Universitat d'Alacant
Universidad de Alicante

Grado en Ingeniería Robótica
Teleoperación

Práctica 4. Interfaz de teleoperación

Autor:
Miguel Ferrer Castellá

Índice

1. Introducción

En esta práctica se diseñó e implementó una interfaz de teleoperación para el control de un brazo robótico *Kinova Kortex* con una pinza *Robotiq 2f-140*. El sistema se desarrolló utilizando el simulador *Gazebo* y el dispositivo háptico *PHANTOM Omni*, lo que permitió integrar retroalimentación de fuerza durante la teleoperación.

El objetivo principal fue proporcionar al operador un control de posición con realimentacion de fuerzas del robot en un entorno simulado, acompañado de una interfaz que organiza la información en paneles modulares. Además, se incluyó el diseño de un controlador para sincronizar el movimiento del dispositivo maestro y el robot esclavo, superando diferencias cinemáticas mediante técnicas de escalado y mapeo del espacio de trabajo. Además, se introdujo la retroalimentación háptica y se probaron modos de operación seguros frente a colisiones y límites espaciales.

Esta práctica constituye una experiencia formativa que combina simulación robótica, diseño de interfaces y algoritmos avanzados de control en el contexto de la teleoperación.

El trabajo desarrollado de esta práctica se ha elaborado en un docker cuya imagen se puede encontrar en este enlace.

2. Entorno

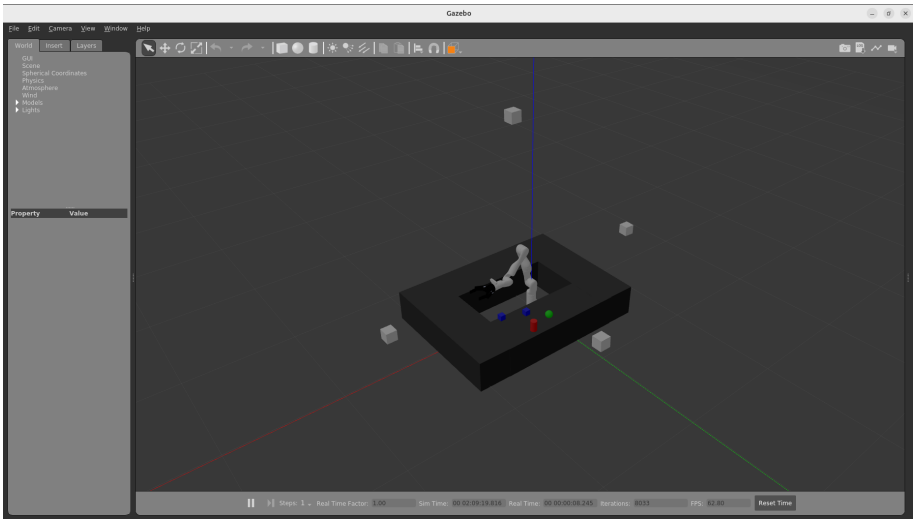

Figura 1: Entorno creado en Gazebo.

El entorno creado, denominado `practica4_2.world`, ha sido diseñado para integrarse con el simulador Gazebo y permitir pruebas específicas con un brazo robótico Kinova. Este mundo incluye una serie de elementos y configuraciones que lo hacen idóneo para ensayos de manipulación, interacción y visión artificial.

En cuanto a la superficie de trabajo, se utiliza un plano base estático denominado `ground_plane` sobre el que se encuentra el robot, que funciona como el suelo del entorno. Este plano está configurado con propiedades físicas específicas, como coeficientes de fricción y parámetros de contacto, que aseguran un comportamiento realista al interactuar con los objetos.
Además, se han dispuesto cuatro mesas de trabajo distribuidas, identificadas como `Mesa_trabajo_0`, `Mesa_trabajo_1`, `Mesa_trabajo_2` y `Mesa_trabajo_3`. Estas mesas ofrecen distintas áreas de interacción para el robot, dificultando mucho las operaciones sin colisión al estar el Kinova rodeado por las 4 mesas.

El mundo incluye también varios objetos de prueba, como cilindros, esferas y cajas, que están diseñados con propiedades físicas específicas para evaluar la capacidad de manipulación del robot y su interacción con diferentes formas y tamaños. Estos objetos, `unit_cylinder`, `unit_sphere`, `box_0` y `box_1`, son los utilizados para probar las tareas de agarre, la colocación y el movimiento de piezas.

Para el monitoreo del entorno, se han incorporado varias cámaras virtuales colocadas en posiciones estratégicas. Estas cámaras, denominadas `camera_Left`, `camera_Front`, `camera_Right` y `camera_Cenital`, ofrecen perspectivas variadas del entorno, facilitando la supervisión visual en la interfaz creada.

Finalmente, para integrar este nuevo mundo en Gazebo, se realizó una modificación en el archivo de lanzamiento predeterminado de ROS ubicado en la ruta `/opt/ros/noetic/share/gazebo_ros/launch/empty_world.launch`. En este archivo, se reemplazó la referencia al mundo por defecto (`empty_world.world`) con el archivo `practica4_2.world`. Al encontrarse ambos en la misma carpeta, esta sustitución garantiza que, al ejecutar Gazebo, se cargue automáticamente el nuevo entorno diseñado.

```
1  <arg name="world_name" default="worlds/practica4_2.world"/>
```

Listing 1: Cambio de entorno

3. Diseño de Control en Posición y Escalado del Espacio de Trabajo

Se ha diseñado un sistema de teleoperación que utiliza el Omni Phantom como robot maestro y un robot simulado My_gen3 como robot esclavo. Este sistema permite el control en posición del esclavo a través de las acciones realizadas sobre el maestro, implementando un acoplamiento cinemático y un escalado del espacio de trabajo.

3.1. Control de Posición y Acoplamiento Cinemático

3.1.1. Mecanismos Cinemáticamente Distintos

Dado que los robots maestro y esclavo son cinemáticamente distintos, se han conectado sus extremos considerando los desplazamientos y orientaciones de la siguiente

manera:

- **Conexión de Posiciones:** Se ha obtenido la posición del maestro en los nodos de posición del Omni.
- **Conexión de Orientaciones:** Se ha definido una orientación constante para que la orientación de la pinza fuera siempre estable. Asi facilitamos el proceso de manipulacion de objetos.
- **Desplazamiento de la Orientación:** Se ha ajustado mediante un factor de ajuste para hacer que la pinza estubiera recta respectoa a su espacio de trabajo.

3.2. Escalado y Mapeo del Espacio de Trabajo

3.2.1. Necesidad del Escalado

Dado que los espacios de trabajo del maestro (Omni Phantom) y del esclavo (My_gen3) tienen diferentes tamaños, se aplicó un escalado para mapear adecuadamente ambos espacios. Este escalado se implementó mediante las siguientes fórmulas:

$$x_{sd} = \mu x_m + x_{offset}, \tag{1}$$

$$x_{md} = \frac{x_s - x_{offset}}{\mu}. \tag{2}$$

Donde μ es el factor de escala.

3.2.2. Configuración de la Escala

La escala fue configurada para maximizar la comodidad del operador, permitiendo que los movimientos del maestro se reflejen de forma proporcional y controlada en el esclavo, evitando singularidades en los espacios de trabajo.

```
def transPos(x, y, z):
    """Transforma la posici n del maestro al rango del esclavo."""
    global goal_x, goal_y, goal_z
    goal_x = 0.2 + ((x - LIM_X_MIN) / (LIM_X_MAX - LIM_X_MIN)) * (0.6 - 0.2)
    goal_y = 0.15 + ((y - LIM_Y_MIN) / (LIM_Y_MAX - LIM_Y_MIN)) * (0.6 - 0.15)
    goal_z = 0.3 + ((z - LIM_Z_MIN) / (LIM_Z_MAX - LIM_Z_MIN)) * (0.65 - 0.3)

def transPosInversa(x, y, z):
    """Transforma posiciones del esclavo al maestro."""
    p_x = LIM_X_MIN + ((x - 0.2) / (0.6 - 0.2)) * (LIM_X_MAX - LIM_X_MIN)
```

```
    p_y = LIM_Y_MIN + ((y - 0.15) / (0.6 - 0.15)) * (LIM_Y_MAX
    - LIM_Y_MIN)
    p_z = LIM_Z_MIN + ((z - 0.3) / (0.65 - 0.3)) * (LIM_Z_MAX
   - LIM_Z_MIN)
    return p_x, p_y, p_z
```

Listing 2: Transformación de posiciones entre maestro y esclavo

3.3. Retroalimentación de Fuerza

Para realizar la retroalimentación de fuerzas, se han implementado dos apartados principales:

3.3.1. Caja de Fuerza

Este enfoque define una zona en el espacio de trabajo del robot que brinda retroalimentación háptica al operador. Al entrar en estas zonas, el dispositivo háptico (en este caso, el Omni) genera una fuerza que puede percibirse físicamente. Esto permite al operador "sentir"las limitaciones o restricciones del espacio del robot.

- Se aplicaron fuerzas en las posiciones límites del espacio de trabajo, orientadas hacia el centro del mismo.

```
LIM_X_MIN = -0.08
LIM_X_MAX =  0.08
LIM_Y_MIN = -0.06
LIM_Y_MAX =  0.018
LIM_Z_MIN = -0.073
LIM_Z_MAX =  0.048
```

Listing 3: Límites del espacio de trabajo del maestro

De esta manera,

```
if data.pose.position.x < LIM_X_MIN:
    force_msg = WrenchStamped()
    force_msg.wrench.force.x = f
    force_msg.wrench.force.y = 0
    force_msg.wrench.force.z = 0
    force_pub.publish(force_msg)
elif data.pose.position.x > LIM_X_MAX:
    force_msg = WrenchStamped()
    force_msg.wrench.force.x = -f
    force_msg.wrench.force.y = 0
    force_msg.wrench.force.z = 0
    force_pub.publish(force_msg)
elif data.pose.position.y < LIM_Y_MIN:
    force_msg = WrenchStamped()
    force_msg.wrench.force.x = 0
    force_msg.wrench.force.y = f
```

```
        force_msg.wrench.force.z = 0
        force_pub.publish(force_msg)
    elif data.pose.position.y > LIM_Y_MAX:
        force_msg = WrenchStamped()
        force_msg.wrench.force.x = 0
        force_msg.wrench.force.y = -f
        force_msg.wrench.force.z = 0
        force_pub.publish(force_msg)
    elif data.pose.position.z < LIM_Z_MIN:
        force_msg = WrenchStamped()
        force_msg.wrench.force.x = 0
        force_msg.wrench.force.y = 0
        force_msg.wrench.force.z = f
        force_pub.publish(force_msg)
    elif data.pose.position.z > LIM_Z_MAX:
        force_msg = WrenchStamped()
        force_msg.wrench.force.x = 0
        force_msg.wrench.force.y = 0
        force_msg.wrench.force.z = -f
        force_pub.publish(force_msg)
    else:
        force_msg = WrenchStamped()
        force_msg.wrench.force.x = 0
        force_msg.wrench.force.y = 0
        force_msg.wrench.force.z = 0
        force_pub.publish(force_msg)
```

Listing 4: Aplicación de las fuerzas según la posición

Esto crea una interacción más intuitiva con el espacio del robot, ya que el operador puede percibir los límites físicos virtuales.

3.3.2. Pozo de Gravedad

Este enfoque introduce un .ªgujero negro"que atrae al dispositivo háptico hacia el centro del brazo robótico en la simulación, pero en las coordenadas del Omni. La fuerza se implementa de la siguiente manera:

- Las coordenadas del robot se transforman al espacio de trabajo del Omni.
- El centro del pozo se establece en estas coordenadas y se aplican fuerzas atractivas que aumentan al alejarse del centro.

```
    # Calcula la posici n transformada
    transPos(x, y, z)
    p_x, p_y, p_z = transPosInversa(rx, ry, rz)
    print("Posici n␣transformada␣inversa␣x:", p_x, "␣y:", p_y
   , "␣z:", p_z)

    # Calcula la fuerza del pozo de gravedad
```

```
7       well_center = np.array([p_x, p_z, p_y])
8       force = compute_gravity_well_force(omni_position,
    well_center)
```

Listing 5: Cálculo del pozo de gravedad.

Una vez calculada la posición en la cual se debe aplicar el pozo de gravedad, se aplican las fuerzas del pozo siendo estas más fuertes cuando el dispositivo omni se encuentra más lejos de la posición relativa con el robot esclavo.

```
1  def compute_gravity_well_force(omni_position, well_center):
2      """
3      Calcula la fuerza que atrae al dispositivo hacia el centro
       de un pozo de gravedad.
4      Par metros:
5      - omni_position: numpy array con la posici n actual (x, y
       , z).
6      - well_center: numpy array con la posici n del centro del
       pozo.
7      Retorna:
8      - force: numpy array con las componentes de la fuerza (x,
       y, z).
9      """
10     displacement = well_center - omni_position  # Vector
       desplazamiento hacia el centro
11     print("well_center␣", well_center)
12     print("omni_position␣", omni_position)
13     distance = np.linalg.norm(displacement)  # Magnitud del
       desplazamiento
14
15     if distance < influence_radius:  # Dentro del radio de
       influencia
16         force_magnitude = k * distance - b * np.linalg.norm(
       omni_position)  # Magnitud de la fuerza
17         force_direction = displacement / distance  #
       Direcci n normalizada
18         # Se podr a activar una fuerza en base a condiciones
       espec ficas
19         force = np.array([0.0, 0.0, 0.0])
20     else:
21         force = np.array([0.0, 0.0, 0.0])  # Fuera del radio
       de influencia
22
23     return force
```

Listing 6: Aplicación del pozo de gravedad.

La idea principal del agujero negro es que, cuando el dispositivo háptico (el Omni) se encuentra dentro de un radio de influencia (definido por la variable influence_radius), el sistema genera una fuerza atractiva que empuja al Omni hacia el centro del pozo de gravedad (es decir, la posición definida en well_center). Esta

fuerza actúa como si el Omni estuviera siendo atraído hacia ese punto, que es la posición del robot de la simulación.
Los beneficios clave son:

1. **Alineación y Precisión:** Facilita que el operador mantenga el dispositivo háptico alineado con el brazo robótico en la simulación, especialmente en tareas de precisión.

2. **Control y Estabilidad:** Evita movimientos bruscos fuera del radio de influencia, generando una experiencia más realista y controlada.

En resumen, el pozo de gravedad mejora la precisión del control y garantiza la sincronización entre el operador y el robot, proporcionando una experiencia de teleoperación efectiva.

4. Colisiones

4.1. Nodos colisiones.cpp y peso.py

Doble escena RViz

En `conclusiones.cpp` se emplea una estrategia basada en dos versiones de la misma escena de MoveIt:

1. **Escena principal**
   Se configura para permitir colisiones (a través de la matriz de colisiones, `AllowedCollisionMatrix`). Al establecer `setEntry(..., true)` para todos los pares "robot - objeto", la planificación del robot no se ve restringida por posibles colisiones.
   Gracias a ello, el robot puede moverse libremente por la escena principal o situarse en cualquier posición que se desee en RViz sin que el planificador bloquee el movimiento.

2. **Escena clon o "diff"**
   En cada ciclo, se genera a partir de la escena principal usando `scene->diff()`. A continuación, se deshabilitan esas mismas entradas en la matriz de colisiones (`setEntry(..., false)`) para que sí se *reporten* las colisiones.
   Acto seguido, se llama a `checkCollision()` sobre esta escena clon. De esta manera se obtienen las colisiones reales (puntos de contacto, penetraciones, etc.) en función de la posición actual del robot.

Al final del proceso, se logra que la planificación (escena principal) ignore las colisiones y permita cualquier movimiento, mientras que, en paralelo, la escena clon refleja la realidad de las colisiones y calcula la información necesaria (contactos más profundos, visualización de marcadores, etc.). Esto resulta especialmente útil cuando se requiere analizar o registrar datos de colisión sin restringir la movilidad del robot en el entorno de RViz.

Coger y Soltar

El desarrollo del nodo colisiones.cpp representa un esfuerzo significativo para gestionar de manera precisa y eficiente las interacciones físicas del robot dentro de un entorno simulado. Este nodo opera en un bucle infinito que permite no solo detectar colisiones, sino también manejar el estado de la pinza del robot, sincronizar objetos en tiempo real y garantizar una representación visual coherente en RViz. A continuación, detallo los aspectos clave de su funcionamiento:

El nodo se basa en un sistema de monitoreo constante que detecta cuándo un objeto es atrapado por la pinza del robot. Para ello, hace uso del tópico `/estado_attach`, configurado de forma global en el archivo `GazeboGraspFix.cpp`. Este nodo no solo indica el momento exacto en que se produce la sujeción de un objeto, sino que también identifica que objeto en particular ha sido manipulado, publicando dicha información en un segundo tópico.

```
<xacro:if value="${vision}">
  <link name="${prefix}camera_link" />
  <joint name="${prefix}camera_module" type="fixed">
    <origin xyz="0 0.05639 -0.00305" rpy="3.14159265358979 -1.57079632679490 1.57079632679490" />
    <parent link="${prefix}end_effector_link" />
    <child  link="${prefix}camera_link" />
  </joint>
  <link name="${prefix}camera_depth_frame" />
  <joint name="${prefix}depth_module" type="fixed">
    <origin xyz="0.0275 0.066 -0.00305" rpy="3.14159265358979 -1.57079632679490 1.57079632679490" />
    <parent link="${prefix}end_effector_link" />
    <child  link="${prefix}camera_depth_frame" />
  </joint>
  <link name="${prefix}camera_color_frame" />
  <joint name="${prefix}color_module" type="fixed">
    <origin xyz="0 0.05639 -0.00305" rpy="3.14159265358979 -1.57079632679490 1.57079632679490" />
    <parent link="${prefix}end_effector_link" />
    <child  link="${prefix}camera_color_frame" />
  </joint>

  <gazebo reference="${prefix}camera_link">
    <sensor type="camera" name="camera_sensor">
      <update_rate>30.0</update_rate>
      <camera>
        <horizontal_fov>1.3962634</horizontal_fov>
        <image>
          <width>640</width>
          <height>480</height>
          <format>R8G8B8</format>
        </image>
        <clip>
          <near>0.03</near>
          <far>300</far>
        </clip>
      </camera>
      <plugin name="camera_controller" filename="libgazebo_ros_camera.so">
        <alwaysOn>true</alwaysOn>
        <updateRate>0.0</updateRate>
        <cameraName>robot_camera</cameraName>
        <imageTopicName>onboard</imageTopicName>
        <cameraInfoTopicName>camera_info</cameraInfoTopicName>
        <frameName>${prefix}camera_link_optical</frameName>
      </plugin>
    </sensor>
  </gazebo>
</xacro:if>
```

```
// MODIFICADO
size_t pos = objName.find("::");
std::string result;
result = objName.substr(0, pos);

std_msgs::String objeto_attached_msg;
objeto_attached_msg.data = result;


std_msgs::Bool attach_msg;
attach_msg.data = true;

ros::Rate rate(10);
objeto_attached_pub.publish(objeto_attached_msg);
estado_attach_pub.publish(attach_msg);
// FIN
```

Figura 2: Código modificado de GazebograspFix.cpp

Cuando el nodo detecta un flanco ascendente en el estado de sujeción (indicando que un objeto ha sido atrapado), registra inmediatamente la posición vertical (z) de la muñeca del robot. Este dato, extraído del topic /gazebo/link_states, actúa como una referencia para calcular la altura del objeto capturado. La altura registrada inicialmente se compara continuamente con la altura actual de la muñeca mientras el estado de sujeción permanece activo. Si se detecta que la altura actual supera significativamente la inicial, un nodo auxiliar (peso.py) activa un booleano que publica el peso estimado del objeto en el topic /peso_objeto. Este enfoque asegura que solo se registre el peso cuando el objeto está siendo levantado. En caso contrario, cuando no hay cambios en la altura, el booleano permanece inactivo, y el topic indica un peso de cero, lo que implica que no hay fuerzas gravitacionales relevantes actuando sobre la pinza.

Un aspecto crucial del nodo es la sincronización en tiempo real entre el entorno

simulado de Gazebo y la representación visual en RViz. Esto se logra actualizando constantemente las posiciones de los objetos en Gazebo a través del topic /gazebo/-model_states al final de cada iteración del bucle infinito. Esta información permite que los objetos manipulables en RViz reflejen con precisión sus contrapartes en el entorno de simulación, garantizando una experiencia inmersiva y coherente para el usuario.

Durante el estado activo de sujeción, el nodo también publica continuamente las coordenadas (x, y, z) de la muñeca del robot en diversos topics. Estos datos son fundamentales para calcular la posición dónde actuará el "pozo de gravedad", cuando se trabaja con el dispositivo omni phantom. En ausencia de este dispositivo, la interfaz refleja el estado de la pinza, mostrando visualmente si está abierta o cerrada, dependiendo del estado de sujeción.

La detección de colisiones se realiza en cada ciclo del bucle infinito utilizando el motor de colisiones de MoveIt!. Si el robot no está sujetando ningún objeto, el nodo calcula la posición media de los puntos de contacto en el efector final, en caso de que existan colisiones. Esta información se publica en topics específicos y también se utiliza para complementar el cálculo del pozo de gravedad.

El nodo optimiza el rendimiento al limitar la impresión de información sobre colisiones únicamente a los momentos en que hay un cambio de estado (por ejemplo, al pasar de no tener colisiones a detectarlas) o cuando se modifican los puntos de contacto en iteraciones consecutivas. Sin embargo, incluso cuando no se imprimen detalles en la terminal, las colisiones se representan visualmente en RViz mediante marcadores. Estos marcadores muestran la posición de los contactos más profundos y la penetración máxima, proporcionando una representación gráfica clara de las interacciones físicas entre el robot y su entorno.

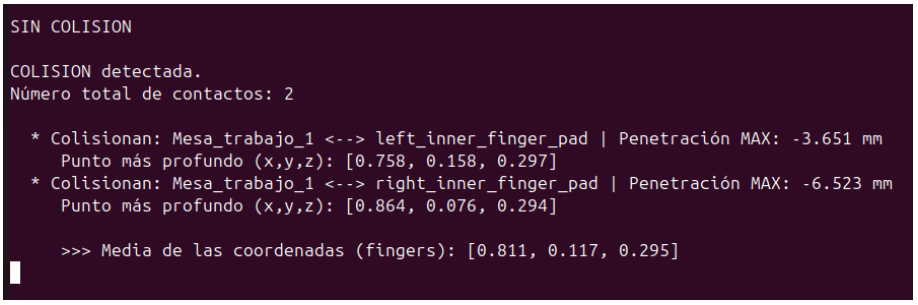

Figura 3: Impresión por cambio de estado a posición con colisiones.

Cada iteración del bucle termina con la publicación de información clave, como el número total de contactos detectados y las posiciones medias de los mismos. Esta información está disponible para otros nodos ROS, que pueden usarla para tareas de análisis, visualización o planificación. Adicionalmente, el nodo actualiza la posición de objetos manipulables en tiempo real, asegurando que el entorno simulado refleje siempre el estado más reciente de la escena de gazebo.

El nodo colisiones.cpp logra integrar de manera efectiva la detección de colisiones, el manejo del estado de sujeción y la sincronización en tiempo real entre Gazebo y RViz. Esta integración no solo permite al robot manipular objetos con precisión, sino que también garantiza una experiencia de simulación rica y detallada para el usuario.

4.2. Posiciones Goals RViz

Para facilitar las pruebas del programa de colisiones sin depender del control manual con el Omni, se han configurado nuevos goals predefinidos para el robot Kinova Gen3 dentro del archivo de configuración `gen3_robotiq_2f_140.srdf.xacro`. Este enfoque permite mover el brazo del robot a posiciones específicas directamente desde RViz, simplificando las pruebas del sistema.

El proceso comenzó identificando las posiciones necesarias para las pruebas, como la posición inicial para trabajar sobre nuestro mundo con posiciones cartesianas (`own_home`), las de recogida para cada objeto (`pick_[objeto]`), las de colocación (`place_[objeto]`) y una posición específica para evaluar solamente las colisiones (`collision`). Con el entorno de simulación de Gazebo y RViz activo, se lleva el brazo a estas posiciones de manera manual utilizando las herramientas de planificación de Moveit en RViz. Una vez en cada posición deseada, se obtienen los valores exactos de las articulaciones a través del topic `/my_gen3/move_group/goal`, que publica los datos de configuración del robot en tiempo real, entre ellos los 7 joints que describen la posición actual del robot.

Con los valores en mano, se edita el archivo xacro para añadir estas posiciones como nuevos `group_state`. A cada configuración se le asigna un nombre representativo, lo que facilita su identificación y uso. Por ejemplo, posiciones como `"pick_box_0"` o `çollision"` permiten a MoveIt interpretar y planificar movimientos hacia dichas posiciones de forma consistente.

```
1  <group_state name="pick_box_0" group="arm">
2       <joint name="$(arg␣prefix)joint_1" value="
   -2.444925654853244" />
3       <joint name="$(arg␣prefix)joint_2" value="
   -1.3845636574937572" />
4       <joint name="$(arg␣prefix)joint_3" value="
   -1.634581495077705" />
5       <joint name="$(arg␣prefix)joint_4" value="
   -2.1474202296064213" />
6       <joint name="$(arg␣prefix)joint_5" value="
   3.1161889730791152" />
7       <joint name="$(arg␣prefix)joint_6" value="
   0.11671086216088344" />
8       <joint name="$(arg␣prefix)joint_7" value="
   -0.0724531823170782" />
```

```
9     </group_state>
```

Para facilitar las pruebas del programa de colisiones sin depender del control manual con el Omni, se han configurado nuevos goals predefinidos para el robot Kinova Gen3 dentro del archivo de configuración `gen3_robotiq_2f_140.srdf.xacro`. Este enfoque permite mover el brazo del robot a posiciones específicas directamente desde RViz, simplificando las pruebas del sistema.

El proceso comenzó identificando las posiciones necesarias para las pruebas, como la posición inicial para trabajar sobre nuestro mundo con posiciones cartesianas (`own_home`), las de recogida para cada objeto (`pick_[objeto]`), las de colocación (`place_[objeto]`) y una posición específica para evaluar solamente las colisiones (`collision`). Con el entorno de simulación de Gazebo y RViz activo, se lleva el brazo a estas posiciones de manera manual utilizando las herramientas de planificación de Moveit en RViz. Una vez en cada posición deseada, se obtienen los valores exactos de las articulaciones a través del topic `/my_gen3/move_group/goal`, que publica los datos de configuración del robot en tiempo real, entre ellos los 7 joints que describen la posición actual del robot.

Con los valores en mano, se edita el archivo xacro para añadir estas posiciones como nuevos `group_state`. A cada configuración se le asigna un nombre representativo, lo que facilita su identificación y uso. Por ejemplo, posiciones como `"pick_box_0"` o `çollision"` permiten a MoveIt interpretar y planificar movimientos hacia dichas posiciones de forma consistente.

Finalmente, se validan los cambios reiniciando el entorno de simulación y verificando que las nuevas configuraciones estuvieran disponibles en RViz. Se prueba que el robot pueda moverse correctamente a cada goal y que el sistema de colisiones reaccione adecuadamente, incluyendo la publicación de mensajes de attach y detach según los eventos detectados.

Este enfoque optimizó las pruebas al eliminar la dependencia del Omni y proporcionó un entorno controlado y reproducible para validar el sistema de colisiones sin necesidad de teleoperación directa.

5. Diseño de la Interfaz de Teleoperación

El diseño de la interfaz de teleoperación está orientado a la simplicidad de uso y a proporcionar al operador toda la información necesaria para manejar el robot de manera eficiente y segura. La interfaz está dividida en diferentes secciones que presentan los datos de forma clara y organizada, facilitando la supervisión y el control del robot. A continuación, se describen las características principales de la interfaz y cómo se implementaron en el código.

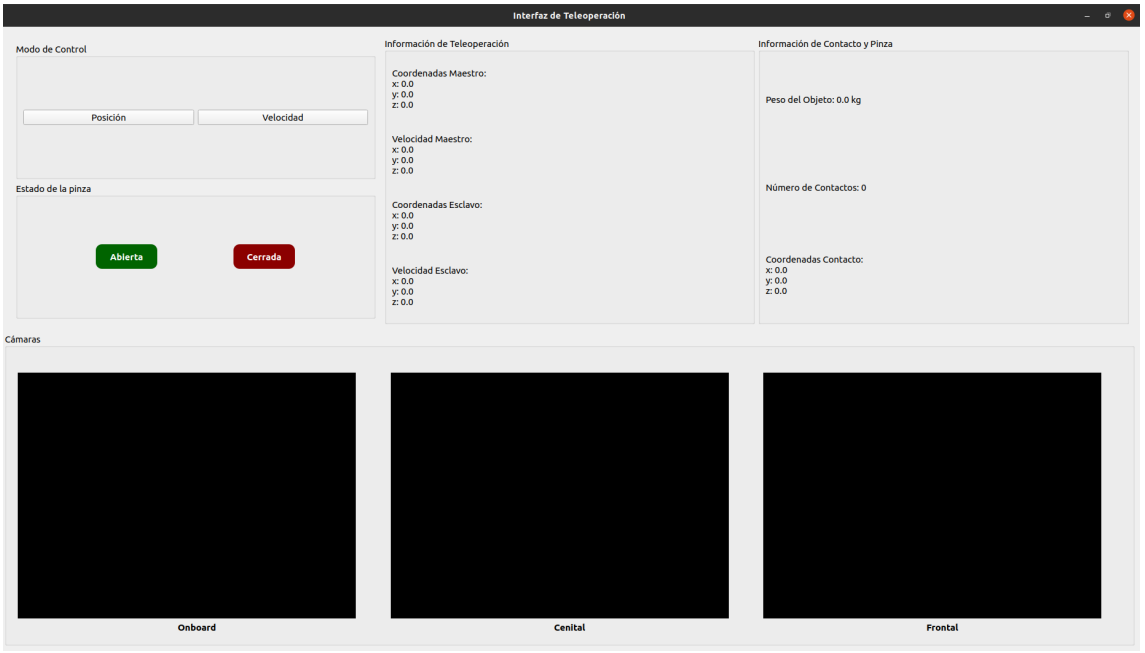

Figura 4: Panel de la interfaz diseñada.

5.1. Estructura de la Interfaz

La interfaz se compone de un diseño principal dividido en un panel superior y un panel inferior, implementado mediante el uso de un `QVBoxLayout` (véase el Código 7). Este diseño permite una disposición clara, separando la información de control e interacción en el panel superior y las visualizaciones de las cámaras en el panel inferior.

```
main_layout = QVBoxLayout()
central_widget.setLayout(main_layout)

# Panel superior
top_panel = self.create_top_panel()
main_layout.addWidget(top_panel)

# Panel inferior (C maras)
camera_panel = self.create_camera_panel()
main_layout.addWidget(camera_panel)
```

Listing 7: Definición del diseño principal de la interfaz.

El panel superior contiene tres secciones principales:

- Información del modo de teleoperación y maestro/esclavo.
- Información de contacto y estado de la pinza.
- Indicadores relacionados con coordenadas y velocidades.

El panel inferior está dedicado exclusivamente a la visualización de las cámaras, lo que permite al operador guiarse a través de las imágenes capturadas, dado que no cuenta con una visualización directa del robot remoto.

5.2. Panel Superior: Controles e Información del Robot

El panel superior contiene tres secciones principales que organizan la información en columnas claramente diferenciadas para facilitar el acceso a los datos más relevantes durante la operación.

- **Columna izquierda:** Información del modo de teleoperación y maestro/esclavo.
- **Columna central: Datos de teleoperación.** Presenta información detallada sobre las coordenadas y velocidades de los sistemas maestro y esclavo. Estas métricas son esenciales para evaluar la sincronización entre el dispositivo de control y el robot, asegurando una operación precisa.
- **Columna derecha: Información del entorno.** Proporciona detalles sobre el peso del objeto manipulado, el número de contactos detectados y sus coordenadas. Estos datos permiten al operador evaluar las condiciones físicas y ajustar el control en tiempo real.

Figura 5: Panel superior de la interfaz diseñada.

La organización en columnas separadas garantiza que el operador pueda centrarse en una categoría específica de información sin distracciones y optimiza la toma de decisiones.

5.2.1. Modo de control y LEDs.

La primera columna está organizada en dos grupos (`QGroupBox`) que separan la información de manera lógica. En el grupo de `Modo de Control`, se definen botones para alternar entre los modos de posición y velocidad (aunque no se haya implementado el control en velocidad, se ha planteado cómo sería la alternancia entre estos),véase el Código 8. Al presionar cada botón, se publica el modo correspondiente en el `topic` de ROS `/modo_teleoperacion` mediante el método `publish_mode`.

```
self.position_mode_btn = QPushButton("Posici n")
self.velocity_mode_btn = QPushButton("Velocidad")
self.position_mode_btn.clicked.connect(lambda: self.
    publish_mode("posicion"))
self.velocity_mode_btn.clicked.connect(lambda: self.
    publish_mode("velocidad"))
...
def publish_mode(self, mode):
    pub = rospy.Publisher("/modo_teleoperacion", String,
    queue_size=10)
    pub.publish(mode)
```

Listing 8: Implementación del modo de teleoperación.

Otro elemento clave es el grupo de `Estado de la Pinza`, donde se actualizan indicadores visuales (LEDs) que representan si la pinza está abierta o cerrada. Estos indicadores cambian su estado con base en los mensajes publicados en el `topic /estado_pinza`.

```
def update_gripper_state(self, msg):
    if msg.data.lower() == "abierta":
        self.led_open.setStyleSheet("background-color: lime; 
    color: black;")
        self.led_closed.setStyleSheet("background-color: 
    darkred;")
    elif msg.data.lower() == "cerrada":
        self.led_open.setStyleSheet("background-color: 
    darkgreen;")
        self.led_closed.setStyleSheet("background-color: red;"
    )
```

Listing 9: Actualización de estado de la pinza.

5.3. Panel Inferior: Visualización de Cámaras

El panel inferior permite la visualización de las imágenes capturadas por las cámaras onboard, cenital y frontal. Cada cámara se define dentro de un widget individual que incluye un título descriptivo y una etiqueta para mostrar las imágenes. Los mensajes de las cámaras se reciben a través de `topics` específicos, como `/my_gen3/robot_camera/onboard`, y se procesan utilizando la biblioteca `cv_bridge` para convertir las imágenes a formato OpenCV y posteriormente a Qt.

```
def callback_camera_onboard(self, msg):
    self.update_camera_image(self.onboard_camera_widget.
    findChild(QLabel), msg, rotate=True)

def update_camera_image(self, camera_label, msg, rotate=False)
    :
    cv_image = self.bridge.imgmsg_to_cv2(msg, "bgr8")
```

```
    if rotate:
        cv_image = cv2.rotate(cv_image, cv2.ROTATE_180)
    height, width, channel = cv_image.shape
    qt_image = QImage(cv_image.data, width, height, 3 * width,
    QImage.Format_RGB888).rgbSwapped()
    pixmap = QPixmap.fromImage(qt_image).scaled(550, 400, Qt.
KeepAspectRatio)
    camera_label.setPixmap(pixmap)
```

Listing 10: Recepción y visualización de imágenes de las cámaras.

Figura 6: Panel superior de la interfaz diseñada.

Cámara Onboard

Desde el archivo de configuración del robot Kinova Gen3 de 7 grados de libertad (`gen3_macro.xacro`), se ha habilitado la cámara que viene incorporada, pero no está configurada. Aunque los enlaces de la cámara ya existían en el archivo original de configuración, no estaban activados ni completamente funcionales dentro del modelo.

Para lograrlo, se diseñó y añadió la sección correspondiente en la configuración, permitiendo que la cámara integrada operase como un sensor activo. Esta integración incluyó parámetros detallados como el campo de visión horizontal, la resolución de la imagen generada, el formato de salida y los límites del rango de visualización (distancias mínima y máxima), intentando ser lo más realistas posibles teniendo en cuenta las especificaciones reales del robot.

La tarea no se limitó únicamente a añadir esta nueva configuración. Fue esencial ajustar la orientación de los enlaces relacionados con la cámara para asegurar que estuviera correctamente alineada en el modelo virtual, reflejando la posición y el ángulo que tendría en el robot físico.

Con la cámara ya configurada y funcional, el operador es capaz de manipular, gracias a esta cámara, objetos con mayor detalle mediante su incorporación en la interfaz. A continuación se muestra el código generado para su inclusión:

```
  <gazebo reference="${prefix}camera_link">
```

```
        <sensor type="camera" name="camera_sensor">
          <update_rate>30.0</update_rate>
          <camera>
            <horizontal_fov>1.3962634</horizontal_fov>
            <image>
              <width>640</width>
              <height>480</height>
              <format>R8G8B8</format>
            </image>
            <clip>
              <near>0.03</near>
              <far>300</far>
            </clip>
          </camera>
          <plugin name="camera_controller" filename="
libgazebo_ros_camera.so">
            <alwaysOn>true</alwaysOn>
            <updateRate>0.0</updateRate>
            <cameraName>robot_camera</cameraName>
            <imageTopicName>onboard</imageTopicName>
            <cameraInfoTopicName>camera_info</
cameraInfoTopicName>
            <frameName>${prefix}camera_link_optical</frameName
>
          </plugin>
        </sensor>
      </gazebo>
```

Listing 11: Cámara Onboard

5.4. Beneficios de la Organización Modular

La división de la interfaz en paneles y grupos específicos proporciona varios beneficios:

- **Claridad visual**: La separación de controles, indicadores y cámaras permite al operador acceder rápidamente a la información relevante.
- **Escalabilidad**: La estructura modular facilita la incorporación de nuevas funcionalidades o elementos de visualización en el futuro.
- **Reducción de errores**: Al presentar los datos de forma organizada, se minimiza la posibilidad de que el operador interprete información incorrecta.

En conjunto, el diseño modular y el uso de `topics` de ROS garantizan una experiencia de teleoperación fluida y eficiente.

6. Vídeo Demostración

Se puede ver un vídeo de demostración de la gran mayoría el funcionamiento de la práctica en este enlace.

7. Conclusiones

El desarrollo de esta práctica permitió diseñar e implementar un sistema de teleoperación que integra múltiples componentes clave, asegurando un control preciso y eficiente del brazo robótico en un entorno simulado. A continuación, se detallan las principales conclusiones obtenidas:

- **Diseño de la interfaz:** La interfaz desarrollada logró organizar de manera clara y modular toda la información necesaria para la teleoperación. Los paneles superiores e inferiores permitieron al operador monitorear los datos esenciales del robot y visualizar las imágenes capturadas por las cámaras del simulador, asegurando una interacción intuitiva. Además, la inclusión de indicadores visuales (LEDs) para el estado de la pinza y los controles de modo de operación mejoraron la accesibilidad de las funciones principales.

- **Control maestro-esclavo:** Se implementó un controlador maestro-esclavo que permitió sincronizar de manera eficiente los movimientos del dispositivo háptico *PHANTOM Omni* y el brazo robótico *Kinova Kortex*. Las diferencias cinemáticas entre ambos sistemas se resolvieron mediante técnicas de mapeo y escalado del espacio de trabajo, garantizando movimientos precisos y proporcionales.

- **Retroalimentación háptica:** La integración de retroalimentación de fuerza en el dispositivo maestro proporcionó al operador una percepción física del entorno simulado. Esto incluyó la implementación de mecanismos como la *Caja de Fuerza*, que permitió percibir los límites del espacio de trabajo, y el *Pozo de Gravedad*, que ofreció mayor estabilidad y alineación en tareas de precisión.

- **Prevención de colisiones:** Se diseñó un sistema de detección de colisiones y generación de fuerzas hápticas asociadas, lo que mejoró la seguridad operativa durante la manipulación de objetos. Los objetivos predefinidos configurados en *RViz* facilitaron las pruebas sin necesidad de control manual continuo, permitiendo evaluar el comportamiento del robot en diferentes escenarios.

- **Integración con simuladores:** El uso del simulador *Gazebo* y las herramientas proporcionadas por *ROS* permitió crear un entorno de pruebas realista con objetos dinámicos y estáticos. La posibilidad de modificar y personalizar el entorno de simulación, incluyendo cámaras y elementos manipulables, aportó un nivel adicional de realismo y flexibilidad al sistema.

- **Organización modular:** La estructura modular del diseño, tanto en la interfaz como en los algoritmos de control, facilitó la incorporación de nuevas

funcionalidades y permitió un desarrollo escalable. Esto asegura que el sistema pueda adaptarse fácilmente a futuros requisitos o aplicaciones.

- **Desempeño general:** Los resultados obtenidos mostraron un sistema de teleoperación robusto, capaz de responder de manera eficiente a las entradas del operador y manejar tareas de manipulación en un entorno simulado. La experiencia de usuario resultó ser fluida y segura, evidenciando la efectividad de los mecanismos implementados.

En conclusión, esta práctica proporcionó una experiencia integral en el diseño, implementación y evaluación de un sistema de teleoperación. Se cumplieron los objetivos planteados, abordando tanto los desafíos técnicos como los de interacción humano-robot. Este proyecto sienta las bases para la aplicación de teleoperación en entornos más complejos y reales, destacando la importancia de integrar simulación, diseño de interfaces y control avanzado en el desarrollo de sistemas robóticos.

8. Bibliografía

- https://wiki.ros.org/ROS/Tutorials/MultipleMachines
- https://github.com/JenniferBuehler/gazebo-pkgs/blob/master/gazebo_grasp_plugin/src/G
- https://moveit.github.io/moveit_tutorials/doc/pick_place/pick_place_tutorial.html
- https://moveit.github.io/moveit_tutorials/doc/bullet_collision_checker/bullet_collision_c
- https://github.com/ageofrobotics/Simulate_Your_Robot_Arm_In_ROS_Noetic/blob/ma
- https://github.com/LearnRoboticsWROS/cobot_test/blob/main/src/attach_detach_obje