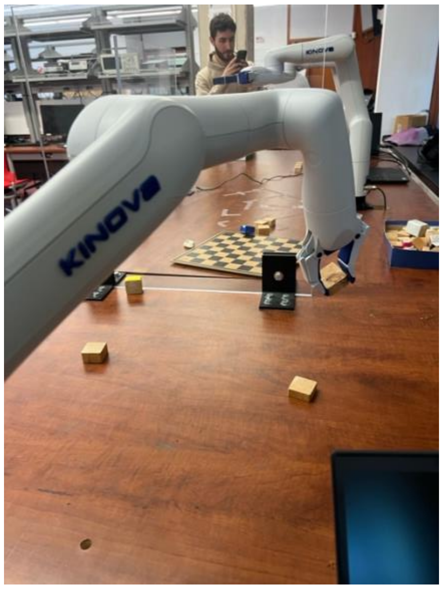
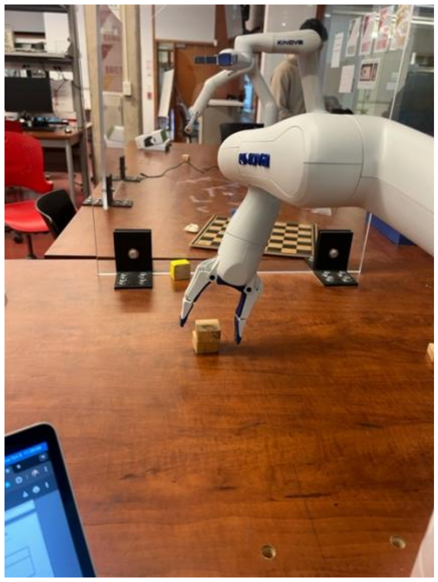
EECS 4421 Assignment1

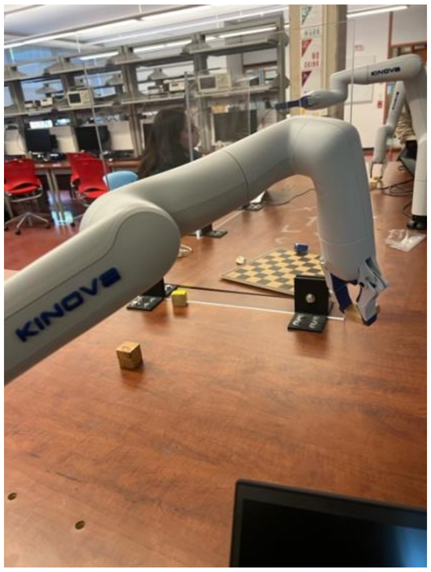
Name: Mahfuz Rahman

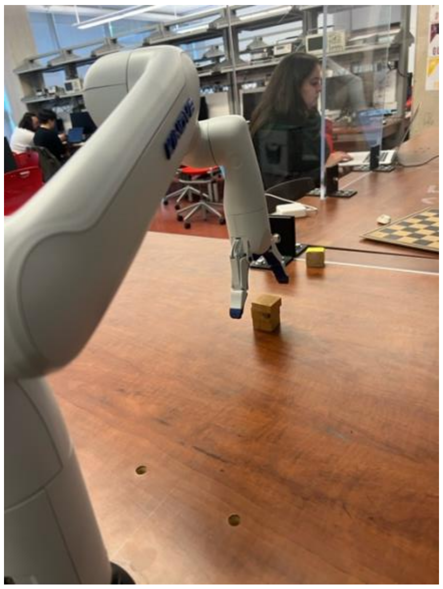
Student No.: 217847518

1. **Use the Kinova Arm in tele operational mode using the Xbox controller to stack a pile of blocks three blocks high. Submit a series of photos of the process of assembling the blocks. [2 marks]**

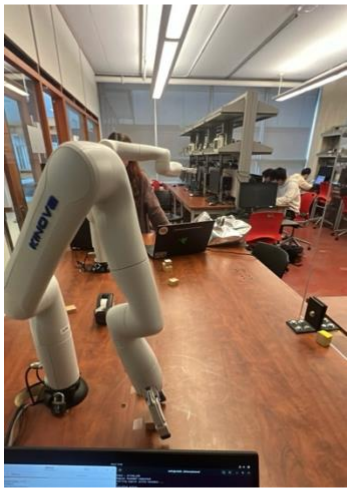
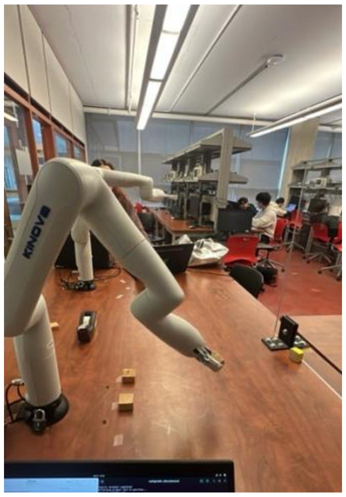
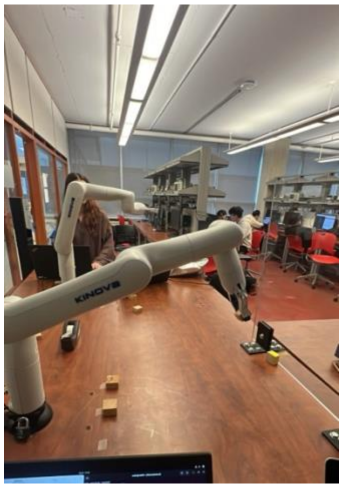
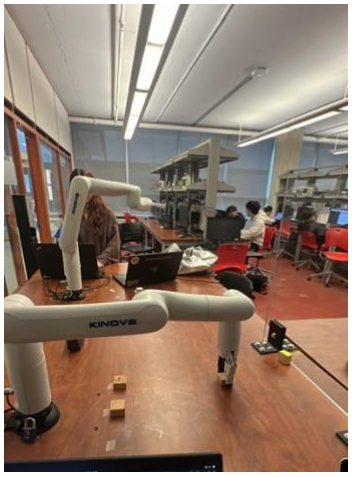
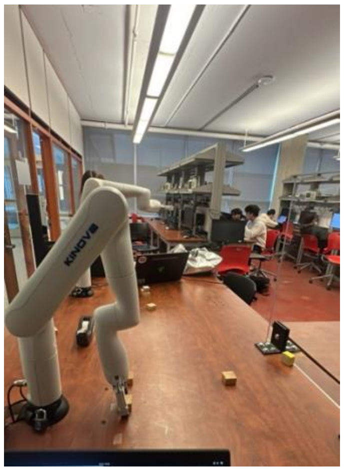
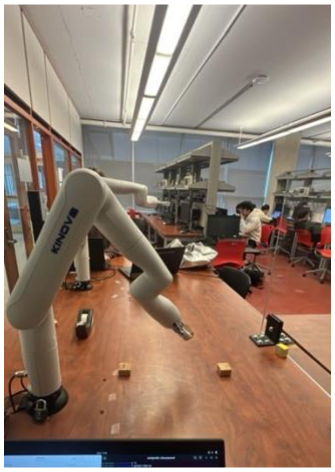
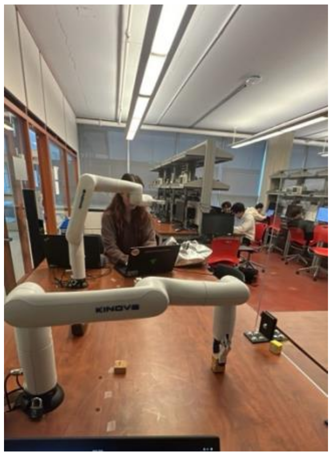
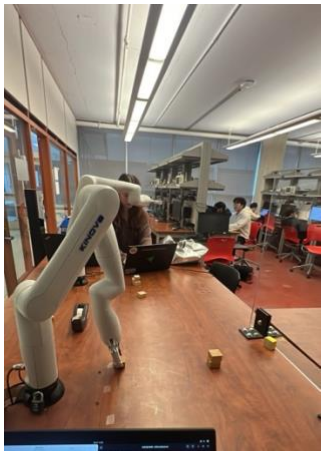
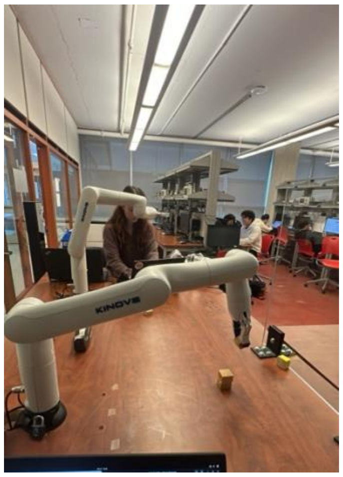
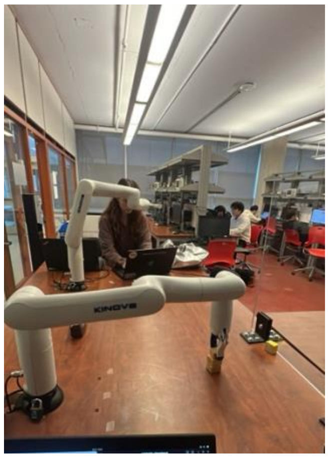
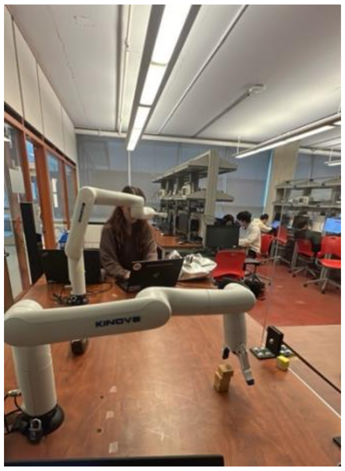
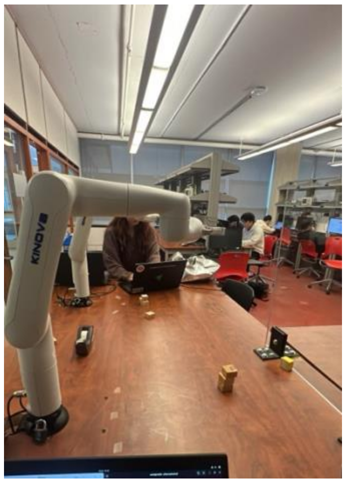








2. **Use the Kinova Arm to grab and pick up a block on the tabletop 20cm in front of the robot aligned with the (x,y) coordinate frame of the arm, and move it to a point at a position (x,y,z) in cm, again aligned with the (x,y) coordinate frame of the arm, and then release it. Us your code to take blocks from the pickup location to build a stack three blocks high at the location of your choice. Photograph this process. How successful were you in this task? Submit your vphoto record, your code, and answer to the question [8 marks].**

In doing this, I used the ‘example_angular_action_movement’ function to move around the robot’s axis to grab and place the blocks. The code is below as follows:

```
###
# KINOVA (R) KORTEX (TM)
#
# Copyright (c) 2018 Kinova inc. All rights reserved.
#
# This software may be modified and distributed
# under the terms of the BSD 3-Clause license.
#
# Refer to the LICENSE file for details.
#
###

# Changes by Mahfuz Rahman
# Using the arm to grap and pick block 20cm away

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 : " + \
            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:
        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)

    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   # (meters)
    cartesian_pose.z = feedback.base.tool_pose_z    # (meters)
    cartesian_pose.theta_x = feedback.base.tool_pose_theta_x + 30 # (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():

    # Import the utilities helper module
    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)

        # Example core
        success = True



    #Home
        success &= example_move_to_home_position(base)



        #2nd axis - increasing value raises it up
        #3rd axis - increasing value raises it down

        ##### First #####
```

```
#Open
set_gripper(base, 0.3)
time.sleep(2)
print(get_gripper(base))

#Reach Initial
success &= example_angular_action_movement(base, [-30,-39,125,0,0,-30])

#Grab
set_gripper(base, 0.8)
time.sleep(2)
print(get_gripper(base))

#Home
success &= example_move_to_home_position(base)

#Reach Final
success &= example_angular_action_movement(base, [0,-45,4,90,82,90])
success &= example_angular_action_movement(base, [0,-89,4,90,82,90])

#Open
set_gripper(base, 0.3)
time.sleep(2)
print(get_gripper(base))

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




#Home
success &= example_move_to_home_position(base)



##### Second #####

#Reach Initial
success &= example_angular_action_movement(base, [0,-37,132,0,0,0])

#Grab
set_gripper(base, 0.8)
time.sleep(2)
```

```
print(get_gripper(base))

#Home
success &= example_move_to_home_position(base)

#Reach Final
success &= example_angular_action_movement(base, [0,-45,4,90,82,90])
success &= example_angular_action_movement(base, [0,-87,4,90,85,90])

#Open
set_gripper(base, 0.0)
time.sleep(2)
print(get_gripper(base))

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

#Home
success &= example_move_to_home_position(base)



##### Third #####

#Reach Initial
success &= example_angular_action_movement(base, [30,-37,128,0,0,30])

#Grab
set_gripper(base, 0.8)
time.sleep(2)
print(get_gripper(base))

#Home
success &= example_move_to_home_position(base)

#Reach Final
success &= example_angular_action_movement(base, [0,-45,4,90,82,90])
success &= example_angular_action_movement(base, [0,-85,4,90,87,90])

#Open
set_gripper(base, 0.0)
time.sleep(2)
print(get_gripper(base))
```

```
        #################################



        #Home
        success &= example_move_to_home_position(base)


        return 0 if success else 1

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

3. **Extend the drive_robot.py code in cpmr_ch2 so that it takes in as parameters the maximum velocity and gain values. Tune these values so that the robot moves 40cm/sec. The current code applies the speed limit separately in the x and y dimensions. Change this so that it applies this limit to the magnitude of the velocity. Test your code. You will find a 'faster' robot useful in Q4, below. Submit the code listing, and screen dumps of images of your robot moving in Gazebo [4 marks]**

Below is the code for the robot moving at 40cm/sec. I have added ‘vel_gain’ and ‘max_vel’ as parameters which can alse be taken from the command prompt. The default value for velocity gain is 5.0 and max velocity is 0.4 (40cm/sec). The velocity is thus applied to the magnitude of the velocity instead of separately in the x and y dimensions.

```
import math
import numpy as np
import rclpy
from rclpy.node import Node
from rclpy.parameter import Parameter
from rcl_interfaces.msg import SetParametersResult
from nav_msgs.msg import Odometry
from geometry_msgs.msg import Twist, Pose, Point, Quaternion

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
```

```python
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.declare_parameter('goal_y', 0.0)
        self.declare_parameter('goal_t', 0.0)
        self.declare_parameter('vel_gain', 5.0)  # Default velocity gain
        self.declare_parameter('max_vel', 0.4)   # Default max velocity (40 cm/sec)

        self._goal_x = self.get_parameter('goal_x').get_parameter_value().double_value
        self._goal_y = self.get_parameter('goal_y').get_parameter_value().double_value
        self._goal_t = self.get_parameter('goal_t').get_parameter_value().double_value
        self._vel_gain = self.get_parameter('vel_gain').get_parameter_value().double_value
        self._max_vel = self.get_parameter('max_vel').get_parameter_value().double_value

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

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


    # This approach applies the velocity to the magnite rather than the separately in the x and y coordinates
    def _listener_callback(self, msg, max_pos_err=0.05):
        pose = msg.pose.pose

        cur_x = pose.position.x
        cur_y = pose.position.y
        o = pose.orientation
        roll, pitch, 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:
            # Applying gain to the velocity
            vx = x_diff * self._vel_gain
            vy = y_diff * self._vel_gain

            # Computing the velocity magnitude
            velocity_magnitude = math.sqrt(vx**2 + vy**2)

            # Applying the velocity limit to the magnitude
            if velocity_magnitude > self._max_vel:
                scale = self._max_vel / velocity_magnitude
                vx *= scale
                vy *= scale


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

            self.get_logger().info(f"At ({cur_x},{cur_y},{cur_t}), goal ({self._goal_x},{self._goal_y},{self._goal_t}), velocity
({vx},{vy})")

        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
            elif param.name == 'vel_gain' and param.type_ == Parameter.Type.DOUBLE:
                self._vel_gain = param.value
            elif param.name == 'max_vel' and param.type_ == Parameter.Type.DOUBLE:
                self._max_vel = param.value
            else:
                self.get_logger().warn(f'{self.get_name()} Invalid parameter {param.name}')
                return SetParametersResult(successful=False)
            self.get_logger().info(f"Changed parameters to goal: ({self._goal_x}, {self._goal_y}, {self._goal_t}), velocity gain:
{self._vel_gain}, max velocity: {self._max_vel}")
        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()
```

4. **Design a version of the bug0 algorithm that assumes that the world is an infinite plane with circular obstacles at known locations and with known radii. Your algorithm should leave circles at tangent points that provide a direct line to the goal location. Develop a mathematical model of the motion of your robot, the location of the leave point(s), and how you would integrate the radius of the circular obstacle with the circular buffer of your robot. Submit a description of your algorithm. This is a pen and paper exercise. [6 marks]**

To achieve the objective, the robot is initially set to move towards the goal.
The robot will stop and re-route upon detecting the circular obstacle with the known radius. This will be calculated using the robot's buffer radius and the obstacle's radius, where,

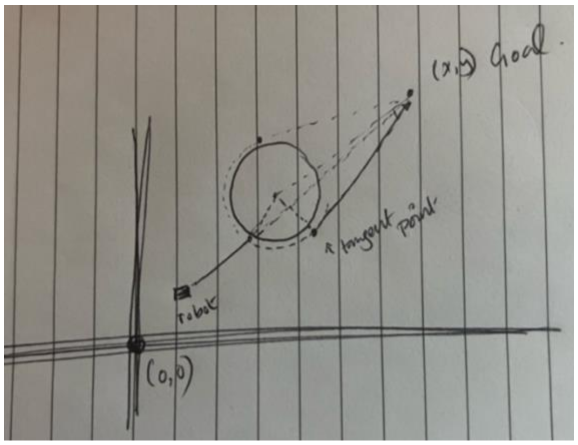
$$R_{effective} = R_{obstacle} + R_{robot}$$

We wull calculate the distance, d, from the robot's current position.

$$d = \sqrt{(r_x - x)^2 + \left(r_y - y\right)^2}$$

Where, r is the robot coordinates and x, y are the obstacles coordinates.
We compare this distance to the $R_{effective}$ as d < $R_{effective}$, and if d is less than R, then the robot has detected the obstacle.

Next, we determine the tangent points using trigonometric calculations, as showed in the below diagram. For simplicity, we only select a single tangent point and do not compare the best tangent point.



After the tangent point is reached, the robot does not encounter any further obstacle, and thus, can proceed towards the goal.

5. **Implement your bug0 algorithm for circular obstacles. Your code must accept a map of obstacles (x,y,r) and place them on the plane. You will then drive your robot to some start location and then run your code with a goal location. You may assume no other obstacles in the space. You may assume that the robot is circular with a known radius r. There is some code in cpmr_ch2 that you might find helpful. Submit screen shots of your code driving the robot to the goal. [6 marks]**

For this solution, I extended my solution from the previous question. I added two new functions, 'is_near_obstacle' and 'follow_boundary' that helps me achieve the outcome.

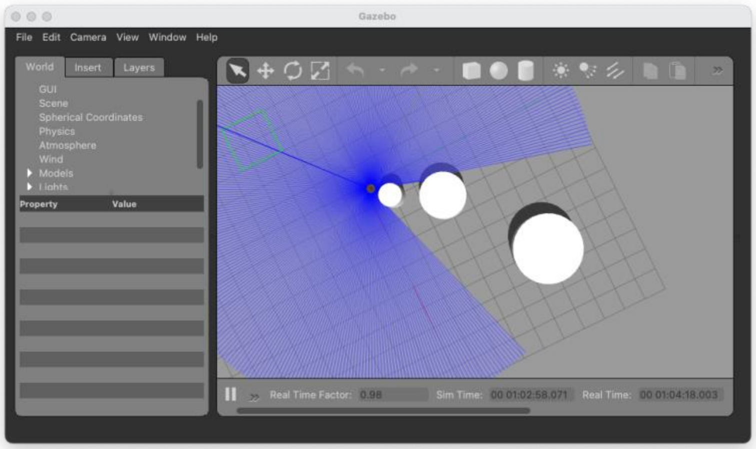
For the map, I added the config. of the obstacles into the default.json file. Also, I included intot the launch file of gazebo to load these maps automatically when I launch gazebo with the robot.

For simplicity, currently the map obstacles is manually configured into the main() function of drive_to_goal.py file.

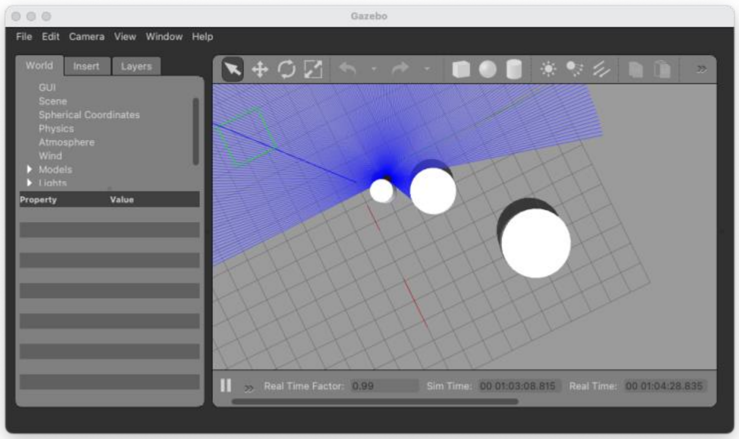
Below is the code for 'drive_to_goal.py' file and screenshots of the robot navigating through the obstacles.

```
import math
import numpy as np
import rclpy
from rclpy.node import Node
from rclpy.parameter import Parameter
from rcl_interfaces.msg import SetParametersResult
from nav_msgs.msg import Odometry
from geometry_msgs.msg import Twist, Pose, Point, Quaternion


def euler_from_quaternion(quaternion):
    """Converts quaternion (w in last place) to euler roll, pitch, yaw"""
    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')

        # Declare parameters for goal position and velocity
        self.declare_parameter('goal_x', 0.0)
        self.declare_parameter('goal_y', 0.0)
        self.declare_parameter('goal_t', 0.0)
        self.declare_parameter('vel_gain', 5.0)
        self.declare_parameter('max_vel', 0.4)

        self._goal_x = self.get_parameter('goal_x').get_parameter_value().double_value
        self._goal_y = self.get_parameter('goal_y').get_parameter_value().double_value
        self._goal_t = self.get_parameter('goal_t').get_parameter_value().double_value
        self._vel_gain = self.get_parameter('vel_gain').get_parameter_value().double_value
        self._max_vel = self.get_parameter('max_vel').get_parameter_value().double_value

        self.add_on_set_parameters_callback(self.parameter_callback)

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

        self.obstacles = []  # Obstacles list to be populated externally

    def _listener_callback(self, msg, max_pos_err=0.05):
        pose = msg.pose.pose
        cur_x = pose.position.x
        cur_y = pose.position.y
        o = pose.orientation
        roll, pitch, 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()
        vx = 0.0
        vy = 0.0

        if dist > max_pos_err:
            if self.is_near_obstacle(cur_x, cur_y):
                self.follow_boundary(cur_x, cur_y, cur_t)
                return
            else:
                # Move towards goal
                vx = x_diff * self._vel_gain
                vy = y_diff * self._vel_gain

                velocity_magnitude = math.sqrt(vx ** 2 + vy ** 2)
                if velocity_magnitude > self._max_vel:
                    scale = self._max_vel / velocity_magnitude
                    vx *= scale
                    vy *= scale

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



            self.get_logger().info(f"At ({cur_x},{cur_y},{cur_t}), goal ({self._goal_x},{self._goal_y},{self._goal_t}), velocity
({vx},{vy})")

        self._publisher.publish(twist)

    def is_near_obstacle(self, x, y):
        # Check if the robot is near any obstacles
        for obstacle in self.obstacles:
            ox, oy, r = obstacle
            distance = math.sqrt((x - ox) ** 2 + (y - oy) ** 2)
            if distance <= (r + 0.3):  # Buffer for robot radius. Currently, 0.3
                return True
        return False

    def follow_boundary(self, cur_x, cur_y, cur_t):
        # Follow the boundary of the closest obstacle
        for obstacle in self.obstacles:
            ox, oy, r = obstacle
            distance = math.sqrt((cur_x - ox) ** 2 + (cur_y - oy) ** 2)
```

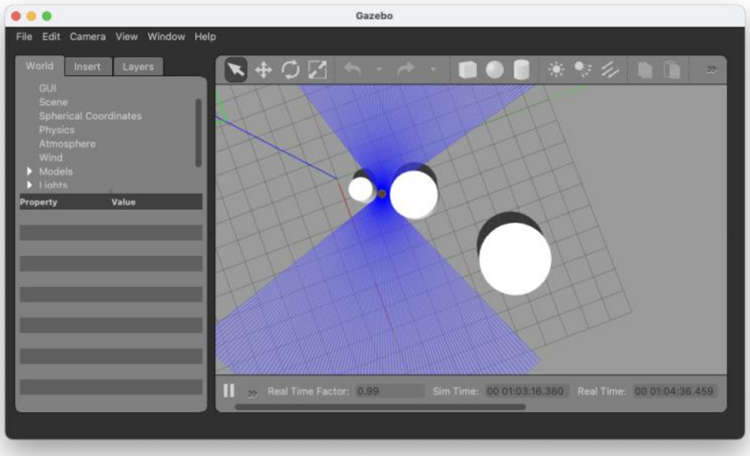
```python
            if distance <= (r + 0.3): # Buffer for robot radius. Currently, 0.3
                # Move robot along the obstacle boundary (simple circular motion)
                tangent_angle = math.atan2(oy - cur_y, ox - cur_x) + math.pi / 2
                boundary_vx = self._max_vel * math.cos(tangent_angle)
                boundary_vy = self._max_vel * math.sin(tangent_angle)

                twist = Twist()
                twist.linear.x = boundary_vx * math.cos(cur_t) + boundary_vy * math.sin(cur_t)
                twist.linear.y = -boundary_vx * math.sin(cur_t) + boundary_vy * math.cos(cur_t)


                self.get_logger().info(f"Following boundary at velocity ({twist.linear.x}, {twist.linear.y})")
                self._publisher.publish(twist)
                return


    def parameter_callback(self, params):
        for param in params:
            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
            elif param.name == 'vel_gain' and param.type_ == Parameter.Type.DOUBLE:
                self._vel_gain = param.value
            elif param.name == 'max_vel' and param.type_ == Parameter.Type.DOUBLE:
                self._max_vel = param.value
        return SetParametersResult(successful=True)


def main(args=None):
    rclpy.init(args=args)
    node = MoveToGoal()
    node.obstacles = [(1.0, 1.0, 0.5), (2.0, 3.0, 1.0), (6.0, 6.0, 1.5)]  # Defining obstacles (x, y, radius)
    rclpy.spin(node)
    rclpy.shutdown()


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



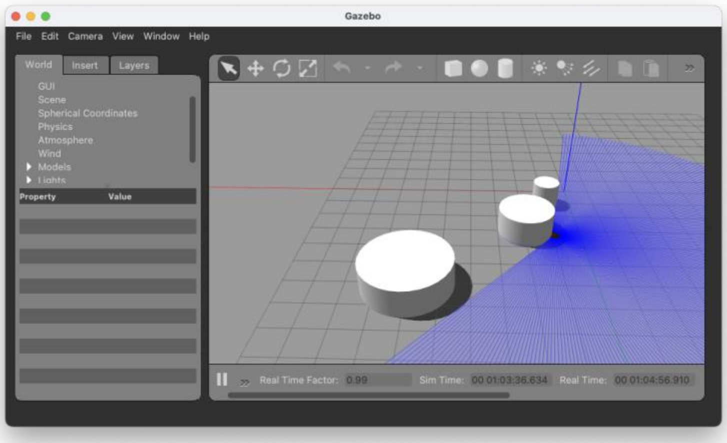
Real Time Factor: 0.98 Sim Time: 00 01:02:58.071 Real Time: 00 01:04:18.003



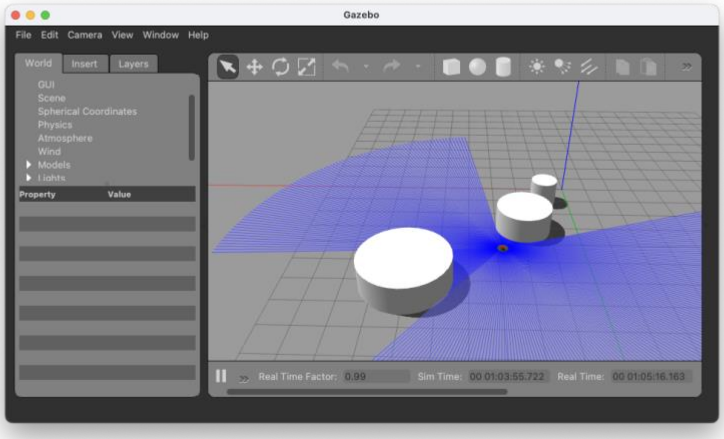
Real Time Factor: 0.99 Sim Time: 00 01:03:08.815 Real Time: 00 01:04:28.835



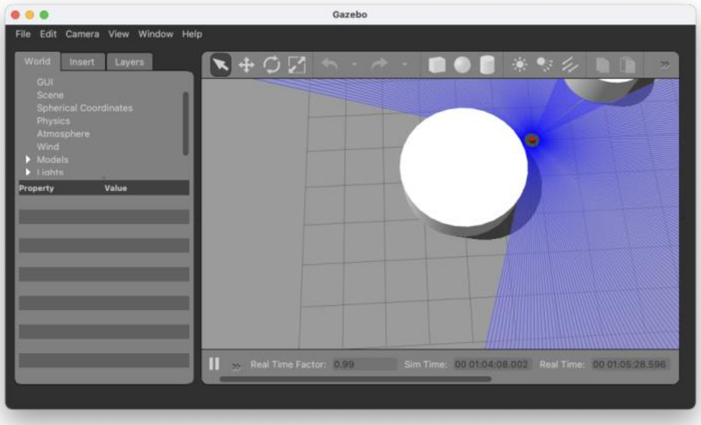
Real Time Factor: 0.99 Sim Time: 00 01:03:16.380 Real Time: 00 01:04:36.459



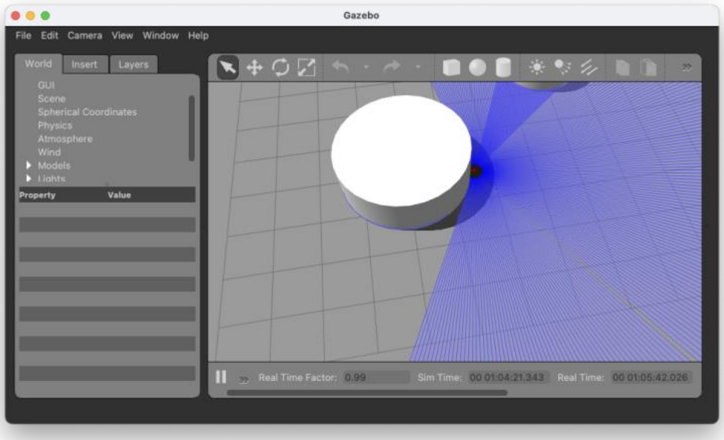
Real Time Factor: 0.99 Sim Time: 00 01:03:36.634 Real Time: 00 01:04:56.910



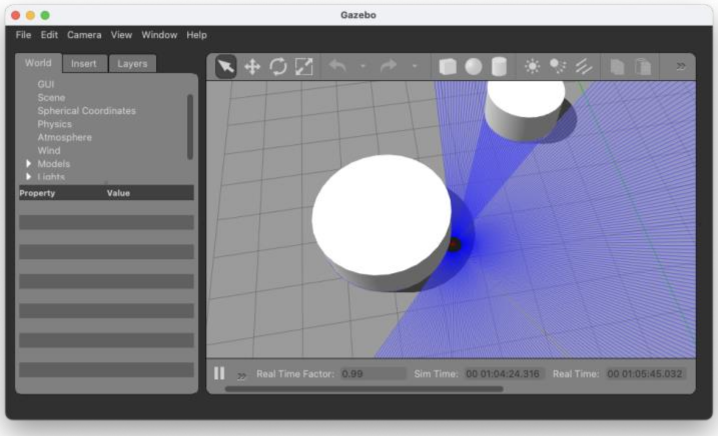
Real Time Factor: 0.99 Sim Time: 00 01:03:55.722 Real Time: 00 01:05:16.163



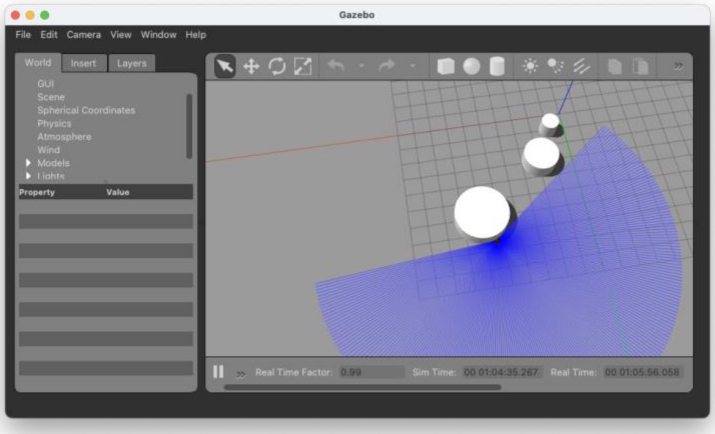
Real Time Factor: 0.99 Sim Time: 00 01:04:08.002 Real Time: 00 01:05:28.596



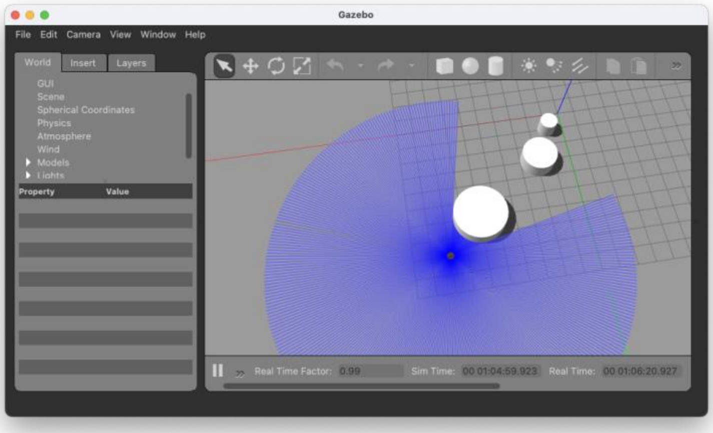
Real Time Factor: 0.99 Sim Time: 00 01:04:21.343 Real Time: 00 01:05:42.026



Real Time Factor: 0.99 Sim Time: 00 01:04:24.316 Real Time: 00 01:05:45.032



Real Time Factor: 0.99 Sim Time: 00 01:04:35.267 Real Time: 00 01:05:56.058



Real Time Factor: 0.99 Sim Time: 00 01:04:59.923 Real Time: 00 01:06:20.927