Nikhil Simha, Vyom Thakkar, Neeloy Chakraborty
simha3, vnt2, neeloyc2
Cleaner Pastures
https://github.com/vyomthakkar/ece470finalproject
https://www.youtube.com/watch?v=pLc1tVorXGs

Final Project Report - Team Cleaner Pastures

Abstract

Team Cleaner Pastures is a project focused on implementing simple object sorting within the CoppeliaSim robotics simulation platform. The importance of this is to provide a proof of concept for a potential trash sorting robot, with every piece of the setup except the sorting algorithm itself simulated accurately. In our project we sorted between red and green cubes, but in the real world this would simply be replaced with trash and recyclable objects - which the robot could be trained on with neural networks. Our robot's main solution is to use a proximity sensor to let the conveyor belt setup know when to stop, at which point the robot picks up a block and uses a vision sensor to move the block to the desired location. To measure our success, we let our simulation run several times and noticed a 90.16% success rate. Qualitatively, we felt pretty satisfied with our results, because we were able to sort the overwhelming majority of blocks. Overall, this project was a great exercise in understanding what work goes into creating a robotics system besides just programming. In the real world, sensors may malfunction and other unexpected things may happen, so those are areas to explore in the future.

Intro

The objective of our project is to make use of the UR3 robot arm, a gripper, and two sensors in order to make a sorting robot. We pitched the robot in our final project video as a consumer product, but a more realistic use case would perhaps be to sort objects that get taken to garbage dumps, as often recyclable objects are mistakenly thrown away. Coming into this project, our team was familiar with python but thoroughly unfamiliar with robotics simulation or CoppeliaSim. In order to solve our task, we did research on how to connect the python remote API [3, 5, 6] as well as on LUA scripting [1]. We also needed to use our knowledge from lectures to implement a functional forward and inverse kinematics system for our robot arm. For equations, we consulted the Modern Robotics Textbook [2]. Finally, for vision sensing, we consulted Coppelia's website [4].

Nikhil Simha, Vyom Thakkar, Neeloy Chakraborty
simha3, vnt2, neeloyc2
Cleaner Pastures
https://github.com/vyomthakkar/ece470finalproject
https://www.youtube.com/watch?v=pLc1tVorXGs

Method

Design Process

Our design process consisted of the following steps:

1. Work with the UR3 robot in CoppeliaSim (previously VREP)
2. Move different colored blocks towards a UR3 on a conveyor belt
3. Stop the conveyor belt once a block passes by proximity sensor
4. Move the robot end effector to the block using forward kinematics
5. Decide which location to place the block, depending on its color
6. Place the block in that location using inverse kinematics

Figure 1 is a block diagram consisting of the specific modules we will be describing in more detail in the future sections of this paper. Also included is Figure 2, which is a screenshot of the scene, with color coding to match the block diagram included.

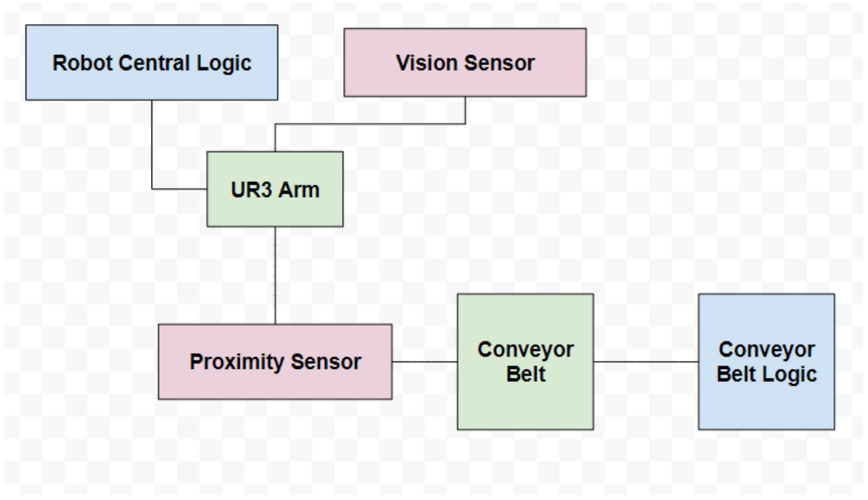
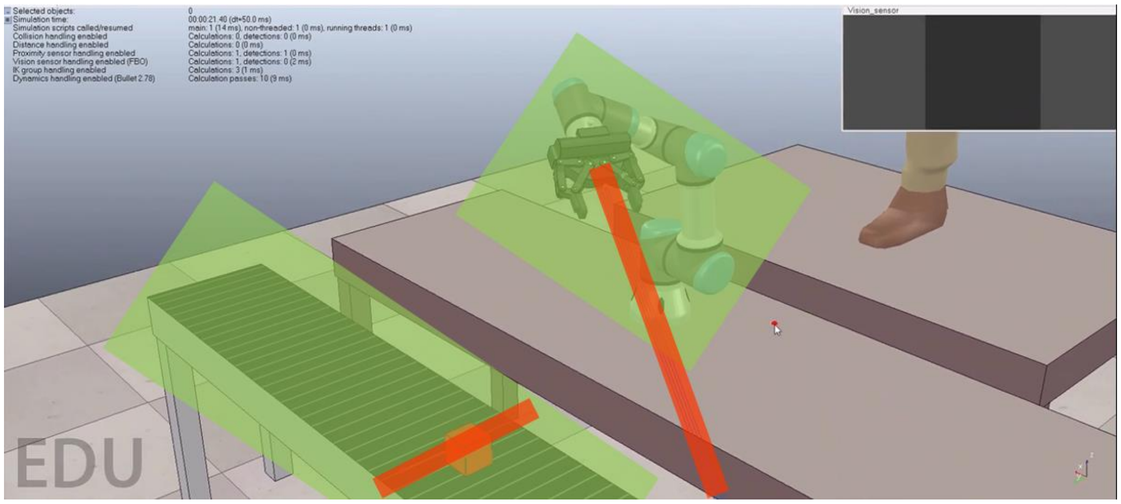


Figure 1. Block diagram of modules in project

Nikhil Simha, Vyom Thakkar, Neeloy Chakraborty
simha3, vnt2, neeloyc2
Cleaner Pastures
https://github.com/vyomthakkar/ece470finalproject
https://www.youtube.com/watch?v=pLc1tVorXGs



```
Selected objects:                     0
Simulation time:                      00:00:21.40 (dt=50.0 ms)
Simulation scripts called/resumed     main: 1 (14 ms), non-threaded: 1 (0 ms), running threads: 1 (0 ms)
Collision handling enabled            Calculations: 0, detections: 0 (0 ms)
Distance handling enabled             Calculations: 0 (0 ms)
Proximity sensor handling enabled     Calculations: 1, detections: 1 (0 ms)
Vision sensor handling enabled (FBO)  Calculations: 1, detections: 0 (2 ms)
IK group handling enabled             Calculations: 3 (1 ms)
Dynamics handling enabled (Bullet 2.78) Calculation passes: 10 (9 ms)
```

Figure 2. Color-coded screenshot of simulation corresponding to Figure 1 module blocks

Modules

In Figure 1, the modules shown in green correspond to equipment and/or robots, the red modules correspond to sensors, and the blue modules correspond to code.

Overall, we can see that the robot central logic is responsible for forward & inverse kinematics as well as decision making for where to move the arm at any time step, and the other modules support the goal of our project to sort blocks. For example, the conveyor belt logic generates blocks, and will stop the conveyor belt once a block passes through the proximity sensor, signaling to the UR3 that a block is ready to be picked up and sorted.

For the second checkpoint, we used the equations discussed in lecture to get the UR3 arm to move to specific configurations via forward kinematics. The only problem with this was that the arm moved far too stiffly.

This was resolved by checkpoint 3, in which we implemented inverse kinematics. The third checkpoint onwards also marked the beginning of giving our robot a decision-making module. This module connected together the vision, proximity, and central logic modules in order to actively allow the robot to compute where and when it should move, and if its gripper should be on.

Existing Code

The main existing code we built off of was from Mechatronics Ninja's Matlab **movel** & **gripper** functions, and converting it to Python with additions we needed [1]. The **gripper** function was used to visually "fake" the grabbing of a block from the conveyor belt. The way this is done is by keeping track of the current gripping state of the end effector, and changing the target velocity of the gripper object in CoppeliaSim to either open or close the gripper as needed. This function is called in the main driver code when the robot needs to try and pick up a block. The **movel** function is elaborated from Mechatronics Ninja to move a dummy target in our scene, which in turn moves the end effector of the robot with inverse kinematics. This topic is further discussed at the end of the **Inverse Kinematics** section of this paper.

Python Remote API

The first checkpoint of our project was dedicated to figuring out how to program a UR3 robot to move its joints to specific angles from the Python API. This was one of our most difficult tasks as we found there were a lack of tutorials dedicated to the setup, but once we figured this out, it provided a solid foundation for understanding the relationship between Python, and the scene running in CoppeliaSim.

Different scripts can be running within the simulation with code written in the regular API [5]. Similarly, majority of what we can do with the regular API in LUA, can be performed in a remote Python connection to the simulation with its own functions [7].

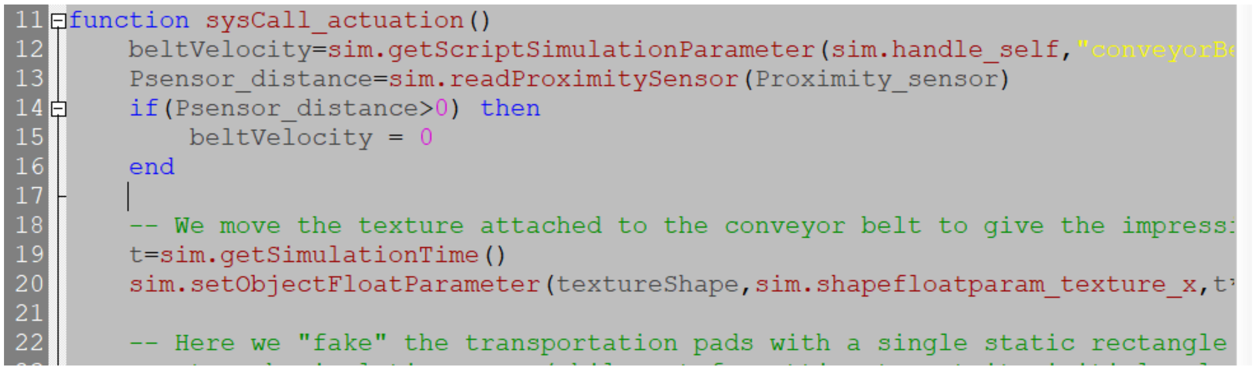
Nikhil Simha, Vyom Thakkar, Neeloy Chakraborty
simha3, vnt2, neeloyc2
Cleaner Pastures
https://github.com/vyomthakkar/ece470finalproject
https://www.youtube.com/watch?v=pLc1tVorXGs

What we discovered was that each object in the running scene has a dedicated “handle,” which can be referenced to perform different actions on that object. So, in a Python program, we first begin a remote session from the CoppeliaSim scene by running “**simExtRemoteApiStart (19999)**” in the LUA terminal, and connect to it from Python by using the Python API’s “**simxStart**” function.

We can then, for example, get the handles of each of the UR3 joints using the “**simxGetObjectHandle**” function, and use the “**simxSetObjectPosition**” function to move the object to a certain position in the world frame. With this basic understanding of how to interact with CoppeliaSim, we were able to move to implement scripts in LUA or Python as we saw fit.

Conveyor belt and Proximity Sensor

The conveyor belt is a built-in object in CoppeliaSim. We made use of it to transport blocks in our simulation. In the code below in Figure 3, lines 13 through 16 are written by us, and add support for the proximity sensor. When a block reaches the proximity sensor’s line of sight, the conveyor belt receives the signal and sets its velocity to 0 until the block is removed and nothing is obstructing its path. The signal from the proximity sensor is also used in the robot’s central logic in Python to know when it should pick a block up.



```
11 function sysCall_actuation()
12     beltVelocity=sim.getScriptSimulationParameter(sim.handle_self,"conveyorB
13     Psensor_distance=sim.readProximitySensor(Proximity_sensor)
14     if(Psensor_distance>0) then
15         beltVelocity = 0
16     end
17     
18     -- We move the texture attached to the conveyor belt to give the impress
19     t=sim.getSimulationTime()
20     sim.setObjectFloatParameter(textureShape,sim.shapefloatparam_texture_x,t
21 
22     -- Here we "fake" the transportation pads with a single static rectangle
```

Figure 3. LUA script for controlling the speed of the conveyor belt

Nikhil Simha, Vyom Thakkar, Neeloy Chakraborty
simha3, vnt2, neeloyc2
Cleaner Pastures
https://github.com/vyomthakkar/ece470finalproject
https://www.youtube.com/watch?v=pLc1tVorXGs

Vision sensor

We used an orthographic vision sensor for our project. This vision sensor was mounted on our UR3 arm as shown in Figure 4:

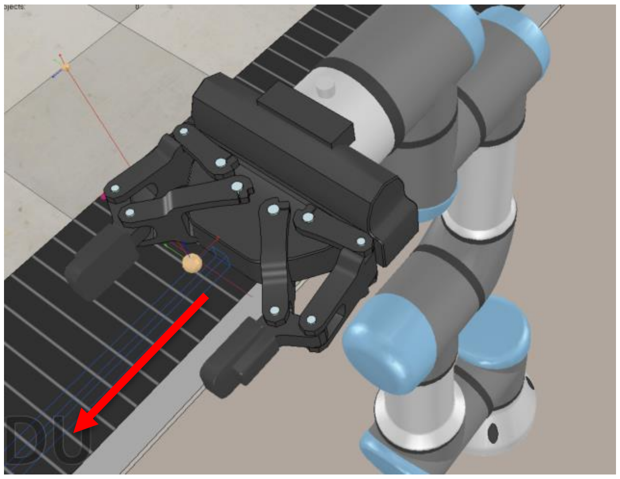


Figure 4. Screenshot of placement of vision sensor, and its field of view

The blue rectangular area facing the same direction as the red arrow in the above picture represents the field of view of the vision sensor. The usage of the vision is explained further in the **Decision Making** section of this paper.

Forward Kinematics

In checkpoint 2, to implement forward kinematics for our project, we prompted the user to provide six values of joint angles (theta) corresponding to the six revolute joints of the UR3 robot in our simulation via the Spyder IDE. We used the user input to move the UR3 robot in the specified joint configuration and we also compute the pose of the end-effector. In order to obtain user input and interact with the UR3 robot simulation we made use of the python CoppeliaSim remote API.

We used the following equation to compute the pose of the end effector $T(\theta)$, where θ corresponds to the vector of joint angles, S_i corresponds to the spatial screw of joint i and M is the homogeneous transformation matrix of the end effector relative to the robot base frame:

$$T(\theta) = e^{[\mathcal{S}_1]\theta_1} \cdots e^{[\mathcal{S}_{n-1}]\theta_{n-1}} e^{[\mathcal{S}_n]\theta_n} M.$$

Figure 5. Transformation matrix of end effector relative to base frame given screw axes of joints, and pose of base

The first step in implementing the forward kinematics was to obtain the homogeneous transformation matrix (M) in order to represent the end-effector frame relative to the base frame in the zero position of the UR3. To obtain this information we made use of a CoppeliaSim remote API function called **simxGetJointMatrix** [3], which returns the homogeneous transformation matrix of a specified joint (which, in our case was the last joint of our UR3). We then computed the six spatial screws of the six revolute joints by inspecting the zero position of the UR3 robot arm in the simulation.

In order to compute the linear velocity component of the spatial screws, we needed a point along the axis of rotation of each joint, and in order to obtain this information, we made use of a helper function **get_joint**. The **get_joint** function returns the center position of each joint with reference to the robot base frame. Next, in order to compute the skew symmetric form of the spatial screws and compute the product of exponentials, we recycled code that we had implemented for HW6.

Finally, in order to move the UR3 robot arm to the joint configuration provided by the user, we make use of the **SetJointPosition** function, a helper function provided by the course staff. This helper function takes as input, a six-dimensional vector of joint angles and internally makes use of the CoppeliaSim remote API function **simxSetJointTargetPosition** in order to set each joint of the UR3 to its corresponding user-provided joint angle.

Nikhil Simha, Vyom Thakkar, Neeloy Chakraborty
simha3, vnt2, neeloyc2
Cleaner Pastures
https://github.com/vyomthakkar/ece470finalproject
https://www.youtube.com/watch?v=pLc1tVorXGs

Inverse Kinematics

Initially in checkpoint 3, to implement inverse kinematics for our project, we prompted the user to provide six different values, namely x, y, and z coordinate values corresponding to the world space and three additional coordinates corresponding to rotation values. We use this user input to move the UR3 robot in the specified joint configuration and we also compute the pose of the end-effector. In order to obtain user input and interact with the UR3 robot simulation we made use of the python CoppeliaSim remote API as in checkpoint 2.

We made use of the modern robotics python package as suggested by the professor to compute the pose of the end effector. The first step in implementing the inverse kinematics values after getting user input is to calculate the M matrix and screw vectors. To preface, these steps are identical to that of our checkpoint 2, and so we include the same explanations. To get M we make use of a CoppeliaSim remote API function called **simxGetJointMatrix** [3], which returns the homogeneous transformation matrix of a specified joint (which, in our case was the last joint of our UR3). We then compute the six spatial screws of the six revolute joints by inspecting the zero position of the UR3 robot arm in the simulation. In order to compute the linear velocity component of the spatial screws, we need a point along the axis of rotation of each joint, and in order to obtain this information, we make use of a helper function **get_joint**. At this point we have the general information needed to calculate inverse kinematics and we can pass the data into the inverse kinematics modern robotics functions.

A similar method of inverse kinematic solver was used in the final checkpoint, but instead of prompting the user for a point to move to, we directly change the position of a dummy target in CoppeliaSim, which we want the end effector to move to at any time. The desired motion of the end effector from its current position to a final destination is encapsulated in the **movel** function in Python.

Given two points, we can calculate a line in three-dimensional space that goes through both points. With help from a Mechatronics Ninja tutorial, to produce the smooth movement, we change the position of the dummy point slowly along the line from its current position to the end position. An inverse kinematics solver then solves for the joint angles needed for the UR3 to move to that dummy location.

Decision making

The decision-making pipeline of our project consists of six distinct steps:

1. The proximity sensor detects a block and the conveyor belt is halted.
2. The UR3 arm picks up the block from the conveyor belt.
3. The UR3 makes use of the vision sensor to capture an image of the block.
4. The image processing logic in our code extracts the color of the block from the image pixels.
5. Based on the color of the block, the block is placed at the corresponding waypoint.
6. The conveyor belt is restarted and steps 1-5 are repeated.

The above is a general breakdown of the different decision-making components in our project, but in this section, we will be focusing on the discussion of decision making as it pertains to the color estimation of blocks. In our backend Python logic, we obtain the output of the vision sensor in the form of a 32 x 32 x 3 numpy array (this represents an image of 32x32 pixels and 3 color channels). Once we obtain the image from the vision sensor, we look at the four corner pixels in the image and determine the color of the block as follows:

- If at least one of the four corner pixels is red (corresponding to a value of 255 in the red channel) we determine that it is an image of a red block.
- If at least one of the four corner pixels is green (corresponding to a value of 255 in the green channel) we determine that it is an image of a green block.

The reason that we probe four corner pixels of each image is because the quality of the picture captured by the vision sensor is not always perfect and is dependent on the orientation of the arm while it is reaching out to pick up a block. There are times when the captured image contains noisy pixel values at the edges. Thus, through experimentation, we found that probing four-pixel values offered a greater color detection accuracy for our system. This is one of the limitations we found our system has as we experimented.

Experimental Setup

After dipping our feet in the world of CoppeliaSim, and understanding the basics of scripts in LUA & Python, we were able to design the scene such that the end goal would be achieved. The robot we decided to use to implement the core of the process was the UR3, because we are familiar with its forward & inverse kinematic derivations, and the tutorials we saw were able to be extended to this robot. This UR3 is placed beside a conveyor belt, such that the blocks coming down the conveyor belt would be reachable in the robot's end effector space.

We have the conveyor belt moving at a velocity of 0.05 m/s until a block passes a proximity sensor, which forces the velocity back to 0 m/s. The blocks themselves are generated through an unordinary method of scripting that we learned from a Mechatronics Ninja tutorial. The table next to the table that the UR3 is placed upon holds a script, which creates a red or green block through the **sim.createPureShape** function every twenty seconds, and is then moved to the beginning of the conveyor belt in the frame of the table itself. This gives the illusion that blocks are being generated infinitely from the front of the conveyor belt. Also, the reason the red and green blocks are generated back and forth is to test how effectively Cleaner Pastures is able to distinguish between the two blocks, and to measure how often blocks are placed correctly in their final position.

Next, a Robotiq 85 gripper was attached to the end effector region of the UR3, and oriented such that it has no rotation with respect to the UR3 end effector frame. This gripper is controlled from the Python script to open or close as it is dropping or picking up a block with the **gripper** function. And finally, we have an orthographic vision sensor as described earlier, which is placed inside of the gripper, so that the decision-making logic will be able to see the color of a block as it's picking it up. Putting all of these elements together, we have the Cleaner Pastures scene, from which we are able to test how accurately and efficiently the UR3 will be able to sort "trash" from "recycling."

Data and Results

In order to test the accuracy and efficiency of our robot, we ran the simulation for five minutes, three separate times, and took note of how many blocks were placed correctly and incorrectly.

Table 1. Success rates of three trials of sorting block simulation

Trial	# Green Blocks Placed Correctly	# Red Blocks Placed Correctly	Speed (total good blocks/min)
1	10/11	9/11	3.8 blocks/min
2	11/11	11/12	4.4 blocks/min
3	8/8	6/8	2.8 blocks/min

From these trials, we can see that we have a robot block sorting speed ranging from **2.8 blocks/min** to **4.4 blocks/min**. These errors in picking up blocks & wide range of speed is a result of the gripper on the robot touching the block in a position that moves the block outside of the view of the camera, right before it's about to make a decision as to where the block should be placed. As a result, a block may be dropped multiple times before it is categorized correctly, or it gets dropped after the proximity sensor (which in that case, the block wouldn't be sorted at all).

By taking the total number of good blocks across all trials, and the total number of blocks attempted to be picked up across all trials, we see the system has a **90.12% success rate**.

Figure 6 is a chart that showcases the success & failure rates of each of the trials:

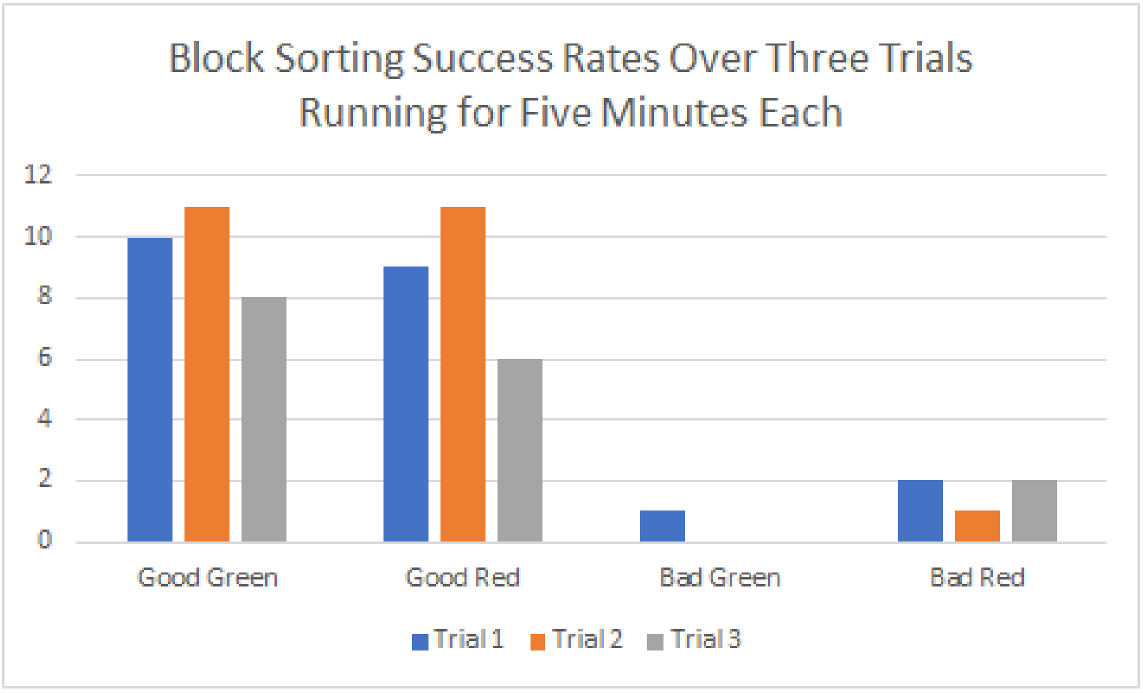


Figure 6. Success rates from Table 1 visualized in graph form

One example of what the scene looked like after running for five minutes with some blocks that the robot failed to sort is shown in Figure 4, where we can see a red block was picked up once, was not recognized by the vision sensor and dropped, and was left unsorted.

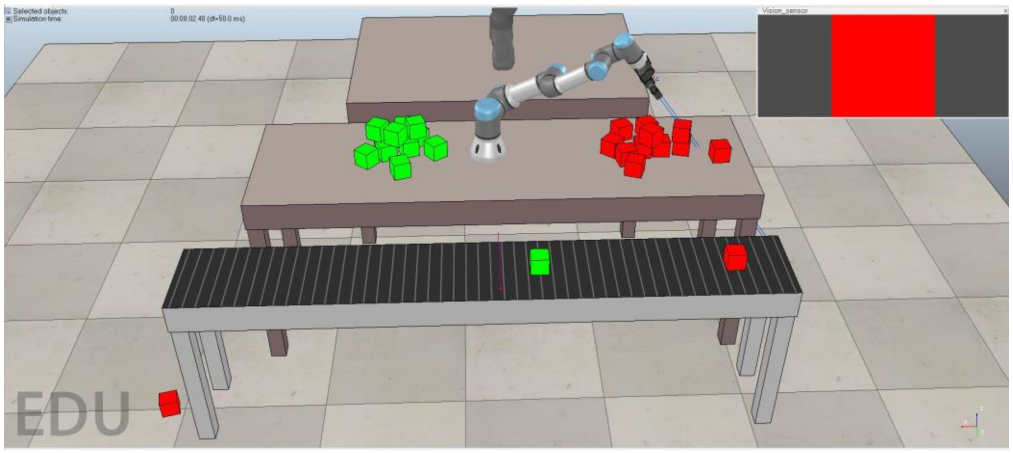


Figure 7. Screenshot of an example of the end state of the simulation where a red block failed to be picked up

Conclusion

By the end of our project work, we overcame difficulties with CoppeliaSim, the remote python API, and the challenges of implementing forward/kinematics and decision making. We were left with a functional sorting robot and the knowledge of how to make more complex simulations in the future. The results of this project definitely met our expectations given the time constraint and we are proud of what we were able to accomplish. There is however room for improvement, namely in terms of accuracy and complexity. Although our simulation is the exact same every time it is ran, there are minor random failures every so often, which is odd, and there may be a need for more dynamic waypoints to avoid awkward collisions or placements of blocks. For organizational purposes, it also would be cool if we could make blocks stack for the future, and there is also the visible limitation of one object being sorted at a time, which could be improved on with more UR3 arms configured for the task and slightly more complex logic. Additionally, our neural network plan that we originally had is still an interesting idea, and we believe our project so far is a great foundation for future additions.

Nikhil Simha, Vyom Thakkar, Neeloy Chakraborty
simha3, vnt2, neeloyc2
Cleaner Pastures
https://github.com/vyomthakkar/ece470finalproject
https://www.youtube.com/watch?v=pLc1tVorXGs

References

[1] www.youtube.com/watch?v=CVoV08T0Aqo&t=582s&ab_channel=mechatronicsNinja

[2] K. M. Lynch and F. C. Park, Modern Robotics: Mechanics, Planning, and Control. Cambridge: Cambridge University Press, 2017.

[3] “Regular API function,” simGetJointMatrix. [Online]. Available: https://www.coppeliarobotics.com/helpFiles/en/regularApi/simGetJointMatrix.htm. [Accessed: 21-Mar-2020].

[4] https://www.coppeliarobotics.com/helpFiles/en/visionSensors.htm

[5] “Regular API Function List (Alphabetical Order).” API Function List (Alphabetical Order), www.coppeliarobotics.com/helpFiles/en/apiFunctionListAlphabetical.htm.

[6] “Joint Simulation of Python and V-REP | Joint Simulation of Python and V-REP.” CSDN, 4 Aug. 2019, blog.csdn.net/qq_29945727/article/details/98469590.

[7] “Remote API Functions (Python).” CoppeliaSim, www.coppeliarobotics.com/helpFiles/en/remoteApiFunctionsPython.htm.