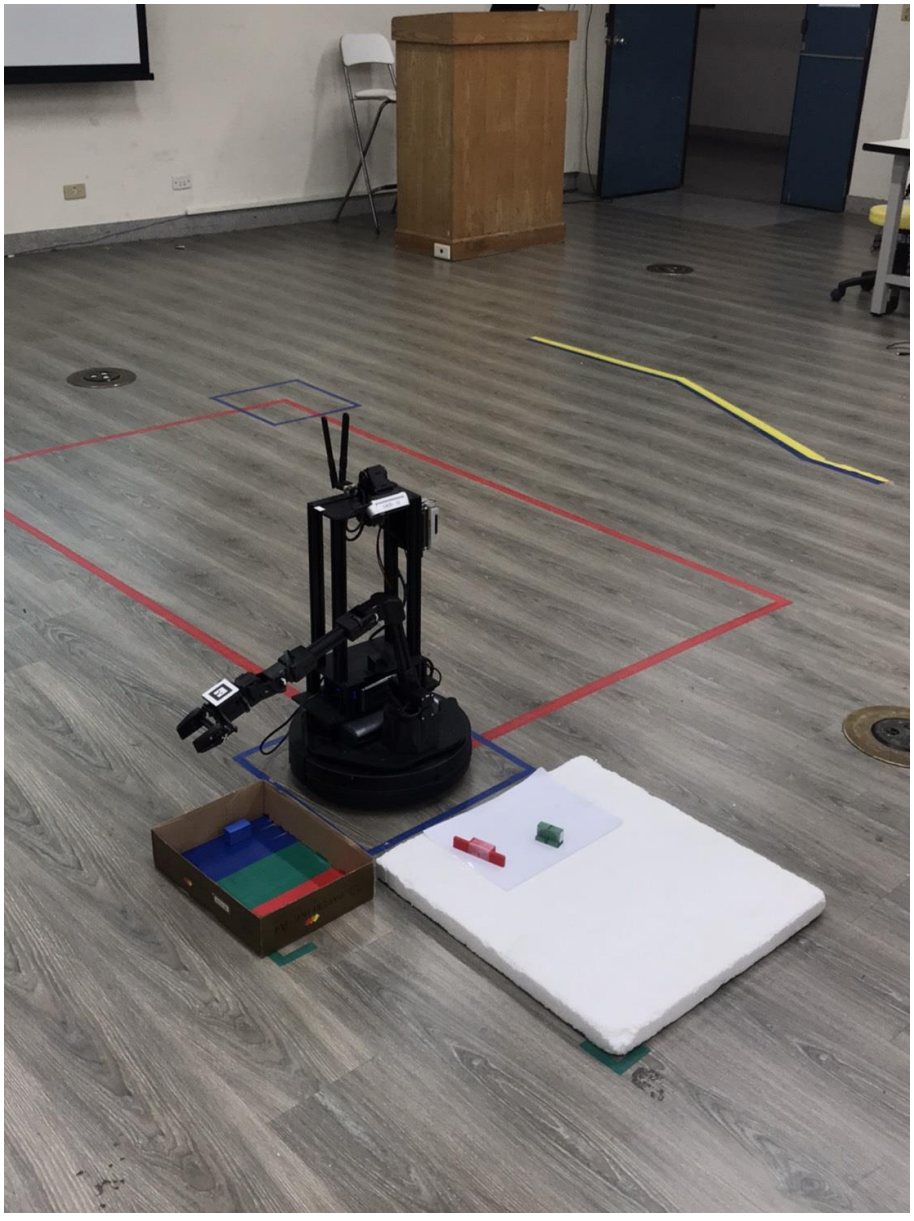
期末專題　　Report

[UEE3713] 機器人系統與應用設計實作 2020

學號: 106030009　　**姓名:**葉蓁　　**日期:** 2020/12/25

1. 目的:

本次實驗任務目標為：
(1)讓機器人精準定位到夾取物件的座標位置點
(2)機器人能用深度相機辨識並找出目標夾取物之位置
(3)機器人能準確夾取物件，並根據其顏色，將其分類放置到分類盒指定位置

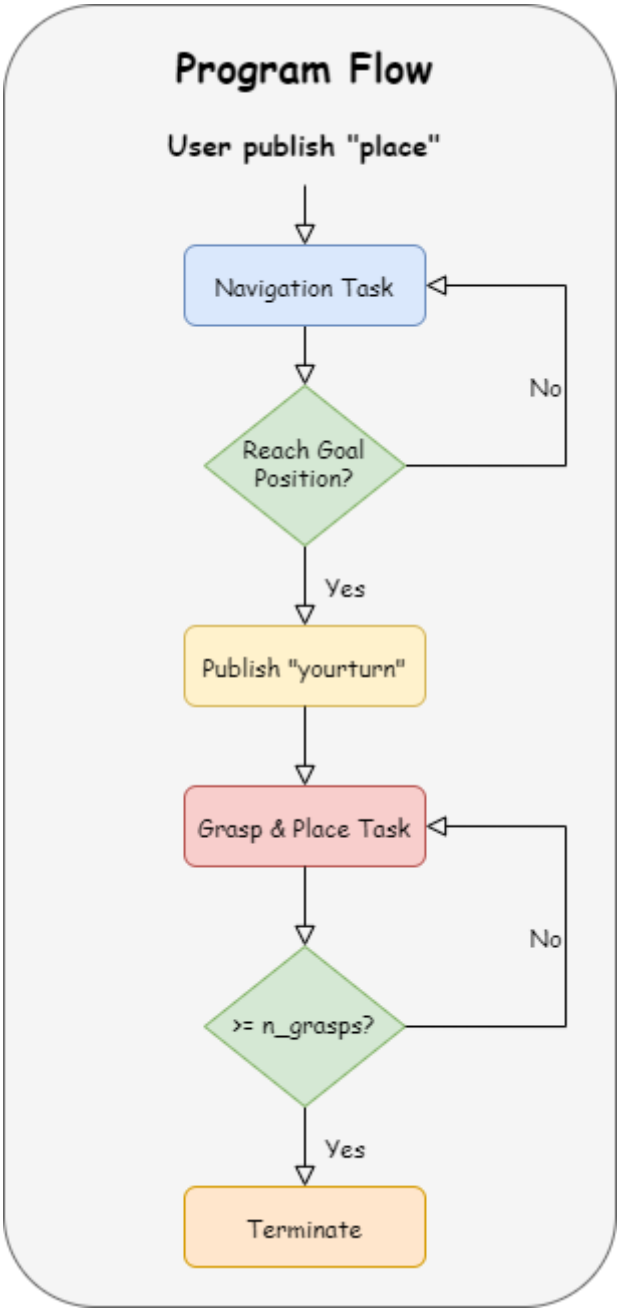


2. 設計說明:

首先概述一下整體程式流程：

(1) Program Flow

首先需要執行定位導航的機器人底盤運動控制，完成此任務後，機器人抵達定位點，發出 /yourturn Topic, 啟動 locobot.py 中的 callback function:
此時進入第二階段程式：夾取任務。
根據使用者(我們)輸入的參數: **--n_grasps 決定執行幾次抓取任務。**
我們設定 6 次，若有抓取失敗，可以直接重新嘗試。



(2) Localization & Navigation

底盤運動導航部分，我們使用 odometry 作為定位方式，仿照之前運動控制 assignment 01 的方式，透過位置點的傳輸，加上 ROS service 方式，將目標定位點傳給 LoCoBot 的 navigation_controller node。

navigation_controller.cpp :

```
216
217         //get robot position
218         tf::StampedTransform transform;
219         tf::Quaternion q;
220         try {
221             tf_.lookupTransform("/odom", "/base_link", ros::Time(0), transform);
222             q = transform.getRotation();
223         }
```

```
83  const double NavigationController::D2R = M_PI / 180.0;          // degrees to rad
84  const double NavigationController::R2D = 180.0 / M_PI;          // rad to degrees
85  const double NavigationController::MaxRobotLinearVelocity = 0.2;        // m/s
86  const double NavigationController::MaxRobotAngularVelocity = 40.0 * D2R;    // radian/s 35
87  const int NavigationController::beam_num = 30;
```

Command_pub_srv_gui.cpp:

```
74  #define STOP_INDEX 0
75  #include <boost/math/constants/constants.hpp>
76  const double pi = boost::math::constants::pi<double>();
77
78  double loc_x[1] = {2.1};
79  double loc_y[1] = {-1.24};
80  double loc_theta[5] = {0}; //radian
81  int loc_index = 0;
82  bool not_finish = true;
```

我們設定：**目標點 (x ,y) = (2.1, -1.2) Theta (yaw) = 0.0 radian**

navigation_controller node 會根據 odometry 的計算，定位其位置，根據離目標點還有多遠，計算出當下需給馬達的速度命令，驅動馬達使機器人前進。

```
 98      // while (myfile >> num && ros::ok()) {
 99      while (ros::ok()) {
100          x = 0.0;
101          y = 0.0;
102          theta = 0.0;
103
104
105      ////////////////////////////////TODO/////////////////////////////////
106
107          x = loc_x[loc_index];
108          y = loc_y[loc_index];
109          theta = loc_theta[loc_index];
110
111      ////////////////////////////////TODO/////////////////////////////////
112
113
114          //ROS_INFO_STREAM("x = " << x << " y = " << y << " theta = " << theta);
115          if(not_finish) srv.request.type = 2;
116          else srv.request.type = 0;
117          srv.request.x = x;
118          srv.request.y = y;
119          srv.request.theta = theta;
120
```

```
113
114          //ROS_INFO_STREAM("x = " << x << " y = " << y << " theta = " << theta);
115          if(not_finish) srv.request.type = 2;
116          else srv.request.type = 0;
117          srv.request.x = x;
118          srv.request.y = y;
119          srv.request.theta = theta;
120
121          if(client_.call(srv)) {
122              ROS_INFO("call service success");
123              srv.request.type = 0;
124              while(srv.response.run_completed == false) { // still running.
125                  usleep(100000);
126                  ROS_INFO("hello servie");
127                  client_.call(srv);
128              }
129              // reach point. Send another place.
130              // loc_index++;
131              // if(loc_index==4){
132              //     loc_index = 4;
133              // }
134              // else{
135              //     loc_index++;
136              // }
137              sleep(wait);
138          }
139          else {
140              ROS_INFO("call service fail");
141          }
142
```

```
143         if(not_finish) srv.request.type = 1;
144         else srv.request.type = 0;
145 
146         if(client_.call(srv)) {
147             ROS_INFO("call service success");
148             srv.request.type = 0;
149             while(srv.response.run_completed == false) { // still running.
150                 usleep(100000);
151                 ROS_INFO("hello servie");
152                 client_.call(srv);
153             }
154             // reach point. Send another place.
155             // loc_index++;
156             if(loc_index== STOP_INDEX){
157                 not_finish = false;
158                 loc_index = STOP_INDEX;
159                 break;
160             }
161             else{
162                 loc_index++;
163             }
164             sleep(wait);
165         }
166         else {
167             ROS_INFO("call service fail");
168         }
169 
```

抵達定位點後，由 flag_switch publish true 到 /yourturn Topic，使 locobot.py 開始執行 callback function。

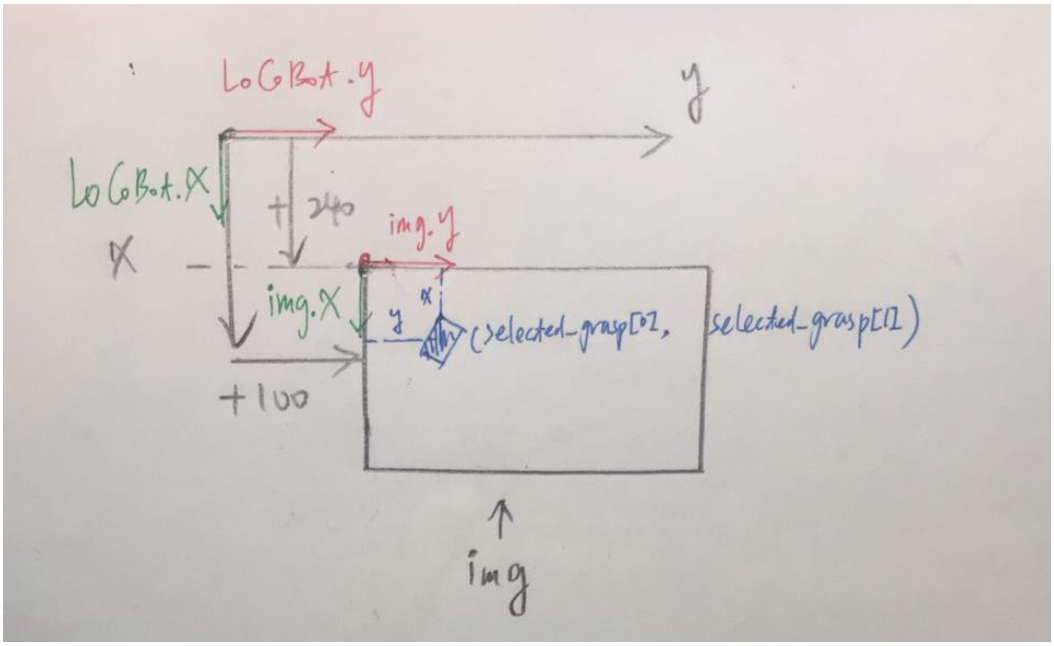
下圖為：初始化 CommandPubSrvGui() 時其 subscribe /place 並 publish /yourturn 的設定。 /place 就是我們使用者要“開始”定位導航任務時，需要送的一個命令。(任意的 string 即可)

```
28 
29 CommandPubSrvGui::CommandPubSrvGui()
30 {
31     place_cmd_ = nh_.subscribe<std_msgs::String>("place", 1000, boost::bind(&CommandPubSrvGui::get_place, this, _1));
32     client_ = nh_.serviceClient<navigation_controller::command>("pos_cmd");
33 
34     flag_switch = nh_.advertise<std_msgs::Bool>("yourturn", 1);
35 
36     //cmd_switch.data
37 }
38 
```

```
3
4       cmd_switch.data = true;
5       for(int i=0;i<3;i++){
6           ROS_INFO("finish");
7           flag_switch.publish(cmd_switch);
8           ROS_ERROR("-----------------------------------");
9           ros::Duration(1.0).sleep();
0       }
1
2   }
3
```

(3) Perception and Reasoning

這步驟其實算是本次任務要能成功最關鍵的地方：
如何用影像辨識，找出物體的座標位置?
可惜我們沒有足夠的時間去研究更好的演算法，
我們直接使用 助教範例提供的 compute_grasp() 去 predict 出目前相片中他認為 特徵點最明顯的位置。
由於我們發現 selected_graph 回傳的是一個 1 維的 vector, 有四個 element，分別代表: **[x, y, z, theta]**
[x, y, z] 表示物體的 絕對位置座標，單位: 公尺(m)
這邊 theta 指的是 yaw 角，單位是 radian，繞著 z 軸轉多少度。
我們從 selected graph 取得 x, y 座標位置，加上相機左上角點的起始座標位置(offset)，**得到以手臂座標為基準點，物件的真正座標。**



```python
    def compute_grasp(self, dims=[(240, 480), (100, 540)], display_grasp=False):
        """
        Runs the grasp model to generate the best predicted grasp.

        :param dims: List of tuples of min and max indices of the image axis.
        :param display_grasp: Displays image of the grasp.
        :type dims: list
        :type display_grasp: bool

        :returns: Grasp configuration
        :rtype: list
        """

        img = self.robot.camera.get_rgb()
        rospy.loginfo('img: {}'.format(img))
        img = img[dims[0][0]:dims[0][1], dims[1][0]:dims[1][1]]
        # selected_grasp = [183, 221, -1.5707963267948966, 1.0422693]
        selected_grasp = list(self.grasp_model.predict(img))
        rospy.loginfo('img: {}'.format(selected_grasp))
        rospy.loginfo('Pixel grasp: {}'.format(selected_grasp))
        img_grasp = copy.deepcopy(selected_grasp)
        x = selected_grasp[0]
        y = selected_grasp[1]
        r = img[x][y][0]
        g = img[x][y][1]
        b = img[x][y][2]
    print("r: "+str(r))
    print("g: "+str(g))
    print("b: "+str(b))

```

接著我們從 img[x][y] 這個 entry, 再去分析 R, G, B 三個 channels 的值。若 R 最大，則判斷他是紅色物件，若 G 最大，判斷為綠色物件。若 B 最大，判斷為藍色物件。

```python
    print("r: "+str(r))
    print("g: "+str(g))
    print("b: "+str(b))

        index = [r,g,b].index(max([r,g,b]))
        if index == 0:
            color = 'red'
        elif index==1:
            color = 'green'
        else:
            color = 'blue'

        selected_grasp[0] += dims[0][0]
        selected_grasp[1] += dims[1][0]
        selected_grasp[:2] = self.get_3D(selected_grasp[:2])[:2]
        selected_grasp[2] = selected_grasp[2]
        rospy.loginfo('imselected_graspg[0]: {}'.format(selected_grasp[0]))
        rospy.loginfo('imselected_graspg[1]: {}'.format(selected_grasp[1]))
        rospy.loginfo('imselected_graspg[2]: {}'.format(selected_grasp[2]))
        if display_grasp:
            self.grasp_model.display_predicted_image()
            # im_name = '{}.png'.format(time.time())
            # cv2.imwrite('~/Desktop/grasp_images/{}'.format(im_name), self.grasp_model._disp_I)
        return selected_grasp, color

    def grasp(self, grasp_pose, color):
        """
```

接著我們會把 grab_pose 和 color 當參數回傳到 callback function, 再傳給 grasp() 函式，使其得以判斷 "現在是夾哪個顏色的物件，要放置到哪"

(4) Manipulation

首先在每次夾取前，需要將手臂 reset 到不會遮住 camera 的位置。
我們設定為右方位置。
透過量測 current_joints 並 trial and error 後，我們使用 F.K. 設定馬達轉角。
程式碼 def reset() :

```
98    def reset(self):
99        """
100       Resets the arm to it's retract position.
101
102       :returns: Success of the reset procedure
103       :rtype: bool
104       """
105       # current_joints = self.robot.arm.get_joint_angles()
106
107       # First go_home
108   self.robot.camera.set_pan(0)
109   self.robot.camera.set_tilt(0.85)
110
111
112       self.robot.arm.go_home()
113       self.robot.arm.set_joint_positions([-3.14/2, 0.2, 0.04601942, 0.00920388, 0.00613592], plan=False)
114
115       self.robot.gripper.open()
116       success = True
117
118       return success
119
```

Reset 我們做三件事情：

a. 讓 camera 回到最適合的角度 (set_pan(0) 和 set_tilt(0.85))
b. 讓手臂先 go_home() 再用 F.K.轉到右邊避免遮住相機視線
c. 讓手臂的 gripper 打開

手臂控制路徑規劃部分：

由於 compute_grasp 幫我們計算出“欲夾取物件之姿態”，適合使用 Inverse Kinematics 方式(已知目標點，計算手臂各軸馬達所需之轉角)。

我們將手臂夾取分為以下幾個步驟：

a. 移動到 “欲夾取物件”的正上方
b. 往下移動到 “欲夾取物件”的 z 座標
c. 進行夾取動作（關閉夾爪）
d. 垂直向上先移動到一個安全的高度
e. 移動到 “欲放置盒子內位置”的正上方
f. 垂直向下移動到合適的高度
g. 打開夾爪，放置物件
h. 向上移動，回到 reset 位置

我們將移動任務分開為這麼多個步驟的原因，是因為就算給定兩點 A, B ，

若單純用 I.K. 計算而不限制條件，手臂會計算 “其認為之最佳路徑”，過程中可能有 Z 座標或其他非預期之移動，導致撞到盒子或其他物件。舉例：假設 A, B 兩點的 Z 座標同樣高度，但距離相對較遠，手臂不一定會只轉動第 1 軸的馬達，他可能搭配其他軸的馬達，因此過程中夾爪 (end effector)的 z 座標可能有上下位移而打到東西的可能。
避免此問題有幾個方法：
(1)同樣用 I.K.計算，但多設幾個路徑必須經過點。
(2)使用 F. K . 來直接寫死馬達絕對轉角。
使用第一個方法的缺點：需要多寫幾個中間點(看起來較冗長)
第二個方法缺點為：需要先 tune 過各個位置對應之大概轉角。
綜合考量後，**我們最後選擇第一個方法：多設幾個中間經過點。**
從程式碼實現上述設計：

1. 我們觀察我們手臂會偏離目標座標左下各一點點，因此我們加了一小段 offset，讓他往右上各 2 公分偏移，藉此我們手臂能夠更準確落在物體正上方。
2. 讓手臂移動到物體正上方 “INIT_HEIGHT” 處，並讓夾爪打開

```
238     def grasp(self, grasp_pose, color):
239         """
240         Performs manipulation operations to grasp at the desired pose.
241 
242         :param grasp_pose: Desired grasp pose for grasping.
243         :type grasp_pose: list
244 
245         :returns: Success of grasping procedure.
246         :rtype: bool
247         """
248     grasp_pose[0] += 0.02
249     grasp_pose[1] -= 0.02
250 
251 
252         # move to the top of object
253         grasp_pose[2] = INIT_HEIGHT
254         self.set_pose(grasp_pose)
255 
256         #current_joints = self.robot.arm.get_joint_angles()
257         #self.robot.arm.set_joint_positions([current_joints[0]-0.25, current_jo
258 
259 
260         # open gripper
261         self.robot.gripper.open()
262 
263     #grasp_pose[0] += 0.05
264     #grasp_pose[1] -= 0.05
265 
266 
```

3. 接著執行一系列動作：
   往下到欲夾取的高度 “GRASP_HEIGHT”，關閉夾爪(夾取)，往上到 “欲移動的高度”(MOVING_HEIGHT)。

```
266
267         # move down.
268         grasp_pose[2] = GRASP_HEIGHT
269         self.set_pose(grasp_pose)
270         # close gripper
271         self.robot.gripper.close()
272         # move up
273         grasp_pose[2] = MOVING_HEIGHT
274         self.set_pose(grasp_pose)
275
276         place_pose = []
277         # Which color to place
```

4. 接著根據不同顏色(傳進此函式的參數 color)，決定放置點在哪。
   如果是 'red' ，要放到 PLACE_RED_POSE。其他以此類推。
   注意我們多使用一個 F.K. 讓手臂真正放下前，先回到盒子正上方，壁面移動時 Z 座標有往下的動作而卡到地板或其他障礙物的風險。

```
277         # Which color to place
278         if color == 'red':
279             place_pose =  PLACE_RED_POSE
280             # self.set_pose(place_pose)
281             # Now: use reset position
282             self.robot.arm.set_joint_positions([-3.14/2, 0.2, 0.04601942, 0.00920388, 0.00613592], plan=False)
283             self.set_pose(place_pose)
284             # robot.arm.set_joint_positions([-1.5, 0.5, 0.3, -0.7, 0.0], plan=False)
285
286         elif color == 'green':
287
288             self.robot.arm.set_joint_positions([-3.14/2, 0.2, 0.04601942, 0.00920388, 0.00613592], plan=False)
289
290             place_pose = PLACE_GREEN_POSE
291             self.set_pose(place_pose)
292             # robot.arm.set_joint_positions([-1.5, 0.5, 0.3, -0.7, 0.0], plan=False)
293         else:
294             self.robot.arm.set_joint_positions([-3.14/2, 0.2, 0.04601942, 0.00920388, 0.00613592], plan=False)
295
296             place_pose =  PLACE_BLUE_POSE
297             self.set_pose(place_pose)
298             # robot.arm.set_joint_positions([-1.5, 0.5, 0.3, -0.7, 0.0], plan=False)
299
```

5. 移動到盒子上方後：首先往下到 PLACE_HEIGHT(欲放開物體的高度，低一點避免物體彈開)，接著鬆開夾爪放置物體。然後先往上移動回到 MOVING_HEIGHT，再 return 回到下一次 grasp 任務迴圈(會先 reset)。回到 MOVING_HEIGHT 的用意一樣是避免打到東西。

```
300         # place lower
301         place_pose[2] = PLACE_HEIGHT
302         self.set_pose(place_pose)
303         # open gripper and drop the object
304         self.robot.gripper.open()
305         print("Place "+color+" Done.")
306 
307         # move up
308         place_pose[2] = MOVING_HEIGHT
309         self.set_pose(place_pose)
310 
311         #self.robot.arm.set_joint_positions([-3.14/2, 0.2, 0.04601942, 0.
312 
313 
314         return True
315 
```

另外因為有許多參數須要透過實驗 再加上 fine tuning, 我們使用參數化將多個需要常常改到的參數拉到最上面，宣告成全域變數，方便使用者改動。

```
47  # ====== Parameters ======= #
48  # We should tune these parameters from time to time.(occasionally)
49  INIT_HEIGHT = 0.25
50  GRASP_HEIGHT = 0.165
51  MOVING_HEIGHT = 0.3
52  PLACE_HEIGHT = 0.15
53 
54      # x: robot front  y: robot left
55  PLACE_RED_POSE = [0.23, -0.35, INIT_HEIGHT, 0]
56  PLACE_GREEN_POSE=[ 0.15, -0.35, INIT_HEIGHT, 0]
57  PLACE_BLUE_POSE =[0.05, -0.35, INIT_HEIGHT, 0]
58 
```

3. 結果

第一次 demo ：

定位導航有些歪掉，導致 camera 只照到藍色和綠色物件。

夾取任務：

藍色方塊：成功夾取，但因為 LoCoBot 本身離盒子太遠，放置的位置(寫死)不夠遠，因此無法放入盒內。

綠色方塊：夾取失敗。

紅色方塊：因相機沒照到，辨識失敗。

第二次 demo:

定位導航有調整回來，進入藍色框框成功。

夾取任務：

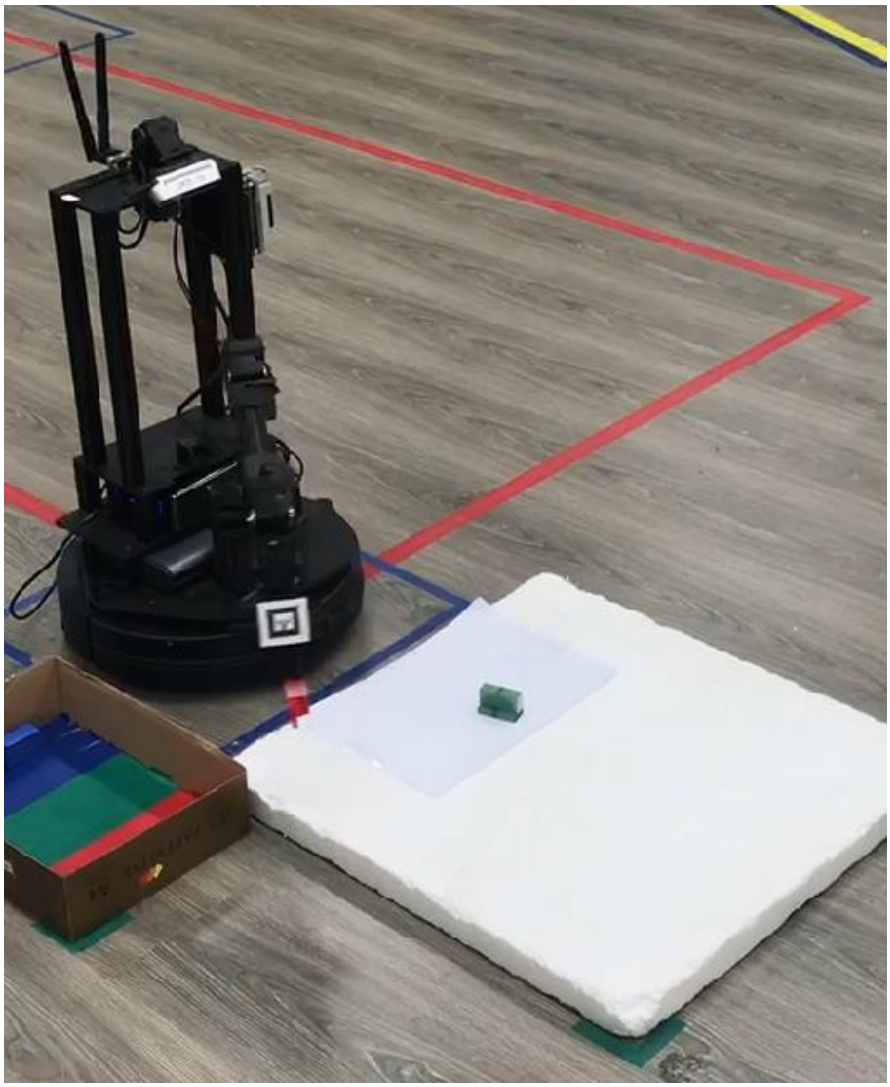
藍色方塊：成功夾取，成功放置

紅色方塊：成功夾取，但因判斷的點位置，b channel 值大於 r channel(中心點判斷錯誤)，所以放置失敗(放到藍色盒子內)

綠色方塊：夾取失敗，兩次夾取都因中心點太低，沒能夾到綠色

方塊。

中心點判斷錯誤的原因，或許可以使用更高的 n_sample 數，用時間換取準確度。



4. 討論:

過程中遇到許多問題：

首先是 “理解程式碼”的部分：

(1)如何取得 r,g,b channels 的值?

透過 print(img), print(selected_graph)後觀察發現：

從 selected_graph[0],[1]拿到 x, y 座標位置

再從 img[x][y][0], [1], [2]拿到 r, g, b 三個值!

(2)如何使用 I.K.和 F.K.的 API?

可以先用 get_current_joints() 讀取現在轉角，作為 F.K.設定的參考。

另外當天還有其他技術問題：

(1)搜不到 locobot 的 wifi hotspot

解法：打開 ~/.bashrc, 把 export ROS master : ….10.42.0.2 (或.1)全部註解掉，並勾選 enable wifi(若接螢幕使用 gui 介面)，記得要重新 source bashrc (. ~/.bashrc)然後重新開一個 terminal (因原先的 termina 的環境變數已經設好)

(2)LoCoBot 校正 camera 的第三個 script 無法運行

原因是因為需要 升級 Pytorch 的版本，但因升級完後還需要降級才能使用這次任務需用到的 api 功能，先考慮不使用這個方法。

(3)Segmentation Fault

可能因為 ctrl + C 沒有完全終止掉曾經開啟的 process，導致同樣的 process 被開了很多遍，占用系統資源，導致 cpu 出錯，memory 存取出現問題。

解法：使用 kill %1 或 kill %2, %3 等去 kill 掉所有尚未終止的 processes。

(4)Bad Callback : locobot.py cannot connect to move_group

(5)Run Time Error

不知道為甚麼計算出的 z 會是 0，導致 Raise RunTimeError，在 locobot.py 的 get3D_camera() 函式中的 get_Z()中。

因觀察：三個物件的 z 值大概是 0.6 左右，解法: 在裡面把 z 設為 0.6，暫時解掉莫名的 run time error 產生。

(6)從 visual studio code 編輯器複製程式碼，或是從記事本等編輯器複製，python 會讀出一些奇怪的亂碼導致編譯失敗。

因為我們的機器從最初的 10 號中間換過 6 號，導致最後要再使用 10 號時須要做程式碼搬移的動作。

最後解法：使用 scp 從 local 端 直接 copy 整個 folder 到遠端 LoCoBot 路徑上 (scp -r ~/navigation_controller locobot@10.42.0.1:~low_cost_ws_my/src/.../navigation_controller)

心得：

1. 首先謝謝三位老師精心準備這學期的課程，真的收穫滿滿!

還有各個助教用心準備的教材，以及不厭其煩地為我們解惑，解決許多 Error

和 debug，非常感謝助教! 真的無限感激!
謝謝老師給我們機會親自動手實作，撰寫程式並實現在 LoCoBot 機器人平台上，千載難逢的機會與永生難忘的寶貴經驗!

2. 針對期末專題競賽，Demo 完畢才發現：
單純以完成這次的 task 的話，在 grasp 其實不用知道現在是夾甚麼顏色，因為特定顏色的方塊會放在固定的位置，只需知道現在 grasp_pose 的 y 座標大概落在哪個區域，即可判斷現在夾取的是哪個物件，就不會有上述 ”因為框框政中心點是在白色紙，導致他判斷成錯的 r, g, b channel 物件” 的問題了~~ QQ 可惜太慢想到。
不過以長久之計，以及要 general 和 robust 的話，不該用這種偷吃步的方法哈哈，理想的優化方法還是要專注在 object detection 的演算法，以及夾爪判定位置的準確度，以及手臂運動控制的定位精度。

附上我們共同撰寫的 hackmd 筆記：
https://hackmd.io/@lovelyrachelhsia/PyRobot

Merry Christmas & Happy New Year!