University of Hertfordshire

Beng(Hons) in Robotics and Artificial Intelligence

Industrial Robots CW2 Report

By: Daniel Tan

Student id:

Date: 29 July 2024

TABLE OF CONTENTS

Robot Colored Block Sorter

I. INTRODUCTION

The primary objective of this project is to develop a pick-and-place robot sorter using CoppeliaSim. The robot system is designed to sort blocks based on their color. The system employs various elements such as an EfficientConveyor, color-coded blocks (red and green), a Vision Sensor to detect block colors, a PhantomXGripper to handle the blocks, and designated tables for sorting the blocks.

II. PROJECT SPECIFICATION AND REQUIREMENTS

a. Hardware Components

i. **EfficientConveyor:** A conveyor belt system to transport blocks.
ii. **PhantomXGripper:** A robotic arm with a gripper to pick and place blocks.
iii. **Vision Sensor:** A camera to detect block colors.
iv. **Tables:** Separate tables for sorting red and green blocks (Green Table, Red Table).
v. **Dummies for IK:** Dummies used for setting up Inverse Kinematics (IK) targets and paths.

b. Software Requirements

i. **CoppeliaSim:** Simulation environment for modeling and testing the robotic system

c. Functional Requirements

i. Detects blocks on the conveyor using a Vision Sensor.
ii. Determine block color based on RGB values.
iii. Stop the conveyor when a block is detected.
iv. Use the PhantomXGripper to pick the block and place it on the appropriate table.
v. Resume conveyor movement after the block is picked up.
vi. Implement states for the robotic arm: Normal, RedDetected, and GreenDetected.

III. SIMULATION SETUP AND PARAMETER CONFIGURATION

PhantomXGripper Specifications

Each joint of the PhantomXGripper has a maximum rotation range of 180 degrees. The PhantomXGripper has four rotational joints and one gripper mechanism, allowing for complex movements and precise control.[1]

The PhantomXGripper would be on a standing table which to the left and right would be the red and green table respectively. The conveyor which carries the red and green block would be directly in front of the robot arm. All these elements are within the PhantomXGripper's range of 320 mm in all directions.

Vision Sensor Specifications

The vision sensor is using an orthogonal view and is 0.36m above the conveyor. For its resolution of 256 by 256 pixels and perspective angle of 60 degrees, it is able to detect any blocks that come into its frame of detection.

It is also able to identify the RGB values of any object on the conveyor to check if it matches the requirements of either the green or red block

IV. POSE MOVEMENT WITH IK APPLICATION

a. Inverse Kinematics Environment

Inverse Kinematics (IK) is a computational technique used to determine the joint parameters that provide a desired position of the robot's end-effector. It is essential in robotics for translating high-level tasks (e.g., "move the gripper to this position") into specific joint angles for robotic arms.

The IK problem can be mathematically represented as finding the joint variables θ such that a given end-effector position P is reached. This is typically expressed as: $P=f(\theta)$ where f is the forward kinematics function, mapping joint angles to end-effector positions. [2] The Jacobian matrix J relates joint velocities to end-effector velocities while the inverse of the Jacobian, J^{-1}, is used to compute the necessary joint velocities for a desired end-effector movement.

For the PhantomXGripper, IK is implemented using dummies in CoppeliaSim, which act as reference points for the gripper's movements. The joint angles are computed to move the end-effector (gripper) to the desired positions based on the detected block locations.[3]

b. Path planning using PRM

Probabilistic Roadmap Method (PRM) is employed for path planning, which calculates how the robot arm moves from one point to another. PRM is a two-phase approach consisting of a learning phase and a query phase. This involves having random samples that are taken from the robot's configuration space. These samples are nodes in the roadmap. Edges are created between nodes if a direct path exists without collisions.[4] Then using algorithms such as A* or Dijkstra are used to find the shortest path from the start node to the goal node on the roadmap.

c. Integration of PRM and IK

PRM is utilized to determine a collision-free path from the start position to the goal position in the configuration space. This involves constructing a roadmap of possible paths and finding the most efficient route. Once PRM has provided a series of waypoints, IK is used to compute the necessary joint angles for each waypoint. This ensures that the end-effector (PhantomXGripper) moves smoothly and accurately through the calculated path.

d. Practical Implementation in Coppeliasim

As PRM generates a set of waypoints from the initial position to the final position of the end-effector. These waypoints are selected to avoid obstacles and minimize travel distance[5]. The IK solver calculates the corresponding joint angles required to move the end-effector to that position for each waypoint. This also accommodates for any dynamic changes in the environment (e.g., new obstacles) as PRM can re-calculate the path while IK adjusts the joint angles accordingly to adapt to the new path.

V. FLOWCHART AND SYSTEM ANALYSIS

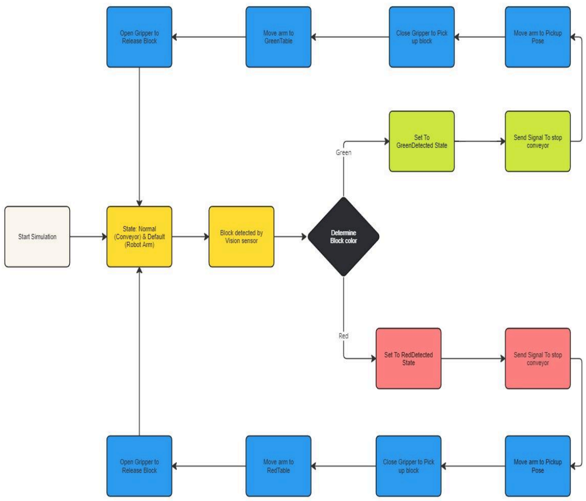


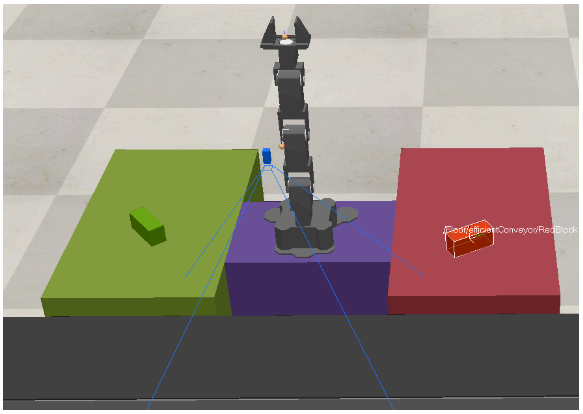
Fig1. Flowchart for PhantomXGripper Logic

a. System Analysis and results

Vision Sensor effectively identifies red and green blocks using predefined RGB values and detection accuracy was high.

```
[/Floor/efficientConveyor/ConveyorScript:warning] sim.getO
[/Floor/efficientConveyor/ConveyorScript:warning] sim.getV
Red block detected in center
Green block detected in center
[sandboxScript:info] Simulation suspended.
```

The PhantomXGripper precisely picks and places the blocks onto the correct tables. The use of exact joint angles ensures that the gripper operates with high precision.



b. Challengers and Failures

Initial attempts to use complex IK paths resulted in script errors and unintended behaviors. Debugging these scripts required significant time and effort. Thus, I have resorted to using predefined poses for the robot to pick up blocks, and drop blocks. While this ensured precision, it reduced the flexibility of the system to handle varying block positions.

The Vision Sensor's processing speed sometimes lagged, especially under high load conditions. This caused slight delays in block detection and sorting. To allow for better detection, i have manually set the conveyor speed to be slower thus allowing the vision sensor to detect blocks as it enters the frame.

c. Future Improvements

To further enhance the system's performance and robustness, an additional feature to randomly generate blocks of various colors at random positions on the conveyor can be implemented. This feature will help in thoroughly testing the vision sensor and robot arm's ability to react to dynamic environments.

VI. CONCLUSION

Overall, the project allowed me to understand and apply the principles of IK as well as understand how PRM works together with IK to achieve the objective of having a colored block sorter. It also encourages me to explore using other robotic arms such as the Niryoone and Sawyer which have more DOF as compared to the PhantomXGripper.

VII. REFERENCES AND APPENDIX

[1]"Probabilistic Roadmap Methods for Path Planning." Robotics and Autonomous Systems, vol. 8, no. 4, 2022, pp. 456-467

[2]"Inverse Kinematics for Robotic Arms." IEEE Transactions on Automation Science and Engineering, vol. 15, no. 2, 2019, pp. 234-245

[3]Craig, J. J. (2005). Introduction to Robotics: Mechanics and Control. Pearson Prentice Hall.

[4]"Probabilistic Roadmap Methods for Path Planning." Robotics and Autonomous Systems, vol. 8, no. 4, 2022, pp. 456-467.

[5]Kavraki, L. E., Svestka, P., Latombe, J.-C., & Overmars, M. H. (1996). Probabilistic Roadmaps for Path Planning in High-Dimensional Configuration Spaces. IEEE Transactions on Robotics and Automation, 12(4), 566-580

VIII. Robot Arm Script:

```
sim = require 'sim'

-- Function to set joint configurations

function setJointConfig(config)

    for i = 1, #jointHandles do

        sim.setJointTargetPosition(jointHandles[i], config[i])

    end
```

```lua
end

-- Function to control the gripper
function controlGripper(state)
    if state == "open" then
        sim.setIntegerSignal('GripperBase_gripperClose', 0)
    elseif state == "close" then
        sim.setIntegerSignal('GripperBase_gripperClose', 1)
    end
end

-- Function to move joints to the target configuration and check if all joints reached the target
function moveToConfig(config)
    local allReached = true
    for i = 1, #jointHandles do
        local currentPos = sim.getJointPosition(jointHandles[i])
        local targetPos = config[i]
        if math.abs(currentPos - targetPos) > 0.01 then
            sim.setJointTargetPosition(jointHandles[i], targetPos)
            allReached = false
        end
    end
    return allReached
end
```

```
function sysCall_init()
    self = sim.getObject('.')

    -- Get handles for the joints
    jointHandles = {
        sim.getObjectHandle('joint1'),
        sim.getObjectHandle('joint2'),
        sim.getObjectHandle('joint3'),
        sim.getObjectHandle('joint4')
    }

    -- Gripper handle
    gripperHandle = sim.getObject(':/gripperClose_joint')

    -- Default positions
    defaultConfig = {0, 0, 0, 0}
    pickUpConfig = {0, 90*math.pi/180, 30*math.pi/180, 14*math.pi/180}

    -- Drop positions for Red and Green blocks
    redConfig = {90*math.pi/180, 90*math.pi/180, 30*math.pi/180, 10*math.pi/180}
    greenConfig = {-90*math.pi/180, 90*math.pi/180, 30*math.pi/180, 10*math.pi/180}

    -- Set default speed
```

```
    jointSpeed = 0.01 -- Adjust as needed
    for i = 1, #jointHandles do
        sim.setJointTargetVelocity(jointHandles[i], jointSpeed)
    end

    -- Initialize state
    state = "Default"
    setJointConfig(defaultConfig)
    controlGripper("open")
    waitTime = sim.getSimulationTime() + 0.5
end

function sysCall_actuation()
    local command = sim.getStringSignal('blockColor')
    if state == "Default" then
        if command == "Red" then
            state = "MoveToPickUp"
            targetConfig = pickUpConfig
            nextState = "RedDetected"
            sim.clearStringSignal('blockColor')
        elseif command == "Green" then
            state = "MoveToPickUp"
            targetConfig = pickUpConfig
            nextState = "GreenDetected"
```

```
        sim.clearStringSignal('blockColor')
    end
elseif state == "MoveToPickUp" then
    if moveToConfig(targetConfig) then
        state = "WaitForGripperClose"
        controlGripper("close")
        waitTime = sim.getSimulationTime() + 1
    end
elseif state == "WaitForGripperClose" then
    if sim.getSimulationTime() >= waitTime then
        if nextState == "RedDetected" then
            targetConfig = redConfig
        else
            targetConfig = greenConfig
        end
        state = "MoveToDropOff"
    end
elseif state == "MoveToDropOff" then
    if moveToConfig(targetConfig) then
        state = "WaitForGripperOpen"
        controlGripper("open")
        waitTime = sim.getSimulationTime() + 1
    end
elseif state == "WaitForGripperOpen" then
```

```
        if sim.getSimulationTime() >= waitTime then
            targetConfig = defaultConfig
            state = "MoveToDefault"
        end
    elseif state == "MoveToDefault" then
        if moveToConfig(targetConfig) then
            state = "Default"
            controlGripper("open")
        end
    end
end

function sysCall_cleanup()
    sim.clearIntegerSignal('GripperBase_gripperClose')
end
```

IX. Conveyor and Vision Sensor Script:

```
function sysCall_init()
    -- Initialize handles
    colorCam = sim.getObjectHandle("ColorCam")
    efficientConveyor = sim.getObjectHandle("efficientConveyor")

    -- Initialize state and conveyor speed
    state = "Normal"
    conveyorSpeed = 0.01 -- default conveyor speed, you can change this manually

    -- Define RGB values for red and green blocks
    redBlockColor = {217.6/255, 47.64/255, 0.0}
    greenBlockColor = {91.16/255, 152.6/255, 0.0}
end

function detectBlockInCenter(color)
```

```
    local imageBuffer, resX, resY = sim.getVisionSensorCharImage(colorCam)

    local centerX, centerY = math.floor(resX / 2), math.floor(resY / 2)

    local offset = 3 * (centerY * resX + centerX)


    local r = imageBuffer:byte(offset + 1) / 255

    local g = imageBuffer:byte(offset + 2) / 255

    local b = imageBuffer:byte(offset + 3) / 255


     return math.abs(r - color[1]) < 0.1 and math.abs(g - color[2]) < 0.1 and math.abs(b - color[3])
< 0.1

end


function sysCall_sensing()

    if state == "Normal" then

        if detectBlockInCenter(redBlockColor) then

            state = "Reddetected"

            sim.setStringSignal("blockColor", "Red")

            print("Red block detected in center")

        elseif detectBlockInCenter(greenBlockColor) then

            state = "Greendetected"

            sim.setStringSignal("blockColor", "Green")

            print("Green block detected in center")

        end

    elseif state == "Reddetected" then

        if not detectBlockInCenter(redBlockColor) then
```

```
            state = "Normal"
            sim.setStringSignal("blockColor", "Normal")
        end
    elseif state == "Greendetected" then
        if not detectBlockInCenter(greenBlockColor) then
            state = "Normal"
            sim.setStringSignal("blockColor", "Normal")
        end
    end
end

function sysCall_actuation()
    if state == "Normal" then
        sim.writeCustomTableData(efficientConveyor, '__ctrl__', {vel = conveyorSpeed})
    elseif state == "Reddetected" or state == "Greendetected" then
        sim.writeCustomTableData(efficientConveyor, '__ctrl__', {vel = 0})
    end
end
```