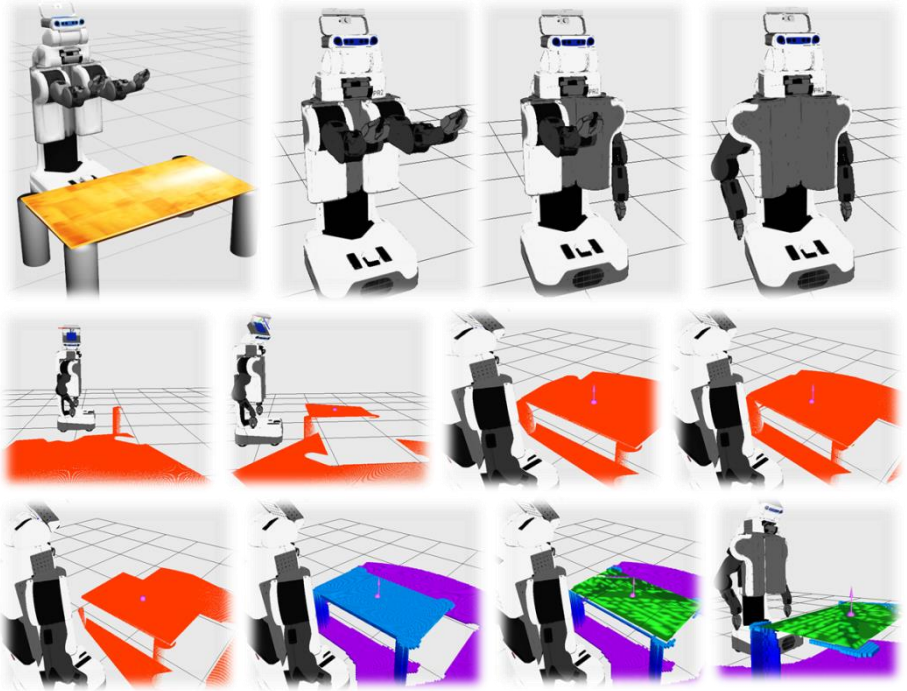


Ilustración 55: Algoritmo de búsqueda de superficie, ordenado de arriba a abajo, izquierda a derecha. El robot quita los brazos de la escena, para posar a buscar la superficie. Cuando la encuentra, la agrega como Collision Object a MoveIt.

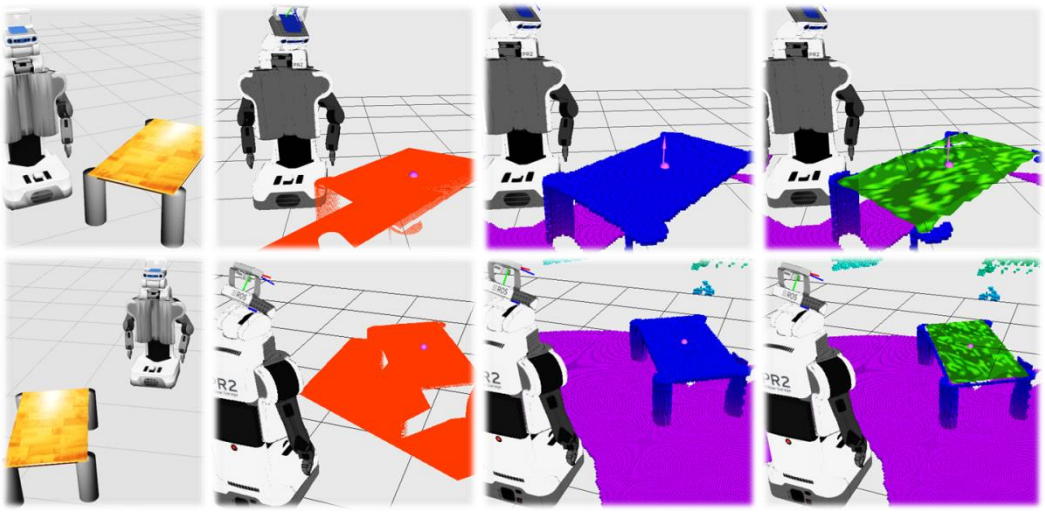


Ilustración 56: Resultado de búsqueda para distintas posiciones de la mesa. La primera columna muestra dos perspectivas vistas desde Gazebo. La segunda grafica la nube de puntos auto-filtrada. La tercera columna muestra la vista del octomap. La última columna muestra el octomap y el resultado de agregar la superficie encontrada como Collision Object, en color verde.

Cambiando tipo de superficie

En la Ilustración 57 se ve al robot identificando sin problema superficies con notorias áreas adecuadas. En el caso de objetos irregulares, como los de Ilustración 58 el algoritmo abortó tras no tener éxito en la búsqueda.

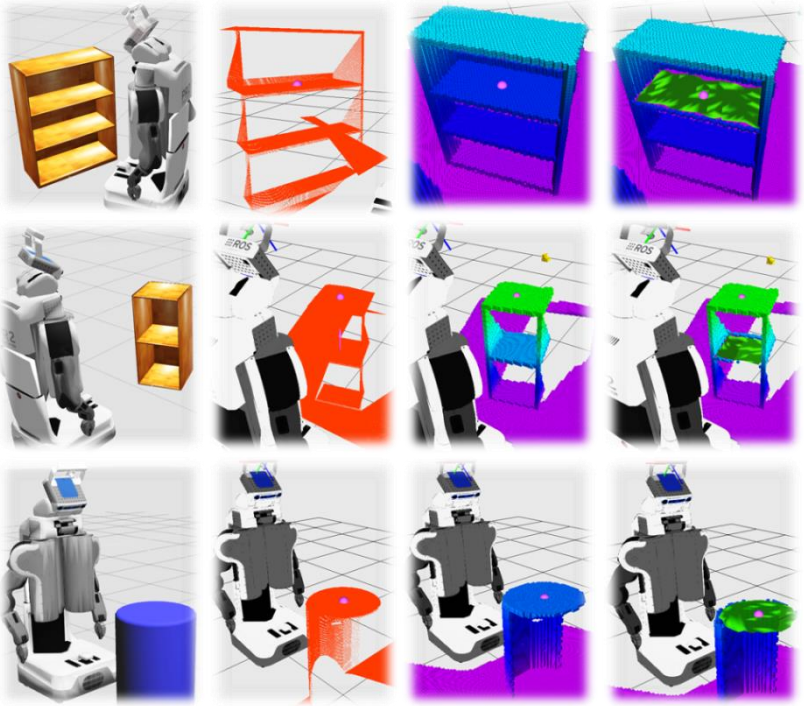


Ilustración 57: Otras superficies. En primera fila, un librero, en segunda fila un cabinete y en la tercera fila un tambor plástico. De izquierda a derecha, las columnas muestran, respectivamente, la perspectiva de gazebo de la escena, la nube de puntos auto-filtrada, la vista del octomap, y finalmente el octomap con la superficie como Collision Object en color verde.

Ilustración 58: Objetos no detectados como superficie. De izquierda a derecha, una rampa en 120°, un cono de construcción y un tambor de señalética.

Buscando entre varias superficies

Se creó un escenario que mezcla distintas superficies en un mismo lugar Ilustración 59. El programa, en su etapa de refinación de superficie encontrada (Algoritmo 9), cambió varias veces su elección de superficie, comportamiento que no es deseado. Al mismo tiempo, el programa identificó dos superficies disconexas como un mismo plano. Esto es esperado dentro de los parámetros de implementación, y su validez depende del usuario, pues podría perfectamente ser un caso válido.

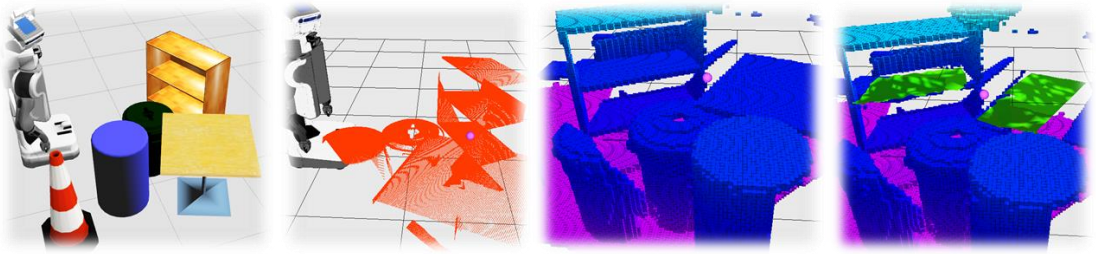


Ilustración 59: Búsqueda de superficie entre varias opciones. De izquierda a derecha: La escena vista desde Gazebo, la nube de puntos auto-filtrada, el octomap y la superficie encontrada en color verde.

Mesa con objetos

El programa se puso a prueba en su capacidad de detección de planos añadiendo ruido a la escena mediante el posicionamiento previo de objetos. En la Ilustración 60 se ve la robustez del método RANSAC para detectar el modelo entre datos heterogéneos.

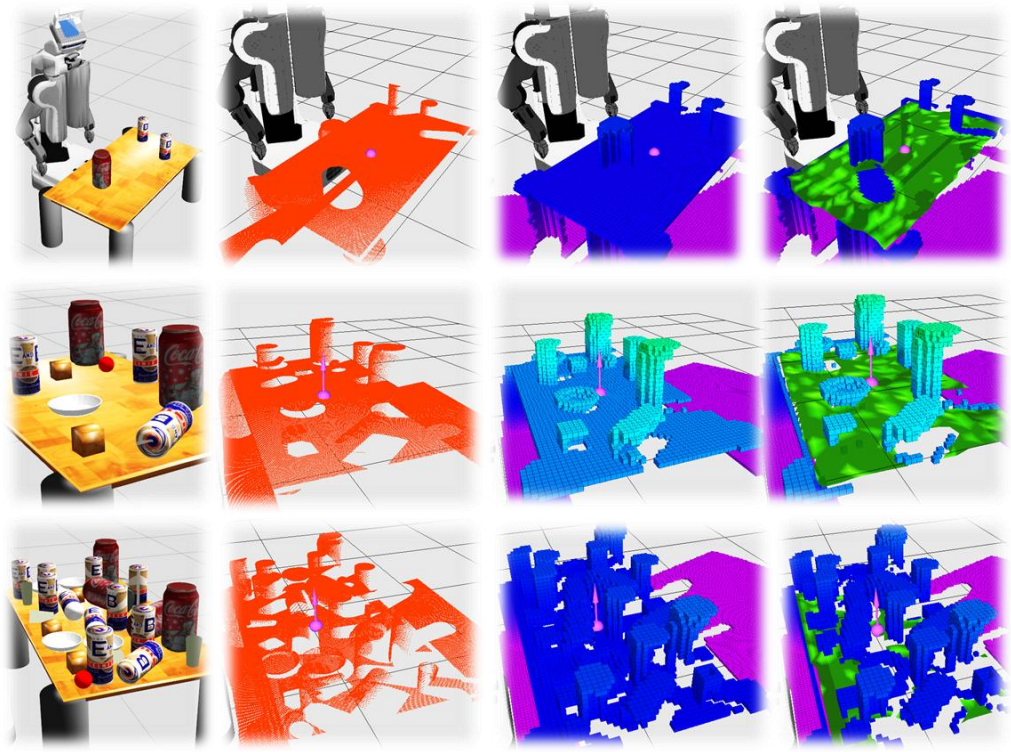


Ilustración 60: Superficie con objetos encima. Cada fila, de arriba abajo, grafica la misma mesa con ascendente cantidad de objetos y desorden. Cada columna de izquierda a derecha muestra la situación vista desde el simulador Gazebo, la nube de puntos auto-filtrada, el octomap generado y finalmente la superficie encontrada en color verde.

Posicionamiento

Para este punto, se utilizará la misma superficie plana, sin objetos encima.

Objetos Simples

Se probó inicialmente evaluar el posicionamiento de un cubo de madera por su simpleza geométrica. Ilustración 61 muestra el estado inicial, el resultado de su modelamiento y la pose estable encontrada, que será alineada con la superficie de apoyo.

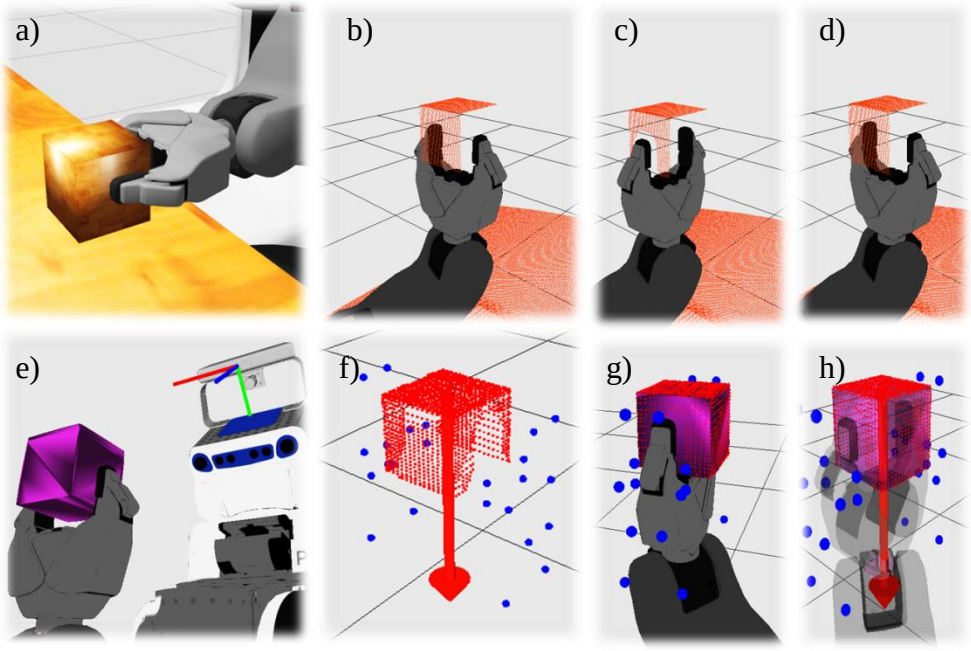


Ilustración 61: Sucesión de detección de pose estable para modelo del cubo. En a) se ve la situación inicial desde Gazebo. b), c) y d) muestran el proceso de digitalización. e) grafica el resultado de la digitalización como Attached Collision Object. f) muestra en rojo el objeto digitalizado y la pose del plano estable encontrado, y en azul los puntos del modelo del gripper. g) y h) muestran dos perspectivas adicionales incluyendo al gripper.

En la Ilustración 62 se ve la pose inicial del gripper, contrastada con la pose final testeada en ese instante dado, y la trayectoria descrita, para llegar a la posición final (Ilustración 63).

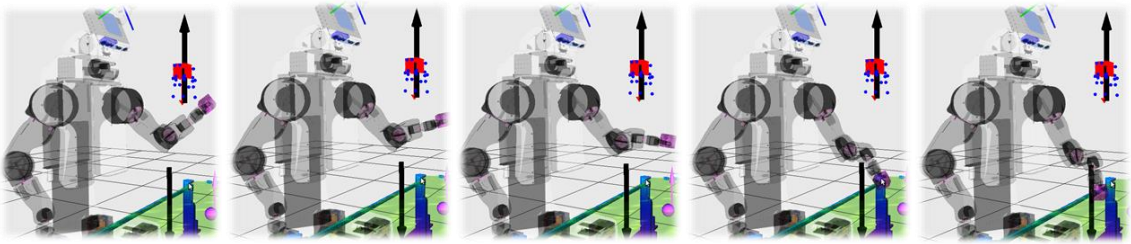


Ilustración 62: Sucesión de cuadros mostrando posicionamiento del cubo. Arriba a la derecha de cada cuadro se muestra en negro la pose inicial del gripper tras digitalización del objeto, y abajo la pose final calculada para el gripper.

Ilustración 63: Posicionamiento final del cubo

Se testeó con más elementos simples de formas cilíndricas y cuboides, todos mostrando resultados exitosos.

Objetos complejos

En la Ilustración 64 se ve el proceso de placing exitoso de un taladro inalámbrico, y en la Ilustración 65 para una manilla de puerta. En ambos casos los resultados fueron similares: correcta detección de mejor pose estable y elección exitosa de pose final para el gripper. Se puede ver en la Ilustración 66 las pruebas del planner sobre distintas rotaciones en torno a una misma pose candidata para el objeto.

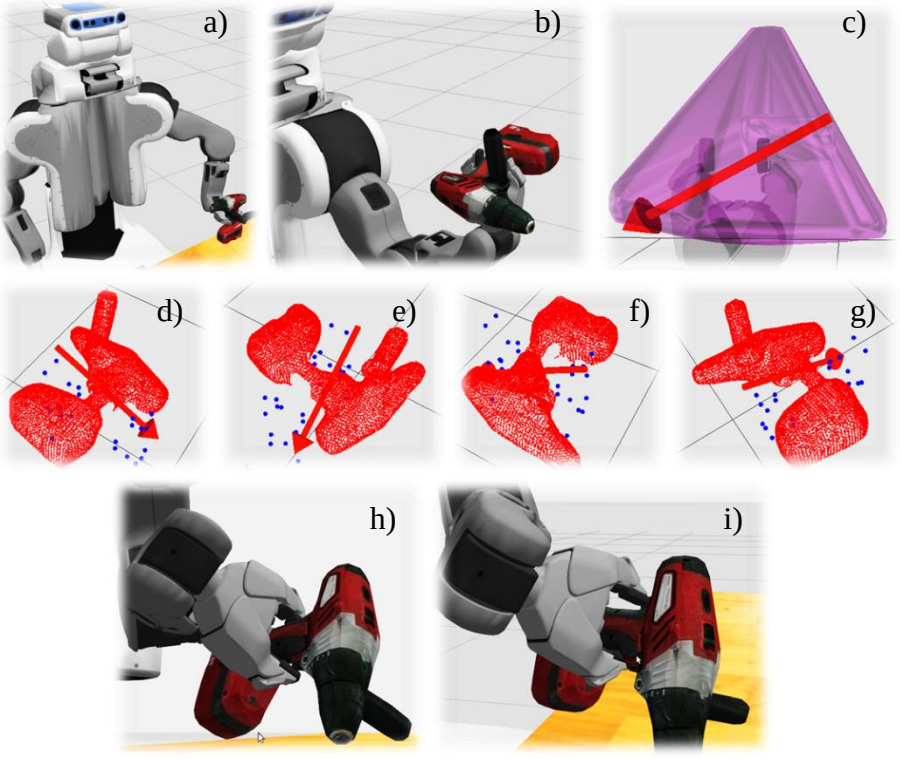


Ilustración 64: Placing de taladro inalámbrico. Se observa en a) y b) el escenario inicial y el proceso de digitalización desde Gazebo, que resulta en su representación como collision object visto en c) en color morado. d), e), f) y g) muestran distintas perspectivas de la nube de puntos aislada de la digitalización y su pose estable. h) muestra cuando se alinea el objeto con la superficie, para proceder a soltarlo, como se muestra en i).

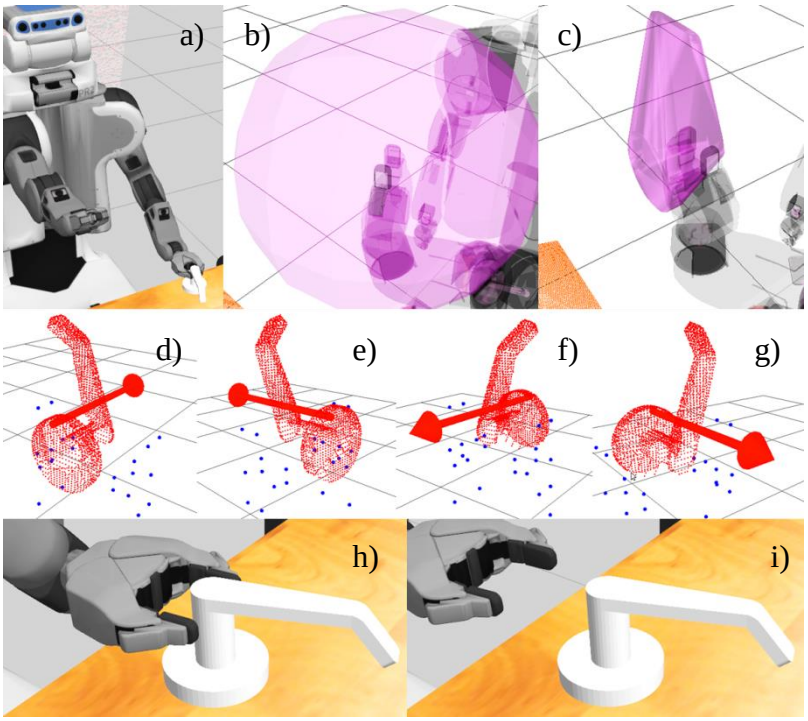


Ilustración 65: Placing de manilla de puerta. Se observa en a) el escenario inicial, en b) el proceso de digitalización, que resulta en su representación como collision object visto en c) en color morado. d), e), f) y g) muestran distintas perspectivas de la nube de puntos aislada de la digitalización y su pose estable. h) muestra cuando se posiciona el objeto y en i) se obserfa el paso posterior, cuando se suelta y se retira el gripper.

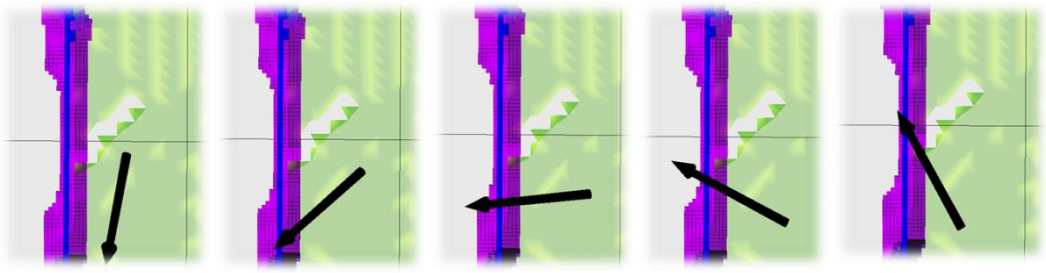


Ilustración 66: El algoritmo probando distintos ángulos de "roll" para una misma pose final preliminar del objeto. De izquierda a derecha se observa en negro la pose tentativa final del gripper, siendo rotada a medida que se descarta. En verde se ve la superficie como collision object, y en morado y azul el octomap.

La Ilustración 67 muestra un caso donde la posición final escogida por el algoritmo se encuentra demasiado a la orilla de la mesa. En este caso, si bien el centro de masas del objeto se ubica sobre la mesa, el movimiento del gripper al abrirse e intentar salir de la escena, perturbó el equilibrio de la manilla, haciéndola perder el equilibrio.

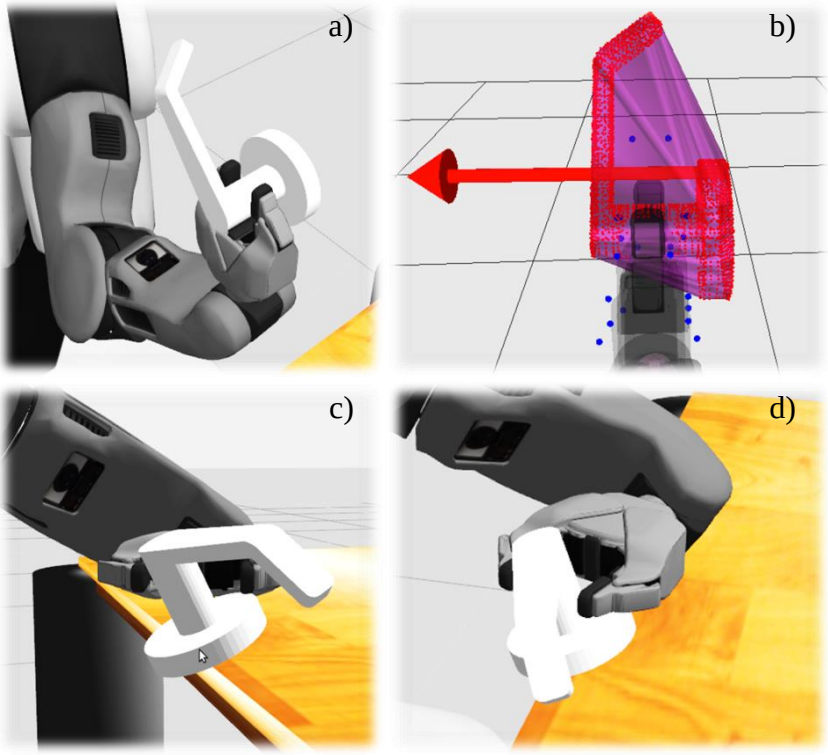


Ilustración 67: Placing de una manilla de puerta. Pose final escogida resulta peligrosa. En a) se ve el proceso de digitalización desde Gazebo, que resulta en lo mostrado en b). En c) y d) se ve desde Gazebo el posicionamiento del objeto a un borde de la superficie.

Constatación de requerimientos

A continuación se evalúa los requerimientos de aceptación de la solución presentados en el Capítulo 3, asignando a cada uno un valor en escala porcentual. Los puntajes asignados corresponden a una evaluación subjetiva de cumplimiento, basada en observaciones del desempeño general del algoritmo y sus partes.

Requerimientos no transables

- ***Detección superficie [60%]*:** La detección de superficies entrega resultados correctos en la mayoría de los casos más sencillos. Cuando se ingresa a la escena un alto nivel de caos, la utilidad de esta funcionalidad disminuye. Notar que este caos puede perfectamente ser considerado como normal en un ambiente doméstico, caso de uso pensado para esta aplicación.
- ***Representación tridimensional de objeto desconocido [Puntuación: 100%]***: El robot siempre pudo obtener una representación del objeto, que resultan ser de extrema utilidad, siempre que se respeten los supuestos iniciales.
- ***Determinación de pose estable [90%]:*** Se pueden tener situaciones donde una pose detectada como "estable" acerque peligrosamente al objeto a una posición inestable. Sin embargo, con el código generado, esto es fácilmente reparable.

- ***Posicionamiento en general [70%]:*** Este paso se ve negativamente afectado por la calidad de detección de superficie. El resto de las operaciones muestran ser eficientes y satisfactorias, siempre que se tuviera previamente un buen espacio para el placing.

Requerimientos Deseables

- ***Desplazamiento [40%]:*** Esta funcionalidad se desarrolló para suplir una preexistente con la que no se contó. El problema de navegación autónoma involucra muchas consideraciones que por sí solas definen un desafío enorme. La herramienta aquí implementada es simple y funcional sólo en entornos controlados.
- ***Implementación en PR2 real [10%]:*** Trabajar con esta máquina es complejo y requiere un nivel de conocimiento previo. Se encontró problemas al crear y ejecutar programas ROS en el PR2 del *Laboratorio RyCh*, pero el impedimento más notable fue el relativo al módulo TF, que mantiene datos que resultan de vital importancia para relacionar los frames de todos los links del robot. Al intentar utilizar esta funcionalidad se obtuvieron datos incorrectos siempre, siendo imposible continuar con el desarrollo.

Trabajo Futuro

En el desarrollo de las distintas subcomponentes de cada módulo, se pudo crear herramientas funcionales, pero en algunos casos evidentemente mejorables. En este apartado se propone para cada módulo distintas alternativas que pueden aumentar la calidad general del algoritmo de posicionamiento.

Módulo 1: Modelamiento del Objeto

Color

En todo el trabajo actual se usó modelos tridimensionales basados en puntos. En el proceso de captura del objeto se eliminó completamente la componente de color. Esta información podría ser rescatada para adquirir aún más características del objeto y mejorar la toma de decisiones respecto al *placing*.

PR2 real

Se generó y testeó el código en el entorno simulado Gazebo, que ejecuta sus versiones virtuales de todos los sensores, actuadores, y componentes ROS en general del PR2. Este está diseñado para proveer de un entorno lo más parecido a la realidad posible, prometiendo que al portar el código de algún proyecto hacia la máquina física, se ejecutará como se espera. Cuando se intentó realizar pruebas con el robot PR2 real en el laboratorio, se dieron situaciones problemáticas que no se pudo resolver para el momento de término de este trabajo.

Por todo esto, queda como trabajo futuro el resolver los impedimentos técnicos que ofrece el robot, para poder ejecutar este algoritmo en entornos reales. Existe la lista de correos "PR2_Users"[11] donde se comparten experiencias y dudas respecto al robot PR2 y ROS. En una oportunidad se envió un mensaje de ayuda a esta comunidad relativo a este trabajo expresando diversas inquietudes, pero pasaron 3 meses sin respuesta. Sin embargo, su reducida comunidad de 150 integrantes se mantiene visiblemente activa, pero en mensajes con problemas concisos y bien detallados. Se propone analizar en mayor detalle aún el comportamiento del PR2, usando toda la información obtenida para poder hacer preguntas más específicas a la comunidad, esperando así obtener feedback de sus experiencias y conocimientos.

Kinect virtual v/s real

Una parte de los componentes simulados por Gazebo son los sensores del robot, dentro de los que se incluye el Kinect. La implementación de este algoritmo de *placing* considera en sus parámetros ajustables el hecho de que este sensor no es perfecto, y la información de profundidad contiene un margen de error, sin embargo, la componente de error no es simulada por Gazebo. Su integración es posible, aunque no se realizó para priorizar otros aspectos.

[11]Acceso previa solicitud en https://groups.google.com/forum/#!forum/pr2_users.

Se propone integrar esta componente de error en el simulador para acercar más a la realidad los resultados de este algoritmo, previa ejecución en la máquina real.

Modelo previo

Para modelar un objeto en ROS existe el formato *SDF* que establece una serie de reglas para definir archivos en lenguaje de marcado, con sus especificaciones físicas. Se podría incluir soporte para objetos previamente conocidos a través de este mecanismo, que entrega más información sobre el objeto que tan solo realizar una captura tridimensional. De estos archivos se puede obtener información de geometría, matriz de inercia, centro de masas, componentes de roce, y otras características.

Módulo 2: Cálculo de pose estable

Parches

Todos los procesos computacionales a los que este programa somete al robot, resultaron ser siempre más rápidos de ejecutar que los procesos básicos que ejecuta el robot para tareas inevitables como la obtención de transformaciones espaciales y planeamiento de trayectorias, variando muy poco estos resultados al cambiar el número de datos procesados (disminuir submuestreo de la escena, adquirir más perspectivas del objeto). Con esto, realizar grandes optimizaciones al algoritmo no resulta de primera prioridad. No obstante se ofrece de todos modos a continuación un listado de posibles mejoras al algoritmo de búsqueda de parches planos y poses estables, pensadas para eliminar redundancias mejorar la calidad general:

- **Feedback sobre poses inestables:** Con todo lo implementado hasta el momento es posible determinar cuándo una pose es de baja estabilidad. Si un parche plano es muy pequeño y/o el centro de masas se proyecta muy cerca de su borde, podría alertarse al usuario que se está escogiendo una pose de baja estabilidad, o sencillamente descartar ese parche y buscar otro, más pequeño, pero que aleje el centro de masas de los bordes.
- **Nueva estructura de datos:** La búsqueda de triángulos contiguos se realizó utilizando un criterio de convexidad. Si se crea una nueva estructura de datos para los triángulos que incluya para cada uno una noción de los polígonos inmediatamente aledaños, podría cambiarse la forma en que se busca los parches, dejando de depender de las normales y pudiendo implementar algoritmos más eficientes basados en teoría de grafos.
- **Evaluación no exhaustiva de parches:** Se genera inicialmente una lista con todos los parches planos encontrados, para luego buscar uno a uno según sus condiciones de estabilidad. Podría en un futuro cambiarse este enfoque por uno heurístico, donde apenas se construya un parche plano, se evalúe su calidad para pose estable. Aunque no sería la pose más estable necesariamente, sí sería válida.
- **Soporte para superficies moderadamente inclinadas:** La estabilidad se valida proyectando el centro de masas sobre el parche plano en cuestión. Esta proyección se basa en la noción de que el vector peso del objeto posee una dirección suficientemente similar a la de la normal de la superficie. Cuando existe una inclinación suficientemente grande, se debe integrar en esta validación. En un futuro se puede relajar el supuesto de inclinación nula a través de la modificación de esta parte del programa.

Módulo 3: Búsqueda de Superficie

La superficie ideal

Los criterios de búsqueda de superficie utilizados en este módulo fueron: su altura, inclinación respecto al suelo y área total abarcada. Se puede prever problemas en este acercamiento.

- **Área abarcada:** El área total abarcada puede no representar el hecho de que exista efectivamente un espacio para posicionar el objeto. Considerar la Ilustración 68 que muestra en (a) una superficie con muchos objetos encima. El modelo plano que el algoritmo de segmentación se observa en (b), donde los objetos se ven como agujeros. Es claro que el objeto en (c) posee un área *no posicionable* en esta superficie, pero el criterio de área total abarcada entregará un falso positivo, afirmando que es adecuada.
- **Bordes:** El algoritmo de posicionamiento escoge cualquier punto alcanzable sobre la superficie para posicionar el objeto. En algunos casos podría darse que el equilibrio mecánico del objeto no sea tan estable en la pose entregada por el Algoritmo 7, y si a esto se añade que el Algoritmo 12 puede determinar que la mejor pose se encuentra muy cerca de un borde, podría caer, o quedar sencillamente en una posición peligrosa.

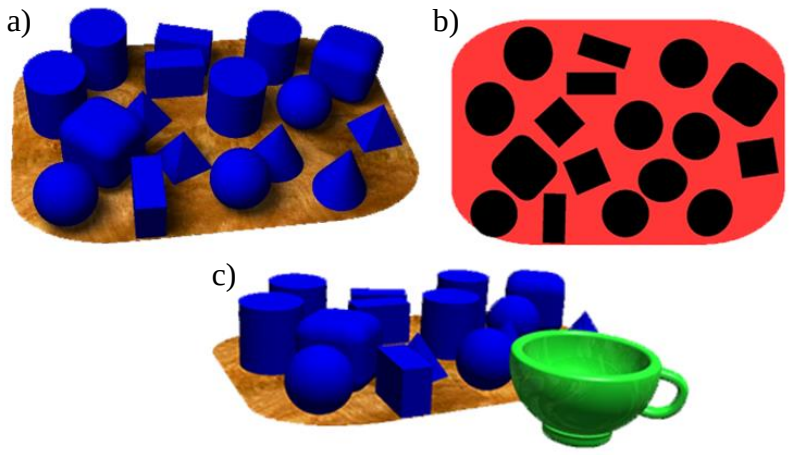


Ilustración 68: Ejemplo de superficie de posicionamiento muy poblada de objetos. En a) se muestra una perspectiva de la superficie. En b) una vista superior, que muestra en rojo las áreas utilizables para posicionamiento. c) muestra en verde un objeto (taza) que no cabría en esta superficie.

Queda como trabajo propuesto mejorar los criterios de aceptación de una superficie estable que aseguren que existe una región donde colocar el objeto y sea alcanzable por el robot. Es interesante considerar la posibilidad de seguir un acercamiento como el de Schuster et al. [17] para reconocer espacios convenientes de posicionamiento sobre una superficie altamente poblada de objetos, o el del trabajo de Emeli et al. [18] que aprovechan la posibilidad de mover estos objetos para forzadamente habilitar un espacio de posicionamiento.

Otras superficies

Todo el análisis se hace en torno a la idea de una superficie plana. Podría ser de interés investigar el caso de superficies con formas más complejas, dependiendo del contexto en que se pretenda usar el algoritmo. En un ambiente doméstico las superficies planas representan gran parte de los posibles lugares de posicionamiento de objetos, pero, tal como se estudia en el trabajo de

Jiang et-al en [19], existen otros que resultan de interés, los que podrían en un futuro ser estudiados e integrados como opciones en el programa.

Módulo 5: Posicionamiento

Reducción de soluciones de pose final

En la creación del conjunto de poses posibles, se reduce las posibilidades considerando sólo el espacio de la superficie al que el robot podría llegar con el brazo, y para cada uno de los puntos que lo componen, se considera una partición equiespaciada de ángulos de *roll* para la tenaza activa, en torno a la pose final del objeto. Hay casos donde muchos de estos ángulos pueden ser descartados, por ejemplo, siempre que el brazo está muy estirado, el gripper no podrá cubrir todas las rotaciones posibles, por implicar alejarse demasiado. Podría crearse alguna rutina que verifique estos casos y los elimine del espacio de posibilidades, pero ésta tendría que considerar además características propias del brazo robótico, como los límites de sus joints, posición de los links e incluso resultados de trayectorias de MoveIt. Por esto, se delegó la responsabilidad de descartar estas poses al planner.

Aprendizaje

El software no hace nada con los resultados obtenidos, tenga éxito o no. Podría añadirse una componente de *machine learning* al programa para mejorar cada vez más estos resultados, basándose en experiencias anteriores.

Integración de módulos

Esta etapa se pensó en el caso particular de la resolución del problema de *placing*, pero sin duda varias partes del código podrían beneficiar el desarrollo de otros proyectos diferentes.

Se propone re-implementar el sistema de servicios, pero esta vez modularizando aún más las funcionalidades del programa en general. Este proceso podría resultar en la generación de:

- Interfaces de nivel superior para trabajar con el PR2, disponibles para todos los potenciales nodos presentes en una instancia dada de la red distribuida ROS
- Más métodos para ejecutar transformaciones espaciales
- Métodos para interacción más intuitiva con MoveIt
- Herramientas de análisis de características de objetos representados sólo por nubes de puntos
- Software de digitalización de objetos manipulados

Conclusión

En Chile, la comunidad de desarrolladores para robótica, específicamente para la plataforma utilizada para implementar este proyecto, no es muy grande, por lo que todo aporte sobre el tema significa una contribución valiosa a la nutrición de las bases de conocimiento sobre robótica autónoma. La Universidad de Chile, en particular las personas que integran al *Laboratorio RyCh*, han trabajado en pos de este mismo objetivo.

Las herramientas de desarrollo empleadas para la presente investigación son mantenidas por agrupaciones independientes de diversos lugares del mundo, mayormente universidades, incluida la nuestra.

A pesar de lo anterior, la difícil accesibilidad al robot limita el número de personas que pueden participar, por lo que se hace difícil cubrir la totalidad de las eventualidades que surgen por el uso de un sistema tan complejo, al contrario de lo que sucede en el caso de otros proyectos, también de característica Open Source, como por ejemplo, el Sistema Operativo Linux.

Con todo esto y pese a las dificultades relatadas, se pudo obtener resultados satisfactorios, coherentes con el objetivo. El proceso de desarrollo y resolución del problema brindó la posibilidad de superar gran cantidad de desafíos que no solo generó conocimiento útil para el alumno memorista, sino que además queda como una herramienta operativa para el equipo del *Laboratorio RyCh* que hasta hoy no contaba con esta funcionalidad.

El robot simulado fue capaz de identificar poses convenientes de posicionamiento en la totalidad de los casos estudiados. La gran limitante al momento de completar la tarea fue la selección de un lugar de posicionamiento adecuado, que constituye un problema no trivial, situación que la metodología utilizada no pudo abordar completamente.

Se entendió que para poder desarrollar en sistemas mantenidos por comunidades pequeñas, hay que agotar todos los recursos posibles de información cuando se tengan problemas, apuntando a retroalimentar al ecosistema con la experiencia adquirida.

Para finalizar, cabe destacar lo fundamental que resultó el apoyo de todo el equipo de personas que, con su experiencia, contribuyó a hacer de este proyecto un desafío significativo para el crecimiento personal y profesional de quien suscribe.

Bibliografía

[1] Quigley, Gerkey, Conley, Faust, Foote, Leibs, Berger, Wheeler, Andrew Ng: "ROS: an open-source Robot Operating System", año 2009.

[2] Rusu, Cousins: 3D is here: Point Cloud Library (PCL), año 2011.

[3] Sucan, Chitta: "MoveIt!", año 2012.

[4] Koenig, Howard: "Design and Use Paradigms for Gazebo, an Open-Source Multi-Robot Simulator", año 2004.

[5] Holz, Holzer, Bogdan Rusu, Behkne: "Real-Time Plane Segmentation Using RGB-D Cameras", año 2012.

[6] Mishra, Shrivastava, Aloimonos: "Segmenting 'Simple' Objects Using RGB-D", año 2012.

[7] Carson, Belongie, Greenspan, Malik: "Blobworld: Image Segmentation Using Expectation-Maximization and Its Application to Image Querying", año 2002.

[8] Silberman, Hoiem, Kohli, Fergus: "Indoor Segmentation and Support Inference from RGBD Images", año 2012.

[9] Chan, Parker, Van Der Loos, Croft: "Grip Forces and Load Forces in Handovers: Implications for Designing Human-Robot Handover Controllers", año 2012.

[10] Pandey, Saut, Sidobre, Alami: "Towards Planning Human-Robot Interactive Manipulation Tasks: Task Dependent and Human Oriented Autonomous Selection of Grasp and Placement", año 2012.

[11] Stueckler, Steffens, Holz, Behnke: "Real-Time 3D Perception and Efficient Grasp Planning for Everyday Manipulation Tasks", año 2011.

[12] Edsinger, Kemp: "Manipulation in Human Environments", año 2006.

[13] Edsinger, Kemp: "Challenges for Robot Manipulation in Human Environments", año 2007.

[14] Holladay, Barry, Kaelbling, Lozano-Pérez: "Object Placement as Inverse Motion Planning". año 2013.

[15] Cowley, Cohen, Marshall, Taylor, Likhachev: "Perception and Motion Planning for Pick-and-Place of Dynamic Objects", año 2013.

[16] Harada, Foissotte, Tsuji, Nagata, Yamanobe, Nakamura, Kawai: "Pick and Place Planning for Dual-Arm Manipulators", año 2012.

[17] Schuster, Okerman, Nguyen, Rehg, Kemp: "Perceiving Clutter and Surfaces for Object Placement in Indoor Environments", año 2010.

[18] Emeli, Kemp, Stilman: "Push Planning for Object Placement in Clutter Using the PR-2", año 2011.

[19] Jiang, Zheng, Lim, Saxena: "Learning to Place New Objects", año 2012.

[20] Nguyen, Anderson, Trevor, Jain, Xu, Kemp: "El-E: An Assistive Robot that Fetches Objects from Flat Surfaces", año 2009.

[21] Choi, Chen, Jain, Anderson, Glass, Kemp: "Hand It Over or Set It Down: A User Study of Object Delivery with an Assistive Mobile Manipulator", año 2009.

[22] Estrela, Cámara-Chávez, Campos, Schwartz, Nascimento: "Sign Language Recognition using Partial Least Squares and RGB-D Information", año 2013.

[23] Ramírez-Hernández, Marín-Hernández, García-Vega: “Hand Detection for Human Robot Interaction”, año 2011.
[24] Edsinger, Kemp: “Human-Robot Interaction for Cooperative Manipulation: Handing Objects to One Another”, año 2007.
[25] Kehoe, Matsukawa, Candido, Kuffner, Goldberg: “Cloud-Based Robot Grasping with the Google Object Recognition Engine”, año 2013.

Anexos

A. Selección de planner para MoveIt

En el sitio web oficial de desarrollo de los planners[12] se encuentra un análisis de ocho de ellos: *RRTConnect, RRT, BKPIECE1, LBKPIECE1, KPIECE1, SBL, EST* y *PRM*, que son usados con dos programas de prueba para luego evaluar sus desempeños comparativamente. Cada uno de ellos aborda el problema de planeamiento de trayectorias usando distintas estrategias, que pueden ser complejas de entender. Basta con ver los resultados de estas pruebas para tomar una decisión respecto a qué planner usar.

La Ilustración 69 muestra la cantidad de estados del grafo generado (arriba izquierda), la memoria utilizada (arriba derecha), tiempo invertido (abajo izquierda) y tiempo de simplificación de trayectoria (abajo derecha) para los ocho planners. Esta información motiva el uso de *RTTConnect* en este proyecto.

[12] Disponible en http://ompl.kavrakilab.org/

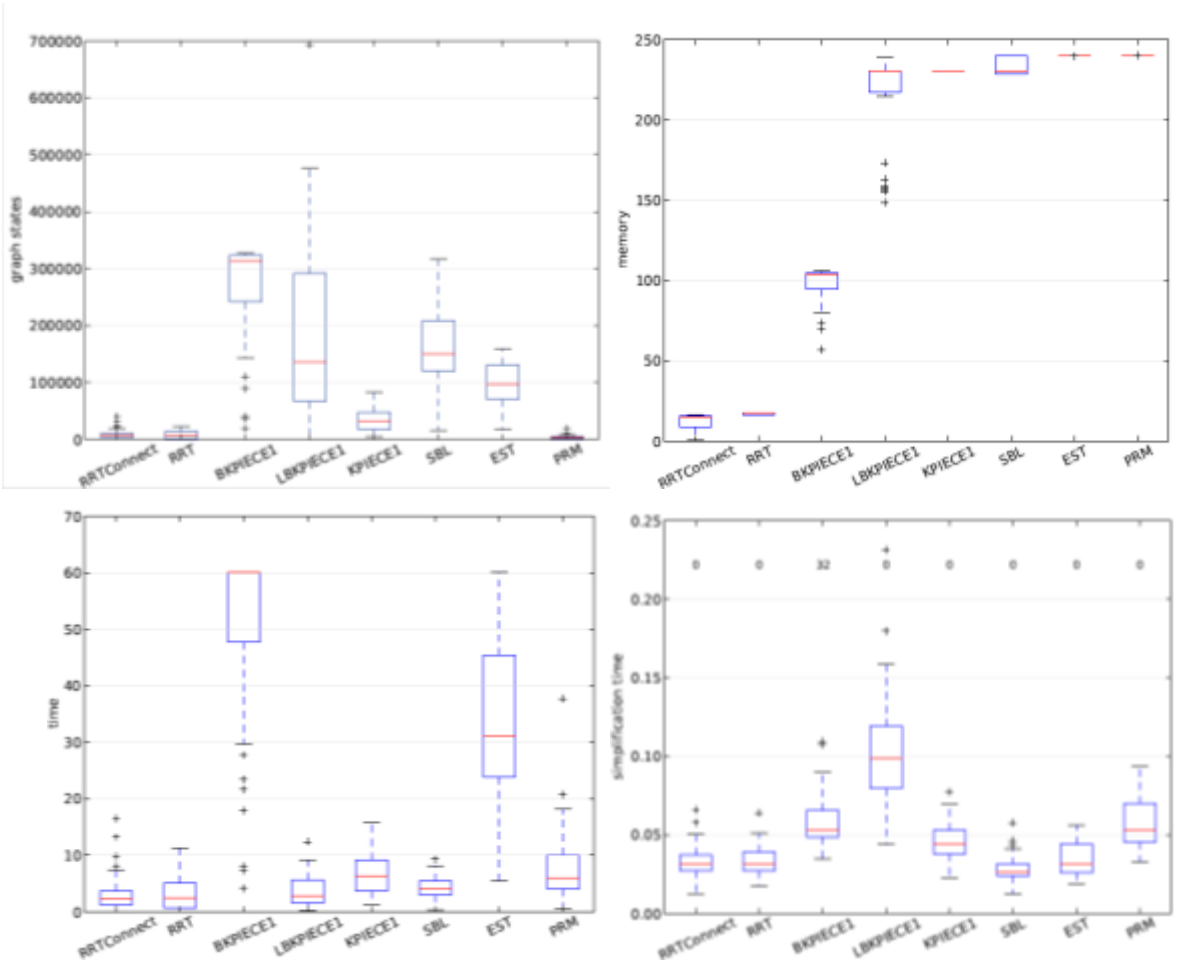


Ilustración 69: Benchmark comparativo de desempeño de planners

B. Filtrado de Gripper

En una primera instancia ninguna funcionalidad existente de auto-filtrado de nubes de puntos había resultado útil, situación ante la que surgió la necesidad de crearlo manualmente. El razonamiento es:

1. Realizar tantas capturas como perspectivas se quieran del objeto
2. Sintetizarlas en una sola nube de puntos alineada
3. Eliminar todo punto suficientemente lejano del centro (límite fijado en las acotaciones del problema)
4. Crear modelo de cajas que encierren a la tenaza (ver **Subcomponente n°1**: Modelamiento del gripper del Módulo 2)
5. Separar todo punto al interior de las cajas de los del exterior y obtener dos nubes de puntos, una para el modelo de la tenaza y otra para el del objeto

Se implementó estas tareas en el Algoritmo 13. Para casos sencillos, este acercamiento logró resultados favorables. En la Ilustración 70 se puede ver el resultado de aplicar las operaciones anteriores a la adquisición de puntos de un gripper que sostiene un cubo de madera, en el entorno Gazebo.

Algoritmo 13: Auto-filtro manual

```
INPUT:
        - Nube de puntos con captura de gripper y objeto
OUTPUT:
        - Nube de puntos con representación 3D del gripper
        - Nube de puntos con representación 3D del objeto
DEFINICIÓN:
        1.      Inicializar conjunto de cajas (definición empírica)
        2.      Inicializar conjuntos vacíos de puntos inliers y outliers
        3.      Borrar puntos bajo la muñeca del gripper (filtro adicional)
        4.      for (puntos dentro de la nube):
                4.1.    if (punto dentro de alguna caja):
                    4.1.1.              añadir punto a conjunto de inliers
        5.      Agregar puntos restantes a conjunto de outliers
        6.      return gripper (inliers)
        7.      return objeto (outliers)
```

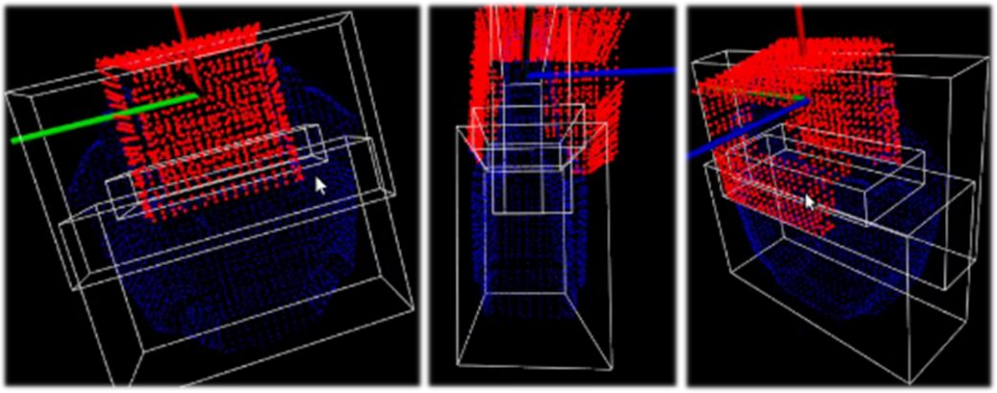


Ilustración 70: Resultado de auto-filtrado de gripper

Este método cuenta con que se conoce de manera bastante precisa la posición del gripper en el espacio, pero el resto del cuerpo del robot puede estar en muchas otras ubicaciones relativas a la tenaza, pudiendo eventualmente aparecer en la captura de puntos. Además, para contrarrestar el error inherente a los datos adquiridos, las cajas deben cubrir un volumen mayor al real, disminuyendo la exactitud de este filtro y borrando parte del objeto.

Siguiendo este enfoque puramente de procesamiento de nubes de puntos, existen ciertas alternativas a probar:

1) ***Separar grupos de puntos con técnicas de* clustering:** PCL, a través de su módulo *Segmentation* permite aplicar implementaciones de algoritmos de clustering, capaz de detectar en una nube de puntos distintos grupos según ciertos criterios, como pueden ser las distancias entre ellos, la tasa de cambio entre normales de una triangulación, entre otros. Con esta técnica se podría separar todos los grupos relativamente disconexos y priorizar aquel grupo con puntos cercanos al origen del sistema de referencias, originalmente posicionado al centro del gripper. Esto asume que la representación del objeto consta de puntos repartidos de forma considerablemente homogénea sobre su superficie.
2) ***Considerar componente de color:*** El sensor Kinect captura datos de tipo RGB-D, es decir, imágenes coloreadas con una componente de profundidad, siendo esta última la única utilizada en este trabajo. Aplicando técnicas de segmentación similares al trabajo de Holz et-al en [5] y Mishra et-al en [6] podría mejorarse la calidad del filtro.

Este no es el único enfoque posible, pues se puede imitar el comportamiento de Robot-self-filter de ROS o del de MoveIt, que usan la representación computacional del robot para saber exactamente qué region del espacio ocupan los links del PR2 en tiempo real. Se decidió no hacer esto en virtud del tiempo.