Agarrar objeto de un color específico

**3. Sesión de Selección y movimiento manual y automática de objetos de interés**

**usando el manipulador y la cámara – Prerrequisito: Sesión 2.2.**

**Actividad guiada II:** Actividad guiada: Paquete en ROS usando los controladores de ROBOTIS

para mover automáticamente el efector final del manipulador a una posición

definida donde se encuentra un objeto de un color específico y agarrarlo.

# **1. ¿CÓMO FUNCIONA?**

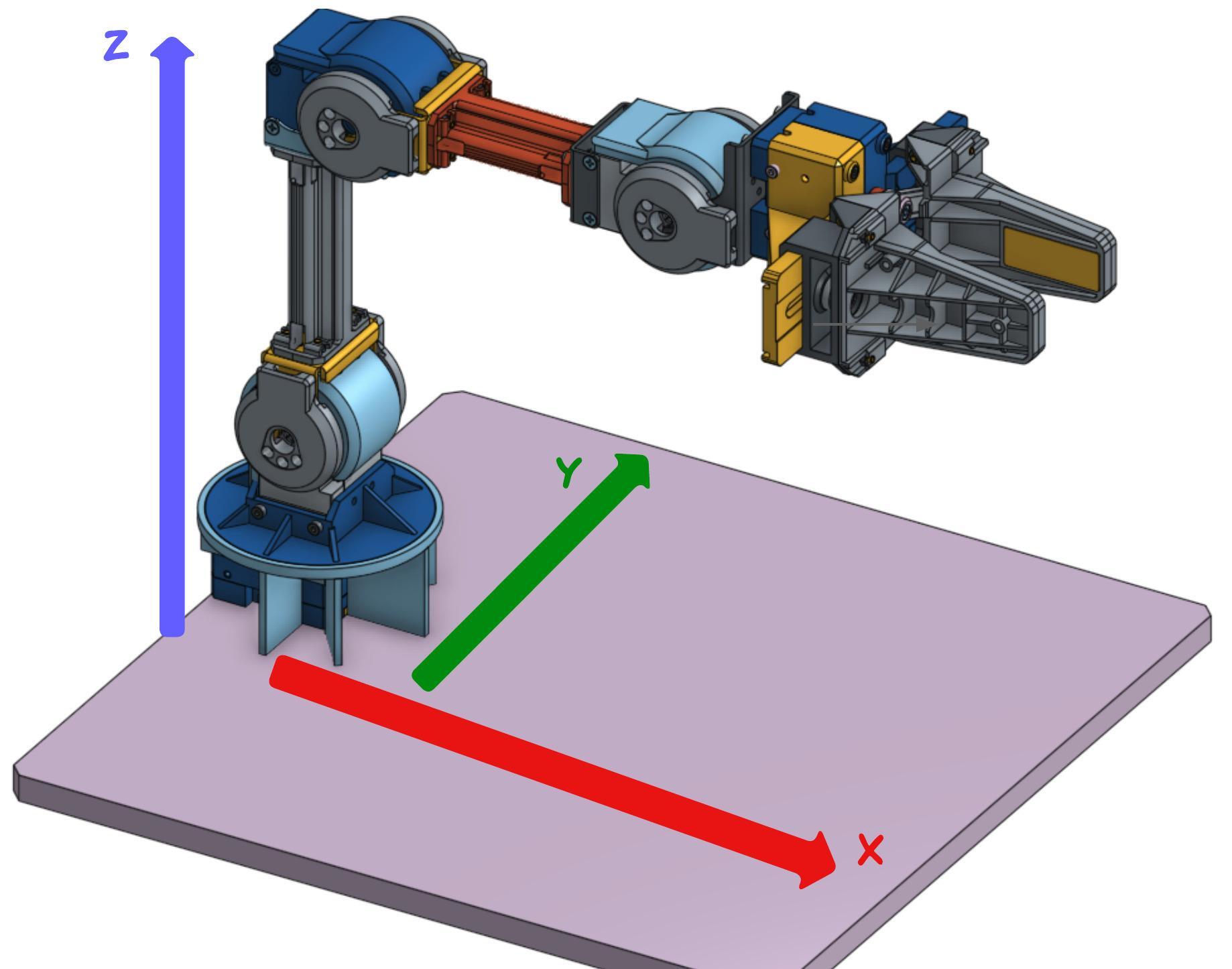
Teniendo en cuenta la actividad guiada I de la Sesión 2.2 en dónde las coordenadas del centroide se publican en el tópico /objeto/Centroide.Aquí también usaremos ese procedimiento y nos suscribimos a ese tópico.

**1.1 Tracking al centroide para agarre automático.**

Una vez identificado el objeto y determinadas sus coordenadas dentro del campo de visión (FOV) publicadas en el tópico /objeto/Centroide, el siguiente paso es realizar un seguimiento (tracking) que permite posicionar el efector final del robot directamente hacia el objeto, facilitando así su agarre.

Dado que la resolución de la imagen es de 320x240 píxeles, se define una zona central dentro del encuadre como el área objetivo. Esta zona representa el punto donde el robot tiene mayor precisión y estabilidad para ejecutar la acción de agarre, ya que se encuentra alineada con el centro del campo visual del sistema. En este caso, el rango horizontal está entre X = 130 y X = 190, y el rango vertical entre Y = 95 y Y = 135.

Como el robot Open Manipulator X trabaja en un plano articular J1,J2,J3,J4 y en un plano cartesiano X,Y,Z pero necesitamos moverlo de forma direccional los movimientos se realizan en el plano cartesiano.



Por lo tanto, si las coordenadas del objeto se encuentran **por debajo del área central** definida, el efector final del robot deberá **disminuir su valor en el eje X** con el fin de mover el objeto hacia la zona superior del campo de visión y alinearlo correctamente para el agarre. Si está por debajo del área deberá aumentar su valor en el eje X. Si está a la izquierda del área central deberá aumentar el valor de Y, y si está a la derecha del área central deberá disminuir su valor en Y hasta que las coordenadas del objeto se encuentren dentro del área central de agarre. En ese momento se ordena al robot desplazarse en dirección al objeto, cerrar el gripper, y luego desplazarse a una posición definida en el algoritmo para abrir el gripper y soltar el objeto.

# **2. PROCEDIMIENTO**

1. Para realizar nuestro paquete que reconozca objetos de un color específico ubicado en una posición definida y agarrarlo. Primero debemos crear nuestro paquete, lo nombramos tracking\_color\_automatico , para ello usamos los siguientes comandos:

$ cd ~/catkin\_ws/src

$ catkin\_create\_pkg tracking\_color\_automatico rospy std\_msgs

$ cd ~/catkin\_ws

$ catkin\_make



1. En nuestro paquete creamos los folders **src, scripts, launch** que contendrán los archivos necesarios. Para ello usa los siguientes comandos:

$ cd ~/catkin\_ws/src/tracking\_color\_automatico

$ mkdir scripts launch

$ cd ~/catkin\_ws

$ catkin\_make

1. Con los folders necesarios ya creados, es hora de crear nuestros scripts. Uno de nuestros scripts nombrado ***posicion\_reconocimiento.py*** cuya finalidad será ubicar el robot Manipulador en la posición de reconocimiento. El script nombrado ***reconocimiento\_color.py*** será el encargado de realizar el reconocimiento del objeto del color específico. y finalmente el script **tracking\_color.py** el cual guiará el robot a las coordenadas en donde se ubique el objeto y lo agarrará.

$ cd ~/catkin\_ws/src/tracking\_color\_automatico/scripts

$ nano posicion\_reconocimiento.py tracking\_color.py reconocimiento\_color.py

1. Ahora, agrega un código como este en el archivo posicion\_reconocimiento.py

#!/usr/bin/env python3

#SOLO LO LLEVA A LA POSICION DE RECONOCIMIENTO

import rospy

from std\_msgs.msg import Bool

from geometry\_msgs.msg import Point # Tipo de mensaje para coordenadas

from sensor\_msgs.msg import JointState, Image

from open\_manipulator\_msgs.srv import (

SetActuatorState, SetActuatorStateRequest,

SetJointPosition, SetJointPositionRequest,

SetKinematicsPose, SetKinematicsPoseRequest)

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

class movimiento\_pos\_reconocimiento:

present\_kinematic\_position = None # Inicialización segura

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.

"""

print("entra a la funcion de cartesiana")

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]

print(f" si le llegan las coordenadas{kinematics\_pose}")

# 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)

return response.is\_planned

except rospy.ServiceException as e:

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

return False

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 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 pos\_reco(self):

kinematics\_pose = [0.2300, 0.0000, 0.0793]

print(f"{kinematics\_pose}")

#print(f"{kinematics\_pose}")

#l gripper angle siempre abierto

path\_time=2.0

if not self.setTaskSpacePath(kinematics\_pose, path\_time):

print("[ERR!!] Failed to send joint angles")

return

print("Send task pose + gripper position")

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 pos\_inicial(self):

"""

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

"""

joint\_angle = 0.01

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

joint\_angle = [0.000, 0.000 , 0.000, 1.456] # Ángulos de la posición inicial

path\_time = 2.0 # Tiempo de movimiento

success = self.setJointSpacePath(joint\_name, joint\_angle, path\_time)

joint\_angle = [0.01] # Ángulo para abrir gripper

joint\_angle=[0.01]

gripper\_closed = self.setToolControl(joint\_angle) #AQUI AGARRA EL OBJETO

if success ==True:

self.logrado = True

while True:

if not rospy.is\_shutdown():

self.publicador\_cumplimiento\_posicion\_reco.publish(self.logrado)

if not self.setToolControl(joint\_angle):

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

return

def pos\_reco(self):

kinematics\_pose = [0.23, 0.00, 0.0793]

print(f"{kinematics\_pose}")

path\_time=2.0

success1 = self.setTaskSpacePath(kinematics\_pose, path\_time)

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

self.kinematics\_pose = None

self.present\_kinematic\_position = [0.0, 0.0, 0.0] # Inicialización segura

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

#Ejecución de movimiento del robot

rospy.init\_node('nodo\_publicador\_pos\_reconocimiento', anonymous=True)

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)

#Ejecución de movimiento del robot

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

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

self.publicador\_cumplimiento\_posicion\_reco = rospy.Publisher('/cumplimiento\_posicion\_reco', Bool, queue\_size=10,latch=True)

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]

self.present\_joint\_angle = temp\_angle

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 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 publicar\_mensaje(self):

position = self.getPresentKinematicsPose()

if not self.setActuatorState(True): # Llamada para habilitar los actuadores

print("No se hbilitaron los actuadores")

return

self.pos\_inicial()

def run(self):

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

rospy.spin()

if \_\_name\_\_ == "\_\_main\_\_":

try:

nodo = movimiento\_pos\_reconocimiento()

nodo.publicar\_mensaje()

except rospy.ROSInterruptException:

pass

1. Ahora, agrega un código como este en el archivo reconocimiento\_color.py

#!/usr/bin/env python3

import rospy

from sensor\_msgs.msg import Image

from cv\_bridge import CvBridge, CvBridgeError

import cv2

import numpy as np

from geometry\_msgs.msg import Point # Tipo de mensaje para coordenadas

class camera\_process:

global hsv\_inferior, hsv\_superior

hsv\_inferior = np.array([100, 50, 50]) # HSV mínimo

hsv\_superior = np.array([140, 255, 255]) # HSV máximo

def camera\_callback(self, data):

bridge = CvBridge()

try:

cv\_image = bridge.imgmsg\_to\_cv2(data, desired\_encoding="bgr8")

except CvBridgeError as e:

print(e)

return

cv\_image = cv2.resize(cv\_image, (320, 240))

hsv = cv2.cvtColor(cv\_image, cv2.COLOR\_BGR2HSV)

mask = cv2.inRange(hsv, hsv\_inferior, hsv\_superior)

contours, \_ = cv2.findContours(mask, cv2.RETR\_EXTERNAL, cv2.CHAIN\_APPROX\_SIMPLE)

for contour in contours:

cv2.drawContours(cv\_image, [contour], -1, (255, 255, 255), 13)

M = cv2.moments(contour)

cx,cy= None, None

if M["m00"] != 0:

cx = int(M["m10"] / M["m00"])

cy = int(M["m01"] / M["m00"])

cv2.circle(cv\_image, (cx, cy), 5, (0, 255, 0), -1)

#publicar e topico de centroide

centroide\_msg = Point()

centroide\_msg.x = cx

centroide\_msg.y = cy

centroide\_msg.z = 0 # Si no hay profundidad, deja en 0

if not rospy.is\_shutdown():

self.publicador\_centroide.publish(centroide\_msg)

# Mostrar imagen procesada en cv\_imagen

imagen\_resized = cv2.resize(cv\_image, (600, 600)) # Redimensionar a 500x500 píxeles

cv2.imshow('Imagen con reconocimiento', imagen\_resized)

cv2.waitKey(1)

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

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

rospy.init\_node('nodo\_reconocimiento\_forma', anonymous=True)

#Ejecución de movimiento del robot

self.camera\_callback = rospy.Subscriber("/usb\_cam/image\_raw", Image, self.camera\_callback)

self.publicador\_centroide = rospy.Publisher('/objeto/Centroide', Point, queue\_size=10)

self.publicador\_dimensiones = rospy.Publisher('/dimensiones\_imagen', Point, queue\_size=10)

try:

rospy.spin()

except KeyboardInterrupt:

print("Shutting down")

cv2.destroyAllWindows()

def run():

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

rospy.spin()

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

Process\_camera = camera\_process()

camera\_process.run()

1. Ahora, agrega un código como este en el archivo tracking\_pos\_final.py

#!/usr/bin/env python3

import rospy

from sensor\_msgs.msg import Image

from cv\_bridge import CvBridge, CvBridgeError

import cv2

import numpy as np

from geometry\_msgs.msg import Point # Tipo de mensaje para coordenadas

import subprocess, os, signal

from open\_manipulator\_msgs.srv import ( SetKinematicsPose, SetKinematicsPoseRequest, SetJointPosition, SetJointPositionRequest)

from open\_manipulator\_msgs.msg import KinematicsPose

import threading

from std\_msgs.msg import Bool

class process:

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

super(process, self).\_\_init\_\_()

rospy.init\_node('tracking', anonymous=True)

self.present\_kinematic\_position = [0.0, 0.0, 0.0] # Inicialización segura

self.kinematics\_pose = None

global x\_min, x\_max, y\_min, y\_max,x,y

global identificacion

x\_min, x\_max = 130, 190

y\_min, y\_max = 95, 135

x=None

y=None

identificacion = False

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

self.centroide\_for\_tracking = rospy.Subscriber("/objeto/Centroide", Point, self.centroide)

self.publicador\_agarrado = rospy.Publisher('state\_agarre', Bool, queue\_size=10)

#Ejecución de movimiento del robot

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

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

# Lanzar hilo para Comparaciones

global running

running = True

self.comparison\_thread = threading.Thread(target=self.ciclo\_comparaciones)

self.comparison\_thread.start()

def ciclo\_comparaciones(self):

rate = rospy.Rate(50) # Ejecuta 50 veces por segundo

while not rospy.is\_shutdown() and running==True:

self.Comparaciones()

rate.sleep()

self.movimiento\_agarre()

def movimiento\_agarre(self):

position = self.getPresentKinematicsPose()

print(f" POSITION OBJETO RECONOCIDO{position}")

kinematics\_pose= [position[0], position[1], 0.05]

path\_time=3.0

print(f"AGARRE: {kinematics\_pose}")

aka1=self.setTaskSpacePath(kinematics\_pose,path\_time)

rospy.sleep(2)

if aka1==True:

position = self.getPresentKinematicsPose()

print(f" POSITION OBJETO RECONOCIDO{position}")

kinematics\_pose= [position[0]+0.05, position[1], 0.05]

path\_time=3.0

print(f"AGARRE: {kinematics\_pose}")

aka2=self.setTaskSpacePath(kinematics\_pose,path\_time)

rospy.sleep(2)

if aka2==True:

joint\_angle=[-0.01]

gripper\_closed = self.setToolControl(joint\_angle) #AQUI AGARRA EL OBJETO

if gripper\_closed is True:

rospy.sleep(2)

print(f"{gripper\_closed}")

kinematics\_pose= [0.13, -0.15, 0.07]

path\_time=2.0

print(f"AGARRE: {kinematics\_pose}")

aka3=self.setTaskSpacePath(kinematics\_pose,path\_time)

rospy.sleep(2)

if aka3==True:

joint\_angle=[0.01]

gripper\_closed = self.setToolControl(joint\_angle)

rospy.sleep(2)

def stop(self):

global running

running = False # Detenemos el bucle de comparaciones

self.comparison\_thread.join() # Esperamos a que el hilo termine

def centroide(self,msg):

#print("esta tracking")

# Extraer las coordenadas de CENTROIDE

global x,y

x = msg.x

y = msg.y

return x,y

def Comparaciones(self):

if self.kinematics\_pose is None:

rospy.logwarn("AÚN NO HAY INFORMACIÓN DEL GRIPPER")

if self.kinematics\_pose is not None:

global x\_min, x\_max, y\_min, y\_max,x,y, identificacion

if x is not None and y is not None:

if x < x\_min: #X PEQUEÑO

position = self.getPresentKinematicsPose()

print(f" POSITION ACTUAL {position}")

kinematics\_pose= [position[0], position[1]+0.01, position[2]]

path\_time=2.0

print("PEQUEÑO MI X. esta a la izquierda del centroide, MOVER A LA DERECHA")

print(f"OBJETIVO: {kinematics\_pose}")

print(f"{x,y}")

aka=self.setTaskSpacePath(kinematics\_pose,path\_time)

rospy.sleep(3)

print(f"CENTROIDE:{x,y}")

if y < y\_min: #Y ES GRANDE

position = self.getPresentKinematicsPose()

print(f" POSITION ACTUAL {position}")

kinematics\_pose= [position[0]+0.01, position[1], position[2]]

path\_time=2.0

print("GRANDE MI Y. esta arriba del centroide, MOVER ABAJO")

print(f"OBJETIVO: {kinematics\_pose}")

aka=self.setTaskSpacePath(kinematics\_pose,path\_time)

rospy.sleep(3)

print(f"CENTROIDE:{x,y}")

if y > y\_max: #Y ES PEQUEÑO

position = self.getPresentKinematicsPose()

print(f" POSITION ACTUAL {position}")

kinematics\_pose= [position[0]-0.01, position[1], position[2]]

path\_time=2.0

print("PEQUEÑO MI Y. esta abajo del centroide, MOVER ARRIBA")

print(f"OBJETIVO: {kinematics\_pose}")

aka=self.setTaskSpacePath(kinematics\_pose,path\_time)

rospy.sleep(3)

print(f"CENTROIDE:{x,y}")

if x > x\_max: #X GRANDE

position = self.getPresentKinematicsPose()

print(f" POSITION ACTUAL {position}")

kinematics\_pose= [position[0], position[1]-0.01, position[2]]

path\_time=2.0

print("PEQUEÑO MI X. esta a la derecha del centroide, MOVER A LA IZQUIERDA")

print(f"OBJETIVO: {kinematics\_pose}")

aka=self.setTaskSpacePath(kinematics\_pose,path\_time)

rospy.sleep(3)

print(f"CENTROIDE:{x,y}")

if y < y\_min: #Y ES GRANDE

position = self.getPresentKinematicsPose()

print(f" POSITION ACTUAL {position}")

kinematics\_pose= [position[0]+0.01, position[1], position[2]]

path\_time=2.0

print("GRANDE MI Y. esta arriba del centroide, MOVER ABAJO")

aka=self.setTaskSpacePath(kinematics\_pose,path\_time)

rospy.sleep(3)

print(f"CENTROIDE:{x,y}")

if y > y\_max: #Y ES PEQUEÑO

position = self.getPresentKinematicsPose()

print(f" POSITION ACTUAL {position}")

kinematics\_pose= [position[0]-0.01, position[1], position[2]]

path\_time=2.0

print("PEQUEÑO MI Y. esta abajo del centroide, MOVER ARRIBA")

aka=self.setTaskSpacePath(kinematics\_pose,path\_time)

rospy.sleep(3)

print(f"CENTROIDE:{x,y}")

if x\_min <= x <= x\_max and y\_min <= y <= y\_max:

print("Centroide dentro del cuadro → Robot detenido")

global running

running = False

else:

print("AÚN NO HAY INFORMACIÓN")

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 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.

"""

#print("entra a la funcion de cartesiana")

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]

#print(f" si le llegan las coordenadas{kinematics\_pose}")

# 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)

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 run(self):

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

rospy.spin()

if \_\_name\_\_ == "\_\_main\_\_":

try:

nodo = process()

nodo.run()

except rospy.ROSInterruptException:

pass



1. Ahora, debes crear el archivo de lanzamiento **reco\_color\_automatico.launch** en el folder launch.

$ cd ~/catkin\_ws/src/tracking\_color\_automatico/launch

$ nano reco\_color.launch tracking\_pos\_final.launch

1. Ahora le agregas las siguientes líneas de código al archivo reco\_color\_automatico.launch

<launch>

<node name="nodo\_posicion\_reconocimiento" pkg="tracking\_color\_automatico" type="posicion\_reconocimiento.py" output="screen"/>

<node name="nodo\_reconocimiento\_color" pkg="tracking\_color\_automatico" type="reconocimiento\_color.py" output="screen"/>

</launch>

1. Ahora le agregas las siguientes líneas de código al archivo tracking\_pos\_final.launch

<launch>

<node name="tracking\_pos\_final\_color" pkg="tracking\_color\_automatico" type="tracking\_pos\_final.py" output="screen"/>

</launch>



1. Ahora, ya es momento de ejecutar nuestro paquete en ROS.

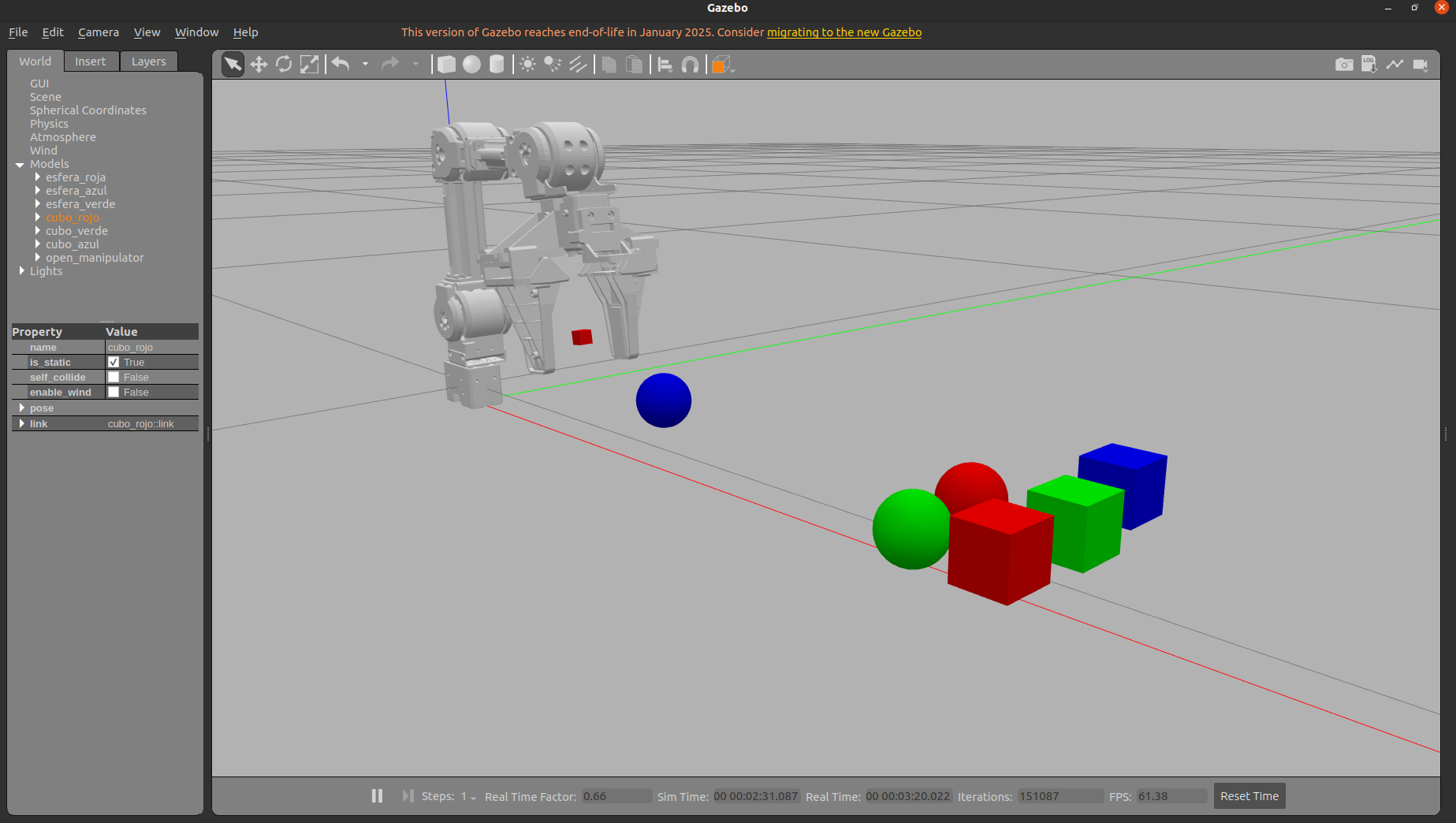
# **3. INICIALIZANDO EL ROBOT**

## **3.1 ROBOT EN GAZEBO**

1. Si deseas usar el robot Manipulator X en Gazebo haz los siguientes pasos:
2. Lanza el mundo en Gazebo con el siguiente comando:

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

¡No olvides presionar Start en la simulación!



1. Ahora, lanza el controlador para poder activar el movimiento y la cámara del robot.

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

Verás lo siguiente:

¡LISTO, Ya está preparado el robot para atendernos!

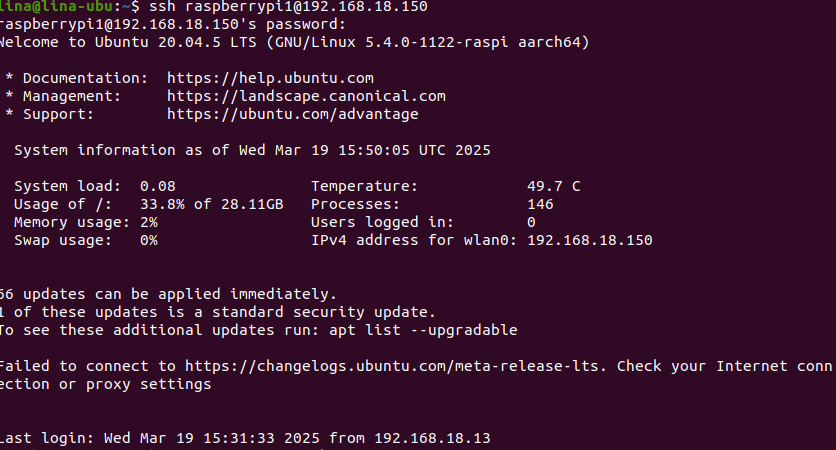
## **3.2 ROBOT REAL**

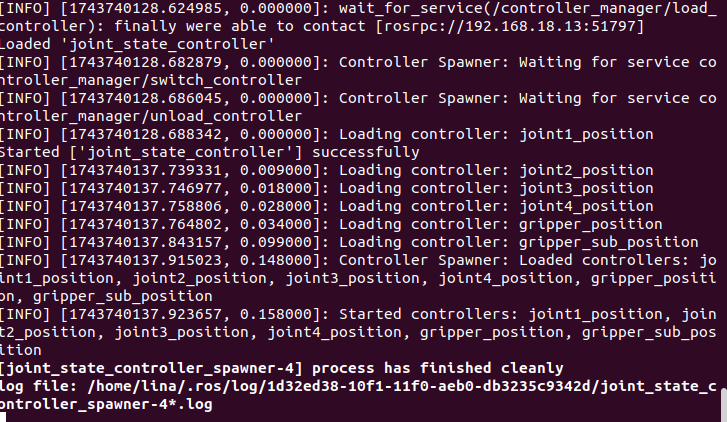
1. Si desea usar el robot real, primero debe conectarlo correctamente, asegúrese de conectar la potencia de la OpenCR y la alimentación de la RaspBerry.
2. Espere unos segundos mientras la raspberry se conecta a la red local. Luego intente conectarse vía ssh con el siguiente comando en la terminal de Ubuntu.

$ ssh raspberrypi1@192.168.18.150

Luego, ingrese la contraseña: **manipulatorx1**

Verás lo siguiente:

****

1. Ahora, ya estará conectado al servidor del robot, primero lanza el roscore para trabajar en un entorno de ROS compartido.

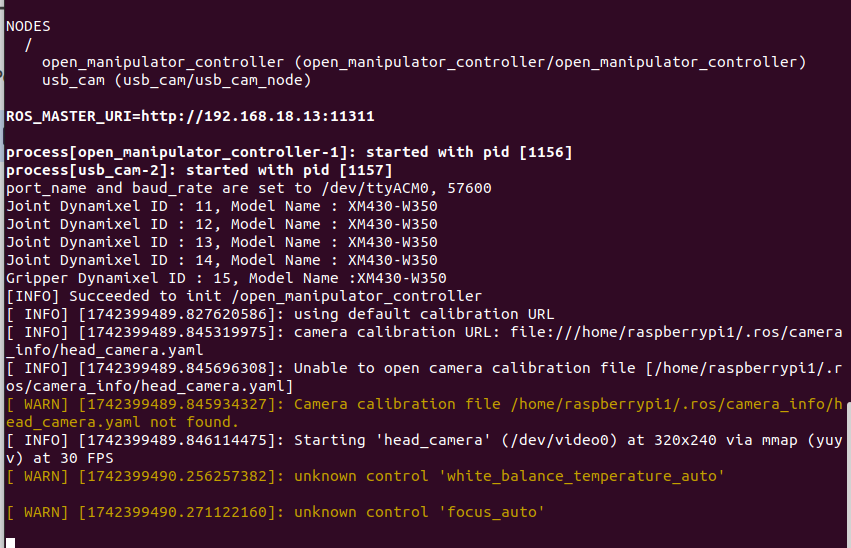
$ roscore



1. Por lo que ya es posible lanzar el controlador que activa los motores y la cámara, lanzando el siguiente comando:

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

Verás lo siguiente:



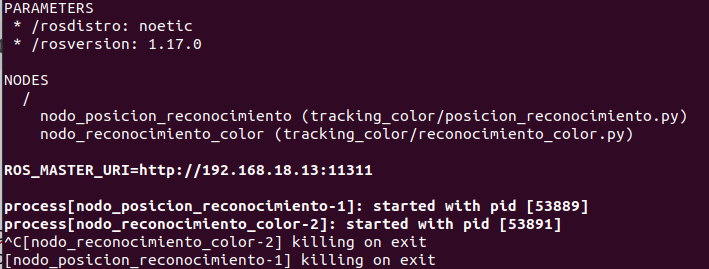
¡LISTO, Ya está preparado el robot para atendernos!

# **4. ¡A CORRER!**

Ahora, ya puedes lanzar el paquete creado. Primero lanzamos el archivo reco\_color.launch para llevar el robot a la posicion de reconocimiento y a ejecutarse el reconocimiento por el color indicado, que en este caso es el azul.

roslaunch tracking\_color reco\_color.launch

verás lo siguiente:



1. Luego, cuando el robot se encuentre en la posición de reconocimiento, ubica el objeto en el campo de visión del robot, verás si el objeto se contornea y se dibuja el centroide es porque ha sido reconocido, es del color deseado.

Cuando lo reconoce, publica las coordenadas en el tópico /objeto/centroide, puedes lanzar este comando para observar:

rostopic echo /objeto/Centroide



1. Ahora, ejecutamos el ptracking\_pos\_final.launch para que nuestro manipulador busque el objeto, se acomode a una posición en la que pueda agarrarlo, y lo transporte al punto en específico que le definimos en nuestro codigo. Para ello, lanzamos el siguiente comando:

**NOTA**: Ten en cuenta que en el robot real, es importante que la iluminación no genere sombras en el ambiente o brillos innecesarios en el objeto.

 roslaunch tracking\_color tracking\_pos\_final.launch

Cuando se ejecute perfectamente, verás algo como esto en la terminal:

