ENEE 467: Robotics Project Laboratory
ROS 2 and Simulation
Lab 4

Learning Objectives

The purpose of this lab is to provide a hands-on introduction to the working with a ROS 2 enabled robotics simulator. After completing this lab students should be able to:

- Write ROS 2 launch files for loading projects consisting of multiple nodes
- Configure the ROS 2 Gazebo bridge
- Feel comfortable working with rigid body simulators
- Write a simple joint space motion planner and execute trajectories in simulation.

Simulation is paramount in the field of robotics. Researchers seek simulators that provide a good approximation of real world dynamics and sensing. Simulation enables safe prototyping and rapid data generation for machine learning that otherwise would be too costly to obtain with real world experimentation.

Lab Instructions

Follow the lab procedure closely. Some code blocks have been populated for you and some you will have to fill in yourself. Pay close attention to the lab Questions (highlighted with a box). Answer these as you proceed through the lab and include your answers in your lab writeup. There are several lab Exercises at the end of the lab manual. Complete the exercises during the lab and include your solutions in your lab writeup.

List of Questions

List of Exercises

How to Read This Manual

Throughout the manual, boxes beginning with a **Question #** title contain questions to be answered as you work through the lab:

> **Question 0.** A question for you.

Boxes with a ✎ symbol (like the one below) are extra theoretical or implementation details provided for your enrichment and insight. We recommend that you read them during the lab:

> ✎ Some extra context and info.

Terminal commands and code blocks will be set in gray-colored boxes (usually for Python syntax)

```python
import math # some important comment you should totally read
```

while terminal outputs will be set in yellow-colored boxes

```
some terminal output
```

Files and directories will be set in magenta color with a teletype font, e.g., `~/some/dir/to/a/file.txt`, code variable names and in-text commands will be set in black teletype font, e.g., `a = [0, 0, 0]`, while Python type hints will be set in green teletype font, e.g., `save(path: str)` signifies that the argument to the `save` function is a string. Lastly, figure captions go below figures, table captions appear above tables, and links to an external online reference are highlighted in blue, eg. see: Wikipedia.

1. Lab Procedure

1.1 Prerequisites

The software for this lab is packaged and distributed as a Docker container. Ensure that the lab-4 Docker image is build on your system:

```
docker image ls
```

and check to see that the following line is included in the output:

```
lab-4-image    latest    0d1b7b7788f3    2 days ago        8.08GB
```

Next, download the code for this lab from the lab-4 repository:

```
cd ~/Labs
git clone git@github.com:ENEE467-F2025/lab-4.git
cd lab-4
```

Finally, launch the Docker container with the following commands:

```
cd docker
docker compose -f lab-4-compose.yml run --rm lab-4-docker
```

You should be greeted with the following prompt indicating you have a terminal inside the active container:

```
robot@desktop:~$
```

where robot is the name of the logged user and desktop is the name of the virtual machine (which may be different in your case).

This lab makes use of the Gazebo Simulator and ROS 2 Jazzy. The Docker images are preconfigured with both ROS 2 and Gazebo Harmonic. To verify your Gazebo installation, run the following command:

```
gz sim
```

which should open the Gazebo quick start menu. Ensure that the version in the upper right corner reads "Harmonic." To make sure Gazebo finds all models needed for this lab, confirm that the GZ_VERSION and GZ_SIM_RESOURCE_PATH environment variables are set by running the following:

```
printenv | grep -E '^(GZ_VERSION|GZ_SIM_RESOURCE_PATH)='
```

which should print both environment variables to the terminal:

```
GZ_SIM_RESOURCE_PATH=/home/robot/ros2_ws/src/ur3e_robotiq/ur3e_robotiq_gz/\
models:/opt/ros/jazzy/share
GZ_VERSION=harmonic
```

1.2 Gazebo EGL Warnings

On startup, Gazebo may output a "failed to create dri3 screen" warning. You may safely ignore the warning, as it a feature and not a bug (see: EGL Warnings).

2. Gazebo Primer

Gazebo is a physics-based robot simulator. It lets us test robot code safely without the real hardware, and is the de facto simulator integrated with ROS 2. There are three primary ways of interacting with the Gazebo simulator: via the GUI (see: Understanding the GUI), via the command line interface (CLI), `gz`, or via ROS 2. While the GUI is useful for visualization, adding shapes, and deleting models, many useful tasks, such as debugging, topic introspection, or automated experiments, are more efficiently handled from the command line, thus we will use it more extensively than the GUI.

2.1 Gazebo Models & Worlds

Gazebo uses model files to represent robots, objects, and entire worlds. These files may define physical properties, kinematics, and sensors, and may be expressed in the Unified Robot Description Format (which we covered in Lab 2), or in the Simulation Description Format (SDF), an XML-like format used primarily in Gazebo to describe robots and environments for simulation. Both formats describe robots, but they are designed for different tools and purposes. Think about which format a ROS 2 visualization tool like RViz would be able to read.

Inspect the `cuboid.sdf` file available in the `models` folder of the `simple_spawner` ROS 2 package for a simple SDF description, but note that SDF files are generally more verbose, and may contain entire environments, with sensors, light sources, physics, robots, animated human agents, etc.

> **Question 1.** Suppose you're developing software for your new robot and need to display a model of your robot in RViz via the `robot_state_publisher`. In what format should your robot models be described (URDF, SDF, or both)? Provide your answer and a short justification in your report. **Hint**: See Robot State Publisher.

2.2 The Gazebo Command Line Interface

Gazebo also provides a command line interface (CLI) via the `gz` command, allowing you to launch worlds, pause or step simulations, and interact with models without the GUI. To try it out, create a new `tmux` session and start Gazebo with a simple world using:

```
gz sim shapes.sdf -r
```

The `-r` flag starts the simulation immediately, equivalent to pressing the play button in the GUI. To see a list of commands supported by the `gz` CLI, in a new `tmux` pane, run:

```
gz help
```

To display more information about a specific command, run `gz <command> help`, where `<command>` is any command from the list returned above. The range of functionalities provided by the `gz` CLI is quite extensive, so we will only highlight a few use-cases below.

2.2.1 Messages

Gazebo also defines its own message types separate from ROS 2, although the underlying objects each message type represents are fundamentally the same. To list all Gazebo message types, run

`gz msg -l`. For a specific message, use the `-i <gz_msg_type>` flag. Try getting the message type for the `gz_msgs.Pose` Gazebo message type (often sued when spawning or teleporting models):

```
gz msg -i gz_msgs.Pose
```

You should see the following output printed to shell:

```
name: gz.msgs.Pose
File: gz/msgs/pose.proto

message Pose {                      # Start of Gazebo message definition
  .gz.msgs.Header header = 1;       # format: <data_type> <field_name>;
  string name = 2;                  # <- standard data types; #s are immaterial
  uint32 id = 3;
  .gz.msgs.Vector3d position = 4;        # . marks field names that are
  .gz.msgs.Quaternion orientation = 5;} # also gz messages
```

We will use this schema shortly when publishing a `Pose` message to position a model at spawn time. For now, note the required fields: `position` is a `Vector3d` containing x, y, and z, while `orientation` is a `Quaternion`.

> **Question 2.** Which fields are defined in the `gz.msgs.Header` message type? (**Hint**: `gz msg`)

2.2.2 Topics

Gazebo topics are independent of ROS 2, though they function similarly. With the simulator running (use the `shapes.sdf` world), list available Gazebo topics from a `tmux` pane using:

```
gz topic -l
```

Gazebo topics can be probed for their publishers and subscribers using:

```
gz topic -i -t <topic_name>
```

Try inspecting the `/world/shapes/pose/info` topic by running the previous command:

```
Publishers [Address, Message Type]:
  tcp://10.174.49.59:44563, gz.msgs.Pose_V
No subscribers on topic [/world/shapes/pose/info]
```

Here, we see that Gazebo publishes a message of type `gz.msgs.Pose_V` (a list of poses) to the topic but there are no subscribers currently listening. Topic data can be viewed directly using

```
gz topic -e -t <topic_name> -n <num>
```

which prints `<num>` messages and then exits (omit `-n` for a continuous stream). Using `-n 1` on the pose topic confirms that one time-stamped dictionary of all model poses is published.

> **Question 3.**
>
>   a. What is the average publishing rate of the `world/shapes/pose/info` topic? **Hint**: Run `gz help topic` to see a list of available options for topic introspection.
>
>   b. **True or False?** All Gazebo topics and their respective messages can be accessed via the `gz` CLI only when the simulator is running and unpaused.**Hint:** Run `gz sim shapes.sdf`, then list topics. Pause and unpause the simulation to observe the difference between topic existence and live message updates.

2.2.3 Publishing Gazebo Messages over Gazebo Topics

In Gazebo, we can publish to topics using the command:

```
gz topic -t <topic_name> -m <msg_type> -p '<message_content>'
```

Terminate all running Gazebo sessions, and run `gz topic -e -t /chatter`. This will wait for incoming Gazebo messages. In another terminal publish a simple string message using:

```
gz topic -t /chatter -m gz.msgs.StringMsg -p 'data: "Hello, Gazebo!"'
```

Monitor the previous (chatter) pane. You should see the text "`Hello, Gazebo!`".

> ✎ Gazebo sometimes does not terminate cleanly; check with `ps aux | grep gz`.

2.2.4 Gazebo Services

Gazebo services provide specific functionality using a request-response model. Open an empty world in Gazebo:

```
gz sim empty.sdf
```

Then check for a list of available services by running `gz service -l` in another `tmux` pane. This should display a list of Gazebo services namespaced by the Gazebo component they affect. Filter the list of services for those with a `create` string (Tip: use `gz service -l | grep create`). Note that Gazebo services are always namespaced under the world they belong to. So if the currently loaded Gazebo world is called `empty`, the fully qualified service is `/world/empty/create`, with `create` being the service endpoint. Check the service's request and response message types uisng:

```
gz service -i -s /world/empty/create # or gz service -is /world/empty/create
```

You should see the following info printed to the terminal

```
Service providers [Address, Request Message Type, Response Message Type]:
  tcp://aa.bbb.cc.dd:eeeee, gz.msgs.EntityFactory, gz.msgs.Boolean
```

indicating that this service's request message is of type `EntityFactory` and the response message is of type `Boolean`, which indicates success (`true`) or failure (`false`). Inspecting the `EntityFactory`

message reveals that it can be defined via raw SDF, an SDF file path, or a Gazebo model:

```
EntityFactory {
  header = 1;
  oneof from {
    sdf=2; sdf_filename=3; model=4; ... }
  pose=7;}
```

```
Boolean {
  header=1;
  data=2;}
```

gz msg -i gz.msgs.EntityFactory   gz msg -i gz.msgs.Boolean

Let's try calling this service to see what functionality this Gazebo service provides. Close all running Gazebo processes, and then change directory to the workspace root using `cd ~/ros2_ws`. Once there, split your `tmux` window into two panes. In one, start the `empty` Gazebo world. Then call the create service in the other by passing the `cuboid.sdf` from Section 2.1:

```
gz service -s /world/empty/create \
  --reqtype gz.msgs.EntityFactory \
  --reptype gz.msgs.Boolean \
  --timeout 1000 \
  --req "sdf_filename: '/home/robot/ros2_ws/src/simple_spawner/models/cuboid.sdf'
name: 'cuboid'"
```

You should see the response message "`data:   true`", and a red cuboid should be spawned in the GUI. The GUI's Entity Tree panel should also list the model's name, i.e., `cuboid`.

Another useful service is the `/world/<world_name>/set_pose` service, which allows us to set the pose of an existing entity in the simulation. Let's see how it works. Still with the empty Gazebo world open and the cuboid spawned, check the service info for the `/world/empty/set_pose` service. This should yield a request type of `gz.msgs.Pose` and a response type of `gz.msgs.Boolean`. Using this information and recalling the definition of the `Pose` message from Section 2.2.1, we can call the service to translate the cuboid to the position (x=1.0, y=0.5, z=0.1) in world coordinates, with no rotation:

```
gz service -s /world/empty/set_pose \
--reqtype gz.msgs.Pose \
--reptype gz.msgs.Boolean \
--req 'name: "cuboid",'\
' position: {x: 0.01, y: 0.1, z: 0.1},'\
' orientation: {x: 0.0, y: 0.0, z: 0.0, w: 1.0}'
```

Verify that the command updates the cuboid's pose. You will need to keep the simulator in view to see the cube teleport. The simulation does not need to be running for the transform to apply, but effects such as gravity or collisions will only be visible when the simulator is unpaused.

Question 4.

    a. What is the command to start the Gazebo simulator without a GUI? **Hint**: Run `gz help sim`, and locate the available option for running the simulator in headless mode.

    b. A model has been spawned at an incorrect pose in a Gazebo world. Describe at least two methods you could use to correct its position and orientation.

---

3. Gazebo-ROS 2 Interoperability

In this section, we will investigate some of the ROS 2 tools that enable interoperability between ROS 2 and Gazebo, including but not limited to starting the simulator, initializing the Gazebo-ROS 2 bridge, spawning objects, etc. Of these, we will explore the following ROS 2 packages:

  i. the `ros_gz_bridge` package
 ii. the `ros_gz_sim` package
iii. the `ros_gz_interfaces` service definition package, which we'll use in Exercise 1.

3.1 Gazebo-ROS 2 Bridge

The Gazebo bridge (implemented within the `ros_gz_bridge` ROS 2 package; see here) allows topics and services to flow between Gazebo and ROS 2, such as joint states, sensor data, and robot commands. It provides interfaces for configuring which Gazebo topics to expose to ROS 2. There are three directions of information flow: ROS 2 to Gazebo (`ROS_TO_GZ`), Gazebo to ROS 2 (`GZ_TO_ROS`), or both ways (`BIDIRECTIONAL`). We can configure the bridge using one of two methods: manually via ROS 2 cli tools, or using a ROS 2 launch file, with an optional YAML file specifying the bridged topics and directions.

3.1.1 Launching the Bridge Manually

The general syntax for initializing the Gazebo bridge is as follows:

```
ros2 run ros_gz_bridge parameter_bridge <gz_topic>@<ros_msg_type>@<gz_msg_type>
```

where the `@` symbol **after** the ROS 2 message type may change depending on whether the bridge is bidirectional (`@`), `GZ_TO_ROS` (`[`), or `ROS_TO_GZ` (`]`).

With the Gazebo simulator running (use the `empty.sdf` world), verify that the `/world/empty/clock` is available by listing the Gazebo topics. Then check available ROS 2 topics by running `ros2 topic list` (you should see the `/clock` topic). Once both topics are available, try bridging the Gazebo `/world/empty/clock` topic and the ROS 2 `/clock` topic using the following command (remember to verify the message types for the bridged ROS 2 and Gazebo topics by running their respective CLI commands):

```
ros2 run ros_gz_bridge parameter_bridge \
  /world/empty/clock@rosgraph_msgs/msg/Clock@gz.msgs.Clock
```

You should see the following output indicating that the bridge has been initialized:

```
Creating GZ->ROS Bridge: [/clock (gz.msgs.Clock) \
-> /clock (rosgraph_msgs/msg/Clock)]
```

In another terminal, issue a `ros2 topic list` command. Notice that now the /clock topic has been modified to the `/world/empty/clock` topic. This means we can stream messages broadcasted on this Gazebo topic directly from ROS 2. Verify this by running

```
ros2 topic echo --once /world/empty/clock
```

which should print a single clock message indicating the current time in seconds and nanoseconds.

We can also bridge multiple topics at a time using the `ros2 run` command. With Gazebo running the `empty.sdf` world, split your `tmux` session to create four extra panes. In one pane, spawn the cuboid from Section 2.2.4:

```
gz service -s /world/empty/create \
  --reqtype gz.msgs.EntityFactory \
  --reptype gz.msgs.Boolean \
  --timeout 1000 \
  --req "sdf_filename: '/home/robot/ros2_ws/src/simple_spawner/models/cuboid.sdf'
name: 'cuboid'"
```

In a second pane, run `gz topic -l | grep cuboid` to locate the cuboid's pose topic. Verify that the `grep` command yields: /model/cuboid/pose. Then bridge the /world/empty/clock and cuboid Gazebo topics to their respective ROS 2 topics using the following command (note the line continuation character `\` separating each bridge directive):

```
ros2 run ros_gz_bridge parameter_bridge \
  /world/empty/clock@rosgraph_msgs/msg/Clock@gz.msgs.Clock \
  /model/cuboid/pose@tf2_msgs/msg/TFMessage@gz.msgs.Pose_V
```

Lastly, run `ros2 topic list` in the third pane, and verify that both topics are visible indicating that the Gazebo bridge is operational:

```
/world/empty/clock
/model/cuboid/pose
```

Try echoing messages from the /model/cuboid/pose topic:

```
ros2 topic echo --once /model/cuboid/pose
```

You should see a list of `TransformStamped` ROS 2 messages, in the terminal, with the `frame_id` highlighting the spawned object.

Question 5. Suppose multiple models (named `model_a`, `model_b`, and `model_c`) have been spawned in your Gazebo world, and a ROS 2–Gazebo bridge has been configured for each. If you want to determine the pose of a specific model, which field in the corresponding `TransformStamped` ROS 2 message should you examine? **Hint**: `ros2 interface show tf2_msgs/msg/TFMessage`.

3.1.2 Launching the Bridge using YAML Configuration Files

When bridging multiple topics, it is often easier to configure the Gazebo bridge using a YAML file. A configuration file, shown below in abridged form, is provided as `bridge.yaml` in the `config` folder of the `simple_spawner` ament_python package located in the `src` directory.

```
- ros_topic_name: "/clock" # global ns
  gz_topic_name: "/world/empty/clock"
  ros_type_name: "rosgraph_msgs/msg/Clock"
  gz_type_name: "gz.msgs.Clock"
  direction: GZ_TO_ROS  # BIDIRECTIONAL or ROS_TO_GZ #
```

```
# ...
```

After confirming that the `bridge.yaml` file exists, to initialize the bridge using this YAML file, terminate all running processes. Then `cd` to the workspace root, and start an empty Gazebo world

```
cd ~/ros2_ws
gz sim empty.sdf
```

Next, in a new `tmux` pane, run the following command:

```
ros2 launch ros_gz_bridge ros_gz_bridge.launch.py bridge_name:=ros_gz_bridge \
config_file:="/home/robot/ros2_ws/src/simple_spawner/config/bridge.yaml"
```

The `ros_gz_bridge` node should print messages indicating that the bridge has successfully started. **Note**: only topics with matching message interfaces can be bridged. For instance, a ROS 2 message of type `geometry_msgs/msg/Pose` can be bridged with the `Pose` Gazebo message, but not with the `Pose_V` Gazebo message. A compatibility table is provided at the following README.

Table 1: Rule of thumb for Gazebo–ROS 2 bridge directions.

Topic / Data	Bridge Direction
Clock, joint states, camera image, depth, TFs	GZ_TO_ROS
Joint / trajectory commands	ROS_TO_GZ
Status / feedback topics	BIDIRECTIONAL

Question 6. A robot has an RGB-D camera that publishes color images as ROS 2 messages on the topic `/rgbd_camera/image` of type `sensor_msgs/msg/Image`. If the corresponding Gazebo topic is `/world/default/camera/color/image_raw`, write a one-line `ros2 run` command using `ros_gz_bridge` to bridge the Gazebo topic to the ROS 2 topic. **Hint**: See the compatibility table available here.

3.2 ROS 2 Launch Files

We can also use ROS 2 Python launch files (see: Launch Files) to simplify ROS2-Gazebo interoperation. As ROS 2 launch files are extensive, we do not cover it in its entirety here.

Inside the `spawner.launch.py` file within the `launch` folder of the `simple_spawner` package, we have provided code for a simple launch file that loads an empty world in Gazebo, spawns the cuboid from the aforementioned sections, and sets up the bridge. To run the launch file, terminate all running Gazebo processes and ROS 2 nodes (keep the container running), then `cd` to `~/ros2_ws`:

```
cd ~/ros2_ws
```

Then build the `simple_spawner` package with `colcon`, and source your workspace:

```
colcon build --symlink-install --packages-select simple_spawner
source install/setup.bash
```

Finally, create one extra `tmux` pane, and run the launch file using:

```
ros2 launch simple_spawner spawner.launch.py
```

This launch file will start Gazebo, spawn the cuboid, and setup the bridge. You should also see an RViz window displaying the pose of the cuboid as a TF marker. Try moving the cuboid in the simulator using the Move button (✛) while the launch file is running. You should see the RViz markers update with each move.
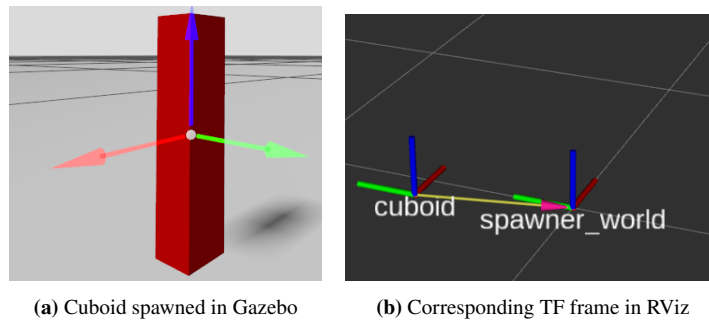

(a) Cuboid spawned in Gazebo   (b) Corresponding TF frame in RViz

Figure 1: Verification of Gazebo–ROS TF integration for the Cuboid Example

3.3 The SpawnEntity ROS 2 Service

ROS 2-Gazebo interopability can also be achieved via ROS2 services, namely: In a `tmux` pane within the Docker container, run

```
ros2 interface show ros_gz_interfaces/srv/SpawnEntity
```

Notice the request and response message types of the service (`EntityFactory`) and `bool`, respectively). Terminate all running Gazebo processes, and in a new `tmux` pane, run:

```
cd ~/ros2_ws
colcon build --symlink-install --packages-select simple_spawner
source install/setup.bash
```

Then start the launch file to start Gazebo advertise the `/world/empty/create` entity Gazebo service, which automatically makes the Gazebo spawn service available (see Section section 2.2.4):

```
ros2 launch simple_spawner spawner.launch.py spawn_cuboid:=false
```

Finally, in another pane, call the `SpawnEntity` service to spawn a green sphere in the loaded world:

```
ros2 service call /world/empty/create ros_gz_interfaces/srv/SpawnEntity "{
  entity_factory: {
    name: 'sphere',
    sdf: '$(< /home/robot/ros2_ws/src/simple_spawner/models/sphere/model.sdf)',
    allow_renaming: true,
    pose: {
      position: { x: 0.21, y: 0.14, z: 0.1 },
      orientation: { x: 0.0, y: 0.0, z: 0.0, w: 1.0 }
    }
```

11

```
  }
}"
```

Initialize the specific bridge for the sphere object:

```
ros2 run ros_gz_bridge parameter_bridge \
  model/sphere/pose@tf2_msgs/msg/TFMessage@gz.msgs.Pose_V
```

Then try echoing for topics on the ROS 2 side

```
ros2 topic echo --once /model/sphere/pose
```

Verify that a list of transforms is printed to shell, with the frame_id highlighting the specific object spawned in Gazebo

```
header:
    stamp:
      sec: 631
      nanosec: 991000000
    frame_id: sphere
  child_frame_id: sphere/link
```

This confirms that the bridge works, and Gazebo and ROS 2 are linked. Great! However, as you can imagine, while the launch file and `SpawnEntity` ROS 2 service are both useful for bootstrapping a simulation session via the terminal, once loaded, the world remains fixed, and adding new objects requires manually issuing *long* CLI commands. This quickly becomes tedious and error-prone, especially if we want to spawn several objects. In Exercise 1, we will develop a solution to handle the spawning of objects *programmatically*, providing a simple CLI interface for user interactivity.

Exercise 1. Interactive Object Spawner (25 pts.)

This exercise builds a fully programmatic Python node (`simple_spawner.py`) contained in an `ament_python` package with the same name that allows:

- Selecting and spawning objects at runtime
- Immediately broadcasting TF frames for RViz visualization

Deliverables: Complete and test the `SimpleSpawner` ROS 2 node class by executing the following steps:

Gazebo Spawner Functionality

a. Complete the class method `_call_spawn_service` responsible for setting up a service request for spawning entities (objects) at specific poses in Gazebo. The class method should create an instance of a `SpawnEntity.Request` object called `request` based on the selected model name from the user (stored in the `model_name` class variable). The `entity_factory` field of `request` instance should have the following data:

    i. name should be set to `self.model_name`

12

      ii. `allow_renaming` should be set to `True`

      iii. `sdf` should be set to the input argument of the `_call_spawn_service` method.

b. After filling the `request` instance, the class method should then send the filled service request by setting the variable `future` to the `call_async` method of the `spawn_client` class method

```
future = self.spawn_client.call_async(request)
```

This sends the request to the `/spawn_entity` service that will spawn the object.

c. Explain why it is necessary to ensure that the service call is non-blocking. What would happen if we didn't send the request asynchronously? **Hint**: see Asynchronous Calls.

Once we obtain the pose of the spawned model via the bridge, we need to convert it to an $SE(3)$ object using ROS 2 interfaces as follows. Let $T \in SE(3)$ represent the target transformation, and let $(R^{\text{model}}, p^{\text{model}})$ be the pose of the model to be spawned, expressed in terms of its rotational and translational components. Then

$$T = \begin{bmatrix} R^{\text{model}} & p^{\text{model}} \\ 0_{1\times 3} & 1 \end{bmatrix}, \quad \text{with } p^{\text{model}} = \begin{bmatrix} x^{\text{model}} \\ y^{\text{model}} \\ z^{\text{model}} \end{bmatrix}, \tag{1}$$

and $R^{\text{model}} = (x^{\text{model}}, y^{\text{model}}, z^{\text{model}}, w^{\text{model}})$ is the orientation expressed as a 4-tuple quaternion.

Transform Publication

d. Create a class variable in the class `__init__` method `br`, and set it to a `TransformBroadcaster` object, passing as argument to the broadcaster, the current instance of the ROS 2 Node class, `self`. The line to modify is highlighted with a `TODO` flag within the `__init__` method

e. Modify the class method `broadcast_tf` that sends the transform (`T`) representing the spawned object's pose as it becomes available. The sent pose is stored in the `model_pose` class variable as a `PoseStamped` ROS 2 message. Fill `T` with particulars from `model_pose` using (1), by matching `model_pose`'s message fields to R^{model} and p^{model}.

Test Your Spawner Node:

Once done with your implementations, test your spawner node by building the `simple_spawner` package

```
cd ~/ros2_ws
colcon build --symlink-install --packages-select simple_spawner
source install/setup.bash
```

Open a terminal (or tmux pane), and start the Gazebo simulator:

```
ros2 launch simple_spawner spawner.launch.py spawn_cuboid:=false
```

In a separate terminal, run the spawner node (make sure `spawn_cuboid=false`):

```
ros2 run simple_spawner simple_spawner_node
```

You should see the following prompt:

```
Select an object to spawn:
  1) Cube
  2) Cylinder
  3) Sphere
Enter choice (1-3):
```

A temporary RViz window will also pop up. Select an object (1, 2, or 3; ignore the RViz OpenGL warnings and still enter your selection in the active terminal). If your implementation above is correct, the selected object will appear in Gazebo, and its TF frame will be broadcast to RViz (see: Figure 2) for one example. Moving the object in Gazebo should update the transforms in the open RViz window. To spawn a new object, you will need to restart the node (and not Gazebo). Since Rviz processes are created on demand, terminating the `simple_spawner_node` will also terminate `rviz2`. Hence, only one object's transform can be visualized at any given instance.

To complete this exercise, provide the following files in addition to your report:

  i. The complete implementation of your `simple_spawner.py` file.

  ii. A screenshot of the Gazebo and RViz winidow for all three cases (Option 1, 2, and 3), i.e., that means you will provide three figures, one for each object.
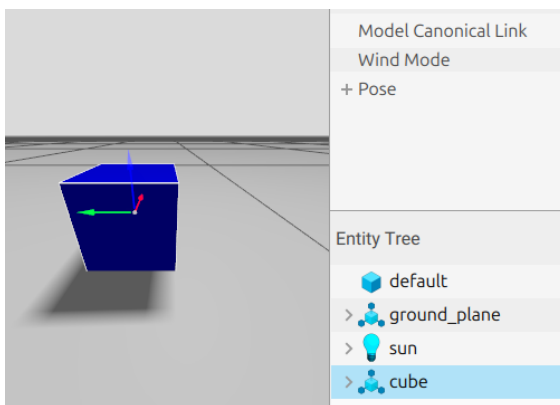
(a) Spawned cube in Gazebo
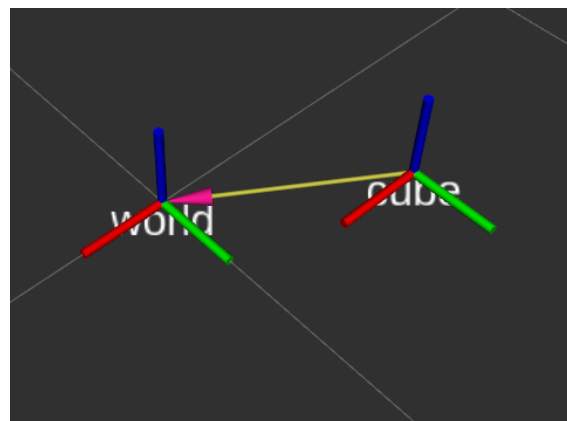
(b) Corresponding TF frame in RViz

Figure 2: Expected Output: The cube object is spawned in Gazebo and its pose is visualized in RViz via TF markers.

4. Simulating the UR3e-HandE Robot with Gazebo

4.1   Building the Required Packages

Three packages containing subpackages you will need for this unit are provided for you:

1. `robotiq_hande` which contains the description package for the Robotiq Hand-e gripper

2. `ur3e` which contains the description package for the UR3e manipulator

3. and `ur3e_robotiq` which contains a description and simulation package for the combined manipulator/gripper system.

Change directories back to the root of the workspace and build the packages:

```
cd ~/ros2_ws
colcon build --symlink-install
source install/setup.bash
```

With the packages built and sourced, we can begin investigating their contents.

4.2   Robot Description Packages

Open the package `robotiq_hande/robotiq_hande_description` and take a look at the contents:

```
config  launch  rviz  urdf  CMakeLists.txt  package.xml
```

The most important folders here are the ones titled `launch` and `urdf`. These two folders contain the code for *launching* the package and the kinematic, geometric, and inertial description of the gripper respectively.

Launch the gripper description package with the following command:

```
ros2 launch robotiq_hande_description view_robot.launch.py
```

Two windows will appear, a Joint State Publisher GUI and an RViz window. RViz provides a 3D visualization for various topics and data accessible to the robot via the ROS 2 network. RViz is not a simulator, rather an interactive tool to visualize the current "state" of the robot. Try playing around with the Joint State Publisher GUI, notice that changes made to the joint state are reflected in the robot's state which can be seen in the RViz window.

> **Question 7.**   What are the joint limits for the Robotiq Hand-E Gripper? Note that all units use the SI convention (meters, kilograms, newtons).

To terminate the description package, close the RViz window or enter `Ctrl + C` in the launch terminal. Description packages are provided for both the UR3e and the combined manipulator-gripper system. To run them use the following commands:

```
# Description package for the UR3e
ros2 launch ur3e_description view_robot.launch.py
```

```
# Description package for the combined system
ros2 launch ur3e_robotiq_description view_robot.launch.py
```

> **Question 8.** The UR3e has six active joints. What are the names of these six joints in the robot's description? (**Hint:** This information is available in the robot's URDF, though the Joint State Publisher GUI also displays the joint names).

5. Writing a Simple Joint Space Planner

5.1 ROS2 JointTrajectory Message Interface

In ROS 2, to specify C-space trajectories, we use the `JointTrajectory.msg` provided by the `trajectory_msgs` package. To inspect its fields, open a `tmux` pane and run:

```
ros2 interface show trajectory_msgs/msg/JointTrajectory --no-comments
```

Observe that each `JointTrajectory.msg` is a time-stamped list of joints (`joint_names`), a list of points along the trajectory (`points`), and the time that each point should be reached relative to the start, i.e.:

$$\frac{t_f - t_0}{N} \text{ or just } \frac{t_f}{N} \text{ if } t_0 = 0.0,$$

here N equal to the length of the `points` field):

```
std_msgs/Header header
        builtin_interfaces/Time stamp
        string frame_id
string[] joint_names
JointTrajectoryPoint[] points
        float64[] positions
        float64[] velocities
        ...
        builtin_interfaces/Duration time_from_start
```

By further inspection of the content of the `points` field, we can see that each element in the list is a `JointTrajectoryPoint` message, specifying the position, velocity, acceleration, and effort of all actuated joints on the robot.

> **Question 9.** If we require the UR3e to move smoothly through 5 waypoints, how many `JointTrajectoryPoints` should our `JointTrajectory` message contain?

5.2 Computing the Desired Joint Configuration via IK

Let the desired end-effector pose be denoted as the $SE(3)$ transformation $T_{\text{goal}} \in SE(3)$. We can decompose it into translational (T_1) and rotational (T_2) components:

$$T \equiv T_{\text{goal}} = T_1 \cdot T_2. \tag{2}$$

The translational component, T_1, is given by

$$T_1 = \begin{bmatrix} 1 & 0 & 0 & x_{\text{goal}} \\ 0 & 1 & 0 & y_{\text{goal}} \\ 0 & 0 & 1 & z_{\text{goal}} \\ 0 & 0 & 0 & 1 \end{bmatrix}, \tag{3}$$

where x_{goal}, y_{goal}, and z_{goal} are the inertial x, y, and z components of T_{goal}'s translation vector, and

$$T_2 = \text{SE}(3)(\mathring{q}_{\text{goal}}) = \begin{bmatrix} \mathcal{R}(\mathring{q}_{\text{goal}}) & 0 \\ 0^\top & 1 \end{bmatrix}, \quad \mathring{q}_{\text{goal}} = s\langle v_x, v_y, v_z \rangle, \quad s^2 + v_x^2 + v_y^2 + v_z^2 = 1 \tag{4}$$

where $\mathring{q}_{\text{goal}}$ is a unit quaternion (see: Unit Quaternion) encoding is the desired rotation, $s \in \mathbb{R}$ is the scalar part of the quaternion (i.e., `pose.orientation.w` in ROS 2), $v = [v_x, v_y, v_z]^\top$ is the vector part of the quaternion, and $\text{SE}(3)(\cdot)$ is shorthand for an operation that casts the unit quaternion to a rotation-only homogeneous transform. Solving the inverse kinematics (IK) for T_{goal} then yields the desired joint configuration q_f.

5.3 Planning a Linear Joint-Space Trajectory

Given the current joint configuration q_0 and the goal configuration q_f, a simple linear interpolation in joint space produces the trajectory. For a total of N trajectory points over a duration T:

$$\Delta t = \frac{T}{N}, \tag{5}$$

where the time stamp for the i-th trajectory point is calculated as:

$$t_i = i \cdot \Delta t, \quad i = 0, 1, \ldots, N - 1 \tag{6}$$

The joint positions at each trajectory point are computed as:

$$q_i^j = q_0^j + \frac{i}{N - 1}\left(q_f^j - q_0^j\right), \quad j = 1, \ldots, n_{\text{joints}} \tag{7}$$

ensuring a smooth path in joint space, with each trajectory point associated with a `time-from-start` field corresponding to its relative position along the line.

NumPy provides a handy function for linear interpolation via the `numpy.linspace` method. Create a small test python file somewhere in the `src` folder, and add the following code block to it:

```python
import numpy as np

start = 0.0
end = 10.0
N = 20

t = np.linspace(start, end, N)

# Step size between consecutive points
dt = t[1] - t[0]
print("Step size:", dt)
```

Try playing with a few vlues of `N`, and observe how the step size (`dt`) changes. We are now ready to write our simple planner.
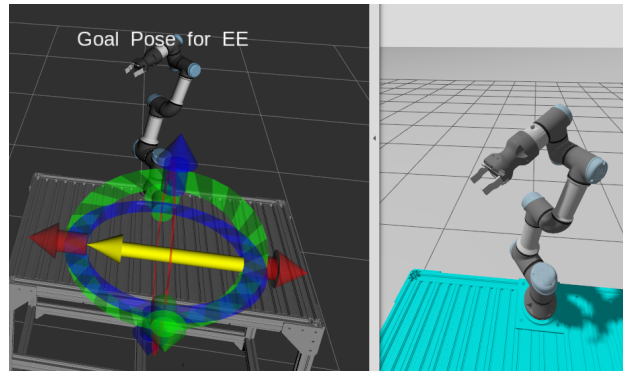
Exercise 2. Simple Joint Space Planner (25 pts.)

Figure 3: RViz interactive marker and UR3e at goal configuration.

Deliverables:

Compute Desired Joint States via IK:

a. The RViz interactive marker publishes the desired goal pose as an $SE(3)$ object stored in the class variable `goal_pose`, which is a message of type `geometry_msgs.msg.Pose`. Since we're planning in joint space, however, we need to compute the joint states (set of named joint angles) that yield the desired goal pose via IK.

   Modify the `TODO`-highlighted sections in the `run` method of the `simple_planner` node to compute the inverse kinematics (IK) solution for `goal_pose`. Specifically, provide the code for `T_1`, `T_2`, and `T` according to equations (3), (4), and (2), respectively. A code template is already present; complete the sub-task by filling in the `TODO`-highlighted blocks:

```python
# TODO (Ex. 2): compute IK to get joint states
# TODO: get translation component of goal_pose class variable
T1 = sm.SE3()

# TODO: get orientation component of goal_pose class variable
# sm.UnitQuaternion takes two arguments:
# a scalar part (s) and a three-dimensional array (v)
# set each to the respective object from goal_pose
T2 = sm.UnitQuaternion(
        # fill in
        ).SE3() # cast to SE3

# TODO: Compose translational and rotational components to get final
# SE3
T = sm.SE3() #
```

Simple Planner Class Method:

b. Create a class method named `simple_planner` in the `SimplePlanner` `Node` class, and add the following code template to it. Complete the sections highlighted with a `TODO`

18

block:

```python
def simple_planner(
    self,
    desired_js: dict[float], # of the form {"joint_name": value,}
    N: int=20,
    T: float=10.
    ) -> JointTrajectory:
    """A simple planner that plans lines in joint space.
    """
    traj = JointTrajectory()
    traj.points = [JointTrajectoryPoint() for _ in range(N)]
    traj.joint_names = list(self.ur3e_joint_states.keys())

    dt = T/N

    # TODO (Ex. 2): create time stamps using np.linspace()
    stamps = np.array([])

    # TODO (Ex. 2): fill the traj.points field for the N trajectory \
    # points
    # using stamps from above
    for i in range(N):
        traj.points[i].time_from_start.sec = 0.0 # m
        nsec = 0.0 # ns
        traj.points[i].time_from_start.nanosec = nsec

    return traj
```

Test the Simple Planner Node:

After completing your modifications, in one tmux pane, build the `simple_planner` package by navigating to the workspace root and running `colcon`:

```
cd ~/ros2_ws/
colcon build --symlink-install --packages-select simple_planner
source install/setup.bash
```

Then launch the UR3e-HandE Gazebo bring-up packages:

```
ros2 launch ur3e_robotiq_gz gz_ros2_control.launch.py
```

This should bring-up an RViz window Then in a second tmux pane, run the `simple_planner` node:

```
ros2 run simple_planner simple_planner_node
```

This should bring up an interactive RVIz marker that you can move around to set the goal end-effector for the planner. Once set, the IK subroutine will kick in and try to compute a solution corresponding to the set goal pose. If a solution is found, the node will print the solution to the terminal and plan lines in joint space (i.e., linear interpolation) to compute a

`JointTrajectory`. Once a trajectory is computed, the node will then prompt you to execute the computed trajectory by pressing the `Enter` key:

```
Goal Pose Recieved, Press Enter to Execute
```

After pressing `Enter`, you should see the robot executing the linear C-space trajectory, as in Figure 3. Moving the RViz interactive markers will advertise new goal poses. Try advertising varied goal poses by moving the interactive marker around. Notice how some plans lead the robot to undesirable configurations?

To complete this exercise, provide the following files along with your report:

i. Your complete implementation of the `simple_planner.py` file

ii. A screenshot of the robot in Gazebo at two distinct goal configurations (i.e., meaning that you would have to send pose goals at least thrice)

iii. The corresponding screenshots of your `tmux` terminal showing the goal configuration solutions returned by the IK solver for each scenario in (ii). above.

Note: You may notice that the robot passes through its mounting table in the simulator. This is an artifact arising from the physics engine used by Gazebo Harmonic, which makes collisions between rigid bodies with complex collision geometries not to be simulated correctly. This lack of collision-awareness is by design, however, to motivate the need for collision-free planning, which we will cover in the next lab.

6. Conclusion and Lab Report Instructions

After completing the lab, summarize the procedure and results into a well written lab report. Include your solutions to all of the question boxes in the lab report. Refer to the List of Questions to ensure you don't miss any. Include your full solution to the exercises in your lab report. If you wrote any Python programs during the exercises, include these files as a separate attachment, appending your team's number and last names (in alphabetical order) to the file, e.g., **Jane Doe**, **Alice Smith**, and **Richard Roe** in **Team 90** submitting their modified `main.py` would submit the file `main_Team90_Doe_Roe_Smith.py` and the report `Lab_<LabNumber>_Report_Team90_Doe_Roe_Smith.pdf`, where `<LabNumber>` is the number for that week's lab, e.g., **0**, **1**, **2**, etc. Submit your lab report along with any code to ELMS under the assignment for that week's lab. You must complete and submit your lab report by **Friday, 11:59 PM** of the week following the lab session. See the **Files** section on ELMS for a lab report template named **Lab_Report_Template.pdf**. Below is a rubric breaking down the grading for this week's lab assignment:

Component	Points
Lab Report and Formatting	10
Question 1.	5
Question 2.	2.5
Question 3.	5
Question 4.	5
Question 5.	2.5
Question 6.	5
Question 7.	5
Question 8.	5
Question 9.	5
Exercise 1.	25
Exercise 2.	25
Total	100