Cognitive Systems and Robotics

Robotic Object Recognition and Sorting: A youBot Approach

Siavash Mortaz Hejri
MSc Artificial Intelligence
Sheffield Hallam University

Introduction

Today, robots and artificial intelligence have seamlessly integrated into daily life, becoming essential contributors to routine tasks. Their applications span household chores, caregiving, education, research, medicine, and various other domains. Continuous technological advancements enhance these machines, simplifying human life and improving task efficiency. Industries experience heightened productivity, quality, and quantity, surpassing previous efficiency levels. Specialized robots, like the KUKA youBot, exemplify this trend, executing meticulous tasks with 24-hour operational capabilities and minimal errors. The integration of robots and AI reflects a transformative shift, profoundly impacting diverse sectors and contributing to an increasingly automated and efficient world.

The youBot, designed by KUKA, represents a cutting-edge robotic platform for mobile manipulation research and education. Featuring an omni-directional mobile base and a 5-degree-of-freedom arm with a two-finger gripper, the youBot boasts advanced capabilities. Despite operating on an open-source control framework, suboptimal performance is noted in current kinematics and trajectory planning methods. Specifically, the youBot's arm trajectory, from the initial to camera-ready position for position detection, is excessively long. In response, (Zhang, Liandong and Zhou, Changjiu, 2023) propose a geodesic-based approach for shortest path planning, aiming to enhance efficiency and offer a more time-effective solution for the youBot's applications.

In my lifting project, precise end-effector trajectory tracking is essential. Controlling individual manipulator joints while specifying Cartesian end-effector trajectories introduces complexity, where small joint position errors accumulate into significant positional discrepancies. The relationship between end-effector and manipulator joint positions depends on the manipulator's configuration. To address these challenges, the project employs a KUKA youBot for accurate and rapid grasping. Unlike existing control schemes that neglect arm configuration and dynamics, the project introduces a torque controller based on the youBot arm's dynamical model, achieving a tenfold increase in grasping speed with similar tracking performance. This approach is demonstrated through autonomous grasping of objects detected by an onboard Kinect-like sensor, showcasing the youBot's ability to grasp objects while in motion (Müggler, E., Fässler, M., Scaramuzza, D., Huck, S., Lygeros, J., 2013).

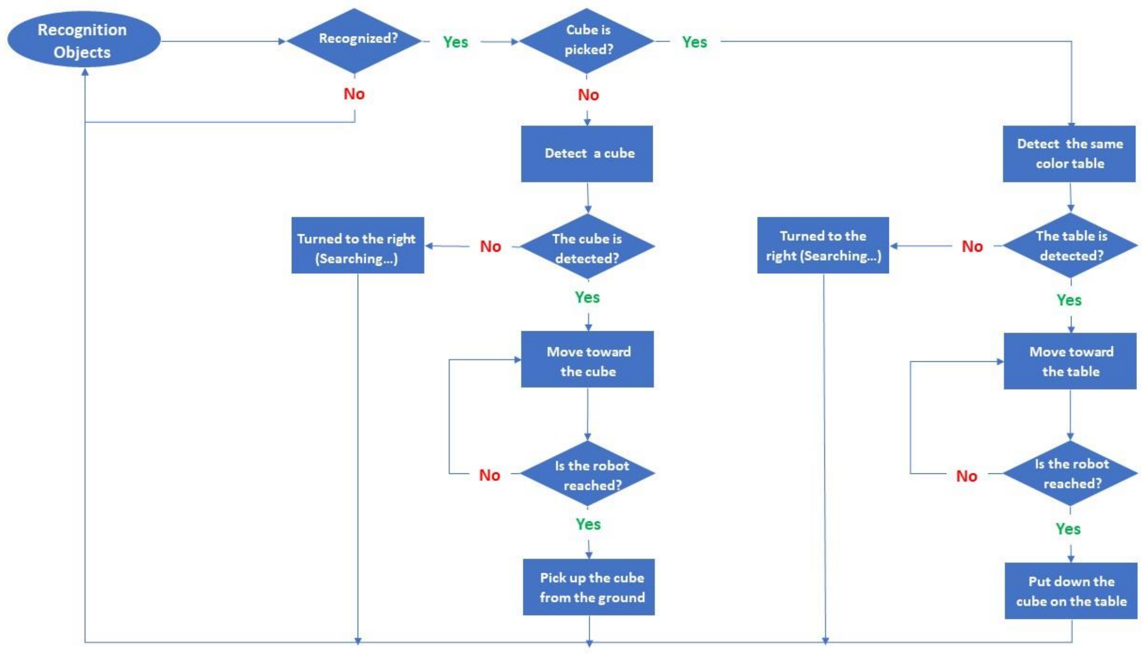
In their study, (Kumar, Rahul and Lal, Sunil and Kumar, Sanjesh and Chand, Praneel, 2014), focus on object detection for a pick-and-place robotic arm dedicated to sorting tasks. Their research prioritizes developing a robust image processing algorithm, encompassing object detection via a feature extraction algorithm, object recognition through classification, and communication of object type and coordinates for robotic arm execution. Challenges include potential pixel data loss during resizing, mitigated by adopting a centered image approach. The classifier achieves 99.33% accuracy, while the feature extraction algorithm demonstrates 83.6443% accuracy. Following experimentation, the overall system performance of the image processing algorithm reaches 82.7162%.

In 2018, (Adamov, 2018) addresses challenges related to the mobility of the KUKA youBot, specifically its mecanum wheel, in his work. The study focuses on the omnidirectional platform with two pairs of mecanum wheels, conducting a kinematic analysis considering wheel rollers and wheel geometry. It explores the precision of an algorithm for calculating mobile platform coordinates based on odometric information, disregarding wheel construction. Additionally, an algorithm is proposed to improve odometric navigation accuracy.

In the specific context of my project, the youBot is employed to recognize objects based on their color and subsequently place them on the appropriate table.

Exploring Developed Routine: Strategies and Innovations

Logical Path

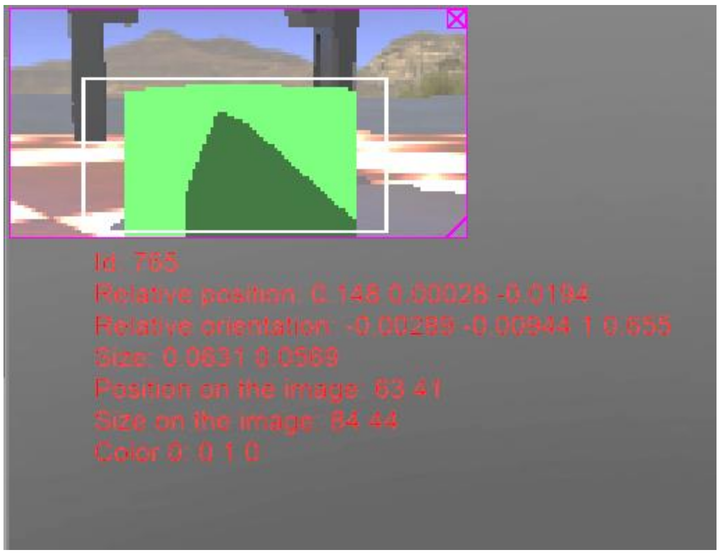


The diagram illustrates the logical progression of my project.

The strategic workflow involves the robot initially surveying the environment via camera recognition color to identify objects. Subsequently, the robot must distinguish between tables and boxes (cubes) and recognize their respective colors. After identifying an object and determining its nature, the robot is guided to select a specific box, such as a red one, tracking, moving toward, and picking it up. This process repeats as the robot searches for an appropriate table, exemplified by a red table. Once located, the robot places the red box on it, resumes searching for other boxes, and positions them on designated tables. Achieving these goals requires a deeper exploration of camera recognition objects.

The camera utilizes "getRecognitionObjects()" function, returning an object with various parameters:

The 'id' denotes the node id corresponding to the object, which can be directly utilized in the 'wb_supervisor_node_get_from_id' supervisor function. Position and orientation are relative to the camera, with units in meters and radians. The 'size' parameter indicates the Y and Z sizes in meters relative to the camera (depth along the Camera X axis is indeterminable). 'Position_on_image' and 'size_on_image' help determine the object's bounding box in the camera image, measured in pixels. 'Number_of_colors' and 'colors' provide the count of object colors and a pointer to the colors array. Each color is represented by three doubles (R, G, and B), making the array size equal to 3 * 'number_of_colors'. Lastly, 'model' returns the model field of the Solid node. (Webots Reference Manual, n.d.)

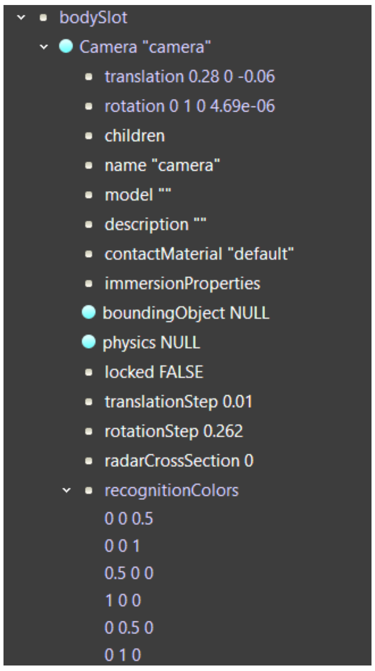


```
Id: 765
Relative position: 0.148 0.00028 -0.0194
Relative orientation: -0.00289 -0.00944 1 0.655
Size: 0.0631 0.0569
Position on the image: 63 41
Size on the image: 84 44
Color 0: 0 1 0
```

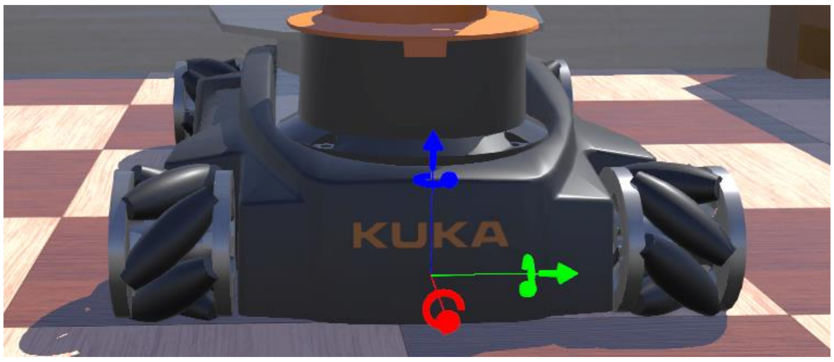
Hence, if the camera fails to detect any objects, the length of the object array obtained from "getRecognitionObjects()" will be zero or null; otherwise, it contains pertinent information as described earlier.

Environment

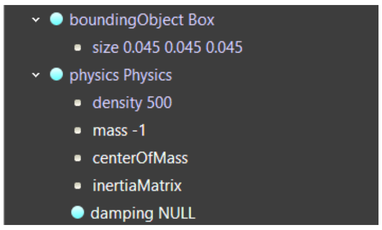
My project utilizes the Cyberbotics Ltd.'s webots software, a free and open-source 3D robot simulator widely employed in industry, education, and research. To simulate my project, I configured a straightforward world environment measuring 5 by 5 meters.

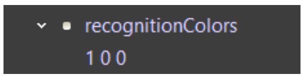


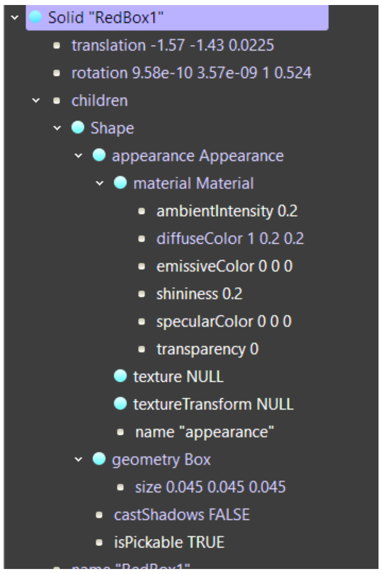
```
⌄ • bodySlot
  ⌄ ● Camera "camera"
      • translation 0.28 0 -0.06
      • rotation 0 1 0 4.69e-06
      • children
      • name "camera"
      • model ""
      • description ""
      • contactMaterial "default"
      • immersionProperties
      ● boundingObject NULL
      ● physics NULL
      • locked FALSE
      • translationStep 0.01
      • rotationStep 0.262
      • radarCrossSection 0
    ⌄ • recognitionColors
        0 0 0.5
        0 0 1
        0.5 0 0
        1 0 0
        0 0.5 0
        0 1 0
```



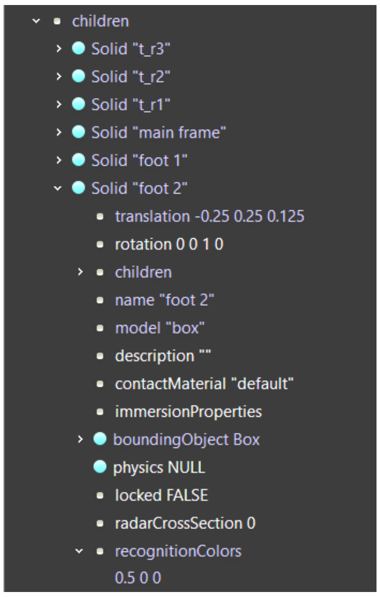
I integrated youBots into the simulated environment and added a basic camera to the robot's 'bodySlot.' The camera, positioned strategically below the KUKA logo, featured color recognition parameters in the 'recognitionColors' section. Red boxes were assigned r=1, g=0, b=0, and red tables were given r=0.5, g=0, b=0. This process extended to green and blue, each with unique color codes. A coding convention of 0.5 represented tables, while a value of 1 indicated objects or boxes.


```
⌄ ● boundingObject Box
    ▪ size 0.045 0.045 0.045
⌄ ● physics Physics
    ▪ density 500
    ▪ mass -1
    ▪ centerOfMass
    ▪ inertiaMatrix
  ● damping NULL
```

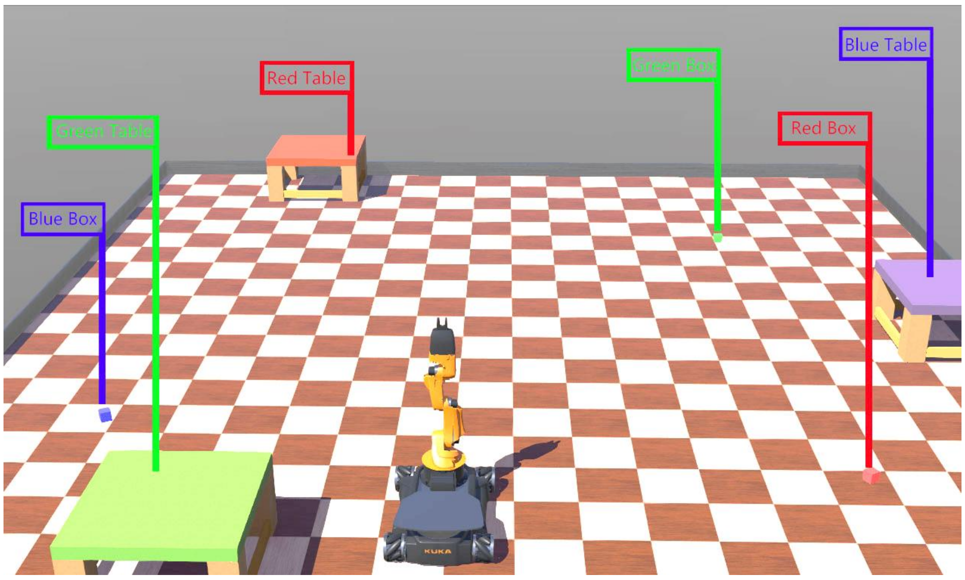

```
⌄ ▪ recognitionColors
    1 0 0
```


```
⌄ ● Solid "RedBox1"
    ▪ translation -1.57 -1.43 0.0225
    ▪ rotation 9.58e-10 3.57e-09 1 0.524
  ⌄ ▪ children
    ⌄ ● Shape
      ⌄ ● appearance Appearance
        ⌄ ● material Material
            ▪ ambientIntensity 0.2
            ▪ diffuseColor 1 0.2 0.2
            ▪ emissiveColor 0 0 0
            ▪ shininess 0.2
            ▪ specularColor 0 0 0
            ▪ transparency 0
          ● texture NULL
          ● textureTransform NULL
          ▪ name "appearance"
      ⌄ ● geometry Box
          ▪ size 0.045 0.045 0.045
        ▪ castShadows FALSE
        ▪ isPickable TRUE
```

To introduce boxes into the simulated environment, I utilized the left panel to add a 'Solid' from 'Base nodes.' Within the 'children' property of this Solid, a 'Shape' was appended, containing a 'Box' with dimensions of (0.045, 0.045, 0.045) meters. The color of the box was defined using the 'Appearance' property with a 'Material' setting of (r=1, g=0.2, b=0.2). Ensuring a realistic appearance required adjustments to the 'boundingObject' property, matching the dimensions of the Solid Box, and setting 'physics' to a 'Physics' configuration with a density of 500 (initially attempted at 1000 but proved too heavy). The 'recognitionColors' for camera detection were established as red (r=1, g=0, b=0), and this box was named RedBox. Replicating this process, two additional boxes were created, colored blue and green, labeled as BlueBox and GreenBox. With these three boxes on the ground, the subsequent step involves introducing three distinct color tables, each corresponding to a specific box.

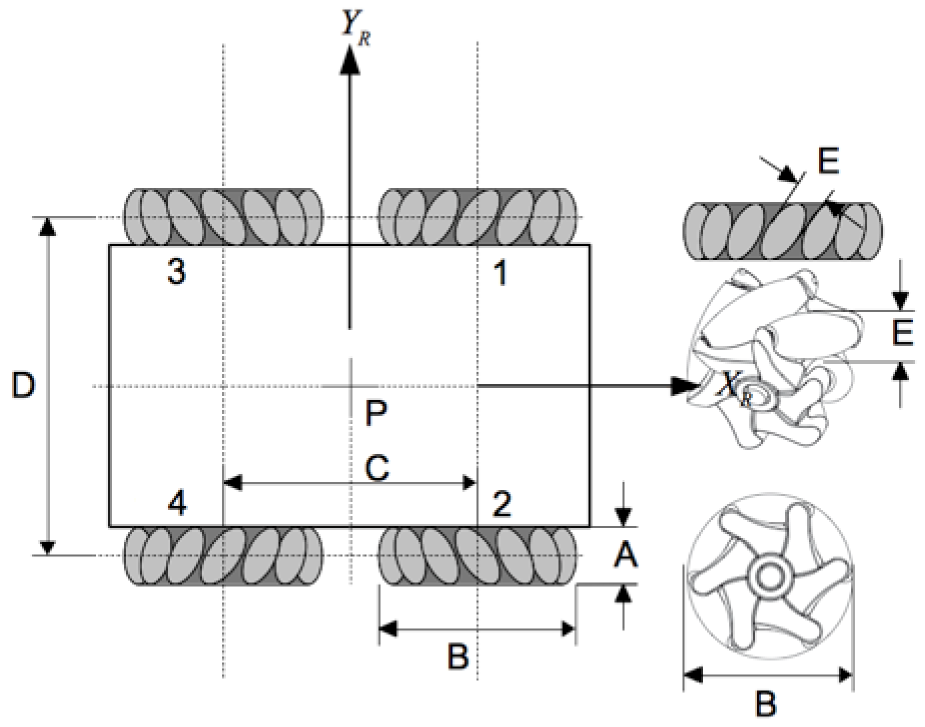

```
⌄ ▪ children
  › ● Solid "t_r3"
  › ● Solid "t_r2"
  › ● Solid "t_r1"
  › ● Solid "main frame"
  › ● Solid "foot 1"
  ⌄ ● Solid "foot 2"
      ▪ translation -0.25 0.25 0.125
      ▪ rotation 0 0 1 0
    › ▪ children
      ▪ name "foot 2"
      ▪ model "box"
      ▪ description ""
      ▪ contactMaterial "default"
      ▪ immersionProperties
    › ● boundingObject Box
      ● physics NULL
      ▪ locked FALSE
      ▪ radarCrossSection 0
    ⌄ ▪ recognitionColors
        0.5 0 0
```

Using PROTO nodes from Webots Projects, I integrated a table under 'objects' and 'tables.' This table has fixed parameters and lacked a recognition color. To rectify this, I converted it to a base node, intending to assign a recognition color to one or two legs. In the 'children' property, within 'Solid' named "foot2," I set 'recognitionColors' to r=0.5, g=0, b=0 (red). I also changed the surface color to red and renamed it "tableRed." This process was repeated for two more tables with recognition colors r=0, g=0.5, b=0 ("tableGreen") and r=0, g=0, b=0.5 ("tableBlue"). Positioned strategically, the environment is now ready for subsequent tasks.



My Created Functions: An In-Depth Exploration

The Dynamics of Wheeled Locomotion

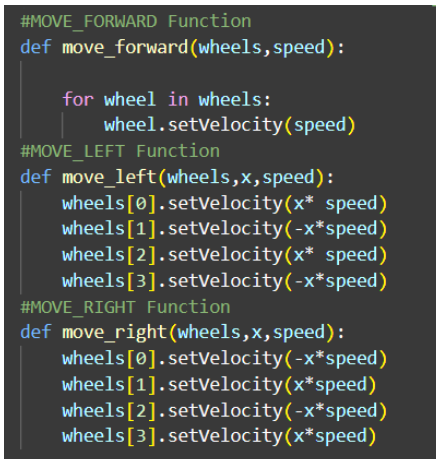


This figure shows the wheels of youBot design (YouBot Detailed Specifications, n.d.)

To maneuver a four-wheeled robot to the left or right, the differential drive method is employed. This technique utilizes two motors positioned on opposite sides of the robot. Steering is achieved by modifying the speed and direction of each motor independently. The term "differential steering" arises from the adjustment of the speed and direction ("difference") between these two wheels. For instance, to shift the robot leftwards, one can increase the speed of the right motor while decreasing the speed of the left motor. Similarly, to navigate the robot to the right, the left motor's speed is increased while the right motor's speed is decreased. (Control of Mobile Robots- 2.2 Differential Drive Robots, n.d.) (MOVING YOUR ROBOT IN LEFT AND RIGHT DIRECTIONS, n.d.) (How Do I Position Mecanum Wheels?, n.d.)

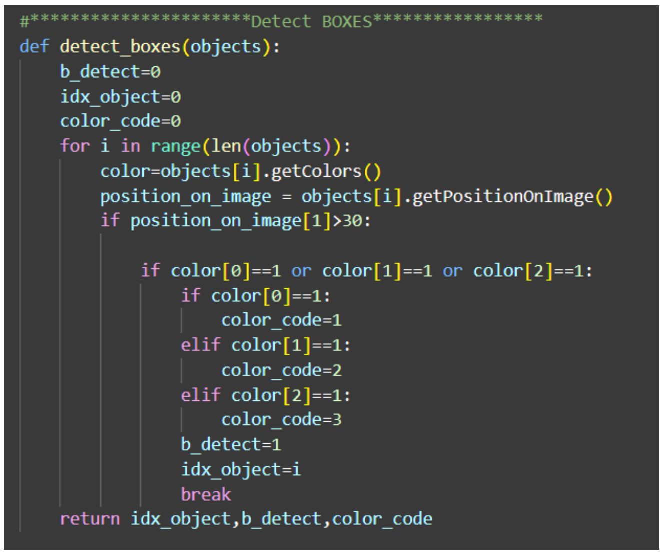
Move Forward Function

The "move_forward" function utilizes two inputs: an array for wheel representation and the wheels' speed. For forward motion, all four wheels must share the same speed. A speed value of 0 halts movement. Lateral movement (left or right) incorporates an additional parameter (x) to regulate each wheel's speed independently. When moving left, wheels 2 and 4 decrease speed (negative x), while wheels 1 and 3 increase speed (positive x). The process reverses for rightward movement.



```
#MOVE_FORWARD Function
def move_forward(wheels,speed):

    for wheel in wheels:
        wheel.setVelocity(speed)
#MOVE_LEFT Function
def move_left(wheels,x,speed):
    wheels[0].setVelocity(x* speed)
    wheels[1].setVelocity(-x*speed)
    wheels[2].setVelocity(x* speed)
    wheels[3].setVelocity(-x*speed)
#MOVE_RIGHT Function
def move_right(wheels,x,speed):
    wheels[0].setVelocity(-x*speed)
    wheels[1].setVelocity(x*speed)
    wheels[2].setVelocity(-x*speed)
    wheels[3].setVelocity(x*speed)
```

Detect Cubes Function

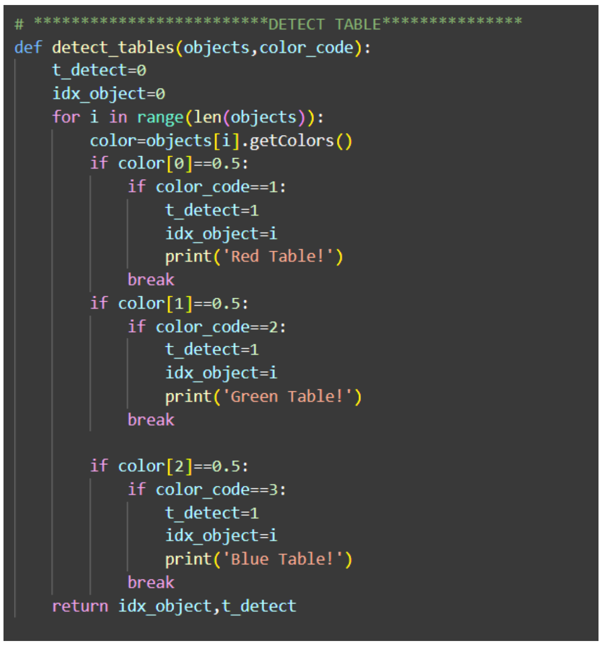


```python
#*********************Detect BOXES****************
def detect_boxes(objects):
    b_detect=0
    idx_object=0
    color_code=0
    for i in range(len(objects)):
        color=objects[i].getColors()
        position_on_image = objects[i].getPositionOnImage()
        if position_on_image[1]>30:

            if color[0]==1 or color[1]==1 or color[2]==1:
                if color[0]==1:
                    color_code=1
                elif color[1]==1:
                    color_code=2
                elif color[2]==1:
                    color_code=3
                b_detect=1
                idx_object=i
                break
    return idx_object,b_detect,color_code
```

The "detect_boxes" function retrieves information about detected objects labeled as 'objects' from the "getRecognitionObjects()" method, capturing data from the camera. It outputs three variables: the object index ("idx_object"), a binary flag indicating box detection ("b_detected"), and the box color code ("color_code"). The function first examines the object's position in the image, considering values > 30 as indicative of being on the ground. The subsequent check involves obtaining RGB color values using the "getColors()" function. A color[0] value of 1 signifies red, designating a box (color recognition set to 1). The function then assigns color_code values: 1 for red, 2 for green, and 3 for blue. Finally, "b_detected" is set to 1, confirming successful box detection with a specific color code.

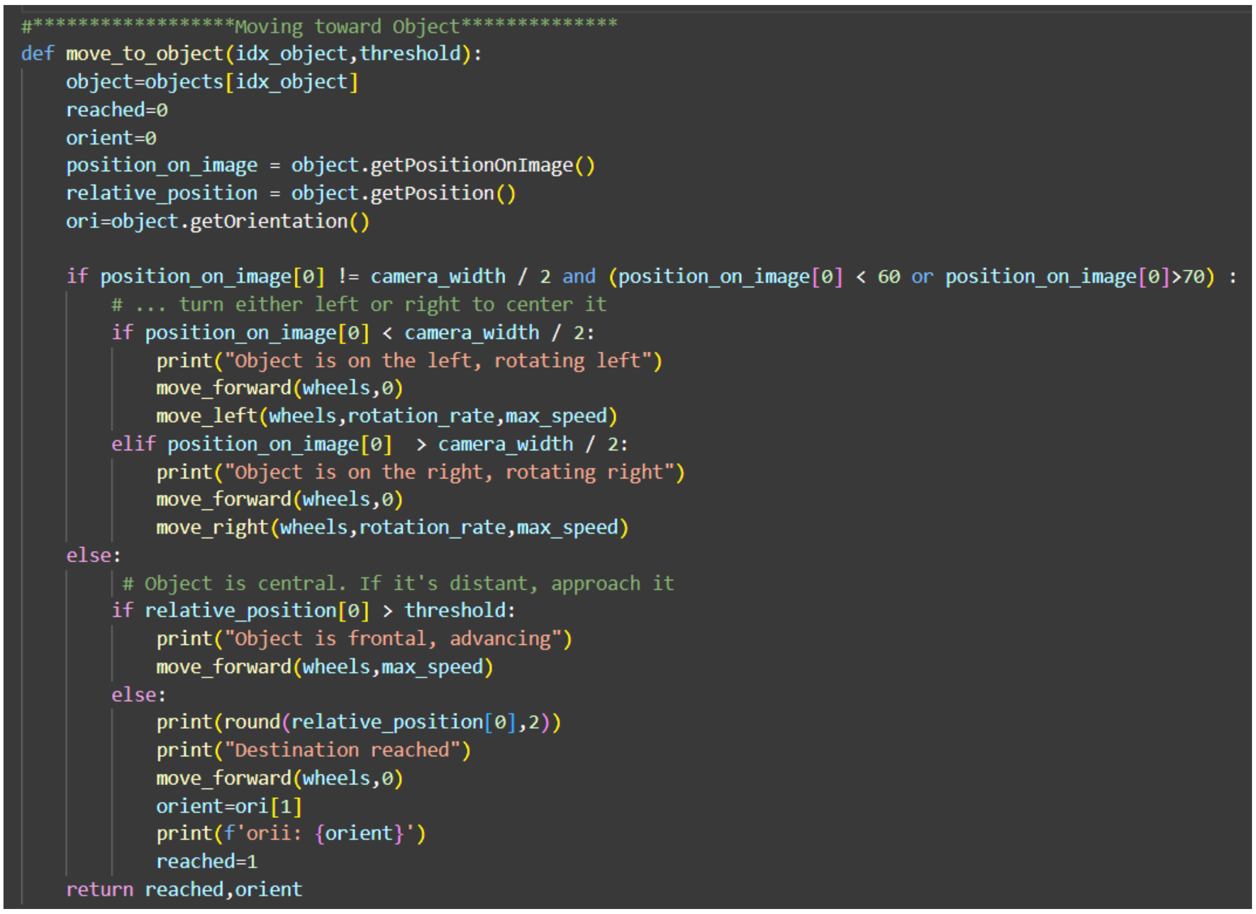
Detect Tables Function

I employed a similar strategy for distinguishing tables. This function relies on the "color_code" obtained from the "detect_boxes" function to identify the corresponding table by comparing the colors of the detected objects. Utilizing the "getColors()" method, I acquired the color for each detected object and examined whether it equaled 0.5, indicating a table. The function returns the index of the table and sets the "t_detect" flag to one, indicating successful table detection.



```python
# ************************DETECT TABLE***************
def detect_tables(objects,color_code):
    t_detect=0
    idx_object=0
    for i in range(len(objects)):
        color=objects[i].getColors()
        if color[0]==0.5:
            if color_code==1:
                t_detect=1
                idx_object=i
                print('Red Table!')
            break
        if color[1]==0.5:
            if color_code==2:
                t_detect=1
                idx_object=i
                print('Green Table!')
            break

        if color[2]==0.5:
            if color_code==3:
                t_detect=1
                idx_object=i
                print('Blue Table!')
            break
    return idx_object,t_detect
```

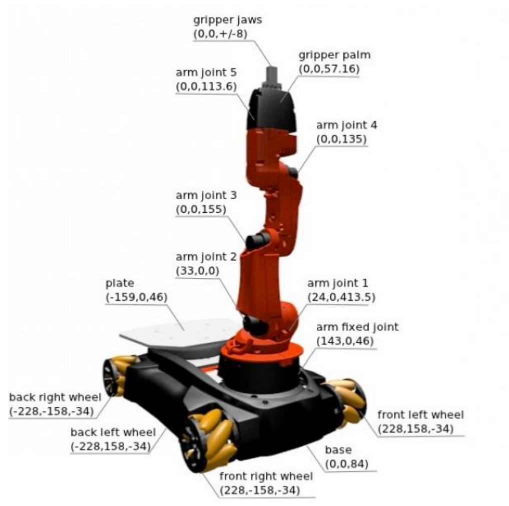
Moving Toward Objects Function

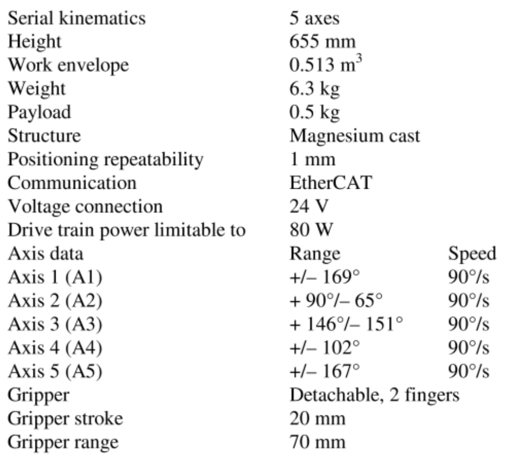


```
#*****************Moving toward Object**************
def move_to_object(idx_object,threshold):
    object=objects[idx_object]
    reached=0
    orient=0
    position_on_image = object.getPositionOnImage()
    relative_position = object.getPosition()
    ori=object.getOrientation()

    if position_on_image[0] != camera_width / 2 and (position_on_image[0] < 60 or position_on_image[0]>70) :
        # ... turn either left or right to center it
        if position_on_image[0] < camera_width / 2:
            print("Object is on the left, rotating left")
            move_forward(wheels,0)
            move_left(wheels,rotation_rate,max_speed)
        elif position_on_image[0]  > camera_width / 2:
            print("Object is on the right, rotating right")
            move_forward(wheels,0)
            move_right(wheels,rotation_rate,max_speed)
    else:
        # Object is central. If it's distant, approach it
        if relative_position[0] > threshold:
            print("Object is frontal, advancing")
            move_forward(wheels,max_speed)
        else:
            print(round(relative_position[0],2))
            print("Destination reached")
            move_forward(wheels,0)
            orient=ori[1]
            print(f'orii: {orient}')
            reached=1
    return reached,orient
```

"move_to_object" uses the index from "detect_boxes" or "detect_tables," plus a position threshold, giving "reached" for object proximity and "orient" for its orientation. The function assesses the object's camera position using a width of 128. Utilizing the "getRecognitionObjects()" function, it obtains height and width values. Dividing the screen at midpoint 64, the function categorizes the object as left (position below 64), right (position beyond 64), or centered (position at 64). The robot navigates based on object position: left or right turns using "move_left()" or "move_right()" if the object is off-center. Forward movement occurs if the object is in front. Assessing x, y, and z distances, the function halts when values are below a set threshold, marking "reached" as 1. Simultaneously, it captures the object's y-axis orientation via the "getOrientation()" function, crucial for aligning the robot's arms during pickup.

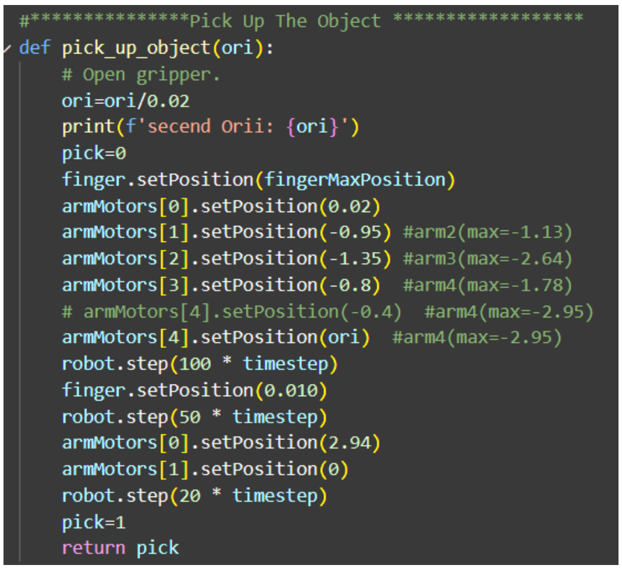
Pick up cubes Function





Serial kinematics	5 axes	
Height	655 mm	
Work envelope	0.513 m^3	
Weight	6.3 kg	
Payload	0.5 kg	
Structure	Magnesium cast	
Positioning repeatability	1 mm	
Communication	EtherCAT	
Voltage connection	24 V	
Drive train power limitable to	80 W	
Axis data	Range	Speed
Axis 1 (A1)	+/– 169°	90°/s
Axis 2 (A2)	+ 90°/– 65°	90°/s
Axis 3 (A3)	+ 146°/– 151°	90°/s
Axis 4 (A4)	+/– 102°	90°/s
Axis 5 (A5)	+/– 167°	90°/s
Gripper	Detachable, 2 fingers	
Gripper stroke	20 mm	
Gripper range	70 mm	

This figure shows the details of wheels, arms, and joint of youBot, and the table is included extra information about them. (KUKA's youBot, n.d.) (KUKA YouBot Kinematics, Dynamics And 3D Model, n.d.)

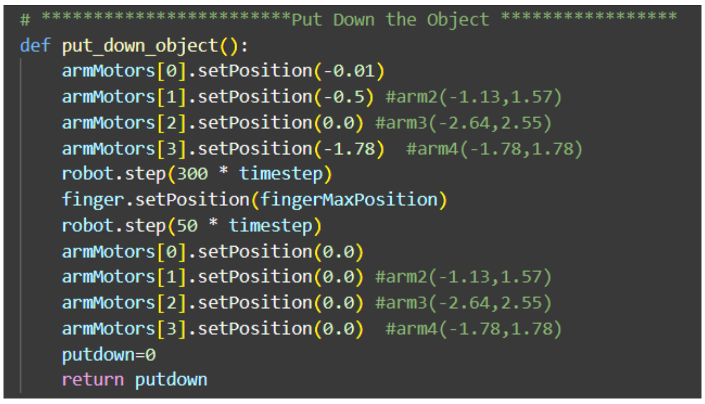


```
#***************Pick Up The Object ******************
def pick_up_object(ori):
    # Open gripper.
    ori=ori/0.02
    print(f'secend Orii: {ori}')
    pick=0
    finger.setPosition(fingerMaxPosition)
    armMotors[0].setPosition(0.02)
    armMotors[1].setPosition(-0.95) #arm2(max=-1.13)
    armMotors[2].setPosition(-1.35) #arm3(max=-2.64)
    armMotors[3].setPosition(-0.8)  #arm4(max=-1.78)
    # armMotors[4].setPosition(-0.4)  #arm4(max=-2.95)
    armMotors[4].setPosition(ori)  #arm4(max=-2.95)
    robot.step(100 * timestep)
    finger.setPosition(0.010)
    robot.step(50 * timestep)
    armMotors[0].setPosition(2.94)
    armMotors[1].setPosition(0)
    robot.step(20 * timestep)
    pick=1
    return pick
```

The "pick_up_object" function uses the orientation from "move_to_object" to rotate the arms' 5th joint. A determined value, 0.02, is applied based on environmental assessment. Adjustments to the finger and joint positions ensure precise approach to the box. The finger position is set at 0.010 for secure grasping. A brief interval using "robot.step" allows the robot to manage its arms. The function outputs a flag, "pick," indicating successful box pickup.

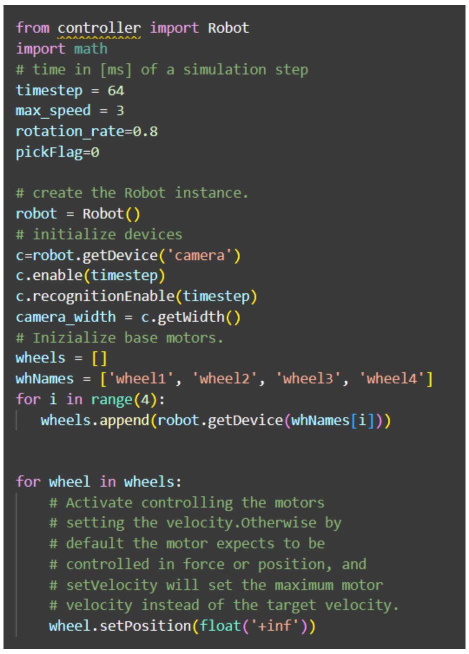
Put down cubes Function

After reaching the table, the robot uses the "put_down_object" function to adjust arm and finger positions for placing the box. Following this, arms reset to default positions, and the function outputs "putdown=0," confirming successful box placement on the table. The robot then resumes searching for another ground box, repeating the sequence.

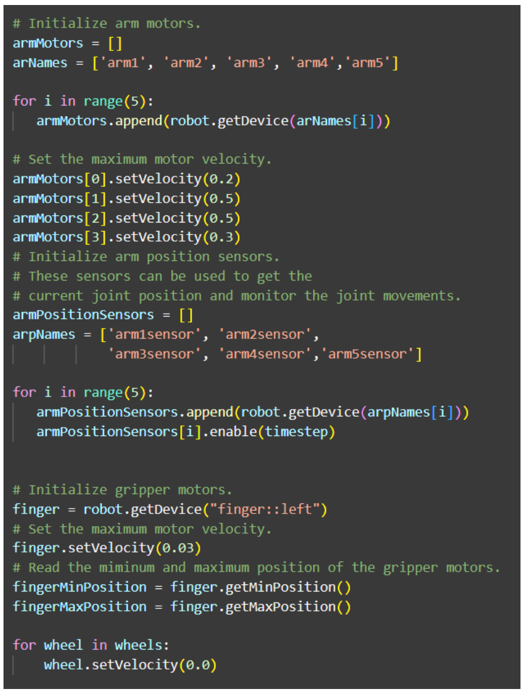


```
# ***********************Put Down the Object *****************
def put_down_object():
    armMotors[0].setPosition(-0.01)
    armMotors[1].setPosition(-0.5) #arm2(-1.13,1.57)
    armMotors[2].setPosition(0.0) #arm3(-2.64,2.55)
    armMotors[3].setPosition(-1.78)  #arm4(-1.78,1.78)
    robot.step(300 * timestep)
    finger.setPosition(fingerMaxPosition)
    robot.step(50 * timestep)
    armMotors[0].setPosition(0.0)
    armMotors[1].setPosition(0.0) #arm2(-1.13,1.57)
    armMotors[2].setPosition(0.0) #arm3(-2.64,2.55)
    armMotors[3].setPosition(0.0)  #arm4(-1.78,1.78)
    putdown=0
    return putdown
```

Initializing



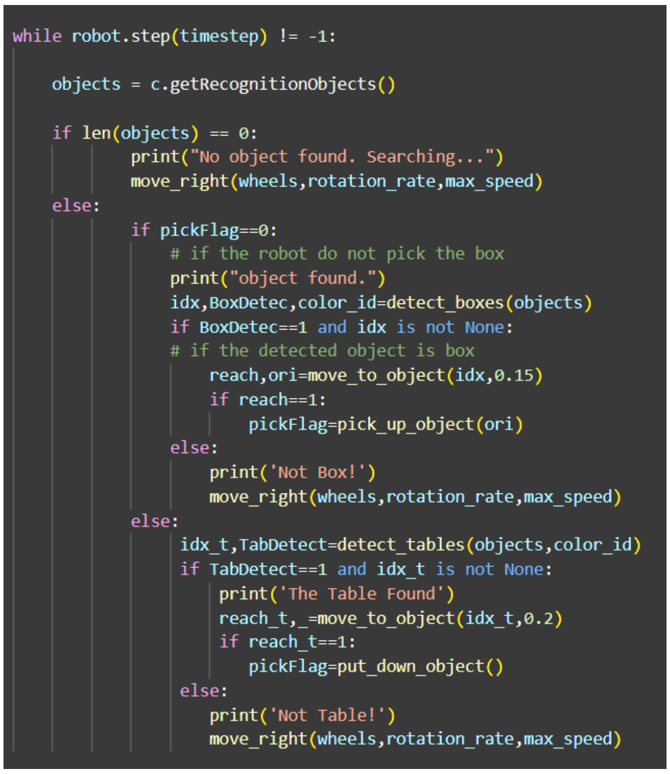
```
from controller import Robot
import math
# time in [ms] of a simulation step
timestep = 64
max_speed = 3
rotation_rate=0.8
pickFlag=0

# create the Robot instance.
robot = Robot()
# initialize devices
c=robot.getDevice('camera')
c.enable(timestep)
c.recognitionEnable(timestep)
camera_width = c.getWidth()
# Inizialize base motors.
wheels = []
whNames = ['wheel1', 'wheel2', 'wheel3', 'wheel4']
for i in range(4):
    wheels.append(robot.getDevice(whNames[i]))


for wheel in wheels:
    # Activate controlling the motors
    # setting the velocity.Otherwise by
    # default the motor expects to be
    # controlled in force or position, and
    # setVelocity will set the maximum motor
    # velocity instead of the target velocity.
    wheel.setPosition(float('+inf'))
```



```
# Initialize arm motors.
armMotors = []
arNames = ['arm1', 'arm2', 'arm3', 'arm4','arm5']

for i in range(5):
    armMotors.append(robot.getDevice(arNames[i]))

# Set the maximum motor velocity.
armMotors[0].setVelocity(0.2)
armMotors[1].setVelocity(0.5)
armMotors[2].setVelocity(0.5)
armMotors[3].setVelocity(0.3)
# Initialize arm position sensors.
# These sensors can be used to get the
# current joint position and monitor the joint movements.
armPositionSensors = []
arpNames = ['arm1sensor', 'arm2sensor',
            'arm3sensor', 'arm4sensor','arm5sensor']

for i in range(5):
    armPositionSensors.append(robot.getDevice(arpNames[i]))
    armPositionSensors[i].enable(timestep)


# Initialize gripper motors.
finger = robot.getDevice("finger::left")
# Set the maximum motor velocity.
finger.setVelocity(0.03)
# Read the miminum and maximum position of the gripper motors.
fingerMinPosition = finger.getMinPosition()
fingerMaxPosition = finger.getMaxPosition()

for wheel in wheels:
    wheel.setVelocity(0.0)
```

I initialized key variables, such as timestep for simulation and max_speed for the robot's default speed, along with rotation_rate for turns and pickFlag to track cube pickup. After creating a robot instance, I focused on camera initialization and activated devices, including wheels, arms, and fingers. Motor control activation involved configuring velocity to ensure proper functioning, as default settings assumed force or position control. To establish target velocity, setVelocity was used. I also set gripper motor positions with getMinPosition and getMaxPosition functions.

Integration of Components



```
while robot.step(timestep) != -1:

    objects = c.getRecognitionObjects()

    if len(objects) == 0:
            print("No object found. Searching...")
            move_right(wheels,rotation_rate,max_speed)
    else:
            if pickFlag==0:
                # if the robot do not pick the box
                print("object found.")
                idx,BoxDetec,color_id=detect_boxes(objects)
                if BoxDetec==1 and idx is not None:
                # if the detected object is box
                    reach,ori=move_to_object(idx,0.15)
                    if reach==1:
                        pickFlag=pick_up_object(ori)
                else:
                    print('Not Box!')
                    move_right(wheels,rotation_rate,max_speed)
            else:
                 idx_t,TabDetect=detect_tables(objects,color_id)
                 if TabDetect==1 and idx_t is not None:
                     print('The Table Found')
                     reach_t,_=move_to_object(idx_t,0.2)
                     if reach_t==1:
                        pickFlag=put_down_object()
                 else:
                    print('Not Table!')
                    move_right(wheels,rotation_rate,max_speed)
```

Within the "robot.step" loop, I implemented the logic. Initially, I obtained an array of detected objects using "getRecognitionObjects." Then, I assessed the array length. If it was zero, signifying no detected objects, the robot turned right to recognize objects. If non-zero, the logic continued to check if a cube was picked up, determined by the "pickFlag," initially set to zero.

Next, the "detect_boxes" function yielded the box's index, detection flag, and color information. Using "move_to_object," the robot approached the box until a set distance. Subsequently, "pick_up_object" lifted the cube, changing "pickFlag" to 1. This concluded the loop, restarting the task with the while loop.

In this iteration, guided by the "pickFlag," the logic shifted to finding a matching table using the "detect_table" function. The robot approached the identified table with the "move_to_object" function, maintaining a specific distance. Upon reaching the table, a flag was returned. Subsequently, the robot utilized the "put_down_object" function to place the cube on the correct table, resetting the "pickFlag" to zero. This indicated the need to repeat the process for another cube in the environment.

My Challenges and Solutions

1. The camera identifies objects and approaches the first one detected, discerning cubes (assigned recognition color 1) from tables (assigned recognition color 0.5).
2. The "detect_boxes" function unintentionally recognizes boxes on tables due to varying heights. I implemented a check using "position_on_image[1]": around 32 indicates the box is on the ground, and around 10 suggests it's on a table.
3. In the "move_to_object" function, when the object was near the center of the camera, the robot became stuck, frequently turning right and left without forward movement. This issue was identified when "position_on_image[0]" was around the center position of the camera (a number between 60 and 70). To address this, I added a condition: if it is greater than 60 and smaller than 70, indicating the object is in the center of the screen, the robot moves forward.
4. For picking up boxes, the gripper encountered difficulty with rotated boxes. To resolve this, I utilized the "getOrientation()" function in the recognition camera within the "move_to_object" function. The obtained orientation parameter was then passed through the "pick_up_object" function, facilitating the rotation of joint 5 in the robot's arm to match the box's rotation.
5. When the density of boxes was set to 1000, the robot struggled to pick them up due to excessive weight for the arms. To mitigate this, I adjusted the density to 500, resulting in successful operations.

Real-world Applications

Using a KUKA youBot for color recognition, object manipulation, and sorting tasks is a practical application with potential real-world implications in various industries. Here are some ways the robot could be utilized and the potential interdisciplinary support it might require:

Manufacturing and Logistics:

- Automated Sorting: The youBot can be employed in warehouses to sort and organize products based on their color or other visual characteristics.
- Assembly Line Assistance: Integration of the robot in manufacturing processes to pick and place components based on color coding.

Retail:

- Inventory Management: The robot can be used for managing inventory by identifying and organizing products based on their color, making restocking more efficient.

Healthcare:

- Lab Automation: In a laboratory setting, the robot can assist in sorting and organizing samples or lab equipment based on color codes.

- Medical Supply Management: Sorting and organizing medical supplies in hospitals or pharmacies based on color can enhance efficiency.

Agriculture:

- Harvesting: The robot can be adapted for precision harvesting by recognizing the color of ripe fruits or vegetables.

Home Assistance:

- Domestic Chores: Sorting and organizing household items, such as laundry or toys, based on color preferences.

Waste Management:

- Recycling: The robot could be used in recycling facilities to sort and separate recyclable materials based on color.

While possessing numerous potential capabilities in practical applications, this robot encounters challenges when operating in desert environments or traversing jagged and rough surfaces. Specifically, its Mecanum wheels exhibit imperfect performance in such conditions, leading to limitations in maneuverability. Due to its inherent design characteristics, the robot may struggle to navigate effectively and, in certain situations, may cease to operate altogether.

Collaboration with Other Sciences

- *Computer Vision:* Enhancements in color recognition algorithms through advancements in computer vision would improve the robot's ability to accurately identify and manipulate objects.
- *Materials Science:* Understanding the properties of different materials and their response to color recognition technologies can influence the design of objects for better detection.
- *Human-Robot Interaction:* Incorporating insights from psychology and human behavior can help design robots that are intuitive and user-friendly for humans to work alongside.
- *Mechanical Engineering:* Continuous improvements in the robot's mechanical design and control systems would enhance its reliability and precision.

Ethical and Social Implications

- *Job Displacement:* The introduction of such robots might lead to job displacement in certain sectors, necessitating ethical considerations and potential strategies for workforce transition.
- *Privacy:* As robots become more integrated into daily life, there may be concerns about privacy and data security, requiring collaboration with experts in cybersecurity and privacy.

Conclusion

In conclusion, my project centers on the utilization of the KUKA youBot and a camera for environmental recognition, aimed at locating cubes on the ground, discerning their colors to identify corresponding tables, and systematically placing the cubes on the appropriate tables. The iterative process involves repeating this cycle until the environment is cleared of cubes initially situated on the ground. Despite encountering challenges such as distinguishing cubes from tables, handling dense cubes, and navigating towards objects, I successfully addressed these issues through the development of specific solutions.

Furthermore, the application of a robot like the KUKA youBot for color recognition and sorting extends beyond my project, presenting diverse opportunities for various industries. To fully harness the potential of such robots in real-world scenarios, collaboration with experts from disciplines such as computer science, materials science, human factors, and ethics becomes imperative. This collaborative approach is vital for addressing the multifaceted challenges and opportunities associated with the deployment of these robots in practical settings.

References

Adamov, B. (2018). Influence of mecanum wheels construction on accuracy of the omnidirectional platform navigation (on exanple of KUKA youBot robot). *IEEE*, 1-4.

Control of Mobile Robots- 2.2 Differential Drive Robots. (n.d.). Retrieved from YouTube: https://www.youtube.com/watch?v=aE7RQNhwnPQ

How Do I Position Mecanum Wheels? (n.d.). Retrieved from YouTube: https://www.youtube.com/watch?v=ArQl1S4aGCg

KUKA YouBot Kinematics, Dynamics And 3D Model. (n.d.). Retrieved from KUKA youBot developrts: http://www.youbot-store.com/developers/kuka-youbot-kinematics-dynamics-and-3d-model

KUKA's youBot. (n.d.). Retrieved from Cyberbotics: https://www.cyberbotics.com/doc/guide/youbot?version=cyberbotics:R2019a-rev1

Kumar, Rahul and Lal, Sunil and Kumar, Sanjesh and Chand, Praneel. (2014). Object detection and recognition for a pick and place robot. *IEEE*, 1-7.

MOVING YOUR ROBOT IN LEFT AND RIGHT DIRECTIONS. (n.d.). Retrieved from YouTube: https://www.youtube.com/watch?v=q8W9jWIWDjc

Müggler, E., Fässler, M., Scaramuzza, D., Huck, S., Lygeros, J. (2013). Torque control of a kuka youbot arm.

Webots Reference Manual. (n.d.). Retrieved from Cyberbotics: https://cyberbotics.com/doc/reference/camera

YouBot Detailed Specifications. (n.d.). Retrieved from youbot-store: http://www.youbot-store.com/wiki/index.php/YouBot_Detailed_Specifications

Zhang, Liandong and Zhou, Changjiu. (2023). Kuka youBot arm shortest path planning based on geodesics. *IEEE*, 2317--2321.