

Trabajo Práctico Final

Robótica Móvil

Alumno: Guillermo Rolle

Curso de Posgrado
FCEIA-UNR

Docente: Taihú Pire

28 de febrero de 2024

Resumen

En este trabajo se implementó un sistema de localización basado en visión por computadora para un manipulador móvil que debe desplazarse en un ambiente interior como, por ejemplo, una fábrica. Los algoritmos fueron implementados en ROS2 y son probados mediante simulación en Gazebo.

El manipulador móvil utilizado es un prototipo de diseño que fue realizado para un Proyecto de Vinculación Tecnológica con una PyME metalúrgica local. Un problema típico que existe para este tipo de bases móviles es el problema de la localización, motivo por el cuál se eligió esta línea de trabajo, buscando resolver el problema de localización de una forma económica pero funcional.

En este informe se desarrollará por completo el trabajo realizado, la idea conceptual, los detalles de implementación en ROS y de la simulación en Gazebo; y la configuración y scripts para ejecutar los distintos nodos para reproducir los resultados.

Índice general

1. Introducción	3
1.1. Motivación	3
1.2. Repositorios	4
1.3. Manipulador Móvil	4
1.4. Marcadores ArUco	5
2. Algoritmo de localización	7
2.1. Idea conceptual	7
2.2. Árbol de Sistemas de Referencia	8
2.3. Implementación	9
2.4. ArUcos en Gazebo	10
3. Resultados	11
4. Guía de uso	15
4.1. Estructura del repositorio	15
4.2. Guía de uso del Docker	16
4.2.1. Compilación del Docker*	16
4.2.2. Preparación del entorno virtual de Python*	16
4.2.3. Compilación del Workspace*	17
4.2.4. Inicio del Contendor	17
4.2.5. Anexar otra terminal al contenedor	17
4.3. Guía de ROS y Gazebo	17
4.3.1. Ejecutar Simulación completa	17
4.3.2. Rosbag con Computer-Vision	18
4.3.3. Calibración de la Cámara	19
5. Conclusiones	22

Listado de Pendientes

■ Agregar calibración de la cámara	19
--	----

Capítulo 1

Introducción

1.1. Motivación

El trabajo desarrollado en el presente informe es una extensión de un Proyecto de Vinculación con el Medio, llevado a cabo por el LAC (Laboratorio de Automatización y Control) en conjunto con una PyME metalmecánica rosarina, enmarcado en el Programa TecnoPyME 2023 de la Provincia de Santa Fe.

En dicho proyecto, se hizo un prediseño mecánico conceptual de un manipulador móvil que permitiría transportar insumos entre los distintos sectores y estaciones de trabajo dentro de la fábrica. La base móvil fue desarrollada pensando en la modularidad de la misma. Es decir, se diseñó un único chasis que permitiría dotarlo de diferentes estructuras de operación y sistemas locomoción, según los requisitos y necesidades del cliente. De esta forma, se busca disminuir los costos de fabricación y maximizar la cantidad de configuraciones posibles del manipulador móvil mediante el acople/desacople de distintos módulos.

En cuanto a la localización de la base móvil dentro de la fábrica, a priorí, se pensó únicamente en procesar la información provista por la odometría de las ruedas motrices. Este método por sí sólo no permite una buena navegación dado que no hay ningún sensor exteroceptivo que procese información del entorno y la odometría acumula error rápidamente, tanto en presencia de deslizamiento de las ruedas, como por errores de parametrización. Por este motivo, en este trabajo se propone un método para mejorar la estimación la pose del robot utilizando visión por computadora.

1.2. Repositorios

El algoritmo de localización utilizando visión por computadora fue implementado en ROS2 y fue probado por simulación en Gazebo. Todos los archivos utilizados se encuentran en un repositorio de GitHub¹.

En el mismo se encuentran las fuentes completas del workspace de ROS, incluyendo los archivos para la simulación en Gazebo; un archivo DockerFile con varios scripts de utilidad para ejecutar las simulaciones en el contenedor Docker; y los archivos fuentes de este mismo informe.

Por otra parte, se creó un video demostrativo del funcionamiento del algoritmo en YouTube². En el video se muestra directamente la visualización en Rviz, comparando las trayectorias con y sin la corrección.

Por último, para reproducir los resultados (o realizar modificaciones en el algoritmo) en otra computadora sin tener que ejecutar una simulación en Gazebo desde cero, se subió un archivo bag de ROS de un ensayo en particular en Mega³ con todos los tópicos necesarios.

1.3. Manipulador Móvil

La base móvil utilizada es un prototipo conceptual desarrollado por el LAC para una PyME metalmecánica local, a la que se le adosó un manipulador *Scorbot ER-IX* (existente en el LAC) para emular el comportamiento de manipulador móvil. Se hace notar que, a fines de este trabajo práctico final de la materia, no hay ningún tipo de control sobre el manipulador y sólamente se busca mejorar la localización de la base móvil. El manipulador adosado, en este contexto, puede ser considerado parte de la base móvil.

En la Figura 1.1 se muestra la configuración del manipulador móvil en el entorno de Gazebo. El sistema de locomoción consta con un módulo de tracción diferencial atrás y dos ruedas cásters en el frente. El módulo con el manipulador, acoplado a la base móvil en parte trasera, tiene una rueda cástter también que hace de apoyo. En el frente, la base móvil tiene acoplado un módulo de sensores: éste tiene una cámara monocular, un sensor LiDaR 360° y una IMU. A los fines de este trabajo, el LiDaR sólamente se utiliza como visualización en Rviz y la IMU es ignorada completamente.

¹<https://github.com/guillerolle/carrolac-project>

²<https://youtu.be/1lgu2i4TX7Q>

³https://mega.nz/file/q0R3BYCK#S2Pc1iaddbtkwS_2Mw9nvdmk7yq-gsmmCQf6HnYI-aI

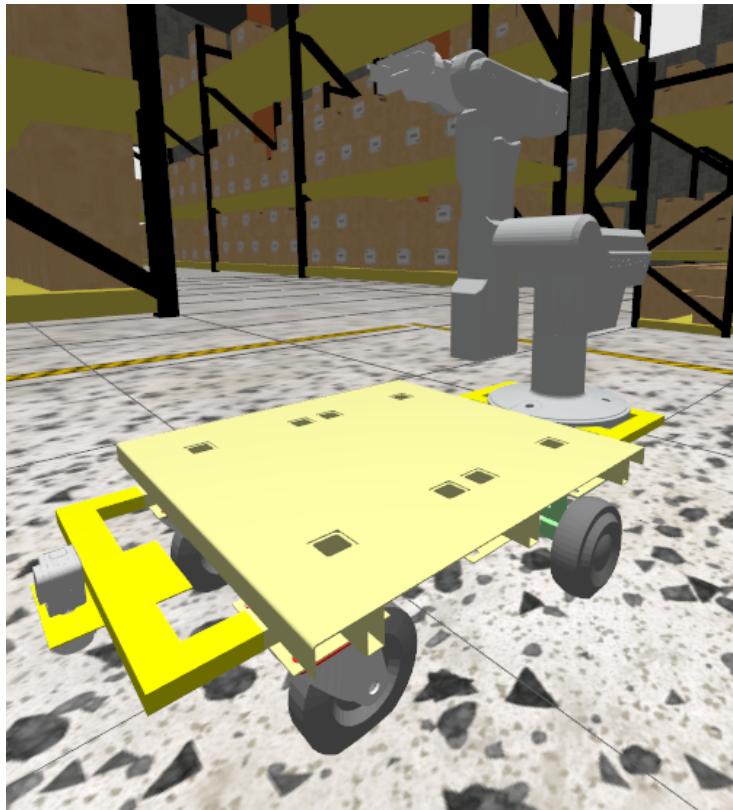


Figura 1.1: Manipulador móvil en Gazebo

1.4. Marcadores ArUco

Este método propone la utilización de marcadores ArUco⁴ para estimar la pose del robot.

Los marcadores ArUco son marcadores cuadrados que están dotados de una matriz binaria blanca y negra que codifica el identificador único del marcador en su interior y tienen un borde negro que permite la rápida detección del mismo. A su vez, el tamaño del marcador determina el tamaño de la matriz interna. Por ejemplo, un marcador de tamaño 4x4 está compuesto por 16 bits. En la Figura 1.2 se muestran ejemplos de marcadores ArUco de distinto tamaño.

⁴https://docs.opencv.org/4.x/d5/dae/tutorial_aruco_detection.html

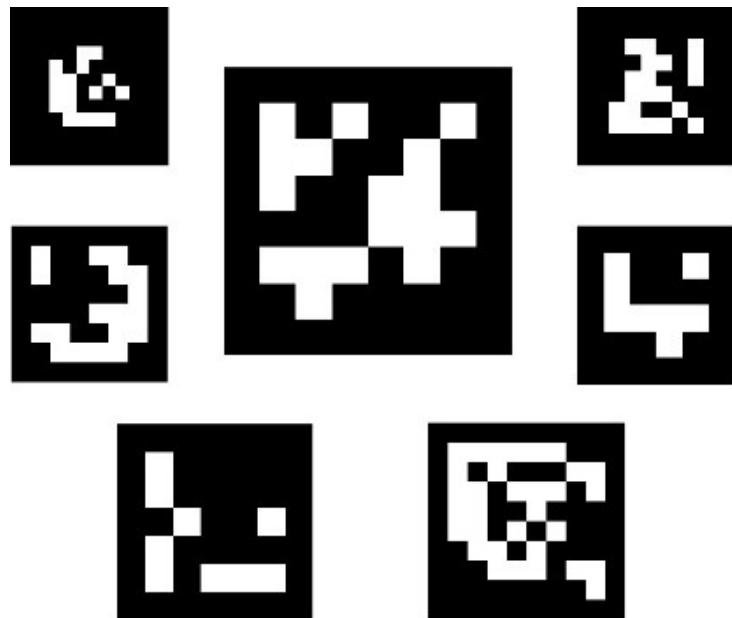


Figura 1.2: Ejemplos de marcadores ArUco

Los marcadores pueden ser encontrados girados en el entorno, por lo que el proceso de detección debe permitir determinar su orientación original para que cada esquina sea identificada inequívocamente. Esto también se hace basándose en la codificación binaria.

En este trabajo se utilizaron marcadores tamaño 6x6, con un diccionario de tamaño de 50 marcadores. En la Figura 1.3 se muestran dos marcadores ArUco dispuestos en el entorno de simulación de Gazebo.

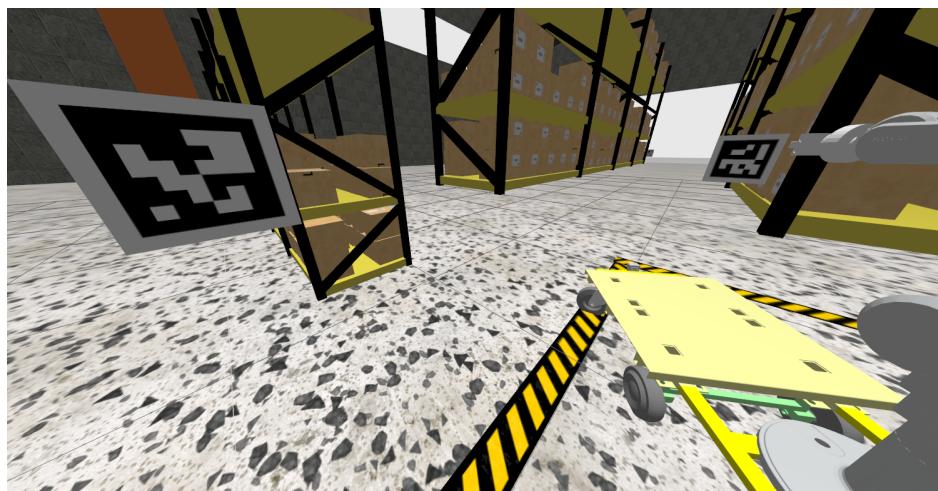


Figura 1.3: Marcadores ArUco en Gazebo

Capítulo 2

Algoritmo de localización

2.1. Idea conceptual

Conceptualmente, la idea es ubicar varios marcadores ArUco en posiciones y orientaciones conocidas dentro de la fábrica y utilizar la cámara para determinar la pose relativa entre el marcador y la base móvil.

En la Figura 2.1 se muestra una captura de la cámara del robot, en donde se ven 2 marcadores ArUco con sus correspondientes sistemas de referencia locales detectados.

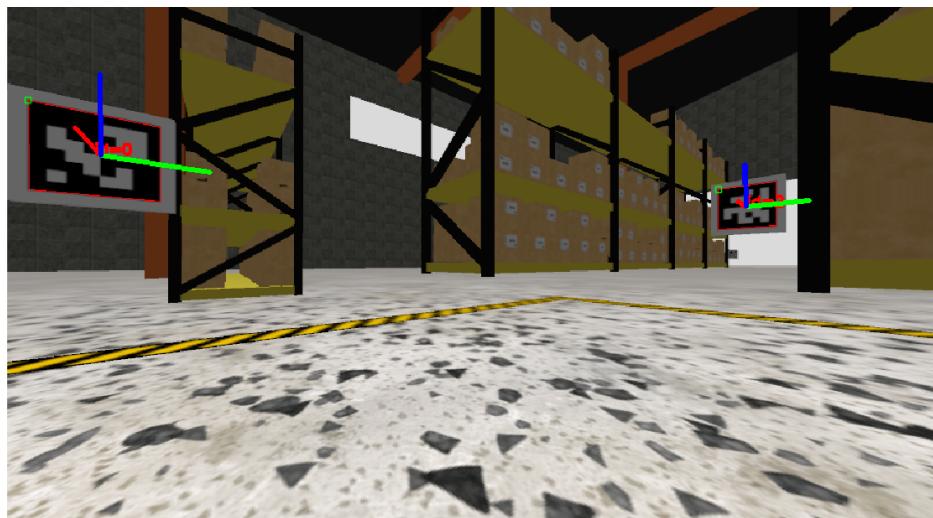


Figura 2.1: Identificación y estimación de la pose de los marcadores

Al conocer la pose real del marcador, se puede hacer una corrección de la pose de la base móvil estimada por la odometría para hacer que coincida la pose real con la pose observada del marcador por el robot. En la Figura 2.2

se muestran los errores relativos entre la pose real de cada marcador y la estimada por el robot en Rviz. Los errores se visualizan con una barra verde y representa el vector error de traslación entre el marcador real y el marcador estimado. El error de rotación también se tiene en cuenta en el algoritmo de localización pero no se muestra en esta figura.

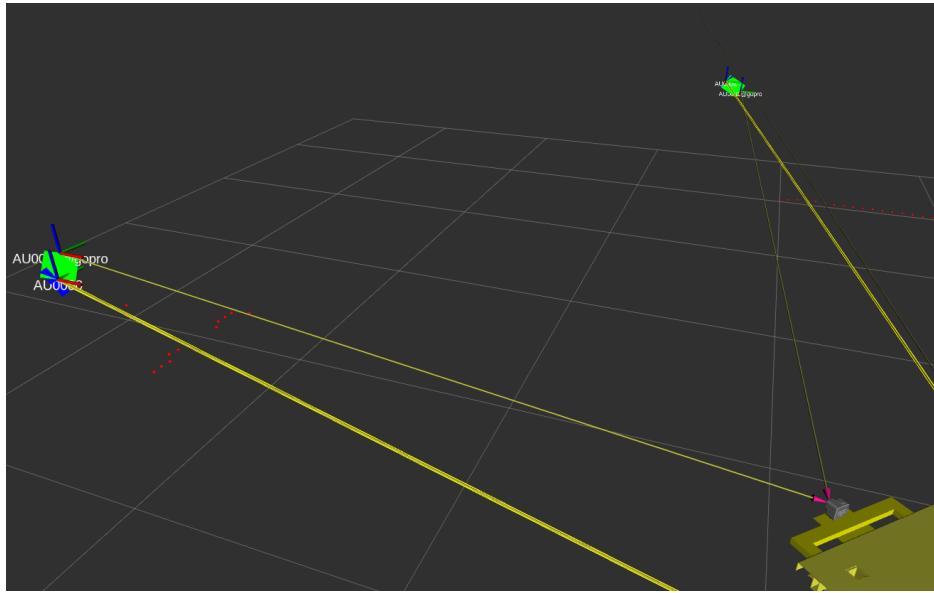


Figura 2.2: Error de translación de los marcadores

2.2. Árbol de Sistemas de Referencia

Es importante explicitar los sistemas de referencia utilizados en el algoritmo y su interdependencia. A continuación se muestra un extracto del árbol de sistemas de referencia utilizado.

```

map    # S.REF FIJO AL MUNDO
|- AU0000  # S.REF DEL ARUCO CON ID=0 RESPECTO A 'map'
|- ...
|- AUwxyz  # S.REF DEL ARUCO CON ID=wxyz RESPECTO A 'map'
|- odom:   # S.REF DE LA ODOMETRIA RESPECTO A 'map'
  |- link_base # S.REF DE LA BASE RESPECTO A 'odom'
    |- ...
      |- ...
        |- camera_optical # PLANO OPTICO DE LA CAMARA
          |- AU0000@camera # S.REF DEL ARUCO CON ID=0 RELATIVO
          |- ...
          |- AUwxyz@camera # S.REF DEL ARUCO CON ID=wxyz RELATIVO

```

Los nombres de algunos sistemas de referencia mostrados no coinciden necesariamente con el nombre exacto implementado para facilitar la interpretación conceptual.

- `map`: Sistema de referencia fijo al mundo. Representa el origen del sistema de coordenadas del entorno.
- `AU****`: Estos sistemas de coordenadas representan la pose de los marcadores ArUco respecto al mapa. Es decir, representan la pose "real" (o de Gazebo) de los marcadores
- `odom`: Es el origen del sistema de referencia de la odometría. Idealmente, coincidiría con `map`, pero como la odometría acumula errores se debe corregir. La diferencia entre `map` y `odom` representa la corrección de localización por el algoritmo implementado.
- `link_base`: Sistema de referencia de la base móvil. La posición relativa entre `odom` y `link_base` está dada por el cálculo de la odometría de la base móvil y debe dibujar una trayectoria suave y continua en el tiempo
- `camera_optical`: representa la posición del plano óptico de la cámara respecto a la base móvil
- `AU****@camera`: representa el sistema de referencia de los marcadores ArUco estimados relativos a la cámara.

Para lograr el objetivo, el algoritmo de localización por computadora debe hacer coincidir la pose global (respecto a `map`) de los sistemas de referencia `AU****` con la de su homólogo asociado `AU****@camera` mediante la modificación de la pose de `odom` respecto a `map`.

En este trabajo en particular, como se utilizó un modelo de cámara tipo GoPro, en la implementación se encontrará el identificador `gopro` en lugar del identificador `camera` donde corresponda.

2.3. Implementación

El algoritmo fue implementado en un nodo de ROS2 en Python en el archivo `src/computer_vision/computer_vision/aruco_detector.py` del repositorio. El nodo se suscribe a los tópicos de la imagen de la cámara y, al recibir un nuevo mensaje de imagen llama a la función `image_callback()`. Esta función utiliza la función `detectMarkers(img)` de la clase

`cv2.aruco.ArucoDetector` para detectar los marcadores en la imagen. Esta función devuelve las esquinas y los identificadores de los marcadores encontrados, que posteriormente son dibujados con `cv2.aruco.drawDetectedMarkers(...)`. Para cada marcador, va a calcular la pose relativa a la cámara con `cv2.solvePnP(...)`. Con estos datos, dibuja el sistema de referencia del marcador en la imagen de la cámara, publica esta nueva imagen en otro tópico y también publica en ROS la transformación `AU****@camera` entre el marcador y la cámara.

Por otra parte, un temporizador llama constantemente a la función `on_tf_timer()` en donde evalúa si hay transformaciones del tipo `AU****@camera` publicadas en ROS. En caso que las hubiera, determina el error tanto en traslación como en rotación entre `AU****` y `AU****@camera` a través de la función `lookup_transform()` de la clase `tf2_ros.Buffer`. Luego, los vectores error son trasladados al sistema de referencia del mapa para formar un error promedio de traslación y un error promedio de rotación entre todos los marcadores detectados en ese momento. Con los vectores errores, se modifica dinámicamente el sistema de referencia `odom` utilizando un filtro proporcional K_p . Se destaca que para reducir el error de orientación, se trabaja con vector de rotación y no con matriz de rotación o cuaternion. Esto es porque el vector de rotación permite sumar o restar las componentes y mantener una rotación suave y continua. El vector de rotación es un vector de 3 componentes en donde el versor asociado representa la dirección sobre la que se efectúa la rotación y la magnitud del vector es la magnitud del ángulo girado.

2.4. ArUcos en Gazebo

Los marcadores ArUco fueron modelados en Blender y colocados en posiciones conocidas en el mundo de Gazebo. Cada marcador tiene un identificador (i) de marcador ArUco al frente y en su lado de opuesto tiene identificador ($i+1$). Por ejemplo, el marcador `AU0000` con pose $[x_0, y_0, z_0, r_{x0}, r_{y0}, r_{z0}]$ tiene la misma posición que el `AU0001` pero rotado 180° en su eje z local.

Capítulo 3

Resultados

El video indicado en la Sección 1.2 muestra la visualización en Rviz de los resultados obtenidos. En el video mostrado, el robot comienza en el origen, es decir, los sistemas de referencia `map`, `odom` y `link_base` coinciden al comienzo de la simulación. Esta situación se muestra en la Figura 3.1.

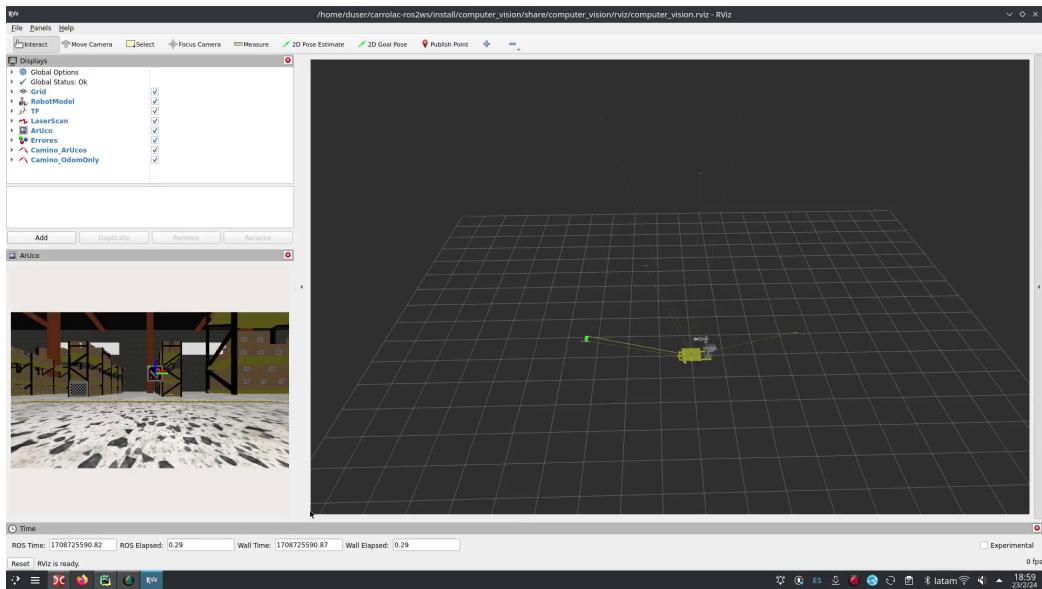


Figura 3.1: Resultados en Rviz. Condición Inicial

A medida que el carro se mueve se computa la odometría y el sistema `link_base` comienza a moverse continuamente respecto a `odom`.

Cuando la cámara del robot capta un marcador aruco, el sistema de referencia local del marcador es dibujado en la propia imagen de la cámara y también se agrega a la visualización 3D con el nombre `AU****@gopro`. Al

darse esta situación se agrega, entre los sistemas de referencias del marcador detectado y el original, una barra verde representando el error de traslación y una azul representa el error de rotación como vector como se muestra en la Figura 3.2

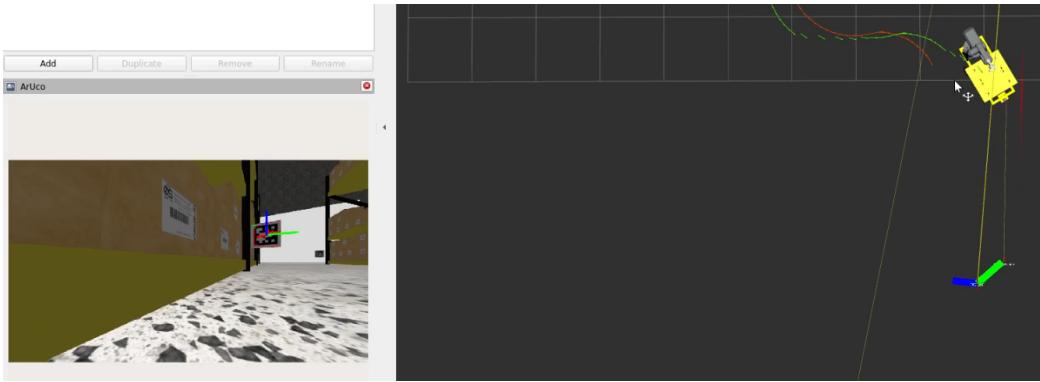


Figura 3.2: Error de traslación (verde) y rotación (azul) locales a un marcador

Los errores de traslación y rotación locales a cada marcador son trasladados instantáneamente al origen del mapa (`map`). En `map` queda finalmente un error de traslación total, en rojo, que representa el promedio de todos los errores de traslación detectados en ese instante; y un error de rotación total, en cian, que representa el promedio de todos los errores de rotación detectados en ese instante. Esta situación se muestra en la Figura 3.3.

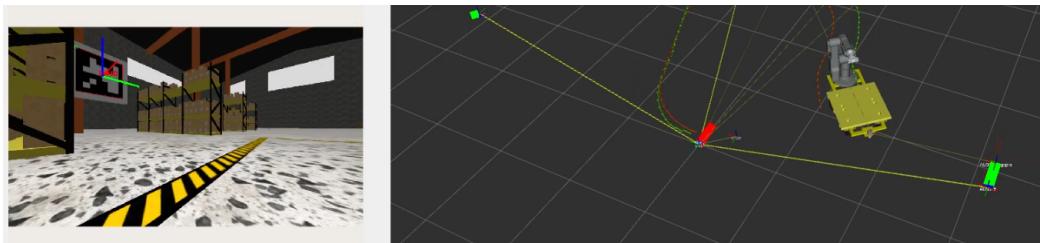


Figura 3.3: Errores trasladados al origen del mapa (`map`).

En este sentido, el sistema `odom` va a trasladarse y rotar respecto a `map` con una velocidad proporcional a los vectores error de traslación y rotación, respectivamente. De esta forma, el sistema `odom` se va a mover suavemente de manera que los sistemas instantáneos `AU****@gopro` converjan en posición y orientación a los `AU****`. Como ya se explicó, esta converjencia se hace desplazando el sistema `odom`. En la Figura 3.4 se muestra la corrección del camino en el sistema fijo `map`. Se puede ver la convergencia asintótica a una

pose que hace coincidir los sistemas del marcador ArUco. Cuanto más grande el error, mayor es la velocidad con la que se reduce el error.

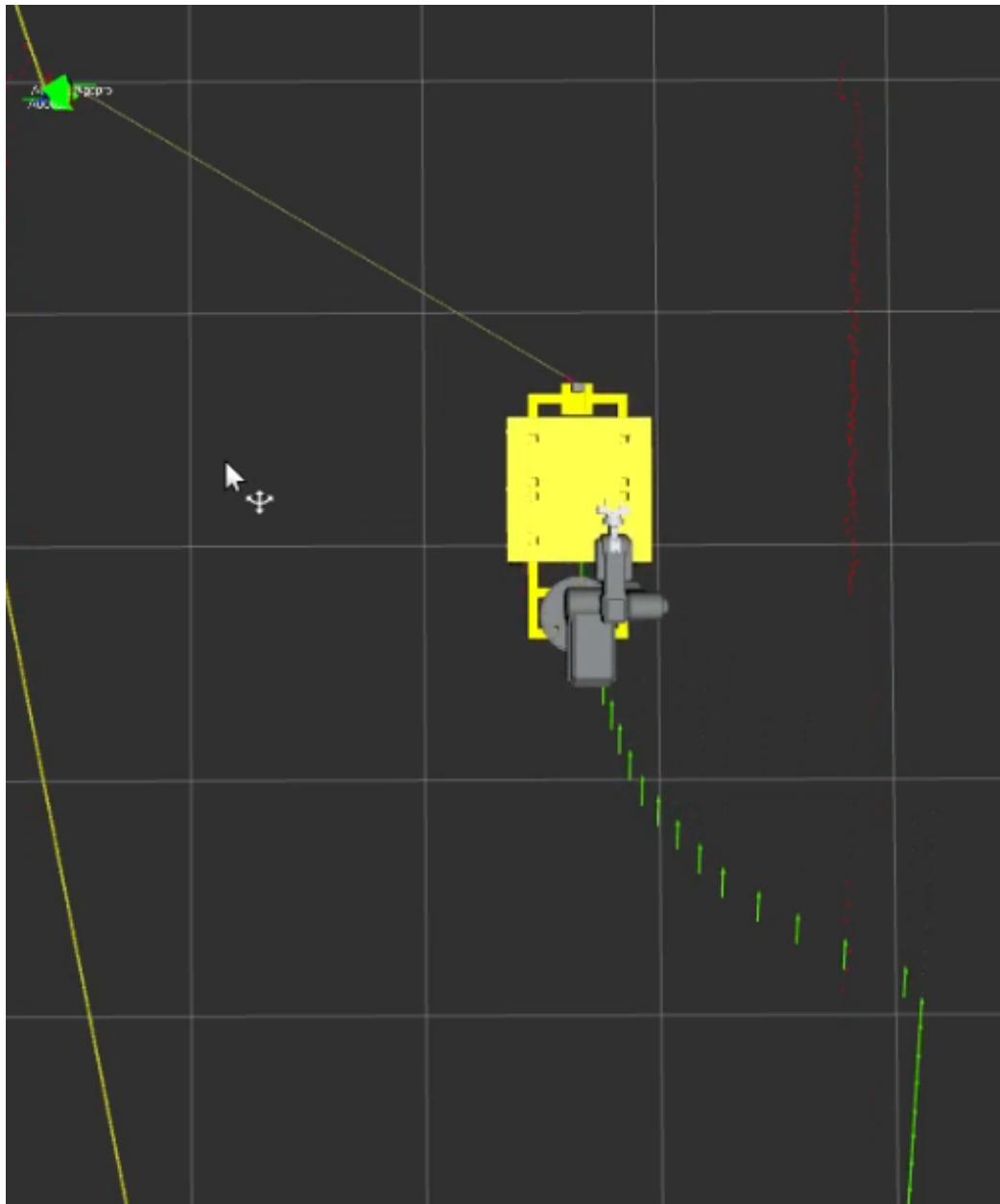


Figura 3.4: Convergencia del camino corregido (verde).

Finalmente, en la Figura 3.5 se muestran comparativamente el camino sin corrección (en naranja), es decir, utilizando únicamente la odometría, y

el camino corregido con los marcadores ArUco (en verde).

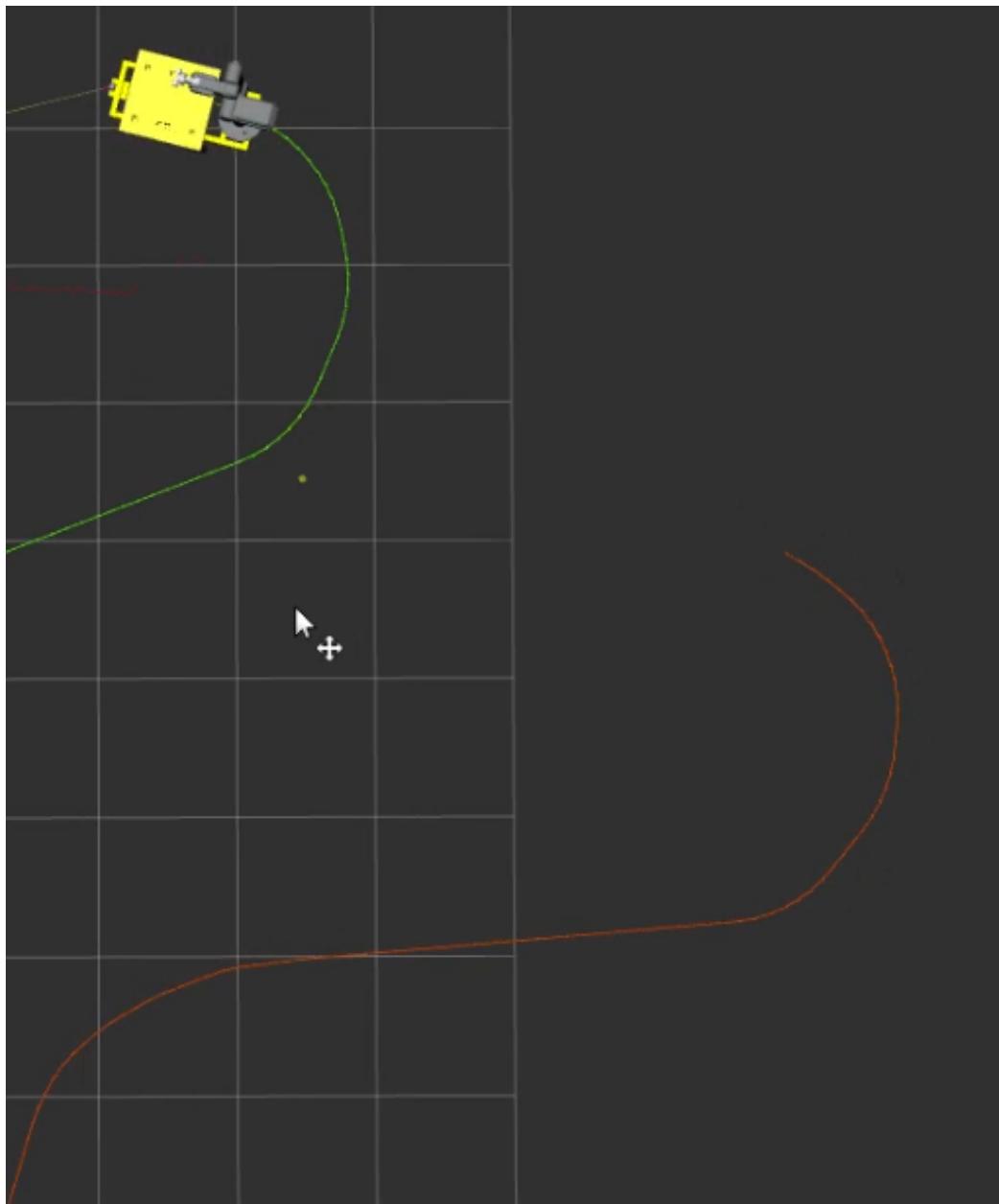


Figura 3.5: Comparación con (verde)/sin (naranja) detección de ArUco

Capítulo 4

Guía de uso

En este capítulo se provee una guía de uso para replicar los resultados obtenidos y/o ejecutar nuevos ensayos y modificaciones en los algoritmos.

4.1. Estructura del repositorio

En primer lugar, se debe clonar en la computadora local el repositorio de Git mencionado en la Sección 1.2:

```
1 $ git clone git@github.com:guillerolle/carrolac-project.git  
2
```

En el directorio base se encuentran las siguientes carpetas:

- **carrolac-ros2ws**: Es el workspace de ROS y Gazebo del proyecto. Dentro del mismo, contiene la carpeta **src** con todos los archivos fuente de los paquetes implementados en este proyecto.
- **docker-home**: Este directorio se usa como punto de montaje para **\$HOME** del contenedor Docker.
- **docker-sources**: Contiene puntos de montaje de distintos archivos fuente de ROS que están incluidos en la imagen de Docker. Permite tener referencias a clases y funciones de los paquetes instalados de ROS desde un IDE externo al Docker.
- **informe**: Archivos fuente del informe
- **scripts**: Scripts de asistencia para ejecución rápida y automática de ciertas funcionalidades.

- **venvs**: En este directorio se dispondrán los entornos virtuales de Python que se utilizan tanto en el IDE externo al Docker para programar los nodos, como el interno a Docker desde donde cargará los paquetes específicos de Python para la ejecución propiamente de los nodos.

4.2. Guía de uso del Docker

El símbolo * en los nombres de las siguientes subsecciones significa que son pasos que se deben ejecutar una única vez para poner en funcionamiento el entorno. No es necesario ejecutarlas cada vez que quiera correr algún nodo. Todos los scripts deben ser ejecutados desde la carpeta **scripts** para que funcionen correctamente.

4.2.1. Compilación del Docker*

Primero se debe compilar la imagen del docker se debe correr el script auxiliar **compile_docker.sh**.

4.2.2. Preparación del entorno virtual de Python*

Antes de poder ejecutar cualquier nodo, es necesario preparar el entorno de Python con las librerías necesarias en este proyecto.

Al ejecutar el script **start_docker_graphics.sh**, se iniciar el contenedor con los puntos de montajes necesarios. Al iniciar docker, debería aparecer la leyenda **SOURCEANDO .BASHRC...** en la terminal, indicando que el **\$HOME** del Docker está correctamente montado. La primera vez que se ejecuta, debería mostrarse un error al hacer **source /.bashrc** porque no se encuentra el entorno virtual de Python.

Dentro de la terminal del contenedor, debería haber una carpeta **/venvs**, que es un punto de montaje de la carpeta **venvs** del repositorio. Para crear el entorno virtual, correr las siguientes líneas:

```
1 $ cd ~/venvs
2 $ ./create_venv.sh
3
```

El script **create_venv.sh** crea un entorno virtual de Python en el directorio correspondiente para ser cargado automáticamente por **.bashrc** y con las librerías necesarias indicadas en **requirements.txt**.

4.2.3. Compilación del Workspace*

Para compilar el Workspace de ROS, desde la máquina HOST se debe correr el script `build_workspace.sh`. El workspace debería compilarse correctamente y los archivos de compilación deberían aparecer en la carpeta `carrolac-ros2ws` del HOST.

4.2.4. Inicio del Contenedor

Para iniciar el contenedor con todas las funcionalidades necesarias luego de ejecutar todos los pasos anteriores al menos una vez, basta con ejecutar el script `start_docker_graphics.sh`. Se iniciará una terminal dentro del contenedor y debería aparecer la leyenda `SOURCEANDO .BASHRC` sin ningún error ni mensajes a continuación.

4.2.5. Anexar otra terminal al contenedor

Para conectarse al contenedor con más de una terminal, una vez iniciado el contenedor como se indica en la subsección 4.2.4, se debe ejecutar el script `connect_docker.sh`.

4.3. Guía de ROS y Gazebo

4.3.1. Ejecutar Simulación completa

Es posible que la primera vez que se ejecute la simulación en Gazebo, ésta tarde bastante para iniciar porque Gazebo debe descargar los modelos de la fábrica utilizados en el mundo.

Para ejecutar el mundo de simulación en Gazebo, una vez iniciado un contenedor de Docker correctamente (sección 4.2.4), se ejecuta el comando

```
1 $ ros2 launch carrolac_gazebo gazebo.launch.py  
2
```

El archivo launch mostrado acepta los siguientes parámetros:

- `sim_on:={True|False}`: Indica si la simulación de Gazebo inicia pausada (False) o corriendo (True)
- `bag_on:={True|False}`: Indica si se generará o no un bag de la simulación. Tener en cuenta que los bag files pueden ser del orden de decenas de GB si se guardan todos los nodos juntos.

- **computer_vision:={True|False}**: Indica si se ejecutará el nodo de computer vision. El algoritmo de localización puede funcionar en vivo durante la simulación o reproduciendo un rosbag.
- **rviz:={True|False}**: Indica si inicia o no Rviz. Debería iniciarse solamente con **computer_vision:=True**
- **rqt_on:={True|False}**: Inicia rqt con una visualización del árbol de TF y la imagen original de la cámara. Útil para debug de códigos.

Por ejemplo, las 2 ejecuciones más comunes de la simulación son:

```
1 $ ros2 launch carrolac_gazebo gazebo.launch.py sim_on:=True
   bag_on:=False computer_vision:=True rviz:=True, rqt_on:=
   False
2
```

para correr el nodo de computer vision en vivo, o:

```
1 $ ros2 launch carrolac_gazebo gazebo.launch.py sim_on:=True
   bag_on:=True computer_vision:=False rviz:=False, rqt_on:=
   False
2
```

para guardar un bag file de la simulación, para luego probar el nodo de computer vision sin gastar recursos computacionales en la simulación.

Control de la plataforma móvil en Gazebo

Cualquiera sea la forma elegida para ejecutar la simulación, la base móvil se puede controlar manualmente con teclado con las teclas WASD:

- W: mueve la base móvil hacia adelante
- S: mueve la base móvil hacia atrás
- A: rota la base móvil en sentido antihorario
- D: rota la base móvil en sentido horario

4.3.2. Rosbag con Computer-Vision

La otra opción, más eficiente para probar el nodo implementado de computer vision es: lanzar un rosbag con los datos de la simulación propiamente dicha y luego lanzar, encima, el nodo de corrección con los ArUco. Se incorporó un archivo launch que gestiona esta situación. Para esto es necesario descargar el rosbag del Mega indicado en la sección 1.2 y descomprimirlo en la carpeta **carrolac-ros2ws/bags**. Una vez hecho esto, se puede ejecutar:

```
1 $ ros2 launch computer\_vision aruco.launch.py bag_on:=True  
2   bag_file:=bags/2024_02_23-18_43_03
```

donde, los argumentos son:

- **bag_on:={True|False}**: Activa o desactiva la carga automática de un rosbag
- **bag_file:=<string>**: Si **bag_on:=True**, indica cuál es el rosbag a cargar

Agregar calibración de la cámara

4.3.3. Calibración de la Cámara

Aunque la cámara es simulada en Gazebo, es necesario identificar los parámetros intrínsecos de la misma. Para esto, en el mismo mundo de Gazebo se incorporó un cartón calibrador tipo checkerboard como se muestra en la Figura 4.1

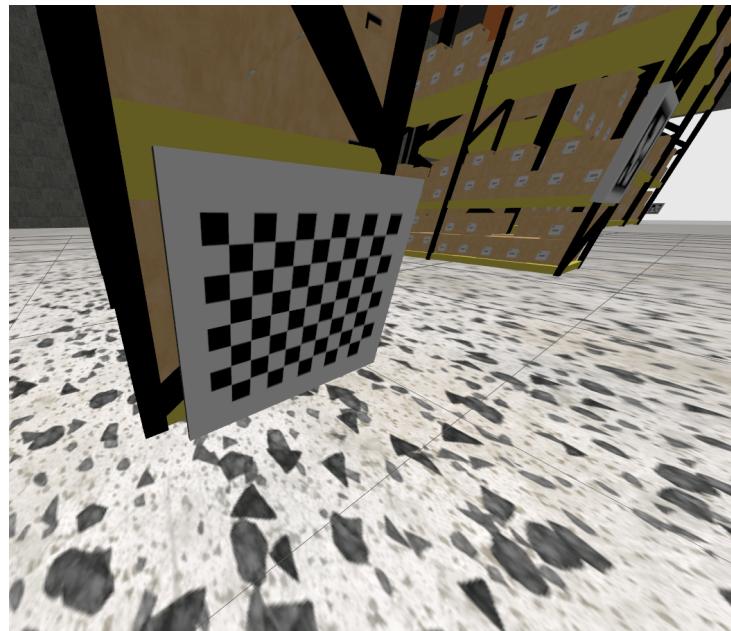


Figura 4.1: Checkerboard para calibración de la cámara

Para calibrar, es entonces necesario mover la base móvil y enfrentarla al checkerboard. Luego, se puede correr el siguiente nodo en otra terminal (usando el script `connect_docker.sh`):

```
1 $ ros2 run camera_calibration cameracalibrator --camera_name  
2   gopro --pattern chessboard --size 7x10 --square 0.015 --no  
   -service-check --ros-args -r image:=/gopro_camera/  
     image_raw --log-level DEBUG
```

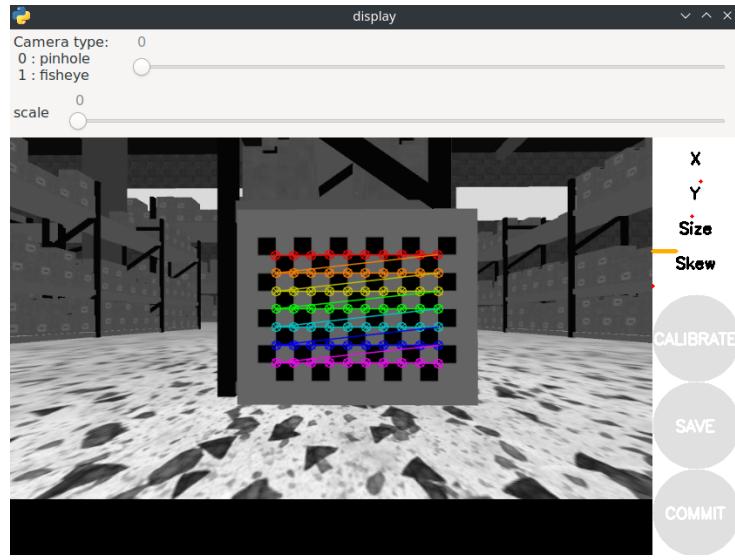


Figura 4.2: Calibrador de la cámara

Una vez abierto el calibrador y detectado el patrón, se puede ejecutar el siguiente script para producir movimientos en el checkerboard y facilitar la calibración de la cámara:

```
1 $ cd ~/carrolac-ros2ws/src/computer_vision/camera_calibrator  
2 $ ./move_calibrator_checkerboard.py  
3
```

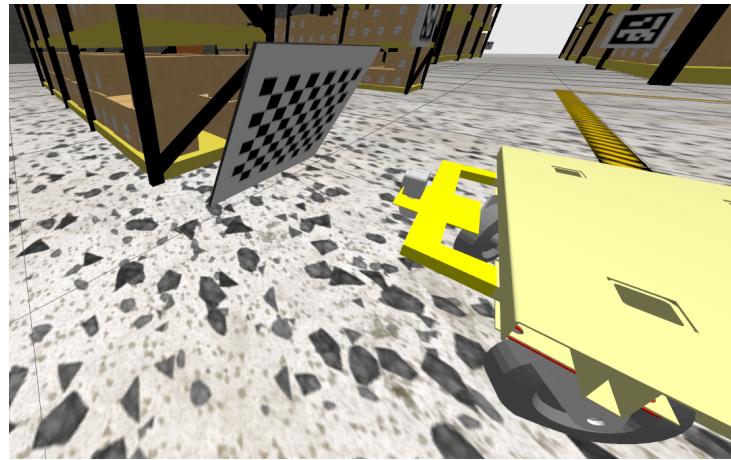


Figura 4.3: Checkerboard inclinado durante calibración

La calibración arrojará los parámetros intrínsecos y distorsión de la cámara, parámetros que son necesarios para detectar correctamente la pose relativa a los marcadores ArUco.

Comentario sobre la implementación

Si bien se pudo calibrar correctamente la cámara, no fue posible incorporar los datos de la calibración en Gazebo. Esto significa que, aunque se pueda modificar la resolución, campo de visión y más parámetros dentro de los mismos URDF y esto es correctamente leído por Gazebo, no se encontró forma de hacer que Gazebo publique correctamente los parámetros en el tópico `image_info`, ya que siempre publica los mismos valores por defecto sin importar los cambios que se hagan en el URDF. Debido a esto, los parámetros de la cámara son hardcodeados en la implementación del nodo en Python directamente. Lógicamente, este método está lejos de ser ideal pero no hubo éxito en implementarlo de una manera más correcta.

Capítulo 5

Conclusiones

En el presente trabajo se implementó un método de localización para ambientes interiores de una base móvil utilizando marcadores ArUco y visión por computadora, aplicado particularmente a un modelo de simulación que es pertinente al grupo del Laboratorio. El prototipo de manipulador móvil usado fue desarrollado exclusivamente para este fin, aunque se prevé su uso para muchos otros estudios de interés para el grupo.

El diseño en formato `urdf` fue pensado de forma modular, de forma que se puedan crear varias bases/manipuladores móviles de diferentes características incluyendo módulos específicos en cada caso. El formato modular de los `urdf` no fue probado exhaustivamente para este trabajo pero sirve para tener una primera versión de archivos.

Los algoritmos de visión por computadora fueron probados en Gazebo y se verificó un funcionamiento aceptable en general, aunque aparecieron muchas aristas para evidentes mejoras. Por ejemplo, al vibrar la cámara cuando se desplaza el robot se introduce mucho ruido en la imagen y esto se traslada a errores grandes en el posicionamiento, incluso llevando a la divergencia en algunos casos (raros), debido también al método de combinación de sensores (usando un filtro Proporcional). Otra problemática encontrada, es que este método requiere el conocimiento de la pose exacta de todos los marcadores ArUco en la fábrica. En la simulación eso no es problema, pero puede ser un problema al intentar implementarlo en la vida real.

Como trabajo a futuro en relación a los contenidos dados en la materia, es de interés incorporarle al robot la generación de un mapa del entorno y verificarlo con planos reales. Esto podría ayudar al robot a navegar por la fábrica, verificando su posición real mediante los ArUcos colocados estratégicamente.