🦾 ACTIVIDAD GUIADA II

**MÓDULO DE ROBOTS MANIPULADORES – PRERREQUISITO:**

**NINGUNO**

**Actividad guiada I:** Paquete en ROS usando los controladores de ROBOTIS para mover el efector final del manipulador a una posición definida en el espacio de trabajo.

**EJECUCIÓN**

1. Para mover el efector final de nuestro robot Open Manipulator x lo haremos mediante los servicios de ROS. Primero, debemos crear un paquete de ROS con las dependencias message\_generation, rospy, std\_msgs. Puedes usar los siguientes comandos:

cd ~/catkin\_ws/src

catkin\_create\_pkg service\_open\_manipulator rospy std\_msgs message\_generation

cd ~/catkin\_ws

catkin\_make



1. En el paquete crea los folders **src, srv, scripts, launch** que contendrán los archivos necesarios para el servicio. Para ello usa los siguientes comandos:

cd ~/catkin\_ws/src/service\_open\_manipulator

mkdir src srv scripts launch

cd ~/catkin\_ws

catkin\_make

1. Luego, se crea el archivo .srv el cual se define la estructura de los datos que se enviarán entre el cliente y el servidor del servicio, para ello usa el siguiente comando:

cd ~/catkin\_ws/src/service\_open\_manipulator/srv

touch go\_location\_manipulator.srv  
cd ~/catkin\_ws

catkin\_make clean

catkin\_make

Cuando se cree el archivo pega el siguiente código en el **go\_location\_manipulator.srv** :

float64[] kinematics\_pose # Lista con coordenadas [x, y, z] destino 1

float64 path\_time # Tiempo de ejecución

---

bool ok # Mensaje de confirmación

1. Ahora, es necesario crear el archivo del cliente y el servidor con los siguientes comandos:

cd ~/catkin\_ws/src/service\_open\_manipulator/scripts

touch go\_location\_server.py go\_location\_client.py

cd ~/catkin\_ws

catkin\_make

En el archivo **go\_location\_server.py** escribe un código como este:

#!/usr/bin/env python3

import rospy

import signal

import sys

import roslaunch

import subprocess, os, signal

from sensor\_msgs.msg import JointState

from open\_manipulator\_msgs.srv import (

SetActuatorState, SetActuatorStateRequest,

SetJointPosition, SetJointPositionRequest,

SetKinematicsPose, SetKinematicsPoseRequest

)

from open\_manipulator\_msgs.msg import JointPosition, KinematicsPose, OpenManipulatorState

from service\_open\_manipulator.srv import go\_location\_manipulator, go\_location\_manipulatorResponse

class servicio\_open\_manipulator(object):

def \_\_init\_\_(self):

self.present\_joint\_angle = None

self.present\_kinematic\_position = [0.0, 0.0, 0.0]

self.kinematics\_pose = None

self.open\_manipulator\_is\_moving = False

self.open\_manipulator\_actuator\_enabled = False

self.process\_sim = None # Para el proceso de la simulación

self.process\_real = None # Para el proceso real

self.joint\_state\_subscriber = rospy.Subscriber('/joint\_states', JointState, self.joint\_states\_callback)

self.kinematics\_pose\_subscriber = rospy.Subscriber('/gripper/kinematics\_pose', KinematicsPose, self.kinematics\_pose\_callback)

self.open\_manipulator\_states\_sub = rospy.Subscriber("/states", OpenManipulatorState, self.manipulator\_states\_callback)

print("hola")

self.set\_actuator\_state\_client = rospy.ServiceProxy('/set\_actuator\_state', SetActuatorState)

self.set\_joint\_position\_client = rospy.ServiceProxy('/goal\_joint\_space\_path', SetJointPosition)

self.set\_tool\_control\_client = rospy.ServiceProxy('/goal\_tool\_control', SetJointPosition)

self.goal\_task\_space\_path\_position\_only= rospy.ServiceProxy('/goal\_task\_space\_path\_position\_only', SetKinematicsPose)

self.Posicion\_inicial= self.positionpreparado()

self.go\_location\_manipulator = rospy.Service('/go\_location\_manipulator', go\_location\_manipulator, self.inicializaMiServicio)

rospy.loginfo(" Servidor 'go\_location\_manipulator' esperando solicitudes...")

def inicializaMiServicio(self,req):

kinematics\_pose=req.kinematics\_pose

path\_time =req.path\_time

rospy.loginfo(f"Recibida solicitud para posición 1 con kinematics\_pose1 = {req.kinematics\_pose}")

success = self.setTaskSpacePath(kinematics\_pose, path\_time) #puede haber problema porque se almacena en sucess

print (f"Se envió kinematics\_pose1,{kinematics\_pose}")

print(success)

response = go\_location\_manipulatorResponse() # Crear instancia de respuesta

response.ok = True # Asignar el valor correcto a la variable 'ok'

return response # Retornar la respuesta correctamente

def positionpreparado(self):

"""

Función para manejar el clic del botón "Posición Inicial".

"""

joint\_name = ["joint1", "joint2", "joint3", "joint4"]

joint\_angle = [0.0, 0.0, 0.0, 0.0] # Ángulos de la posición "Preparado para atender al cliente"

path\_time = 2.0 # Tiempo de movimiento

if not self.setJointSpacePath(joint\_name, joint\_angle, path\_time):

print("[ERR!!] Failed to send service")

return

pass

def joint\_states\_callback(self, msg):

"""

Callback para recibir mensajes de /joint\_states.

Extrae las posiciones de las articulaciones y actualiza el estado.

"""

temp\_angle = [0.0] \* 5

for i in range(len(msg.name)):

if msg.name[i] == "joint1":

temp\_angle[0] = msg.position[i]

elif msg.name[i] == "joint2":

temp\_angle[1] = msg.position[i]

elif msg.name[i] == "joint3":

temp\_angle[2] = msg.position[i]

elif msg.name[i] == "joint4":

temp\_angle[3] = msg.position[i]

elif msg.name[i] == "gripper":

temp\_angle[4] = msg.position[i]

print(f"{self.present\_joint\_angle}")

self.present\_joint\_angle = temp\_angle

def joint\_states\_callback(self, msg):

""" Retorna los ángulos actuales de las articulaciones. """

return self.present\_joint\_angle

def kinematics\_pose\_callback(self, msg):

"""

Callback para recibir mensajes de KinematicsPose.

Extrae la posición y actualiza el estado.

"""

temp\_position = [msg.pose.position.x, msg.pose.position.y, msg.pose.position.z]

self.present\_kinematic\_position = temp\_position

self.kinematics\_pose = msg.pose

def getPresentKinematicsPose(self):

""" Retorna la posición cinemática actual. """

return self.present\_kinematic\_position

def setActuatorState(self, actuator\_state):

"""

Activa o desactiva los actuadores del manipulador.

:param actuator\_state: True para habilitar, False para deshabilitar.

:return: True si fue exitoso, False si falló.

"""

try:

srv = SetActuatorStateRequest()

srv.set\_actuator\_state = actuator\_state

response = self.set\_actuator\_state\_client(srv)

return response.is\_planned

except rospy.ServiceException as e:

rospy.logerr(f"Service call failed: {e}")

return False

def manipulator\_states\_callback(self, msg):

"""

Callback para recibir el estado del manipulador.

"""

self.open\_manipulator\_is\_moving = msg.open\_manipulator\_moving\_state == msg.IS\_MOVING

self.open\_manipulator\_actuator\_enabled = msg.open\_manipulator\_actuator\_state == msg.ACTUATOR\_ENABLED

def get\_open\_manipulator\_moving\_state(self):

""" Retorna si el manipulador está en movimiento. """

return self.open\_manipulator\_is\_moving

def get\_open\_manipulator\_actuator\_state(self):

""" Retorna si el actuador del manipulador está habilitado. """

return self.open\_manipulator\_actuator\_enabled

def setJointSpacePath(self, joint\_name, joint\_angle, path\_time):

"""

Envía una solicitud para mover el manipulador en el espacio articular.

:param joint\_name: Lista con nombres de articulaciones.

:param joint\_angle: Lista con ángulos de destino.

:param path\_time: Tiempo de ejecución.

:return: True si el servicio fue exitoso, False si falló.

"""

try:

srv = SetJointPositionRequest()

srv.joint\_position.joint\_name = joint\_name

srv.joint\_position.position = joint\_angle

srv.path\_time = path\_time

response = self.set\_joint\_position\_client(srv)

return response.is\_planned

except rospy.ServiceException as e:

rospy.logerr(f"Service call failed: {e}")

return False

def setToolControl(self, joint\_angle):

"""

Controla la herramienta (gripper).

:param joint\_angle: Lista con la posición deseada del gripper.

:return: True si el servicio fue exitoso, False si falló.

"""

try:

srv = SetJointPositionRequest()

srv.joint\_position.joint\_name.append("gripper")

srv.joint\_position.position = joint\_angle

response = self.set\_tool\_control\_client(srv)

return response.is\_planned

except rospy.ServiceException as e:

rospy.logerr(f"Service call failed: {e}")

return False

def setTaskSpacePath(self, kinematics\_pose, path\_time):

"""

Envía una solicitud para mover el efector final en el espacio cartesiano.

:param kinematics\_pose: Lista con las coordenadas [x, y, z] de destino.

:param path\_time: Tiempo de ejecución.

:return: True si la planificación fue exitosa, False si falló.

"""

try:

srv = SetKinematicsPoseRequest()

srv.end\_effector\_name = "gripper"

srv.kinematics\_pose.pose.position.x = kinematics\_pose[0]

srv.kinematics\_pose.pose.position.y = kinematics\_pose[1]

srv.kinematics\_pose.pose.position.z = kinematics\_pose[2]

# Mantener la orientación actual del efector final

srv.kinematics\_pose.pose.orientation = self.kinematics\_pose.orientation

srv.path\_time = path\_time

# response = self.goal\_task\_space\_path\_position\_only\_client(srv)

response = self.goal\_task\_space\_path\_position\_only(srv)

print("envía bien los datos al servicio")

print(f"Respuesta del servicio (is\_planned): {response.is\_planned}")

return response.is\_planned

except rospy.ServiceException as e:

rospy.logerr(f"Service call failed: {e}")

return False

def start\_robot\_controller\_real(self):

"""Inicia el nodo del controlador del manipulador."""

try:

self.process = subprocess.Popen(

["roslaunch", "open\_manipulator\_controller", "open\_manipulator\_controller.launch", "use\_platform:=true"],

stdout=subprocess.PIPE,

stderr=subprocess.PIPE,

preexec\_fn=os.setsid # Importante para poder terminarlo después

)

except Exception as e:

rospy.logerr(f"Error al lanzar el nodo: {e}")

def stop\_robot\_controller\_real(self):

"""Detiene el nodo del controlador del manipulador."""

if self.process:

os.killpg(os.getpgid(self.process.pid), signal.SIGTERM) # Terminar el grupo de procesos

self.process.wait() # Esperar a que termine

self.process = None

print("Nodo del controlador detenido.")

else:

print("No hay un nodo en ejecución para detener.")

def start\_robot\_controller\_sim(self):

"""Inicia el nodo del controlador del manipulador."""

try:

self.process = subprocess.Popen(

["roslaunch", "open\_manipulator\_controller", "open\_manipulator\_controller.launch", "use\_platform:=false"],

stdout=subprocess.PIPE,

stderr=subprocess.PIPE,

preexec\_fn=os.setsid # Importante para poder terminarlo después

)

except Exception as e:

rospy.logerr(f"Error al lanzar el nodo: {e}")

def stop\_robot\_controller\_sim(self):

"""Detiene el nodo del controlador del manipulador."""

if self.process:

os.killpg(os.getpgid(self.process.pid), signal.SIGTERM) # Terminar el grupo de procesos

self.process.wait() # Esperar a que termine

self.process = None

print("Nodo del controlador detenido.")

else:

print("No hay un nodo en ejecución para detener.")

def run(self):

""" Mantiene el nodo ROS en ejecución. """

rospy.spin()

##########################################################

def main():

rospy.init\_node('manipulator\_service\_server', anonymous=True)

rospy.loginfo(" Nodo 'manipulator\_service\_server' iniciado...")

service\_object = servicio\_open\_manipulator()

service\_object.run()

##########################################################

if \_\_name\_\_ == '\_\_main\_\_':

main()

En el archivo **go\_location\_client.py** escribe un código como este:

#!/usr/bin/env python3

import rospy

from service\_open\_manipulator.srv import go\_location\_manipulator, go\_location\_manipulatorRequest

class client\_open\_manipulator(object):

def \_\_init\_\_(self):

rospy.init\_node('manipulator\_service\_client', anonymous=True)

rospy.loginfo(" Nodo 'manipulator\_service\_client' iniciado...")

rospy.loginfo('Esperando por el servicio....')

rospy.wait\_for\_service('/go\_location\_manipulator')

print ("hola, ya reconocí el servidor")

#Se crea un proxy del servicio

self.client\_open\_manipulator= rospy.ServiceProxy('/go\_location\_manipulator', go\_location\_manipulator)

#se definen los tres destinos a los que tiene que ir el robot

self.kinematics\_pose=[0.255,0.012,0.069]

self.path\_Time=2.0

def SendPositions(self):

try:

# Crear la solicitud de las tres posiciones

solicitud = go\_location\_manipulatorRequest()

solicitud.kinematics\_pose = self.kinematics\_pose

solicitud.path\_time = self.path\_Time

# Llamar al servicio

respuesta = self.client\_open\_manipulator(solicitud)

rospy.loginfo(f"Respuesta del servidor: {respuesta.ok}")

#verifica si la rrspuesta fue true

if respuesta.ok == True:

rospy.loginfo("Las tres respuestas fueron enviadas exitosamente.")

else:

rospy.logwarn(f"Error al enviar los arreglos. Respuesta: {respuesta.ok}")

except rospy.ServiceException as e:

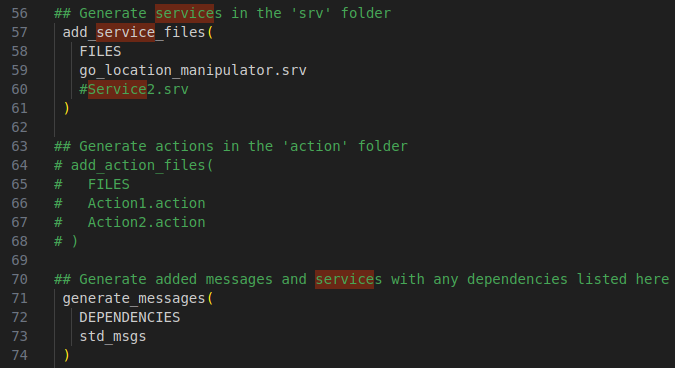
rospy.logerr(f"Error al llamar al servicio: {e}")

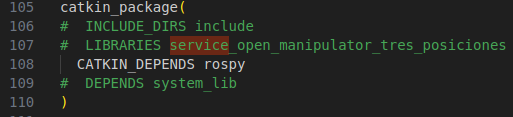
if \_\_name\_\_ == '\_\_main\_\_':

client=client\_open\_manipulator()

client.SendPositions()



1. Editar esto en el archivo CMakeList:  
   



1. Luego, es necesario crear un archivo de lanzamiento .launch que ejecute el nodo del servidor del servicio. para ello usa el siguiente comando:

cd ~/catkin\_ws/src/service\_open\_manipulator/launch

touch service\_manipulator\_launch.launch

cd ~/catkin\_ws

catkin\_make

En el archivo **service\_manipulator\_launch.launch** escribe un código como el siguiente:

<launch>

<node pkg="service\_open\_manipulator" type="go\_location\_server.py" name="go\_location\_manipulator" output="screen">

</node>

</launch>

1. Ahora para ejecutar correctamente el paquete creado ejecuta los siguientes comandos:

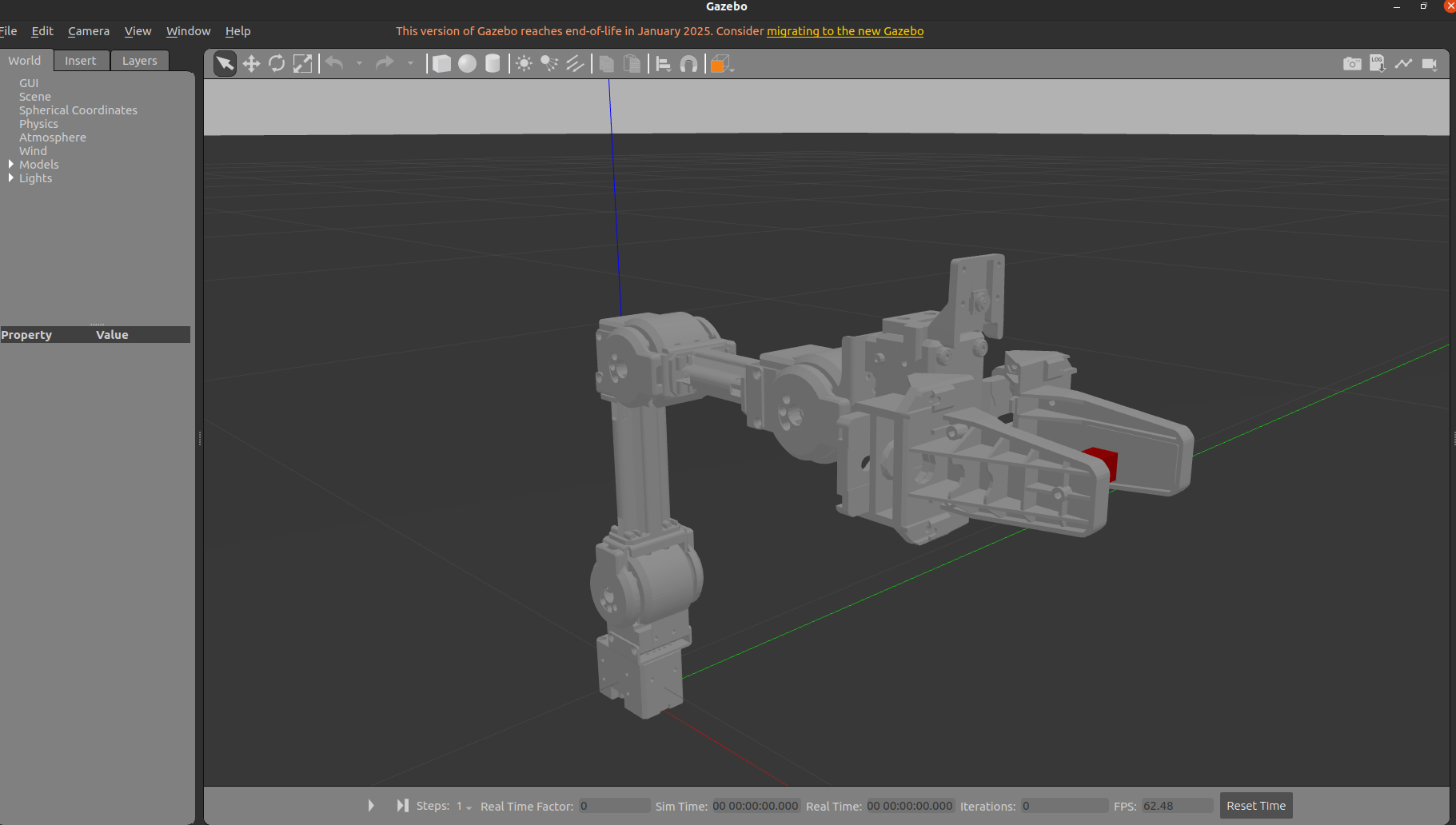
Primero, lanza el roscore para trabajar en el entorno de ROS.

roscore

Luego, si deseas ejecutar el movimiento en el robot simulado gazebo primero debes lanzar el entorno simulado gazebo, así:

roslaunch open\_manipulator\_gazebo open\_manipulator\_gazebo.launch

Se ejecutará el entorno de gazebo.



Cuando finalmente se ejecute el gazebo, no olvides presionar start para iniciar la simulación y permitir que el tiempo avance, si no lo haces es posible que no se ejecuten los movimientos del robot. Luego, lanza el controlador del robot con el comando:

roslaunch open\_manipulator\_controller open\_manipulator\_controller.launch use\_platform:=false

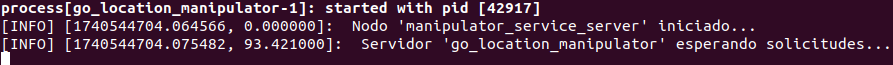
Si deseas ejecutar el movimiento en el ROBOT REAL entonces solo usa este comando:

roslaunch open\_manipulator\_controller open\_manipulator\_controller.launch usb\_port:=/dev/ttyACM0 baud\_rate:=57600

Finalmente, es necesario ejecutar el lanzador del nodo servidor con este comando:

roslaunch service\_open\_manipulator service\_manipulator\_launch.launch

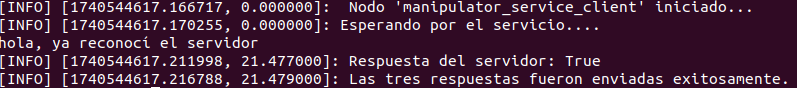
En la terminal visualizarás el mensaje de inicio del nodo servidor.



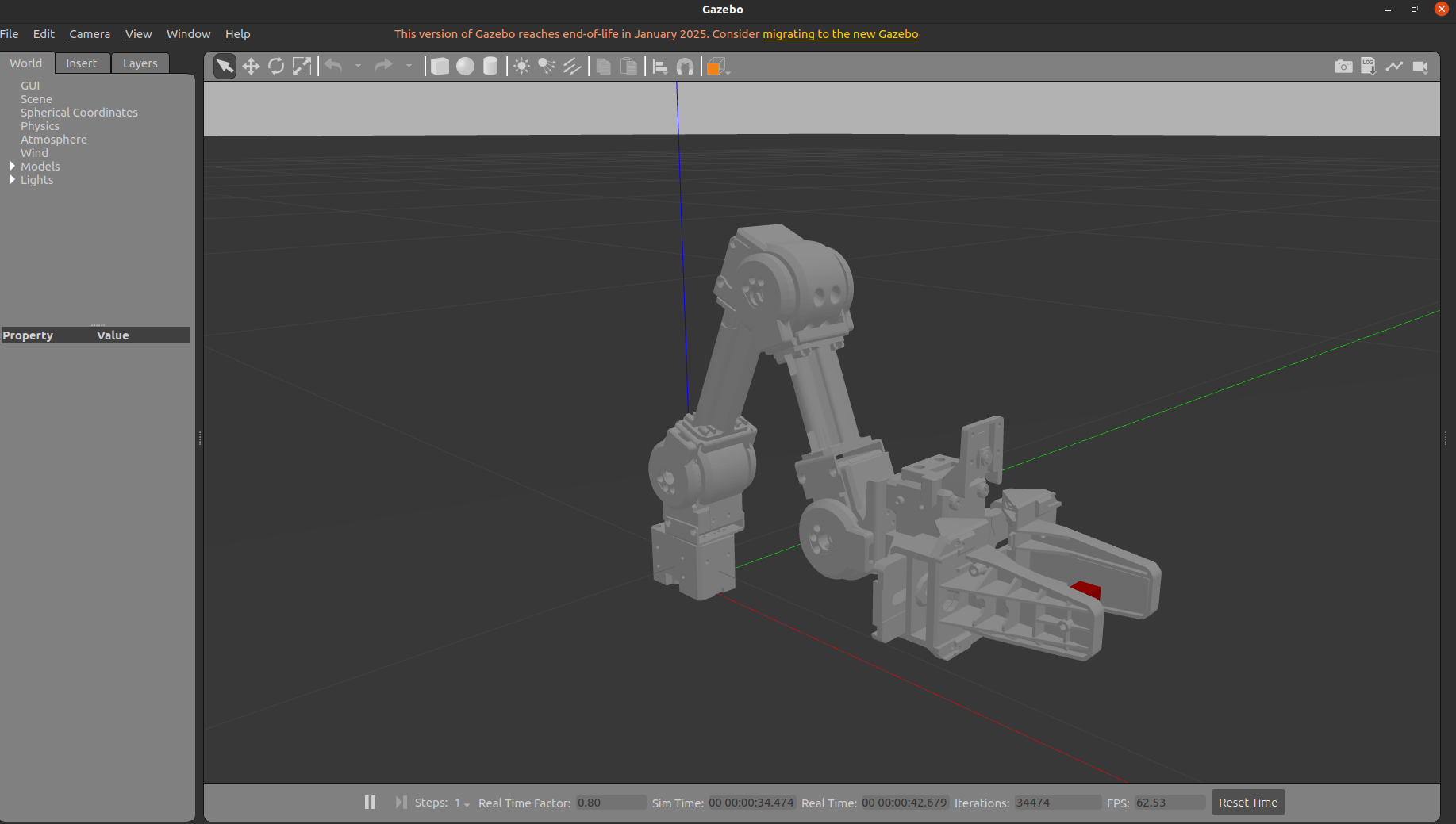
Y para ejecutar el cliente del servicio usa el siguiente:

rosrun service\_open\_manipulator go\_location\_client.py

En la terminal visualizarás el mensaje de inicio y ejecución del nodo cliente.



y al mismo tiempo, podrás observar el correcto movimiento a la coordenada [x,y,z]= [0.255,0.012,0.069] definida:



**NOTA IMPORTANTE:** Si se encuentra trabajando con el **ROBOT REAL** y desea deshabilitar el servicio con CTRL+C, antes de hacerlo envíe al robot a una posición segura, para evitar que el robot sufra daños por golpe al deshabilitar los actuadores, lo puede hacer editando el archivo **go\_location\_client.py** y modificando la posición de destino por:

self.kinematics\_pose=[0.19,-0.0030,0.028]



luego, lance nuevamente el cliente con el comando:

rosrun service\_open\_manipulator go\_location\_client.py

Y listo! ya puedes salir con CTRL+C y salir de la terminal.