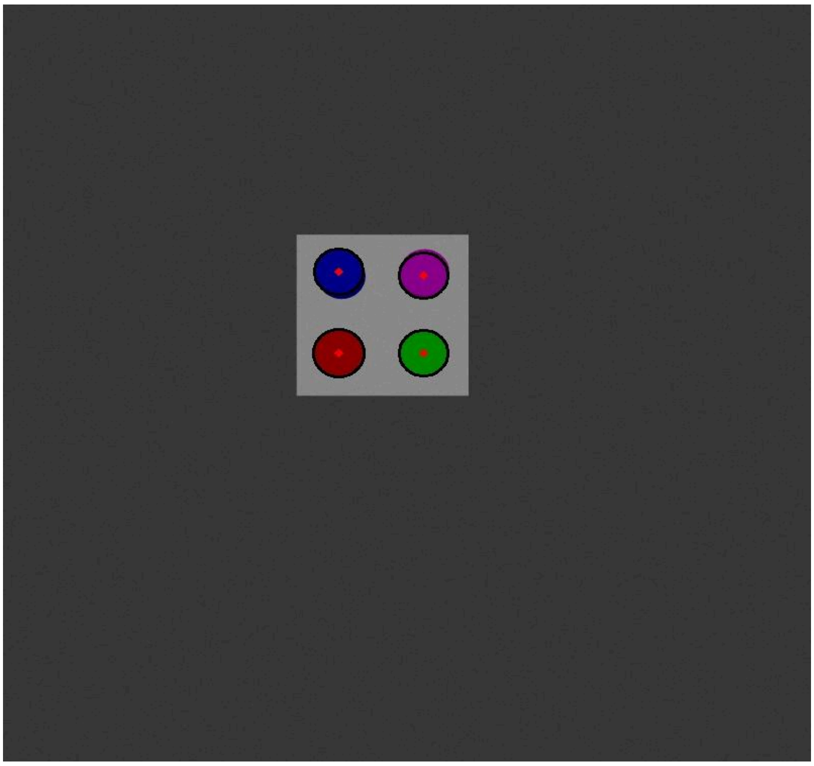
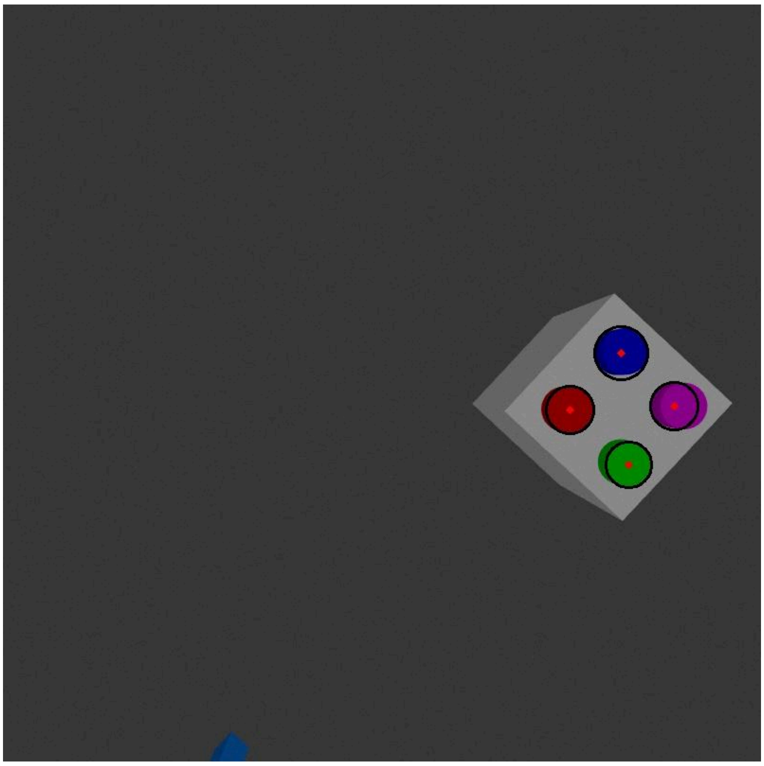
Report HW_5: **Farhan Seliya** and **Sarthak Mehta**

Step 1:



```python
def find_center(mask, color_name):
    # Get coordinates of all non-zero pixels in the mask
    y_indices, x_indices = np.where(mask > 0)
    if len(x_indices) > 0 and len(y_indices) > 0:
        cX = int(np.mean(x_indices))
        cY = int(np.mean(y_indices))

        centers[color_name]=(cX,cY)
        # Draw the center on the original image
        cv2.circle(current_frame, (cX, cY), 5, (255, 255, 255), -1)
        cv2.putText(current_frame, f"{color_name} center", (cX - 20, cY - 20), cv2.FONT_HERSHEY_SIMPLEX, 0.5, (255, 255, 255), 1)
    else :
        centers[color_name] = None  # No center found


combined_mask = cv2.bitwise_or(mask_blue, mask_red)
combined_mask = cv2.bitwise_or(combined_mask, mask_green)
combined_mask = cv2.bitwise_or(combined_mask, mask_pink)

find_center(mask_blue,'blue')
find_center(mask_red,'red')
find_center(mask_green,'green')
find_center(mask_pink,'pink')

for color, center in centers.items():
    if center:
        self.get_logger().info(f"Center of {color} found at: {center}")
    else:
        self.get_logger().info(f"Center of {color} not found")

# PLACE YOUR CODE HERE. PROCESS THE CURRENT FRAME AND PUBLISH IT. IF YOU ARE HAVING DIFFICULTY PUBLISHING IT YOU CAN USE THE FOLLOWING LINES T
cv2.imshow("output_image_centers", current_frame)
cv2.imwrite("initial_config.png", current_frame)
cv2.waitKey(1)
```

Step 2:



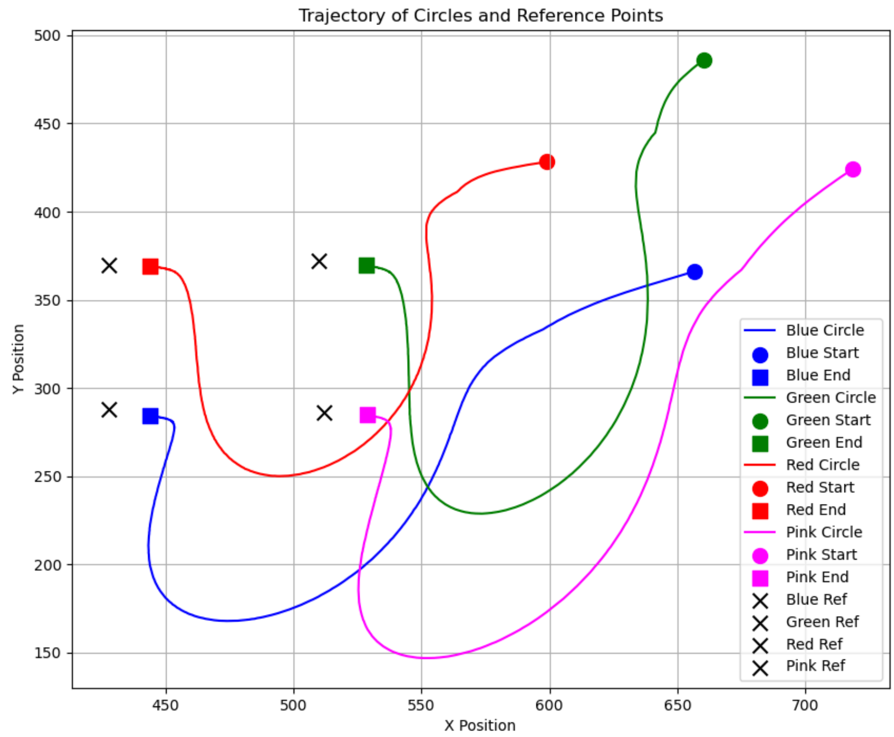
```python
def find_center(mask, color_name):
    # Get coordinates of all non-zero pixels in the mask
    y_indices, x_indices = np.where(mask > 0)
    if len(x_indices) > 0 and len(y_indices) > 0:
        cX = int(np.mean(x_indices))
        cY = int(np.mean(y_indices))

        centers[color_name]=(cX,cY)
        # Draw the center on the original image
        cv2.circle(current_frame, (cX, cY), 5, (255, 255, 255), -1)
        cv2.putText(current_frame, f"{color_name} center", (cX - 20, cY - 20), cv2.FONT_HERSHEY_SIMPLEX, 0.5, (255, 255, 255), 1)
    else :
        centers[color_name] = None  # No center found


combined_mask = cv2.bitwise_or(mask_blue, mask_red)
combined_mask = cv2.bitwise_or(combined_mask, mask_green)
combined_mask = cv2.bitwise_or(combined_mask, mask_pink)

find_center(mask_blue,'blue')
find_center(mask_red,'red')
find_center(mask_green,'green')
find_center(mask_pink,'pink')

for color, center in centers.items():
    if center:
        self.get_logger().info(f"Center of {color} found at: {center}")
    else:
        self.get_logger().info(f"Center of {color} not found")

# PLACE YOUR CODE HERE. PROCESS THE CURRENT FRAME AND PUBLISH IT. IF YOU ARE HAVING DIFFICULTY PUBLISHING IT YOU CAN USE THE FOLLOWING LINES T
cv2.imshow("output_image_centers", current_frame)
cv2.imwrite("initial_config.png", current_frame)
cv2.waitKey(1)
```

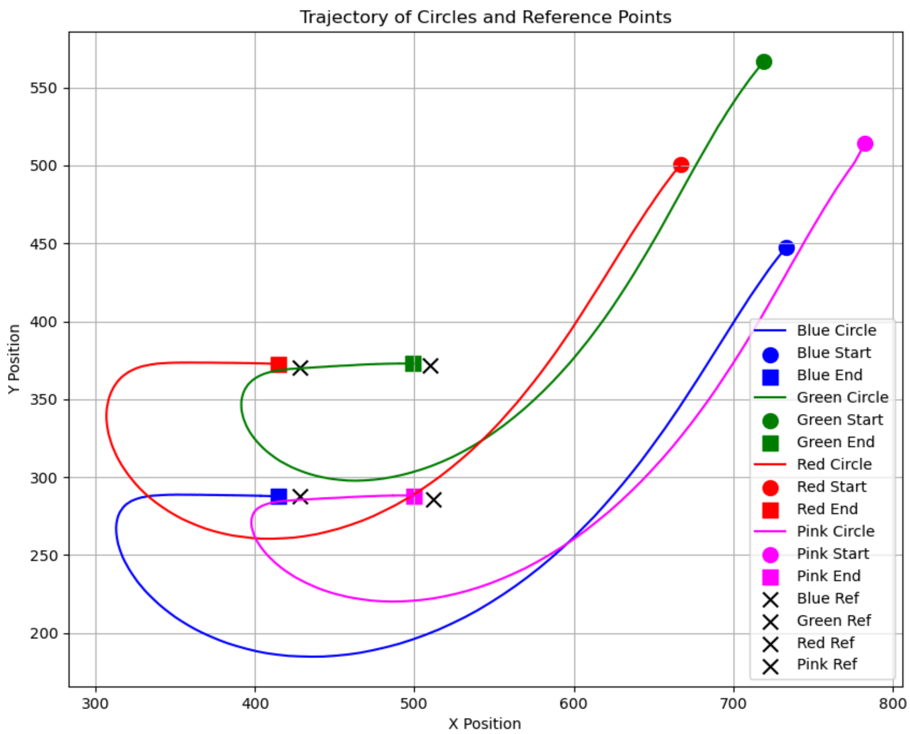
<u>The above was done using the command</u>:

ros2 topic pub /forward_position_controller/commands std_msgs/msg/Float64MultiArray "{data: [0.25, -1.0]}"

The above is also the position of the current situation, i.e. [0.25, -1.0]

Step 3:





Two different trajectories are for the same desired position but different start positions.

```python
# Basic ROS 2 program for subscribing to real-time image streaming
# This code is a modified version by Berk Calli from the original author, Addison
Sears-Collins
# - Reference: https://automaticaddison.com

# Import necessary libraries
import rclpy  # ROS 2 Python client library
from rclpy.node import Node  # Base class for creating ROS nodes
from sensor_msgs.msg import Image  # Message type for image data
from cv_bridge import CvBridge  # Utility to convert ROS images to OpenCV format
import cv2  # OpenCV library for image processing
import numpy as np  # Library for numerical operations
import math  # Math library for mathematical operations
from enum import Enum  # Enumeration support
from std_msgs.msg import Int32MultiArray, Float64MultiArray  # Message types for
integer and float arrays
from sensor_msgs.msg import JointState  # Message type for joint states


# Define an enumeration for color identification
class ColorCategories(Enum):
   ERROR = 0
   BLUE = 1
   GREEN = 2
   RED = 3
   PURPLE = 4


# Function to convert pixel coordinates to real-world units
def ConvertPixelToUnit(x_pixel, y_pixel):
   pixel_to_unit = 0.0029411765  # Conversion factor from pixels to units
   origin_x = 400  # Origin x-coordinate in pixels
   origin_y = 400  # Origin y-coordinate in pixels
   real_x = (-(x_pixel - origin_x)) * pixel_to_unit  # Convert X pixel to real-world
X
   real_y = (-(y_pixel - origin_y)) * pixel_to_unit  # Convert Y pixel to real-world
Y
   return real_x, real_y  # Return the real-world coordinates


# Class to handle position calculations
class PositionCalculator():
   # Define constants for conversion and reference positions
   pixel_to_unit = 0.0029411765
   origin_x = 400
   origin_y = 400

   # Convert reference pixel coordinates to real-world units
```

```
    purple_x, purple_y = ConvertPixelToUnit(512, 286)
    blue_x, blue_y = ConvertPixelToUnit(428, 288)
    green_x, green_y = ConvertPixelToUnit(510, 372)
    red_x, red_y = ConvertPixelToUnit(428, 370)

    # Store reference positions and associated color values
    reference_positions = [
        blue_x, blue_y, ColorCategories.BLUE.value,
        green_x, green_y, ColorCategories.GREEN.value,
        red_x, red_y, ColorCategories.RED.value,
        purple_x, purple_y, ColorCategories.PURPLE.value
    ]

    @staticmethod
    def CalculatePositionDelta(current, target):
        """Compute the position deltas between current and target positions."""
        delta_values = np.array([], dtype=np.int32)
        index_target = 0
        while index_target < len(target) - 2:  # Ensure we don't exceed bounds
            index_current = 0
            while index_current < len(current) - 2:  # Ensure we don't exceed bounds
                if target[index_target + 2] == current[index_current + 2]:  # Match
colors
                    current_x, current_y = ConvertPixelToUnit(current[index_current],
current[index_current + 1])  # Get current real coordinates
                    delta_values = np.append(delta_values, (current_x -
target[index_target]))  # Calculate X delta
                    delta_values = np.append(delta_values, (current_y -
target[index_target + 1]))  # Calculate Y delta
                    break  # Exit inner loop on match
                index_current += 3  # Move to next color
            index_target += 3  # Move to next target color
        delta_values = delta_values.reshape(-1, 1)  # Reshape for output
        return delta_values  # Return deltas

    @staticmethod
    def ComputeJacobian(current):
        """Compute the pseudoinverse of the Jacobian matrix based on current
positions."""
        # Initialize color positions
        blue_coords = [0.0, 0.0]  # Blue
        green_coords = [0.0, 0.0]  # Green
        red_coords = [0.0, 0.0]  # Red
        purple_coords = [0.0, 0.0]  # Purple
        index = 0

        # Assign current real-world coordinates to corresponding colors
        while index < len(current):
```

```
            current_x, current_y = ConvertPixelToUnit(current[index], current[index +
1])
            if current[index + 2] == ColorCategories.BLUE.value:
                blue_coords[0], blue_coords[1] = current_x, current_y
            elif current[index + 2] == ColorCategories.GREEN.value:
                green_coords[0], green_coords[1] = current_x, current_y
            elif current[index + 2] == ColorCategories.RED.value:
                red_coords[0], red_coords[1] = current_x, current_y
            elif current[index + 2] == ColorCategories.PURPLE.value:
                purple_coords[0], purple_coords[1] = current_x, current_y
            index += 3  # Move to next color

        # Construct the Jacobian matrix
        jacobian_matrix = [
            [-1, 0, blue_coords[1]], [0, -1, -blue_coords[0]],
            [-1, 0, green_coords[1]], [0, -1, -green_coords[0]],
            [-1, 0, red_coords[1]], [0, -1, -red_coords[0]],
            [-1, 0, purple_coords[1]], [0, -1, -purple_coords[0]]
        ]

        pseudo_inverse = np.linalg.pinv(jacobian_matrix)  # Compute the pseudoinverse
        return pseudo_inverse  # Return the pseudoinverse


# Main class for the calculation node
class MotionCalculatorNode(Node):

    def __init__(self):
        super().__init__('MotionCalculatorNode')  # Initialize the node with a name
        self.joint_angles = [0, 0]  # Initialize joint angles
        self.cv_bridge = CvBridge()  # Instantiate the CvBridge

        # Create subscribers to relevant topics
        self.detected_center_subscriber = self.create_subscription(
            Float64MultiArray, 'detected_center', self.process_detected_centers, 10)

        self.joint_state_subscriber = self.create_subscription(
            JointState, '/joint_states', self.update_joint_states, 10)

        # Create publishers for output topics
        self.image_publisher = self.create_publisher(Image, 'output_image', 10)
        self.delta_coords_publisher = self.create_publisher(Int32MultiArray,
'delta_coords', 10)
        self.velocity_publisher = self.create_publisher(Float64MultiArray,
'/forward_velocity_controller/commands', 10)

    def CalculateMotionJacobian(self, angle1, angle2):
        """Calculate the motion Jacobian for the given joint angles."""
```

```
        link1_length = 1  # Length of the first link
        link2_length = 1  # Length of the second link
        # Calculate the Jacobian elements based on the angles
        jacobian_row1 = link1_length * math.sin(angle1) + link2_length *
math.sin(angle1 + angle2)
        jacobian_row2 = link2_length * math.sin(angle1 + angle2)
        jacobian_col1 = -link1_length * math.cos(angle1) - link2_length *
math.cos(angle1 + angle2)
        jacobian_col2 = -link2_length * math.cos(angle1 + angle2)
        return np.array([[jacobian_row1, jacobian_col1], [jacobian_row2,
jacobian_col2], [-1, -1]])  # Return the Jacobian matrix

    def AppendMatrixToFile(self, matrix, file_name):
        """Append a matrix to a specified text file."""
        with open(file_name, 'a') as file:  # Open file in append mode
            np.savetxt(file, matrix, delimiter=',')  # Save matrix to file
            file.write("\n")  # Add a newline for separation

    def update_joint_states(self, data):
        """Callback function to handle incoming joint state data."""
        self.joint_angles[0] = data.position[0]  # Update first joint angle
        self.joint_angles[1] = data.position[1]  # Update second joint angle

    def process_detected_centers(self, data):
        """Callback function to handle incoming detected center data."""
        current_positions = np.array(data.data)  # Convert incoming data to a NumPy
array

        # Save current positions to a file for trajectory tracking
        self.AppendMatrixToFile(current_positions, 'trajectory_1.txt')

        # Calculate deltas between current and reference positions
        position_deltas =
PositionCalculator.CalculatePositionDelta(current_positions,
PositionCalculator.reference_positions)

        # Save delta values to a file
        self.AppendMatrixToFile(position_deltas, 'error_1.txt')

        print(position_deltas)  # Display the delta values
        print("Robot Moving...")

        if len(position_deltas) == 8:  # Ensure there are enough deltas for
processing
            print(position_deltas)
            jacobian_pseudo_inverse =
PositionCalculator.ComputeJacobian(current_positions)  # Compute the Jacobian
pseudoinverse
```

```
            velocity_command = np.matmul(jacobian_pseudo_inverse, position_deltas)  # 
Calculate the velocity commands
            motion_jacobian = self.CalculateMotionJacobian(self.joint_angles[0], 
self.joint_angles[1])  # Calculate the motion Jacobian
            joint_velocity_commands = np.matmul(np.linalg.pinv(motion_jacobian), 
velocity_command)  # Calculate joint velocity commands

            # Create a message for joint velocities
            joint_velocity_message = Float64MultiArray()
            joint_velocity_message.data = [float(joint_velocity_commands[0]), 
float(joint_velocity_commands[1])]
            self.velocity_publisher.publish(joint_velocity_message)  # Publish the 
joint velocity commands
            print("Velocity Publishing")


# Main function for executing the script
def main(args=None):
   rclpy.init(args=args)  # Initialize the ROS client library
   motion_calculator_node = MotionCalculatorNode()  # Create an instance of the 
motion calculator node

   # Keep the node active to process callbacks
   rclpy.spin(motion_calculator_node)

   # Clean up the nodes
   motion_calculator_node.destroy_node()  # Destroy the motion calculator node

   # Shut down the ROS client library
   rclpy.shutdown()


# Entry point of the script
if __name__ == '__main__':
   main()  # Execute the main function
```

To run the package:

1. Start by running the gazebo package for spawning the robot.
   Using the following command:

```
ros2 launch rrbot_gazebo rrbot_world.launch.py
```

2. Spawn the object in the gazebo environment.
   Using the command:

```
ros2 launch rrbot_gazebo object_spawn.launch.py
```

3. Move the robot to an initial position.
   Using the following command:

```
ros2 topic pub /forward_position_controller/commands std_msgs/msg/Float64MultiArray
"{data: [0.25, -1.0]}"
```

4. Start the velocity controller

```
ros2 control switch_controllers --start forward_velocity_controller --stop
forward_position_controller
```

5. Run the python node "**feature_points.py**" inside the **src/opencv_test_py/opencv_test_py.**

6. Run the python node "**visual_servoing.py**" inside the same package as above.