# ECE 470 Final Project Report (Spring 2020)

**NetIDs:** Jianhan Ma (jianhan3)
Boyang Zhou (boyangz3)
**Team Name:** Table Sorting Robot
**Link to GitHub:** https://github.com/Kingspenguin/ECE-470-project-update
**Link to YouTube video:** https://youtu.be/n0BksecMlLs

## 1 Abstract

In our final project, we simulated a table sorting robot system which includes a robot arm, a radar, a vision sensor as well as a proximity sensor. We aimed on achieving the goal that the system can detect all the objects on the table and sort them based on their categories. The importance of our project lies on the assumption that a great many people have encountered the situation that they have to deal with a table of mess after being tired after work. Our system is based on decision making and trajectory generation strategies. The radar will detect the objects and plan the routine for the robot arm, while the vision sensor helps the robot decide on whether to take a certain object to a preset place. The system finally work out fine, as the radar can precisely detect the objects, and the robot arm can grasp and transport the objects efficiently. Through this project, we got to know better of the general design logistics of robotic systems. As we made some simplifications by defining categories of the objects as “colors”, our future step may be set up a more complex and integrated logic for the vison sensor module.

## 2 Introduction

Our final project aims on simulating an interactive robot arm table sorter. We came up with this idea because we believe everyone has the experience that when you finish your study at your desk, feeling exhausted, what is waiting for you to deal with is a messy table with all the things scattered on it. Our automatic robot can help you free from all the clutters on your desk and help you spare more time to enjoy your life. The sorter is basically based on UR3 robot arm, a proximity sensor, a vision sensor, as well as a radar embedded on the table. The most important part of the robot is that it can accurately detect all the objects on the table and sort them based on their different categories. For simplification, “categories” here are perceived as

different colors by the detector. If an object is detected to be “green”, it means that it needs to be sorted. In short, the robot arm will wait for the radar to build a “map” of objects on the table, move the arm to these objects one by one, and decide whether to gasp the object to a “sorting area” based on the color information returned by the vision sensor.

## 3 Method

The methods that we apply in our projects are mainly related to concepts about rigid body motion, decision making and motion planning. For example, we implemented forward kinematics to move the robot arm to a specific configuration, and implemented a sensor feedback mechanism to guide the robot’s decision making on whether to gasp the object, and designed a trajectory for the robot to gasp the object to a preset area.

The block diagram below shows how the system generally works:

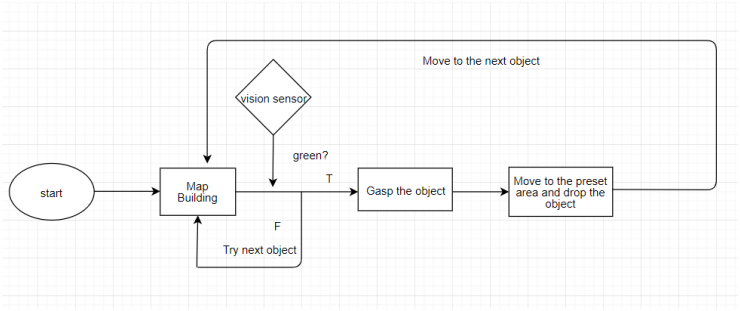



Figure 1. Block Diagram

First, the robot uses the radar to build the location map of objects on the table. Then it will visit all the objects and use the vision sensor to check whether an object is the one we want to sort to the right place. If the object needs to be sorted, which means that the vision sensor detects the object to be green, the robot will use the JacoHand to grip the object and move it to the preset destination. A proximity sensor was embedded in the JacoHand to check whether the object is properly gasped. Finally, the robot arm will move to the next objects until all the objects are sorted

The forward and inverse kinematics module in our implementation is similar to how it was implemented in the lab and we modified the home position to fit our scene.

```
def lab_fk(theta1, theta2, theta3, theta4, theta5, theta6):
    # Initialize the return_value
    return_value = [None, None, None, None, None, None]
    # =========== Implement joint angle to encoder expressions here ===========
    print("Foward kinematics calculated:\n")

    # ================== Your code starts here ====================#
    M, S = Get_MS()
    s1=[S[0][0],S[1][0],S[2][0],S[3][0],S[4][0],S[5][0]]
      s2=[S[0][1],S[1][1],S[2][1],S[3][1],S[4][1],S[5][1]]
      s3=[S[0][2],S[1][2],S[2][2],S[3][2],S[4][2],S[5][2]]
      s4=[S[0][3],S[1][3],S[2][3],S[3][3],S[4][3],S[5][3]]
      s5=[S[0][4],S[1][4],S[2][4],S[3][4],S[4][4],S[5][4]]
      s6=[S[0][5],S[1][5],S[2][5],S[3][5],S[4][5],S[5][5]]
      brs1=braket_s(s1)#change the s to the bracket form
      brs2=braket_s(s2)
      brs3=braket_s(s3)
      brs4=braket_s(s4)
      brs5=braket_s(s5)
      brs6=braket_s(s6)
      ra1=np.dot(brs1,dtor(theta1))
      #before dot the [s] and theta, transform the theta to radius unit
      ra2=np.dot(brs2,dtor(theta2))
      ra3=np.dot(brs3,dtor(theta3))
      ra4=np.dot(brs4,dtor(theta4))
      ra5=np.dot(brs5,dtor(theta5))
      ra6=np.dot(brs6,dtor(theta6))
      t0=np.dot(expm(ra1),expm(ra2))
      t1=np.dot(t0,expm(ra3))
      t2=np.dot(t1,expm(ra4))
      t3=np.dot(t2,expm(ra5))
      t4=np.dot(t3,expm(ra6))
      T=np.dot(t4,M)# the T06
      print('\nd06 is\n',[T[0][3],T[1][3],T[2][3]]) #the d06
      print('\nx={},y={},z={}\n'.format(T[0][3],T[1][3],T[2][3]))
      # ============================================================#

      print(str(T) + "\n")

      return_value[0] = theta1 + 180
      return_value[1] = theta2
      return_value[2] = theta3
      return_value[3] = theta4 - (0.5*180)
      return_value[4] = theta5
      return_value[5] = theta6
      return return_value

```

The core part, as well as the hardest part of our project are the radar and vision sensor. These two components help our robot to sense the environment on the table and enable our robot to be automatic, which means it can handle different situations of the messy table and makes its own decision to sort all the proper objects on the table.

The radar module would extract the polar coordinates data returned by the sensor, and transform it to the Euclidean coordinates. Then the module calculates the centroid of the objects and plot the map of outlines of objects.

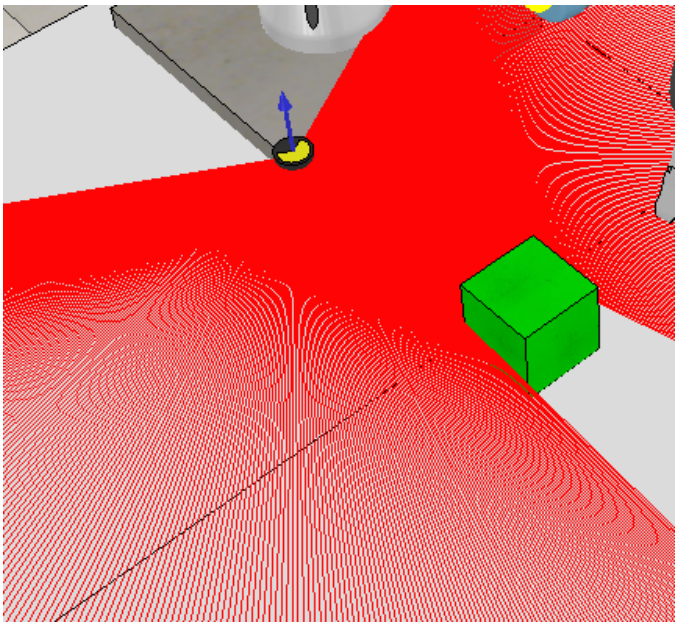


Figure 2. The radar embedded on the table

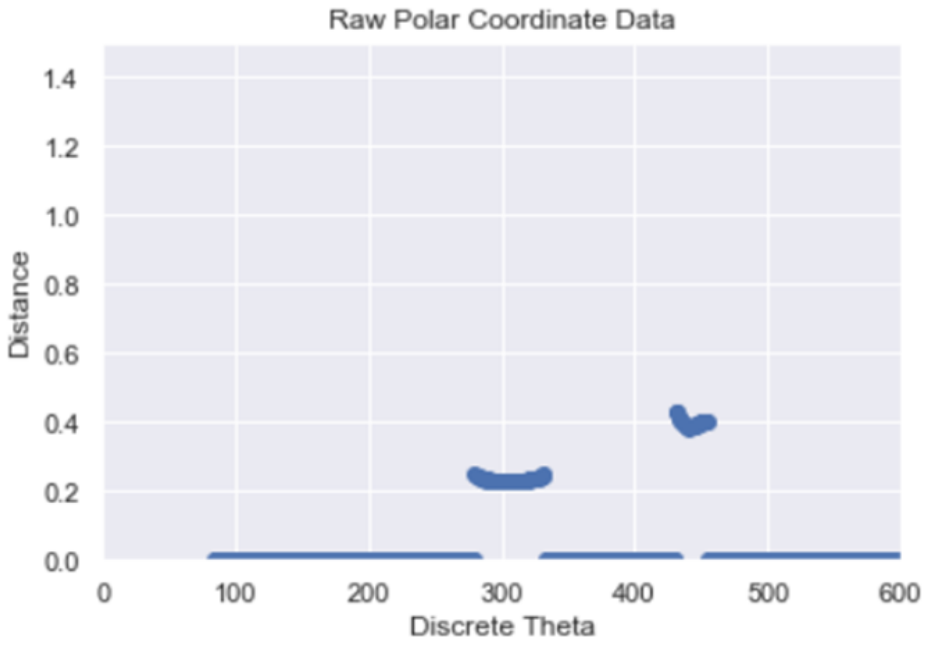



Figure 3. The Map of Raw Data

x_lable is the discrete angle of radar

y_lable is the distance between the objects and radar

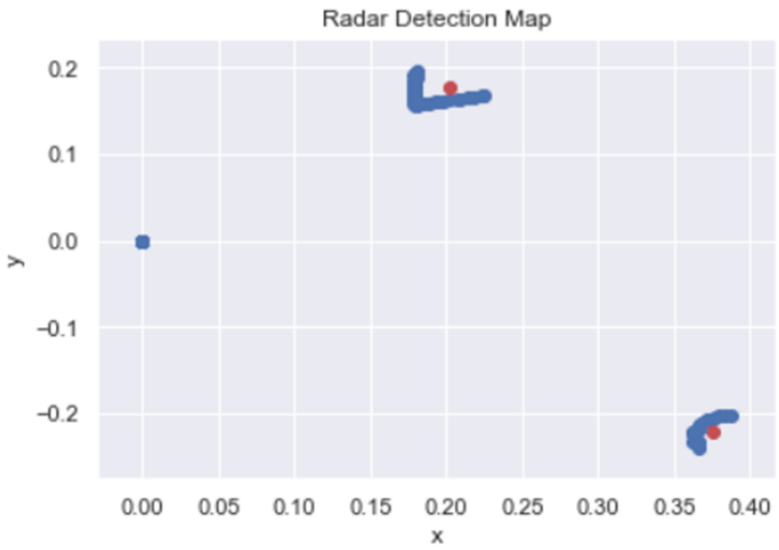



Figure 4. The Map of Processed Data

The blue dot is the outline of objects

The red dot is the approximate centroid of the objects by our algorithm

The radar_detection( ) function basically process the data and plot the detection map, the code we use here is inspired by an online tutorial [1].

```
def radar_detection(clientID):

    errorCode, ranges = vrep.simxGetStringSignal(clientID, 'scan ranges', vrep.simx_opmode_streaming)
    time.sleep(0.1)


    errorCode, ranges = vrep.simxGetStringSignal(clientID, 'scan ranges', vrep.simx_opmode_buffer)

    ranges = vrep.simxUnpackFloats(ranges)
      x = range(len(ranges))
      angle_discrete=x[86:600]
      distance = ranges[86:600]
      temp=np.ones(len(angle_discrete))*599
      real_angle=(temp-angle_discrete)*(240.0/684)
      #transfrom the polar frame coordinates to eculidian frame
      eculidian_x=[]
      eculidian_y=[]
      for i in range(len(real_angle)):
          y=distance[i]*cos(np.pi*real_angle[i]/180)#remember the x, y coordinates is interchanged in our reference frame
          x=distance[i]*sin(np.pi*real_angle[i]/180)
          eculidian_x.append(x)
          eculidian_y.append(y)

      blocks=[]
      count_block=0
      first_one=1 #signal to show that the current value is the first non zero value of a block data
      detection_map=np.zeros(len(distance))#store the whether a position is empty map
      for i in range(len(distance)):
          if distance[i]!=0 and first_one==1:#enter one block region
              first_one=0
              count_block+=1
              detection_map[i]=1
          if distance[i]==0 and first_one==0:
              first_one=1
      #find the change index
      change_one_index=[]
      change_zero_index=[]
```

```
    for i in range(1,len(detection_map)):
        if detection_map[i-1]==0 and detection_map[i]==1:
            change_one_index.append(i)
        if detection_map[i-1]==1 and detection_map[i]==0:
            change_zero_index.append(i)
    centroid_count_x=[]#store each sub block data
    for i in range(count_block):
        centroid_count_x.append([])
    centroid_count_y=[]#store each sub block data
    for i in range(count_block):
        centroid_count_y.append([])
    current_sub_block=-1
    for i in range(len(distance)):
        #check what sub block we are read currently
        if i in change_one_index:
            current_sub_block+=1
        if distance[i]!=0:#enter one block region
            centroid_count_x[current_sub_block].append(eculidian_x[i])
            centroid_count_y[current_sub_block].append(eculidian_y[i])
    # find each blocks' centroid
    centroid_x=[]
    centroid_y=[]
    for i in range(len(centroid_count_x)):
        sub_x=centroid_count_x[i]
        sub_y=centroid_count_y[i]
        centroid_x.append(max(sub_x)/2+min(sub_x)/2)#(np.max(sub_x)-
np.min(sub_x))/2)
        centroid_y.append(max(sub_y)/2+min(sub_y)/2)#(np.max(sub_y)-
np.min(sub_y))/2)
    plt.figure()
    plt.scatter(eculidian_x,eculidian_y)
    plt.xlabel('x')
    plt.ylabel('y')
    plt.title('Radar Detection Map')
    plt.scatter(centroid_x,centroid_y,marker='o', c='r')
    return eculidian_x,eculidian_y,centroid_x,centroid_y,count_block
    #return the eculidian_x eculidian_y list, the block centroid list
and the counted block number
```

The vision sensor module generally guides the robot's decision making procedure. By calculating the RGB data returned by the vision sensor, the robot arm would know whether the object it is trying to gasp is green or not [2]. Besides, a proximity sensor is also fixed on the robot arm, which checks whether the arm has properly gasped the object.

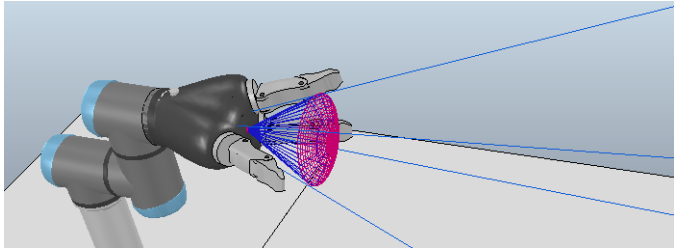


Figure 5. JacoHand with cone-type proximity sensor and a vision sensor in it

And the work of the vison sensor module is generally carried out by the "track_green_object( )" function, which is modified from a an code from a Github directory[2], which extracts the RGB information from the depth image returned by the vision sensor.

```
1. def track_green_object(image):
2.
3.      # Blur the image to reduce noise
4.      blur = cv2.GaussianBlur(image, (5,5),0)
5.
6.      # Convert BGR to HSV
7.      hsv = cv2.cvtColor(blur, cv2.COLOR_BGR2HSV)
8.
9.      # Threshold the HSV image for only green colors
10.       lower_green = numpy.array([40,70,70])
11.       upper_green = numpy.array([80,200,200])
12.
13.       # Threshold the HSV image to get only green colors
14.       mask = cv2.inRange(hsv, lower_green, upper_green)
15.
16.       # Blur the mask
17.       bmask = cv2.GaussianBlur(mask, (5,5),0)
18.
19.       # Take the moments to get the centroid
20.       moments = cv2.moments(bmask)
21.       m00 = moments['m00']
22.       centroid_x, centroid_y = None, None
23.       if m00 != 0:
24.           centroid_x = int(moments['m10']/m00)
25.           centroid_y = int(moments['m01']/m00)
26.
27.       # Assume no centroid
28.       ctr = None
29.
30.       # Use centroid if it exists
31.       if centroid_x != None and centroid_y != None:
32.           ctr = (centroid_x, centroid_y)
33.       return ctr
```

Using proximity sensor is an efficient way to detect targets. This sensor works most precisely in cone-type (Figure 5). Proximity sensor is placed in the JacoHand to detect items passed to it, which is written in the **JacoHandHasItem** Function. To enable the sensor in the code, we need to run the proximity twice: Run with **simx_opmode_streaming** mode first time and **simx_opmode_buffer** in the second time. **time.sleep(1)** must be added, or two instructions will be done at the same time, resulting the failure of detection. [1]

```
def suctionHasItem():

        ret, state, arr1, value, arr2 = vrep.simxReadProximitySensor(clientID,
suction_sensor_handle, vrep.simx_opmode_streaming)

        time.sleep(0.1)

        ret, state, arr1, value, arr2 = vrep.simxReadProximitySensor(clientID,
suction_sensor_handle, vrep.simx_opmode_buffer)

        return state
```

To make the JacoHand grasp and release things, we define two functions named **suctionGrasp()** and **suctionRelease()**, the connection between python code and vrep is through **simxSetStringSignal**. Besides, we need to replace code in JacoHand's child script

```
1. def suctionGrasp():
2.   vrep.simxSetStringSignal(clientID,'suctionPad','true',vrep.simx_opmode
   _oneshot)
3.   time.sleep(1)
4.   print("Try to grasp.")
5.   return
6.
7. def suctionRelease():
8.   vrep.simxSetStringSignal(clientID,'suctionPad','false',vrep.simx_opmod
   e_oneshot)
9.   time.sleep(1)
10.          print("Release item.")
11.          return
```

## 4 Experimental Setup

To achieve our task to sort the table, we need one UR3 robot and JacoHand to simulate the human arms and hand, and some cubes to simulate objects (Figure 6).

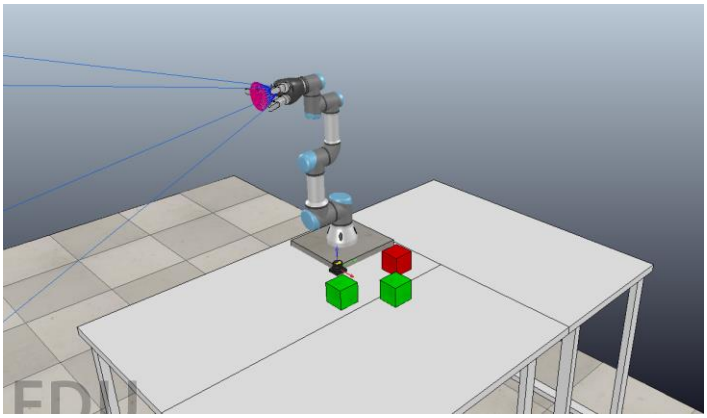


Figure 6. Vrep scene

One challenge is to make the JacoHand grasp the item precisely. “Precisely” has two meanings: Item shall neither fall to the ground when the robot is moving nor get stuck in the JacoHand. When simulating catching process, we adjust some variables to change the physical status of cubes: Cubes should be detectable to proximity sensor (Figure 4); reduce the mass and principal moments of spheres and increase Newton friction coefficients to prevent it from falling (Figure 5).

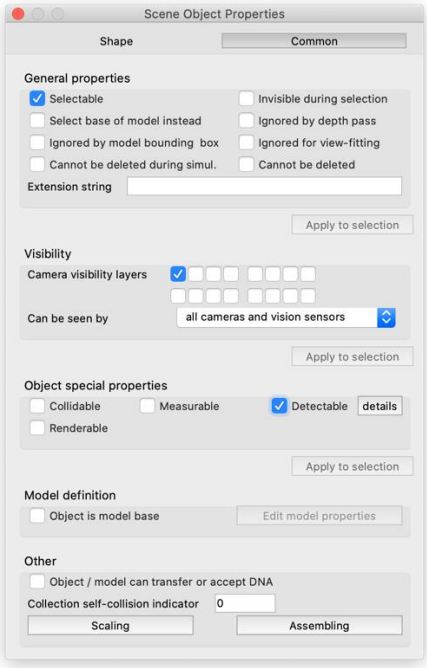



Figure 7. Detectable Property to enable cubes to be detected by sensor

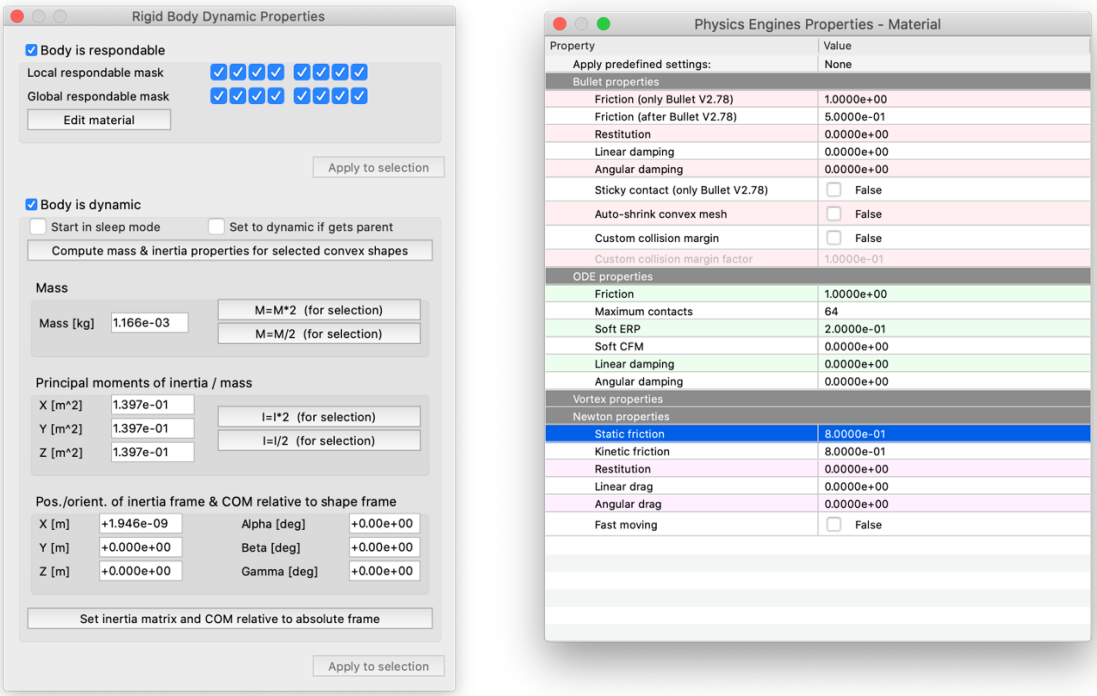


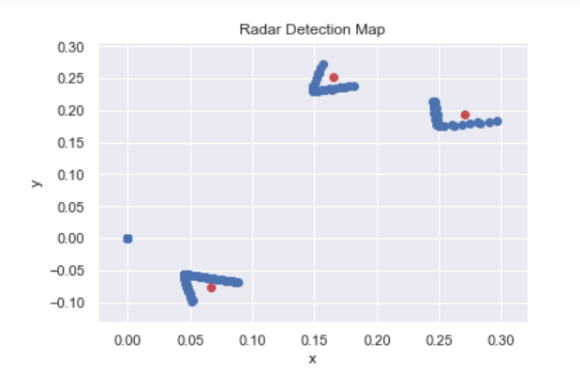

Figure 8. Adjust mass, principal moments, Static friction and Kinetic friction of the cudes. Mass and principal moments should be as small as possible. Friction coefficients should not be larger than 1.

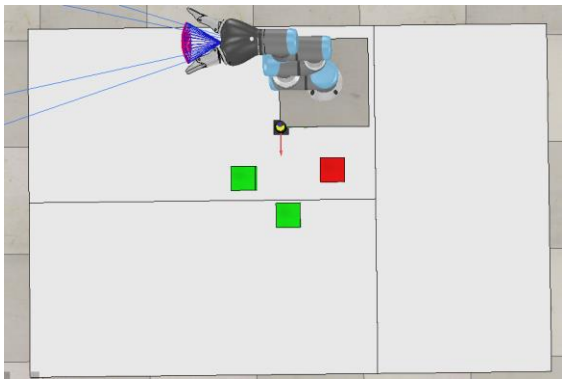
We also did several tests to adjust position of JacoHand where it should release cubes. As the detection area of the radar is limited, there isn't much space to place all the cubes, and the accuracy of the radar defers between different locations. After like hundreds of tires of adjustment we finally placed the cubes to a relatively accurate position so that the cubes can be transported between both areas.

## 5 Data and Results

During the trials, "success" is defined as the following: First, the radar needs to build an accurate map for objects. Second is that the arm should move to and grasp the objects precisely.

In achieving the first goal, one difficulty we encountered is that the height at which we place the radar would influence the accuracy greatly, as the objects get detected by blocking the waves from the radar. We adjusted the height and finally find a relatively proper one:



The errors that caused difficulties in the second goal actually also comes from the occasionally bad performance of the radar. We solve this problem by adding some offsets the detected coordinates of objects. Although it will be very inconvenient if we change the locations of the cubes, but this is the most efficient solution for now.

```
[109.59945578189371, -49.283798227020306, 112.97651314262825, -153.69271491560795, -90.0, -70.40054421810628]
The calculated invKinematics theta 0 is -70.40054421810629
The calculated invKinematics theta 1 is -49.283798227020306
The calculated invKinematics theta 2 is 112.97651314262825
The calculated invKinematics theta 3 is -63.692714915607944
The calculated invKinematics theta 4 is -90.0
The calculated invKinematics theta 5 is -70.40054421810628
```

## 6 Conclusion

In our final project we design an interactive robot arm table sorter with the UR3 robot arm, a proximity sensor, a vision sensor and a radar. We control and simulate our table sorter robot with the remoteApi interface of CoppeliaSim and Python. Through the python code we accomplish our object color detection function and object location detection function of our automatic table sorting robot. Based on the two main functions and our robot arm moving functions (inverse-kinematics and robot hand control function), we finish our project with the functionality that the robot can detect the location of objects on the table and sort the targeted color objects to the pre-fixed destination regardless of the number of objects and the initial distribution of the objects on the table.

In this project, we have learned how to simulate a robot working scene on the CoppeliaSim and control the simulation with the remoteApi interface between CoppeliaSim and Python. For the simulation control part, first, we have learnt how to apply the inverse kinematics algorithm

according to the real robot arm and use the algorithm to control the movement of the robot arm. Besides, we learnt to use the opencv module in python for color detection. Finally, we understand the working principle of a simple radar and create a function to interpret the raw data created by the radar.

If we were given more time, we want to create a more advanced detection system. For the location detection part, we will try to add more radar sensor on the table and make them collaborate to get a more accurate location. Because, our current design with a single radar can only detect one side of an object and use that single side location information to predict the location of the object. This will introduce relatively large error in our location detection system and will be significantly influenced by the shape of the object (our current system gives a much more accurate result with cubic object than the sphere one). By add more radar, we can detect more side of the object (even the outline of the object). With the more detailed information, we can definitely give a more accurate location detection result. For the color detection part, we may change the location of the vision sensor. Instead of fixing it to the robot arm, we may change it to the location right above table. In this way, we can build a color map of the table and enable our robot to decide a sort process based on the overall situation. By doing this, we can cut off the unnecessary movement of visiting the untargeted object (our current design is that we let our robot arm visits every object and detect color one by one).

## 7 References


[1] 培培哥, “V-REP Simulation: Obtaining Data From lase Radar in Python.” *CSDN*, blog.csdn.net/u014695839/article/details/88377586. Accessed 10 March 2019.

[2] nemilya, “V-Rep API Python OpenCV Demo” Github, https://github.com/nemilya/vrep-api-python-opencv. Accessed 8 April 2016.