Escuela Politécnica Superior

Robot móvil manipulador TiaGo en un entorno hospitalario

Máster Universitario en Automática y Robótica

Trabajo Fin de Máster

Autor:
Pasqual Joan Ribot Lacosta
Tutor/es:
Jorge Pomares Baeza

Universitat d'Alacant
Universidad de Alicante

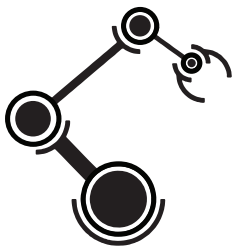
Mayo 2019

Robot móvil manipulador TiaGo en un entorno hospitalario

.

Autor
Pasqual Joan Ribot Lacosta

Tutor/es
Jorge Pomares Baeza
Física, ingeniería de sistemas y teoría de la señal.



Máster Universitario en Automática y Robótica

Escuela
Politécnica
Superior

Universitat d'Alacant
Universidad de Alicante

ALICANTE, Mayo 2019

Resumen

La robótica en el sector de los servicios se está aún iniciando. Con pocas aplicaciones implementadas en el mundo real, la gran parte de lo realizado en este ámbito se encuentra en un contexto de investigación. Esta situación, sin embargo, no continuará por mucho tiempo. Los rápidos avances en sensores y actuadores están culminando en robots como TiaGo, que está expresamente diseñado para este sector. TiaGo es un robot diseñado por PAL Robotics, empresa situada en Barcelona, que combina la tecnología de los robots móviles con un brazo robot antropomórfico.

El presente proyecto se centra en el uso de TiaGo en un entorno hospitalario. En la planta de recuperación de parálisis del Hospital de San Vicente del Raspeig están introduciendo la robótica como herramienta de ayuda y rehabilitación. Durante el trabajo se ha modelado dicha planta en Gazebo, modelo sobre el cual se han hecho las posteriores simulaciones.

El objetivo principal del proyecto es programar TiaGo para dar de comer a un paciente. Esto se ha hecho controlando su base, brazo, torso, mano y cabeza mediante ROS (*Robotics Operating System*). ROS es un *middleware* encargado de la comunicación entre los sensores, los actuadores y los algoritmos de control del robot. Además, también se ha implementado un programa de visión artificial para reconocer la cara del paciente. La visión está implementada combinando *OpenCV*, librería de visión, con *dLib* , librería centrada en el *machine learning.*

Abstract

Robotics in the service sector is still beginning. With few applications currently adapted, a big part of what is being done in this sector is in research. However, this situation will not last long. As actuators and sensors are advancing rapidly, new robots like TiaGo are beginning to appear, which is specifically designed for this purpose. TiaGo is a robot designed by PAL robotics, a Barcelona based company. It combines mobile robot technology with an anthropomorphic robotic arm.

This project revolves around the use of TiaGo in an hospital environment. In the paralysis rehabilitation deparment of the Hospital San Vicente del Raspeig, robotics is being introduced as a helpful tool for recovery. During this project a model of the plant has been created in Gazebo, in which the simulation takes place.

The main objective of this project is to program TiaGo in order to feed a patient. This is done controlling its base, arm, torso, hand and head through ROS (Robotics Operating System). ROS is a middleware in charge of communications between the robot's sensors, actuators and control algorithms. Furthermore, an artificial vision application has been developed in order to detect de patient's face. This application uses the libraries OpenCV, focused on vision, and DLib, focused on machine learning.

Agradecimientos

Agradezco a mi tutor Jorge Pomares Baeza la oportunidad de trabajar en este proyecto. Dar las gracias a la Universidad de Alicante y al profesorado del Máster de Automática y Robótica, por su dedicación y empeño en mi educación. Por último, agradecer a mi familia y a Alicia todo el apoyo y ánimos que me han dado este año.

Índice general

Índice de figuras

Índice de Códigos

1 Introducción

1.1 Motivación

La robótica en el sector servicios presenta muchos retos. Existen preocupaciones reales sobre su papel en roles en los que entra en contacto directo con personas, a diferencia de los entornos en los que ya es un actor fundamental como es la industria.
Una de las principales diferencias entre el sector industrial y el sector servicios es que en la industria los entornos son altamente controlados y predecibles. En servicios, el robot entra en contacto directo con una persona a la que puede estar sirviendo una café u operando del corazón. Para que los robots sean capaces de desenvolverse correctamente en estas situaciones, todos sus componentes, tanto software como hardware, deberán tener una alta precisión y velocidad de reacción.
Sobre todo, en el sector sanitario se ha abierto un debate a nivel mundial que entra en razones éticas y morales (Stahl, 2016) las cuáles no se abarcan dentro del contendio de este trabajo. Aunque la aplicación planteada de dar de comer a un paciente no presente altos riesgos (Song, 2012), la mera presencia de un robot en la planta del hospital ya requiere que este tenga una características específicas. TiaGo es un robot diseñado expresamente para cumplir estas características como son la precisión, alta maniobrabilidad y velocidad de procesamiento. Todas estas razones expresadas resumen la motivación de este trabajo de contribuir a la investigación sobre robótica para servicios, teniendo en cuenta las condiciones de la situación hospitalaria así como las limitaciones de la simulación en Gazebo.

1.2 Objetivos

Los objetivos de este trabajo son:

- Investigar sobre los controladores existentes para TiaGo implementados en ROS y seleccionar controladores para la base, el brazo, la cabeza y la mano.
- Implementar un código que, usando los controladores selccionados, ejecute las órdenes correspondientes en la simulación deseada.
- Desarrollar un programa, usando librerías de *OpenCV* y *dLib*, para el reconocimiento facial del paciente. Se desea reconocer cuando la boca está abierta.
- Modelar un entorno en *Gazebo* con los datos recogidos de la planta de rehabilitación del Hospital de San Vicente del Raspeig.

1.3 Estructura de la memoria

La memoria empieza con el resumen de este trabajo en el que se expone brevemente el objetivo, las razones y desarrollo del trabajo. Posteriormente, la introducción desarrolla más profundamente la motivación y los objetivos.
En el capítulo de marco teórico se exponen los elementos que participan en el desarrollo del trabajo, tanto el *software* utilizado como el robot Tiago. Asimismo, en el siguiente capítulo se incluye la metodología que explica las herrmietnas usadas. En el capítulo de desarrollo se detallan las funciones implementadas y las librerías usadas.
A continuación, en el apartado de resultados se presentan diversas imágenes del proceso, así como explicaciones sobre el funcionamiento de la simulación. Por último, en las conclusiones se incluyen ideas para futuras ampliaciones al trabajo realizado y maneras de mejorarlo que no han podido ser abarcadas.

2 Marco teórico

En este capítulo se exponen los elementos conceptuales, el *software* y las implementaciones relacionadas con la investigación.

2.1 Robótica de servicio

Los robots en tareas de servicio pueden asistir a lo humanos en tareas sucias, repetitivas, aburridas, a distancia o peligrosas. Si cumple una o más de estas características y no necesita de un factor creativo, la terea es susceptible de ser realizada por robots. Según (*ISO8373*, 2012) podemos definir algunos aspectos de estos robots:

- Los robots de servicio son aquellos que realizan tareas útiles para los humanos excluyendo las aplicaciones industriales.
- Estos robots requieren un cierto grado de autonomía, que les permita realizar la tarea ordenada en función de su estado actual y sus sentidos, sin intervención humana.

Según la *International Federation of Robotics* (*IRF*, 2019), cualquier robot que realice las tareas mencionadas es considerado un robot de servicio, tanto si es teleoperado como si es completamente autónomo.
Algunos ejemplos de robots de servicio son:

- Roomba: Posiblemente el robot de servicio más extendido en el mundo. Se encarga de limpiar el suelo. Diseñado por iRobot (*iRobot*, 2019).
- PatrolBot: Robot diseñado para tareas de vigilancia en grandes superfícies (*SMP*, 2019).
- Cosero: Robot similar a TiaGo, pero con dos brazos y una apariencia más humana. Puede realizar múltiples tareas y está muy presente en la investigación del sector (Stückler, 2016).

2.2 Robótica en la medicina

Nuestra aplicación está planteada para que se realice en la planta de rehabilitaicón por parálisis del Hospital de San Vicente del Raspeig. A continuación veremos varias aplicaciones existentes de robótica en hospitales.

- Robots quirúrgicos: La gran precisión de los brazos antropomórficos combinada con una mejora sustancial de la visión artifical han culminado en resultados como los que se observan en la figura 2.1. Cada vez más se están usando herramientas automatizadas para ayudar a los profesionales de la medicina en operaciones.

- Robots para rehabilitación: Para pacientes con mobilidad reducida o parálisis se está popularizando el uso de exoesqueletos para la ayuda de la recuperación del movimiento (Worthen-Chaudhari, 2014). Es uno de los aspectos de la medicina en el que mejores resultados se están obteniendo. Por ello, el Hospital de San Vicente ha iniciado esta colaboración con la Universidad de Alicante.

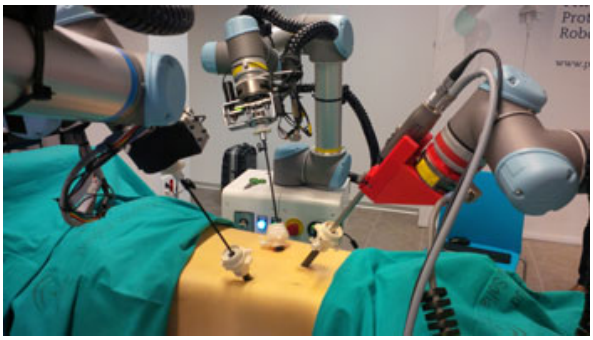


Figura 2.1: Robots UR en una operación *Fuente: Tecnalia* (2019)

2.3 Robótica asistiva

Los robots asistivos son robots de servicio que están diseñados para ayudar a personas con discapacidad o mobilidad limitada a realizar tareas cotidianas. Además, estos robots también deberán ser capaces de interaccionar de manera social con los usuarios (Broekens, 2009), por eso que muchos de ellos tienen forma antropomórfica. La tarea planteada en este trabajo es una tarea de robot asistivo. A continuación se muestran varios ejemplos de trabajos previos realizados en este sector.

- Alogirtmo para sillas robotizadas para cruzar la calle cuando el usuario tiene problemas de visión (Baker, 2005).
- Aplicación para asilos en los que se utilizan múltiples robots heterogéneos. El equipo de robots es capaz de organizar un horario de actividades para los usuarios y localizar al usuario correspondiente en el momento de su actividad (Booth, 2017).
- Estudio de robots sociales en la ayuda al cumplimiento de la rehabilitación en pacientes con problemas cardíacos (Casas, 2018).

En materia de tareas en la asistencia de la alimentación a pacientes con mobilidad reducida existen muchas aplicaciones con el mismo objetivo que el presente trabajo. En la revisión del estado de los robots diseñados para estas tareas, Naotunna remarca que hasta un 15 % de la población mundial se podría ver beneficiada por sistemas como el que se ha desarrollado (Naotunna, 2015). Algunas aplicaciones con el mismo objetivo que han servido de referencia para este trabajo se exponen a continuación:

- El sistema desarrollado por Perera, con un control basado en señales encefalográficas y un *tracking* de la boca. También se detecta cuando la boca está abierta, al igual que en la aplicación de este trabajo (Perera, 2017).

- La aplicación realizada por Tomimoto para un robot ortogonal. Se controla el movimiento mediante la cabeza del usuario, registrada por una cámara Kinect (Tomimoto, 2017).
- El robot diseñado por Gai y Zeng, que es de carcaterísticas más simples (ver figura 2.2), pero eso ayuda a facilitar su adaptación. Tiene capacidad para reconocer la voz del usuario y cambiar la comida que se ofrece (Fkerong Gai, 2016).
- El dispositivo creado por Nozaki, especialmente diseñado para pacientes con mobilidad solo en la cabeza. Es un robot simple con 2 grados de libertad y así se puede imprimir en 3D y extenderse más facilmente (Nozaki, 2016).

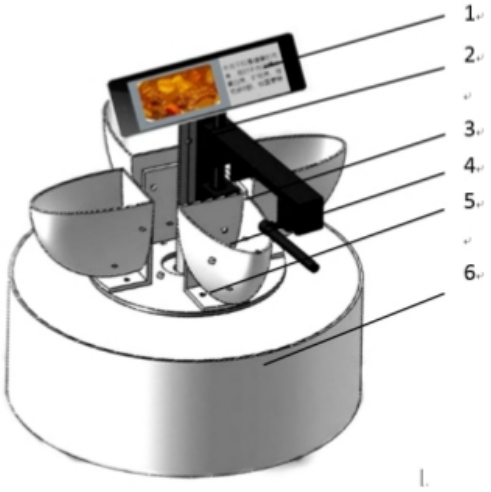


Figura 2.2: Estructura del robot de asitencia para la alimentación.*Fuente: Atlantis Press* (2016)

A parte de las investigaciones en este campo, existen varios robots comerciales expresamente diseñados para el próposito de asistir en la alimentación. Un ejemplo de estos robots se puede ver en la figura 2.3. My Spoon, un robot creado por SECOM (*My Spoon*, 2019), se puede controlar con un joystick o hacer uso del modo automático. Puede trabajar con multitud de comidas y es muy popular en Japón.

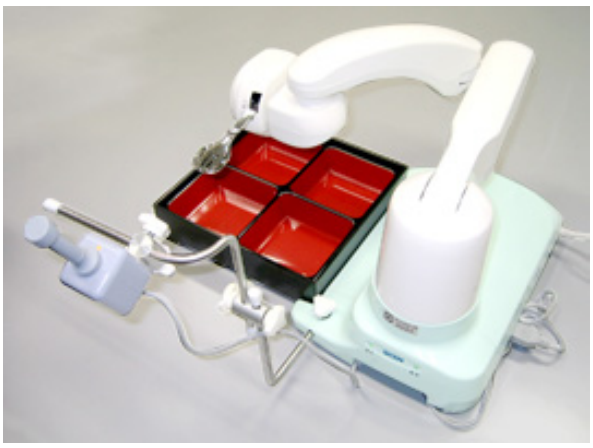


Figura 2.3: Robot MySpoon. *Fuente: SECOM* (2019)

Otro ejemplo de estos robots aparece en la figura 2.4. Obi está diseñado por la empresa que lleva su mismo nombre (*Obi*, 2019). Es fácilmente transportable y implementa nuevas técnicas para coger la comida, reposicionarla o partirla. Su precio de mercado es de 5950 doláres.

Figura 2.4: Robot Obi. *Fuente: Obi* (2019)

2.4 PAL robotics

PAL robotics es la empresa que ha diseñado el robot usado en este trabajo, TiaGo. La empresa está en Barcelona y se centra en el diseño y fabricación de robots humanoides, como los que aparecen en las figuras 2.5 y 2.6.

Figura 2.5: Robot Talos *Fuente: Talos* (2019)

Figura 2.6: Robot Reem *Fuente:Reem-C robot* (2019)

Los robots de las figuras 2.5 y 2.6 son bípedos a diferencia de TiaGo que tiene una base circular con ruedas. PAL es un de las empresas más punteras del sector robótico a nivel nacional, habiendo recibido numerosos premios que reconocen su innovación y capacidad de solucionar los nuevos retos de este sector (Ayri, 2016).

2.5 El robot TiaGo

TiaGo (PAL, 2019c) es el robot elegido para realizar la investigación de este trabajo. El robot se muestra en la figura 2.7.

Figura 2.7: Robot TiaGo *Fuente:TiaGo Robot* (2019)

2.5.1 Modelos

Existen tres modelos diferentes de TiaGo. Sus diferencias se muestran en la figura 2.8. El modelo elegido para para la simulación del presente trabajo es el modelo *STEEL*.

Figura 2.8: Los tres modelos de TiaGo. *Fuente : modelos de TiaGo* (2019)

2.5.2 Especificaciones generales

Las principales especificaciones generales a destacar son (*Fuente : datasheet de TiaGo*, 2019):

- El robot dispone de conenctividad Wi-Fi 802.11 n/ac y Bluetooth 4.0.
- El robot puede disponer de 1 o 2 baterías de 36V/20Ah. Según el número de baterías su autonomía oscila entre 4-5 horas con una y 8-10 horas con dos.
- Lleva un sistema de altavoces y micrófono.
- Su OS es Ubunutu LTS x64 y viene con integración completa de ROS de fábrica.
- Cuenta con dos puertos USB y dos puertos Ethernet.
- El robot tiene un total de 12 grados de libertad sin contar el actuador en el extremo del brazo.
- Según el modelo, su ordenador a bordo tiene las características: i5/ RAM 4GB / SSD 256 GB o i7 / RAM 8GB / SSD 512 GB

2.5.3 Partes principales

A continuación se detallan las especificaciones de las partes del robot.

2.5.3.1 La base

La base móvil usa un accionamiento diferencial y tiene una velocidad máxima de 1 m/s. Está diseñada para funcionamiento *indoor*. Cabe destacar que la misma base es comercializada como un robot por separado que se observa en la figura 2.9.

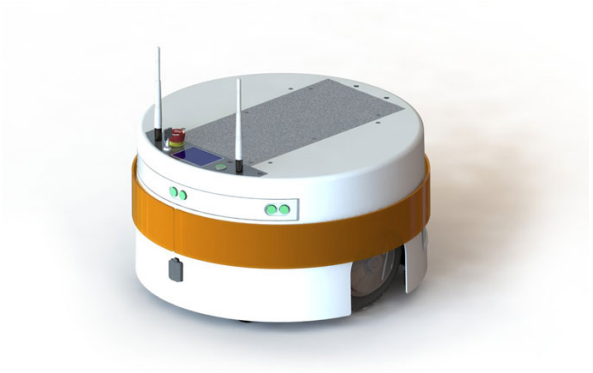


Figura 2.9: Base de TiaGo. *Fuente:TiaGo Base* (2019)

En la base se encuentra el sensor láser que tiene una distancia de detección variable en función del modelo, de entre 5.6 metros y 25 metros. Para detectar lo que se encuentra detrás del robot, existen 3 sónars de 1 metro de detección.

2.5.3.2 El cuerpo

El cuerpo es la parte central de TiaGo y está formado por el brazo y el torso. El torso tiene una articulación prisimática que perimite aumentar la altura del robot en 35cm. El brazo dispone de 7 grados de libertad, una longitud de 87 cm en su máxima extensión, como aparece la figura 2.10, y una capacidad de carga de 3kg.

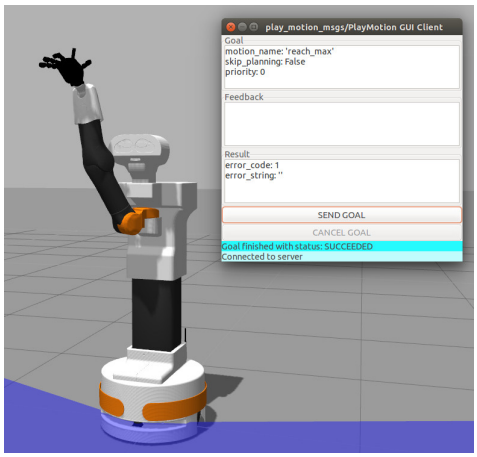


Figura 2.10: Simulación de TiaGo con el torso y el brazo extendidos. *Fuente:TiaGo Cuerpo Extendido* (2019)

2.5.3.3 La cabeza

El cuello de TiaGo tiene 2 grados de libertad que le perimiten mirar hacia cualquier dirección. En el lugar de los ojos hay una cámara RGB-D que proporciona imagen en color y profundidad, pudiendo recrear los entornos mediante nubes de punto. La teconología es la

misma que utilizan la populares cámaras Kinect. En el caso de TiaGo el modelo que el robot lleva incorporado es la cámara Asus Xtion (ver figura 2.11).

Figura 2.11: Cámara Asus Xtion. *Fuente: Asus Xtion* (2019)

2.5.3.4 La mano

Existen dos tipos de actuadores para el extremo del brazo de Tiago: el *gripper* (figura 2.12) y la mano *Hey5* (figura 2.13). Ambos son intercambiables ya que utilizan el mismo método de conexión con el brazo. En la presente simulación se opta por el *gripper*.

Figura 2.12: TiaGo con el gripper *Fuente:Tiago con gripper* (2019)

Figura 2.13: TiaGo con la mano Hey5 *Fuente:Tiago con hey 5* (2019)

2.6 TiaGo en competiciones robóticas

PAL es una empresa que colabora activamente con la investigación en la robótica de servicio de toda la comunidad mundial. Por ello, en numerosas ocasiones han prestado sus robots para competiciones robóticas en las que se plantean situaciones cotidianas en las que TiaGo realiza una función de servicio.

Algunos ejemplos de estas competiciones son:

- *European Robotics League* (2018)
- *IROS Mobile Manipulation Hackathon* (2018) (ver figura 2.15)
- *RoboCup@Home 2018* (2018)

- *World Robot Summit* (2018) (ver figura 2.14)

Figura 2.14: TiaGo en una tarea de limpieza. *World Robot Summit* (2018)

Figura 2.15: TiaGo en una tarea de camarero. *IROS Mobile Manipulation Hackathon* (2018)

2.7 *Software*

En esta sección se hará una introducción de todo el *software* que se ha empleado durante el desarrollo de este proyecto.

2.7.1 ROS

ROS (Standford, 2007) es la base de toda la investigación de este trabajo. Es un *software* que estandariza la comunicación entre los diversos elementos que participan en los procesos robóticos: los sensores, los actuadores, los controladores, los algoritmos, etc. Los principales conceptos básicos para entender ROS son:

- Nodos: Son los ejecutables que participan en la comunicación. Pueden estar programados en C++ o Python. En el caso de este proyecto se ha optado por Python por su rapidez y sencillez.

- *Topics*: Son los canales de comunicación. Están definidos por un nombre específico y un único tipo de mensaje que se puede publicar en ellos. Los nodos se pueden suscribir a los topics para recibir la información publicada en ellos o publicar información para que otros nodos la reciban. En la figura 2.16 vemos un esquema de la relación entre nodos y topics.

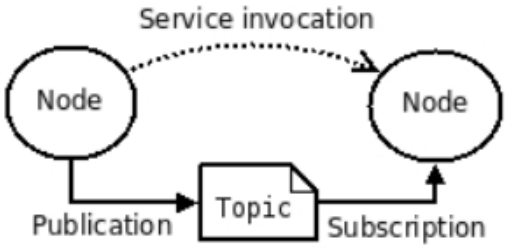


Figura 2.16: Esquema de la comunicación de ROS. *Fuente:ROS wiki* (2019)

- *Callbacks*: Son la rutinas de servicio de interrupción que se generan cuando un nodo suscrito a un topic detecta que se ha publicado algo en ese canal. En esta rutina se hace el tratamiento de datos, como puede ser guardar la posición del robot en el callback generado por el topic donde un sensor de odometría publica las lecturas.
- *Launch*: Son los archivos que ejecutan múltiples nodos. En ellos se suelen lanzar nodos para el robot, su sensores, el entorno de simulación, software de visualización de datos como Rviz etc. Se codifican en formato XML. En el código 2.1 vemos un fragmento de un fichero *launch* que lanza una simulación de TiaGo en Gazebo.

Código 2.1: Ejemplo de Launch de un TiaGo

```
<?xml version="1.0" encoding="UTF-8"?>

<launch>
  <arg name="world" default="empty"/> <!-- empty, ... (see ../worlds) -->
  <!--deprecated argument, you should specify the parameters below-->
  <!-- They are described in tiago.urdf.xacro -->
  <arg name="robot" default="titanium"/>
  <arg name="arm" default="$(eval {'iron': False}.get(arg('robot'), True))
      ↪ "/>
  <arg name="end_effector" default="$(eval {'iron': 'false', 'steel': 'pal-
      ↪ gripper', 'titanium': 'pal-hey5'}.get(arg('robot'), 'pal-gripper'))
      ↪ "/>
  <arg name="ft_sensor" default="$(eval {'titanium': 'schunk-ft'}.get(arg('
      ↪ robot'), 'false'))"/>
  <arg name="laser_model" default="sick-571"/>
  <arg name="camera_model" default="orbbec-astra"/>
  .
  .
  .
</launch>
```

- URDF (*Undefined Robot Description Format*): Son ficheros de definción de un robot. Se definen medinate *links* conectados entre sí por *joints*, así como las propiedades dinámicas, cinemáticas, visuales y de colisión del robot. También se programa en XML. En los códigos 2.2 y 2.3 vemos un ejemplo de un trozo de un URDF en el que se define un *link* y un *joint* de TiaGo.

Código 2.2: Ejemplo de URDF de TiaGo, definiendo un link

```
<link name="${name}_1_link">
    <inertial>
      <!--<origin xyz="0.061191 ${reflect * -0.022397} -0.012835" rpy
          ↪ ="0.00000 0.00000 0.00000"/>-->
      <origin xyz="0.061191 -0.022397 ${reflect * -0.012835}" rpy="0.00000
          ↪  0.00000 0.00000"/>
      <mass value="1.563428"/>
      <inertia ixx="0.002471" ixy="${reflect * -0.001809}" ixz="-0.001202"
               iyy="0.006132" iyz="${reflect * 0.000494}"
               izz="0.006704"/>
               .
               .
               .
  </link>
```

Código 2.3: Ejemplo de URDF de TiaGo, definiendo un joint

```
  <joint name="${name}_1_joint" type="revolute">
    <parent link="${parent}" />
    <child link="${name}_1_link" />
    <xacro:insert_block name="origin"/>
    <axis xyz="0 0 1" />
    <limit lower="${0.0 * deg_to_rad}" upper="${157.5 * deg_to_rad}"
        ↪ effort="${arm_1_max_effort}" velocity="${arm_1_max_vel}" />
    <dynamics friction="${arm_friction}" damping="${arm_damping}"/>

    <safety_controller k_position="20"
                       k_velocity="20"
                       soft_lower_limit="${0.0 * deg_to_rad + arm_eps}"
                       soft_upper_limit="${157.5 * deg_to_rad - arm_eps}" />
  </joint>
```

2.7.2 Gazebo

Gazebo (Gazebo, 2004) es un simulador diseñado especialmente para robótica. Su conveniente diseño hace que se puedan probar rápidamente algoritmos, robots, entrenamientos de IA, etc. En Gazebo se simulan todos los elementos presentes en la realidad, los sensores y actuadores actúan acordes con su entorno. La simulación física es limitada, más allá de la gravedad y colisones simples no se puede decir que sea un simulador demasiado robusto en este aspecto.

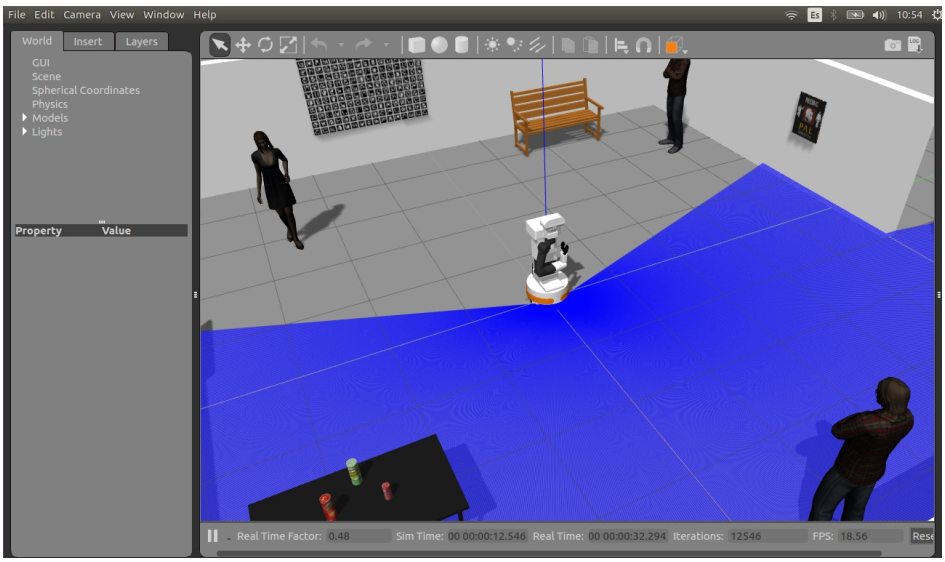


Figura 2.17: Simulación de TiaGo en Gazebo. Fuente: Elaboración propia.

En este trabajo Gazebo se ha utilizado para el diseño de los entornos y para realizar todas las simulaciones con TiaGo. En la figura 2.17 vemos un ejemplo de una simulación en Gazebo, con TiaGo en un entorno predefinido. Destacar de esta figura el campo de visión láser de TiaGo, marcado en azul.

2.7.3 RViz

RViz (*ROS Visualizer*, 2019) es un visualizador de simulaciones robóticas. Se comunica con todos los topics y nodos de ROS y representa de manera gráfica lo que se publica en ellos. En la figura 2.18 aparece una ventana de Rviz durante una simulación de nuestro programa.

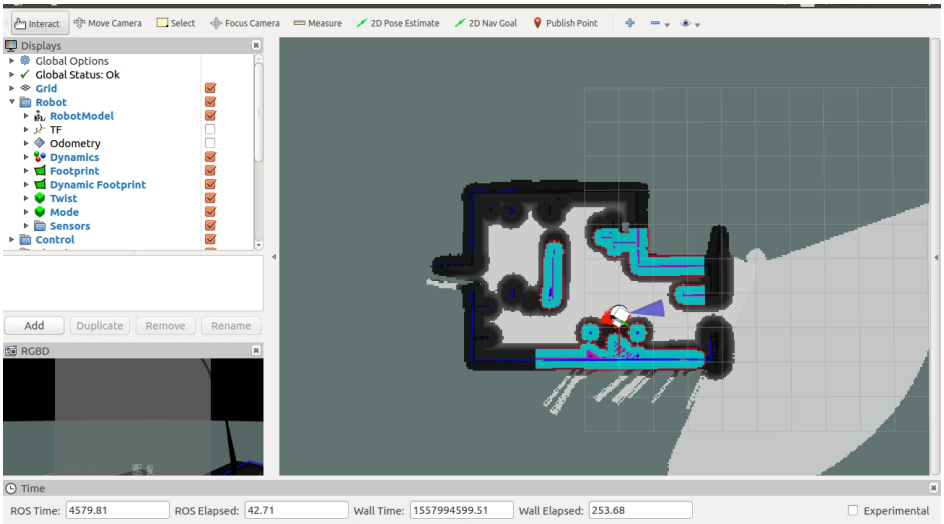


Figura 2.18: Simulación vista en RViz. Fuente: Elaboración propia.

2.7.4 OpenCV y dLib

OpenCV (Intel, 2000) es una librería de programación enfocada principalmente a la visión artificial en tiempo real y en el *machine learning*. Es *open-source* y por eso es muy común en el sector de la investigación.
Por otra parte, dLib (King, 2002) es un librería que engloba muchas herramientas: *networking*, GUI, estructura de datos, procesamiento de imágenes, minado de datos, *machine learning* etc. También es *open-source*.
Para el presente propósito OpenCV se usa en la captura de las imágenes y dLib hace el reconocimiento facial necesario.

2.7.5 Move It

Move It es un proyecto abierto resultado de la colaboración de varias organizaciones y comunidades mundiales (Sucan, 2018). Es una librería que crea varios servicios y funciones para facilitar la planificación en movimientos de brazos robóticos. Las funciones principales y más útiles de Move It se integran en el nodo move group, el cual usamos en la implementación de este trabajo. En la figura 2.19 aparece una esquema detallando los elementos de este nodo y cómo se relacionan entre sí.

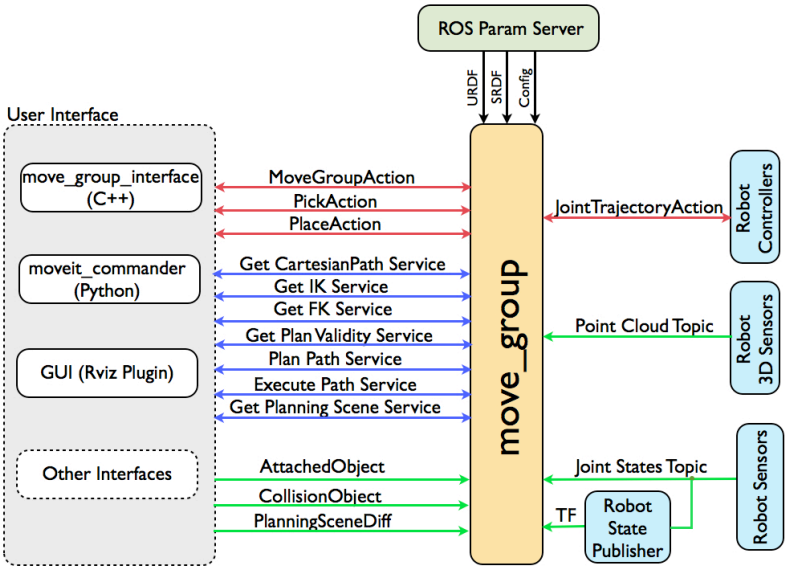


Figura 2.19: Esquema del nodo move group. *Fuente: MoveIt.ros* (2019)

3 Metodología

3.1 Pasos del desarrollo

Para este proyecto la metodología usada ha sido lo más ágil posible a fin de abarcar todas las diferentes vertientes que están presentes en el trabajo, teniendo en cuenta el límite de tiempo y de recursos. Como en todos los proyectos de investigación los diferentes elementos que forman el trabajo han pasado por las siguientes fases:

- Investigación
- Implementación
- Pruebas
- Depuración de errores
- Análisis de resultados

Los elementos que se han creado siguiendo estos pasos se detallan en el capítulo de desarrollo.

3.2 Herramientas usadas

A continuación se enumeran las herramientas necesitadas para el trabajo. De material físico únicamente se ha empleado un ordenador portátil, dado que todo ha sido simulado y no se ha podidio precisar de un TiaGo real. La lista de herramientas *software* es la siguiente:

- ROS
- Gazebo
- RViz
- OpenCV
- DLib
- Paquetes de simulación de TiaGo
- ActionLib
- MoveIt

Para la programación de estas herramientas se ha usado Python por las siguientes razones:

- La gran cantidad de librerías de ROS que están en este lenguaje.
- Es fácil y simple de usar.
- Es un lenguaje con el que el proceso de prueba y error se realiza muy rápidamente.

4 Desarrollo

4.1 Diseño del entorno

En esta sección se expondrá la manera en que se creó el entorno a semejanza de la planta del Hospital de San Vicente del Raspeig (ver figura 4.1).

Figura 4.1: Hospital de San Vicente del Raspeig.*Fuente:Diario Información* (2019)

4.1.1 Recogida de datos

Durante la primera fase de la investigación se realizó una visita a la planta del hospital por la que podría trabajar TiaGo. Se tomaron las medidas necesarias para modelar los entornos en Gazebo lo más fielmente posible.
El mobiliario se tuvo en cuenta al inicio pero, siendo este más facilmente modificable, se opto por variar la composición a fin de facilitar el tráfico del robot.
Durante la recogida de datos también se expusieron varias cuestiones por parte del personal médico:

- La necesidad de sistemas como el que se plantean en este trabajo, ya que hacen que los pacientes no se sientan tan mal por necesitar la ayuda constante de otra persona.
- Los buenos resultados que están dando otros sistemas que ya están implementados. Aunque no tienen ningún robot en la planta, sí que tienen otros sistemas con inteligencia artificial que colaboran con la rehabilitación de los pacientes.
- Las limitaciones que pueden presentar aún estos sistemas. Por la gran cantidad de posibles tareas que pueden realizar, los robots tendrían que tener un nivel de autonomía inalcanzable ahora mismo para que pudieran ser de ayuda al máximo de su potencial.

Con los elementos mencionados anteriormente y los datos recogidos de la planta, procedemos al desrrollo del proyecto propiamente. Por motivos de intimidad de los pacientes no se tomaron fotografías del interior del hospital.

4.1.2 Modelo de la planta

La planta del hospital tiene las siguientes características principales:

- 14 habitaciones: 12 a un lado del pasillo y 2 al otro lado. Esto se debe a que en el lado con menos habitaciones se encuentran salas comunes y despachos que no han sido modelados.
- Las habitaciones se comunican todas por un pasillo de 60 metros de longitud y 3 metros de ancho.
- Las habitaciones en el lado en el que hay 12 tienen balcón, 2 con balcón individual mientras que las otras 10 se conectan con un balcón común.
- Cada habitación mide 7x5 metros y todas disponen de un baño 2x2 metros.
- Las puertas se han mantenido del tamaño real en el modelo ya que eran lo bastante grandes para que pasara TiaGo. No se han modelado las puertas de los baños ya que el robot no puede operar en zonas húmedas.

Con estos datos se ha creado el entorno de la planta en Gazebo que se observa en las figuras 4.2 y 4.3.

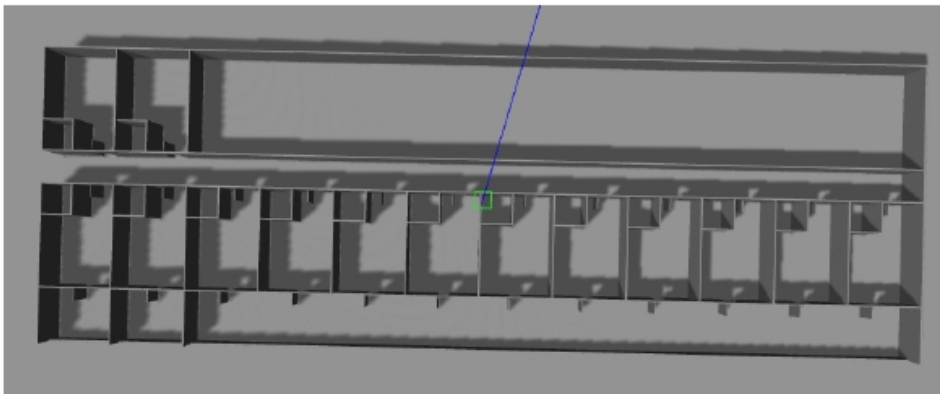


Figura 4.2: Modelo creado de la planta. Vista cenital. Fuente: Elaboración propia

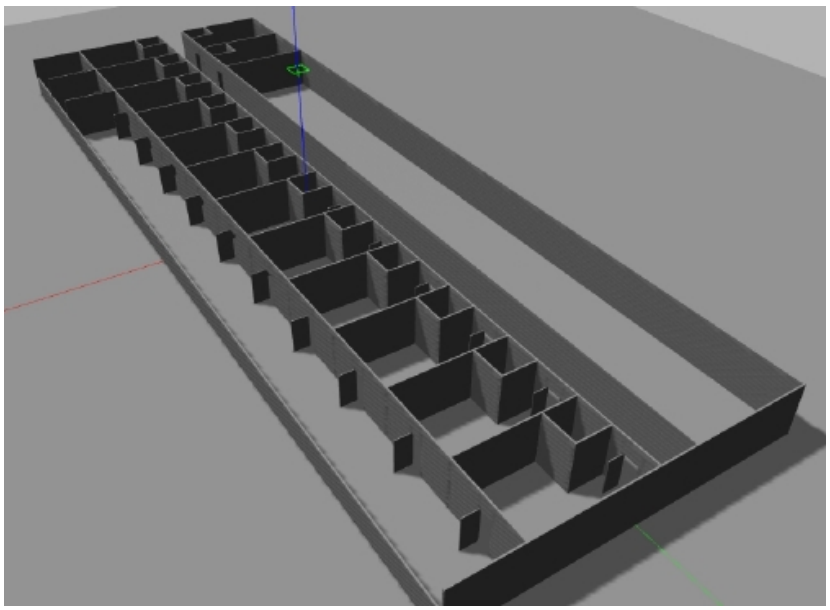


Figura 4.3: Modelo creado de la planta. Vista en picado. Fuente: Elaboración propia

El entorno de la planta no se detalló con mobiliario para cada habitación, ya que el nivel de trabajo sería desproporcionado para algo que no es el foco principal de este trabajo. Por eso se opto por detallar únicamente una habitación como se expone en la sección siguiente.

4.1.3 Modelo de la habitación

Para la habitación, además del tamaño ya mencionado en la sección anterior, hay que resaltar:

- La cama de hospital: Gazebo no tiene ningún modelo de cama de hospital en sus librerías, ni en las instaladas por defecto ni en las *online*. Es por eso que se tuvo que importar un modelo externo. El encontrar un modelo 3D de cama de hospital gratiuto no fue fácil (blenderpl, 2019). Tampoco lo fue importarlo al programa, ya que para ello es necesario definir multitud de parámetros y se encontró poca ayuda al respecto en la web. Finalmente, se consiguió importar el modelo y definir las caracteríticas físicas necesarias con el resultado que se observa en la figura 4.4.

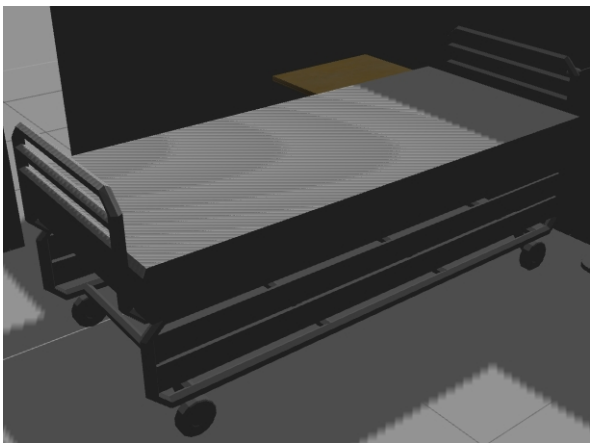


Figura 4.4: El modelo de la cama, en la simulación de Gazebo. Fuente: Elaboración propia

- El mobiliario: Se ha añadido el mobiliario suficiente para que la simulación sea realista pero sin que muchos elementos dificulten el paso del robot o relenticen el tiempo de ejecución. En la figura 4.5 se puede ver uno de los muebles que participan en la simulación, la mesa desde donde TiaGo recoge la comida.

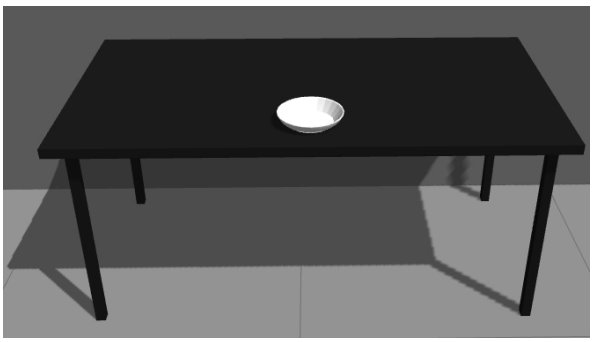


Figura 4.5: Modelo de una mesa y un plato sobre ella. Fuente: Elaboración propia

- El paciente: A la hora de simular el movimiento de alimentar se ha querido aportar realismo con un modelo de paciente. Este simula estar sentado en una silla mientras TiaGo realiza su tarea. En la figura 4.6 se observa la posición del paciente en la habitación.

Figura 4.6: Modelo del paciente en la silla y con la comida enfrente. Fuente: Elaboración propia

Con los elementos mencionados vemos el modelo creado en las figuras 4.7 y 4.8.

Figura 4.7: Modelo de la habitación. Vista cenital. Fuente: Elaboración propia

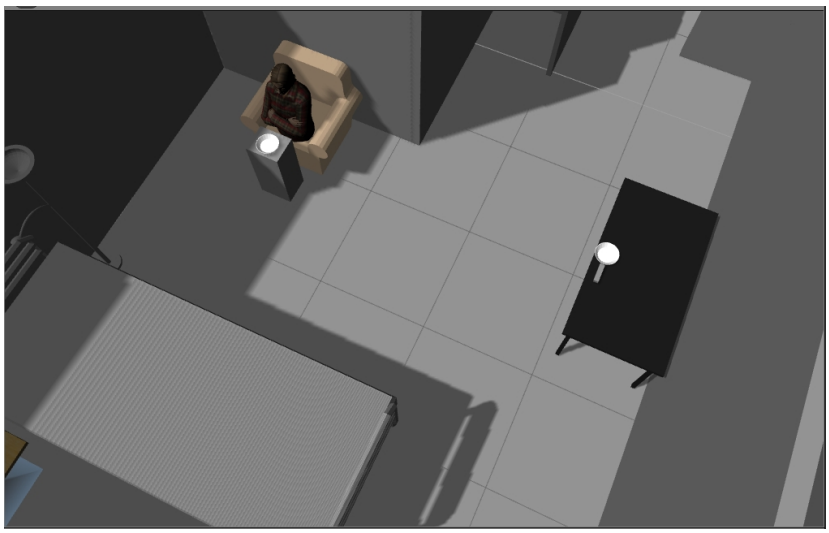


Figura 4.8: Modelo de la habitación. Vista centrada en el área de trabajo del robot. Fuente: Elaboración propia

4.2 Control de TiaGo

Para el control de TiaGo ha sido necesario investigar los controladores existentes para cada parte. Cuando ha sido necesario se han elegido también los planificadores necesarios. En esta sección se expondrán solamente los códigos realizados, mientras que los resultados de cada controlador aparecen en las figuras del capítulo de resultados.

4.2.1 Control de la base

Move base es un stack de ROS. Un stack es un conjunto de paquetes lo cuales están relacionados entre sí de una manera predeterminada con una función.
El objetivo de este stack es la navegación del robot. Para ello se necesitan los siguientes elementos:

- Un mapa del entorno. Move base contiene algoritmos de planificación globales y locales que se combinan, pero no puede planificar solo con uno de los dos tipos. Por tanto, para realizar la planificación global necesita un plano del entorno por el que el robot navegará.
- Sensores de rango. Independientemente del tipo de sensor, láser, utlrasonido, etc., el stack necesita información del entorno inmediato del robot para realizar la planificación local y esquivar aquellos obstáculos que no aparecen reflejados en el mapa del entorno. Además estos sensores le permiten al robot situarse inicalmente en el entorno, comparando lo que ve al inicio de la ejecución con el mapa.
- Sensores de odometría. Sirven para mejorar la precisión del cálculo de situación del robot durante su navegación y medir correctamente las distancias desde el origen o hasta el destino.

- Planificadores locales y globales. Estos vienen por defecto pero se ha tenido que modificar sus especificaciones para reducir la tolerancia al objetivo asegurando mayor precisión en la posición final.
- Una base motora con 2 grados de libertad. La base deberá estar controlada por un topic en el que se publiquen mensajes de tipo Twist. Estos mensajes están compuestos de una velocidad lineal y una velocidad angular, los dos parámetros necesarios para mover el robot en cualquier dirección.

En la figura 4.9 se puede observar un esquema del funcionamiento de move base.

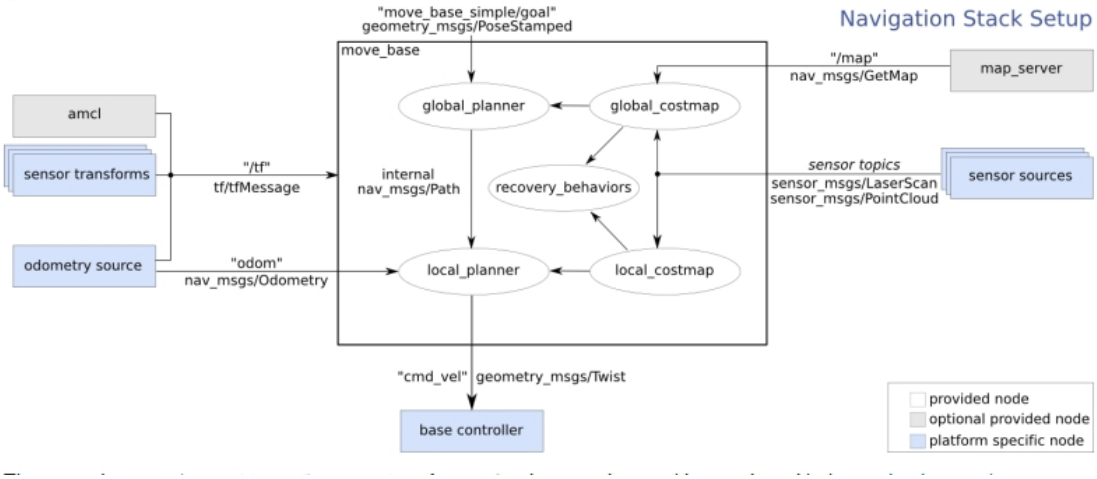


Figura 4.9: Esquema de la comunicación y acciones del stack move base. *Fuente: Move base ROS Wiki* (2019)

El código que se ha implementado para la realización de la navegación del robot se puede ver en 4.1.

Código 4.1: Subprograma para mover la base

```
def move_base(self,x,y,yaw):
    client = actionlib.SimpleActionClient('move_base',MoveBaseAction) # Creamos un cliente para la accion
        ↪ move base
    client.wait_for_server() # Esperamos a que el cliente se cree en el servidor
    base_goal = MoveBaseGoal() # Creamos un mensaje del tipo MoveBaseGoal
    base_goal.target_pose.header.frame_id="map" # Definimos el encabezado del mensaje
    base_goal.target_pose.header.stamp = rospy.Time.now()

    quaternion=tf.transformations.quaternion_from_euler(0,0,yaw) # Transformamos la rotacion de Euler a
        ↪ cuaternion
    base_goal.target_pose.pose.position.x = x #Definimos la posicion deseada en el mensaje
    base_goal.target_pose.pose.position.y = y
    base_goal.target_pose.pose.position.z = 0
    base_goal.target_pose.pose.orientation.x=quaternion[0] #Definimos la rotacion deseada en el mensaje
    base_goal.target_pose.pose.orientation.y=quaternion[1]
    base_goal.target_pose.pose.orientation.z=quaternion[2]
    base_goal.target_pose.pose.orientation.w = quaternion[3]
    client.send_goal(base_goal) # Enviamos el goal
    wait=client.wait_for_result() # Esperamos que se alcance o llegue altimeout
```

```
18    print("Point reached or unreachable") # Confirmamos por consola
```

Como se observa, la librería actionlib se encarga de enviar el objetivo al stack move base, desde ahí ya se controla al robot y se planifica su trayectoria.

4.2.2 Control del brazo y torso

El control del brazo se realiza usando funciones de la librería Move It. Se debe definir un punto en 3 dimensiones (x,y,z) y una orientación expresada en ángulos de Euler (roll,pitch,yaw). Este punto objetivo estará referenciado a la base del robot para facilitar su descripción. El código para mover el brazo aparece en 4.2.

Código 4.2: Subprograma para mover el brazo.

```
1 def move_arm(self,x,y,z,roll,pitch,yaw,wait):
2
3     arm_goal = geometry_msgs.msg.PoseStamped() # Creamos un mensaje de tipo PoseStamped
4     quaternion = tf.transformations.quaternion_from_euler(roll, pitch, yaw) # Convertimos los angulos de ↩
          ↪ Eurler a cuaternion
5     arm_goal.header.frame_id = "base_footprint" # Definimos el encabezdo del mensaje
6     arm_goal.header.stamp = rospy.Time.now()
7     arm_goal.pose.position.x = x # Definimos la posicion deseada para arm_tool
8     arm_goal.pose.position.y = y
9     arm_goal.pose.position.z = z
10    arm_goal.pose.orientation.x=quaternion[0] # Definimos la orientacion deseada para arm_tool
11    arm_goal.pose.orientation.y=quaternion[1]
12    arm_goal.pose.orientation.z=quaternion[2]
13    arm_goal.pose.orientation.w = quaternion[3]
14    group= MoveGroupInterface("arm_torso", "base_footprint") # Creamos una interfaz de MoveIt definiendo↩
          ↪  el grupo a planear y la referencia
15    group.moveToPose(arm_goal,"arm_tool_link",wait=wait) # Mandamos el goal con el link que alcanzara ↩
          ↪ dicha posicion
16    print("Moved arm or unreachable") # Confirmamos por consola
```

Move It se encarga de planear el movimiento, intentando que el *arm tool link* alcance la posición. Lo hace moviendo el brazo y el torso si es necesario.
Si el punto es inalcanzable o si existen obstáculos que no permitan el acceso sin colisión el robot no moverá el brazo. Los obstáculos que pueden interferir en el movimiento del brazo se detectarán con las cámaras RGB-D situadas en la cabeza del robot.
Debido a las propiedades de la planificaión, es posible que aunque la situación inicial y final del robot y el entorno sean las mismas el movimiento varíe en diferentes ejecuciones.

4.2.3 Control de la cabeza

Para la cabeza no ha sido necesario ninguna planificación debido a que no debería ocurrir nunca la situación de que existan obstáculos en el área de movimiento de la cabeza y que estos obstáculos no hayan sido detectados por la cámara RGB-D y esquivados durante la navegación.
En vez de definir una posición de la cabeza, definimos un punto sobre el *frame* captado por la cámara. Este punto de dos dimensiones será el nuevo centro del frame una vez el robot ha movido la cabeza. Es por eso que el control de la cabeza está mejor explicado si decimos que se apunta la cabeza y no que simplemente se mueve. El código 4.3 muestra el código para apuntar la cabeza en 4 direcciones.

Código 4.3: Subprograma para apuntar la cabeza.

```
def move_head(self,direction):

    client=actionlib.SimpleActionClient('/head_controller/point_head_action',PointHeadAction) #Creamos ↩
        ↪ un cliente del tipo PointHead
    client.wait_for_server()# Esperamos que el servidor confirme que el cliente es correcto
    head_goal = PointHeadGoal() # Creamos un mensaje de tipo PointHead
    point=geometry_msgs.msg.PointStamped() # Creamos un mensaje de tipo Point
    point.header.stamp=rospy.Time.now() #Definimos el encabezado del Point
    point.header.frame_id="/xtion_rgb_optical_frame" #Frame captado por la camara RGB
    if direction=="down": # En funcion de la direccion a la que queremos mirar se define el punto
        point.point.x=0
        point.point.y=0.75
    elif direction=="up":
        point.point.x=0
        point.point.y=-0.75
    elif direction=="right":
        point.point.x=0.75
        point.point.y=0
    elif direction=="left":
        point.point.x=-0.75
        point.point.y=0
    else:
        point.point.x=0
        point.point.y=0
    point.point.z=1
    head_goal.target=point
    head_goal.pointing_axis.x=0 # Eje sobre el que queremos apuntar la cabeza
    head_goal.pointing_axis.y=0
    head_goal.pointing_axis.z=1
    head_goal.pointing_frame="/xtion_rgb_optical_frame" # Frame de la camara
    head_goal.min_duration.secs=1 #Parametros del movimiento
    head_goal.min_duration.nsecs=0
    head_goal.max_velocity=0.25
    client.send_goal(head_goal)
    wait=client.wait_for_result()
    print("Moved head")
```

4.2.4 Control del gripper

Como se observa en el código 4.4, el proceso para abrir o cerrar el *gripper* se realiza por un código más extenso que en los otros controles. Es extraño debido a que el movimiento es el más simple de los que hace el robot.
Esto es por que no existe una librería que simplifique el proceso y se deben definir hasta cuatro mensajes que van encapsulados dentro del mensaje final.

Código 4.4: Subprograma para abrir o cerrar el gripper.

```
def gripper(self,state):

    client=actionlib.SimpleActionClient('/gripper_controller/follow_joint_trajectory',↩
        ↪ FollowJointTrajectoryAction) # Creamos el cliente
    client.wait_for_server()# Esperamos a que el server confirme la creacion

    goal=FollowJointTrajectoryGoal()# Mensaje para definir el objetivo final
    traj=JointTrajectory() # Mensaje que contiene la trayectoria
    point=JointTrajectoryPoint() # Mensaje que contiene el punto deseado
    path_tol=JointTolerance() # Mensaje para la tolerancia del camino
    goal_tol=JointTolerance() # Mensaje para la tolerancia del objetivo

```

```
12  header=Header() # Definimos un encabezado para usar en el resto de mensajes
13  header.seq=1
14  header.stamp=rospy.Time.now()
15  header.frame_id=''
16
17  path_tol.name='path' # Definimos tolerancias bajas
18  path_tol.position=0.1
19  path_tol.velocity=0.1
20  path_tol.acceleration=0.1
21  goal.path_tolerance=[path_tol]
22
23  goal_tol.name='goal'
24  goal_tol.position=0.1
25  goal_tol.velocity=0.1
26  goal_tol.acceleration=0.1
27  goal.goal_tolerance=[goal_tol]
28
29  goal.goal_time_tolerance.secs=0.1
30  goal.goal_time_tolerance.nsecs=0
31
32
33  traj.header=header
34  traj.joint_names=["gripper_left_finger_joint", "gripper_right_finger_joint"] # Especificamos que ↩
        ↪ articulaciones deseamos mover
35
36  if state=="open": # Segun el parametro del usuario abrimos o cerramos
37      point.positions=[0.044,0.044]
38  elif state=="close":
39      point.positions=[0.01,0.01]
40  else:
41      print("ERROR")
42
43  point.velocities=[] # Informacion no necesaria para este movimiento
44  point.accelerations=[]
45  point.time_from_start.secs=1
46  point.time_from_start.nsecs=0
47
48  traj.points=[point] # Encapsulamos los mensajes
49  goal.trajectory=traj
50  client.send_goal(goal) # Enviamos el mensaje final
51  wait=client.wait_for_result()
52  print("Gripper done") #Confirmamos por consola
```

Las posiciones de las articulaciones del *gripper* han sido determinadas por prueba y error. Cuando está abierto sus posciones son las máximas que pueden alcanzar, mientras que para cerrar se han elegido en función del objeto que se ha de coger.

4.3 Tareas

4.3.1 *Pick and place*

Para realizar la acción de *pick and place* añadimos un objeto en forma de prisma rectangular, como se observa en la figura 4.10. El objeto se sitúa al lado de plato, actuando como un posible mango del plato. Por limitaciones físicas de Gazebo no se han podido juntar el mango con el plato y transportar también el plato. Para mejorar la simulación física se ha aumentado el rozamiento del prisma y se ha reducido su masa.

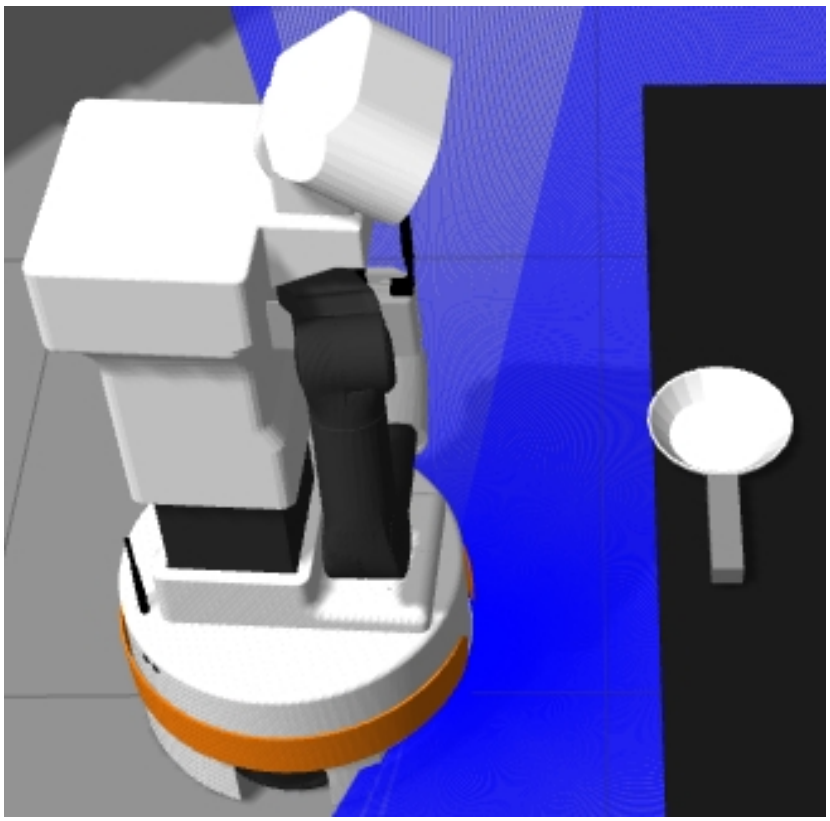


Figura 4.10: Prisma que se transporta, en su posición inicial. Fuente: Elaboración propia.

El proceso de navegar hacia el primer objeto, cogerlo, navegar hacia el segundo objeto y dejarlo se realiza mediante el código 4.5.

Código 4.5: Tarea de pick and place.

```
1
2# Go to plate 1
3tiago.move_base(1.6,−4,0)
4
5#Pick
6tiago.move_head(”down”)
7tiago.move_arm(0,−0.6,0.9,0,0.785,0,True)
8tiago.move_arm(0.6,−0.3,1.1,0,1.57,0,True)
9tiago.move_arm(0.6,−0.3,0.9,0,1.57,0,True)
10tiago.gripper(”close”)
11tiago.move_arm(0.6,−0.3,1,0,1.57,0,True)
12tiago.move_head(”up”)
13
14# Go to plate 2
15tiago.move_base(−0.75,−5,1.57)
16
17#Place
18tiago.move_head(”down”)
19tiago.move_arm(0.55,−0.2,1,0,1.57,0,True)
20tiago.move_arm(0.55,−0.2,0.95,0,1.57,0,True)
21tiago.gripper(”open”)
22tiago.move_arm(0.55,−0.2,1,0,1.57,0,True)
```

4.3.2 Alimentación

El proceso de alimentación se divide en dos partes. Primero colocamos la herramienta en la posición deseada y la movemos de manera específica para coger comida del plato. Después, activamos la parte de visión y procedemos a alimentar al paciente.
La tarea se ha programado con el planteamiento de que previamente se cambia la herramienta por una cuchara como aparece en la figura 4.11. Esto no ha podido simularse en Gazebo, ya que no se ha encontrado ninguna herramienta en forma de cuchara.

Figura 4.11: TiaGo con una herramienta de cuchara. *Fuente: IRI Team (UPC-CSIC)* (2018)

Esta tarea se realiza en el código 4.6.

Código 4.6: Tarea de alimentación del paciente.

```
1
2 #Ready spoon
3 tiago.move_base(-0.75,-5,0)
4 tiago.move_arm(0.55,-0.25,0.85,1.57,0,1.57,True)
5 tiago.move_base(-0.75,-5,1.57)
6
7 #Spoon up
8 tiago.move_arm(0.55,-0.2,0.85,1.57,0.3,1.57,True)
9 tiago.move_arm(0.55,-0.15,0.85,1.57,0,1.57,True)
10
11 # Ready to face
12 tiago.move_arm(0.5,0,0.9,1.57,0,0,True)
13
14 feed_time=0 # Limite de tiempo para alimentar
15 while True:
16     rospy.sleep(0.5)
17     print("feed_time",feed_time)
18     if tiago.face_pose_detected==True:
19         feed_register=tiago.feed
20         tiago.face_detection() #Activamos el detector de la boca abierta
21         if feed_register!=tiago.feed and tiago.feed==True: # Si el estado de la boca ha cambiado y esta ↩
               ↪ abierta, alimentamos
22             print("Proceed to feed")
```

```
            tiago.move_arm(tiago.pose_face.pose.position.x-0.25,tiago.pose_face.pose.position.y,tiago.↩
                ↪ pose_face.pose.position.z-0.1,1.57,0,0,False)

        elif feed_register!=tiago.feed and tiago.feed==False: # Si el estado de la boca ha cambiado y esta ↩
            ↪ cerrada, retiramos el brazo
            print("Backing up")
            tiago.move_arm(0.5,0,0.9,1.57,0,0,False)
            feed_time=0
            pass
        else:
            pass

        if feed_register==tiago.feed and tiago.feed==True: # Contamos el tiempo en que se esta alimentando
            feed_time=feed_time+1
        else:
            feed_time=0

        if feed_time>10: # Al finalizar la alimentacion volvemos al plato y repetimos el proceso
            #Spoon up
            tiago.move_arm(0.55,-0.2,0.85,1.57,0.3,1.57,True)
            tiago.move_arm(0.55,-0.15,0.85,1.57,0,1.57,True)
            # Ready to face
            tiago.move_arm(0.5,0,0.9,1.57,0,0,True)
        else:
            pass
    else:
        pass
```

4.4 Visión artifical

4.4.1 Reconocimineto de características faciales

Para la visión, como ya se ha comentado, combinamos OpenCV con Dlib. El código para la detección de una boca abierta se presenta en 4.7.

Código 4.7: Subprograma para la detección de una boca abierta.

```
def face_detection(self):
        #ret, img= self.cap.read()
        #self.win.clear_overlay()
        #self.win.set_image(img)
        img=dlib.load_rgb_image("face3.jpg")
        dets = self.detector(img, 1)
        if len(dets)==1:
            shape = self.predictor(img, dets[0])

            #self.win.add_overlay(shape)
            #Deteccion de la boca
            H_m=float(abs(shape.part(62).y-shape.part(66).y))
            L_m=float(abs(shape.part(60).x-shape.part(64).x))
            N_m=H_m/L_m #Calculo de la relacion de la apertura de la boca

            if N_m>0.3: # Limite por el que se determina que la boca esta abierta.
                print("Mouth is open")
                self.timer=self.timer+1 # Inicio de la cuenta
            else:
                print("Mouth is closed")
                self.timer=0

            if self.timer>4: # Momento en que activamos el movimiento de alimentacion
                self.feed=True
```

```
25
26              else:
27                  self.feed=False
28          else:
29              print("ERROR-Detected no or more than one face.")
```

Para la detección de la cara hemos obtenido un predictor del reconocimiento facial de Internet (Mallick, 2015). El cálculo entre puntos que se realiza para determinar se hace siguiendo el artículo Ji (2018). Como se ve en la imagen 4.12, la cara se divide en varios puntos característicos. En este trabajo, interesan:

- Los puntos 63 y 67 nos indican la distancia entre el labio superior y el labio inferior.
- Los puntos 61 y 65 nos indican la distancia entra las dos comisuras del labio.

Destacar que en el código los puntos tienen un índice de un número menor que los de la imagen. Esto se debe a que el vector del código tiene un punto en la posición 0. El valor de 0.3 para limitar la relación de las distancias de la boca y determinar que está abierta se ha obtenido tras prueba y error.

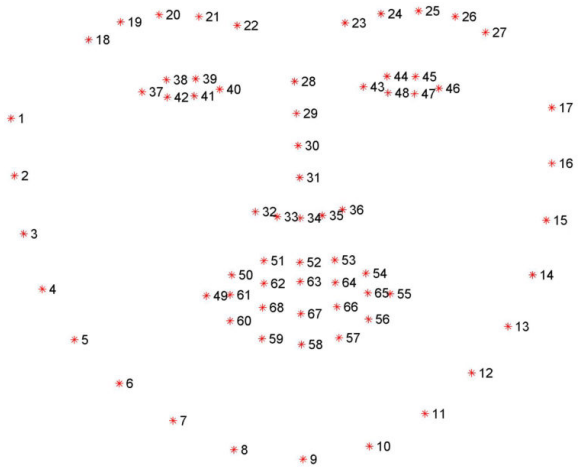


Figura 4.12: Los 68 puntos característicos en que se divide la cara. *Fuente: Pyimagesearch.com* (2019)

Para la obtención de la imagen podemos usar la cámara del ordenador o una foto guardada. Por motivos de limitación de computación, el funcionamiento es mucho más fluido cuando se usa una foto guardada. Por último, en el código 4.7 tenemos un *timer* para evitar falsos positivos. Si se detecta durante varios ciclos que la boca está abierta, procedemos a mandar la orden al brazo del movimiento de alimentación.

4.4.2 PAL Face Detector

TiaGo tiene un paquete por defecto para detectar caras a través de la cámara RGB-D (PAL, 2019b). Este paquete solo detecta la cara y la marca en una *bounding box* pero no da información sobre los puntos característicos ni su posición en el espacio.

Código 4.8: Código del suscriptor y su callback de PAL Face Detector.

```
1
2def ___init___(self):
3    rospy.Subscriber("/pal_face/faces", FaceDetections, self.callback_face) # Creamos un nodo suscrito al ↩
         ↪ topic de PAL face
4
5def callback_face(self,face): #Callback que se realiza solo si se ha detectado alguna cara
6    if len(face.faces)==1: #Comrpobamos el numero de caras detectadas
7        self.face_detected=True
8    else:
9        self.face_detected=False
```

En el código 4.8, vemos cómo creamos un *callback* que nos determinará el instante en que se detecte una cara.

4.4.3 PAL Texture Detector

Otro paquete de TiaGo que se ha aprovechado para este trabajo, se utiliza para detectar texturas desde la cámara RGB-D. Al lanzar su nodo le damos una imagen con la textura a detectar y el tamaño de la textura (su altura y su base). En la presente simulación esto será una imagen de la cara. A través de homografía se calcula la posción en el espacio del centro de la textura referenciada a la cámara. Si aplicamos la transformación necesaria podemos referenciar esta posición a la base y mover el brazo a dicha posición. En el código 4.9 aparece el *callback* suscrito al *topic* de este paquete (*PAL texture detector*, 2019).

Código 4.9: Código del suscriptor y el callback de PAL Texture Detector.

```
1rospy.Subscriber("/texture_detector/pose",PoseStamped,self.callback_poseface)
2def callback_poseface(self,pose): # Callback que se reliza cuando se obtiene la posicion de la cara
3    if self.face_pose_detected==False:
4        #Obtenemos la transformacion entre camara y base
5        transform = self.tf_buffer.lookup_transform("base_footprint","xtion_rgb_optical_frame",rospy.Time↩
             ↪ (0),rospy.Duration(1.0))
6        #Aplicamos la transformacion
7        self.pose_face = tf2_geometry_msgs.do_transform_pose(pose, transform)
8        self.face_pose_detected=True
9        print("Position saved", self.pose_face)
10   else:
11       pass
```

4.5 ROS Launch

4.5.1 Launch principal

El archivo *launch* es el ejecutable que lanza todos los nodos necesarios para la simulación. A continuación se listan los nodos utilizados. El código creado se ha obtenido partiendo de la base de un *launch* en el paquete tiago 2dnav gazebo (PAL, 2019a) añadiendo más nodos según las necesidades.

- Robot TiaGo.
- El brazo de TiaGo.
- El actuador de la muñeca.

- El sensor láser.
- La cámara RGB-D.
- El mundo de la simulación y la posición del robot en el mundo.
- El planificador global y local.
- El método de localización.
- El mapa del entorno para la navegación.
- RViz.
- *Pal Face Detector*.

Con todos estos nodos y sus correspondientes argumentos lanzamos nuestra simulación. El archivo *launch* completo está adjunto en los anexos.

4.5.2 Launch de Pal Texture detector

Por motivos de depuración y adaptación, se ha decidido que el nodo de este paquete sea lanzado en un *launch* aparte. En el código 4.10 aparece el launch de esta función.

Código 4.10: Ros Launch para PAL texture detector.

```
<?xml version="1.0" encoding="UTF-8"?>
<launch>

 <arg name="sample_image" value="(find pal_texture_detector)/objects/face1.png" />
 <arg name="rectified_image" default="/xtion/rgb/image_rect_color" />
 <arg name="camera_info" default="/xtion/rgb/camera_info" />

 <node name="texture_detector" pkg="pal_texture_detector" type="pal_texture_detector_node" args="(arg sample_image)" output="screen">
  <remap from="rectified_image" to="(arg rectified_image)"/>
  <remap from="camera_info" to="(arg camera_info)"/>
  <rosparam>
   enable_ratio_test: True
   enable_homography: True
   homography_iterations: 6
   estimate_pose: True
   object_width: 0.2 <!-- in meters. Only required if estimate_pose is True -->
   object_height: 0.2426 <!-- in meters. Only required if estimate_pose is True -->
   enable_visual_debug: True
  </rosparam>
 </node>
</launch>
```

5 Resultados

5.1 Resultados del modelado

5.1.1 Modelo de la habitación

Sobre el modelado de los entornos en Gazebo ya se han mostrado los resultados. La imágenes 5.1 y 5.2 muestran lo obtenido de las simulaciones en estos modelos con TiaGo en el entorno creado.

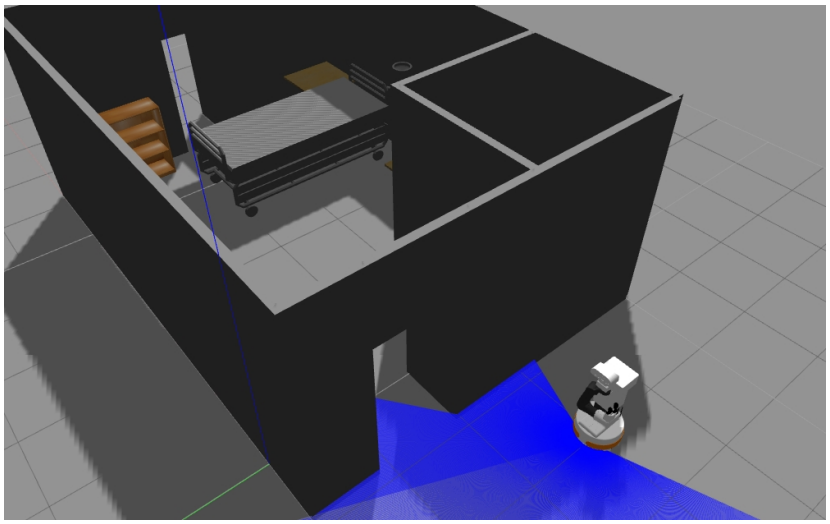


Figura 5.1: TiaGo fuera de la habitación, en su posición inicial. Fuente: Elaboración propia.

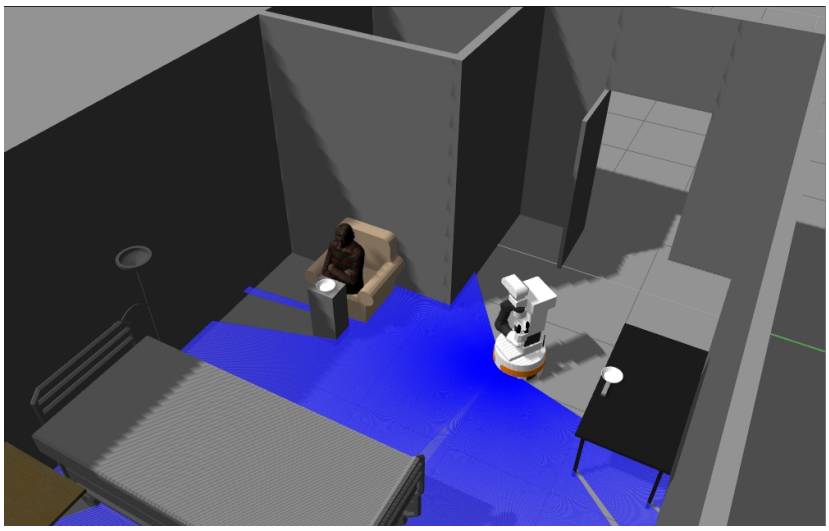


Figura 5.2: TiaGo dentro de la habitación. Fuente: Elaboración propia.

5.1.2 Mapa del entorno

Para poder navegar por el entorno modelado, debemos guardar un mapa previamente. Con los comandos que aparecen en 5.1 podemos lanzar la simulación, mover el robot por ella con el teclado y guardar el mapa. El mapa resultante de nuestro modelo aparece en la figura 5.3.

Código 5.1: Comandos para generar y guradar un mapa.

```
roslaunch tiago_2dnav_gazebo tiago_mapping_public.launch # Lanzamos la ↩
    ↪ simulacion
rosrun key_teleop key_teleop.py # Activamos el control por teclado
rosservice call /pal_map_manager/save_map "directory: ''" # Guardamos el mapa
```

Figura 5.3: Mapa de la habitación. Fuente: Elaboración propia.

5.2 Resultados del control

5.2.1 Resultados del control de la base

Para mover la base necesitamos definir un punto (x,y) y una orientación (yaw). Llamamos al subprograma con esta información, como aparece en el código 5.2

Código 5.2: Llamada al control de la base.

```
1 tiago.move_base(1.6,-4,0)
```

En la figura 5.4 vemos lo que aparece en RViz durante la navegación. Se observa el mapa que hemos guardado del entorno. La flecha roja representa el objetivo de la navegación y la línea azul que la une con el robot es la trayectoria planificada. También se representan las lecturas de los sensores que se utlizan para realizar el *costmap* y representar los objetos que no están representados en el mapa guardado.

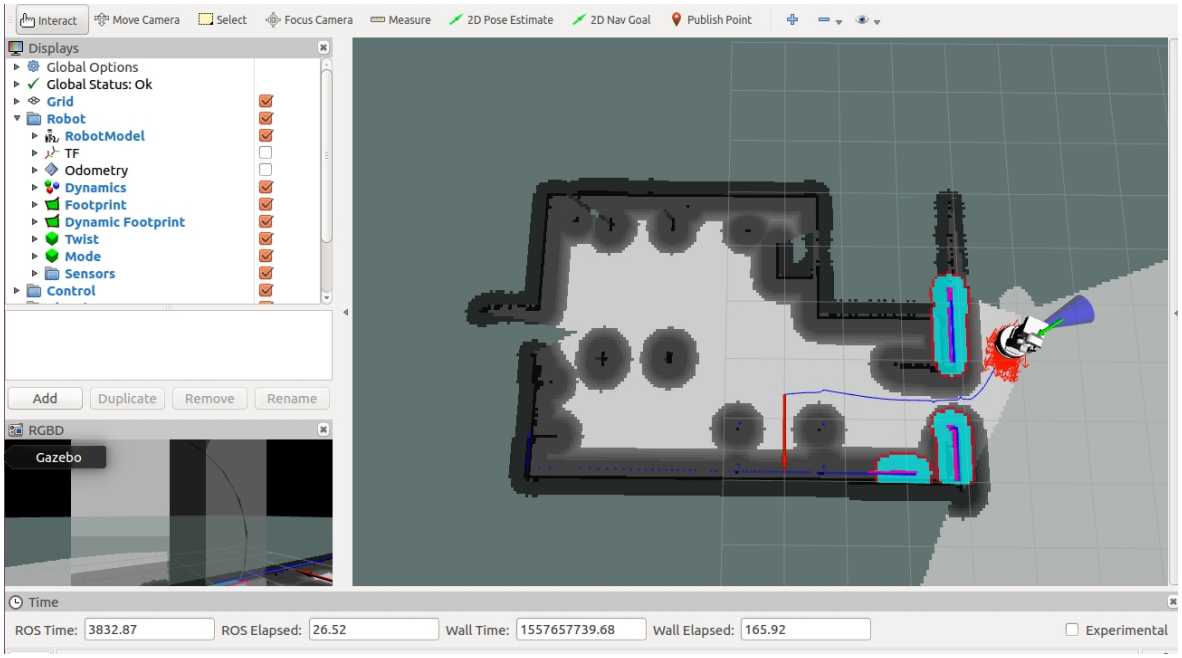


Figura 5.4: RViz durante la navegación. Fuente: Elaboración propia.

En las figuras 5.5 y 5.6 se puede ver la simulación en Gazebo del robot en tránsito.

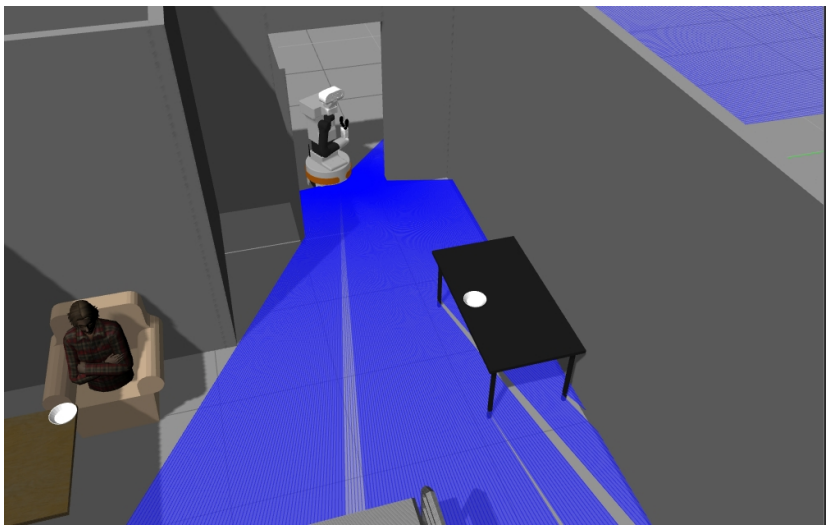


Figura 5.5: TiaGo navegando por el entorno mientras cruza la puerta. Fuente: Elaboración propia.

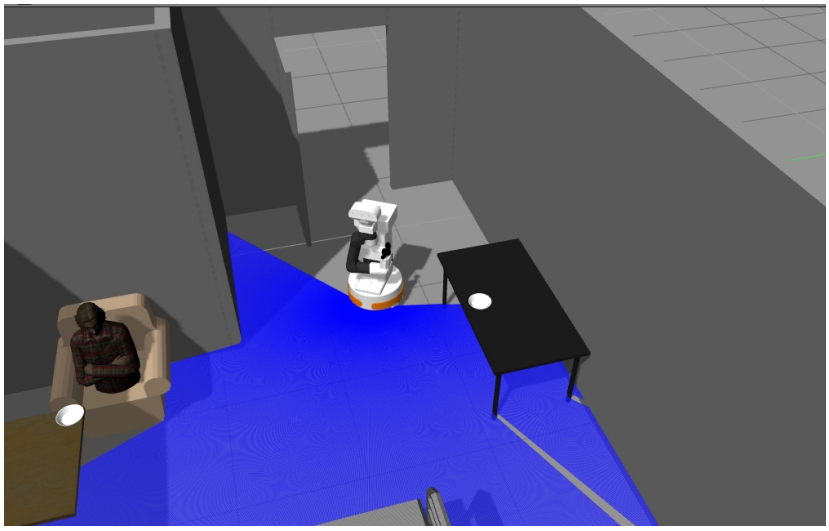


Figura 5.6: TiaGo navegando por el entorno mientras se acerca al objetivo. Fuente: Elaboración propia.

Por último, en la figura 5.7 vemos el robot en la posición final del objetivo.

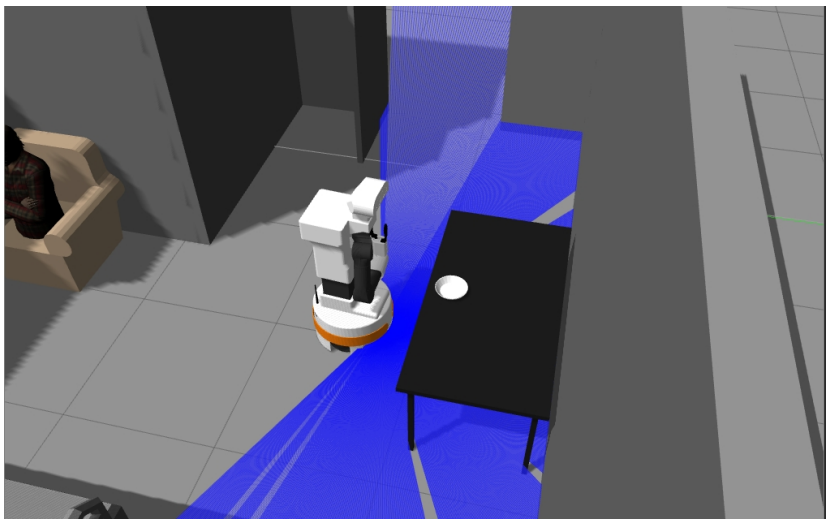


Figura 5.7: TiaGo poscionado en el objetivo tras la navegación. Fuente: Elaboración propia.

En la figura 5.8 vemos el entorno en el que se ha añadido un obstáculo no representado en el mapa. En la figura 5.9 vemos la planificación global previa a la detección de este obstáculo, que intersecta con el obstáculo.

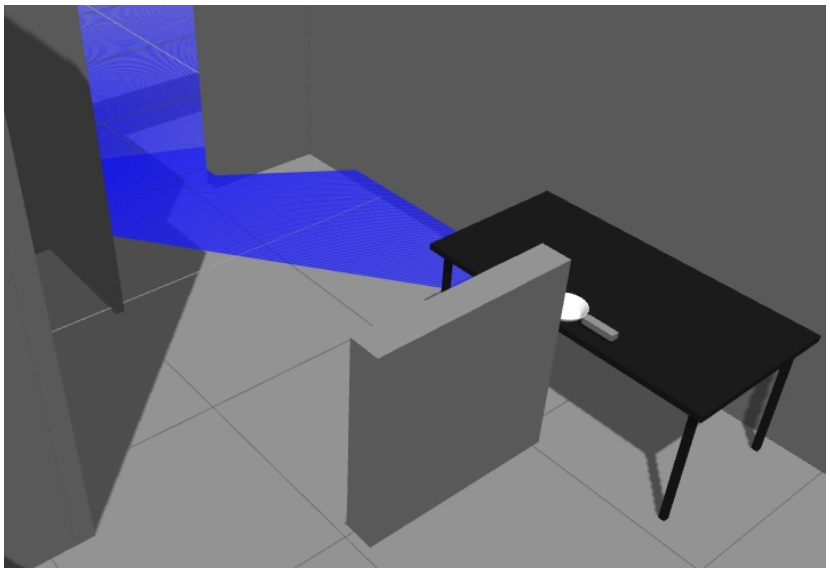


Figura 5.8: Entorno con un nuevo obstáculo. Fuente: Elaboración propia.

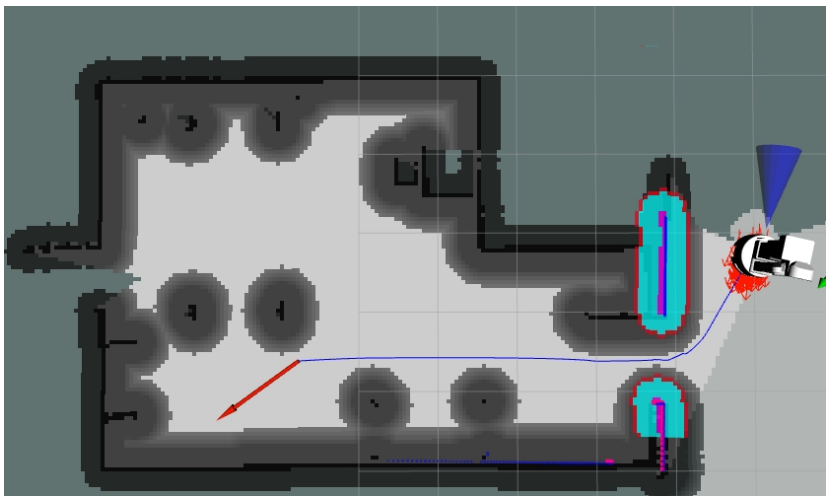


Figura 5.9: Planificación previa. Fuente: Elaboración propia.

Finalemente, cuando se detecta el obstáculo, el navegador planifica un trayectoria de nuevo acorde con la visión del obstáculo y alcanza el objetivo mediante la nueva trayectoria que se observa en la figura 5.10.

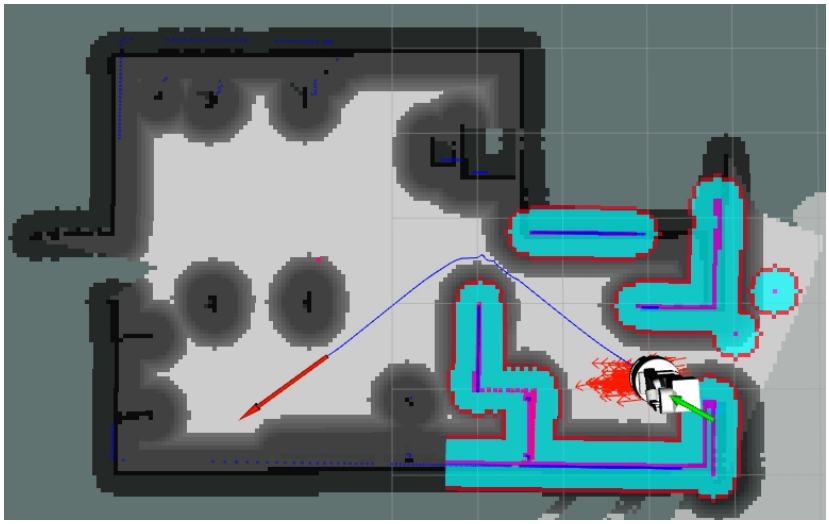


Figura 5.10: Planificación rectificada. Fuente: Elaboración propia.

5.2.2 Resultados del control del brazo y torso

El movimiento del brazo se realiza llamando al subprograma creado y con la información: (x,y,z) y (roll,pitch,yaw). Además, debemos específicar si queremos que la orden sea bloqueante o no del proceso. Un ejemplo de llamada aparece en el código 5.3.

Código 5.3: Llamada al control del brazo y torso.

```
1 tiago.move_arm(0,−0.6,0.9,1.57,0,−1.57,True)
```

Al enviar la orden Move It inicia la planificación y informa por consola del resultado (ver figura 5.11).

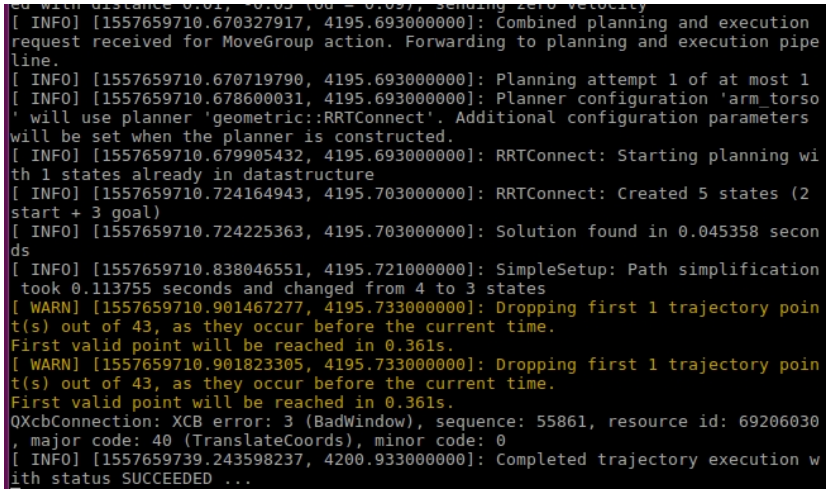


Figura 5.11: Consola durante el proceso de movimiento del brazo. Fuente: Elaboración propia.

En las figuras 5.12 y 5.13 aparece un movimiento de extensión del brazo.

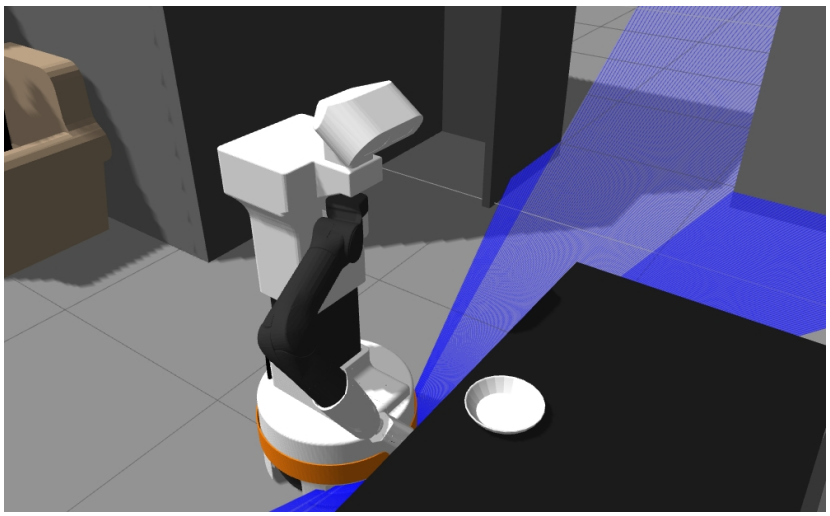


Figura 5.12: Inicio del movimiento del brazo. Fuente: Elaboración propia.

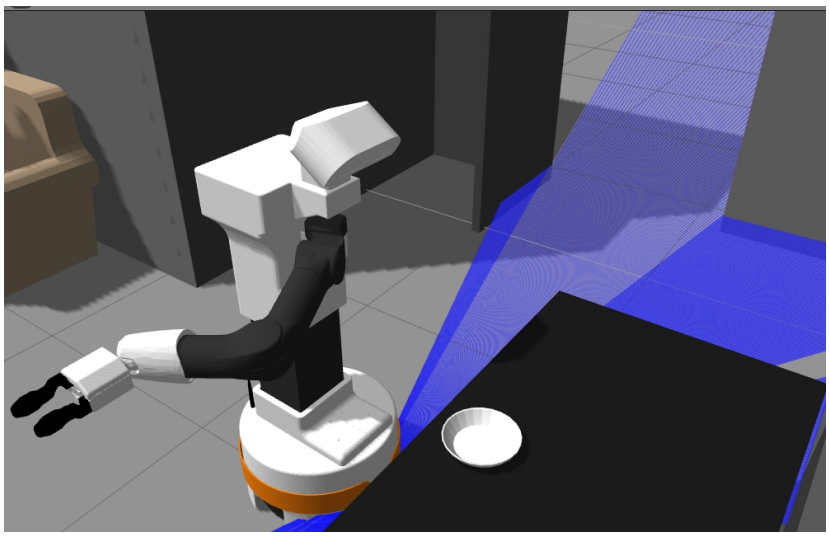


Figura 5.13: Brazo extendido. Fuente: Elaboración propia.

Move It, como ya se ha explicado, tiene integrado un planficador para evitar obstáculos. Un problema encontrado durante las pruebas ha sido que el movimiento a veces hacía que el brazo colisionará con el entorno. Esto era debido a que existían obstáculos que el robot no detectaba con la cámara RGB-D pero que estaban en la trayectoria planeada. Para evitar colisiones se han usado puntos intermedios entre planificaciones, dado que el entorno es estático y podemos predecir las posiciones que están libres de obstáculos.

5.2.3 Resultados del control de la cabeza

Para mover la cabeza de TiaGo debemos llamar al subprograma tal y como aparece en el código 5.4. Los parámetros de la función pueden ser:

- Up
- Down
- Left
- Right

Código 5.4: Llamada al control de la cabeza

```
1 tiago.move_head("down")
```

En las figuras 5.14 y 5.15 se observan dos posiciones de la cabeza de TiaGo resultado del control implementado.

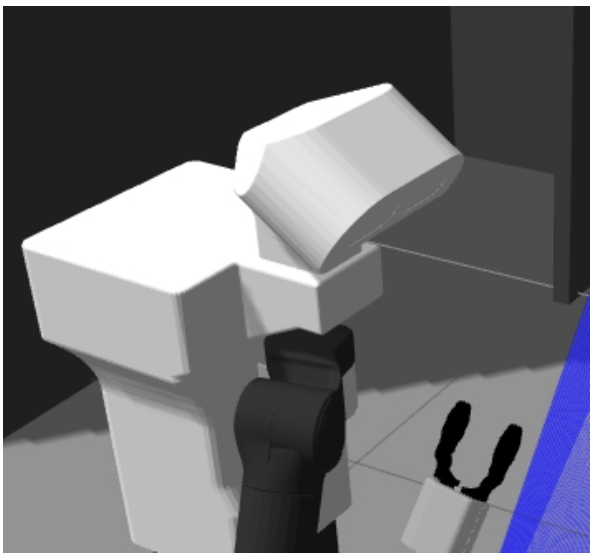


Figura 5.14: TiaGo mirando hacia abajo. Fuente: Elaboración propia.

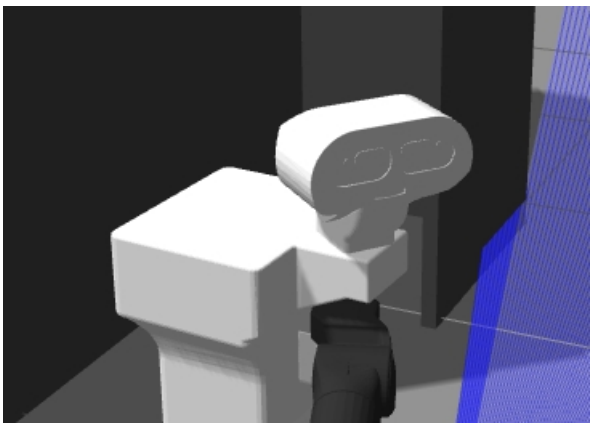


Figura 5.15: TiaGo mirando hacia la derecha. Fuente: Elaboración propia.

5.2.4 Resultados del control del gripper

La orden para mover el gripper se manda como aparece en el código 5.5.

Código 5.5: Llamada al control del gripper

```
1 tiago.gripper("close")
```

En la figura 5.16 se observa a TiaGo en la posición para cojer el objeto con la pinza abierta. En la figura 5.17 vemos el resultados del movimiento de cierre.

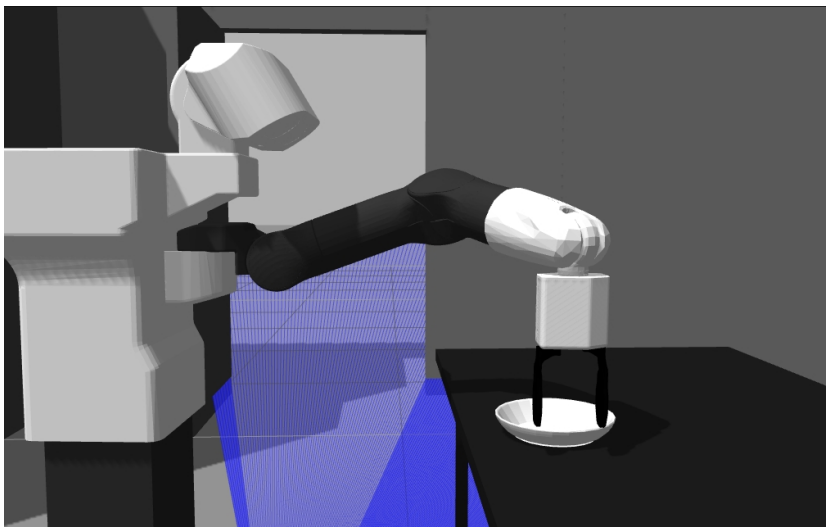


Figura 5.16: Gripper abierto. Fuente: Elaboración propia.

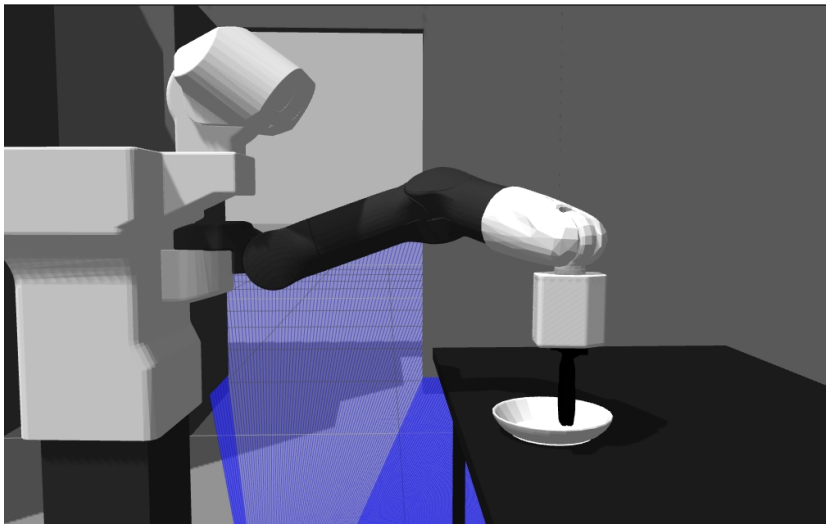


Figura 5.17: Gripper cerrado. Fuente: Elaboración propia.

5.3 Resultados de las tareas

5.3.1 Resutlados *pick and place*

En las figuras 5.18, 5.19 y 5.20 se representan algunos instantes del proceso de *pick and place*.

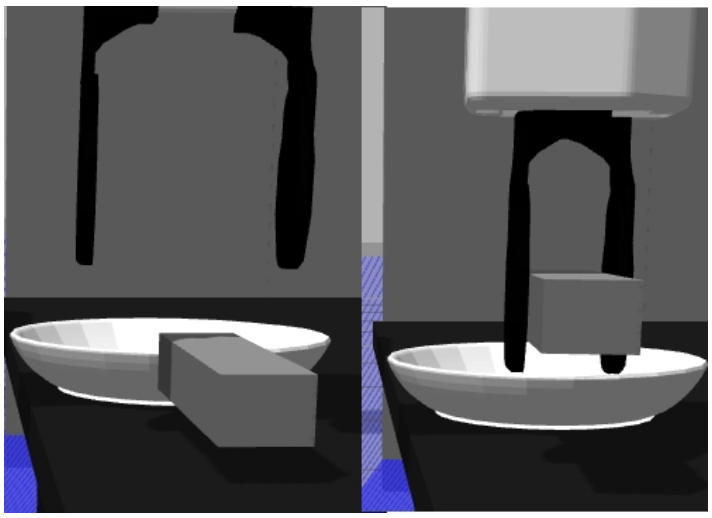


Figura 5.18: TiaGo cogiendo el objeto. Fuente: Elaboración propia.

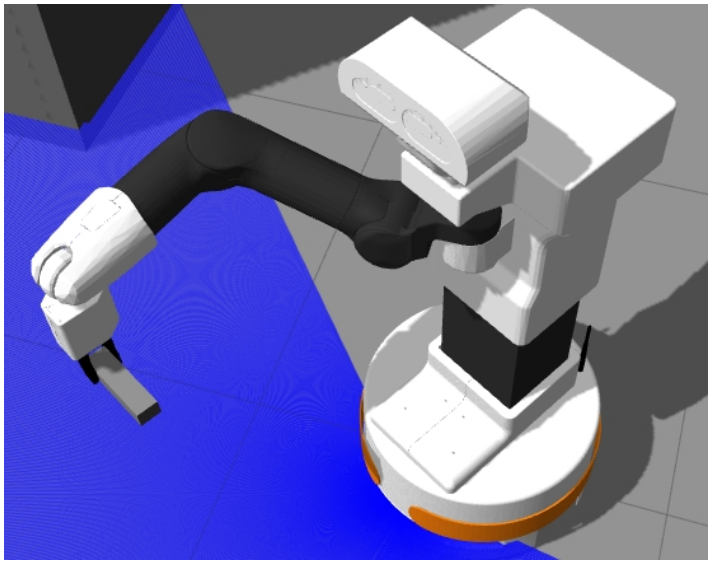


Figura 5.19: TiaGo transportando el objeto. Fuente: Elaboración propia.

Figura 5.20: TiaGo soltando el objeto. Fuente: Elaboración propia.

Para el correcto funcionamiento de esta tarea ha sido importante asegurar que las trayectorias del brazo sean las correctas y añadir puntos para el acercamiento del brazo en los momentos de coger y soltar.
En la figura 5.21 vemos el movimiento del *arm tool link* para coger el objeto. En la figura 5.22, aparece la trayectoria realizada para dejar el objeto. Estas figuras se interpretan de izquierda a derecha inicialmente y después de arriba a abajo.

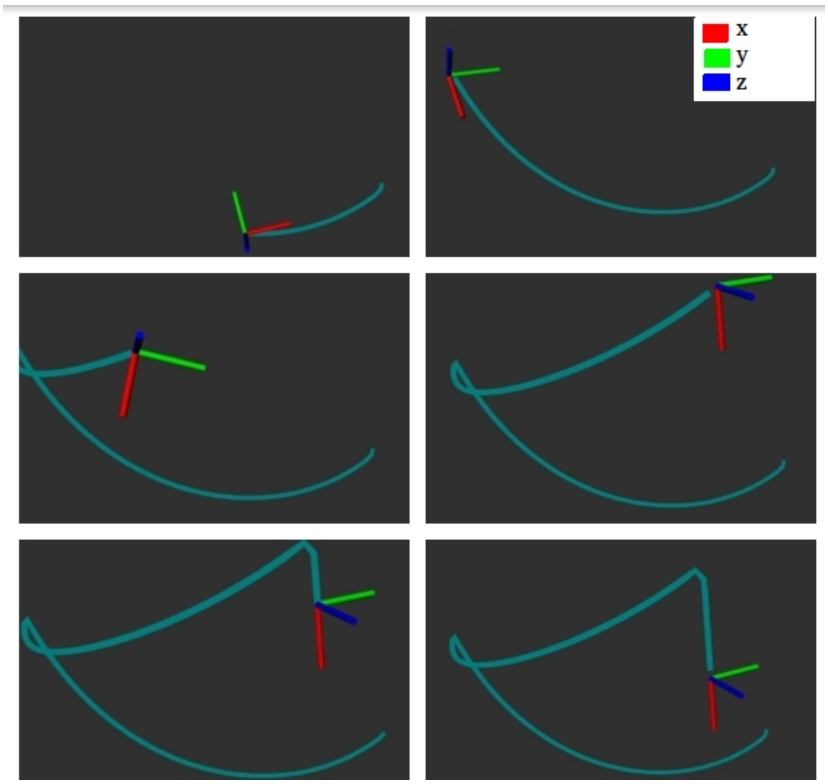


Figura 5.21: Trayectoria al coger el objeto. Fuente: Elaboración propia.

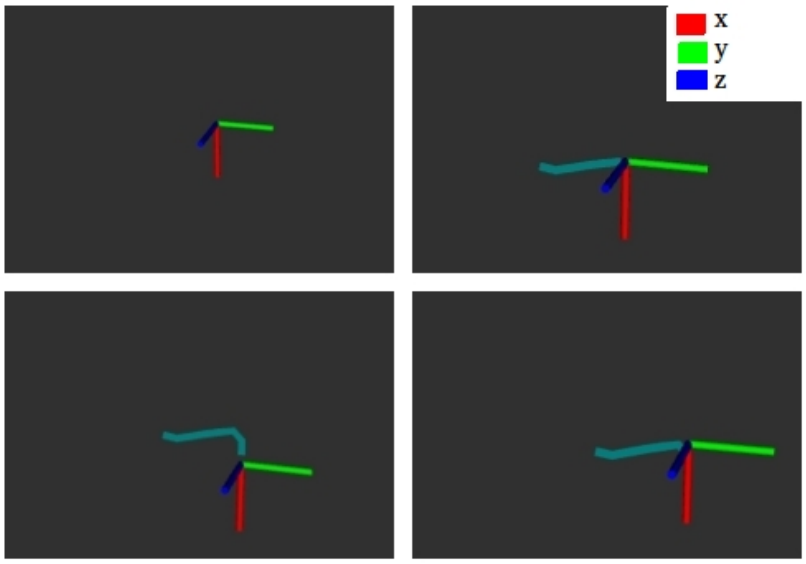


Figura 5.22: Trayectoria al dejar el objeto. Fuente: Elaboración propia.

Como se observa en las figuras 5.21 y 5.22, el movimiento de *pick* es mucho más complejo que el de *place*. Esto se debe a que para realizar el *place*, el brazo ya estaba colocado de la misma manera en que se termina el proceso de *pick*, facilitando la trayectoria.

5.3.2 Resultados alimentación

La manera en que se ejecuta esta tarea depende de los resultados de la visión, que se muestran en la sección siguiente. Algunos ejemplos de momentos durante este proceso se muestran en las figuras 5.23 y 5.24.

Figura 5.23: TiaGo cogiendo comida del plato. Fuente: Elaboración propia.

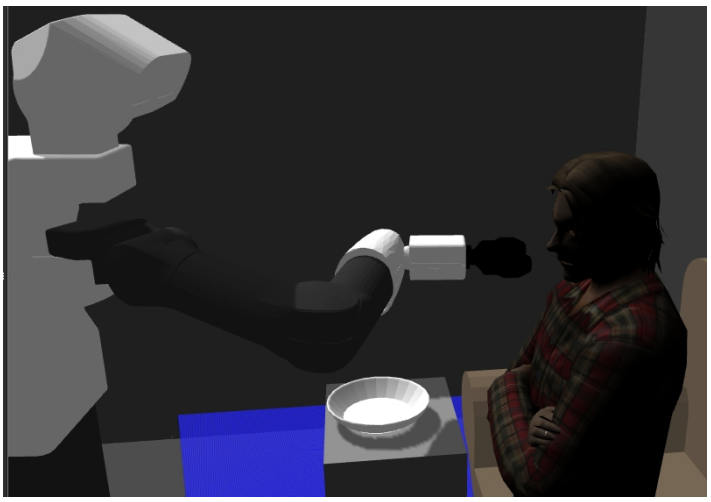


Figura 5.24: TiaGo alimentando al paciente. Fuente: Elaboración propia.

Para el funcionamiento de esta tarea sin complicaciones, los movimientos del brazo han de ser precisos, lo que se ha logrado añadiendo puntos intermedios entre trayectorias. Además, es importante mantener la orientación de la herramienta constante durante el proceso de alimentación a fin de no derramar la comida de la cuchara. Con las posiciones seleccionadas no ocurre esta situación.
En la figura 5.25 vemos la trayectoria que se realiza para alimentar al paciente. Se observa que es un trayectoria corta, pero en la que es importante la orientación para apuntar la cuchara hacia el plato o hacia el paciente. Cabe destacar que el último paso de esta trayectoria es variable, ya que depende del punto en que el se detecta la cara del paciente.

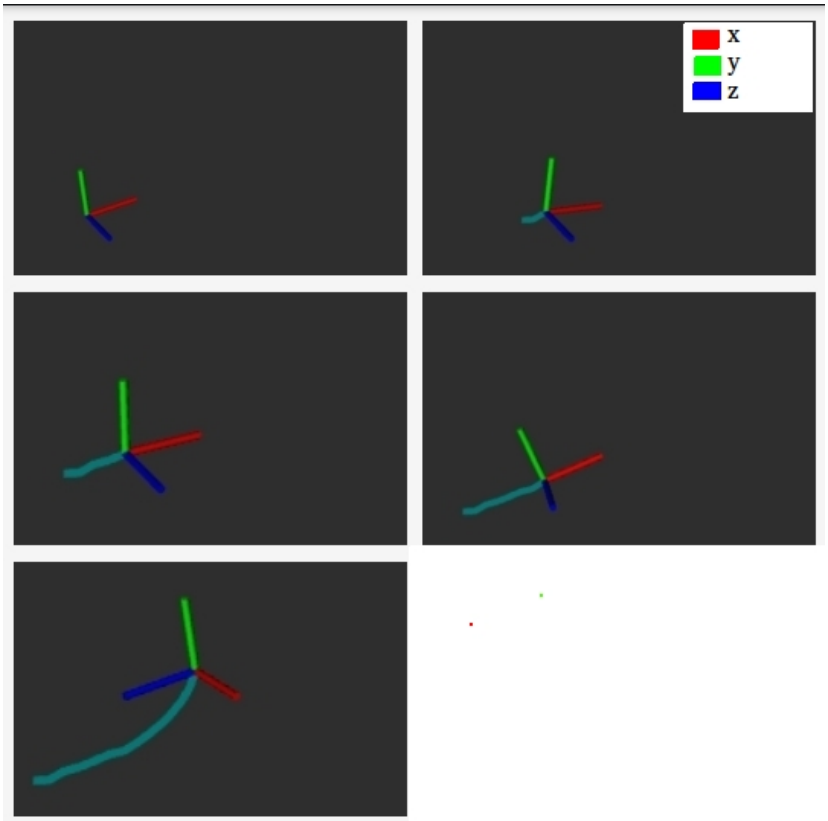


Figura 5.25: Trayectoria de alimentación al paciente. Fuente: Elaboración propia.

5.4 Resultados de la visión

5.4.1 Resultados de Pal Face Detector

Al ser un paquete de ROS, el detector facial de PAL está funcionando desde que se ejecuta el archivo *launch*. En la figura 5.26 vemos al robot situado frente al paciente. En un terminal podemos lanzar el comnado que aparece en 5.6 para ver lo que obtiene la cámara de TiaGO.

Código 5.6: Comando para visualizar la imagen para el reconocimiento facial.

```
1 rosrun image_view image_view image:=/pal_face/debug
```

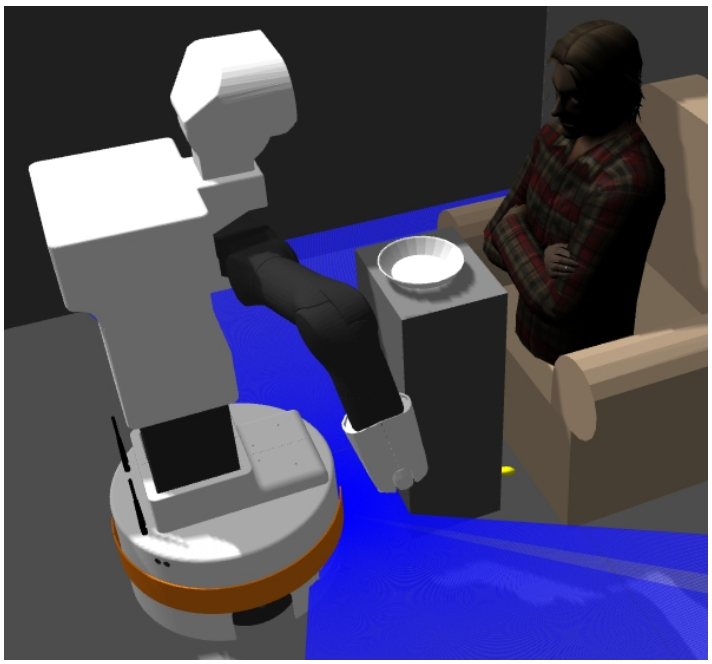


Figura 5.26: Situación de TiaGo frente al paciente. Fuente: Elaboración propia.

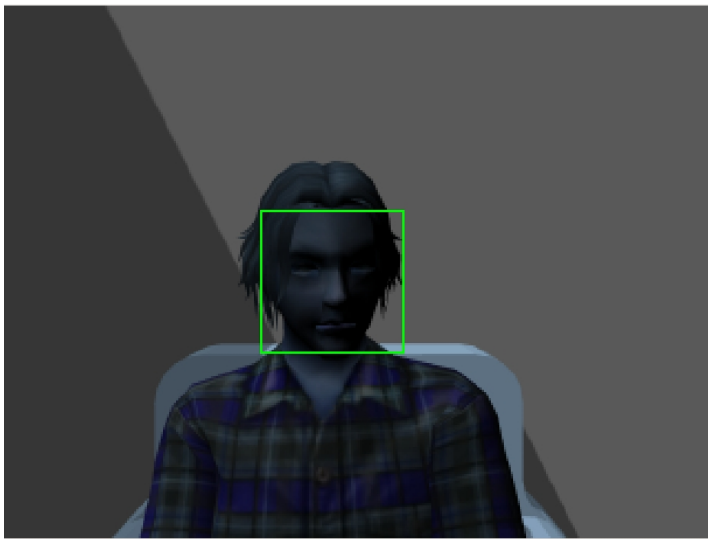


Figura 5.27: Imagen captada por la cámara RGB-D. Fuente: Elaboración propia.

En la figura 5.27 vemos la detección de la cara, marcada en un cuadro verde.

5.4.2 Resultados de Pal Texture Detector

Este paquete se ha elegido para que el movimiento final de alimentar al paciente se haga hacia una posición desconocida y se descubra la posición en función de la detección de la

cara. Para empezar su ejecución debemos pasarle al paquete una imagen a detectar, como la de la figura 5.28.

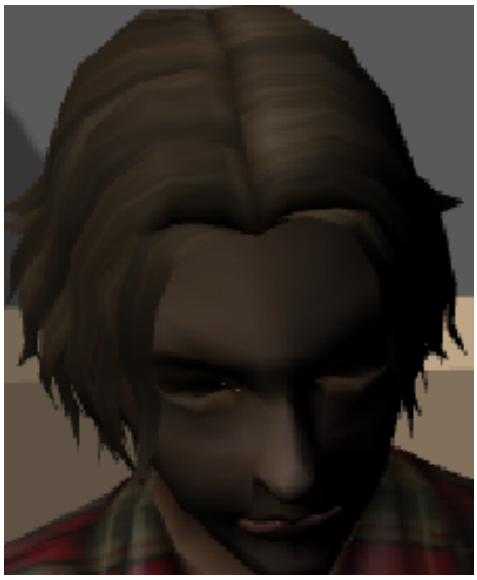


Figura 5.28: Imagen a detectar con Pal Texture Detector. Fuente: Elaboración propia.

La imagen a detectar es siempre de la misma cara, pero por las tolerancias de los controladores y los planificadores, TiaGo no se encontrará siempre en la misma posición exacta frente al paciente. Por eso el ángulo con el que ve la cara del paciente puede variar y es posible que el detector de texturas no sea capaz de reconocer nada.
Para solucinar esto, podemos ver la imagen que se publica en el *topic* dónde se hace el reconocimineto con el comando 5.7 y se abre una ventana como la figura 5.29. Desde esta imagen podemos extraer una nueva textura de la cara con el ángulo de visión actual y relanzar el nodo. También será importante redefinir el tamaño de la textura para que se respete la relación entre base y altura de la nueva imagen.

Código 5.7: Comando para visualizar la imagen para el reconocimiento de texturas.

```
1 rosrun image_view image_view image:=/texture_detector/debug
```

Figura 5.29: Imagen sobre la que se reconoceran las texturas. Fuente: Elaboración propia.

Cuando se detecta la textura correctamente aparecen las siguientes figuras en las que se representan los resultados. Para que ocurra esto se deben dar por lo menos 10 coincidencias claras entre la captura y la imagen original.

- Coincidencias entre la imagen original y la captura, ver figura 5.30.
- Nueva imagen rectificada según el ángulo de visión, ver figura 5.31.
- Coincidencias entre la imagen rectificada y la captura, ver figura 5.32.

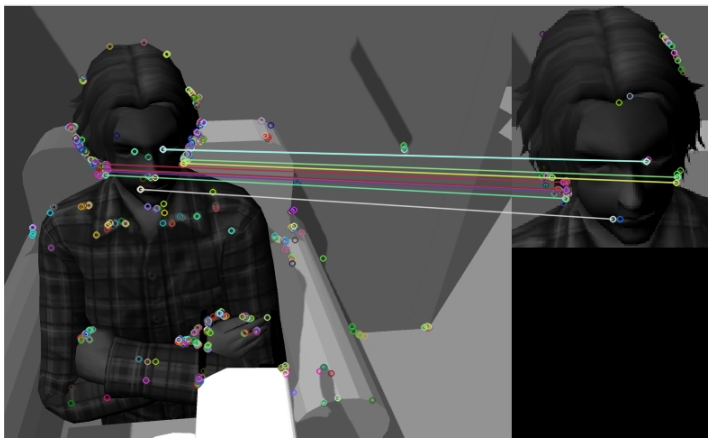


Figura 5.30: Coincidencias entre la imagen original y la captura. Fuente: Elaboración propia.

Figura 5.31: Nueva imagen rectificada según el ángulo de visión. Fuente: Elaboración propia.

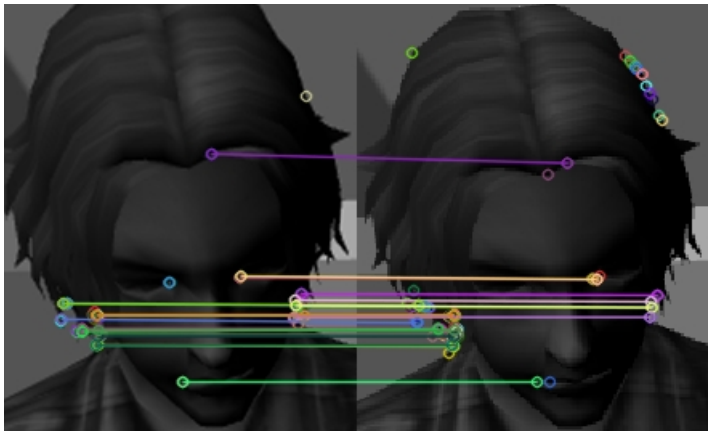


Figura 5.32: Coincidencias entre la imagen rectificada y la captura. Fuente: Elaboración propia.

Tambíen se puede observar la detección marcada en la imagen de la cámara, como aparece en la figura 5.33.

Figura 5.33: La textura detectada aparece marcada en verde. Fuente: Elaboración propia.

Finalmente, con la textura correctamente detectada se calcula mediante homografía la posición del centro de la textura en el espacio. En las figuras 5.34 y 5.35 se ve esta posición representada desde Rviz.

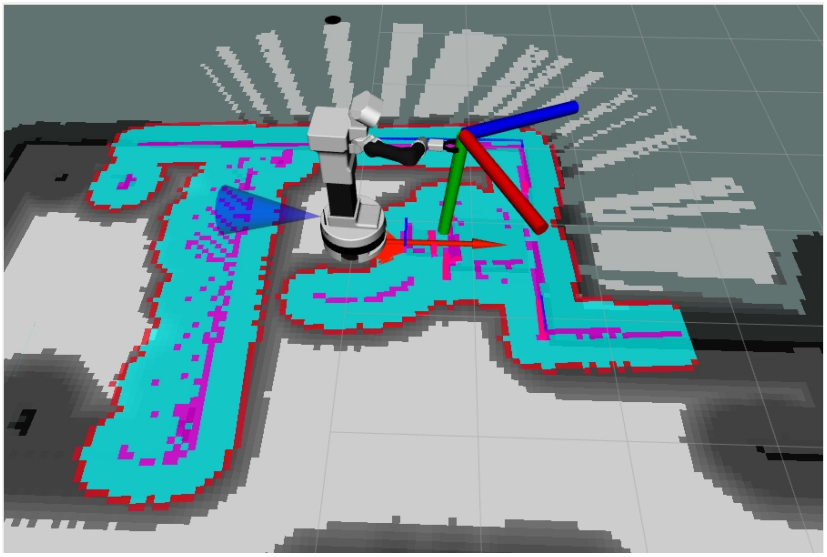


Figura 5.34: Posición de la cara en el espacio. Fuente: Elaboración propia.

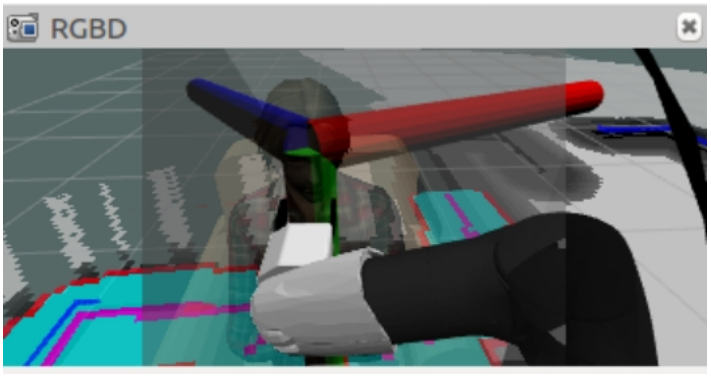


Figura 5.35: Posición de la cara vista desde la cámara. Fuente: Elaboración propia.

Para analizar la precisión de este método de obtención de la posición de la cara, se ha realizado un muestreo. Con TiaGo en frente del paciente, se ha dejado el nodo *PAL Texture Detector* en marcha hasta que se han obtenido 100 posiciones referenciadas a la cámara en las que se ha calculado el centro de la cara. La vista de la cámara durante este muestreo aparece en 5.36, esta imagen nos ayuda a situar mejor los resultados. Analizamos los datos obtenidos mediante *MatLab*.

Figura 5.36: Visión de la cámara durante el muestreo. Fuente: Elaboración propia.

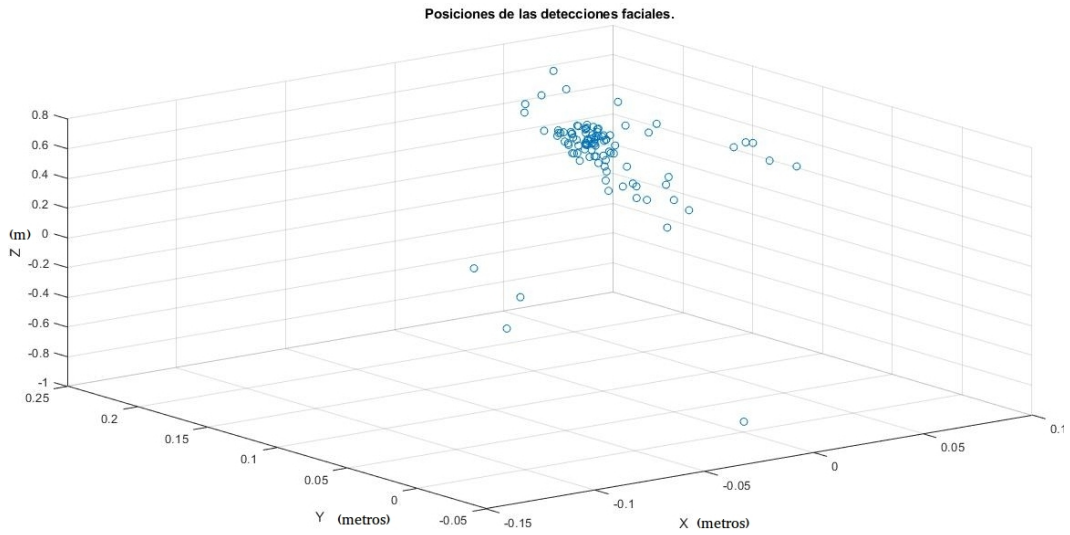


Figura 5.37: Posiciones de la cara. Fuente: Elaboración propia.

En la figura 5.37, vemos los puntos en el espacio de los 100 valores obtenidos. Se observa la presencia de varios *outliers* que se han producido en situaciones en las que existe error de reconocimiento. Mediante el análisis de esta figura, podemos afirmar que más del 80 % de los resultados se encuentran a menos de 0.1 metros de diferencia en las tres coordenadas.

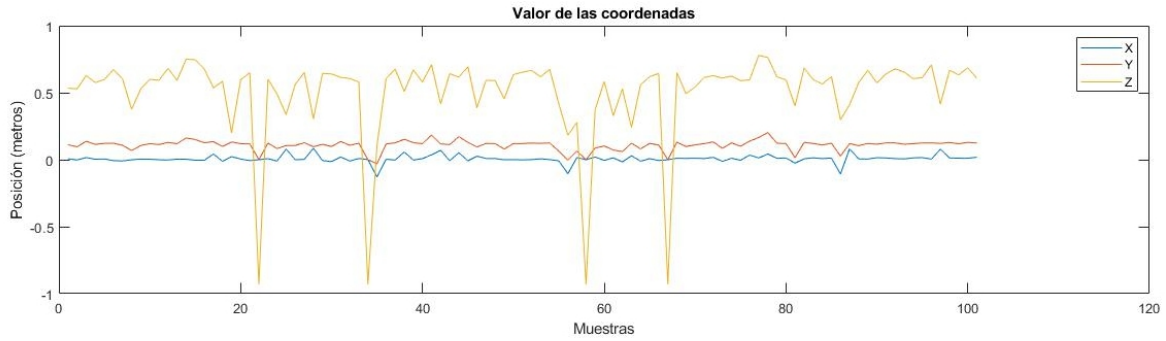


Figura 5.38: Coordenadas de las posiciones de la cara. Fuente: Elaboración propia.

La figura 5.38 muestra los valores de cada coordenada en función del número de muestras. Con este representación podemos ver claramente en que momentos se han producido errores. Son destacables algunos errores producidos en el eje Z por su magnitud. Esto es debido a que este eje representa la profundidad de la cámara y, aunque se trate de una cámara RGB-D, esta coordenada siempre es más susceptible a errores.

	Mediana	Media	Desviación estándar
X	0.059	0.067	0.0297
Y	0.1197	0.1087	0.039
Z	0.5989	0.5071	0.3204

Figura 5.39: Datos del muestreo. Fuente: Elaboración propia.

Por último, podemos extraer varios valores interesantes sobre los datos obtenidos. Estos valores aparecen en la figura 5.39. Como ya se ha mencionado, el eje Z que representa la profundidad es en el que se producen más errores y por eso su desviación típica es mayor. En cambio, los valores de X y de Y presentan menos desviación estándar, con valores bajos.
El hecho de que solo podamos obtener la posición de la cara a través de este método hace que no sea posible calcular el error, ya que no sabemos la posición exacta que se debería obtener. Lo que sí podemos analizar es el valor medio de la posición obtenida y si tiene relación con la simulación. Estando el robot justo delante del paciente y a su misma altura, es lógico que en X y Y se obtengan valores bajos, no más de 0.15 metros. También es correcto que la distancia entre el robot y el paciente sea de 0.5 metros aproximadamente, que es lo que obtenemos en el eje Z.

5.4.3 Resultados del reconocimineto de características faciales

Para obetener las imágenes en las que se reconocen las características faciales podemos usar la webcam de un ordenador o imágenes guardadas. Por limitaciones del simulador no se ha podido extraer la imagen de la cámara RGB-D de TiaGo para ejecutar el reconocimiento sobre ella. Como aparece en el apartado anterior, si hemos podido ver su contenido pero para hacer el reconocimiento de características faciales es necesario guardar la imagen. Además, no se han encontrado modelos en Gazebo de humanos con la boca abierta, lo que habría limitado las pruebas de la presente simulación.
En la figura 5.40 se observa un resultado en el que se detecta correctamente una boca abierta. Las líneas azules son las que resultan de unir los 68 puntos característicos mencionados anteriormente. Las fotos para realizar estas pruebas han sido obtenidas en *Face Stock Photos* (2019).

Figura 5.40: Boca abierta reconocida correctamente. Fuente: Elaboración propia.

En las figura 5.41 aparece un ejemplo de una persona con la boca cerrada. En este caso, el programa lo detecta correctamente y no daría la orden para alimentar al paciente.

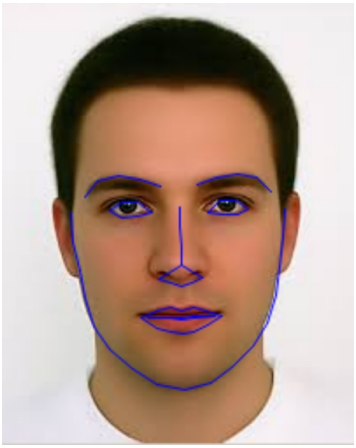


Figura 5.41: Boca cerrada reconocida correctamente. Fuente: Elaboración propia.

Las figuras 5.42 y 5.43 son ejemplos de situaciones en las que la detección podría dar problemas. Para que no sea así se ha ajustado el valor de la relación de apertura de la boca a partir del cual el programa afirma que la boca está abierta. Después de múltiples pruebas, estos ejemplos se reconocen correctamente como bocas cerradas.

Figura 5.42: Expresión ambigua 1. Fuente: Elaboración propia.

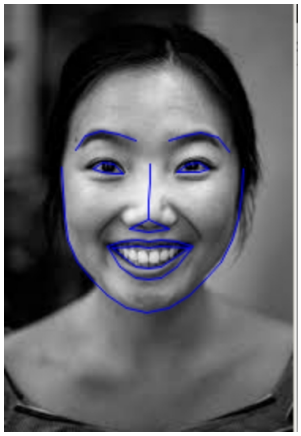


Figura 5.43: Expresión ambigua 2. Fuente: Elaboración propia.

6 Conclusiones

6.1 Desarrollo

El desarrollo de este proyecto se ha dividido en tres fases principales, de las cuales destacan las siguientes conclusiones.

- Estudio: La primera fase ha consistido en el estudio de los programas elegidos para la implementación del trabajo. Se ha recopilado toda la información necesaria de TiaGo, de Ros y de Gazebo. Además, se ha investigado sobre los diferentes métodos posibles para resolver la parte de visión artificial de este trabajo. En esta fase se incluye también la visita al Hospital de San Vicente del Raspeig para recoger datos y conocer las necesidades del personal hospitalario. Esta fase ha transcurrido durante los primeros meses del curso 18/19, aunque la obtención de nueva información útil fue una constante durante todo el proyecto.

- Implementación: La parte principal del trabajo fue la implementación del programa y el diseño de los entornos. Se han probado varios controladores diferentes hasta hallar con los elegidos y para cada controlador se ha investigado la mejor manera de utilzar sus funcionalidades.
  El mayor problema del trabajo fue con la parte de visión. El código que detecta cuando la boca está abierta tardó aproximadamente un mes en desarrollarse, no tanto por la implementación final en sí, sino por multitud de otras posibles soluciones que fueron probadas pero no satifactorias.
  Posteriormente, la parte de visión dentro del simulador también conllevo dificultades. El paquete *PAL Face Detector* no es capaz de proporcionar la posición de la cara en el espacio. Se ha mantenido en el proyecto para tener información adicional de lo que percibe el robot, pero el paquete que se usa principalmente es *PAL Texture Detector*.
  Asimismo, el paquete *PAL Texture Detector* tiene una alta variabilidad en sus resultados y no siempre reconoce fiablemente la cara. Se ha podido resolver esto ejecutando varias veces este nodo con distinitas texturas a reconocer: imágenes de la cara desde distintos ángulos. Debido a estos contratiempos esta fase del proyecto ha durado aproximadamente 4 meses.

- Test: La última fase del proyecto ha consistido en probar todo lo implementado y asegurar que esto funciona correctamente. Durante esta fase se han hecho las modificaciones necesarias a la implementación, pero también a los paquetes de TiaGo. Se han afinado las tolerancias de sus planificadores y aumentado la frequencia de publicación de algunos *topics*. También se han ajustado las posiciones que el robot alcanza, tanto de la base como del brazo. Durante esta fase el problema de la alta computación se mostró de manera que no se podía realizar la simulación y obtener una imagen en vivo de

la cámara. Esto limitó las pruebas realizadas al uso de fotos gurdadas. Esta fase ha transcurrido en 2 meses y se ha combinado con la redacción de esta memoria.

6.2 Resultados

Los resultados obtenidos demuestran que se han cumplido los objetivos marcados. El control del robot es completo, usando todas sus articulaciones. Los movimientos de la base y el brazo están planificados para que sean eficientes y para evitar colisiones. El objetivo de la alimentación del paciente se ha conseguido, tanto con los movimientos del brazo pertinentes como mediante la visión artificial. La visión implementada es capaz de detectar la posición de la cara del paciente y el estado de la boca. El movimiento del brazo a la hora de alimentar al paciente depende de la información recibida mediante la visión artificial.
Cabe destacar que las limitaciones del simulador han hecho que estos resultados hayan sido lentos de obtener. La comunicación podría ser más fluida y eso disminuiría los tiempos de reacción, pero el hecho de que funcionen tantos procesos a la vez sobre un portátil de gama media ha limitado este factor.

6.3 Posibles ampliaciones

A continuación se mencionan varias posibles vías por las que podría continuar este trabajo.

- Mejorar el tiempo de ejecución: Python ha sido elegido por su facilidad de uso y rapidez para hacer pruebas, pero tiene el inconveniente de que al ser un lenguaje interpretado puede retrasar el proceso. Ahora que ya sabemos como realizar el proyecto se podría traducir a C++ para mejorar la velocidad de ejecución.

- Probar sobre el robot real: El siguiente paso lógico de este proyecto sería llevar la aplicación al robot TiaGo en la realidad. Además el hecho de que se hayan diseñado los entornos del Hospital de San Vicente del Raspeig facilitaría las pruebas en ese mismo entorno real. Para llevar esto a cabo, se puede predecir que se deberían ajustar múltiples parámetros tanto del control como de la visión.

- Creación de una GUI: Se podría crear una interfaz gráfica para el control del robot que usase los controladores implementados. Mediante un panel se podrían definir los puntos a alcanzar y movimientos a realizar, facilitando más el uso del programa sin tener que usar los comandos diseñados.

- Mejorar el reconocimiento facial: Mientras que la detección de cuándo la boca está abierta funciona correctamente, se podría mejorar el reconocimiento de la posición de la cara. El paquete que se usa ahora está diseñado para texturas planas y por eso el resultado varía en función del ángulo con el que se mira la cara. Se podría diseñar un reconocimiento que tenga en cuenta las tres dimensiones del modelo.

Bibliografía

Ayri. (2016). Premio innovacion. Descargado de https://ayri11.com/notas-de-prensa/pal-robotics/pal-robotics-ganadores-del-premio-innovacion-en-el-foro-ayri11-con-el-robot-tiago/

Baker, M. (2005). *Automated street crossing for assistive robots.* Descargado de https://ieeexplore.ieee.org/abstract/document/1501081

blenderpl. (2019). Descargado de https://free3d.com/3d-model/hospital-bed-85989.html

Booth, K. E. (2017). *Robots in retirement homes: Person search and task planning for a group of residents by a team of assistive robots.* Descargado de https://ieeexplore.ieee.org/abstract/document/8268000

Broekens, J. (2009). *Assistive social robots in elderly care: a review.* Descargado de http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.714.6939&rep=rep1&type=pdf

Casas, J. (2018). *Social assistive robot for cardiac rehabilitation: A pilot study with patients with angioplasty.* Descargado de https://dl.acm.org/citation.cfm?id=3177052

European robotics league. (2018). Descargado de http://blog.pal-robotics.com/a-rewarding-european-robotics-league-tournament-at-pal-robotics/

Face stock photos. (2019). Descargado de https://www.pexels.com/search/face/

Fkerong Gai, J. Z. (2016). *Design on a desktop meal-assistance robot.* Descargado de https://www.atlantis-press.com/proceedings/icseee-16/25867173

Fuente: Asus xtion. (2019). Descargado de https://www.asus.com/3D-Sensor/Xtion_PRO/

Fuente: Atlantis press. (2016). Descargado de https://www.atlantis-press.com/proceedings/icseee-16/25867173

Fuente : datasheet de tiago. (2019). Descargado de http://tiago.pal-robotics.com/wp-content/uploads/2018/03/Datasheet_TIAGo-Hardware-Software.pdf

Fuente:diario información. (2019). Descargado de https://www.diarioinformacion.com/alacanti/2008/05/28/servicio-oftalmologia-san-vicente-da-citas-demora-9-meses/759596.html

Fuente: Iri team (upc-csic). (2018). Descargado de https://www.youtube.com/watch?v=dM9DoZ2z6To&feature=youtu.be

Fuente : modelos de tiago. (2019). Descargado de http://tiago.pal-robotics.com/wp-content/uploads/2018/03/Datasheet_TIAGo-Hardware-Software.pdf

Fuente: Move base ros wiki. (2019). Descargado de http://wiki.ros.org/move_base

Fuente: Moveit.ros. (2019). Descargado de https://moveit.ros.org/documentation/concepts/

Fuente: Obi. (2019). Descargado de https://meetobi.com/tech-specs//

Fuente: Pyimagesearch.com. (2019). Descargado de https://www.pyimagesearch.com/2017/04/03/facial-landmarks-dlib-opencv-python/

Fuente:reem-c robot. (2019). Descargado de https://www1.ireviews.com/pal-robotics-reem-c-reivew

Fuente:ros wiki. (2019). Descargado de http://wiki.ros.org/ROS/Concepts

Fuente: Secom. (2019). Descargado de https://www.secom.co.jp/english/myspoon/

Fuente: Talos. (2019). Descargado de http://blog.pal-robotics.com/the-humanoid-robot-talos-has-been-officially-presented-at-laas-cnrs-toulouse-france/

Fuente: Tecnalia. (2019). Descargado de https://www.tecnalia.com/en/robotics/medical-robotics/surgical-robotics/surgical-robotics.htm

Fuente:tiago base. (2019). Descargado de https://www.roboticgizmos.com/tiago-base-modular-robot/

Fuente:tiago con gripper. (2019). Descargado de https://www.hisparob.es/ultimos-dias-aplica-para-alquilar-un-robot-tiago-y-compite-en-el-sciroc-challenge/

Fuente:tiago con hey 5. (2019). Descargado de https://robots.ieee.org/robots/tiago/

Fuente:tiago cuerpo extendido. (2019). Descargado de http://blog.pal-robotics.com/tiago-simulation-tutorial-1-how-to-control-the-cobot/

Fuente:tiago robot. (2019). Descargado de https://robots.nu/en/robot/tiago-service-robot

Gazebo. (2004). *Gazebo.* Descargado de http://gazebosim.org/

Intel. (2000). *Opencv.* Descargado de https://opencv.org/

Irf. (2019). Descargado de https://ifr.org/

irobot. (2019). Descargado de https://www.irobot.es/

Iros mobile manipulation hackathon. (2018). Descargado de http://blog.pal-robotics.com/tiago-robot-iros-mobile-manipulation-hackathon/

Iso8373. (2012). Descargado de https://www.iso.org/standard/75539.html

Ji, Y. (2018). Eye and mouth state detection algorithm based on contour feature extraction. Descargado de https://www.spiedigitallibrary.org/journals/journal-of-electronic-imaging/volume-27/issue-05/051205/Eye-and-mouth-state-detection-algorithm-based-on-contour-feature/10.1117/1.JEI.27.5.051205.full?SSO=1

King, D. E. (2002). *dlib.* Descargado de github.com/davisking/dlib

Mallick, S. (2015). *Facial landmark detection.* Descargado de https://www.learnopencv.com/facial-landmark-detection/

My spoon. (2019). Descargado de https://www.secom.co.jp/english/myspoon/

Naotunna, I. (2015). *Meal assistance robots: A review on current status, challenges and future directions.* Descargado de https://ieeexplore.ieee.org/abstract/document/7404980

Nozaki, T. (2016). *Development of meal assistance device for patients with spinal cord injury.* Descargado de https://ieeexplore.ieee.org/abstract/document/7496381

Obi. (2019). Descargado de https://meetobi.com/tech-specs/

PAL. (2019a). *Localization and path planning.* Descargado de http://wiki.ros.org/Robots/TIAGo/Tutorials/Navigation/Localization

PAL. (2019b). *Pal face detector.* Descargado de http://wiki.ros.org/Robots/TIAGo/Tutorials/FaceDetection

PAL. (2019c). *Tiago robot.* Descargado de http://tiago.pal-robotics.com/

Pal texture detector. (2019). Descargado de http://wiki.ros.org/Robots/TIAGo/Tutorials/HomographyEstimation

Perera, C. J. (2017). *Eeg-controlled meal assistance robot with camera-based automatic mouth position tracking and mouth open detection.* Descargado de https://ieeexplore.ieee.org/abstract/document/7989208

Robocup@home 2018. (2018). Descargado de http://blog.pal-robotics.com/homer-team-wins-robocup-home-2018/

Ros visualizer. (2019). Descargado de https://wiki.ros.org/rviz

Smp. (2019). Descargado de https://smprobotics.com/products_autonomous_ugv/security-patrol-robot/

Song, W.-K. (2012). Novel assistive robot for self-feeding. Descargado de https://www.researchgate.net/publication/221923712_Novel_Assistive_Robot_for_Self-Feeding

Stahl, B. C. (2016). Ethics of healthcare robotics : Towards responsible research and innovation. Descargado de https://www.sciencedirect.com/science/article/pii/S0921889016305292

Standford. (2007). *Roboitcs operating system.* Descargado de https://www.ros.org/

Stückler, J. (2016). Mobile manipulation, tool use, and intuitive interaction for cognitive service robot cosero. Descargado de https://www.frontiersin.org/articles/10.3389/frobt.2016.00058/full

Sucan, I. A. (2018). *Moveit.* Descargado de https://moveit.ros.org/

Tomimoto, H. (2017). *Meal-assistance robot operated by head movement.* Descargado de https://link.springer.com/chapter/10.1007/978-3-319-64051-8_1

World robot summit. (2018). Descargado de http://blog.pal-robotics.com/world-robot-summit-tiago-clean-toilet-interact-customer/

Worthen-Chaudhari, L. C. (2014). Poststroke upper extremity rehabilitation: A review of robotic systems and clinical results. Descargado de https://www.tandfonline.com/doi/abs/10.1310/tsr1406-22

7 Anexo 1. Código Launch

Código 7.1: Código Launch de la simulación

```
<?xml version="1.0" encoding="UTF-8"?>
<launch>
  <arg name="namespace" default="/"/>
  <arg name="robot" default="steel"/>
  <arg name="arm" default="(eval {'iron': False}.get(arg('robot'), True))"/>
  <arg name="end_effector" default="(eval {'iron': 'false', 'steel': 'pal-gripper', 'titanium': 'pal-hey5'}.get(arg('robot'), 'pal-gripper'))"/>
  <arg name="ft_sensor" default="(eval {'titanium': 'schunk-ft'}.get(arg('robot'), 'false'))"/>
  <arg name="laser_model" default="sick-571"/>
  <arg name="camera_model" default="orbbec-astra"/>

  <arg name="world" default="hospital_hab"/>
  <arg name="gzpose" default="-x 0.0 -y -0.0 -z 0.0 -R 0.0 -P 0.0 -Y 0.0"/>
  <arg name="tuck_arm" default="true"/>

  <arg name="planner" default="base"/>
  <arg name="global_planner" default="global_planner"/>
  <arg name="local_planner" default="pal"/>
  <arg name="localization" default="amcl"/>
  <arg name="map" default="(env HOME)/.pal/tiago_maps/configurations/(arg world)"/>

  <arg name="rviz" default="true"/>
  <arg name="gzclient" default="true"/>
  <arg name="recording" default="false"/>
  <arg name="extra_gazebo_args" default=""/>

  <arg name="public_sim" default="false"/>
  <arg name="advanced_navigation" default="false"/> <!-- Requires extra software from PAL Robotics-->

  <arg name="sim_sufix" value="_public_sim" if="(arg public_sim)"/>
  <arg name="sim_sufix" value="" unless="(arg public_sim)"/>

  <arg name="verbose_publishing" default="false"/> <!-- If false the node will only publish when there is some face detected -->
  <arg name="rate" default="5"/> <!-- Rate at which the node will look for faces and publish -->
  <arg name="image_topic" default="/xtion/rgb/image_raw"/>

```

```
<node name="pal_face" pkg="pal_face_detector_opencv" type="pal_face_detector_opencv" args="(arg verbose_publishing) (arg rate)" output="screen">
  <remap from="/pal_face/image" to="(arg image_topic)"/>
  <rosparam>
    processing_img_width: 320
    processing_img_height: 240
    rel_min_eye_dist: 0.02
    rel_max_eye_dist: 0.2
  </rosparam>
</node>

<env name="PAL_HOST" value="tiago" />

<node name="update_maps_link" pkg="pal_navigation_sm" type="base_maps_symlink.sh" args="tiago_maps"/>

<group ns="(arg namespace)">

  <include file="(find tiago_gazebo)/launch/tiago_gazebo.launch">
    <arg name="world" value="(arg world)"/>
    <arg name="arm" value="(arg arm)"/>
    <arg name="end_effector" value="(arg end_effector)"/>
    <arg name="ft_sensor" value="(arg ft_sensor)"/>
    <arg name="laser_model" value="(arg laser_model)"/>
    <arg name="camera_model" value="(arg camera_model)"/>
    <arg name="gzpose" value="(arg gzpose)"/>
    <arg name="tuck_arm" value="(arg tuck_arm)"/>
    <arg name="gui" value="(arg gzclient)"/>
    <arg name="public_sim" value="(arg public_sim)"/>
    <arg name="recording" value="(arg recording)"/>
    <arg name="extra_gazebo_args" value="(arg extra_gazebo_args)"/>
    <arg if="(eval local_planner == 'teb')" name="use_dynamic_footprint" value="true"/>
  </include>

  <include file="(find tiago_2dnav)/launch/navigation.launch">
    <arg name="state" value="localization"/>
    <arg name="planner" value="(arg planner)"/>
    <arg name="global_planner" value="(arg global_planner)"/>
    <arg name="local_planner" value="(arg local_planner)"/>
    <arg name="localization" value="(arg localization)"/>
    <arg name="map" value="(arg map)"/>
    <arg name="public_sim" value="(arg public_sim)"/>
    <arg name="rgbd_sensors" value="(arg advanced_navigation)"/>
  </include>

 <group if="(arg advanced_navigation)">
    <node name="rviz" pkg="rviz" type="rviz" if="(arg rviz)"
         args="-d (find tiago_2dnav)/config/rviz/advanced_navigation.rviz"/>
</group>
```

```
83     <group unless="(arg advanced_navigation)">
84      <node name="rviz" pkg="rviz" type="rviz" if="(arg rviz)"
85           args="-d (find tiago_2dnav)/config/rviz/navigation(arg sim_sufix).↩
                 ↪ rviz"/>
86    </group>
87  </group>
88
89  <group if="(arg public_sim)">
90    <node name="relay_map" pkg="topic_tools" type="relay" args="/map /vo_map"/>
91  </group>
92
93  <group unless="(arg public_sim)">
94    <include file="(find pal_loc_measure)/launch/pal_loc_measure.launch">
95      <arg name="training" value="False"/>
96    </include>
97  </group>
98
99  <group if="(arg advanced_navigation)">
100   <!-- Advanced navigation -->
101   <include file="(find pal_head_manager)/launch/tiago_head_manager.launch"/>
102   <node pkg="tiago_2dnav" type="navigation_camera_mgr.py" name="↩
          ↪ navigation_camera_mgr" />
103   <node pkg="pal_zoi_detector" type="zoi_detector_node" name="zoi_detector" ↩
          ↪ />
104   <include file="(find pal_waypoint)/launch/pal_waypoint.launch"/>
105   <include file="(find pal_navigation_manager)/launch/poi_navigation_server.↩
          ↪ launch"/>
106   <include file="(find tiago_laser_sensors)/launch/rgbd_cloud_laser.launch">
107     <arg name="cloud" value="/xtion/depth_registered/points"/>
108   </include>
109   <include file="(find pal_map_utils)/launch/map_configuration_server.launch↩
          ↪ ">
110     <env name="PAL_HOST" value="tiagoc"/>
111   </include>
112 </group>
113
114 </launch>
```

8 Anexo 2. Código del proceso.

Código 8.1: Código del proceso

```
#!/usr/bin/python

import sys
import os
import dlib
import glob
import turtle
import cv2
import sys
import copy
import rospy
import rospy
from moveit_python import *
from control_msgs.msg import PointHeadAction, PointHeadGoal,FollowJointTrajectoryAction, ↩
    ↪ FollowJointTrajectoryActionGoal, FollowJointTrajectoryGoal,JointTolerance
from trajectory_msgs.msg import JointTrajectory,JointTrajectoryPoint
import moveit_commander
import moveit_msgs.msg
import tf
from geometry_msgs.msg import PoseStamped, PointStamped
from move_base_msgs.msg import MoveBaseAction, MoveBaseGoal
import actionlib
from actionlib_msgs.msg import GoalID
from nav_msgs.msg import Odometry
import math
from std_msgs.msg import String, Header
from moveit_commander.conversions import pose_to_list
from pal_detection_msgs.msg import FaceDetections
import tf2_ros
import tf2_geometry_msgs
rospy.init_node('tiago_client_py')
error_log = open("error_log.txt", "w") # Registro para analizar los errores
class tiago():

    cap = cv2.VideoCapture(0)
    detector = dlib.get_frontal_face_detector()
    predictor = dlib.shape_predictor("predictor.dat")
    #win = dlib.image_window()
    feed=False;
    face_detected=False
    face_pose_detected=False
    timer=0
    pose_face=PoseStamped()
    tf_buffer = tf2_ros.Buffer() #tf buffer length
    tf_listener = tf2_ros.TransformListener(tf_buffer)

    def __init__(self):
        rospy.Subscriber("/pal_face/faces", FaceDetections, self.callback_face) # Creamos un nodo suscrito al↩
            ↪  topic de PAL face
        rospy.Subscriber("/texture_detector/pose",PoseStamped,self.callback_poseface)


```

```
51  def callback_face(self,face): #Callback que se realiza solo si se ha detectado alguna cara
52      if len(face.faces)==1: #Comprobamos el numero de caras detectadas
53          self.face_detected=True
54      else:
55          pass
56
57  def callback_poseface(self,pose): # Callback que se reliza cuando se obtiene la posicion de la cara
58      error_log.write("seq: "+str(pose.header.seq)+"\n")
59      error_log.write("x: "+str(pose.pose.position.x)+"\n")
60      error_log.write("y: "+str(pose.pose.position.y)+"\n")
61      error_log.write("z: "+str(pose.pose.position.z)+"\n")
62      if self.face_pose_detected==False:
63          #Obetenemos la transformacion entre camara y base
64          transform = self.tf_buffer.lookup_transform("base_footprint","xtion_rgb_optical_frame",rospy.↩
                ↪ Time(0),rospy.Duration(1.0))
65          #Aplicamos la transformacion
66          self.pose_face = tf2_geometry_msgs.do_transform_pose(pose, transform)
67          self.face_pose_detected=True
68          print("Position saved", pose)
69      else:
70          pass
71  def move_head(self,direction):
72
73      client=actionlib.SimpleActionClient('/head_controller/point_head_action',PointHeadAction) #↩
            ↪ Creamos un cliente del tipo PointHead
74      client.wait_for_server()# Esperamos que el servidor confirme que el cliente es correcto
75      head_goal = PointHeadGoal() # Creamos un mensaje de tipo PointHead
76      point=PointStamped() # Creamos un mensaje de tipo Point
77      point.header.stamp=rospy.Time.now() #Definimos el encabezado del Point
78      point.header.frame_id="/xtion_rgb_optical_frame" #Frame captado por la camara RGB
79      if direction=="down": # En funcion de la direccion a la que queremos mirar se define el punto
80          point.point.x=0
81          point.point.y=0.65
82      elif direction=="up":
83          point.point.x=0
84          point.point.y=-0.65
85      elif direction=="right":
86          point.point.x=0.65
87          point.point.y=0
88      elif direction=="left":
89          point.point.x=-0.65
90          point.point.y=0
91      else:
92          point.point.x=0
93          point.point.y=0
94      point.point.z=1
95      head_goal.target=point
96      head_goal.pointing_axis.x=0 # Eje sobre el que queremos apuntar la cabeza
97      head_goal.pointing_axis.y=0
98      head_goal.pointing_axis.z=1
99      head_goal.pointing_frame="/xtion_rgb_optical_frame" # Frame de la camara
100     head_goal.min_duration.secs=1 #Parametros del movimiento
101     head_goal.min_duration.nsecs=0
102     head_goal.max_velocity=0.25
103     client.send_goal(head_goal)
104     wait=client.wait_for_result()
105     print("Moved head")
106
107 def move_base(self,x,y,yaw):
108
109     client = actionlib.SimpleActionClient('move_base',MoveBaseAction) # Creamos un cliente para la ↩
            ↪ accion move base
110     client.wait_for_server() # Esperamos a que el cliente se cree en el servidor
111     base_goal = MoveBaseGoal() # Creamos un mensjae del tipo MoveBaseGoal
```

```
112         base_goal.target_pose.header.frame_id="map" # Definimos el encabezado del mensaje
113         base_goal.target_pose.header.stamp = rospy.Time.now()
114 
115         quaternion=tf.transformations.quaternion_from_euler(0,0,yaw) # Transformamos la rotacion deseade ↩
                ↪ de Euler a cuaternion
116         base_goal.target_pose.pose.position.x = x #Definimos la posicion deseada en el mensaje
117         base_goal.target_pose.pose.position.y = y
118         base_goal.target_pose.pose.position.z = 0
119         base_goal.target_pose.pose.orientation.x=quaternion[0] #Definimos la rotacion deseada en el mensaje
120         base_goal.target_pose.pose.orientation.y=quaternion[1]
121         base_goal.target_pose.pose.orientation.z=quaternion[2]
122         base_goal.target_pose.pose.orientation.w = quaternion[3]
123         client.send_goal(base_goal) # Enviamos el goal
124         wait=client.wait_for_result() # Esperamos que se alcance o timeout
125         print("Point reached or unreachable") # Confirmamos por consola
126 
127     def move_arm(self,x,y,z,roll,pitch,yaw,wait):
128 
129         arm_goal =PoseStamped() # Creamos un mensaje de tipo PoseStamped
130         quaternion = tf.transformations.quaternion_from_euler(roll, pitch, yaw) # Convertimos los angulos de↩
                ↪  Eurler a cuaternion
131         arm_goal.header.frame_id = "base_footprint" # Definimos el encabezdo del mensaje
132         arm_goal.header.stamp = rospy.Time.now()
133         arm_goal.pose.position.x = x # Definimos la posicion deseada para arm_tool
134         arm_goal.pose.position.y = y
135         arm_goal.pose.position.z = z
136         arm_goal.pose.orientation.x=quaternion[0] # Definimos la orientacion deseada para arm_tool
137         arm_goal.pose.orientation.y=quaternion[1]
138         arm_goal.pose.orientation.z=quaternion[2]
139         arm_goal.pose.orientation.w = quaternion[3]
140         group= MoveGroupInterface("arm_torso", "base_footprint") # Creamos una interfaz de MoveIt ↩
                ↪ definiendo el grupo a planear y la referencia
141         group.moveToPose(arm_goal,"arm_tool_link",wait=wait) # Mandamos el goal con el link que ↩
                ↪ alcanzara dicha posicion
142         #print("Moved arm or unreachable") # Confirmamos por consola
143 
144     def gripper(self,state):
145 
146         client=actionlib.SimpleActionClient('/gripper_controller/follow_joint_trajectory',↩
                ↪ FollowJointTrajectoryAction) # Creamos el cliente
147         client.wait_for_server()# Esperamos a que el server confirme la creacion
148 
149         goal=FollowJointTrajectoryGoal()# Mensaje para definir el objetivo final
150         traj=JointTrajectory() # Mensaje que contiene la trayectoria
151         point=JointTrajectoryPoint() # Mensaje que contiene el punto deseado
152         path_tol=JointTolerance() # Mensaje para la tolerancia del camino
153         goal_tol=JointTolerance() # Mensaje para la tolerancia del objetivo
154 
155         header=Header() # Definimos un encabezado para usar en el resto de mensajes
156         header.seq=1
157         header.stamp=rospy.Time.now()
158         header.frame_id=''
159 
160         path_tol.name='path' # Definimos tolerancias bajas
161         path_tol.position=0.1
162         path_tol.velocity=0.1
163         path_tol.acceleration=0.1
164         goal.path_tolerance=[path_tol]
165 
166         goal_tol.name='goal'
167         goal_tol.position=0.1
168         goal_tol.velocity=0.1
169         goal_tol.acceleration=0.1
170         goal.goal_tolerance=[goal_tol]
```

```
171
172         goal.goal_time_tolerance.secs=0.1
173         goal.goal_time_tolerance.nsecs=0
174
175
176         traj.header=header
177         traj.joint_names=["gripper_left_finger_joint", "gripper_right_finger_joint"] # Especificamos que ↩
                ↪ articulaciones deseamos mover
178
179         if state=="open": # Segun el parametro del usuario abrimos o cerramos
180             point.positions=[0.044,0.044]
181         elif state=="close":
182             point.positions=[0.01,0.01]
183         else:
184             print("ERROR")
185
186         point.velocities=[] # Informacion no necesaria para este movimiento
187         point.accelerations=[]
188         point.time_from_start.secs=1
189         point.time_from_start.nsecs=0
190
191         traj.points=[point] # Encapsulamos los mensajes
192         goal.trajectory=traj
193         client.send_goal(goal) # Enviamos el mensaje final
194         wait=client.wait_for_result()
195         print("Gripper done") #Confirmamos por consola
196
197     def face_detection(self):
198         #ret, img= self.cap.read()
199         #self.win.clear_overlay()
200         #self.win.set_image(img)
201         img=dlib.load_rgb_image("face3.jpg")
202         dets = self.detector(img, 1)
203         if len(dets)==1:
204             shape = self.predictor(img, dets[0])
205
206             #self.win.add_overlay(shape)
207             #Deteccion de la boca
208             H_m=float(abs(shape.part(62).y-shape.part(66).y))
209             L_m=float(abs(shape.part(60).x-shape.part(64).x))
210             N_m=H_m/L_m #Calculo de la relacion de la apertura de la boca
211
212             if N_m>0.3: # Limite por el que se determina que la boca esta abierta.
213                 print("Mouth is open")
214                 self.timer=self.timer+1 # Inicio de la cuenta
215             else:
216                 print("Mouth is closed")
217                 self.timer=0
218
219             if self.timer>4: # Momento en que activamos el movimiento de alimentacion
220                 self.feed=True
221
222             else:
223                 self.feed=False
224         else:
225             print("ERROR-Detected no or more than one face.")
226
227
228 if __name__ == '__main__':
229     try:
230         tiago = tiago()
231         # Go to plate 1
232         tiago.move_base(1.6,-4,0)
233
```

```
234         #Pick
235         tiago.move_head("down")
236         tiago.move_arm(0,-0.6,0.9,0,0.785,0,True)
237         tiago.move_arm(0.6,-0.3,1.1,0,1.57,0,True)
238         tiago.move_arm(0.6,-0.3,0.9,0,1.57,0,True)
239         tiago.gripper("close")
240         tiago.move_arm(0.6,-0.3,1,0,1.57,0,True)
241         tiago.move_head("up")
242 
243         # Go to plate 2
244         tiago.move_base(-0.75,-5,1.57)
245 
246         #Place
247         tiago.move_head("down")
248         tiago.move_arm(0.55,-0.2,1,0,1.57,0,True)
249         tiago.move_arm(0.55,-0.2,0.95,0,1.57,0,True)
250         tiago.gripper("open")
251         tiago.move_arm(0.55,-0.2,1,0,1.57,0,True)
252 
253         #Ready spoon
254         tiago.move_base(-0.75,-5,0)
255         tiago.move_arm(0.55,-0.25,0.85,1.57,0,1.57,True)
256         tiago.move_base(-0.75,-5,1.57)
257 
258         #Spoon up
259         tiago.move_arm(0.55,-0.2,0.85,1.57,0.3,1.57,True)
260         tiago.move_arm(0.55,-0.15,0.85,1.57,0,1.57,True)
261 
262         # Ready to face
263         tiago.move_arm(0.5,0,0.9,1.57,0,0,True)
264         print("Start of cycle")
265         feed_time=0
266         while True:
267             rospy.sleep(0.5)
268             if tiago.face_pose_detected==True:
269                 feed_register=tiago.feed
270                 tiago.face_detection()# Deteccion de la boca
271                 if feed_register!=tiago.feed and tiago.feed==True: # Si esta abierta y no hay cambios, ↩
                        ↪ alimentamos
272                     print("Proceed to feed")
273                     tiago.move_arm(tiago.pose_face.pose.position.x-0.25,tiago.pose_face.pose.position.y,tiago.↩
                            ↪ pose_face.pose.position.z-0.1,1.57,0,0,False)
274 
275                 elif feed_register!=tiago.feed and tiago.feed==False: # Si esta cerrada y no hay cambios , ↩
                        ↪ retrocedemos
276                     tiago.move_arm(0.5,0,0.9,1.57,0,0,False)
277                     feed_time=0
278                     pass
279                 else:
280                     pass
281 
282                 if feed_register==tiago.feed and tiago.feed==True: # Contamo el tiempo que alimentamos
283                     feed_time=feed_time+1
284                 else:
285                     feed_time=0
286 
287                 if feed_time>10: # Si se supera el limite de tiempo, retiramos el brazo y reiniciamos el proceso
288                     #Spoon up
289                     tiago.move_arm(0.55,-0.2,0.85,1.57,0.3,1.57,True)
290                     tiago.move_arm(0.55,-0.15,0.85,1.57,0,1.57,True)
291                     # Ready to face
292                     tiago.move_arm(0.5,0,0.9,1.57,0,0,True)
293                 else:
294                     pass
```

```
295             else:
296                 pass
297 
298         rospy.spin()
299     except rospy.ROSInterruptException: pass
300     error_log.close()
```