ECE 470: Lab 3
Camera Sensing and Particle Filter

Yutong Xie
NetID: yutongx6
Partner: Ananmay Jain
TA: Ben Walt
Section: Wednesday 2PM

October 16, 2019

1 Introduction

Camera is a significant sensor for a intelligent robot, robot relies on the camera to feel the world and acquire the environment information. Hence, how to make the robot know his position in the real world is essentially important. In this lab, we try to use **OpenCV** library to find small orange blobs and their center coordinates. Then we should be able to find the perspective transform between the camera frame and world frame. At the same time, we will implement the *Particle Filter* algorithm with real world vision measurements. We treat the orange blob as the vehicle and measure state of the vehicle using the above camera.

2 Method

2.1 Threshold camera image to distinguish only bright orange pixels

First, we want to distinguish only the bright orange pixels from the camera image. Generally, we can set the threshold of R,G,B value for the orange. However, we tried to use H,S,V (Hue, Saturation, Value) color space to achieve our goal, because it can handle different shading and lighting conditions much better and give more accurate results. When we execute the *lab3_image_tf_exec.py*, a red dot appears. Additionally, it will give the pixel position and corresponding H,S,V color. Originally, the red dot is at the left corner of our crop image (I will talk it later in the report). I moved the orange blob to that and get several H,S,V values. Then, I move set the red dot to several positions including the bottom right corner of the crop image and get other several H,S,V values. The reason to do this is that I want to get the H,S,V values of orange under different lighting and shading situations. Then, I get the result and set the lower bound and upper bound.

```
lower =(10,200,155)      # oranger lower
upper = (20,255,245)   # orange upper
```

First, the image will be transferred to the HSV image. And when I get the HSV range of orange color shown above. We can use the following instruction to convert the HSV image to a bineary image of only orange pixels. All orange pixels are one(white) and all other pixels are zeros(black).

```
mask_image = cv2.inRange(hsv_image, lower, upper)
```

After we mask the image, we get the following result.

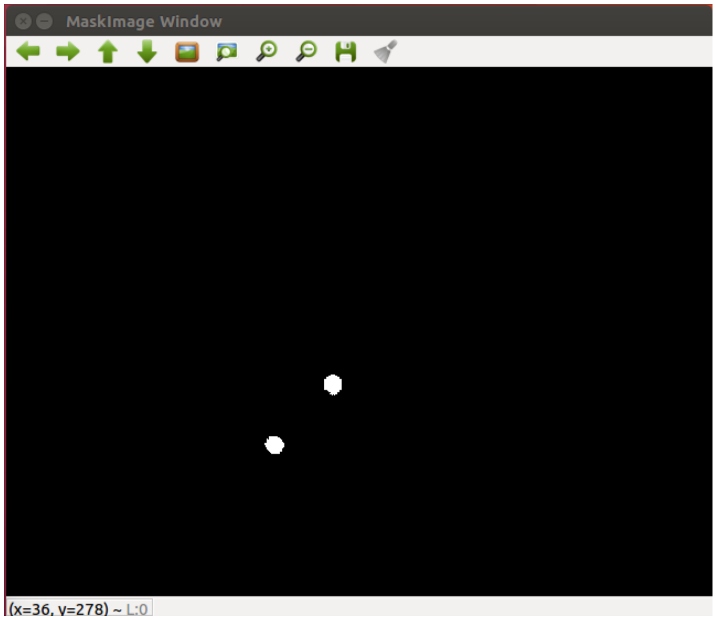


Figure 1: The binary image after masking.

Also, in this step, we set a crop image which is part of the mask image. The requirement is that the crop image should cover all the grid so that the blob will not extend the boundary of it. We modify the values of **crop_top_row, crop_bottom_row, crop_top_col and crop_bottom_col** to set the crop image.

2.2 Use the OpenCV simpleBlobdetector library function to find the centroid in pixels of one or multiple circular orange blobs

In this section, we tried to use **OpenCV's simpleBlobdetector** library to detect the blob in the camera view. We set several parameters in the function **blob_search_init** as shown below.

```
# Filter by Color
params.filterByColor = False
# Filter by Area.
params.filterByArea = True
params.minArea = 100
# Filter by Circularity
params.filterByCircularity = True
params.minCircularity = 0.2
# Filter by Inerita
params.filterByInertia = False
# Filter by Convexity
params.filterByConvexity = False
# Any other params to set
params.minThreshold = 200
params.maxThreshold = 255
```

At the beginning, we found that the detection result is not accurate when we only filter by circularity. Then, we set the **minArea** to 100, which means that the detector can filter out all the blobs that have less then 100 pixels. It shows that the effect is good enough to detect all the blobs. After the blobs are found, the number of blobs and the centroid of each blobs are printed in the terminal. Additionally, we draw red circles around all the blobs using function **cv2.drawKeypoints()** to help find the blob easily. I paste the code here for reference. The function **cv2.threshold()** in the code is to inverse the

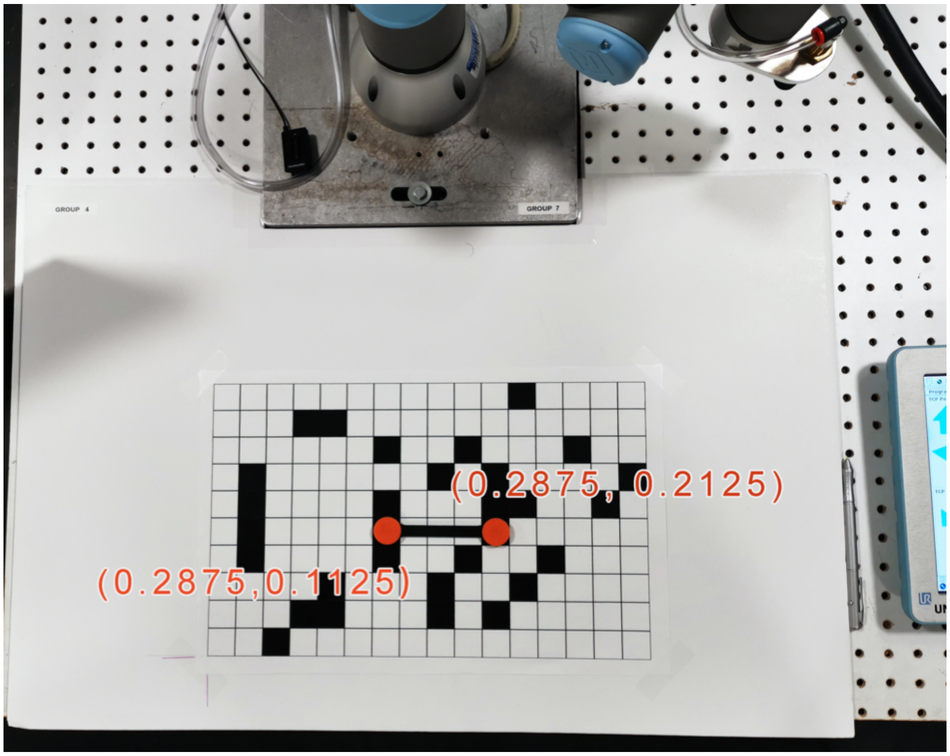


Figure 2: Where to put the calibration stick

binary image. We think that it will be easier for the detector to detect the blobs.

```
im_with_keypoints = image

retval, threshold = cv2.threshold(mask_image, 200, 255, cv2.THRESH_BINARY_INV)

keypoints = detector.detect(threshold)

im_with_keypoints = cv2.drawKeypoints(image, keypoints, np.array([]), (0,0,255),
    cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)
```

2.3 Perspective Transform

In this section, we should determine the perspective transform of the camera. There are six unknown values you must determine:$O_r, O_c, \beta, \theta, T_x, T_y$. (O_r, O_c) is the principal point given by the row and column coordinates of the center of the image. We can calculate the value of them by dividing the width and height variables by 2.

$$O_r = \frac{1}{2}height \tag{1}$$

$$O_c = \frac{1}{2}width \tag{2}$$

Then, we move to find the rest four parameters. A calibration stick is used in this procedure and it is placed as the Fig 2 shows. β is a constant value that scales distances in space to distances in the image. We can get the centroid of two blobs using the method we mentioned before. The value of beta can be achieved by dividing the distance between two blobs by the exact distance which is 0.1m. Assume the centroid of two blobs is $(x_1, x_2), (y_1, y_2)$, we can calculate the distance between them using this equation.

$$d = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2} \tag{3}$$

Then, β can be easily achieved by equation $\beta = d/0.1$. The parameter θ is the angle of rotation between the world frame and the camera frame.The calibration stick is horizontal in the world frame but not for the camera frame. Here, we try to use arcsin function to find the angle.

$$\theta = arcsin((y_2 - y_1)/d) \tag{4}$$

When we acquire the angle between the world frame and the camera frame, we can calculate the rotation matrix.

$$R = \begin{bmatrix} cos\theta & -sin\theta \\ sin\theta & cos\theta \end{bmatrix} \tag{5}$$

For the point in camera frame and world frame, we have the relationship below.

$$\begin{bmatrix} x^w \\ y^w \end{bmatrix} = \begin{bmatrix} cos\theta & -sin\theta \\ sin\theta & cos\theta \end{bmatrix} \begin{bmatrix} x^c \\ y^c \end{bmatrix} + \begin{bmatrix} T_x \\ T_y \end{bmatrix} \tag{6}$$

Now we have the rotation matrix, the position of left blob in world frame (0.2875, 0.1125), and the position of left blob in the camera frame. Then, we can calculate the Tx and Ty. The code here is how I find four parameters.

```
d = ( (x1 - x2)**2 + (y1 - y2)**2 )**0.5

beta = d/0.1  # Pixels per meter

theta = math.asin((y2 - y1)/ d)

Rz = np.array([[math.cos(theta), -1*math.sin(theta)], [math.sin(theta), math.cos(theta
    )]])

A = Rz.dot(np.array([[x1/beta], [y1/beta]]))

cam_origin_x = yw - A[0,0]
cam_origin_y = xw - A[1,0]

tx = cam_origin_x
ty = cam_origin_y
```

When we get the four parameters, we assign them to the corresponding variables in file **lab3_image_exec.py** for the next step.

2.4 Implement particle filter HW problem with real world vision measurements

In this step, we just fill the particle filter code into the corresponding function in file **lab3_move_exec.py**. I modify the **__init__** function first to generate the particles. Then, I add the function **Sample_Motion_Model** and function **Measurement_Model**. What we need to pay attention to is that the **Measurement_Model** function needs x and y parameter here. This is how we make particle filter get the coordinates found by USB camera. Finally, we add the **calcPosition** function.

3 Data and Results

3.1 Threshold camera image to distinguish only bright orange pixels

In this section, we get the mask image as shown in Fig 1, and we also get the crop image which covers all the grid.

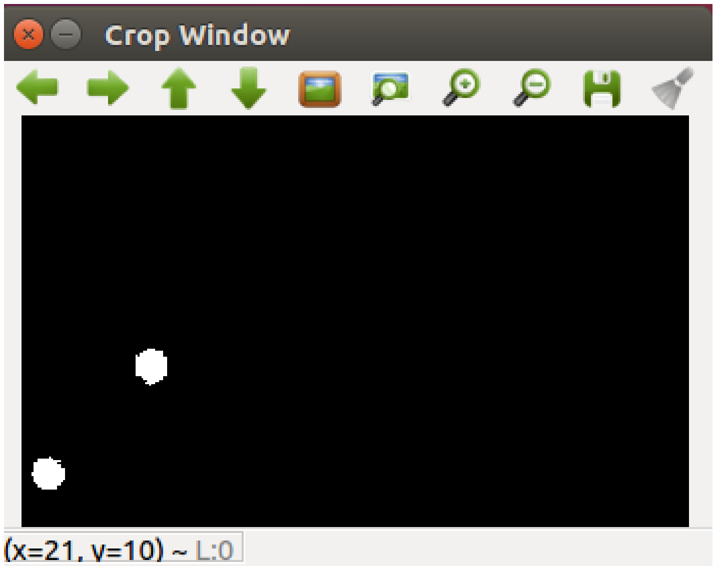


Figure 3: The crop image derived by mask image

3.2 Use the OpenCV simpleBlobdetector library function to find the centroid in pixels of one or multiple circular orange blobs

In this section, we are successful to find all the blob and print their centroid in the terminal as shown in Fig 6. Also, We draw a circle around the blob which can be seen in the following figure.

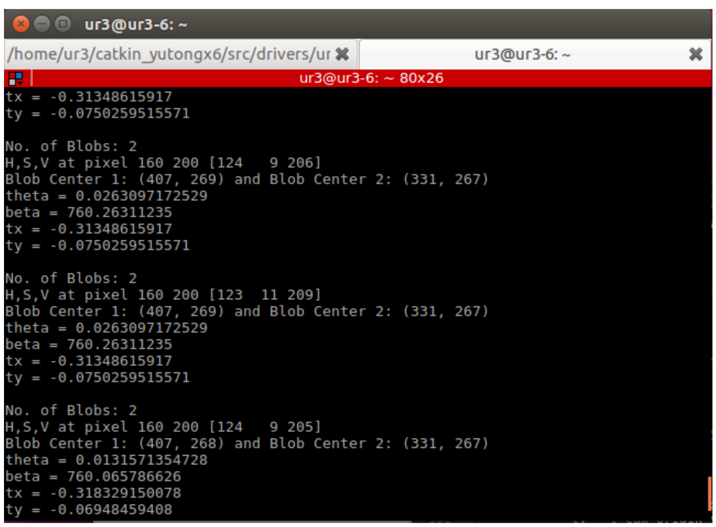


Figure 4: The blobs detected by OpenCV detector.

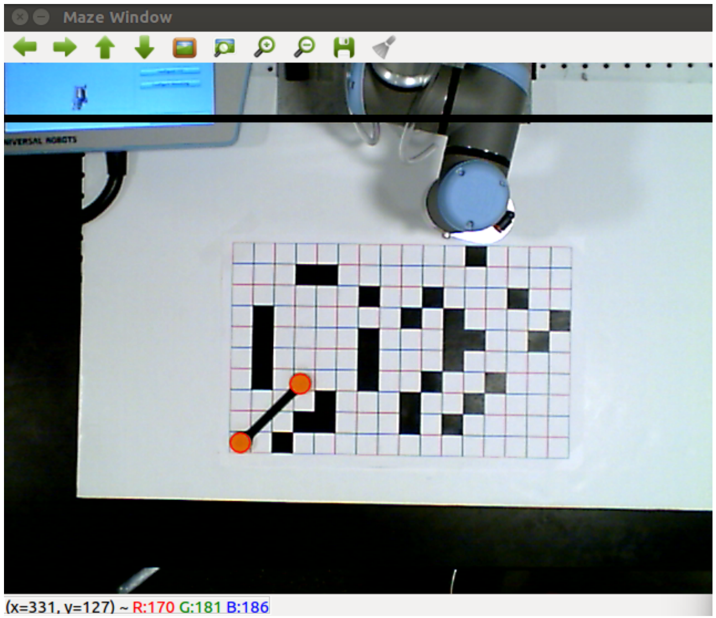


Figure 5: The keypoints image with indication circle

3.3 Perspective Transform

In this section, we run the **lab3_image_tf_exec.py** file to calculate four parameters of perspective transform. The result is shown below.

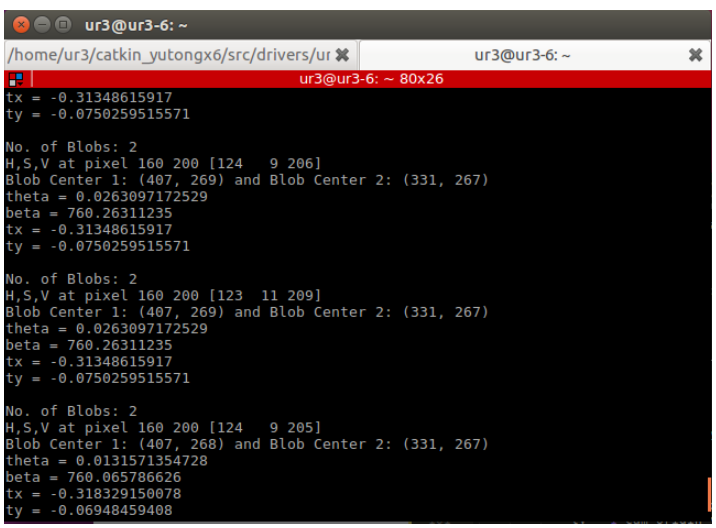
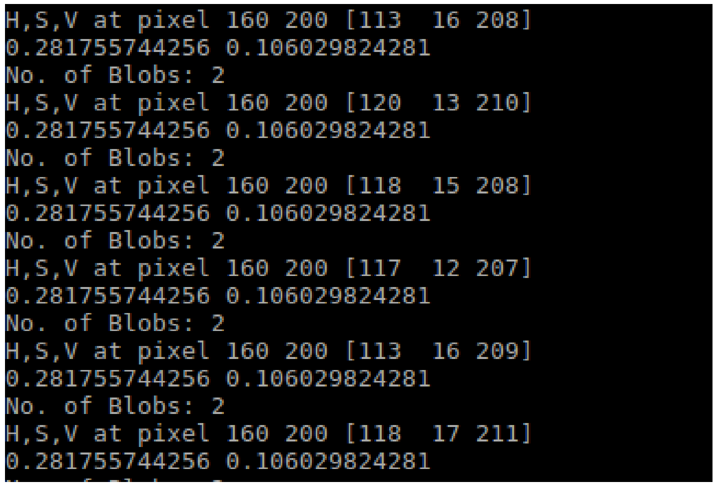


Figure 6: The transform parameters calculated by lab_3_image_tf_exec.py

When I input the four parameters into **lab3_image_exec.py** file and rosrun that, it gives the position of left blob which shown in Fig 7. The coordinate is (0.28,0.11) which is very close to the given coordinate.



```
H,S,V at pixel 160 200 [113  16 208]
0.281755744256 0.106029824281
No. of Blobs: 2
H,S,V at pixel 160 200 [120  13 210]
0.281755744256 0.106029824281
No. of Blobs: 2
H,S,V at pixel 160 200 [118  15 208]
0.281755744256 0.106029824281
No. of Blobs: 2
H,S,V at pixel 160 200 [117  12 207]
0.281755744256 0.106029824281
No. of Blobs: 2
H,S,V at pixel 160 200 [113  16 209]
0.281755744256 0.106029824281
No. of Blobs: 2
H,S,V at pixel 160 200 [118  17 211]
0.281755744256 0.106029824281
```

Figure 7: The position of left blob calculated by lab_3_image_exec.py

3.4 Implement particle filter HW problem with real world vision measurements

In this section, we use the robotic arm to implement the particle filter. We can use the direction key to move the arm and the turtle will move correspondingly. As shown in Fig 8, the blob move to one of the grids by user manipulation. In the turtle graphics interface, the turtle move to the corresponding grid.

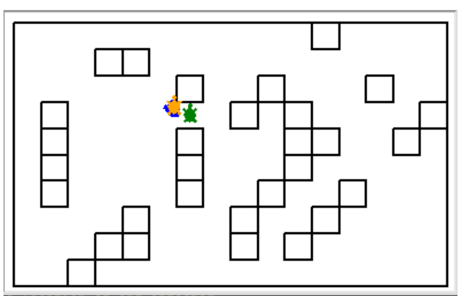


Figure 8: The output of turtle graphics interface.

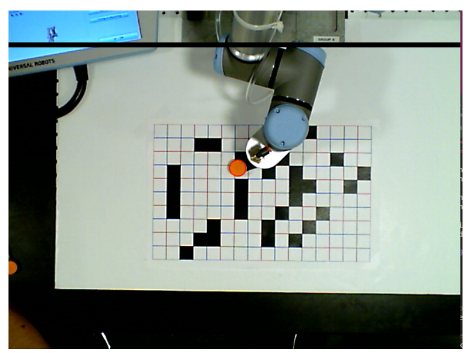


Figure 9: The image view of camera.

From the figure above, we can see that the estimated position doesn't correspond to the actual position well. I think whether the blob is in the center of grid is very important. When I placed the blob in the center of grid, the effect of particle filter improved a lot. Fig 10 demonstrates that the estimated position and actual position overlapped very well.

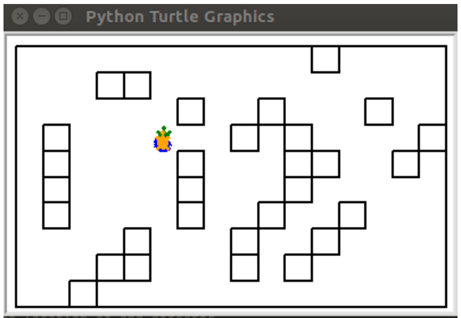


Figure 10: The output of turtle graphics interface.

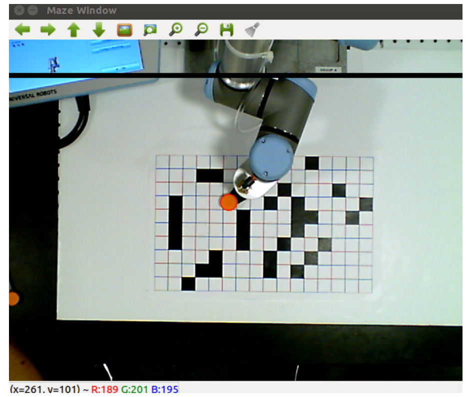


Figure 11: Placed the blob to the center of grid.

4 Conclusion

In this lab, I learned how to use OpenCV and simplbe BlobDetector library to find the blob from camera image. Different parameters can be set for different detect tasks. Also, I can find the transformation matrix between camera frame and world frame by using calibration stick. Finally, I learned how to implement the particle filter with vision measurement and see the result in turtle graphics interface.

References

[1] HSL and HSV. (2019, September 23). Retrieved from https://en.wikipedia.org/wiki/HSL_and_HSV.

[2] Mallick, S. (2015, February 17). Home. Retrieved from http://www.learnopencv.com/blob-detection-using-opencv-python-c/.

Appendices

A Python Script: lab3_func.py

```
#!/usr/bin/env python

import cv2
import numpy as np

"""
To init blob search params, will be init (called) in the ImageConverter class
"""
def blob_search_init():

  # Setup SimpleBlobDetector parameters.
  params = cv2.SimpleBlobDetector_Params()

  ################# Your Code Start Here #################

  # Filter by Color
  params.filterByColor = False
  # Filter by Area.
  params.filterByArea = True
```

```
    params.minArea = 100
    # Filter by Circularity
    params.filterByCircularity = True
    params.minCircularity = 0.2
    # Filter by Inerita
    params.filterByInertia = False
    # Filter by Convexity
    params.filterByConvexity = False
    # Any other params to set???
    params.minThreshold = 200
    params.maxThreshold = 255


    ################## Your Code End Here ##################

    # Create a detector with the parameters
    blob_detector = cv2.SimpleBlobDetector_create(params)

    return blob_detector


"""
To find blobs in an image, will be called in the callback function of image_sub subscriber
"""
def blob_search(image, detector):

    # Convert the color image into the HSV color space
    hsv_image = cv2.cvtColor(image, cv2.COLOR_BGR2HSV)


    ############################# Your Code Start Here #############################

    # Find lower & upper for orange

    lower =(10,200,155)     # oranger lower
    upper = (20,255,245)   # orange upper

    ############################## Your Code End Here ##############################


    # Define a mask using the lower and upper bounds of the orange color
    mask_image = cv2.inRange(hsv_image, lower, upper)

    crop_top_row = 160
    crop_bottom_row = 370
    crop_top_col = 250
    crop_bottom_col = 570

    crop_image = mask_image[crop_top_row:crop_bottom_row, crop_top_col:crop_bottom_col]

    blob_image_center = []

    ############################# Your Code Start Here #############################

    # Call opencv simpleBlobDetector functions here to find centroid of all large enough blobs in
    # crop_image. Make sure to add crop_top_row and crop_top_col to the centroid row and column found

    # Make sure this blob center is in the full image pixels not the cropped image pixels
```

```python

  im_with_keypoints = image

  retval, threshold = cv2.threshold(mask_image, 200, 255, cv2.THRESH_BINARY_INV)
  # Draw centers on each blob, append all the centers to blob_image_center as string
    in format "x y"
  keypoints = detector.detect(threshold)

  for i in range(len(keypoints)):
    blob_image_center.append([keypoints[i].pt[0], keypoints[i].pt[1]])

  print("No. of Blobs: " + str(len(blob_image_center)))


  im_with_keypoints = cv2.drawKeypoints(image, keypoints, np.array([]), (0,0,255), cv2
    .DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)

  ############################## Your Code End Here ##############################

  # Draw small circle at pixel coordinate crop_top_col, crop_top_row so you can move a
     color
  # under that pixel location and see what the HSV values are for that color.
  image = cv2.circle(image, (int(crop_top_col), int(crop_top_row)), 3, (0, 0, 255),
    -1)
  print('H,S,V at pixel ' + str(crop_top_row) + ' ' + str(crop_top_col) + ' ' + str(
    hsv_image[crop_top_row,crop_top_col]))

  cv2.namedWindow("Maze Window")
  cv2.imshow("Maze Window", im_with_keypoints)

  cv2.namedWindow("MaskImage Window")
  cv2.imshow("MaskImage Window", mask_image)

  cv2.namedWindow("Crop Window")
  cv2.imshow("Crop Window", crop_image)

  cv2.waitKey(2)

  return blob_image_center
```

B Python Script: lab3_image_exec.py

```python
#!/usr/bin/env python

import sys
import cv2
import copy
import time
import numpy as np
import math

import rospy
from std_msgs.msg import String
from sensor_msgs.msg import Image
from cv_bridge import CvBridge, CvBridgeError
from lab3_func import blob_search_init, blob_search


######################## Replace values below in Part2 of Lab3
    #########################

```

```python
# Params for camera calibration
theta = 0.0131571354728
beta = 760.56311235
tx = -0.31348615917
ty = -0.0705519480519


# ##########################################################################################


class ImageConverter:

    def __init__(self, SPIN_RATE):

        self.bridge = CvBridge()
        self.image_sub = rospy.Subscriber("/cv_camera_node/image_raw", Image, self.image_callback)
        self.coord_pub = rospy.Publisher("/coord_center", String, queue_size=10)
        self.loop_rate = rospy.Rate(SPIN_RATE)
        self.detector = blob_search_init()

        # Check if ROS is ready for operation
        while(rospy.is_shutdown()):
          print("ROS is shutdown!")

    def image_callback(self, data):
      global theta
      global beta
      global tx
      global ty

      try:
        raw_image = self.bridge.imgmsg_to_cv2(data, "bgr8")
      except CvBridgeError as e:
        print(e)

      cv_image = cv2.flip(raw_image, -1)
      cv2.line(cv_image, (0,50), (640,50), (0,0,0), 5)

      blob_image_center = blob_search(cv_image, self.detector)

      if(len(blob_image_center) == 0):
        print("No blob found!")
        self.coord_pub.publish("")
      elif(len(blob_image_center) == 1):
        x1 = int(blob_image_center[0][0])
        y1 = int(blob_image_center[0][1])
        # x2 = int(blob_image_center[1][0])
        # y2 = int(blob_image_center[1][1])

        # if x1 > x2:
        #   T = [x1,x2, y1, y2]
        #   x2 = T[0]
        #   x1 = T[1]
        #   y2 = T[2]
        #   y1 = T[3]

        A = np.array([[x1/beta], [y1/beta]])
        Rz = np.array([[math.cos(theta), -1*math.sin(theta)], [math.sin(theta), math.cos(theta)]])
```

```
        B = Rz.dot(A) + np.array([[tx], [ty]])

        xw = B[1,0]
        yw = B[0,0]

        xy_w = str(xw) + str(' ') + str(yw)
        print(xy_w)
        self.coord_pub.publish(xy_w)


def main():

    SPIN_RATE = 20 # 20Hz

    rospy.init_node('lab3ImageNode', anonymous=True)

    ic = ImageConverter(SPIN_RATE)

    try:
      rospy.spin()
    except KeyboardInterrupt:
      print("Shutting down!")

    cv2.destroyAllWindows()


if __name__ == '__main__':
    main()
```

C Python Script: lab3_image_tf_exec.py

```
#!/usr/bin/env python

import sys
import cv2
import copy
import time
import numpy as np
import math

import rospy
from std_msgs.msg import String
from sensor_msgs.msg import Image
from cv_bridge import CvBridge, CvBridgeError
from lab3_func import blob_search_init, blob_search


# Params for camera calibration
theta = 0
beta = 0
tx = 0
ty = 0

class ImageConverter:

    def __init__(self, SPIN_RATE):

        self.bridge = CvBridge()
        self.image_sub = rospy.Subscriber("/cv_camera_node/image_raw", Image, self.image_callback)
        self.loop_rate = rospy.Rate(SPIN_RATE)
```

```python
        self.detector = blob_search_init()

        # Check if ROS is ready for operation
        while(rospy.is_shutdown()):
          print("ROS is shutdown!")

    def image_callback(self, data):

        global theta
        global beta
        global tx
        global ty

        try:
            # Convert ROS image to OpenCV image
            raw_image = self.bridge.imgmsg_to_cv2(data, "bgr8")
        except CvBridgeError as e:
            print(e)

        # Flip the image 180 degrees
        cv_image = cv2.flip(raw_image, -1)

        # Draw a black line on the image
        cv2.line(cv_image, (0,50), (640,50), (0,0,0), 5)

        # cv_image is normal color image
        blob_image_center = blob_search(cv_image, self.detector)

        # Given world coordinate (xw, yw)
        xw = 0.2875
        yw = 0.1125

        # Only two blob center are found on the image
        if(len(blob_image_center) == 2):

            x1 = int(blob_image_center[0][0])
            y1 = int(blob_image_center[0][1])
            x2 = int(blob_image_center[1][0])
            y2 = int(blob_image_center[1][1])

            print("Blob Center 1: ({0}, {1}) and Blob Center 2: ({2}, {3})".format(x1, y1, x2, y2))

            ##################################### Your Code Start Here #####################################

            # Calculate beta, tx and ty, given x1, y1, x2, y2

            d = ( (x1 - x2)**2 + (y1 - y2)**2 )**0.5

            if x1 > x2:
                T = [x1,x2, y1, y2]
                x2 = T[0]
                x1 = T[1]
                y2 = T[2]
                y1 = T[3]

            beta = d/0.1  # Pixels per meter


            theta = math.asin((y2 - y1)/ d)

```

```
            Rz = np.array([[math.cos(theta), -1*math.sin(theta)], [math.sin(theta), math.cos(theta)]])


            # Calculate Tx, Ty

            A = Rz.dot(np.array([[x1/beta], [y1/beta]]))

            cam_origin_x = yw - A[0,0]
            cam_origin_y = xw - A[1,0]

            tx = cam_origin_x
            ty = cam_origin_y




            #################################### Your Code End Here ####################################

            print("theta = {0}\nbeta = {1}\ntx = {2}\nty = {3}\n".format(theta, beta, tx, ty))

        else:
            print("No Blob found! ")


def main():

    SPIN_RATE = 20 # 20Hz

    rospy.init_node('lab3ImageCalibrationNode', anonymous=True)

    ic = ImageConverter(SPIN_RATE)

    try:
        rospy.spin()
    except KeyboardInterrupt:
        print("Shutting down!")

    cv2.destroyAllWindows()


if __name__ == '__main__':
    main()
```

D Python Script: lab3_move_exec.py

```
#!/usr/bin/env python

import sys
import copy
import time
import numpy as np

# For ROS
import rospy
from std_msgs.msg import String
from ur3_driver.msg import command
from ur3_driver.msg import position
from helper import lab_invk
```

```python

# For Particle Filter
import turtle
import scipy as sp
import scipy.stats as st
import matplotlib.pyplot as plt
from getkey import getkey, keys


################################## ROS SETUP NO NEED TO MODIFY BELOW ##################################


SPIN_RATE = 20
PI = 3.1415926535
current_io_0 = False
current_position_set = False
thetas = [0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
current_position = [120*PI/180.0, -90*PI/180.0, 90*PI/180.0, -90*PI/180.0, -90*PI/180.0, 0*PI/180.0]


"""
Callback function for getting current robot's world coordinate
"""
def coord_callback(msg):

  global blob_center
  blob_center = msg.data


"""
Whenever ur3/position publishes info, this callback function is called.
"""
def position_callback(msg):

  global thetas
  global current_position
  global current_position_set

  thetas[0] = msg.position[0]
  thetas[1] = msg.position[1]
  thetas[2] = msg.position[2]
  thetas[3] = msg.position[3]
  thetas[4] = msg.position[4]
  thetas[5] = msg.position[5]

  current_position[0] = thetas[0]
  current_position[1] = thetas[1]
  current_position[2] = thetas[2]
  current_position[3] = thetas[3]
  current_position[4] = thetas[4]
  current_position[5] = thetas[5]
  current_position_set = True


"""
Move robot arm from one position to another
"""
def move_arm(pub_cmd, loop_rate, dest, vel, accel):

  global thetas
```

```
  global SPIN_RATE

  error = 0
  spin_count = 0
  at_goal = 0

  driver_msg = command()
  driver_msg.destination = dest
  driver_msg.v = vel
  driver_msg.a = accel
  driver_msg.io_0 = current_io_0
  pub_cmd.publish(driver_msg)

  loop_rate.sleep()

  while(at_goal == 0):

    if( abs(thetas[0]-driver_msg.destination[0]) < 0.0005 and \
      abs(thetas[1]-driver_msg.destination[1]) < 0.0005 and \
      abs(thetas[2]-driver_msg.destination[2]) < 0.0005 and \
      abs(thetas[3]-driver_msg.destination[3]) < 0.0005 and \
      abs(thetas[4]-driver_msg.destination[4]) < 0.0005 and \
      abs(thetas[5]-driver_msg.destination[5]) < 0.0005 ):

      at_goal = 1
      #rospy.loginfo("Goal is reached!")

    loop_rate.sleep()

    if(spin_count >  SPIN_RATE*5):
      pub_cmd.publish(driver_msg)
      rospy.loginfo("Just published again driver_msg")
      spin_count = 0

    spin_count = spin_count + 1

  return error


"""
Check if the robot move out of the boundry of the Maze
"""
def check_boundary(pos):

  if(pos[0]>=0.5):
    pos[0] = 0.5
  elif(pos[0]<=0.1):
    pos[0] = 0.1
  else:
    pass

  if(pos[1]>=0.5):
    pos[1] = 0.5
  elif(pos[1]<=-0.1):
    pos[1] = -0.1
  else:
    pass

  return pos


"""
```

```python
Detect which key has been pressed
"""
def which_key(get_key):

  key = get_key
  key_pressed = None
  if key == keys.UP:
    # print("UP key")
    key_pressed = "3"
  elif key == keys.DOWN:
    # print("DOWN key")
    key_pressed = "1"
  elif key == keys.RIGHT:
    # print("RIGHT key")
    key_pressed ="2"
  elif key == keys.LEFT:
    # print("LEFT key")
    key_pressed = "4"
  else:
    # Handle other text characters
    key_pressed = "Wrong key is pressed!"

  return key_pressed

################################## ROS SETUP NO NEED TO MODIFY ABOVE ##################################


timsSpan = 100
numberOfParticles = 5000

oldpos0 = 0 # x
oldpos1 = 0 # y

num_rows = 10
num_cols = 16

grid_height = 25
grid_width = 25

grid_height_in_m = 0.025
grid_width_in_m = 0.025

window_height = grid_height*num_rows  # 250
window_width = grid_width*num_cols    # 400

t_step = 0
canmove = 1

xCAM = 0
yCAM = 0

# store coordinate of orange blob
blob_center = None

# end-effector coordinate in world frame
startgridx = 0  # 0 to 15
startgridy = 0  # 0 to 9

gridzero_inRobotWorld_x = 0.3875
gridzero_inRobotWorld_y = -0.0405

```

```
# initialize the location of end-effector
new_pos = [gridzero_inRobotWorld_x, gridzero_inRobotWorld_y, 0.035, -45]

"""
class for Maze
"""
class Maze(object):

  def __init__(self,dimension=2, maze = None):
    '''
    maze: 2D numpy array.
    passages are coded as a 4-bit number, with a bit value taking
    0 if there is a wall and 1 if there is no wall.
    The 1s register corresponds with a square's top edge,
    2s register the right edge,
    4s register the bottom edge,
    and 8s register the left edge.
    (numpy array)
    '''
    self.dimension = dimension       # 2
    self.grid_height = grid_height   # 25
    self.grid_width = grid_width     # 25
    self.window = turtle.Screen()
    self.window.setup (width = window_width, height = window_height) # window_width
    =400, window_height=250

    if maze is not None:
      self.maze = maze
      self.num_rows = maze.shape[0]
      self.num_cols = maze.shape[1]
      self.fix_maze_boundary()
      self.fix_wall_inconsistency()
    else:
      assert num_rows is not None and num_cols is not None, 'Parameters for maze
    should not be None.'
      self.create_maze(num_rows = num_rows, num_cols = num_cols)

    self.height = self.num_rows * self.grid_height  # 250
    self.width = self.num_cols * self.grid_width    # 400

    self.turtle_registration()

  def turtle_registration(self):
    turtle.register_shape('tri', ((-3, -2), (0, 3), (3, -2), (0, 0)))

  def check_wall_inconsistency(self):
    wall_errors = list()
    # Check vertical walls
    for i in range(self.num_rows):
      for j in range(self.num_cols-1):
        if (self.maze[i,j] & 2 != 0) != (self.maze[i,j+1] & 8 != 0):
          wall_errors.append(((i,j), 'v'))
    # Check horizonal walls
    for i in range(self.num_rows-1):
      for j in range(self.num_cols):
        if (self.maze[i,j] & 4 != 0) != (self.maze[i+1,j] & 1 != 0):
          wall_errors.append(((i,j), 'h'))
    return wall_errors

  def fix_wall_inconsistency(self, verbose = True):
    # Whenever there is a wall inconsistency, put a wall there.
    wall_errors = self.check_wall_inconsistency()
```

```
    if wall_errors and verbose:
      print('Warning: maze contains wall inconsistency.')
    for (i,j), error in wall_errors:
      if error == 'v':
        self.maze[i,j] |= 2
        self.maze[i,j+1] |= 8
      elif error == 'h':
        self.maze[i,j] |= 4
        self.maze[i+1,j] |= 1
      else:
        raise Exception('Unknown type of wall inconsistency.')
    return

  def fix_maze_boundary(self):
    # Make sure that the maze is bounded.
    for i in range(self.num_rows):
      self.maze[i,0] |= 8
      self.maze[i,-1] |= 2
    for j in range(self.num_cols):
      self.maze[0,j] |= 1
      self.maze[-1,j] |= 4

  def create_maze(self, num_rows, num_cols):

    self.num_rows = num_rows
    self.num_cols = num_cols

    self.maze = np.zeros((num_rows, num_cols), dtype = np.int8)
    self.maze[6,1] = 15
    self.maze[5,1] = 15
    self.maze[4,1] = 15
    self.maze[3,1] = 15
    self.maze[0,2] = 15
    self.maze[8,3] = 15
    self.maze[1,3] = 15
    self.maze[8,4] = 15
    self.maze[2,4] = 15
    self.maze[1,4] = 15
    self.maze[7,6] = 15
    self.maze[5,6] = 15
    self.maze[4,6] = 15
    self.maze[3,6] = 15
    self.maze[6,8] = 15
    self.maze[2,8] = 15
    self.maze[1,8] = 15
    self.maze[7,9] = 15
    self.maze[3,9] = 15
    self.maze[6,10] = 15
    self.maze[5,10] = 15
    self.maze[4,10] = 15
    self.maze[1,10] = 15
    self.maze[9,11] = 15
    self.maze[5,11] = 15
    self.maze[2,11] = 15
    self.maze[3,12] = 15
    self.maze[7,13] = 15
    self.maze[5,14] = 15
    self.maze[6,15] = 15

    self.fix_maze_boundary()
    self.fix_wall_inconsistency(verbose = False)

```

```
  def permissibilities(self, cell):
    '''
    Check if the directions of a given cell are permissible.
    Return: (down, right, up, left)
    '''
    cell_value = self.maze[cell[0], cell[1]] #(row number, col number)
    return (cell_value & 1 == 0, cell_value & 2 == 0, cell_value & 4 == 0, cell_value & 8 == 0)

  def distance_to_walls(self, coordinates):
    '''
    Measure the distance of coordinates to nearest walls at four directions.
    Return: (up, right, down, left)
    '''
    x, y = coordinates

    i = int(y // self.grid_height)
    j = int(x // self.grid_width)
    d1 = y - y // self.grid_height * self.grid_height
    while self.permissibilities(cell = (i,j))[0]:
      i -= 1
      d1 += self.grid_height

    i = int(y // self.grid_height)
    j = int(x // self.grid_width)
    d2 = self.grid_width - (x - x // self.grid_width * self.grid_width)
    while self.permissibilities(cell = (i,j))[1]:
      j += 1
      d2 += self.grid_width

    i = int(y // self.grid_height)
    j = int(x // self.grid_width)
    d3 = self.grid_height - (y - y // self.grid_height * self.grid_height)
    while self.permissibilities(cell = (i,j))[2]:
      i += 1
      d3 += self.grid_height

    i = int(y // self.grid_height)
    j = int(x // self.grid_width)
    d4 = x - x // self.grid_width * self.grid_width
    while self.permissibilities(cell = (i,j))[3]:
      j -= 1
      d4 += self.grid_width

    return [d1, d2, d3, d4]

  def show_maze(self):

    turtle.setworldcoordinates(0, 0, self.width * 1.005, self.height * 1.005)
    wally = turtle.Turtle()
    wally.speed(0)
    wally.width(1.5)
    wally.hideturtle()
    turtle.tracer(0, 0)

    for i in range(self.num_rows):
      for j in range(self.num_cols):
        permissibilities = self.permissibilities(cell = (i,j))
        turtle.up()
        wally.setposition((j * self.grid_width, i * self.grid_height))
        # Set turtle heading orientation
        # 0 - east, 90 - north, 180 - west, 270 - south
```

```
        wally.setheading(0)
        if not permissibilities[0]:
          wally.down()
        else:
          wally.up()
        wally.forward(self.grid_width)
        wally.setheading(90)
        wally.up()
        if not permissibilities[1]:
          wally.down()
        else:
          wally.up()
        wally.forward(self.grid_height)
        wally.setheading(180)
        wally.up()
        if not permissibilities[2]:
          wally.down()
        else:
          wally.up()
        wally.forward(self.grid_width)
        wally.setheading(270)
        wally.up()
        if not permissibilities[3]:
          wally.down()
        else:
          wally.up()
        wally.forward(self.grid_height)
        wally.up()
    turtle.update()

  def show_valid_particles(self, particles, show_frequency = 1):
    turtle.shape('tri')
    for i, particle in enumerate(particles):
      if i % show_frequency == 0:
        turtle.setposition((particle[1], particle[0]))
        turtle.setheading(90)
        turtle.color('blue')
        turtle.stamp()
    turtle.update()

  def show_estimated_location(self, estimate):
    y_estimate, x_estimate, heading_estimate= estimate[0], estimate[1], 0
    turtle.color('orange')
    turtle.setposition((x_estimate, y_estimate))
    turtle.setheading(90 - heading_estimate)
    turtle.shape('turtle')
    turtle.stamp()
    turtle.update()

  def clear_objects(self):
    turtle.clearstamps()

  def show_robot_position(self, robotX, robotY, robotHeading=0):
    turtle.color('green')
    turtle.shape('turtle')
    turtle.shapesize(0.7, 0.7)
    turtle.setposition((robotX, robotY))
    turtle.setheading(90 - robotHeading)
    turtle.stamp()
    turtle.update()

  def finish(self):
```

```python
    turtle.done()
    turtle.exitonclick()

"""
Defulat 2-D model for roomba robot in a room
"""
class default_2D_Model:

  def __init__(self):

    self.height = num_rows * grid_height
    self.width = num_cols * grid_width

    self.grid_height = grid_height
    self.grid_width = grid_width

    self.num_rows = num_rows
    self.num_cols = num_cols

    self.y = 12 + grid_height*startgridy
    self.x = 12 + grid_width*startgridx

    self.map = Maze()

    self.accuracy = 15
    self.motionNoise = 20

    self.max = [num_rows*grid_height, num_cols*grid_width]
    self.min = [0, 0]
    self.map.show_maze()


  """
  input: postion [x, y]
  return the map reading, which are the distances to the closest wall on four
  directions at this position [d1, d2, d3, d4]
  """
  def readingMap(self, position):
    validPosition = [0,0]
    for i in range(2):
      validPosition[i] = max(int(position[i]), self.min[i])
      validPosition[i] = min(int(position[i]), self.max[i]-1)

    reading = self.map.distance_to_walls((validPosition[1], validPosition[0]))
    return reading


  """
  return the sensor reading, which are the distances to the closest
  walls on four directions [d1, d2, d3, d4]
  """
  def readingSensor(self,x_camera,y_camera):

    global grid_width_in_m
    global grid_height_in_m

    global grid_height
    global grid_width

    xpix = x_camera*(grid_width/grid_width_in_m) + 12
    ypix = y_camera*(grid_height/grid_height_in_m) + 12

```

```
    reading = self.map.distance_to_walls((xpix, ypix))

    return reading


  """
  input: the position of the previous particle [x',y'], (optional) the control
  signal integer currentControl
  return: if the robot is at the position of the previous particle, the current
  robot position [x,y]
  Control command:0 halt, 1 down, 2 right, 3 up, 4 left
  """
  def simulateNextPosition(self, previousEstimate, currentControl=0):

    for i in range(2):
      previousEstimate[i] = max(previousEstimate[i], self.min[i])
      previousEstimate[i] = min(previousEstimate[i], self.max[i])
    x, y = previousEstimate[0], previousEstimate[1]
    cellX, cellY = int(x // self.grid_width), int(y // self.grid_height)

    if cellX > 15:
      cellX = 15

    if cellY > 9:
      cellY = 9

    permissibilities = self.map.permissibilities((cellY, cellX))  #(down, right, up, left)

    if (currentControl == 3 and permissibilities[2]):
      y += self.grid_height
    elif (currentControl == 1 and permissibilities[0]):
      y -= self.grid_height
    elif (currentControl == 2 and permissibilities[1]):
      x += self.grid_width
    elif (currentControl == 4 and permissibilities[3]):
      x -= self.grid_width

    x += np.random.normal(0, self.motionNoise)
    y += np.random.normal(0, self.motionNoise)
    nextEstimate = np.array([x, y])

    for i in range(2):
      nextEstimate[i] = max(self.min[i], nextEstimate[i])
      nextEstimate[i] = min(self.max[i], nextEstimate[i])

    return nextEstimate


  def run(self, currentControl=0):

    """
        Input: Control command: 0 halt, 1 down, 2 right, 3 up, 4 left
        Can only move from the center of one cell to the center of one of four neighboring cells
        """

    global canmove
    global blob_center

    cellX, cellY = int(self.x // self.grid_width), int(self.y // self.grid_height)
    permissibilities = self.map.permissibilities((cellY, cellX))
```

```python

    if (currentControl == 3 and permissibilities[2]):
      self.y += self.grid_height
      canmove = 1
    elif (currentControl == 1 and permissibilities[0]):
      self.y -= self.grid_height
      canmove = 1
    elif (currentControl == 2 and permissibilities[1]):
      self.x += self.grid_width
      canmove = 1
    elif (currentControl == 4 and permissibilities[3]):
      self.x -= self.grid_width
      canmove = 1
    else:
      canmove = 0

    self.map.show_robot_position(self.x, self.y, 0)


  def plotParticles(self, particles):
    """
        Input is 2D python list containing position of all particles: [[x1,y1], [x2,y2], ...]
        """
    self.map.show_valid_particles(particles)


  def plotEstimation(self, estimatePosition):
    """
        Input is the estimated position: [x, y]
        """
    self.map.show_estimated_location(estimatePosition)


  def readMax(self):
    """
        Return the max value at each dimension [maxX, maxY, ...]
        """
    return self.max


  def readMin(self):
    """
        Return the min value at each dimension [minX, minY, ...]
        """
    return self.min


  def readPosition(self):
    """
        Return actual position, can be used for debug
        """
    return (self.x, self.y)



class particleFilter:

  def __init__(self, dimension = 2, model = default_2D_Model(), numParticles = numberOfParticles, timeSpan = timsSpan, resamplingNoise = 0.01, positionStd = 5):

    self.model = model
```

```
    self.numParticles = numParticles
    self.dimension = dimension
    self.timeSpan = timeSpan

    self.std = positionStd
    self.curMax = self.model.readMax()
    self.curMin = self.model.readMin()
    self.resNoise = [x*resamplingNoise for x in self.curMax]

        ############# The initial particles are uniformely distributed #############
        ## TODO: self.particles = ? self.weights = ?

        # Generate uniformly distributed variables in x and y direction within [0, 1]
        # Hint: np.random.uniform(0, 1, ...)

        # Spread these generated particles on the maze
        # Hint: use self.curMax, remember X direction: self.curMax[0], Y direction: self.curMax[1]
        # particles should be something like [[x1,y1], [x2,y2], ...]

        # Generate weight, initially all the weights for particle should be equal, namely 1/num_of_particles
        # weights should be something like [1/num_of_particles, 1/num_of_particles, 1/num_of_particles, ...]

        ###########################################################################
        # Student finish the code below
          self.particles = np.random.uniform(0, 1, (self.numParticles, 2))
          self.particles = np.multiply(self.particles, self.curMax)
          self.weights = np.ones(self.numParticles)
          self.weights *= (1 / self.numParticles)


    ###########################################################################


  def Sample_Motion_Model(self, u_t=0):

        ####### Sample the Motion Model to Propagate the Particles ########
        ## TODO: self.particles = ?

        # For each particle in self.particles [[x1,y1], [x2,y2], ...], get the nextEstimate
        # Hint: use self.model.simulateNextPosition(?, u_t)
        # Update self.particles

        ###########################################################################
          for i in range(self.numParticles):
            self.particles[i] = self.model.simulateNextPosition(self.particles[i], u_t)
    ###########################################################################


  def Measurement_Model(self, x, y):

    #################### Measurement Motion Model ####################
        ## TODO: update self.weights, normalized

        # Get the sensor measurements for robot's position
        # Hint: use self.model.readingSensor(x_camera,y_camera)

        # For each particle in self.particles [[x1,y1], [x2,y2], ...], get the its
```

```
    position
        # Hint: use self.model.readingMap(position)

        # Calculate distance between robot's postion and each particle's position
        # Calculate weight for each particle, w_t = exp(-distance**2/(2*self.std))

        # Collect all the particles' weights in a list
        # For all the weights of particles, normalized them
        # Hint: pay attention to the case that sum(weights)=0, avoid round-off to zero

        # Update self.weights

        ###############################################################################
          position = self.model.readingSensor(x, y)

          weight_particle = np.zeros((self.numParticles, 1))

          for i in range(self.numParticles):
            distance = np.array(self.model.readingMap(np.array(self.particles[i])))
            rel_distance =  np.linalg.norm(distance - position)

            weight = np.exp(-1*(rel_distance**2) / (2*self.std))
            weight_particle[i] = weight

            sum_weights = sum(weight_particle)

          if sum_weights == 0:
            print("Sum of weights is 0")
            self.weights = np.ones(self.numParticles) * float (1 / self.numParticles)
          else:
            self.weights = weight_particle / sum_weights

        ###############################################################################



  def calcPosition(self):

    ############ Calculate the position update estimate ###############
        ## TODO: return a list with two elements [x,y],  estimatePosition

        # For all the particles in direction x and y, get one estimated x, and one estimated y
        # Hint: use the normalized weights, self.weights, estimated x, y can not be out of the
        # boundary, use self.curMin, self.curMax to check

        ###############################################################################
        # Student finish teh code below

          closest_elements = [0, 0]

          weights = sum(np.multiply(self.particles, self.weights))

          x = max(min(weights[0], self.curMax[0]), self.curMin[0])
          y = max(min(weights[1], self.curMax[1]), self.curMin[1])
          closest_elements = [x, y]

          return closest_elements


        ###############################################################################
```

```python



  def resampling(self):

    newParticles = []

    N = len(self.particles)

    cumulative_sum = np.cumsum(self.weights)
    cumulative_sum[-1] = 1. # avoid round-off error

        # Resample according to indexes
        # The probability to be selected is related to weight
    for i in range(N):
      randomProb = np.random.uniform()
      index = np.searchsorted(cumulative_sum, randomProb)
      newParticles.append(self.particles[index])

    self.particles = newParticles


#     # Method 2: Roulette Wheel
#     def resampling(self):
#         newParticles = []
#         N = len(self.particles)
#         index = int(np.random.random() * N)
#         beta = 0
#         mw = np.max(self.weights)
#         for i in range(N):
#             beta += np.random.random() * 2.0 * mw
#             while beta > self.weights[index]:
#                 beta -= self.weights[index]
#                 index = (index + 1) % N
#             newParticles.append(self.particles[index])
#         self.particles = newParticles

#################################### NO NEED TO MODIFY BELOW ####################################

def blob_center_trans(blob_center_str):
  """
  input: blob_center_str
  """
  global gridzero_inRobotWorld_x
  global gridzero_inRobotWorld_y

  if(len(blob_center_str) == 0):
    x = -1
    y = -1
  else:
    xy_list = blob_center_str.split()
    y = gridzero_inRobotWorld_x - float(xy_list[0])
    x = float(xy_list[1]) - gridzero_inRobotWorld_y

  return (x, y)

def looprun(partfilt,step_sz,cmd,rate):

  global new_pos

```

```
  global oldpos0
  global oldpos1

  global t_step
  global blob_center

  global xCAM
  global yCAM


  #Control command:0 halt, 1 down, 2 right, 3 up, 4 left
  control = 0

  # If not the initial round, generate new samples
  if (t_step > 0):
    partfilt.Sample_Motion_Model(control)
  else:
    time.sleep(0.2)
    (xCAM, yCAM) = blob_center_trans(blob_center)


  # Assign weights to each particles
  partfilt.Measurement_Model(xCAM,yCAM)

  # Estimate current position
  estimatePosition = partfilt.calcPosition()

  print('Estimated Position: ' + str(estimatePosition))

  # Resample the particles
  partfilt.resampling()

  # Plot particles
  partfilt.model.plotParticles(partfilt.particles)

  # Plot estimated position
  partfilt.model.plotEstimation(estimatePosition)

  print('Use the arrow keys to command the robot left, right, forward and backward')
  if (t_step > 0):
    key_result = which_key(getkey())
    if(len(key_result) == 1):
      if(int(key_result) == 3):
        control = 3
        oldpos0 = new_pos[0]
        oldpos1 = new_pos[1]
        new_pos[0] -= step_sz
      elif(int(key_result) == 1):
        control = 1
        oldpos0 = new_pos[0]
        oldpos1 = new_pos[1]
        new_pos[0] += step_sz
      elif(int(key_result) == 2):
        control = 2
        oldpos0 = new_pos[0]
        oldpos1 = new_pos[1]
        new_pos[1] += step_sz
      elif(int(key_result) == 4):
        control = 4
        oldpos0 = new_pos[0]
        oldpos1 = new_pos[1]
        new_pos[1] -= step_sz
```

```
    else:
      control = 0
      print(key_result)


  partfilt.model.map.clear_objects()

  partfilt.model.run(control)

  print('Actual Postions: ' + str(partfilt.model.readPosition))

  if canmove == 1:
    new_pos = check_boundary(new_pos)
    # print(new_pos)
    new_dest = lab_invk(new_pos[0], new_pos[1], new_pos[2], new_pos[3])
    # print(new_dest)
    move_arm(cmd, rate, new_dest, 4, 4)
    rospy.loginfo("Destination reached!")
  else:
    new_pos[0] = oldpos0  # x
    new_pos[1] = oldpos1  # y

  t_step = t_step + 1

  print('\n\nSleeping ...')

  time.sleep(0.5)

  (xCAM, yCAM) = blob_center_trans(blob_center)

  print('\n\n')


"""
Program run from here
"""
def main():

  global SPIN_RATE
  global new_pos
  global oldpos0
  global oldpos1

  pf = particleFilter(2)

  # Initialize ROS node
  rospy.init_node('lab3MoveNode')

  # Initialize publisher for ur3/command with buffer size of 10
  pub_command = rospy.Publisher('ur3/command', command, queue_size=10)
  sub_position = rospy.Subscriber('ur3/position', position, position_callback)
  sub_coord = rospy.Subscriber('/coord_center', String, coord_callback)

  # Check if ROS is ready for operation
  while(rospy.is_shutdown()):
    print("ROS is shutdown!")

  # Initialize the rate to publish to ur3/command
  loop_rate = rospy.Rate(SPIN_RATE)

  home_init = lab_invk(new_pos[0],new_pos[1],new_pos[2],new_pos[3])

```

```
  rospy.loginfo("Moving robot ...\n")
  move_arm(pub_command, loop_rate, home_init, 4, 4)
  rospy.loginfo("Home initialization finished!\n")
  time.sleep(1)
  rospy.loginfo("Press direction keys to move the robot!")

  step_size = 0.025

  oldpos0 = new_pos[0]
  oldpos1 = new_pos[1]

  while not rospy.is_shutdown():
    looprun(pf, step_size, pub_command, loop_rate)


if __name__ == '__main__':

  try:
    main()
    # When Ctrl+C is executed, it catches the exception
  except rospy.ROSInterruptException:
    pass
```