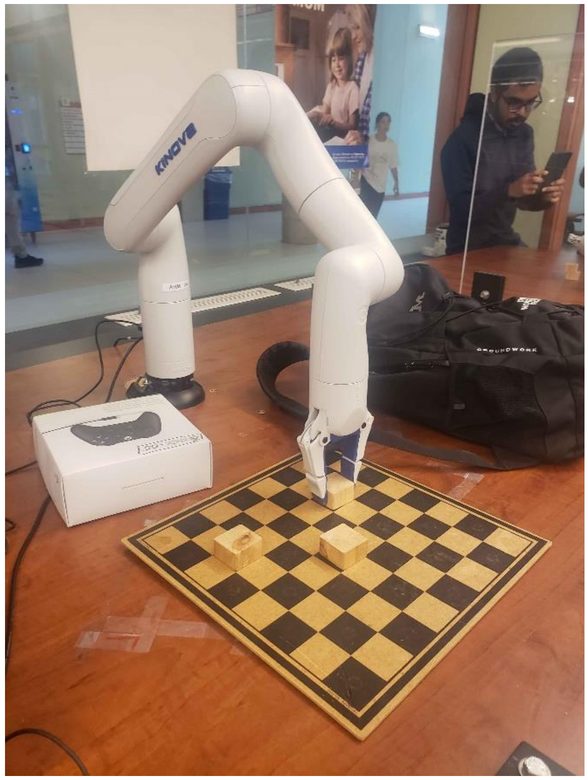
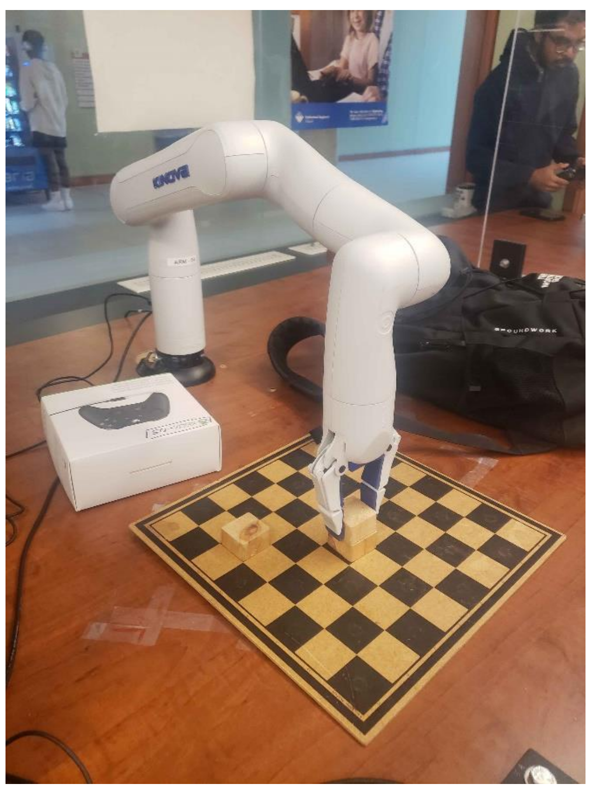
Assignment 1

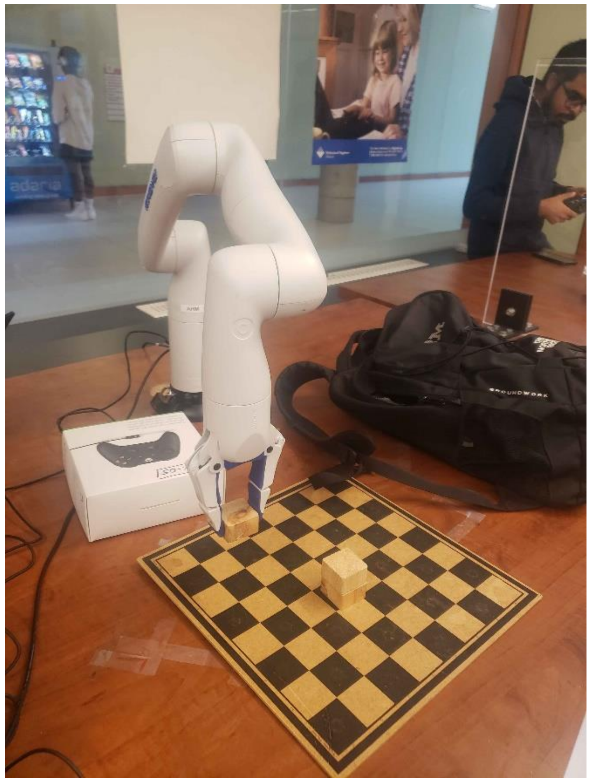
Christopher Budd

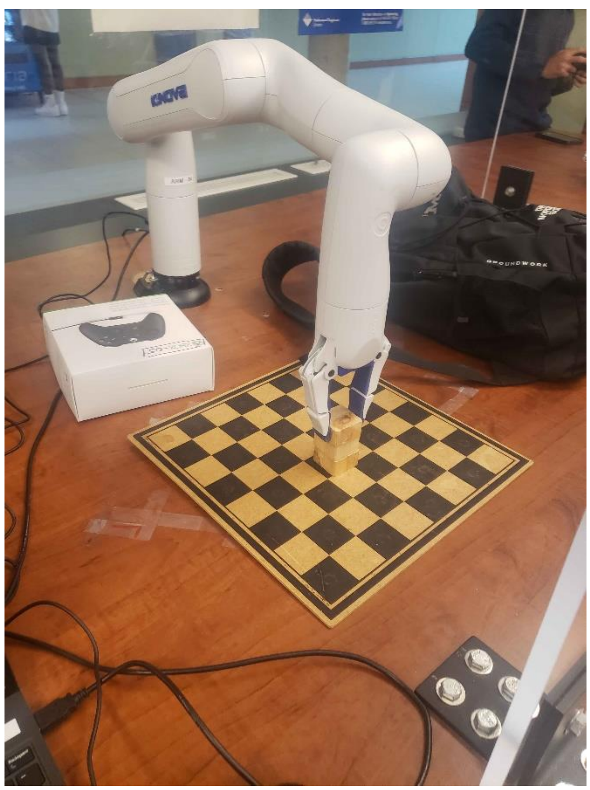
218919068

Question 1







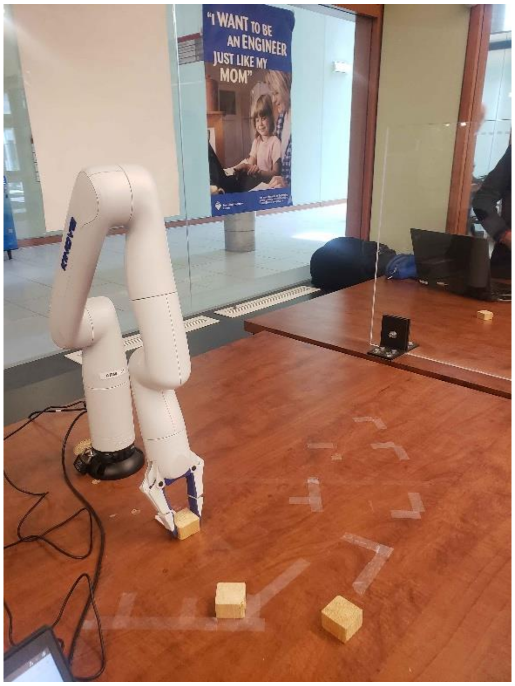


Question 2

Pictures


AN ENGINEER
JUST LIKE MY
MOM"
KINOVA


"I WANT TO BE
AN ENGINEER
JUST LIKE MY
MOM"


"I WANT TO BE
AN ENGINEER
JUST LIKE MY
MOM"


"I WANT TO BE
AN ENGINEER
JUST LIKE MY
MOM"


"I WANT TO BE
AN ENGINEER
JUST LIKE MY
MOM"


"I WANT TO BE
AN ENGINEER
JUST LIKE MY
MOM"

Code

Build_tower.py

```
import sys
import os
import time
import threading

from kortex_api.autogen.client_stubs.BaseClientRpc import BaseClient
from kortex_api.autogen.client_stubs.BaseCyclicClientRpc import BaseCyclicClient

from kortex_api.autogen.messages import Base_pb2, BaseCyclic_pb2, Common_pb2


# Maximum allowed waiting time during actions (in seconds)
TIMEOUT_DURATION = 20

# Create closure to set an event after an END or an ABORT
def check_for_end_or_abort(e):
    """Return a closure checking for END or ABORT notifications

    Arguments:
    e -- event to signal when the action is completed
        (will be set when an END or ABORT occurs)
    """
    def check(notification, e = e):
        print("EVENT : " + \
```

```python
                Base_pb2.ActionEvent.Name(notification.action_event))
        if notification.action_event == Base_pb2.ACTION_END \
        or notification.action_event == Base_pb2.ACTION_ABORT:
            e.set()
    return check

def set_gripper(base, position):
    gripper_command = Base_pb2.GripperCommand()
    finger = gripper_command.gripper.finger.add()

    # Close the gripper with position increments
    print("Performing gripper test in position...")
    gripper_command.mode = Base_pb2.GRIPPER_POSITION
    finger.value = position
    print(f"Going to position {position}")
    base.SendGripperCommand(gripper_command)


def get_gripper(base):
    gripper_request = Base_pb2.GripperRequest()
    gripper_request.mode = Base_pb2.GRIPPER_POSITION
    gripper_measure = base.GetMeasuredGripperMovement(gripper_request)
    if len (gripper_measure.finger):
        print(f"Current position is : {gripper_measure.finger[0].value}")
        return gripper_measure.finger[0].value
    return None


def example_move_to_home_position(base):
    # Make sure the arm is in Single Level Servoing mode
    base_servo_mode = Base_pb2.ServoingModeInformation()
    base_servo_mode.servoing_mode = Base_pb2.SINGLE_LEVEL_SERVOING
    base.SetServoingMode(base_servo_mode)

    # Move arm to ready position
    print("Moving the arm to a safe position")
    action_type = Base_pb2.RequestedActionType()
    action_type.action_type = Base_pb2.REACH_JOINT_ANGLES
    action_list = base.ReadAllActions(action_type)
    action_handle = None
    for action in action_list.action_list:
        if action.name == "Home":
            action_handle = action.handle

    if action_handle == None:
```

```python
        print("Can't reach safe position. Exiting")
        return False

    e = threading.Event()
    notification_handle = base.OnNotificationActionTopic(
        check_for_end_or_abort(e),
        Base_pb2.NotificationOptions()
    )

    base.ExecuteActionFromReference(action_handle)
    finished = e.wait(TIMEOUT_DURATION)
    base.Unsubscribe(notification_handle)

    if finished:
        print("Safe position reached")
    else:
        print("Timeout on action notification wait")
    return finished

def example_angular_action_movement(base, angles=[0.0, 0.0, 0.0, 0.0, 0.0, 0.0]):

    print("Starting angular action movement ...")
    action = Base_pb2.Action()
    action.name = "Example angular action movement"
    action.application_data = ""

    actuator_count = base.GetActuatorCount()

    # Place arm straight up
    print(actuator_count.count)
    if actuator_count.count != len(angles):
        print(f"bad lengths {actuator_count.count} {len(angles)}")
    for joint_id in range(actuator_count.count):
        joint_angle = action.reach_joint_angles.joint_angles.joint_angles.add()
        joint_angle.joint_identifier = joint_id
        joint_angle.value = angles[joint_id]

    e = threading.Event()
    notification_handle = base.OnNotificationActionTopic(
        check_for_end_or_abort(e),
        Base_pb2.NotificationOptions()
    )

    print("Executing action")
    base.ExecuteAction(action)
```

```python
    print("Waiting for movement to finish ...")
    finished = e.wait(TIMEOUT_DURATION)
    base.Unsubscribe(notification_handle)

    if finished:
        print("Angular movement completed")
    else:
        print("Timeout on action notification wait")
    return finished


def example_cartesian_action_movement(base, base_cyclic):

    print("Starting Cartesian action movement ...")
    action = Base_pb2.Action()
    action.name = "Example Cartesian action movement"
    action.application_data = ""

    feedback = base_cyclic.RefreshFeedback()

    cartesian_pose = action.reach_pose.target_pose
    cartesian_pose.x = feedback.base.tool_pose_x          # (meters)
    cartesian_pose.y = feedback.base.tool_pose_y - 0.1    # (meters)
    cartesian_pose.z = feedback.base.tool_pose_z - 0.2    # (meters)
    cartesian_pose.theta_x = feedback.base.tool_pose_theta_x # (degrees)
    cartesian_pose.theta_y = feedback.base.tool_pose_theta_y # (degrees)
    cartesian_pose.theta_z = feedback.base.tool_pose_theta_z # (degrees)

    e = threading.Event()
    notification_handle = base.OnNotificationActionTopic(
        check_for_end_or_abort(e),
        Base_pb2.NotificationOptions()
    )

    print("Executing action")
    base.ExecuteAction(action)

    print("Waiting for movement to finish ...")
    finished = e.wait(TIMEOUT_DURATION)
    base.Unsubscribe(notification_handle)

    if finished:
        print("Cartesian movement completed")
    else:
```

```
        print("Timeout on action notification wait")
    return finished

def main():
    sys.path.insert(0, os.path.join(os.path.dirname(__file__), ".."))
    import utilities

    # Parse arguments
    args = utilities.parseConnectionArguments()

    # Create connection to the device and get the router
    with utilities.DeviceConnection.createTcpConnection(args) as router:

        # Create required services
        base = BaseClient(router)
        base_cyclic = BaseCyclicClient(router)

        success = True
        set_gripper(base, 0.0)
        time.sleep(2)
        success &= example_move_to_home_position(base)

        #First Block
        success &= example_angular_action_movement(base, [48,-58,78,0,0,0])

        set_gripper(base, 1.0)
        time.sleep(2)
        success &= example_angular_action_movement(base, [0,0,78,0,0,0])

        # Stack Spot
        success &= example_angular_action_movement(base, [0,-58,78,0,0,0])
        set_gripper(base, 0.0)
        time.sleep(2)

        # No hit
        success &= example_angular_action_movement(base, [0,0,60,0,0,0])

        # 2 Block
        success &= example_angular_action_movement(base, [-4,-37,126,0,0,0])
        set_gripper(base, 1.0)
        time.sleep(2)

        success &= example_angular_action_movement(base, [0,0,126,0,0,0])

        # Stack Spot
```

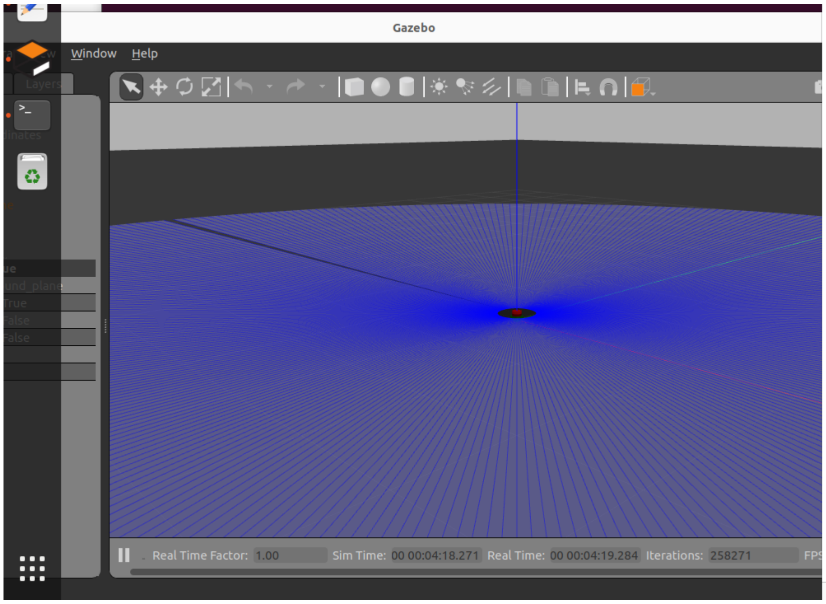
```python
        success &= example_angular_action_movement(base, [0,-52,83,0,0,0])

        set_gripper(base, 0.0)
        time.sleep(2)

        # Reset For next
        success &= example_angular_action_movement(base, [0,-40,80,0,0,0])

        # 3 Block
        success &= example_angular_action_movement(base, [-9,-47,98,0,0,30])
        set_gripper(base, 1.0)
        time.sleep(2)

        # Reset and setup
        success &= example_angular_action_movement(base, [0,-40,80,0,0,0])

        # Last Drop
        success &= example_angular_action_movement(base, [0,-47,84,0,0,0])
        set_gripper(base, 0.0)
        time.sleep(2)

        return 0 if success else 1

if __name__ == "__main__":
    exit(main())
```

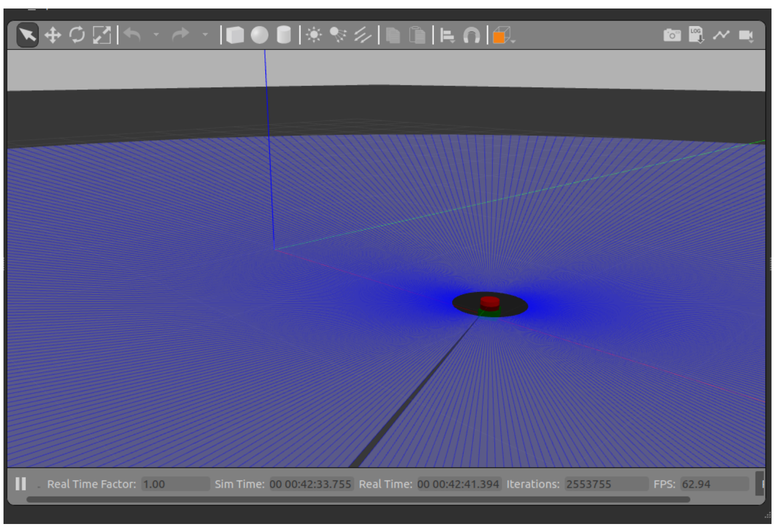
How successful were you in this task?

I would say 70% successful as although it does work, it could be better. The tower itself was stable enough but if I had adjusted the arm more, it would have been better.

Question 3

Pictures





Code

```
import math
import numpy as np
import rclpy
from rclpy.node import Node
from rclpy.parameter import Parameter
from rcl_interfaces.msg import SetParametersResult
```

```python
from nav_msgs.msg import Odometry
from geometry_msgs.msg import Twist, Pose, Point,
Quaternion
from nav_msgs.msg import Odometry

def euler_from_quaternion(quaternion):
    """
    Converts quaternion (w in last place) to euler roll,
pitch, yaw
    quaternion = [x, y, z, w]
    """
    x = quaternion.x
    y = quaternion.y
    z = quaternion.z
    w = quaternion.w

    sinr_cosp = 2 * (w * x + y * z)
    cosr_cosp = 1 - 2 * (x * x + y * y)
    roll = np.arctan2(sinr_cosp, cosr_cosp)

    sinp = 2 * (w * y - z * x)
    pitch = np.arcsin(sinp)

    siny_cosp = 2 * (w * z + x * y)
    cosy_cosp = 1 - 2 * (y * y + z * z)
    yaw = np.arctan2(siny_cosp, cosy_cosp)

    return roll, pitch, yaw


class MoveToGoal(Node):
    def __init__(self):
```

```
        super().__init__('move_robot_to_goal')
        self.get_logger().info(f'{self.get_name()}
created')

        self.declare_parameter('goal_x', 0.0)
        self._goal_x =
self.get_parameter('goal_x').get_parameter_value().doubl
e_value
        self.declare_parameter('goal_y', 0.0)
        self._goal_y =
self.get_parameter('goal_y').get_parameter_value().doubl
e_value
        self.declare_parameter('goal_t', 0.0)
        self._goal_t =
self.get_parameter('goal_t').get_parameter_value().doubl
e_value

        self.declare_parameter('max_vel', 0.4)
        self._max_vel =
self.get_parameter('max_vel').get_parameter_value().doub
le_value

        self.declare_parameter('cmd_gain', 5.0)
        self._cmd_gain =
self.get_parameter('cmd_gain').get_parameter_value().dou
ble_value


self.add_on_set_parameters_callback(self.parameter_callb
ack)
        self.get_logger().info(f"initial goal
{self._goal_x} {self._goal_y} {self._goal_t}")
```

```
        self.get_logger().info(f"maximum velocity
{self._max_vel}")

        self._subscriber =
self.create_subscription(Odometry, "/odom",
self._listener_callback, 1)
        self._publisher = self.create_publisher(Twist,
"/cmd_vel", 1)


    def _listener_callback(self, msg, vel_gain=5.0,
max_vel=0.2, max_pos_err=0.05):
        pose = msg.pose.pose
        max_vel = self._max_vel
        vel_gain = self._cmd_gain
        cur_x = pose.position.x
        cur_y = pose.position.y
        o = pose.orientation
        roll, pitchc, yaw = euler_from_quaternion(o)
        cur_t = yaw

        x_diff = self._goal_x - cur_x
        y_diff = self._goal_y - cur_y
        dist = math.sqrt(x_diff * x_diff + y_diff *
y_diff)

        twist = Twist()
        if dist > max_pos_err:
            x = max(min(x_diff * vel_gain, max_vel), -
max_vel)
            y = max(min(y_diff * vel_gain, max_vel), -
max_vel)
```

```
            twist.linear.x = x * math.cos(cur_t) + y *
math.sin(cur_t)
            twist.linear.y = -x * math.sin(cur_t) + y *
math.cos(cur_t)

        angle_diff = math.atan2(math.sin(self._goal_t -
cur_t), math.cos(self._goal_t - cur_t))


        if abs(angle_diff) > max_pos_err:
            self.get_logger().info(f"Twist
{angle_diff}")
            twist.angular.z = max(min(angle_diff *
vel_gain*4, max_vel*5), -max_vel*5)
            self.get_logger().info(f"Twist ang
{twist.angular.z}")
        self.get_logger().info(f"at
({cur_x},{cur_y},{cur_t}) goal
({self._goal_x},{self._goal_y},{self._goal_t})")
        self._publisher.publish(twist)

    def parameter_callback(self, params):
        self.get_logger().info(f'move_robot_to_goal
parameter callback {params}')
        for param in params:
            self.get_logger().info(f'move_robot_to_goal
processing {param.name}')
            if param.name == 'goal_x' and param.type_ ==
Parameter.Type.DOUBLE:
                self._goal_x = param.value
            elif param.name == 'goal_y' and param.type_
== Parameter.Type.DOUBLE:
```

```
                self._goal_y = param.value
            elif param.name == 'goal_t' and param.type_
== Parameter.Type.DOUBLE:
                self._goal_t = param.value
            else:

self.get_logger().warn(f'{self.get_name()} Invalid
parameter {param.name}')
                return
SetParametersResult(successful=False)
            self.get_logger().warn(f"Changing goal
{self._goal_x} {self._goal_y} {self._goal_t}")
        return SetParametersResult(successful=True)



def main(args=None):
    rclpy.init(args=args)
    node = MoveToGoal()
    try:
        rclpy.spin(node)
    except KeyboardInterrupt:
        pass
    rclpy.shutdown()

if __name__ == '__main__':
    main()
```

Question 4

Layout

If not at goal(if dist > max_pos_err)

    Drive towards goal

    For object in object list

        If an object in the way of the robot within a range

            Follow the obstacle boundary moving to the left or right.

or based on the lab 1 code

For o in map

    Check if in range and in the way

        InWay = true

If inway

    Follow the obstacle boundary

Else

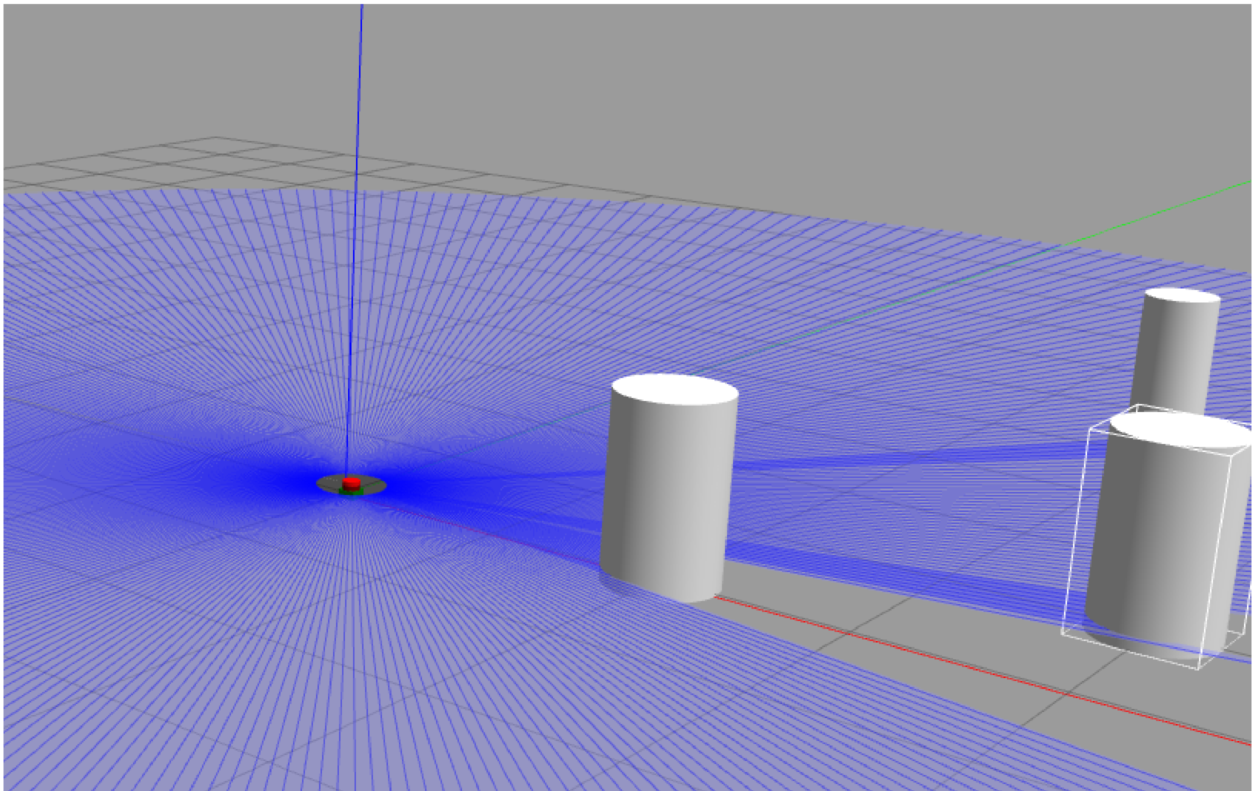
    Drive towards goal

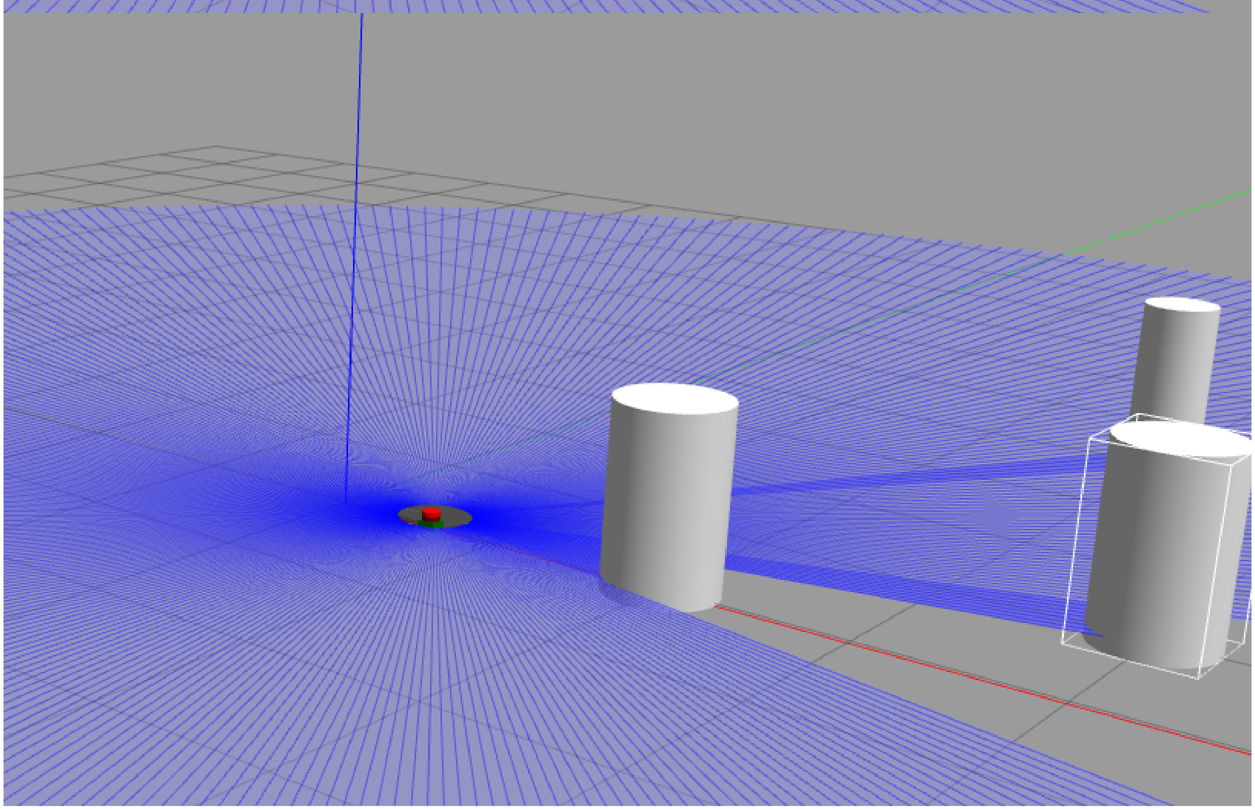
Explanation

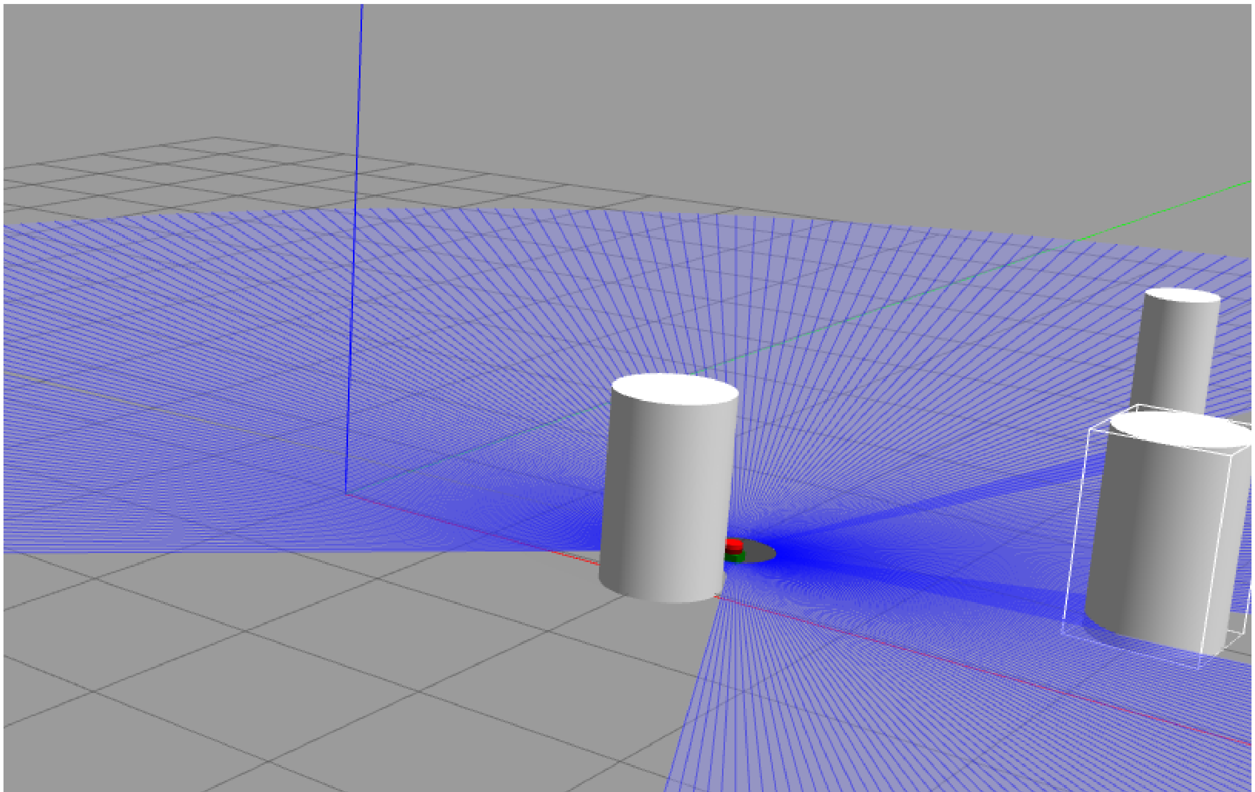
Using _listener_callback, if the robot has not reached its goal, move towards the goal until an object is in its way. It checks the list when it moves to simulate it discovering the object by sensors or hitting it. It then follows the object boundary until it can move towards the goal.

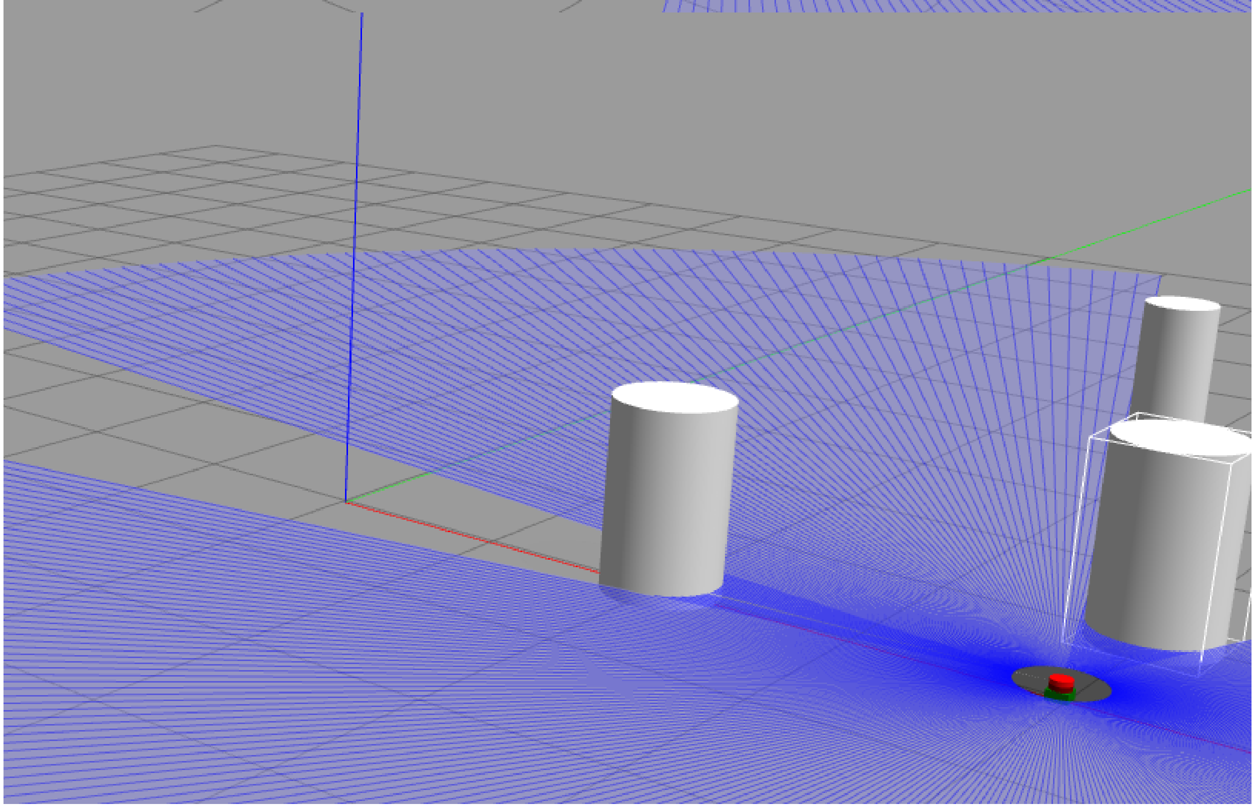
Question 5

Pictures









Code

```
    def _listener_callback(self, msg, vel_gain=5.0,
max_vel=0.2, max_pos_err=0.05):
        pose = msg.pose.pose
        max_vel = self._max_vel
        vel_gain = self._cmd_gain
        cur_x = pose.position.x
        cur_y = pose.position.y
        o = pose.orientation
        roll, pitchc, yaw = euler_from_quaternion(o)
        cur_t = yaw

        x_diff = self._goal_x - cur_x
        y_diff = self._goal_y - cur_y
        dist = math.sqrt(x_diff * x_diff + y_diff *
y_diff)

        twist = Twist()
        obstacle_detected = False
        for o in self._map.keys():
            obj_x, obj_y, obj_r = self._map[o]['x'],
self._map[o]['y'], self._map[o]['r']
            obj_dist = math.sqrt((cur_x - obj_x)**2 +
(cur_y - obj_y)**2)
            self.get_logger().info(f"Obstacle States {o}
{obj_dist}")
            if obj_dist <= obj_r + 0.3:  # 0.5 is a
safety margin
                obstacle_detected = True
                break
```

```
        if obstacle_detected:
            angle_to_obj = math.atan2(obj_y - cur_y,
obj_x - cur_x)
            tangent_angle = angle_to_obj + math.pi/2

            x_tangent = math.cos(tangent_angle)
            y_tangent = math.sin(tangent_angle)

            x = max(min(x_tangent * vel_gain, max_vel),
-max_vel)
            y = max(min(y_tangent * vel_gain, max_vel),
-max_vel)

            twist.linear.x = x * math.cos(cur_t) + y *
math.sin(cur_t)
            twist.linear.y = -x * math.sin(cur_t) + y *
math.cos(cur_t)
            self.get_logger().info(f"Obstacle detected")
        elif dist > max_pos_err:
            x = max(min(x_diff * vel_gain, max_vel), -
max_vel)
            y = max(min(y_diff * vel_gain, max_vel), -
max_vel)
            twist.linear.x = x * math.cos(cur_t) + y *
math.sin(cur_t)
            twist.linear.y = -x * math.sin(cur_t) + y *
math.cos(cur_t)

        angle_diff = math.atan2(math.sin(self._goal_t -
cur_t), math.cos(self._goal_t - cur_t))
```

```
        if abs(angle_diff) > max_pos_err:
            self.get_logger().info(f"Twist {angle_diff}")
            twist.angular.z = max(min(angle_diff * vel_gain*4, max_vel*5), -max_vel*5)
            self.get_logger().info(f"Twist ang {twist.angular.z}")
        self.get_logger().info(f"at ({cur_x},{cur_y},{cur_t}) goal ({self._goal_x},{self._goal_y},{self._goal_t})")
        self._publisher.publish(twist)
```