Proyecto Fin de Grado
Grado en Ingeniería en Tecnologías Industriales

Reconocimiento gestual para interacción humano-robot basado en ROS

Autor: José Andrés Millán Romera

Tutor: D. Guillermo Heredia Benot

Ingeniería de Sistemas y Automática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2015

Proyecto Fin de Grado

Grado en Ingeniería en Tecnologías Industriales

Reconocimiento gestual para interacción humano-robot basado en ROS

Autor:

José Andrés Millán Romera

Tutor:

D. Guillermo Heredia Benot

Profesor Titular

Ingeniería de Sistemas y Automática

Escuela Técnica Superior de Ingeniería

Universidad de Sevilla

Sevilla, 2015

Proyecto Fin de Grado: Reconocimiento gestual para interacción humano-robot basado en ROS

Autor: José Andrés Millán Romera
Tutor: D. Guillermo Heredia Benot

El tribunal nombrado para juzgar el trabajo arriba indicado, compuesto por los siguientes profesores:

Presidente:

Vocal/es:

Secretario:

acuerdan otorgarle la calificación de:

El Secretario del Tribunal

Fecha:

Índice

1 Resumen **1**

2 Introducción **3**

2.1 Alcance 3

2.2 ¿Qué es EuRoC? 3

2.3 Resumen 4

2.4 Objetivos 4

2.4.1 Objetivo 1: Lanzamiento y ejecución de tres desafíos industrialmente relevantes 4

2.4.2 Objetivo 2: Apoderamiento de plataformas robóticas e infraestructuras de referencia 5

2.4.3 Objetivo 3: Sostenibilidad y adaptabilidad a los usuarios finales 5

2.5 Challenge 1 5

3 Estado del arte **7**

3.1 Robótica 7

3.2 Reconocimiento gestual 8

3.3 Simulación Robótica 10

4 Desarrollos teóricos para el tratamiento de nubes de puntos **13**

4.1 Representación de la posición y la orientación 13

4.2 Algoritmo de correspondencia de nubes de puntos 16

5 ROS y Herramientas **21**

5.1 ROS 21

5.1.1 Diseño modular y distribuido 22

5.1.2 Historia 23

5.1.3 Comunidad activa y colaborativa 23

5.1.4 Conceptos básicos 24

5.2 Herramientas ROS 25

5.2.1 Gazebo 25

5.2.2 Rviz 26

5.2.3 TF 27

5.2.4 Mercurial 27

6 Descripción del entorno de simulación **29**
6.1 Cámaras 30
6.2 Humano 30
6.3 Pieza de trabajo 30
6.4 Brazo robótico 31
7 Evaluación de las tareas **33**
7.1 Subtarea 1. Reconocimiento gestual del humano 34
7.2 Subtarea 2. Localización de la pieza de trabajo 35
7.3 Subtarea 3. Estimación de la posición apuntada 36
8 Tratamiento de la nube de puntos de la cámara Kinect **39**
8.1 Nociones iniciales 39
8.2 Subtarea 2. Localización de la pieza de trabajo 42
8.3 Detección de la pieza 44
8.4 Subtarea 1. Reconocimiento gestual del humano 47
8.5 Detección de la mano 50
8.6 Subtarea 3. Estimación de la posición apuntada 53
8.7 Detección del punto señalado 55
9 Conclusiones y mejoras **57**
10 Anexo 1: Arranque del sistema **61**
11 Anexo 2: Código **63**
11.1 vision_stereo_node 63
11.2 detect_pointing 73
11.3 algorithm 76

1 Resumen

El objeto de esta memoria es la descripción de la solución al problema de identificación y posicionamiento del gesto de señalización de un ser humano en una estación de trabajo mediante una cámara Kinect, todo ello simulado en un entorno virtual en ROS. El sentido de este trabajo es su localización dentro del marco del concurso de robótica a nivel europeo EuRoC, en el que participé junto con mis compañeros de equipo del Centro Avanzado de Tecnologías Aeroespaciales (CATEC) y dentro del cual yo me encargué de la primera tarea (dividida en tres subtareas) que me dispongo a presentar.

En primer lugar se describe el contexto en el que se encuentra el trabajo para poder entender bien el porqué de sus objetivos, de sus formas de implementación y de su entorno de trabajo.

El segundo gran bloque habla del estado del arte para poder entender cual es el estado actual tanto de la robótica como del tratamiento de nubes de puntos y reconocimiento gestual. Su fin es el de describir con qué tecnología contamos, cuál es el partido que le podemos sacar a ésta y en qué caminos debemos aún seguir avanzando para logar optimizar operaciones de la robótica como la que aquí se presenta u otras muchas.

Seguidamente se explican unos conceptos básicos de los desarrollos matemáticos que emplean las herramientas de trabajo con puntos en el espacio aquí usadas. Creo que, aunque éstas nos faciliten tanto el trabajo con algoritmos ya desarrollados y de fácil implementación, siempre es necesario comprender qué es lo que se está empleando para un uso óptimo de las mismas.

A continuación, se expone qué es ROS y cuáles son las herramientas que nos ofrece para alcanzar nuestro objetivo para poder entender cuál es la metodología de trabajo que sigo, cuáles son los canales de comunicación de ROS que empleo y qué partes del problema solucionan estas herramientas y cuáles tengo yo que solucionar con ellas.

Justo antes de entrar en faena, es necesario analizar el entorno de simulación. Esto es útil para ver bien cuáles son los objetivos de cada tarea y los elementos físicos y sus movimientos con los que tendremos que lidiar durante el desarrollo de las mismas. Una vez comprendido esto y sus objetivos, vemos la forma que tendrá el concurso de corregirnos y cuáles serán los criterios de evaluación.

Ahora sí, vemos cuál es la solución que aporto a cada una de las necesidades que van surgiendo durante el desarrollo de cada una de las tareas. Para ello, son desplegados seis diagramas de flujos en los que intento expresar de la mejor forma posible cómo se va tratando la información obtenida mendiante la cámara Kinect, siempre enlazando con el

código real que implementa la solución de los diagramas, y cómo van variando algunas de las variables de control más importantes.

Para finalizar, son adjuntados dos anexos. En el primero de ellos explicaré cómo se inicia la máquina virtual para que se comprenda dónde está situada y cómo se va a ejecutar la solución a las diferentes tareas aportada por nuestro equipo. En el segundo código incluyo un extracto o selección de la parte del código que considero más importante para la clara comprensión de la implementación de la solución, aunque dejando comentarios de todo el resto de código.

2 Introducción

2.1 Alcance

El proyecto que presento tiene como contexto el concurso de Robótica a nivel Europeo de la organización EuRoC [1], desarrollada en el Área de Automatización y Robótica del Centro Avanzado de Tecnologías Aeroespaciales (CATEC). Dentro del concurso, participamos en el primer Challenge, dónde se proponen soluciones a una situación estándar de trabajo de interacción operario-robot.

Dentro del primer Challenge, el proyecto está situado en la primera fase, donde se nos proporciona un entorno de simulación facilitado por la organización para implementar soluciones al desafío lanzado. Esta primera fase consta de dos escenarios distintos. El primero, donde se desarrolla mi proyecto, tiene lugar entre un robot manipulador y su interacción con una pieza de trabajo y el segundo consiste en un desafío de cooperación entre dos brazos robóticos.

En esta fase, se me encomienda una tarea de aplicación industrial: detección de la posición de una pieza de trabajo, reconocimiento del gesto de señalización de un ser humano e identificación del punto perteneciente a la pieza señalado por el humano. Todo ello contando con la información obtenida de una cámara Kienct. La solución del problema planteado será evaluada por la organización EuRoC en busca de la mayor eficiencia, rapidez y precisión posibles.

2.2 ¿Qué es EuRoC?

EuRoC (European Robotic Challenge) consiste en tres desafíos de robótica destinados a estimular nuevas innovaciones para la industria de manufacturación europea.

Figura 2.1 EuRoC.

El consorcio EuRoC se compone de un total de 9 beneficiarios: 5 académicos/investigación, 3 compañías y una PYME.

Los cinco beneficiarios académicos/investigadores (CREATE, CNRS, DLR, ETHZ, IPA) del consorcio han sido elegidos de manera que los tres retos de EuRoC estén cubiertos en un alto nivel científico. Por su parte, los tres beneficiarios industriales (AIR, ASC, KUKA) garantizan la relevancia industrial de los retos, buscando las tecnologías más prometedoras para los posibles usuarios finales impulsada por el mercado de la investigación robótica. Además, un beneficiario con una gran experiencia en estos desafíos (INNO) se ha incluido en el consorcio para agilizar los procesos, garantizar los profesionales de diseño, la ejecución de los desafíos y la comparabilidad entre los tres escenarios de cada uno de ellos.

2.3 Resumen

La industria de manufacturación europea necesita soluciones competitivas para mantener el liderazgo mundial en productos y servicios. La explotación de sinergias a través de expertos en aplicaciones, proveedores de tecnología, integradores de sistemas y proveedores de servicios acelerará el proceso de llevar las tecnologías innovadoras de los laboratorios de investigación a los consumidores industriales finales. Para facilitar este contexto, la iniciativa EuRoC propone poner en marcha tres retos de la industria relevantes:

- Reconfigurable Interactive Manufacturing Cell (RIMC).
- Shop Floor Logistics and Manipulation (SFLM).
- Plant Servicing and Inspection (PSI).

El objetivo es afianzar el avance de la industria europea a través de una serie de experimentos aplicados, mientras se adapta un enfoque innovador que garantice la evaluación del desempeño comparativo. Cada desafío es lanzado a través de una convocatoria abierta estructurándose en 3 etapas.

En cada desafío serán seleccionados 45 concursantes. Los experimentos tendrán lugar en un entorno simulado con el objetivo de que una baja barrera de entrada permita a los nuevos jugadores competir con los equipos robóticos establecidos. En esta primera etapa perteneciente al desafío Reconfigurable Interactive Manufacturing Cell (RIMC) es en el que participamos el equipo del área de Automática y Robótica de CATEC.

Tras esta primera etapa, 15 equipos serán admitidos para la segunda, donde deberán formar un equipo estándar compuesto por expertos en investigación, proveedores de tecnología, integradores de sistemas y los usuarios finales.

Los equipos están obligados al uso de estas plataformas robóticas de referencia facilitadas por dicho consorcio para la segunda etapa. Después de una evaluación intermedia con el concurso público, 6 equipos avanzarán a la última etapa para mostrar el trabajo previamente desarrollado, esta vez en un entorno real con los usuarios finales.

2.4 Objetivos

2.4.1 Objetivo 1: Lanzamiento y ejecución de tres desafíos industrialmente relevantes

Un factor clave para impulsar la innovación en robótica y manufacturación de Europa es el fortalecimiento de la colaboración y el intercambio de ideas entre la industria y la

comunidad investigadora. Hacia este objetivo se ponen en marcha y ejecutan tres desafíos industrialmente relevantes en la robótica europea con aplicabilidad en la fábrica del futuro. Estos desafíos cubren los escenarios de aplicación más prometedores en la industria robótica europea.

2.4.2 Objetivo 2: Apoderamiento de plataformas robóticas e infraestructuras de referencia

El segundo objetivo de este proyecto es capacitar a la robótica de plataformas e infraestructuras de referencia para permitir a los investigadores concentrarse en su investigación "desafiante" en lugar de perder tiempo con problemas de bajo nivel relacionados con diseño y mantenimiento. Asimismo, los entornos se establecerán de tal manera que permitan la comparación de los diferentes enfoques metodológicos para la resolución de casos de uso típicos de fabricación.

Además, se proporciona la funcionalidad de alto nivel en las áreas de percepción, planificación y control que permita a los aspirantes probar y validar su I+D en contextos significativos. Los componentes de software tienen la misma funcionalidad, pero diferente aplicación y rendimiento, por lo que serán intercambiables y, en definitiva, comparables. Con la concesión por parte de numerosos investigadores y desarrolladores de tecnología que dan acceso a estas plataformas y entornos de referencia y ofrecen apoyo intensivo a través de los anfitriones del concurso y los fabricantes de robots, se espera que se levanten unos niveles de rendimiento sin precedentes hacia el final de este proyecto. Además, existe la intención de proporcionar, con una configuración específica, acceso remoto completo y seguro para la programación de los robots en la web, así como para el uso de los recursos de computación distribuida.

Si este concepto demuestra ser viable y eficiente, se podrían abrir nuevas perspectivas y posibilidades para los retos del futuro aprovechando el potencial multiplicativo de las tecnologías de nube.

2.4.3 Objetivo 3: Sostenibilidad y adaptabilidad a los usuarios finales

El tercer objetivo de este proyecto es el desarrollo de soluciones sostenibles: las soluciones de aplicaciones desarrolladas se pondrán a prueba de forma continua. En primer lugar en entornos de simulación y posteriormente, en entornos reales. Las instalaciones se verán aumentadas en niveles de madurez a través del proyecto. Aunque las soluciones son impulsadas para necesidades del usuario final, éstas serán suficientemente generales para que puedan ser aplicadas a otros usuarios finales, tareas y situaciones, y permitir una rápida comercialización. La robustez tendrá que aumentar significativamente para conseguir que los robots trabajen de forma continuada. Siguiendo esta metodología, se definirán criterios de referencia claros en los procesos de evaluación. Los problemas que no puedan resolverse en el plazo de este proyecto se integrarán en el proceso de actualización de la robótica y los planes de trabajos futuros.

2.5 Challenge 1

El escenario de aplicación es abierto para fomentar la innovación y creatividad en el desarrollo de aplicaciones para al fabricación novedosas que involucren la colaboración entre humanos y robots. Sin embargo, la viabilidad y rentabilidad de las soluciones deberán ser demostradas. Además, la seguridad en el desarrollo de la solución para el trabajo tendrá

que ser garantizada a través de las medidas apropiadas. Las aplicaciones pueden implicar un brazo, dos o múltiples configuraciones de brazo, donde cada uno de ellos es reconfigurable. Además, la distribución de las tareas asignadas al empleado humano y a los brazos robóticos también puede reconfigurarse dinámicamente en línea.

Encontramos problemas tales como:

- Técnicas de percepción y cognición adaptativas en presencia de diseños complejos de células de trabajo y entornos dinámicos con cambios de iluminación y tolerancias.
- Técnicas para montajes robustos en presencia de piezas con tolerancias flexibles.
- Interacción humano-robot segura y productiva en presencia de situaciones ambiguas y lugares de trabajo reducidos.
- Métodos de control de sistemas robóticos de cooperación multi-función en un entorno industrial relevante.

En la primera fase de este challenge tiene lugar mi proyecto. Consta de 4 tareas interactivas robot-operario robot-robot, que se realizarán en dos escenarios de simulación distintos. la tarea 1 y 2 tendrán lugar en el primer escenario mientras que la 3 y la 4, en el segundo.

La tarea 1, centro de este proyecto, consta de tres diferentes subtareas en grado de dificultad ascendente, aunque en cuanto a nuestra solución, como veremos más adelante, alteramos el orden de la subtarea 1 con la 2. En la primera nuestro objetivo es reconocer si se produce una señalización a alguno de los distintos agujeros de la pieza. En la segunda, sólo tendremos que dar la posición de la pieza de trabajo. Por último, en la tercera, nuestra misión consistirá en indicar qué agujero está siendo señalado por el humano.

La tarea 2 consiste en la programación de nuestro robot para que realice una secuencia de remachado.

Las tareas 3 y 4 consisten en control de coordinación de dos brazos manipuladores robóticos, en la tarea 3 se trata de introducir un objeto que porta un brazo dentro de otro objeto que porta otro brazo La tarea 4 tiene como objetivo mover un objeto de forma coordinada entre ambos manipuladores esquivando un obstáculo del escenario.

3 Estado del arte

3.1 Robótica

El nacimiento de la robótica surgió de la búsqueda de una continuación del trabajo del hombre en la producción de bienes y en la explotación de recursos naturales basándose en conocimientos científicos. En sus inicios sólo desarrollaban una ayuda al trabajo físico del hombre pero en la actualidad son capaces de suplirlo casi completamente en actividades intelectuales después de un procesado de información. Muchos de estos robots perseguían imitar los movimientos del ser humano aunque ya en 1915, Leonardo Torres Quevedo dijo "Lo que busco son aparatos que obtengan los mismos resultados que una persona, sin que por ello tengan que reproducir sus mismos gestos". [4]

Esta objetivo se alcanza mediante el control automático de procesos que, después de haber estudiado el entorno con diferentes métodos de sensorización y una realimentación, sea capaz de generar respuestas sin la intervención del hombre que persigan una consigna o un movimiento previamente descritos por el hombre.

Existen muchas funciones que desarrolla el robot en el medio como:

- Terrestres (vehículos, robots con patas, manipuladores industriales).
- Aéreos (dirigibles).
- Acuáticos (nadadores, submarinos)
- Híbridos (trepadores)

Y existen más en función del control de movimiento como:

- Autónomos.
- Teleoperados.
- Robots fijos: automatización de procesos industriales, asistencia médica, etc.
- Robots móviles: exploración, transporte.
- Reproducir ciertas capacidades de los organismos vivos.
- Otros: entretenimiento.

Allá en 1971, el Stanford Arm, un pequeño brazo de robot de accionamiento eléctrico, se desarrolló en Stanford University. Desde entonces, son muchos los diferentes brazos robóticos empleados para desarrollar una grandísima multitud de tareas diferentes. En todas ellas, hasta hace pocos años, los brazos robóticos han trabajado sin inteligencia, de manera secuencial y sin conocer su entorno, símplemente siguiendo unos puntos de forma repetitiva. Acualemente, se está dotando tanto a los brazos robóticos como a los robots en general de sensores que les permitan conocer el mundo que los rodea y poder actuar en consecuencia, es decir, realimentar con información externa. Gracias a ello se ha amentado tanto la eficiencia y autonomía del robot, como la seguridad con la que trabaja.

Todo esto ha sido posible gracias a sensores tan variados como pueden ser las cámaras, objeto de este trabajo, o sensores de temperatura o esfuerzos. Dentro de los diferentes sensores por visión, podría limitarse a un cálculo de una cierta distancia de seguridad o la distancia a un objeto que se quiera coger. Pero la ciencia ha ido más lejos aún, permitiendo con estos sensores dar nueva información al robot de cuáles son sus sus tareas y los objetivos de las mismas gracias al reconocimiento gestual.

3.2 Reconocimiento gestual

La interacción entre el ser humano y los dispositivos electrónicos es un campo que está en continua evolución buscando la forma más intuitiva de comunicación. Son muchas las empresas multinacionales que están invirtiendo grandes cantidades de dinero en obtener nuevos interfaces y canales de comunicación con la máquina no sólo en el campo de los videojuegos, puntero en esta área (Kinect de Microsoft, PlayStation-Eye de Sony o Wii de Nintendo entre otros), sino también en el control de entornos multimedia. Los sistemas más avanzados utilizan la visión, pero también se están desarrollando otros muchos proyectos que usan sensores muy diferentes. Para ello, es necesario el uso de dos tecnologías independientes pero aunque van de la mano: hardware que sea capaz de adquirir datos del entorno de una forma precisa y software que comprenda, trate e interprete la información adquirida por los sensores. Los diferentes sensores encargados de la obtención de la información espacial se pueden clasificar en:

- Tecnología basada en el contacto: acelerómetros, pantallas táctiles, guantes instrumentalizados, joysticks… Cuya tecnología puede estar basada en la mecánica, en sistemas inerciales, magnéticos o ultrasónicos.
- Tecnología basada en la visión: cámaras tradicionales monoculares, cámaras de infrarrojos, cámaras estéreo, Kinects o marcadores corporales (activas si emiten luz o pasivas sólo la reflejan). Estas últimas para ayudar a la determinación de la posición de los puntos donde se encuentran mediante su singularización.

Figura 3.1 Guante (izquierda) y traje corporal (derecha) sensorizados.

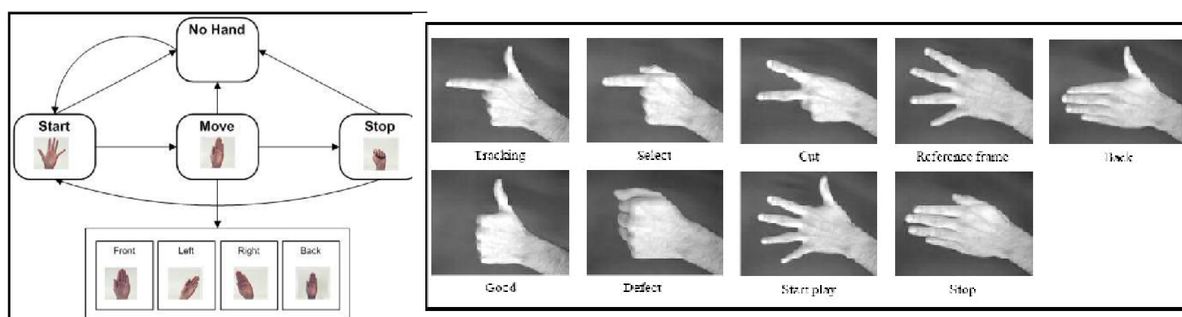


Figura 3.2 Posible diagrama para la comunicación gestual(izquierda) y posible colección de gestos memorizados(derecha).

Al hablar de reconocimiento gestual, lo normal es pensar en gestos manuales, pero hoy en día los dispositivos nos permiten la transmisión de información casi con cualquier parte del cuerpo como el caso de los videojuegos comentados anteriormente, o de sólo una parte del mismo, bien pudiera ser la cabeza, los ojos, las piernas, los brazos, las manos o incluso sólo los dedos. Seguns estudios [2], existen setecientos mil mensajes no verbales que el ser humano es capaz de hacer, de los cuales quinientos mil son gestos faciales y quinientos manuales.

El reto de la detección de gestos manuales se está afrontando mediante la sustracción del fondo, es decir, una cámara cenital y un fondo homogéneo que facilite la separación de zonas de la imagen. Una característica muy importante del sensor utilizado es que pueda medir la profundidad, opción que mejora en gran medida la detección del gesto. Una alternativa que se está utilizando mucho actualmente pero que aún es reciente es el Time-of-flight (tiempo de vuelo) que devuelve profundidad en tiempo real sin consumir un elevado coste computacional.

Hasta mediados de los 80, la tecnología perseguía una comprensión total del entorno de una forma estática y pasiva que tomase todo el tiempo necesario, para posteriormente asimilar y procesar esa información sin tener en cuenta qué es exactamente lo que se buscaba. Desde entonces, la lógica se invirtió para poner la visión al servicio del objetivo, es decir, ir a buscar las imágenes de una forma inteligente. Esto significa trabajar con resultados previos para modificar los parámetros propios tanto de adquisición como de procesamiento. “La percepción debe estar guiada por la acción”. Es buscar aquellas propiedades de la imagen que nos sean útiles para la tarea y prescindir de las que no. En ocasiones, es necesario un tratamiento posterior a la captación de las nubes pero anterior al tratamiento de las mismas: la reconstrucción de superficies que tienen imperfecciones u oclusiones parciales. [8]

En la actualidad, la forma de comprensión de gestos ha estado supeditada en parte a la observación del movimiento. Esto conlleva el seguimiento de uno o más objetos y la posterior caracterización de la trayectoria. Recientemente se han hecho estudios sobre la caracterización del movimiento no rígido elástico. Otra forma de comprensión de gestos es la que atañe este proyecto: la detección basada en modelos. Este área se alimenta de la detección y extracción de puntos, líneas o regiones peculiares que se adapten al modelo anteriormente guardado mediante un proceso de correspondencia. Esa búsqueda de características singulares suele presentar un alto coste computacional, problema que suelen compartir casi todas las posibles soluciones a este problema.

Otro punto a tener en cuenta suele ser que la interpretación de un gesto está directamente supeditada a la cultura y la zona geográfica del comunicador, por lo que un mismo gesto

podría ser interpretado de una forma u otra según la localización. Este problema tiene simple solución, ya que para la comunicación con la interfaz es necesario el aprendizaje de un “lenguaje corporal” propio con el que comunicarse. Existen diferentes gestos a la hora de comunicarnos con al máquina, entre los cuales podemos destacar:

- Emblemas: se traducen directamente en una comunicación verbal. Dependen de la cultura. Ejemplos: “no” o “adiós”.
- Ilustraciones: representan o enfatizan lo que se está diciendo. Ejemplos: señalización de un punto o representación de una figura.
- Reguladores: controlan la interacción. Ejemplos: previamente definidos para la comunicación.
- Adaptadores: sirven para relajar el cuerpo y no deben ser tenidos en cuenta ya que suelen ser inconscientes. Ejemplos: meneo de la cabeza o movimiento rápido de una pierna.

3.3 Simulación Robótica

La simulación juega un papel crucial previo en los proyectos robóticos gracias a que nos permite realizar las operaciones de validación y verificación de aplicaciones robóticas antes de la construcción física del robot, lo que conlleva un importante ahorro tanto de tiempo como de dinero. Su empleo no sólo se basa en un carácter, por llamarlo de alguna manera, preventivo del producto. La simulación virtual también tiene una importante aplicación a la hora de entrenar el manejo y comprender las características del dispositivo antes de emplear el robot real.

Simuladores basados en comportamientos robóticos nos permiten crear mundos simplificados con objetos rígidos y programar robots que interactúen con ellos. En ocasiones, un entorno con condiciones extremas u operaciones en un área remota no nos permiten verificar todos los procedimientos y problemas físico-mecánicos ni por ello testear de una forma sólo experimental. Por lo tanto, pueden presentarse ocasiones en las que haya que tomar decisiones con el único análisis de los resultados de una simulación. Una de las aplicaciones más populares es el modelado 3D y su representación que requieren de buenos motores de leyes físicas y buenos gráficos para una emulación aceptable del robot y el mundo que lo rodea. Esto implica que cada robot tendrá unas propiedades gráficas y unas físicas. Las técnicas de simulación e implementación robótica necesariamente pasan por una descripción analítica del sistema físico-mecánico. Sin embargo, a veces puede no contarse con todos los parámetros reales para simular un entorno cien por cien realista. Se emplean entonces técnicas de manejo de datos con la probabilidad y estadística. Es necesario también el modelado numérico con su respectivo software y toolboxes. Son empleados simuladores multidominio e híbridos con métodos que soporten una simulación rápida en tiempo real.

En la actualidad, la simulación robótica nos proporciona una grandísima variedad de elementos: diferentes familias de robots (UGV para tierra, UAV para aire, AUV para agua, brazos robóticos, manos robóticas, humanoides, avatares…), actuadores (Cadenas cinemáticas genéricas, actuadores de fuerza controlada…), subactuadores, sensores (Odometría,

IMU, GPS, cámaras, lásers, emisores y receptores...), mecanismos, manipuladores, herramientas robóticas...). Llegando incluso a existir simuladores empleados en tareas en el espacio como satélites o aeronaves.

Científicos e ingenieros han venido desarrollando conjuntamente una gran variedad de técnicas de modelado y simulación creando diferentes softwares, de los cuales destacaré a continuación los más importantes. Morse es un simulador de un robot de código abierto con soporte completo a ROS . Usa OpenGL como su motor 3D y tienen un render realista, que hace que la simulación sea más sencilla de entender. El sistema funciona con comandos en línea, pero también puede usarse Python para controlar los robots en el simulador. Un componente sorprendente de Morse es su habilidad para modelar la interacción humano/-robot. Otro simulador similar es Gazebo, también para ROS. Como es el usado en este trabajo, lo desarrollaré más adelante. El siguiente ejemplo es Webots, el cual utiliza ODE (Open Dynamics Engine) para la detección de colisiones, proporcionando también una simulación precisa de velocidad, inercia y fricción. Una de las características que han hecho famoso ARS es que funciona exclusivamente con Python y su facilidad para generar la documentación. Otro simulador muy famoso es V-Rep no sólo porque los controladores pueden ser escritos en casi todos los lenguajes de programación existentes, sino por la versatilidad que le proporciona su distribuida arquitectura de control: cada objeto puede ser controlado independientemente por un script interno, un plugin, un nodo de ROS, un cliente remoto API o el cliente. Es empleado para desarrollo rápido de algoritmo y simulaciones cadenas de montaje.

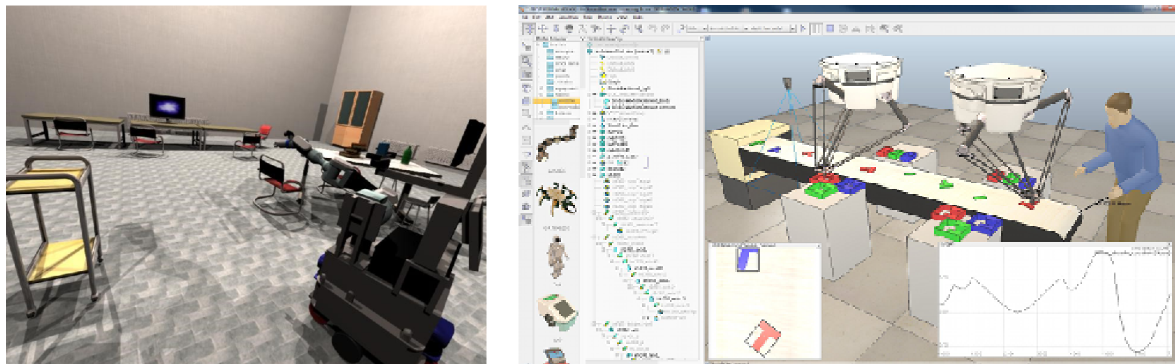


Figura 3.3 Morse 1.0 (izquierda) y V-Rep (derecha).

4 Desarrollos teóricos para el tratamiento de nubes de puntos

4.1 Representación de la posición y la orientación

Sabemos que la posición de un punto en el espacio euclídeo tridimensional viene determinada por tres cantidades, que llamamos sus coordenadas, y decimos que están expresadas en algún sistema de referencia, formado por tres ejes, usualmente rectilíneos. [4]

En lo sucesivo usaremos exclusivamente sistemas de referencia rectilíneos, ortogonales (es decir, con sus tres ejes perpendiculares dos a dos), normalizados (es decir, las longitudes de los vectores básicos de cada eje son iguales) y dextrógiros (el tercer eje es producto vectorial de los otros dos).

Usaremos, pues, simplemente el término "sistema" para referirnos a sistemas ortonormales.

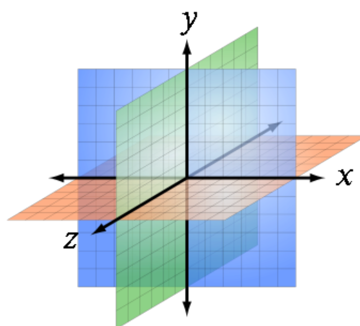


Figura 4.1 Sistema Ortonormal.

Las coordenadas de un punto, denotadas por (x, y, z), son las proyecciones de dicho punto perpendicularmente a cada eje, o, equivalentemente, las componentes del vector que lo une al origen de coordenadas. En lugar de usar estas, nos será más conveniente el uso de las llamadas coordenadas homogéneas, en la forma:

$$\begin{pmatrix} x' \\ y' \\ z' \\ w \end{pmatrix} entonces \begin{pmatrix} x' = xw \\ y' = yw \\ z' = zw \end{pmatrix}$$

siendo w una cantidad arbitraria, que se suele tomar como 1. Si, como resultado de algún cálculo, w fuese distinto de 1, las coordenadas usuales se reconstruyen simplemente dividiendo las tres primeras coordenadas homogéneas entre esta cuarta.

La traslación de un punto x, y, z por un vector v es un punto x', y', z' tal que:

$$\begin{pmatrix} x' \\ y' \\ z' \end{pmatrix} = \begin{pmatrix} x \\ y \\ z \end{pmatrix} + \begin{pmatrix} v_x \\ v_y \\ v_z \end{pmatrix}$$

Pero también como el producto de una matriz por un vector homogéneo, en la forma:

$$\begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & v_x \\ 0 & 1 & 0 & v_y \\ 0 & 0 & 1 & v_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

Esto tiene la ventaja de que, si: $\overrightarrow{x}' = H\overrightarrow{x}$ entonces $\overrightarrow{x} = H^{-1}\overrightarrow{x}'$

Donde se puede calcular la inversa, que resulta ser:

$$H^{-1} = \begin{pmatrix} 1 & 0 & 0 & v_x \\ 0 & 1 & 0 & v_y \\ 0 & 0 & 1 & v_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

lo cual es consistente con el hecho de que x está trasladado por un vector -v respecto a x'. Respecto a la rotación alrededor de un eje, en el caso bidimensional, se esta rotando con respecto a un eje z perpendicular al plano de la figura. Rotación

Llamando i, j a los vectores básicos del sistema original, e i' y j' a los del sistema girado, se tiene que

$$\overrightarrow{x} = x\overrightarrow{i} + y\overrightarrow{j} = x'\overrightarrow{i}' + y'\overrightarrow{j}'\overrightarrow{i}' = \cos(\theta)\overrightarrow{i} + \sin(\theta)\overrightarrow{j}\overrightarrow{j}' = -\sin(\theta)\overrightarrow{i} + \cos(\theta)\overrightarrow{j}$$

Es decir, que

$$\overrightarrow{x} = x'(\cos(\theta)\overrightarrow{i} + \sin(\theta)\overrightarrow{j}) + y'(-\sin(\theta)\overrightarrow{i} + \cos(\theta)\overrightarrow{j})$$

Igualando componente a componente, escribimos la matriz como

$$\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{pmatrix} \begin{pmatrix} x' \\ y' \end{pmatrix}$$

Si generalizamos a tres dimensiones, como la coordenada z no varía y la cuarta coordenada homogénea sigue siendo 1, tenemos

$$\begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} \cos(\theta) & -\sin(\theta) & 0 & 0 \\ \sin(\theta) & \cos(\theta) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix}$$

Para hallar la transformación inversa basta ver que desde el punto de vista de R', R esta rotado un ángulo $-\theta$, luego podemos afirmar que

$$\begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} \cos(-\theta) & -\sin(-\theta) & 0 & 0 \\ \sin(-\theta) & \cos(-\theta) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} = \begin{pmatrix} \cos(\theta) & \sin(\theta) & 0 & 0 \\ -\sin(\theta) & \cos(\theta) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix}$$

Esta operación es más simple que invertir la matriz, aunque por supuesto, equivalente. En general, si hubiéramos rotado alrededor de otro de los ejes básicos, se puede ver que

$$Rot(x,\theta) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\theta) & -\sin(\theta) & 0 \\ 0 & \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$Rot(y,\theta) = \begin{pmatrix} \cos(\theta) & 0 & \sin(\theta) & 0 \\ 0 & 1 & 0 & 0 \\ -\sin(\theta) & 0 & \cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$Rot(z,\theta) = \begin{pmatrix} \cos(\theta) & -\sin(\theta) & 0 & 0 \\ \sin(\theta) & \cos(\theta) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Cabe destacar el cambio de signo en la rotación alrededor del eje y, debido a que, si el eje alrededor del cual rotamos nos apunta, los otros dos forman un ángulo de 90º en el caso de x y z, pero de -90º en el caso de y. Se pueden aplicar a un punto tantas transformaciones sucesivas (rotaciones y traslaciones) como se quiera. La operación resultante vendría dada por una matriz que sería producto de las matrices de cada operación, aplicadas en el orden correcto, dado que el producto de matrices no es conmutativo. Se pone más a la derecha la primera transformación que se aplique, siendo expresado como

$$Y = T_2 R_3 T_1 R_2 R_1 X$$

Significa que se aplica al punto X la rotación 1, seguida de la rotación 2, seguida de la traslación 1, luego la rotación 3 y por último la traslación 2. Veamos ahora cual sería la matriz de rotación respecto a un eje cualquiera. Sea un eje que pasa por el origen definido por un vector unitario alrededor del cual giraremos un ángulo θ.

Esta rotación se podrá descomponer en tres rotaciones sobre los ejes básicos, lo que equivaldrá a:

- Rotar un ángulo α alrededor de x, con lo que P pasara a la posición P'.
- Rotar un ángulo $-\beta$ alrededor de y, con lo que P' pasara a la posición P''.
- Rotar un ángulo β alrededor de z, que es la rotación que se pide.
- Rotar un ángulo β alrededor de y, deshaciendo la segunda rotación
- Rotar un ángulo $-\alpha$ alrededor de x, deshaciendo la primera rotación.

Entonces tenemos que

$$R_{\vec{r},\theta} = R_{x,-\alpha} R_{y,\beta} R_{z,\theta} R_{y,-\beta} R_{x,\alpha}$$

O también

$$R_{\vec{r},\theta} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & c\alpha & s\alpha & 0 \\ 0 & -s\alpha & c\alpha & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} c\alpha & 0 & s\alpha & 0 \\ 0 & 1 & 0 & 0 \\ -s\alpha & 0 & c\alpha & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} c\alpha & -s\alpha & 0 & 0 \\ s\alpha & c\alpha & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} c\alpha & 0 & -s\alpha & 0 \\ 0 & 1 & 0 & 0 \\ s\alpha & 0 & c\alpha & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & c\alpha & -s\alpha & 0 \\ 0 & s\alpha & c\alpha & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Debemos destacar que cualquier secuencia consecutiva de transformaciones se puede especificar de dos formas:

- Realizando la rotación que lleva un sistema al otro, alrededor de los ejes iniciales.
- Realizando la rotación que lleva un sistema al otro alrededor de uno de los ejes girados, es decir, los que resultaron de la última transformación.

En el primer caso, la matriz que describe esta transformación deberá pre-multiplicarse por la que describía las transformaciones efectuadas hasta el momento, obteniendo la transformación total.

En el segundo caso la matriz que describe esta transformación deberá postmultiplicarse por la que describía las transformaciones efectuadas hasta el momento, obteniendo la transformación total.

4.2 Algoritmo de correspondencia de nubes de puntos

La información 3D se puede representar y obtener de diferentes formas. Puede ser de una forma organizada como un mapa de vóxels (píxel en 3D) o una imagen de rango (cuadrícula en 2D con información 3D) o en forma desorganizada como pudiera ser una nube de puntos sin relación entre ellos. También puede contener información puramente geométrica y espacial o también los tres canales de color RGB. Además, no tiene por qué representar un objeto entero, sino que puede contener la información vista desde un solo punto de vista (2.5D). Como es lógico, lo óptimo para el "matching" de imágenes sería 3D vs. 3D, pero eso no siempre es posible ya que las imágenes que obtenemos desde un sensor como el que usamos en este proyecto tienen muchas oclusiones. [5]

El paradigma a seguir para encontrar similitudes entre nubes de puntos precisa de un paso inicial de búsqueda de puntos característicos clave o "keypoints" de un modelo que servirán para ser correspondidos con los puntos de la escena empleando descriptores. Estos puntos característicos deben ser distintivos, es decir, actos para su descripción efectiva (definibles localmente) y también deben ser repetibles después de variaciones del punto de vista o ruidos (definibles globalmente). En cuanto a la escala, es necesario tenerla en cuenta ya que un punto es saliente en su entorno o no.

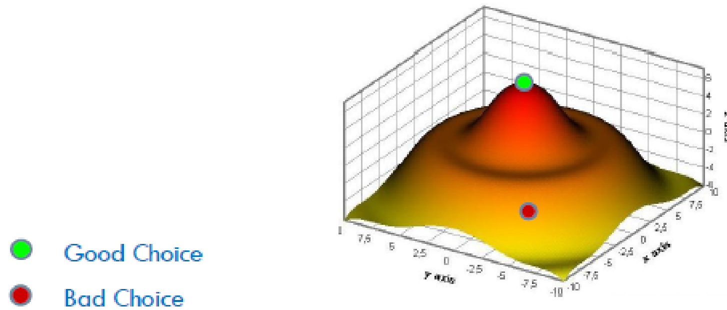


Figura 4.2 Importancia en el elección de puntos.

Matriz de covarianza:

$$M(p_i) = \frac{1}{\sum_{k}^{j=1} p_i} \sum_{k}^{j=1} p_i (p_j - p_i)(p_j - p_i)^T$$

Sus autovalores, en orden descendiente serán λ_1,λ_2,λ_3 siendo la saliencia $(\rho(p))$ el tercer valor. El siguiente paso será el descarte de los autovalores que tengan la misma propagación en las direcciones principales, ya que un marco de referencia local no podría ser definido en él.

$$\frac{\lambda_2(p)}{\lambda_1(p)} < Th_{12} \wedge \frac{\lambda_3(p)}{\lambda_2(p)} < Th_{23}$$

Cada uno de los descriptores puede contener información de diferentes regiones. Los más simples, los puntuales son eficaces y sencillos pero no son bustos ante el ruido y ni quizá suficientemente descriptivos. Los locales pueden ser útiles para atajar las oclusiones y el desorden y son invariantes ante rotaciones o traslaciones. Por último, los globales son perfectos para recuperación y categorización ya que son más robustos y ofrecen información completa de superficies.

Un descriptor muy usado es el histograma con información geométrica de los ángulos que forma entre sí la normal o sus derivadas en las direcciones principales.

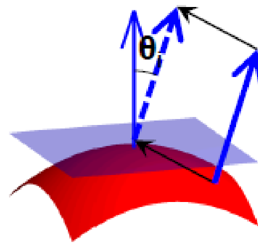


Figura 4.3 Ángulo formado por las normales de dos puntos contiguos.

El PFC (Point Features Histogram) computa tres valores para cada par en una misma vecindad basados en un marco centrado en el punto del que estemos buscando la información. U v w Estos tres valores serán los ángulos representados a continuación:

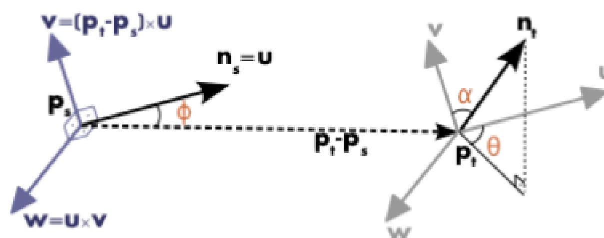


Figura 4.4 Representación de los ángulos descriptivos de una pareja de puntos.

A veces se emplea un PFC simplificado para un punto pero a su vez complementado con la adición de los PFCs de sus puntos vecinos ponderados.

En general la taxonomía de los descriptores está compuesta por los histogramas recién vistos, por transformaciones si el dominio es compacto e invariante, por proyecciones del 3D en 2D y por descriptores basados en gráficas con información extraída de las superficies (como podría ser el esqueleto de una superficie generada por un animal). Por su parte, para el estudio de la forma local de la nube se estudian los ángulos (triangulando desde el punto de estudio actual), distancias (entre puntos vecinos o con el punto de estudio actual), área de triángulo (3 puntos) o volumen de tetraedro (4 puntos). Este estudio se realiza para su posterior comparación de disimilitud. Estas son algunas de las diferentes formas de estudiar la "firma" o "huella" de un punto.

Hay diferentes formas decidir qué vecinos se utilizan. Se pueden elegir M puntos en la vecindad e ir aumentando un radio de búsqueda en el espacio que rodea al punto de estudio hasta que se complete la lista de vecinos. También puede elegirse un radio constante y tomar como vecinos todos los que queden dentro de esa hiperesfera. Una aproximación popular es el uso de kd-árboles que sectoricen el espacio, tomando como vecinos todos los puntos del mismo sector. El algoritmo "Flann", empleado en el código que nos han proporcionado usado en este trabajo, elabora un estudio previo de la nube de puntos con la que se trabaja y elige qué algoritmo de búsqueda de vecinos es el óptimo en cada caso.

En cuanto a la generación de grupos correspondientes, se elabora una lista de correspondencias punto-punto tal que $C = c_1, c_2, \ldots, c_n$ siendo $c_i = p_{i,escena}, p_{i,modelo}$. El siguiente paso es agrupar esas correspondencias de acuerdo a que mantengan, entre puntos de la misma escena o del mismo modelo, las transformaciones relativas en los seis grados de

libertad. Es decir, que la topología de las nubes de puntos se mantenga inalterada (consistencia geométrica). Estas transformaciones locales se harán tomando un único marco de referencia local centrado en uno de los puntos dentro del grupo llamado "centroide". Los puntos que no se encuentren dentro de ningún grupo se conocerán como "outliers". Como es lógico, el grupo finalmente escogido será el que más correspondencias seguras contemple.

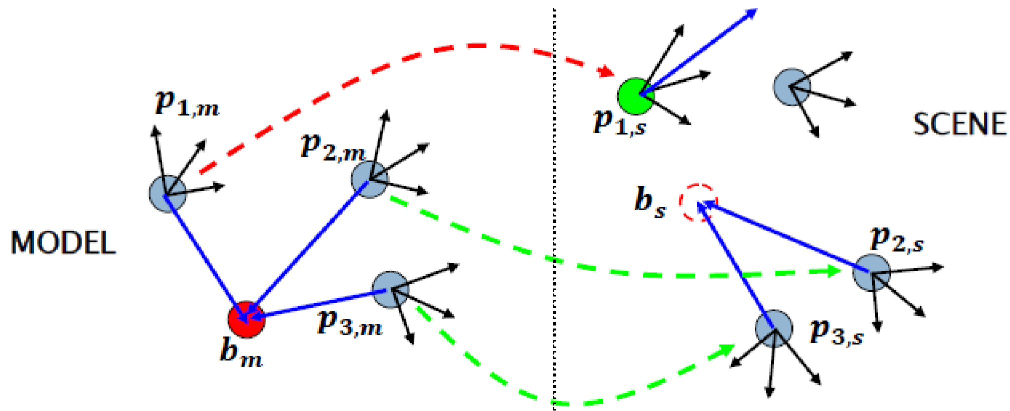


Figura 4.5 Ejemplo de elección de puntos dentro de un mismo grupo.

Un último algoritmo propuesto es ICP. Consiste en un pequeño movimiento en los seis grados de libertad del modelo dentro de la escena siguiendo una distribución estadística que permita hacer ligeras variaciones de la posición y orientación comprobando si se consigue alguna mejora en el error. Si esto fuera cierto, se volvería a repetir esta leve corrección local hasta hallar un error mínimo

5 ROS y Herramientas

5.1 ROS

ROS (Robot Operating System) es un framework, basado en Ubuntu, para el desarrollo de software para robots que provee la funcionalidad de un sistema operativo en un clúster heterogéneo. ROS se desarrolló originalmente en 2007 bajo el nombre de switchyard por el Laboratorio de Inteligencia Artificial de Stanford para dar soporte al proyecto del Robot con Inteligencia Artificial de Stanford (STAIR). Desde 2008, el desarrollo continúa primordialmente en Willow Garage, un instituto de investigación robótico con más de veinte instituciones colaborando en un modelo de desarrollo federado [3].

ROS provee los servicios estándar de un sistema operativo tales como abstracción del hardware, control de dispositivos de bajo nivel, implementación de funcionalidad de uso común, paso de mensajes entre procesos y mantenimiento de paquetes. Está basado en una arquitectura de grafos donde el procesamiento toma lugar en los nodos que pueden recibir, mandar y multiplexar mensajes de sensores, control, estados, planificaciones y actuadores, entre otros. La librería está orientada para un sistema UNIX (Ubuntu (Linux)) aunque también se está adaptando a otros sistemas operativos como Fedora, Mac OS X, Arch, Gentoo, OpenSUSE, Slackware,Debian o Microsoft Windows, considerados como 'experimentales'.

ROS tiene dos partes básicas: la parte del sistema operativo, ros, como se ha descrito anteriormente y ros-pkg, una suite de paquetes aportados por la contribución de usuarios (organizados en conjuntos llamados pilas o en inglés stacks) que implementan la funcionalidades tales como localización y mapeo simultáneo, planificación, percepción, simulación, etc.

ROS es software libre bajo términos de licencia BSD. Esta licencia permite libertad para uso comercial e investigador. Las contribuciones de los paquetes en ros-pkg están bajo una gran variedad de licencias diferentes.

Figura 5.1 Diferentes licencias de ROS.

Las áreas que incluye ROS son:

- Un nodo principal de coordinación.
- Publicación o subscripción de flujos de datos: imágenes, estéreo, láser, control, actuador, contacto, etc.
- Multiplexión de la información.
- Creación y destrucción de nodos.
- Los nodos están perfectamente distribuidos, permitiendo procesamiento distribuido en múltiples núcleos, multiprocesamiento, GPUs y clústeres.
- Login.
- Parámetros de servidor.
- Testeo de sistemas.

Las áreas que incluirán las aplicaciones de los paquetes de ROS son:

- Percepción
- Identificación de Objetos
- Segmentación y reconocimiento
- Reconocimiento facial
- Reconocimiento de gestos
- Seguimiento de objetos
- Egomoción
- Comprensión de movimiento
- Estructura de movimientos (SFM)
- Visión estéreo: percepción de profundidad mediante el uso de dos cámaras
- Movimientos
- Robots móviles
- Control
- Planificación
- Agarre de objetos

5.1.1 Diseño modular y distribuido

Ros fue diseñado para ser lo más distributivo y modular posible, de modo que los usuarios pueden utilizar ROS tanto como deseen. Su modularidad le permite seleccionar y elegir qué partes son útiles y qué partes prefiere implementar uno mismo.

La naturaliza distributiva de ROS también fomenta una gran comunidad de paquetes contribuidas por usuarios que añaden un gran valor a la parte superior del núcleo del sistema ROS. En el último recuento se contaron más de 3000 paquetes en el ecosistema

ROS, y eso que solo son los paquetes que la gente ha hecho público. Estos paquetes varían en la fidelidad, cubriendo desde pruebas de conceptos de implementación de nuevos algoritmos hasta drivers de alta capacidad y calidad industrial. La comunidad de usuarios de ROS construye sobre la parte superior de una infraestructura común para proporcionar un punto de integración que ofrece acceso a controladores de hardware, robots genéricos, herramientas de desarrollo, bibliotecas externas útiles, y mucho más.

5.1.2 Historia

ROS es un largo proyecto con muchos antepasados y colaboradores. La necesidad de un marco de colaboración abierto fue requerido por muchas personas en la comunidad de investigación robótica, y muchos proyectos se han creado para alcanzar este objetivo.

Varios esfuerzos en la Universidad de Stanford a mitad de los años 2000 envolviendo integrativamente tanto STanford Al Robot (STAIR) como el programa Personal Robots (PR) crearon prototipos internos de sistemas de software flexibles y dinámicos destinado al uso robótico. En 2007, Willow Garage, una incubadora cercana de visionarios robóticos, aportó importantes recursos para ampliar estos conceptos mucho más allá y crear implementaciones bien probadas. Este esfuerzo se vio impulsado por un sinfín de investigadores que contribuyeron con su tiempo y experiencia para las ideas principales de ROS y sus paquetes de software fundamentales. En todo momento, el software fue desarrollado en abierto usando una licencia de código abierto (BSD open-source), y poco a poco se ha convertido en una plataforma ampliamente utilizada en la comunidad de investigación robótica.

Desde sus inicios, ROS se desarrolló en múltiples instituciones y para múltiples robots, incluyendo muchas de las instituciones que recibieron los robots PR2 de Willow Garage. A pesar de que habría sido mucho más fácil para todos los contribuyentes poner su código en los mismos servidores, en los últimos años, el modelo “federado” se ha convertido en una de las grandes fortalezas del ecosistema ROS. Cualquier grupo puede comenzar su propio repositorio de código ROS en sus propios servidores, y mantener la propiedad y control por completo, sin necesidad de permiso de nadie. Si deciden poner su repositorio a disposición del público, pueden recibir el reconocimiento y crédito que se merecen sus logros, y beneficiar se de la información técnica específica y mejoras como todos los proyectos de software de código abierto.

5.1.3 Comunidad activa y colaborativa

En los últimos años, ROS ha crecido para contar con una gran comunidad de usuarios en todo el mundo. Históricamente, la mayoría de los usuarios se encontraban en los laboratorios de investigación, para cada vez más estamos viendo una adopción en el sector comercial, en particular en la industria y servicios robóticos.

La comunidad ROS es muy activa. Según las últimas mediciones, la comunidad ROS cuenta con más de 1500 en la lista de ROS distributiva, más de 3300 usuarios en la wiki de documentación colaborativa y unos 5700 usuarios en foro de la página web. La wiki cuenta con más de 22000 páginas y más de 30 ediciones diarias. En el foro tienen lugar unas 13000 preguntas hechas hasta la fecha, con una tasa de respuesta del 70 %.

ROS no solo ofrece por sí mismo un gran valor a la mayoría de los proyectos, sino que también representa una oportunidad para establecer contactos y colaborar con los expertos

en robótica mundiales que forman parte de la comunidad de ROS. Una de las filosofías básicas de ROS es compartir el desarrollo de componentes comunes.

5.1.4 Conceptos básicos

Los conceptos fundamentales de la aplicación de ROS son nodos, mensajes, topics y servicios.

Los nodos son procesos ejecutables. ROS es diseñado para ser modular en una escala de grano fino: Un sistema típicamente se compone de muchos nodos. En este contexto, el término “nodo” es intercambiable por “módulo”. El uso del término “nodo” surge de visualizaciones de ROS basados en sistemas tiempo de ejecución: cuando muchos nodos se están ejecutando, se realizan las comunicaciones “peer-to-peer”, es decir, el enlace entre los distintos nodos son estas comunicaciones.

Los nodos se comunican entre sí mediante paso de mensajes. Un mensaje es una estructura de datos que pueden ser distintos tipos (enteros, punto flotantes, booleanos, etc) apoyados por tipos primitivos y constantes. Los mensajes pueden estar compuestos de otros mensajes, y matrices de otros mensajes, anidados de forma profunda.

En nodo envía un mensaje mediante su publicación en un “topic”. Un topic puede interpretarse como un buzón de mensajes dónde llega siempre un mismo tipo o estructura de dato, de esa forma, si un nodo está interesado en un determinado tipo de dato, se suscribirá en el topic correspondiente. Puede haber varios publicados

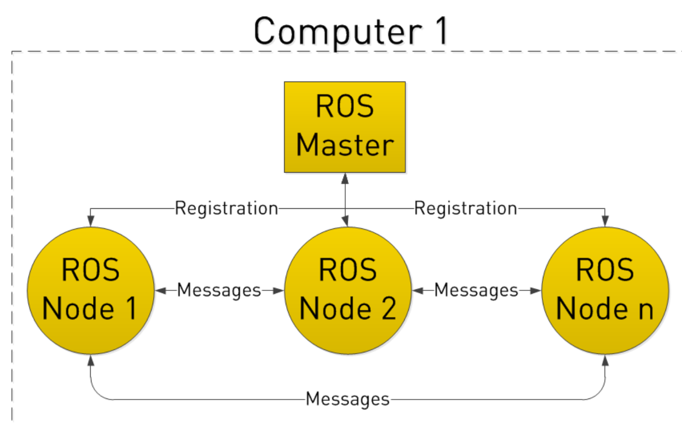


Figura 5.2 Estructura de ROS.

Aunque el modelo de publicación-suscripción basado en topics es flexible, no es apropiado para transacciones síncronas, que pueden permitir el diseño de algunos nodos. Para esto, ROS nos facilita los “servicios”, que se define con un nombre de cadena de caracteres y dos mensajes proporcionados: Uno de solicitud y otro de respuesta. Esto es análogo a los servicios web, que son definidos por los URI y tienen tipos de documentos de solicitud y respuesta bien definidos. A diferencia de los topics, solo un nodo puede requerir un servicio, mientras que será respondido por otro.

Finalmente también hay que destacar la capacidad de flexibilidad a la hora de la programación, ya que permite usar varios lenguajes de programación como C++, Python, Lisp u Octave, permitiendo además, que un sistema esté compuesto por nodos escritos en distinto lenguajes, aumentado su flexibilidad y modularidad.

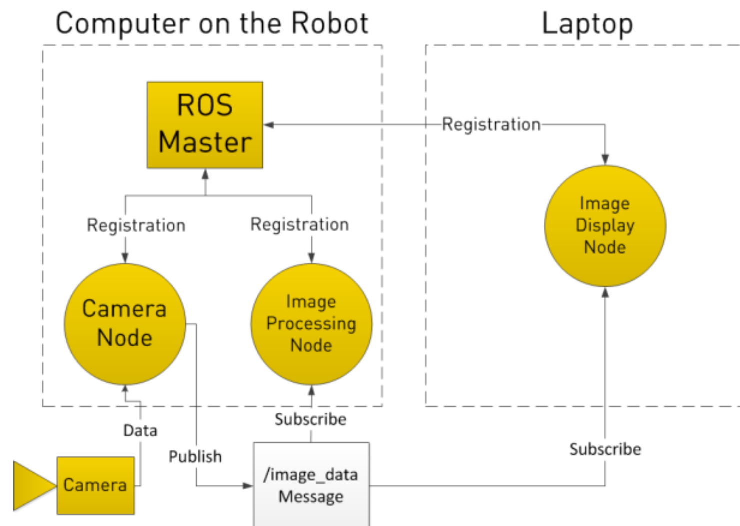


Figura 5.3 Comunicación de ROS con agentes externos.

5.2 Herramientas ROS

5.2.1 Gazebo

La simulación robótica es una herramienta esencial en el tool-box robótico. Un simulador bien diseñado permite probar rápidamente algoritmos, robots diseñados y realizar pruebas de regresión utilizando escenarios realistas. Gazebo ofrece la posibilidad de simular con precisión y eficiencia poblaciones de robots en entornos interiores y exteriores complejos. Genera tanto la realimentación de los sensores como las interacciones físicas entre objetos tratándolos como cuerpos rígidos, a través de gráficos de alta calidad e interfaces programables. Además de tener una comunidad activa.

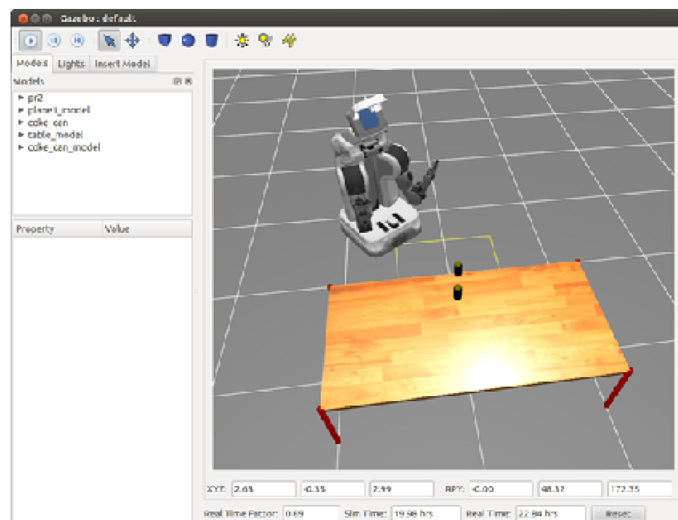


Figura 5.4 Gazebo.

5.2.2 Rviz

RVIZ (ROS Visualization) es un visualizador 3D que permite la visualización de datos de sensor e información de estado del sistema de ROS. Usando RVIZ, se puede ver la configuración actual de Baxter en un modelo virtual del robot, además de las representaciones en vivo de los valores de sensor publicados en los topics de ROS, incluyendo datos de cámara, mediciones de sensores de distancia infrarrojos, datos de sonar, etc. Algunos de los datos más importantes que podemos visualizar se encuentran:

- RobotModel: Muestra una representación visual de un robot en la posición dada (definida por la transformación TF del momento).
- TF: Descrita a continuación, nos permite visualizar los distintos ejes de referencia que conforma nuestro entorno.
- Point Cloud: Muestra los datos de una nube de puntos, configuradas por diferentes opciones disponibles.
- Camera: Crea una nueva ventana con la perspectiva de una cámara, y superpone la imagen en la parte superior de la misma.
- Laser Scan: Muestra los datos de una exploración por láser, con diferentes opciones para los modos de representación, acumulación, etc.
- Otras herramientas como Axes, Effort, Grid, Map, Markers, Path, Pose, Pose Array. . . etc.

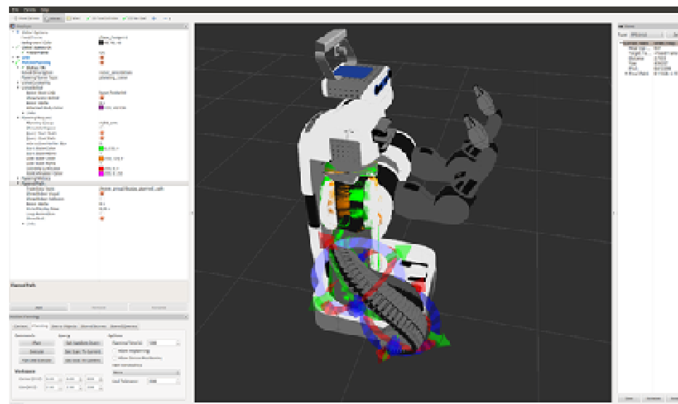


Figura 5.5 RVIZ.

5.2.3 TF

El sistema TF de ROS permite coordinar múltiples sistemas de referencias y mantener la relación entre ellos en una estructura de árbol. TF se distribuye de manera que la información acerca de la coordinación de todos los sistemas de referencia del sistema están disponibles para todos los nodos de la red de ROS. Además del acceso a esta información, nos permite recuperar transformaciones entre ellos, transformar puntos, vectores y otras entidades entre dos ejes de referencia. Algunas aplicaciones son:

- tf_monitor: Imprime información sobre el actual árbol de coordenadas por consola.
- tf_echo: Imprime información sobre la transformación relativa entre dos ejes de referencia.
- static_transform_publisher: Publica un nuevo eje de referencia a través de una transformación estática de un eje ya existente.
- view_frames: Genera un PDF con nuestro árbol de tf.

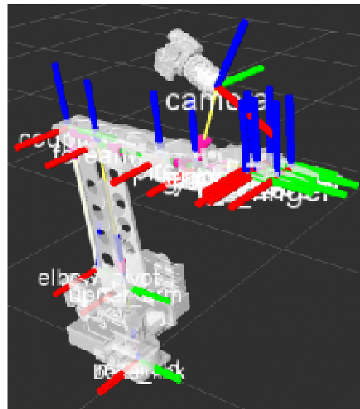


Figura 5.6 TF.

5.2.4 Mercurial

Es un sistema de control de versiones multiplataforma, un software libre para la línea de comandos, implementado principalmente en Python pero que incluye una implementación binaria en C. Escrito originariamente para funcionar en GNC/Linux. Con una interfaz web integrada, sus metas de desarrollo son el rendimiento y escalabilidad, un desarrollo distribuido sin necesidad de servidor, gestión robusta de archivos tanto de texto como binarios y capacidades avanzadas de ramificación e integración manteniendo la sencillez conceptual. Las diferentes facilidades que nos ofrece Mercurial son:

- Monitorizar los archivos añadidos.
- Repositorio compartido.
- Guardar cambios localmente de forma temporal trabajando en diferentes clones.
- Copiar y mover archivos.
- Revisar historial.

- Arreglar errores en versiones anteriores.
- Mezcla o fusión de dos clones diferentes originales de un mismo código.

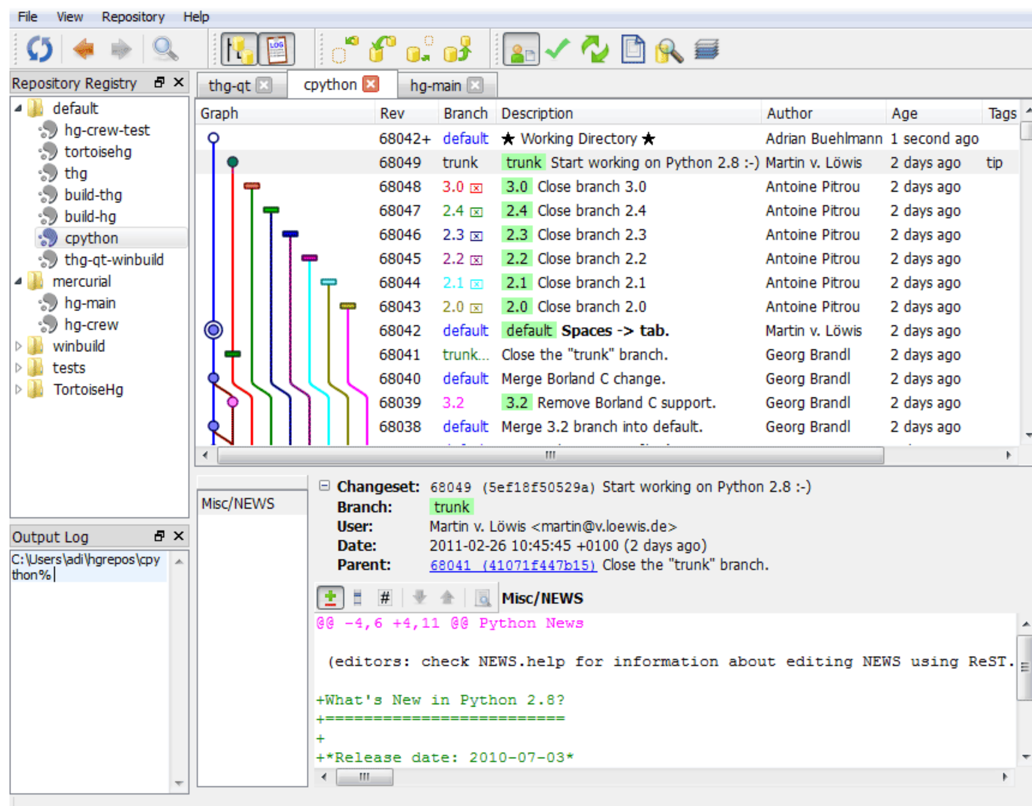


Figura 5.7 Mercurial.

6 Descripción del entorno de simulación

La organización del campeonato EuRoC nos prevé de dos escenarios diferentes en los que se desarrollan las cuatro diferentes tareas que lo componen. La tarea uno (la que nos ocupa en este documento) y la dos se desarrollan en un mismo entorno. Quedando agrupadas la tres y la cuatro en otro entorno diferente con bastantes similitudes que no nos atañe.

El escenario en el que tuvimos que trabajar simula un área de trabajo común que incluye una mesa de trabajo sobre la que se sitúa una pieza poliédrica con agujeros repartidos en grupos de tres en algunas de sus caras. Ésta será para nosotros a partir de ahora la "pieza de trabajo". Al lado de la pieza y fijado a la mesa de trabajo, se encuentra un brazo robótico que se moverá alrededor de la pieza con el fin de practicarle diferentes remaches con el actuador que posee en el extremo. Ya fuera de la mesa, un humano simulado que imita a un operario con sólo dos articulaciones (los dos hombros) se moverá por una extensión de unos 2mx5m de una forma en principio aleatoria.

Por último, la escena de trabajo la completan tres diferentes cámaras: una estéreo localizada junto al actuador final del brazo robótico y dos Kinects. La cámara estéreo tiene como objetivo la localización precisa desde pocos centímetros una vez que se están inmerso en la tarea de remachado con el brazo robótico. La primera de ellas nos proporciona una visión panorámica de toda la escena, tanto de la mesa de trabajo y sus componentes como de la primera Kinect y del humano en todo momento. La segunda Kinect, la que emplearemos para este trabajo, se sitúa a una cierta distancia justo encima de la mesa de trabajo y nos permite ver tanto el brazo robótico y la pieza como la parte frontal del humano, brazos incluidos, cuando éste se sitúa cerca de la mesa.

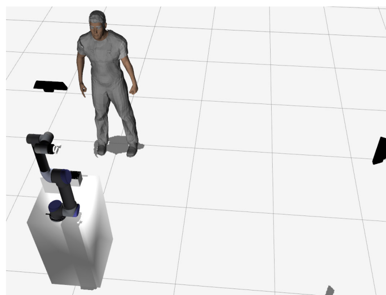


Figura 6.1 Entorno de simulación.

6.1 Cámaras

Acabamos de comentar que para el desarrollo de las tareas no empleamos ni la Kinect 1 ni la estéreo, sólo la Kinect 2, por lo que sólo se explicarán las cámaras Kinect, ya que 1 y 2 tienes características similares menos la que atañe a su localización como se explicaba anteriormente. Estas cámaras nos proporcionan en tiempo real sendas nubes de puntos con información de color (RGB) y profundidad. Como cualquier cámara normal de visión, de un elemento opaco sólo podremos obtener información de las caras que nos muestras, por lo que de la pieza sólo observaremos las caras superiores y del humano sólo observaremos la espalda al alejarse y el torso y la parte superior del brazo cuanto está cerca mirando a la pieza.

6.2 Humano

La simulación de humano se basa en un simple movimiento casi aleatorio por la zona cercana a la mesa de trabajo. Consta simplemente de movimientos de orientación y traslación 2D de una manera un poco terca ya que no posee articulaciones para las piernas, sólo en los hombros. Éstas le permites hacer gestos de señalización con la derecha, cuya mano nunca pierde el gesto de señalización con el dedo índice. La otra mano está cerrada y en principio no sirve más que para crear un gesto no útil o interpretable por nuestra parte. El movimiento del operario está completamente fuera de nuestro control, por lo que sólo nos podremos limitar a observar sus movimientos.

Figura 6.2 Simulación de humano.

6.3 Pieza de trabajo

Como hemos comentado anteriormente, ésta se sitúa en la mesa de trabajo junto al brazo robótico. Está compuesta por 9 caras diferentes con casi siempre ángulos rectos o agudos entre las mismas. Los 33 agujeros para el remachado están agrupado sólo en tres de sus caras, las que el humano puede apreciar de forma completa y casi siempre subagrupados en grupos de tres cada una de las esquinas de las caras menos en una de ellas, que están agrupados en dos o uno. Contamos con una descripción geométrica perfecta de la pieza, lo que significa que una vez que tengamos localizado el marco base de la pieza, sabremos de una forma precisa donde se encuentran todos los agujeros de remache.

Figura 6.3 Planos de la pieza de trabajo.

6.4 Brazo robótico

Como hemos comentado anteriormente, ésta se sitúa en la mesa de trabajo junto al brazo robótico. Está compuesta por 9 caras diferentes con casi siempre ángulos rectos o agudos entre las mismas. Los 33 agujeros para el remachado están agrupado sólo en tres de sus caras, las que el humano puede apreciar de forma completa y casi siempre subagrupados en grupos de tres cada una de las esquinas de las caras menos en una de ellas, que están agrupados en dos o uno. Contamos con una descripción geométrica perfecta de la pieza, lo que significa que una vez que tengamos localizado el marco base de la pieza, sabremos de una forma precisa donde se encuentran todos los agujeros de remache.

Figura 6.4 Brazo Robótico.

7 Evaluación de las tareas

Cada una de las tres diferentes subtareas tiene un propósito diferenciado, subiendo el nivel de dificultad muy sensiblemente. A la hora de la evaluación, las tareas se sucederán en orden preestablecido y comandado por los examinadores, por lo que nuestra solución debe proveer un único código que afrontes todas y cada una de ellas conforme nos las vayan pidiendo. No obstante, a la hora de trabajar en la solución, podremos ejecutar las tareas como nos plazca a modo de entrenamiento. Por ello, hay una serie de canales de información de ROS globales y constantes comunes a todas tareas explicados aquí:

Tabla 7.1 Canales de comunicación con ROS.

T/TF/S/A	L/E	Nombre	Explicación
Servicio	Escritura	execute_task	Para pasar el número de la tarea que queramos que se ejecute.
Servicio	Escritura	init_task	Cargado de todos los recursos necesarios para llevar a cabo la tarea pedida.
Acción	Escritura	solve_task	Inicio de la tarea requerida (solo usado por el simulador).
Topic	Lectura	kinect2/*	Conjunto de topics con información de profundidad + RGB.
TF	Lectura	kinect2_cloud_link	Origen de la nube de puntos de la cámara Kinect2.
TF	Lectura	kinect2_link	Origen de la cámara Kinect2.

A continuación paso a explicar en qué consisten cada una de las tareas, con qué elementos tengo que trabajar y cómo se evaluará cada una de ellas. Destacar que el término "Ground truth" se refiere a topics que estarán solamente durante los periodos de entrenamiento con los cuales podremos evaluar nosotros mismos nuestros resultados. Como es obvio, estos topics no estarán disponibles durante la evaluación final por parte de EuRoC.

7.1 Subtarea 1. Reconocimiento gestual del humano

En esta tarea las cámaras Kinect nos servirán para observar el movimiento del humano por la escena. Se realizarán 20 trayectorias diferentes, en las cuales el humano puede realizar un gesto de señalización o no. Nuestro cometido será determinar al final de cada una de ellas si esa señalización se ha producido, indicándolo en un topic de valor booleano. Tanto los movimientos (posición y ángulos de las articulaciones del humano) como el tiempo transcurrido se generan basándose en valores distribuidos estadísticamente, de forma que al terminar los 20 ejemplos, se hayan experimentado todas las diferentes variantes que ofrece la distribución.

Figura 7.1 Ilustración de la Tarea 1.1.

Tabla 7.2 Canales de comunicación con ROS para la subtarea 1.

T/TF/S/A	L/E	Nombre	Explicación
Topic	Lectura	t1_1_gesturerecog_gt	Información "Ground truth" booleana sobre el gesto actual del humano.
Topic	Lectura	t1_1_trajectory_index	Índice de la trayectoria llevada a cabo actualmente.
Topic	Escritura	t1_1_gesturerecog	Resultado de nuestro análisis. Debe ser true si se ha realizado una señalización.

Evaluación: contará como acierto o fallo cada uno de los diferentes ejemplos recreados así como el tiempo de computación.

7.2 Subtarea 2. Localización de la pieza de trabajo

En este caso, debemos emplear las cámaras para observar y determinar la posición y orientación de la pieza de trabajo que se encuentra sobre la mesa. Cada vez que se reinicia esta tarea, la pieza se reposicionará siguiendo una distribución estadística. Contaremos con un tiempo determinado, al final del cual, nuestra solución vuelve a ser dar la información requerida en un topic que esta vez contendrá una posición y una orientación.

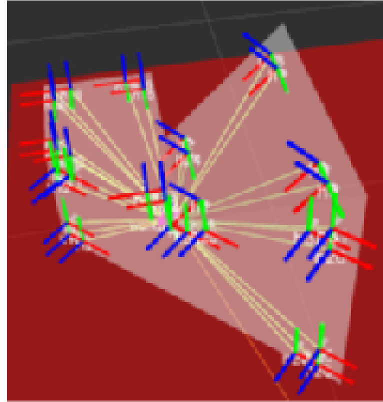


Figura 7.2 Ilustración de la Tarea 1.2.

Tabla 7.3 Canales de comunicación con ROS para la subtarea 2.

T/TF/S/A	L/E	Nombre	Explicación
Topic	Lectura	time_budget	Cuenta atrás del tiempo que nos queda para resolver la tarea.
TFframe	Lectura	workpiece_link	Información “Ground truth” de la posición de la pieza.
Topic	Escritura	t1_2_workpiece	Posición observada de la pieza según nuestro análisis.

Evaluación: Pasados 30 segundos, la posición y orientación que estemos publicando serán comparadas con la posición verdadera. Se evaluará también el tiempo de computación.

7.3 Subtarea 3. Estimación de la posición apuntada

Esta subtarea es la unión de las dos tareas anteriores. En primer lugar, debemos posicionar la pieza y por consiguiente, los agujeros de la pieza. Posteriormente, tendremos que esperar un gesto se señalización, determinar su origen y dirección y finalmente informar de qué agujero es el señalado. Contaremos con un tiempo determinado, al final del cual, debemos dar la información requerida en un topic que contendrá de nuevo una posición y una orientación.

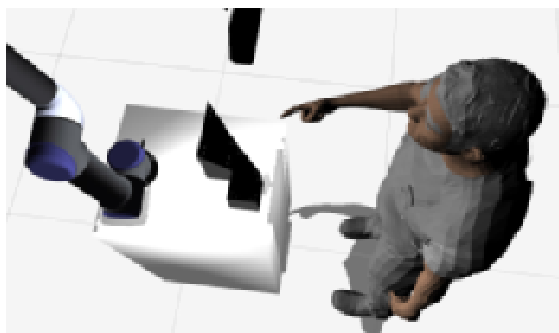


Figura 7.3 Ilustración de la Tarea 1.3.

Tabla 7.4 Canales de comunicación con ROS para la subtarea 3.

T/TF/S/A	L/E	Nombre	Explicación
Topic	Lectura	time_budget	Cuenta atrás del tiempo que nos queda para resolver la tarea.
TFframe	Lectura	workpiece_link	Información "Ground truth" de la posición de la pieza.
TFframe	Lectura	pointed_to	Información "Ground truth" de la posición del agujero señalado.
Topic	Escritura	t1_3_pointedpose	Resultado de nuestro análisis. Posición obervada del agujero señalado.

Evaluación: Pasados 30 segundos, la posición y orientación que estemos publicando serán comparadas con la posición verdadera. Se evaluará también el tiempo de computación.

TASK 1					
TASK	SUB-TASKS	BENCHMARK	METRICS	SCORING INTERVALS	POINTS GIVEN (MAX SCORE 30 POINTS)
Derive from human gesture where the holes for riveting are *Total 30 points*	1. Recognize pointing gesture *Total 10 points*	1.1 Success of gesture recognition (20 gesture samples)	1.1A Number of correctly recognized pointing gestures (true positives) out of 20 tests (containing 10 true positives and 10 true negatives) Penalty points apply[1]	0-4 true positives	0
				5-7 true positives	2
				8-9 true positives	5
				10 true positives	10
	2. Localize object based on CAD data Number of runs: 3, score comes from average *Total 10 points*	2.1 Accuracy of localization	2.1A Deviation in position and orientation[2]	Errors above 10 mm or 10°	0
				Errors from 6 mm to 10 mm or from 5° to 10°	2
				Errors from 3 mm to 6 mm or from 3° to 5°	5
				Errors below 3 mm or 3°	10
	3. Localize position on object at which the human is pointing Number of runs: 3, score comes from average *Total 10 points*	3.1 Accuracy of localization	3.1A Deviation in position relative to object origin	Errors above 10 mm	0
				Errors from 10 mm to 6 mm	2
				Errors from 6 mm to 3 mm	5
				Errors below 3 mm	10

Figura 7.4 Tabla de evaluación de la tarea 1 dividida en subtareas.

8 Tratamiento de la nube de puntos de la cámara Kinect

8.1 Nociones iniciales

Para facilitar la interacción con los nodos, se utilizan las librerías proporcionadas por ROS para C++. En primer lugar se crea el NodeHandle nh_ (->Ver en código= con el que controlar todos los publicadores y subscriptores necesarios. Por lo tanto, a partir de ahora cuando se hable de leer o escribir en algún nodo, se obviará el paso intermedio de la función publicadora o subscriptora necesaria. Notar que todas estas funciones construidas con las clases proporcionadas en las librerías de ROS, si son creadas por mí contendrán barra baja al final de su nombre. Ejemplo: "kinect2_sub_". Para el tratamiento de nubes de puntos, empleo la librería ofrecida por ROS PCL [9] [10].

El desarrollo del código se localiza en tres archivos diferentes, lanzados de manera independiente pero que se suplementan y llaman unos a otros. El primero, "visión_stereo_-node" es el que lleva la voz cantante, conteniendo casi todos las funciones importantes y haciendo llamadas a los otros dos a modo de funciones herramienta. En segundo lugar lanzamos "detect_pointng", código que alberga las funciones encargadas simplemente de las detecciones de los objetos. En tercer y último lugar, utilizamos "algorithm" que, como su nombre indica, ejecuta todos los algoritmos, tanto matemáticos como pueden ser seccionamiento de nubes de puntos o distancias, como el algoritmo de correspondencia.

La función base que regirá el flujo de información entre las diferentes funciones de detección de cada elemento es "kinect2_point_cloud_callback" (->Ver en código) dentro de la clase "VisionSystem". Esta se ejecutará cada vez que el nodo que nos proporciona la nube de puntos de la Kinect 2 sea actualizado, por lo tanto es necesario tener un tiempo de computación y tratamiento de las nubes menor que el tiempo de refresco de la cámara. En el código se pueden observar la iniciación y preparación de las cámaras Kinect 1 y estéreo, cuyo funcionamiento sería idéntico al explicado en este documento para la Kinect 2. Los objetivos se cumplieron satisfactoriamente con sólo una Kinect bien posicionada, por lo que no fue necesario emplearlas y quedaron vacías de contenido útil. Con la excepción de un pequeño código para la estéreo necesario por mi compañero de equipo en cargado de la tarea 2.

En cuanto al seguimiento de los diagramas de flujos para cada una de las tres tareas, lo primero que hay que notar es que tienen muchas partes en común con algunos puntos en los

que divergen. Para que quede claro, podríamos decir que si la tarea 3 se compone de tres tramos (Detección de la pieza, detección de la mano y lanzamiento del rayo, detección del punto señalado), la tarea 1 sólo está compuesta por los dos primeros tramos y la segunda por el primero únicamente. Por ello, la solución propuesta consiste en un código que da en su totalidad la solución de la tarea 3 y que va encontrando en el camino variables de control o "llaves". El sentido de los 6 diagramas de flujos diferentes viene de que primero comentamos de una forma global cómo se consigue la tarea sin entrar en detalle del trabajo de detección del objeto y en su diagrama contiguo nos centramos de una forma más clara y extensa en ese único objetivo de detección del objeto buscado en cada momento.

La información de la tarea actual nos llega por un topic llamado "current_task". Tras leerlo, debemos dar un valor a cada una de las llaves en función del objetivo que le indiquen continuar con la siguiente tarea o finalizar. Antes de empezar a ejecutar nuestro código, lo prioritario es conocer qué tarea es la que nos traemos entre manos.

Tabla 8.1 Definición de las variables de control.

Variable de control	Explicación
workpiece_detection_mode	Llave para iniciar el primer tramo.
safe_computation_power_workpiece	Llave para segundo tramo camino seguro.
best_workpiece_detected	Llave para terminar el primer tramo.
hand_detection_mode	Llave para iniciar el segundo tramo.
best_hand_detected	Llave para terminar el segundo tramo.
hand_detection_mode	Llave para iniciar el segundo tramo.
alternative_hand_detection_mode	Llave para terminar el segundo tramo alternativamente.
alternative_hand_detected	Llave para terminar el segundo tramo alternativo.
pointing_detection_mode	Llave para iniciar el tercer tramo.

Como cabe esperar, el procesado de las imágenes no siempre funciona o simplemente aún no ha aparecido en escena el humano haciendo una señalización. Por ello, el flujo de datos en ese momento no puede continuar ya que no hay un objetivo que conseguir. Esto requiere el uso de otras nuevas variables de control que nos indiquen cuando se ha finalizado satisfactoriamente un tramo y podemos pasar al siguiente si es necesario. El empleo de todas estas variables de control será comprendido más adelante mediante los diferentes diagramas de flujos que explican cada una de las tareas y detecciones empleadas.

El algoritmo para la correspondencia o "matching" de nubes de puntos es el creado por el ingeniero italiano Federico Tombari (->Ver en código). El código empleado en este es un extracto del código original propio suyo. Para poder encontrar tanto la pieza de trabajo como la mano, necesitamos tener un modelo de cada una de ellas previamente guardado y cargado en la solución que demos. Estos modelos se consiguieron mediante recortes de las nubes de puntos obtenidas durante los periodos de entrenamiento y se le pasan al algoritmo de correspondencia cada vez que lo ejecutamos para hacer una búsqueda en otra nube de puntos.

A lo largo del código se presenta la utilización de diferentes parámetros necesarios en el algoritmo de correspondencia (como pudiera ser el radio de búsqueda de puntos vecinos) y los límites en los diferentes cortes que se practican a las nubes de puntos con el fin de

reducir el coste computacional y facilitar el trabajo del algoritmo de correspondencia. A todos estos parámetros se les buscó un valor inicial en un programa propio desarrollado por un compañero de CATEC. Posteriormente fue necesario modificarlos en tiempo real durante la simulación, por lo que empleé un panel de control desde el cual poder manejarlos de una forma fácil y sencilla (->Ver en código). Este panel de control se encuentra en la librería de “PerceptionConfig” de “vision_system_configuration”.

Con la idea de no cargar demasiado de explicaciones técnicas este apartado de solución, incluyo dos anexos en los cuales explico:

- El código para poner en funcionamiento la simulación.
- El código en C++ que resuelve toda la solución propuesta.

La implementación de los diagramas en C++ la he llevado a cabo gracias al gran conocimiento y a su siempre disposición de ayudarme y enseñarme por parte de mis compañeros de CATEC y la página web "C con clase".[11]

8.2 Subtarea 2. Localización de la pieza de trabajo

En el siguiente diagrama muestro una información general de los pasos a seguir, cómo funcionan las variables de control globales y qué nueva información tratada obtenemos en la subtarea 2:

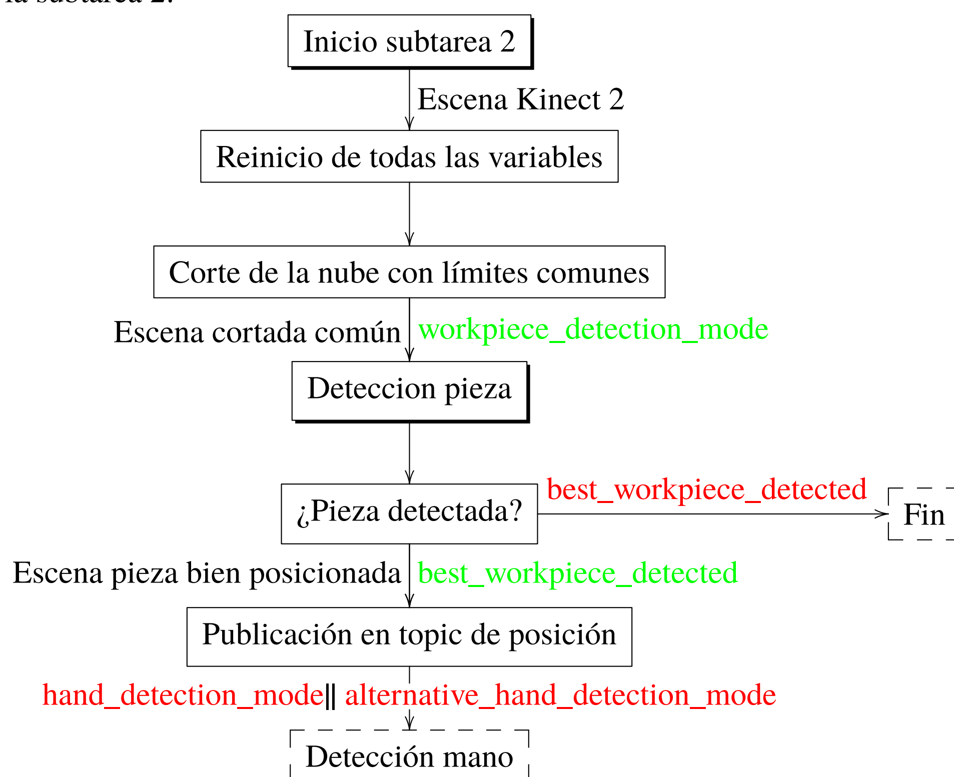


Diagrama explicativo de la subtarea 2

Tabla 8.2 Variables de control para la subtarea 2.

Variable de control	Valor inicial
workpiece_detection_mode	True.
safe_computation_power_workpiece	False.
best_workpiece_detected	False.
hand_detection_mode	False.
best_hand_detected	False.
hand_detection_mode	False.
alternative_hand_detection_mode	False.
alternative_hand_detected	False.
pointing_detection_mode	False.

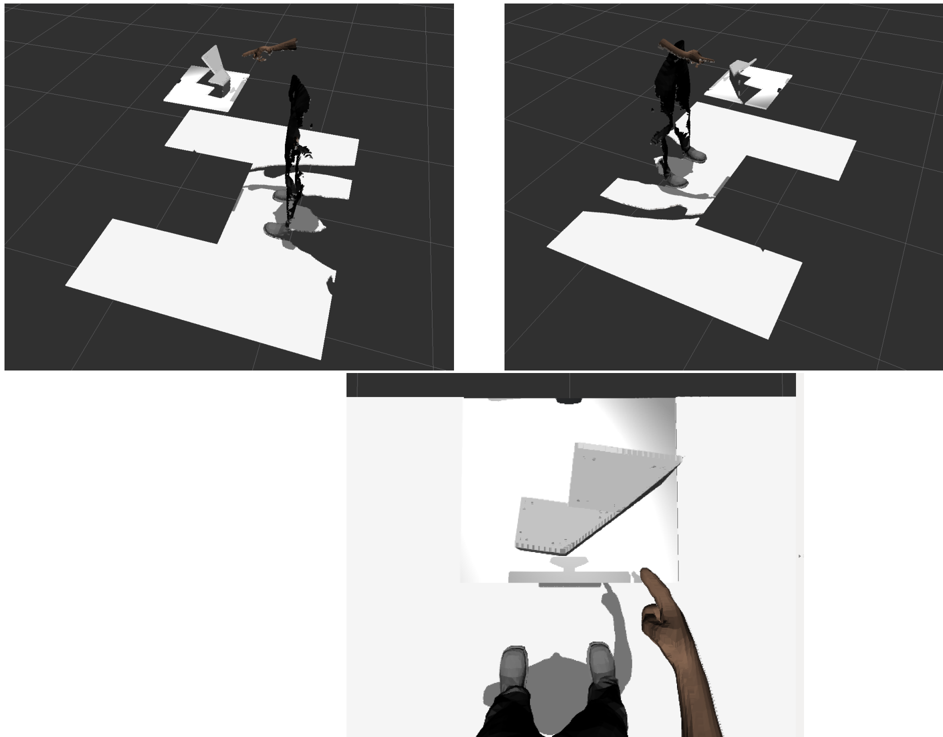


Figura 8.1 Vistas de la nube de puntos proporcionada por la Kinect2.

Procedo a explicar en qué consiste el tramo 1, único componente de esta tarea. Lo primero que deberemos hacer al iniciar una tarea nueva, es el reiniciación de todas las variables que vayamos a emplear en ella. El segundo paso a tener en cuenta es la llave para entrar a la detección de la pieza "workpiece_detection_mode". La primera información con la que contamos es la nube de puntos bruta de la Kinect ofrecida en el grupo de nodos "kinect2/*" guardada por nostoros como "scene_msg". El primer tratamiento que debemos hacerle a la nuebe es un nuevo corte con límites comunes, ya que no toda la parte de la escena nos interesa, sólo la que engloba la mesa y los alrededores, llamada "scene_-filtered_common". Después del procesamiento consistente en un nuevo corte y el proceso de correspondencia básicamente, debemos haber obtenido una nube de puntos que forma nuestro modelo de la pieza localizada en el lugar de la pieza verdadera, nube que llamamos "best_workpiece_cloud_" y cuyo marco podemos ver en "workpiece_estimated_result".

Después de la detección, por supuesto, deberemos comprobar que la pieza ha sido detectada, mirando la llave "best_workpiece_detected". Si no está activa, ha habido algún problema por lo que deberemos esperar a que la información de la Kinect se actualice para volver a intentarlo, pero no podremos seguir. Si por el contrario ya está activa y la detección ha tenido éxito, deberemos publicar los resultados obtenidos en el topic de evaluación, en este caso llamado "t1_2_workpiece" que contendrá el marco antes mostrado.

8.3 Detección de la pieza

Haciendo zoom dentro de la subtarea 2, vamos a ver con más profundidad cómo funciona la detección de la pieza, el algoritmo que se sigue y la información que nos proporciona:

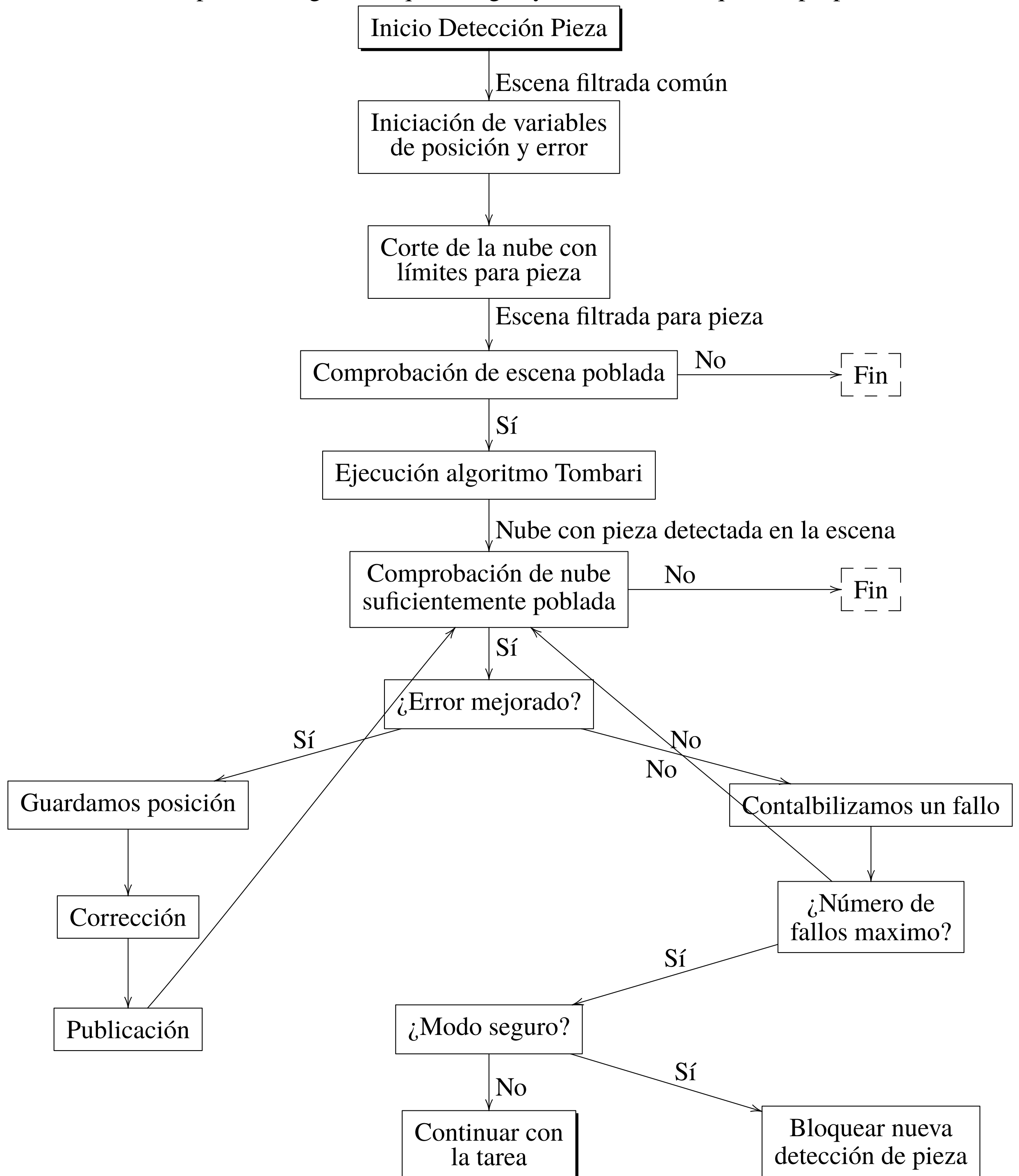


Diagrama explicativo de la detección de la pieza

Para la búsqueda de la posición de la pieza, la información que recibimos es la escena cortada con los límites comunes llamada “scene_filtered_common” y por supuesto el modelo de la pieza de trabajo. El problema es que esta nube de puntos es muy grande y ya sabemos que no será necesario buscar la pieza en la mayor parte de ella ya que esta siempre se encuentra en sobre la mesa de trabajo. Por ello, procedemos a hacerle un nuevo corte a la escena que nos limite el espacio de búsqueda al deseado: “passed_workpiece”. Ahora sí estamos preparados para realizar la búsqueda, no sin antes comprobar que la nube de puntos no está vacía, en cuyo caso tendríamos que devolver un fallo y no podríamos seguir, sabiendo que este sería un error nuestro ya que siempre hay pieza.

Ahora sí es el momento de pasar esa escena doblemente filtrada y el modelo de la pieza a la función del algoritmo de correspondencia. Una vez se haya ejecutado, obtendremos una nube llamada “detected_workpiece_cloud” con marco “worpiece_pose_temp”.de la cual tenemos que comprobar que esté suficientemente poblada (más de 200 puntos). Si esto no es así, es que no hemos tenido éxito en la detección, por lo que deberíamos avisar y parar hasta nuevo intento. Si hemos tenido éxito, se activaría la llave interna “workpiece_detected” y lo informaríamos.

Otro paso muy importante en este punto es el error, calculado como la suma de la distancia desde cada uno de los puntos de nuestro modelo posicionado en la escena hasta su punto más cercano de la nube de puntos de la kinect en la que lo hemos posicionado. Ahora debemos comprobar si el error obtenido en este intento es menor que el error hasta el momento. En caso positivo, tanto el marco como el error recién hallados, pasarán a guardarse como los mejores hasta el momento como “best_worpiece_pose_estimation_” y “best_workpiece_alginment_error_” respectivamente. En caso contrario, vamos a seguir intentando mejorar nuestra detección un número determinado de intentos definido en “best_detected_in_many_detections_” y bloqueando momentáneamente la búsqueda de la mano con la llave “ready_to_detect_hand”. Si se cumple ese número de intentos de detección de la pieza y no hemos conseguido mejorarla, nos quedaremos con la mejor detección hasta la fecha y volveremos a activar “ready_to_detect_hand”.

Cada vez que consigamos una mejora en la localización de la pieza tendremos que reajustar su marco. La explicación de este paso viene del hecho de que nuestro modelo de la pieza tiene el marco que le dio el programa que usamos cuando lo obtuvimos, es decir, que la posición base que lo define nuestro modelo no tiene por qué coincidir con la posición base que define a la pieza real aunque las nubes de puntos coincidan perfectamente (por ejemplo nuestro marco de nuestro modelo podría estar definido en la esquina más alta de la pieza y el marco real en la esquina superior izquierda, eso significa que la relación entre ellos es siempre la misma). Por lo tanto, hay que efectuarle a ese marco una traslación y una rotación constantes transformándose en “workpiece_estimated_result”.

Dependiendo de en qué tarea estemos, estará activa la llave “safe_computation_power_-workpiece”. Ésta se encargará de cerrar el “workpiece_detection_mode” con el fin de no perder más tiempo buscando la pieza. Es decir, quedarnos con la mejor pieza hasta el momento durante el resto de la subtarea hasta que nos evalúen y se vuelva a producir otra trayectoria de prueba del humano dentro de la subtarea, momento en el cual se reinicien todas las variables de control.

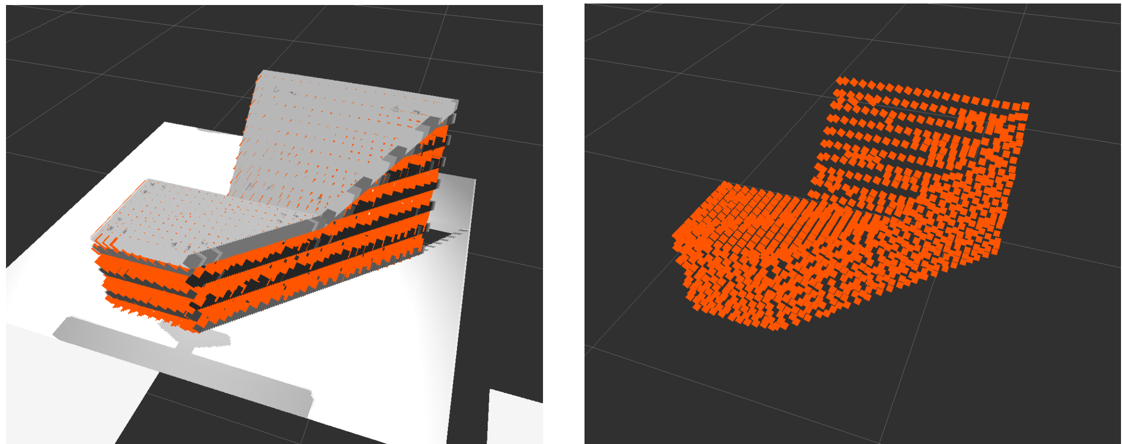


Figura 8.2 Modelo de la pieza detectada posicionado en la nube de la kinect 2.

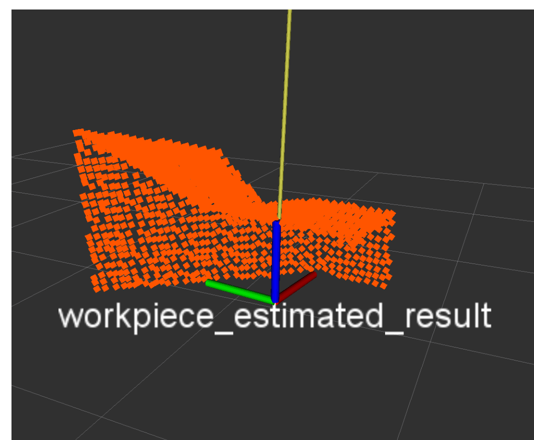


Figura 8.3 Frame corregido de la pieza de trabajo.

8.4 Subtarea 1. Reconocimiento gestual del humano

En el siguiente diagrama muestro una información general de los pasos a seguir, cómo funcionan las variables de control globales y qué nueva información tratada obtenemos en la subtarea 1:

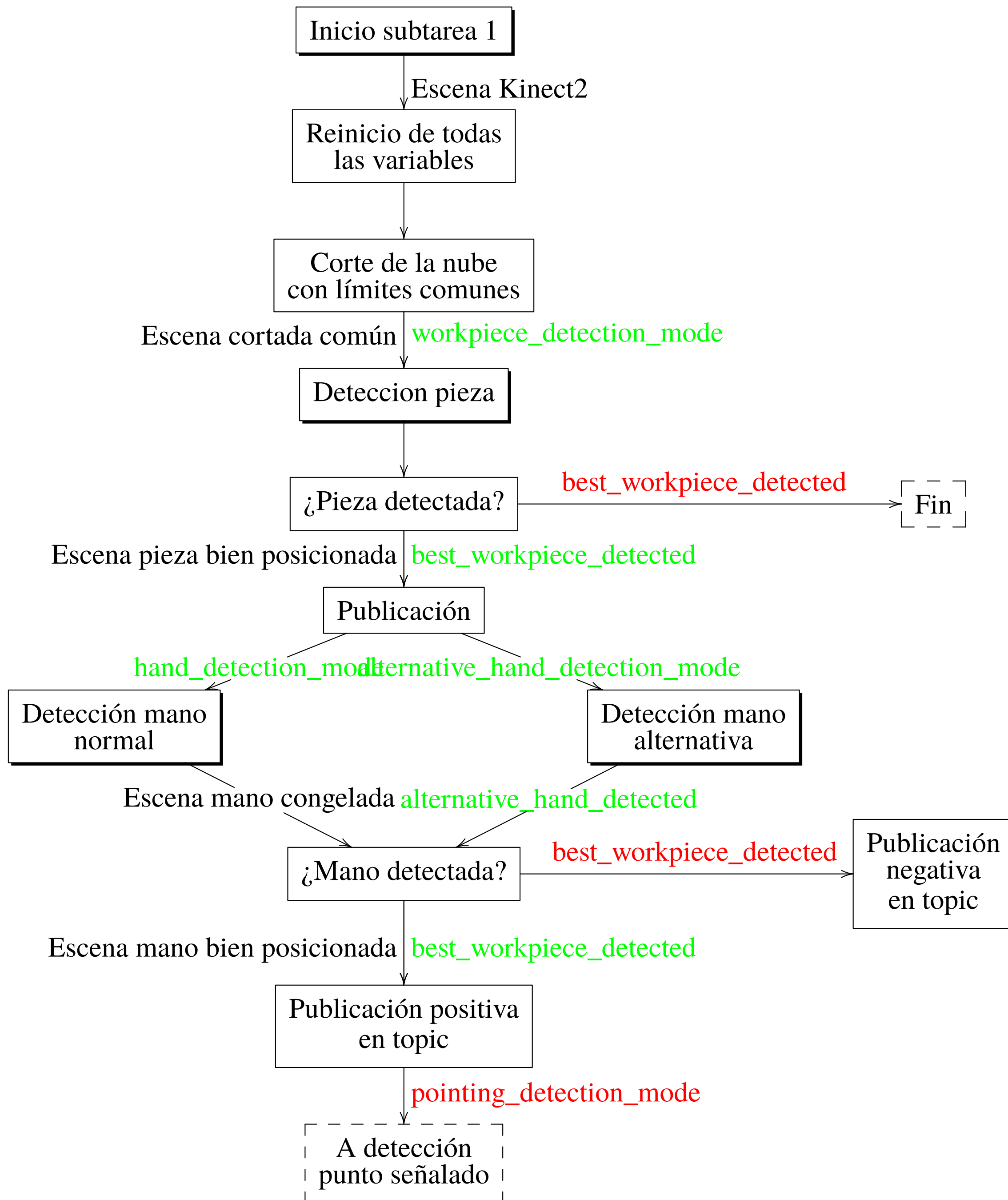


Diagrama explicativo de la subtarea 1

Tabla 8.3 Variables de control para la subtarea 1.

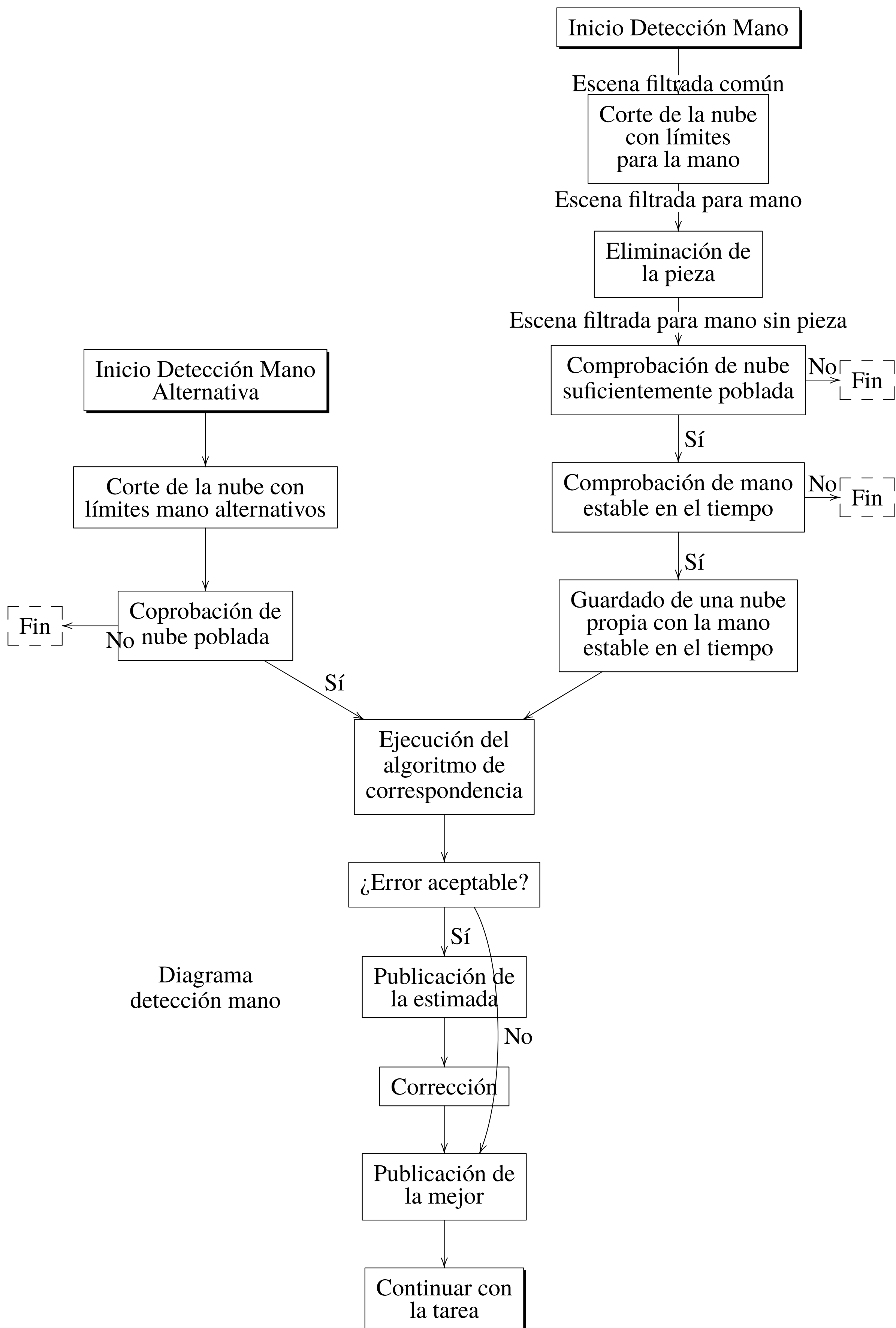
Variable de control	Valor inicial
workpiece_detection_mode	True.
safe_computation_power_workpiece	True.
best_workpiece_detected	False.
hand_detection_mode	True.
best_hand_detected	False.
hand_detection_mode	False.
alternative_hand_detection_mode	True.
alternative_hand_detected	False.
pointing_detection_mode	False.

En esta tarea deben desarrollarse el tramo 1 y 2. El primero lo acabo de desarrollar en el apartado anterior, por lo que procederé a seguir explicándolo desde ese punto. Esta vez no se nos evalúa la determinación de la posición de la pieza, sino saber si se ha producido un gesto o no, por lo que no es necesario publicar esa posición en el topic de evaluación. Sin embargo, la solución es única para las tres tareas, por lo que aún así se publica aunque no se lea por parte del examinador. Una vez determinada la posición de la pieza y abierta a la llave para salir del primer tramo, nos encontramos con las llave que abren el segundo. Ahora contamos con dos nubes de puntos, la de la pieza y la inicial bruta de la Kinect.

Este tramo es peculiar porque tiene dos formas diferentes de detectar la mano: la normal ("hand_detection_mode") y la alternativa ("alternative_hand_detection_mode"). La primera consiste en esperar a que la mano permanezca quieta durante cierto tiempo cerca de la pieza de trabajo. La segunda vía es tan simple como detectar si la mano entra en una determinada región del espacio y si es así detectar la mano en los puntos que hayan entrado en esa región. Estas dos situaciones indican que se ha producido un gesto de indicación. El desarrollo de la detección se llevará a cabo de una forma similar a la de la pieza.

Así pues, podremos encontrarnos con la situación de que no se esté produciendo ningún gesto y por lo tanto no obtengamos ninguna detección (o que nuestro algoritmo esté fallando). En ese caso, no podremos continuar y tendremos que esperarnos a nueva actualización de la nube de puntos ofrecida por la Kinect. En cambio, si se tiene éxito en alguna de las dos vías, se activará la llave “best_hand_detected” o “alternative_hand_detected”. Cualquiera de estas dos nos permitirá terminar este tramo y publicar la nube con el modelo de la mano colocado en la posición de la mano real. Ésta nube se llamará “detected_hand_in_scene” y su marco “hand_estimated_result”. Sin olvidar que tenemos que publicar nuestro resultado en el topic requerido, llamado esta vez “t1_1_gesturerecog” que contiene información booleana, es decir, si alguna de las dos llaves se ha activado, escribiremos un True.

8.5 Detección de la mano



La detección de la mano es algo más compleja ya que durante los entrenamientos nos dimos cuenta de que había una parte pequeña del volumen a la que siempre que llegaba la mano se producía una señalización. Por ello, decidimos tomar un camino alternativo al normal, los dos partiendo de la misma escena inicial con el primer filtro “scene_filtered_common”. En primer lugar explicaré la detección de la mano por el procedimiento normal y posteriormente la alternativa.

Al igual que para la pieza, el primer paso vuelve a ser recortar el espacio en el que buscamos la mano para facilitar la búsqueda computacionalmente obteniendo “hand_cut_filtered”. Sabemos que no todo el tiempo que aparezca la mano en ese espacio será una señalización sino que tendremos que detectar que la mano permanece quita un cierto periodo de tiempo. Para ello, vamos a estar comprando los puntos que compongan ese espacio con los puntos que componían ese espacio en instantes anteriores para detectar si las nubes no han experimentado ninguna diferencia. Esto se hará comparando las distancias entre cada uno de los puntos de una nube con la otra, que proporciona un error llamado “snapshot_distance_-error”.

El número de escenas seguidas que deben ser iguales se lo diremos con la variable “number_of_frames_for_stable_hand”. Así pues, poco a poco iremos obteniendo y actualizando una lista llamada “last_hand_filtered_scene_lists_”. Una vez detectado que esa nube de puntos no se ha movido en un cierto tiempo, tendremos que guardarnos la escena de la gesticulación para poder posteriormente tratarla y buscar en ella la mano. Por lo tanto, hacemos una “congelación” de la escena copiando la última escena filtrada en una congelada en el tiempo “hand_stable_in_time”.

Llegados a este punto, pasamos a explicar la forma alternativa de detección de mano, ya que se unirá a la forma normal en este preciso momento. Partiendo de la misma escena con el corte básico, le efectuamos un tercer corte espacial “alternative_hand_detection_”. Así acotamos el espacio al que sabemos que si llega es porque se está efectuando una señalización. Así pues sólo tendremos que esperar a que el número de puntos que entren en este espacio sea mayor que uno. En ese momento, podremos empezar a detectar la mano en esos puntos indicando antes que hemos detectado que hay mano de una forma alternativa con el indicador “alternative_hand_detected”. Esta vez no hace falta congelar la imagen ya que hemos detectado señalización justo al principio (no como en el normal que tuvimos que esperar un cierto tiempo para asegurarnos de que estaba parada) y tenemos todo el tiempo que la mano esté quieta para efectuarle el algoritmo de correspondencia.

La situación actual sería que, por alguno de los dos caminos, sabemos que se ha producido una señalización y tenemos una escena donde buscar la mano. No queda nada más que hacer que efectuar el algoritmo de correspondencia para colocar nuestro modelo de la mano sobre la mano real. Posteriormente, como es costumbre, comprobamos el éxito observando si hay una cantidad suficiente de puntos en la escena resultante del algoritmo de correspondencia “detected_hand_in_scene” cuyo marco es “hand_icp_pose”. En caso negativo tendríamos que devolver un fallo sabiendo que ha sido el código el que ha fallado ya que al cien por cien seguro había una mano que detectar. Si por el contrario todo ha salido bien, activaremos la llave “best_hand_detected” ya demás, si la hayamos de modo alternativo activaremos también “alternative_hand_detection_and_tombardi_executed_” con el fin de no

repetir ese camino alternativo en la próxima actualización. Posteriormente actuaremos como con la mano y calcularemos el error obtenido y veremos si es el mejor hasta el momento con “best_hand_alginment_error_”. Si no lo es, olvidaremos esta solución, pero si sí lo es, guardaremos como los mejores estos resultados en el marco “best_hand_pose_estimation_”.

Por último, y otra vez como en el modo de detección de pieza, tendremos que hacer ese reajuste de nuestro modelo para asemejarlo con el de la solución en “hand_estimated_-result”.

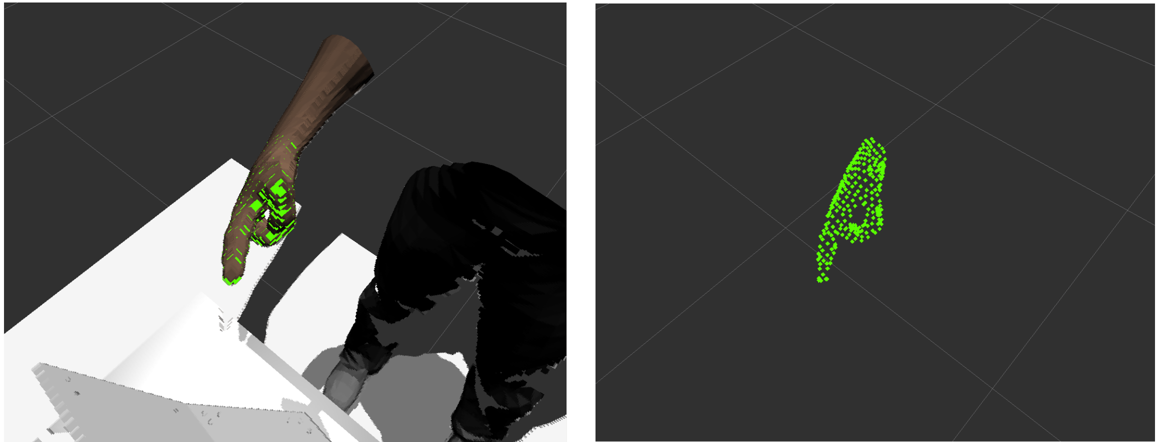


Figura 8.4 Modelo de la mano detectada posicionado en la nube de la kinect 2.

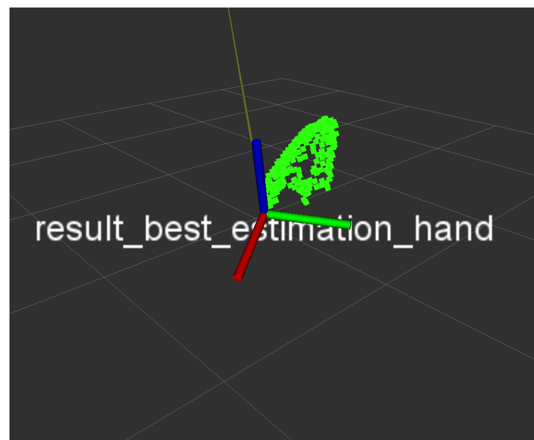


Figura 8.5 Frame corregido de la mano.

8.6 Subtarea 3. Estimación de la posición apuntada

En el siguiente diagrama muestro una información general de los pasos a seguir, cómo funcionan las variables de control globales y qué nueva información tratada obtenemos en la subtarea 3:

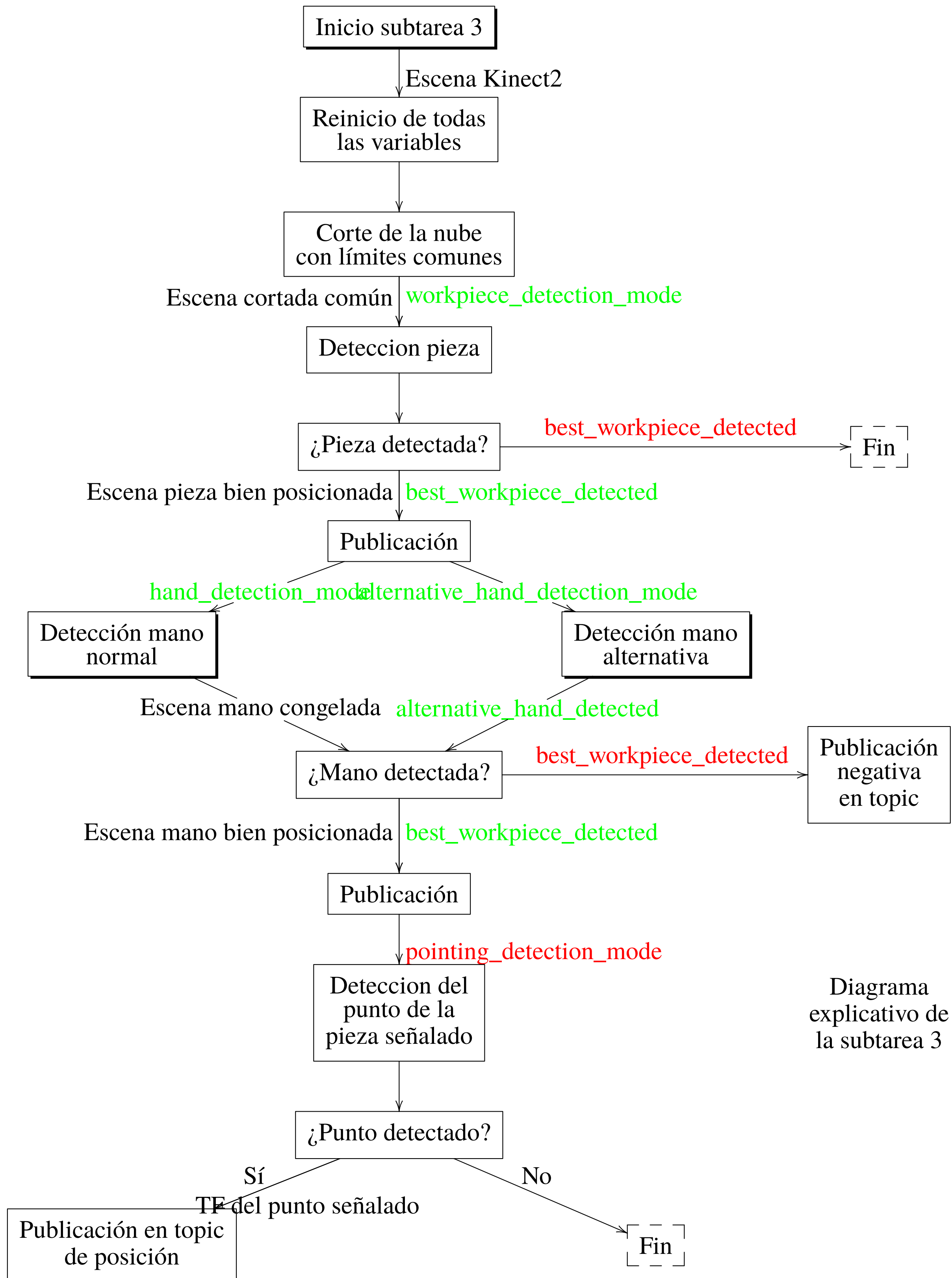


Tabla 8.4 Variables de control para la subtarea 3.

Variable de control	Valor inicial
workpiece_detection_mode	True.
safe_computation_power_workpiece	True.
best_workpiece_detected	False.
hand_detection_mode	True.
best_hand_detected	False.
hand_detection_mode	False.
alternative_hand_detection_mode	True.
alternative_hand_detected	False.
pointing_detection_mode	True.

Recordar que en esta tercera y última tarea ya se encuentran los tres tramos, los dos primeros explicados con anterioridad. Por ello, sólo indicar que llegados a este punto en la tarea 3 no hace falta publicar ninguna información aún pero el código lo hace porque están las tres tareas integradas en una. Para poder continuar, debe estar activa la llave de este tramo, “pointing_detection_mode”. Sabemos que la información con la que contamos con nuestro modelo de la pieza bien localizado, nuestro modelo de la mano bien localizado y siempre podemos seguir observando la escena de la Kinect pero en este momento ya le hemos extraído toda la información que necesitábamos.

Una vez que hemos hallado la posición de la pieza, automáticamente sabemos la posición de cada uno de los agujeros, por lo que sólo queda activar la detección del punto señalado, que lance el rayo desde la mano y detecte cual es el punto que pasa lo suficientemente cerca por él. Si en este punto fallara nuestra solución, sería un problema propio, ya que el espacio en el que observamos la mano implica que si se queda quieta hay una señalización hacia un agujero con toda seguridad. Por lo tanto, si no se obtuviera ningún punto pasaríamos a error. En caso exitoso, obtendríamos el marco final y objetivo de este trabajo, llamado “intersection_with_workpiece”. Si este existe, lo publicamos en el topic evaluable “t1_3_-pointedpose”.

8.7 Detección del punto señalado

Haciendo zoom dentro de la subtarea 3, vamos a ver con más profundidad cómo funciona la detección del punto señalado, el algoritmo que se sigue y la información que nos proporciona:

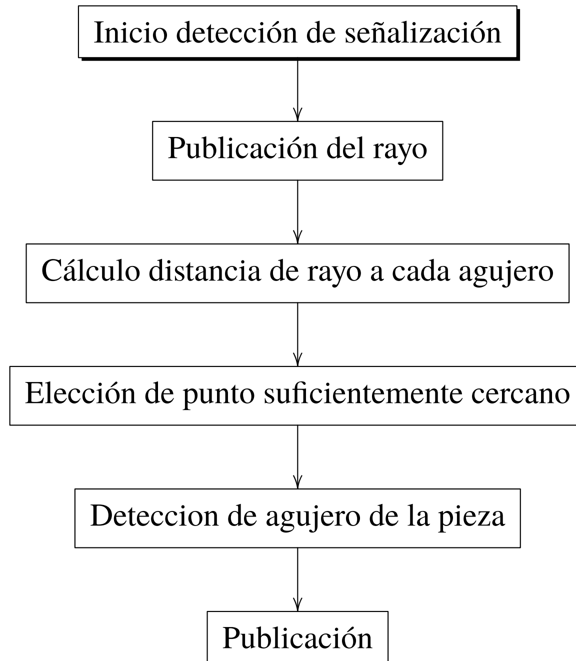


Diagrama explicativo de ladetección del punto señalado

Esta vez ya contamos con los dos marcos de las nubes de puntos que necesitamos, “workpiece_estimated_result” y “best_hand_pose_estimation_”. Por lo tanto no será necesario efectuar ninguna búsqueda con el algoritmo de correspondencia ni hacer ningún corte más. Desde el instante en que tengamos localizada la mano, lanzaremos el rayo desde su marco una vez obtenida la mejor. Una vez comprobado que las llevas de mano y pieza están abiertas, pasaremos a detectar cuál de los puntos de la pieza es el que cumple la distancia mínima con el rayo recién publicado. Así obtendremos el punto “workpiece_intersection_point”. Una vez obtenido el punto de la nube de puntos de la pieza, tendremos que observar cuál es el marco del agujero más cercano al mismo y guardarlo en “nearest_hole”.

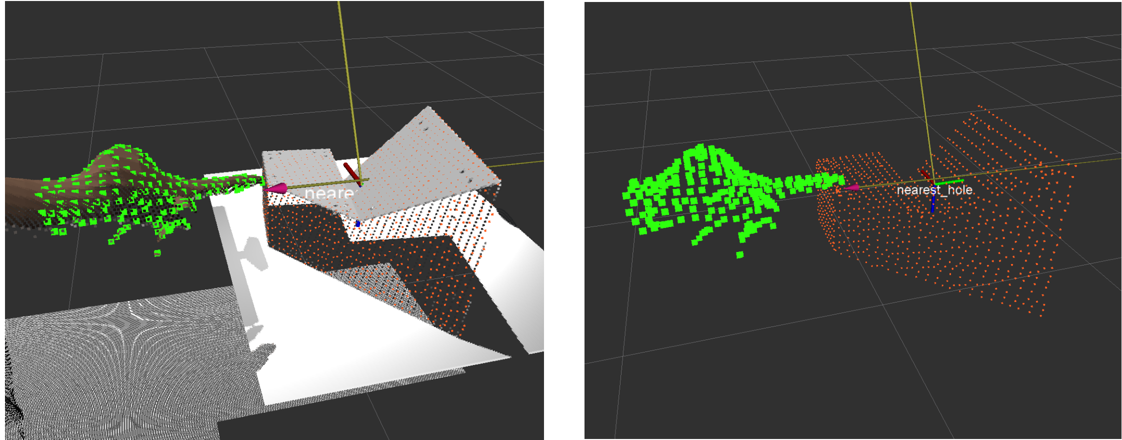


Figura 8.6 Rayo desde mano y punto señalado.

9 Conclusiones y mejoras

La primera conclusión destacable es la derivada de la puntuación obtenida en el concurso EuRoC: de los 32 equipos aspirantes iniciales, sólo 16 llegaron a presentar una solución final, quedando el grupo de CATEC en la 8º posición, consiguiendo así el pase a la siguiente fase al estar entre los 15 primeros. Las puntuaciones de nuestro equipo se obtuvieron de las dos primeras tareas ya que en la tercera y cuarta no se consiguió un resultado aceptable que presentar. En esta primera tara se obtuvo una buena puntuación en las dos primeras subtareas y levemente peor en la última, achacada a la acumulación de errores de precisión.

Presento a continuación una lista de mejoras que se podrían seguir desarrollando para optimizar el resultado:

- Creación de una máquina de estados para el control de los diferentes modos o tramos que se van siguiendo a lo largo de las subtareas.
- Empleo cooperativo de las dos cámaras Kinects ofrecidas en el entorno de simulación en búsqueda de un aumento de precesión en el posicionamiento de los objetos y de una secuencialización del trabajo computacional de ambas. Es decir, usar la Kinect 1 para controlar la posición del humano a lo largo de toda la estación de trabajo y sólo emplear la Kinect 2 una vez que tengamos por seguro que el humano está cerca de la mesa.
- El algoritmo de correspondencia, como se explicó en el apartado de desarrollo teórico, implementa al final del proceso el algoritmo ICP para un último ajuste del posicionamiento. Hubo un intento de aplicarlo pero no se obtuvo un resultado satisfactorio por lo que, al no disponer de más tiempo, se descartó la idea.

Bibliografía

[1] Página web oficial concurso EuRoC *http://www.euroc-proyect.eu*

[2] O. Patsadu, C. Nukoolkit & B. Watanapa, Human gesture recognition using Kinect camera. In Computer Science and Software Engineering (JCSSE), 2012 International Joint Conference, IEEE.

[3] Página web oficial de ROS *http://ros.org*

[4] A. O. Baturone, Robótica: manipuladores y robots móviles, Marcombo, 2005.

[5] F. Tombari, "3D Object Recognition in Clutter with the Point Cloud Library", Workshop on ROS and its applications for Robotic Manipulation (ROS-RM 2014), Alicante, 2014.

[6] *http://robotik-jjlg.blogspot.com.es/2009/03/estado-del-arte-de-robotica-2.html*

[7] *http://www.robotic.de/314*

[8] F. Flórez-Revuelta, Modelo de representación y procesamiento de movimiento para diseño de arquitecturas de tiempo real especializadas, Tesis doctoral, Alicante, 2002.

[9] D. Holz, A. Ichim, F. Tombari, R. Rusu, S. Behnke, "Registration with the Point Cloud Library – A Modular Framework for Aligning 3D Point Clouds", IEEE Robotics and Automation Magazine (RAM), in press.

[10] *http://wiki.ros.org/pcl*

[11] *http://c.conclase.net/curso/*

10 Anexo 1: Arranque del sistema

Para iniciar nuestra infraestructura proporcionada por EuRoC y la resolución al problema propuesto, debemos instalar previamente los paquetes desde el servidor de la organización, una vez instalados debemos distinguir dos extremos de comunicación:

- Nodo servidor: En su ejecución inicializa todo lo necesario para comenzar la tarea, es decir, el entorno de simulación en gazebo (los modelos visuales de todos los elementos del escenario), topics, servicios, TF frames… etc. Una vez inicializado todo, se mantiene a la espera de la ejecución del nodo cliente, que resolverá la tarea.
- Nodo cliente: Contiene el ejecutable para la resolución de la tarea, es decir, la llamada a la función de la máquina de estado, una vez lo arrancamos, resuelve la tarea automáticamente.

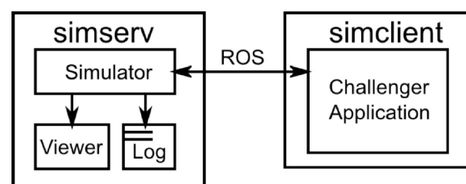


Figura 10.1 Esquema de comunicación servidor-cliente.

Para arrancar el nodo servidor debemos hacer un roslaunch en la consola Linux o sistema operativo usado: roslaunch ipa325_euroc_sim track1.launch euroc_test_com_server:=False

Este launch contiene el nodo servidor llamado ipa325_com_server.py, una vez lanzado el launch, se nos inicializará toda la estructura y permanecerá a la espera cliente, que también se llama por medio de roslaunch: roslaunch ipa325_com_client ipa325_com_client.launch.

De Nuevo permanecerá en stadby la ejecución esperando que le indiquemos que tarea vamos a resolver, esto se lo indicaremos a través de un servicio ROS rosservice, que identificará qué tarea queremos resolver e irá a la parte del código del nodo cliente relativa a la tarea seleccionada: rosservice call solvetaskid:'1'

Una vez iniciado el servicio comenzará automáticamente la resolución de la tarea, que podremos seguir sus resultados a través de Gazebo o RVIZ de forma visual, o a través de los mensajes que van imprimiendo los nodos por pantalla en la consola.

11 Anexo 2: Código

NOTA: Aclarar que en este anexo se añade el código que creo más importante para la comprensión de la solución desarrollada, eliminando gran parte del conjunto del código con el fin de no sobrecargar un anexo ya de por sí extenso. Cada grupo de líneas de código contiene su debido comentario explicativo, dejando el comentario de las líneas que he suprimido por completo o sólo la llamada y el inicio de su definición de algunas funciones. Por supuesto, quien esté interesado en el código completo, no tiene más que ponerse en contacto conmigo y se lo facilitaré en cuanto pueda.

11.1 vision_stereo_node

```
// Librerías

    // Nombres de nuestros modelos para piezas y mano
#define HAND_MODEL_FILENAME "hand_model.pcd"
#define WORKPIECE_MODEL_FILENAME "workpiece_model.pcd"

    // Creación de la clase para visión
class VisionSystem
{ // Inicio de variables booleanas de control

    // Inicio de variables necesarias para la búsqieda de la pieza

    // Inicio de variables necesarias para la búsqueda de la mano

    // Inicio de variables de ROS
ros::NodeHandle nh_;
ros::Subscriber stereo_camera_sub_;
ros::Subscriber kinect1_sub_;
ros::Subscriber kinect2_sub_;
ros::Subscriber current_task_sub_;
ros::Publisher visualization_markers_pub_;
ros::Publisher best_detected_workpiece_count_pub_;
ros::Publisher gesture_recognition_pub_;
ros::Subscriber gesture_index_sub_;
ros::Subscriber time_budget_sub_;
ros::Publisher workpiece_pose_estimation_pub_;
ros::Publisher pointed_hole_pub_;

    // Creación del lector y publicador de TFs
tf::TransformBroadcaster transf_broadcaster_;
```

```
tf::TransformListener transf_listener_;

    // Sistema de configuración de parámetros (dynamic_reconfigure)

    // Inicilización parámetros para el algoritmo de correspondencia

    // Creación del objeto para creación, tratamiento y publicación de nubes de puntos
PointClouds clouds_;
std::map<std::string,ros::Publisher> auxiliar_cloud_publishers_;

    // Inicialización de los nombres de las cámaras

    // Inicialización de variables de control para cada tarea

    public:

    VisionSystem()
{
// Inicialización punteros a nubes de modelos

    // Valoración de los modelos de la pieza y mano
last_trajectory_index_=-1;
number_of_frames_for_stable_hand=2;
best_detected_in_many_detections_=3;

    // Reinicio de todas las variables de reconocimiento
this->restart_workpiece_pose_recognition();
this->restart_gesture_recognition();
ready_to_detect_hand=false;
workpiece_detection_mode=true;
alternative_hand_detection_mode=true;
safe_computation_power_workpiece=false;

    // Lectura y escritura en los canales de ROS proporcionadas y llamadas a funciones necesarias
stereo_camera_sub_ = nh_.subscribe<pcl::PointCloud<pcl::PointXYZ> > ("/simulation_ur5/multisense_sl/-
camera/points2", 2, &VisionSystem::vision_estereo_node_Callback,this);
kinect1_sub_=nh_.subscribe<pcl::PointCloud<pcl::PointXYZ> >("/kinect1/depth/points",2,&VisionSystem::kinect1_-
point_cloud_callback,this);
kinect2_sub_=nh_.subscribe<pcl::PointCloud<pcl::PointXYZ> >("/kinect2/depth/points",2,&VisionSystem::kinect2_-
point_cloud_callback,this);
visualization_markers_pub_=nh_.advertise<visualization_msgs::Marker>("/visualization_marker",1);
// Publicación en los tres topics de evaluación
gesture_recognition_pub_=nh_.advertise<std_msgs::Bool>("/t1_1_gesturerecog",1);
workpiece_pose_estimation_pub_=nh_.advertise<geometry_msgs::PoseStamped>("/t1_2_workpiece",1);
pointed_hole_pub_=nh_.advertise<geometry_msgs::PoseStamped>("/t1_3_pointedpose",1);

    best_detected_workpiece_count_pub_=nh_.advertise<std_msgs::Int32>("/best_workpiece_detected_count",1);
gesture_index_sub_=nh_.subscribe<std_msgs::Int32>("t1_1_trajectory_index",1,&VisionSystem::on_new_-
gesture_started,this);
current_task_sub_=nh_.subscribe<std_msgs::Int32>("current_task",1,&VisionSystem::on_current_task_up-
dated,this);
time_budget_sub_=nh_.subscribe<std_msgs::Int32>("time_budget",1,&VisionSystem::on_time_budget,this);

    //
```

```
    }

    // Función de inicio de la subtarea 2
void init_task_1_2()
{
// Valoración de variables internas de control y de modo
this->safe_computation_power_workpiece=false;
this->workpiece_detection_mode=true;
this->hand_detection_mode = false;
this->ready_to_detect_hand=false;
this->alternative_hand_detection_mode=false;
this->pointing_detection_mode=false;
this->number_of_frames_for_stable_hand= -1;
this->best_detected_in_many_detections_=1;
// Reinicio de la posición de la pieza this->restart_workpiece_pose_recognition();
}
    // Función para controlar, cada vez que timebudget cambie, si ha terminado el tiempo
void on_time_budget(const std_msgs::Int32::ConstPtr& time_buget_msg )
{ if(time_buget_msg->data==30 && current_task==2)
{
this->init_task_1_2();
}

    }

    // Función para actualizar las variables internas de control cuando cambie la tarea
void on_current_task_updated(const std_msgs::Int32::ConstPtr& current_task_msg )
{
this->current_task=current_task_msg->data;

    switch(current_task_msg->data)
{
//Valoración de variables internas de control y de modo de la subtarea 1
case 1:
this->safe_computation_power_workpiece=true;
this->pointing_detection_mode = false;
this->workpiece_detection_mode = true;
this->ready_to_detect_hand = false; //se pone a true luego solo
this->hand_detection_mode = true;
this->alternative_hand_detection_mode=true;
this->number_of_frames_for_stable_hand=1;
this->best_detected_in_many_detections_=2;
this->restart_workpiece_pose_recognition();
this->restart_gesture_recognition();
break;
// Valoración de variables internas de control y de modo de la subtarea 2
case 2:
this->init_task_1_2();
break;
// Valoración de variables internas de control y de modo de la subarea 3
case 3:
this->safe_computation_power_workpiece=true;
this->hand_detection_mode = true;
this->workpiece_detection_mode=true;
this->ready_to_detect_hand=false;
this->pointing_detection_mode=true;
```

```
this->alternative_hand_detection_mode=false;
this->number_of_frames_for_stable_hand=2;
this->best_detected_in_many_detections_=1;
this->restart_workpiece_pose_recognition();
this->restart_gesture_recognition();
this->config_.hand_snapshot_cloud_dist_error=0.02;
break;
//tarea 2
}
}

    // Función para controlar, cada vez que haya un gesto nuevo, volver a reconocer
void on_new_gesture_started(const std_msgs::Int32::ConstPtr& trajectory_index )
{
// Comprobación de una nueva trayectoria e impresión por pantalla del resultado
if(trajectory_index->data!=this->last_trajectory_index_)
{
ROS_INFO("for trajectory %d hand detection was: %d",this->last_trajectory_index_,this->best_hand_detec-
ted);
this->restart_gesture_recognition();
}
this->last_trajectory_index_=trajectory_index->data;
}

    // Función que reinicia todas las variables necesarias cada vez que tengamos que reconocer la pieza
void restart_workpiece_pose_recognition()
{
detected_workpieces_poses.clear();
this->count_best_workpiece_=0;
this->best_workpiece_detected_=false;
best_workpiece_alginment_error_=std::numeric_limits<double>::max();
best_workpiece_cloud_=pcl::PointCloud<pcl::PointXYZ>::Ptr(new pcl::PointCloud<pcl::PointXYZ>);
}

    // Función que reinicia todas las variables necesarias cada vez que tengamos que reconocer un gesto
void restart_gesture_recognition()
{
this->best_hand_detected=false;
this->alternative_hand_detected=false;
this->alternative_hand_detection_and_tombardi_executed_=false;

    pcl::PointCloud<pcl::PointXYZ>::Ptr best_cloud=auxiliar_cloud("best_detected_hand","",ros::Time::now());
this->best_hand_alginment_error_=std::numeric_limits<double>::max();

    this->last_hand_filtered_scene_lists_.clear();
for(int i=0;
i< this->number_of_frames_for_stable_hand;
i++) this->last_hand_filtered_scene_lists_.push_back(pcl::PointCloud<pcl::PointXYZ>::Ptr(new pcl::PointCloud<pcl::PointX
}

    // Función para reconfiguración de los parámetros para el algoritmo de correspondencia
stereo_perception::PerceptionConfig reconfigure_callback(stereo_perception::PerceptionConfig &config,
uint32_t level) {
}
```

```
    // Función para cargar los modelos desde el archivo en el que están en el disco
void load_models(std::string path)
{
// Imprimimos que estamos buscando la pieza, si existe la guardamos y si no, avisamos
// Imprimimos que estamos buscando la mano, si existe la guardamos y si no, avisamos
}

    // Función para controlar la actuación cada vez que varíe la kinect1 (No la empleamos)
void kinect1_point_cloud_callback(const pcl::PointCloud<pcl::PointXYZ>::ConstPtr& scene_msg)

    // Función para controlar la actuación cada vez que varíe la stereo (No la empleamos)
void vision_estereo_node_Callback(const pcl::PointCloud<pcl::PointXYZ>::ConstPtr& scene_stereo)

    // Función para buscar la mano con unos parámetros alternativos

    {
if(alternative_hand_detection_mode)
{
// Corta la nube de la kinect con unos límites alternativos

    point_cloud_cut_filter(*scene_filtered_common,*alternative_hand_detection_,
config_.alternative_hand_detection_cut_x_min,
config_.alternative_hand_detection_cut_x_max,
config_.alternative_hand_detection_cut_y_min,
config_.alternative_hand_detection_cut_y_max,
config_.alternative_hand_detection_cut_z_min,
config_.alternative_hand_detection_cut_z_max);

    // Comprueba si la nube está suficientemente poblada y lo avisa en la variable interna de control
if(alternative_hand_detection_->size()>10)
{
this->alternative_hand_detected=true;
}
}
}

    // Función para buscar la mano
{
if(hand_detection_mode)
{
// Corta la nube de puntos con los límites configurados
point_cloud_cut_filter(*scene_filtered_common,*hand_cut_filtered,
config_.hand_detection_cut_x_min,
config_.hand_detection_cut_x_max,
config_.hand_detection_cut_y_min,
config_.hand_detection_cut_y_max,
config_.hand_detection_cut_z_min,
config_.hand_detection_cut_z_max);

    // Presentar los límites en el panel de control

    // Extracción de los puntos diferentes entre la nube en la que detectamos la workpiece y en la que detecta-
mos la mano
if(best_workpiece_detected_)
{ differences(hand_cut_filtered,this->best_workpiece_cloud_,*substract_filter_hand_scene, 0.05);
}
```

```
else
{
//pcl::copyPointCloud(*hand_cut_filtered,*substract_filter_hand_scene);
}

    // Reinicio del error para la detección de la mano parada
double snapshot_distance_error=0;

    // Comprobación de que esta nube no está vacía
if(substract_filter_hand_scene->size()==0)
{ snapshot_distance_error=-1;
}
// Búsqueda de la nube en la que la mano queda estable
else
{ for(int i=0;i< this->number_of_frames_for_stable_hand;i++) { // Comprobación de que no se vacíe, haya
menos diferencias con la anterior, actualización y haya error menor que el indicado
if(last_frame_i->size()>0 )
{
double error_i=cloud_distances(last_frame_i,substract_filter_hand_scene);
if(error_i>snapshot_distance_error)
snapshot_distance_error=error_i;

    }
// Si se vacía, volvemos al inicio
else
{ snapshot_distance_error=-1;
break;
}
}
}

    // Informamos de la distancia
ROS_INFO("distancia snapshot: %lf",snapshot_distance_error);

    // Declaración de la nube con la mano estable en el tiempo

    // Si hemos detectado la mano quieta, guardamos la nube en ese instante
if(snapshot_distance_error!=-1
&& snapshot_distance_error < this->config_.hand_snapshot_cloud_dist_error) {
pcl::copyPointCloud(*substract_filter_hand_scene,*hand_stable_in_time);
}

    // Si hemos detectado mano por alguan de las dos formas...
if(ready_to_detect_hand &&
( hand_stable_in_time->size()>0
|| (this->alternative_hand_detected && !alternative_hand_detection_and_tombardi_executed_)))
{

    //Iniciación de las variables para el algoritmo de correspondencia

    // Actualización de la variable de control de mano detectada
bool hand_detected=false;
// Función de la detección de la mano
if(hand_stable_in_time->size()>0)
hand_detected=detect_hand(this->hand_model_,hand_stable_in_time,this->config_,detected_hand_in_sce-
ne,hand_pose,hand_icp_pose, this->clouds_, hand_alignment_error);
```

```
else if(this->alternative_hand_detected)
hand_detected=detect_hand(this->hand_model_,substract_filter_hand_scene,this->config_,detected_hand_-
in_scene,hand_pose,hand_icp_pose, this->clouds_, hand_alignment_error);

    // Si la hemos detectado...
if(hand_detected)
{ // Si era por camino alternativo, actualziamos variable interna de control
if(this->alternative_hand_detected)
this->alternative_hand_detection_and_tombardi_executed_=true;

    // Informamos de detección
ROS_INFO("hand detected");

    // Si es el mejor alineamiento hasta el momento, actualizacmos
if(hand_alignment_error < best_hand_alginment_error_ )
{
this->best_hand_detected=true;
best_hand_alginment_error_=hand_alignment_error;
best_hand_icp_=hand_icp_pose;
best_hand_pose_estimation_=hand_pose;
pcl::PointCloud<pcl::PointXYZ>::Ptr best_cloud=auxiliar_cloud("best_detected_hand",scene_frame_id,ros::Time::now());
pcl::copyPointCloud(*detected_hand_in_scene,*best_cloud);

    B=(best_hand_pose_estimation_.inverse())*T_gt;
publish_tf_pose(B,"result_best_estimation_hand","best_estimation_hand");
}

    // Publicamos posición y rayo
publish_tf_pose(hand_pose,"estimation_hand",scene_frame_id);
publish_ray_direction("estimation_hand");
}
// Si no la hemos detectado, informamos
else
{
ROS_INFO("hand undetected");
}
}

    // Si hemos detectado la mejor, le reajustamos la posición y orientación
if(best_hand_detected)
{ this->publish_tf_pose(best_hand_pose_estimation_,"best_estimation_hand",scene_frame_id);
this->publish_tf_pose(best_workpiece_icp_,"best_hand_icp","best_estimation_hand");

    // Relacciona el marco de nuestro modelo con el de referencia "groundtruth"
hand_estimated_result.setOrigin(tf::Vector3(0.9784642600708429, 0.8730360776090249, 0.14144287868431737));
hand_estimated_result.setRotation(tf::Quaternion(0.8275482132140098, 0.5343738274157473, 0.12335703934957688,
-0.11996502916371758));
this->publish_tf_pose(hand_estimated_result, "result_best_estimation_hand", "best_estimation_hand");
}

    //Actualización de la lista de escenas de manos filtradas
}
}

    // Función para buscar la pieza
void workpiece_detection(const pcl::PointCloud<pcl::PointXYZ>::ConstPtr& scene_filtered_common,std::string
```

```
scene_frame_id)
{
if(workpiece_detection_mode)
{
// Iniciación de las variables de posición y error de la pieza

    // Función de detección de la pieza
bool workpiece_detected=detect_workpiece(this->workpiece_model_,scene_filtered_common,this->config_-
,detected_workpiece_cloud,workpiece_pose,workpiece_icp_pose,this->clouds_,workpiece_algnment_error);

    // Una vez la hemos detectado...
if(workpiece_detected)
{
// Informamos de la detección
ROS_INFO("workpiece detected");

    // Si la encontrada es la mejor hasta el momento
if(workpiece_algnment_error < best_workpiece_alginment_error_ )
{ // Actualización de las variables internas de control
count_best_workpiece_=0;
this->best_workpiece_detected_=true;
best_workpiece_alginment_error_=workpiece_algnment_error;
best_workpiece_icp_=workpiece_icp_pose;
best_worpiece_pose_estimation_=workpiece_pose;
// Guardamos la nueva pieza detectada como la mejor
pcl::PointCloud<pcl::PointXYZ>::Ptr best_cloud=auxiliar_cloud("best_detected_workpiece",scene_frame_-
id,ros::Time::now());
pcl::copyPointCloud(*detected_workpiece_cloud,*best_cloud);
he pcl::copyPointCloud(*detected_workpiece_cloud,*this->best_workpiece_cloud_);
detected_workpieces_poses.push_back(std::make_pair(workpiece_pose,workpiece_algnment_error));

    }
// Si la encontrada no es la mejor...
else
{
// Actualizamos el contador de piezas detectadas
count_best_workpiece_++;
//Si llegamos al máximo de piezas buscadas, actualizamos variables de control
if(count_best_workpiece_==best_detected_in_many_detections_)
{
//Si estamos en modo seguro...
if(safe_computation_power_workpiece)
//anulamos una nueva detección de pieza
this->workpiece_detection_mode=false;
// Avisamos de que ya se han hecho las busqeudas de pieza necesarias y podemos comenzar con la mano
this->ready_to_detect_hand=true;
}

    }

    // Publicamos las posiciones de la pieza
this->publish_tf_pose(workpiece_pose,"estimation_workpiece",scene_frame_id);
this->publish_tf_pose(workpiece_icp_pose,"icp","estimation_workpiece");

    }
// Si no se detecta, informamos
```

```
}

    // Cuando detectamos la pieza con menor error
if(best_workpiece_detected_)
{
// Publiacamos la posición de la pieza
this->publish_tf_pose(best_worpiece_pose_estimation_,"best_estimation_workpiece",scene_frame_id);
this->publish_tf_pose(best_workpiece_icp_,"best_icp","best_estimation_workpiece");

    // Hacemos la transformación necesaria para asemejar al "groundtruth" y la publicamos
workpiece_estimated_result.setOrigin(tf::Vector3(0.009501468853467254, 0.1770824852417221, 0.9993797447934892));
workpiece_estimated_result.setRotation(tf::Quaternion(0.7673877326472915, -0.6409471140522919, 0.015848995358506764,
0.007188471090111266));
this->publish_tf_pose(workpiece_estimated_result, "workpiece_estimated_result", "best_estimation_work-
piece");
// Publicación en topic de evaluación de la posición
this->workpiece_pose_estimation_pub_.publish(workpiece_estimated_result);
}
}

    // Función para detectar el punto señalado
void pointing_detection(std::string scene_frame_id)
{
if(pointing_detection_mode)
{

    // Actualización de la variable de control de haber detectado el punto
bool success_pointed_detection=false;
// Cuando tengamos pieza y mano detectadas...
if(best_hand_detected && best_workpiece_detected_)
{
// Creación de la variable de intersección
// Función de detección de punto de la pieza señalado
bool detecting_pose=detect_pointing(this->best_workpiece_cloud_,this->best_hand_pose_estimation_,this-
>config_,intersection_with_workpiece);
// Función de detección de adujero de la pieza más cercano
bool nearest_hole=detect_closest_hole(intersection_with_workpiece,holes_list);
// Publicación en topic de evaluación de la posición
this->pointed_hole_pub_.publish(nearest_hole);
// Si detectamos, informamos y publicamos
if(detecting_pose)
{
// Informamos y publicamos
}
// Si no la detectamos, lo publicamos
}

    }
}

    // Función que controla todo el trabajo de la kinect2 cada vez que se actualiza
void kinect2_point_cloud_callback(const pcl::PointCloud<pcl::PointXYZ>::ConstPtr& scene_msg)
{

    // Iniciación de la escena filtrada
// Corte de la escena con los parámetros comunes
```

```
point_cloud_cut_filter(*scene_msg,*scene_filtered_common,
this->config_.common_cut_x_min,
this->config_.common_cut_x_max,
this->config_.common_cut_y_min,
this->config_.common_cut_y_max,
this->config_.common_cut_z_min,
this->config_.common_cut_z_max);

    // Comunicación de los límites con el panel de control
// Algoritmos de percepción
this->workpiece_detection(scene_filtered_common,scene_msg->header.frame_id);
this->hand_alternative_detection(scene_filtered_common,scene_msg->header.frame_id);
this->hand_detection(scene_filtered_common,scene_msg->header.frame_id,T_gt);
this->pointing_detection(scene_msg->header.frame_id);

    // Publicación de resultados
if(this->best_hand_detected || this->alternative_hand_detected)
publish_ray_direction("result_best_estimation_hand");
recognition_msg.data=true;
this->gesture_recognition_pub_.publish(recognition_msg);
}
// Publicación de las nubes de puntos
this->publish_auxiliar_clouds();
}

    // Función para la creación de nubes
pcl::PointCloud<pcl::PointXYZ>::Ptr auxiliar_cloud(std::string name,const std::string& frame_id, ros::Time
timestamp)
// Función para la publicación de nubes
void publish_auxiliar_clouds()

    // Funciones para la publicación de poses con diferentes entradas
void publish_tf_pose(Eigen::Affine3d pose,std::string child_name, std::string parent_name)
void publish_tf_pose(tf::StampedTransform t,std::string child_name, std::string parent_name)
// Función para la publicación del rayo
void publish_ray_direction(std::string estimation_hand_frame_id)

    //Función para la publicación del punto seleccionado
void publish_tf_point(Eigen::Vector3d point,std::string child_name, std::string parent_name)
{
tf::StampedTransform t;
t.stamp_=ros::Time::now();
t.frame_id_=parent_name;
t.child_frame_id_=child_name;
t.setOrigin(tf::Vector3(point[0],point[1],point[2]));
t.setRotation(tf::Quaternion(0,0,0,1));
this->transf_broadcaster_.sendTransform(t);
}

    // Función para informar cuántas detecciones de pieza y ver hole_guess
void update()

    }

    //Función principal
int main(int argc, char **argv)
```

```
{
ros::init(argc, argv, "vision_system");
VisionSystem perception;
ros::WallRate loop_rate(30);

    while (ros::ok())
{

    ros::spinOnce();
loop_rate.sleep();
perception.update();
}

    return 0;
}
```

11.2 detect_pointing

```
// Librerías

    // Función para la detección de la pieza
bool detect_workpiece(
//inputs
pcl::PointCloud<pcl::PointXYZ>::ConstPtr workpiece_model,
pcl::PointCloud<pcl::PointXYZ>::ConstPtr scene,
const stereo_perception::PerceptionConfig& config,
//outputs
pcl::PointCloud<pcl::PointXYZ>::Ptr detected_workpiece_cloud,
Eigen::Affine3d& workpiece_pose,
Eigen::Affine3d& icp_pose,
PointClouds& clouds,
double& alignment_error)
{ //Corte con los límites para la pieza
point_cloud_cut_filter(*scene,*passed_workpiece,
config.workpiece_cut_x_min,
config.workpiece_cut_x_max,
config.workpiece_cut_y_min,
config.workpiece_cut_y_max,
config.workpiece_cut_z_min,
config.workpiece_cut_z_max);

    // Si no hay puntos tras el corte, dar error máximo
if(passed_workpiece->size()==0)
{
alignment_error=std::numeric_limits<double>::max();
return false;
}

    // Declaración de nubes de puntos característicos

    // Creación de variables de transformación de salida
```

```
// Informamos estar buscando

// Función que ejecuta el algoritmo de correspondencia
tombardiRegistration(workpiece_model,
passed_workpiece,
*detected_workpiece_cloud,
worpiece_pose_temp,
icp_transform_temp,

model_keypoints,
scene_keypoints,

config.use_hugh_workpiece,
config.model_ss_workpiece,
config.scene_ss_workpiece,
config.k_neighbours_workpiece,
config.rf_rad_workpiece,
config.descr_rad_workpiece,
config.cg_size_workpiece,
config.cg_thresh_workpiece,
config.icp_max_iter_workpiece,
config.icp_corr_distance_workpiece,
config.hv_clutter_reg_workpiece,
config.hv_inlier_th_workpiece,
config.hv_occlusion_th_workpiece,
config.hv_rad_clutter_workpiece,
config.hv_regularizer_workpiece,
config.hv_rad_normals_workpiece,
config.hv_detect_clutter_workpiece);

// Detección del error
alignment_error= cloud_distances(detected_workpiece_cloud,workpiece_model);
// Comprueba si la nube está suficientemente poblada
return detected_workpiece_cloud->size()>200;
}

// Función para la detección de la mano
bool detect_hand(//inputs
pcl::PointCloud<pcl::PointXYZ>::ConstPtr hand_model,
pcl::PointCloud<pcl::PointXYZ>::ConstPtr scene,
const stereo_perception::PerceptionConfig& config,

//outputs
pcl::PointCloud<pcl::PointXYZ>::Ptr detected_hand_cloud,
Eigen::Affine3d& hand_pose,
Eigen::Affine3d& icp_pose,
PointClouds& clouds,
double& alignment_error)
{
// Si no hay puntos, dar error máximo
if(scene->size()==0)
{
alignment_error=std::numeric_limits<double>::max();
return false;
}
```

```
    // Creación de las nubes de puntos de las escenas

    // Creación de la pose y transformación

    // Informamos estar buscando
ROS_INFO("looking for hand...");
hand_pose_temp.setIdentity();
// Función que ejecuta el algoritmo de correspondencia
tombardiRegistration(hand_model,
scene,
*detected_hand_cloud,
hand_pose_temp,
icp_transform_temp,

    model_keypoints,
scene_keypoints,

    // Mismas variables del algoritmo de correspondencia, esta vez con valores para la mano
);

    alignment_error= cloud_distances(detected_hand_cloud, hand_model);
return detected_hand_cloud->size()!=0;

    }

    //el resultado en formato matriz de float se convierte a matriz de double

    //se calcula el error entre el modelo alineado con la escena alineados

    // Función para ver la distancia de la mano al punto
inline float distanceRayToPoint(const Eigen::Vector3d& ray_origin,
const Eigen::Vector3d& ray_direction,
const pcl::PointXYZ& p)
{ return ray_direction.cross(Eigen::Vector3d(p.x-ray_origin[0],p.y-ray_origin[1],p.z-ray_origin[2])).norm();
}

    // Función para la detección de la intersección rayo-pieza
bool compute_ray_object_intersection(//inputs
const Eigen::Vector3d& ray_origin,
const Eigen::Vector3d& ray_direction,
const stereo_perception::PerceptionConfig& config,
const pcl::PointCloud<pcl::PointXYZ>& object_to_intersectt,

    //outputs
Eigen::Vector3d& workpiece_intersection_point)
{
//dos opciones lenta pero funciona sguro
//eficiente pero no sabemos que precision vamos a obtener:
//construir un modelo de plano con ransac
//e interseccionar con el rayo

    // Para cada punto de la pieza, ver si es la distancia más corta con la intersección hasta el momento
for(int i=0;i< object_to_intersectt.size();i++)
// Cálculo de la distancia rayo-punto
if(distanceRayToPoint(ray_origin,ray_direction,object_to_intersectt[i])<config.hand_ray_to_point_min_dis-
tance)
```

```
{
workpiece_intersection_point[0]=p[0];
workpiece_intersection_point[1]=p[1];
workpiece_intersection_point[2]=p[2];
return true;
}

    return false;
}

    // Función para la detección del agujero señalado
bool detect_pointing( //inputs
pcl::PointCloud<pcl::PointXYZ>::ConstPtr workpiece_scene,
Eigen::Affine3d& hand_pose,
const stereo_perception::PerceptionConfig& config,
//outputs
Eigen::Vector3d& workpiece_intersection_point)
{

    // Creación de variables de posición
return compute_ray_object_intersection(hand_point,ray_direction,config, *workpiece_scene,workpiece_in-
tersection_point);
}
```

11.3 algorithm

```
// Librerías

    // Función para definir características de nube de puntos (color y tamaño)
struct CloudStyle {
CloudStyle(double r, double g, double b, double size) : r(r), g(g), b(b), size(size) {
}
}

    // Creación de algunos estilos de nubes (color y tamaño)

    // Función para la eliminación de puntos entre dos nubes análogas
void differences(pcl::PointCloud<pcl::PointXYZ>::ConstPtr input_a, pcl::PointCloud<pcl::PointXYZ>::ConstPtr
input_b, pcl::PointCloud<pcl::PointXYZ>&output, double search_radius)

    // Función para la detección de la distancia total entre puntos de nubes
double cloud_distances(pcl::PointCloud<pcl::PointXYZ>::ConstPtr cloud_a, pcl::PointCloud<pcl::PointXYZ>::ConstPtr
cloud_b )

    // Función para cortar nubes de puntos (PointType)
void point_cloud_cut_filter(const pcl::PointCloud<PointType>::ConstPtr input,
pcl::PointCloud<PointType>& output,
std::string axis,
double minvalue,
double maxvalue)

    // Función para cortar nubes de puntos (PointXYZ)
void point_cloud_cut_filter(const pcl::PointCloud<pcl::PointXYZ>& input, pcl::PointCloud<pcl::PointXYZ>&output,float
minx,float maxx,
float miny,
```

```
float maxy,
float minz,
float maxz)

    // Función para aplicación del algoritmo de correspondencia
void tombardiRegistration(pcl::PointCloud<PointType>::ConstPtr model,
pcl::PointCloud<PointType>::ConstPtr scene
pcl::PointCloud<PointType>& output_scene,
Eigen::Affine3f& out_matrix,
Eigen::Affine3f& icp_out_matrix,
pcl::PointCloud<PointType>::Ptr model_keypoints,
pcl::PointCloud<PointType>::Ptr scene_keypoints,
bool use_hough_,
float model_ss_,
float scene_ss_,
int k_neighbours,
float rf_rad_,
float descr_rad_,
float cg_size_,
float cg_thresh_,
int icp_max_iter_,
float icp_corr_distance_,
float hv_clutter_reg_,
float hv_inlier_th_,
float hv_occlusion_th_,
float hv_rad_clutter_,
float hv_regularizer_,
float hv_rad_normals_,
bool hv_detect_clutter_)
{ ROS_INFO_STREAM(ss.str());

    /**
* Compute Normals
*/

    /**
* Downsample Clouds to Extract keypoints
*/

    /**
* Compute Descriptor for keypoints
*/

    /**
* Find Model-Scene Correspondences with KdTree
*/

    /**
* Clustering
*/

    /**
* Stop if no instances
*/
```

```
    /**
* ICP // No empleado
*/

    /**
* Visualization
*/
/*
}
```