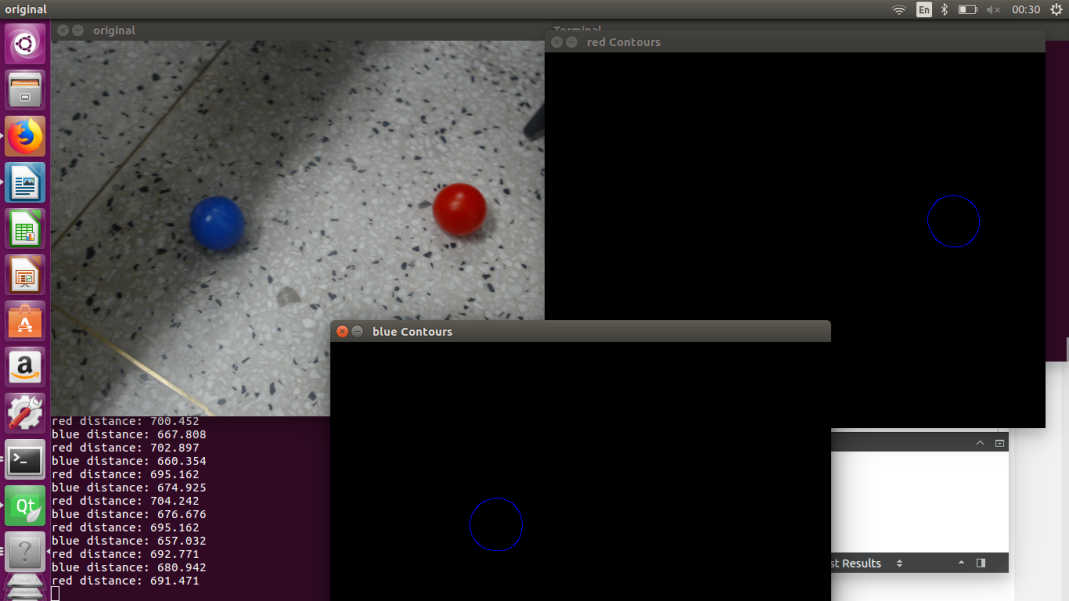
Progress Report 1 (3/19 – 4/1)

Simeneh S. Gulelat 20140931

In the OpenCV section, we apply a methodology of detection and localization which is used to find the position and orientation of the camera (i.e. robot) by analyzing the associated images. More specifically, our goal is to extract the location of all the balls and construct a 6-DOF vector of the robot’s position and orientation from the video feed coming from the camera.

To achieve this goal, we need camera images as input as well as the intrinsic camera calibration parameters of the web-cam such as the focal lengths, tangential distortion parameters and the position of the principal point (i.e. origin of camera coordinates) and extrinsic parameters such as transformation matrix between the world coordinates and camera coordinates. As output, we extract the differences in rotation matrix and translation vector between consecutive frames

Color identification of the object is possible by thresholding the color range of the blue and red balls. We do this by using OpenCV to convert the RGB values of the image to HSV and selecting the Hue range for Blue and Range. After that, we applied Canny edge-detection to isolate the edges of the object and search for contours in the image corresponding to the balls. Then we are able to obtain the length of the ball contours and calculate the pixel-wise diameter of the balls. Finally, using simple perspective geometry, and knowing the size of the ball, we get the distance between the ball and camera.

  
Fig1. Demo of the Distance Calculation and Edge Detection Output

As mentioned above, we use Visual Odometry to do tracking. In order to map the trajectory of the camera using video feed, we first identify features in the scene and use a tracking algorithm called KLT tracker to locate the corresponding pixel coordinates of various objects (at least) in the scene between image at (t) and image at (t-1). This correspondence is expressed by the essential matrix between Image at ‘t’ and image at ‘t-1’.The distance measurement we made in object detection will be used to set the scale of the scene.

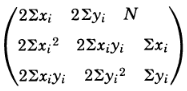
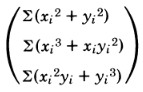
An example of how a KLT tracker should work is detailed in the following pictures where 4 points are tracked in a i) extract feature points ii) generate correspondences between N=4 points in F=5 frames. iii) final tracking output appears as translation vector field.

Due to the instability of the hough circle transform, we decided to create a way of detecting the size and location of the red and blue circles from the output of the find contour attribute. Since, the location of the location of all the edge points have been selected by using the Canny algorithm and classified using “findContour()”, it is possible to apply a linear regression technique to find a circle fit for the edge points. The output is detailed below.

Problem: Since the Hough transform is very unstable and requires a lot of fine tuning of several parameters that are not really relevant to our problem, it is important to have a better method that draws circles around the balls and can be controlled via distance, error percentage and number of sampling points.

Our Solution: Use a circle fitting method that uses least squares method with error defined by :

subjected to the constraints: 

We solved:

=

Using the above method, it was possible to:

1) Detect and localize each of the blue and red balls separately regardless of their distance or proximity. This allows us to use a prioritizing algorithm that primarily targets the closest balls during path planning.

2) Find the distance to each ball and prioritize the balls based on proximity: Depending on the relative location of all the blue and red balls, we can generate an optimal path for the robot to follow.

Progress Report 2 (4/2 - 4/15)

Simeneh S. Gulelat 20140931

Regarding the result of the previous circle tracking algorithm we implemented, the error reduction applied proved to be inconsistent. Therefore, we tried to improve the initial method that uses Hough transform.

Initially we solved the least squares method to minimize

Afterwards, we included a condition in the code to filter all objects of which outline deviates from the fitted circle by some margin, say u. However, the error u appears to fluctuate a lot. So we resorted to a different method of using the internal OpenCV library ball tracker.

A code for tracking the blue and red balls was provided by the TA, which we used as a template and modified to identify the orange basket as well. Using the parameter setting windows, we can adjust the tracking parameters. After thresholding, the Canny Edge Detector was applied and the ball edge identified and tracked using the Hough transform.

Moreover, instead of Visual Odometry, which we planned to do in the beginning using a KLT tracker, we have changed our strategy of integrating with the ROS system by simply using the readings of the blue and red ball positions from the web-cam as input for calculating the trajectory cost function.

Once the camera is mounted, there will be three stages of navigation: First, is to localize the objects i.e. robot position, target position (blue balls) and obstacle position (red balls) using the OpenCV code. Then the robot trajectory planning is deployed by minimizing a certain criterion. The final step is to convert the path coordinates in image plane to coordinates in the real world.

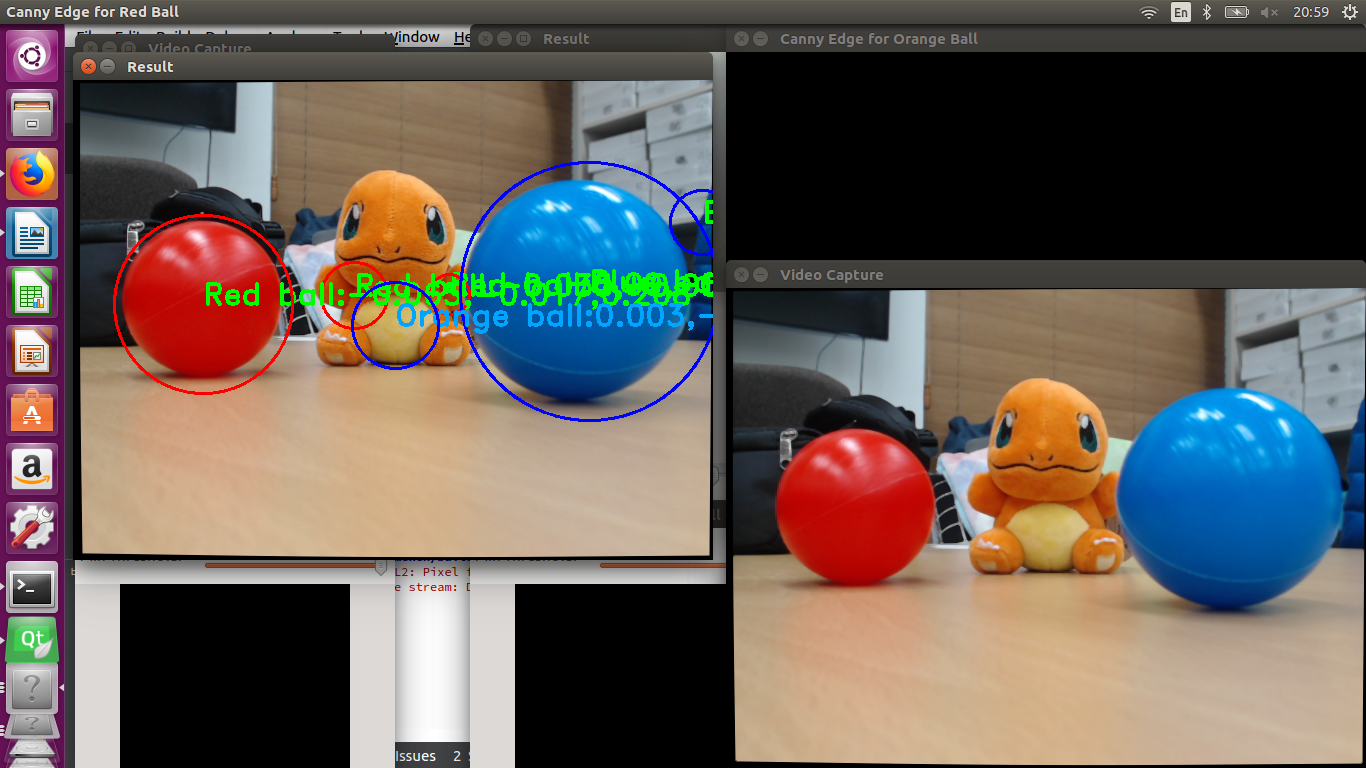


Fig 2. The modified ball tracker used to identify the the red and blue balls together with the distances, as well as an orange object

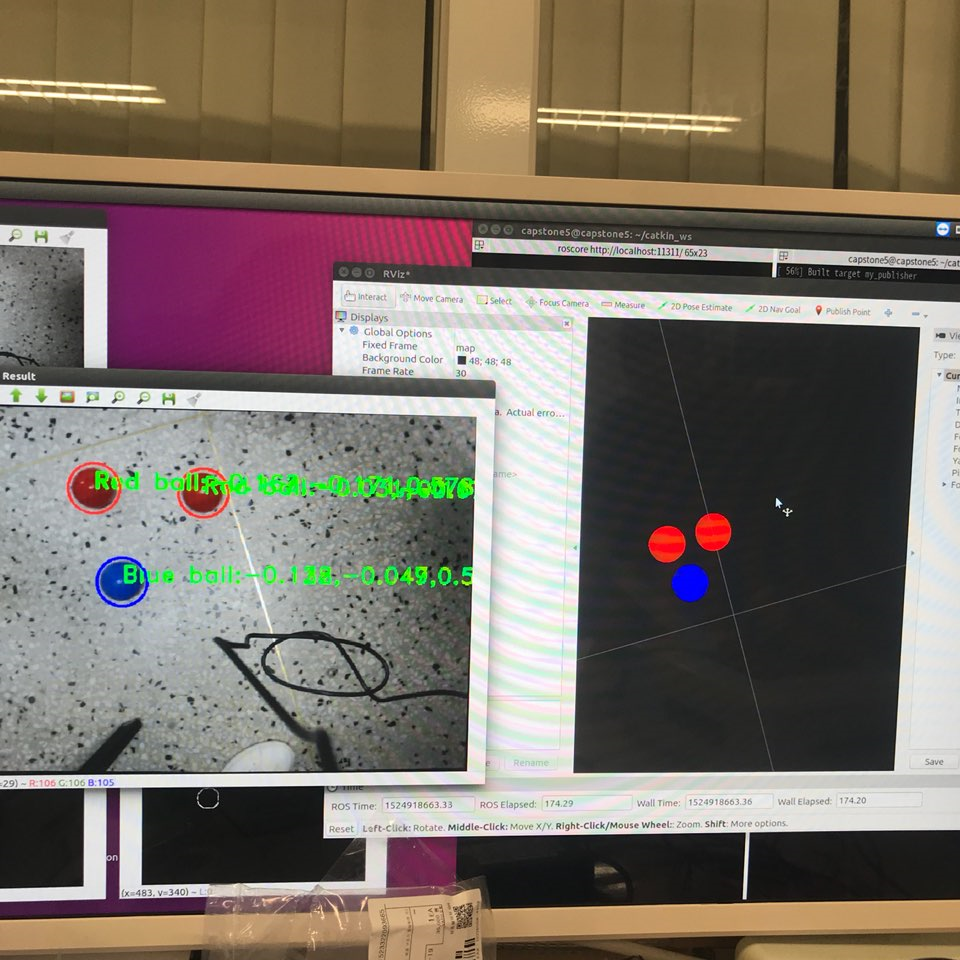
In addition, since we are planning to use two cameras, one for navigating and another for tuning and adjusting the pick-up mechanism. However, the problem we are now facing is that our ball tracking code is accurate beyond a distance of 10 cm from the camera. That is because the edges of the ball go outside the field of view of the camera and the Canny Edge detector is not able to identify the correct outline of the ball. Another reason is the reflectivity of the balls which lead the detection algorithm to (incorrectly) identify the reflections of light source as valid outlines. A possible solution for this problem is to increase the filtering degree on the image before measuring the HSV values.

Progress Report 3 (4/16– 4/29)

Simeneh S. Gulelat 20140931

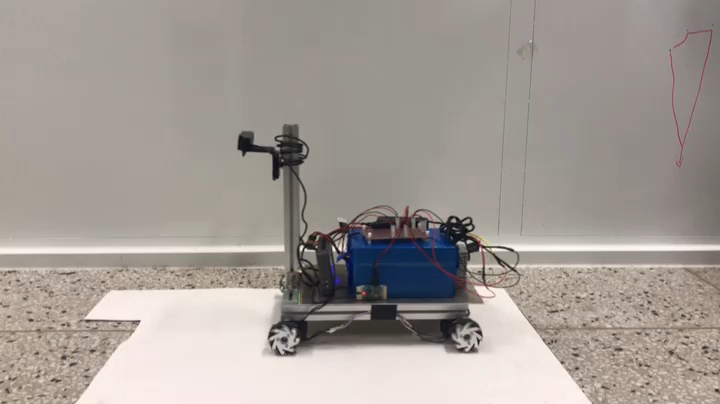
Integration of OpenCV with ROS

Following the development of the OpenCV code for detecting and localizing the blue and red circles, we integrated the output results from the OpenCV i.e. the position of each ball in camera coordinates to the node of the ROS. During the first integration, the object detection code produced duplicated objects from the camera input. For example, if two red balls were placed in the field of view of the camera, the code showed that fours balls were detected, two of each having circles very close to one another.

  
Fig 1. Integration with ROS using one blue and two red balls

After successive trials of tuning the blurring parameters of the ‘*GaussianBlur* ’ function, as well as the minimum detection pixel radius, we found out that the error was caused by the ‘findContour()’ function. ‘findContour()’ identifies two contours (inner and outer) for every object that is found by the canny edge detector. Since the Canny detector generates thick boundaries for the edges detected, we are able to find two contours for each edge, when we should only find one. The problem was solved, albeit temporarily, by applying the ‘findContours()’ function on the initially threshold image instead of the canny edge detector output. Identifying contours in the color threshold images allowed us to have only one contour for every object.

Moreover, we recalibrated the camera to obtain the intrinsic and extrinsic parameters. Since the camera is placed on the robot, the rotation matrix and translation vectors changed. The distortion vector also has new values.



Camera Location used

Calibration matrix:

Distortion vector:

Using the code for ball tracking, we converted the pixel coordinates of the circle centers into the camera coordinates using the following transformation:

u = (x-intrinsic\_data[2])/intrinsic\_data[0]; v = (y-intrinsic\_data[5])/intrinsic\_data[4]; //in pixel coordinates where (x, y) is the circle center in pixel coordinates  
Zc = (intrinsic\_data[0]\*fball\_radius)/(2\*(float)radius) ; //distance from the camera   
Xc = u\*Zc ; Yc = v\*Zc ;  
Xc = roundf(Xc \* 1000) / 1000;  
Yc = roundf(Yc \* 1000) / 1000;  
Zc = roundf(Zc \* 1000) / 1000;

Progress Report 4 (4/30 – 5/13)

Simeneh S. Gulelat 20140931

We have included new changes to the ball tracking code provided by the instructor. Since we chose to remove the canny edge detector due to duplicated edges, we faced a new challenge of properly estimating the 3D positions of the detected red and blue balls. Therefore, in the existing code, we added conditions that manually fix the radius of the balls depending on how far away they are. However, a problem rises that the minimum threshold ball pixel size “iMin\_tracking\_ball\_size” is set to a small value to avoid detecting reflections on the ball. Since OpenCV cannot differentiate between a small object placed near or a large object placed at a distance, we cannot know if what is detected is just a reflection or a ball placed far away. So we included a condition that corrects the radius only when the size is above the pixel size threshold.

Also, by calibrating the erosion and dilation operations, we were able to control the amount of noise correction we obtained from the reflections on the ball. For example, using a dilation structuring element that is bigger than the erosion structuring element works differently than if the converse was applied. For example to remove the reflections from the ball, we applied erosion and dilation operations using kernel sizes of 6x6 and 12x12 respectively. However, we faced some drawbacks in completely removing the inner circles so we tried to use a new method that identifies which contours are real balls and which ones are reflections. Examples of morphological operations (erosion and dilation) on thresholded images are shown below where a kernel size of 5x5 was used to remove pixels at the boundary and then add more pixels later.

    
a) Original image b) Eroded image c) Dilated image

Next we applied a different approach to eliminate the ball reflections created during thresholding. The method involves measuring the distances between the centers of any two pairs of contours of the same color and comparing its magnitude with the radius of the first ball. If the distance is below the radius of the circle, then the second circle must be a reflection inside the first circle and hence must be removed. If not, then we keep it. This method can also be applied to resolve the duplicated detection of edges created by the canny edge detector.

In this case, if the distance between two circles of similar color is found to be below some arbitrary error value, then the two circles must be duplicates of the same edge and the second circle can be removed. Since , the first circle is inside the main circle i, so it must be a reflection. Also since, , the second circle must be another ball. The code for correcting the number of detected red circles is given below with some annotations. After including the ball duplication code, we realized that it also helps remove the duplicates created by the canny edge detector.

//remove reflections

vector<Point2f> temp\_center\_r = center\_r;

vector<Point2f> temp\_center\_b = center\_b;

vector<float> temp\_radius\_r = radius\_r;

vector<float> temp\_radius\_b = radius\_b;

//red

size\_t i = 0;

while (i < temp\_radius\_r.size()){ //when there’s some contours in the field of view

bool something = true; //assign dummy Boolean

for (size\_t j = 0; j < temp\_radius\_r.size() ; j++){

if (i!=j&&(norm(temp\_center\_r[i] - temp\_center\_r[j]) < temp\_radius\_r[j])) {

//when the distance between the center of the circles is below the

radius of the outer circle, remove the center and radius from the

vectors

temp\_center\_r.erase(temp\_center\_r.begin()+i); //remove ith element from vector

temp\_radius\_r.erase(temp\_radius\_r.begin()+i);

something = false;

break;

}

}

if(something){

i++;

}

}

Progress Report 5 (5/14 – 5/27)

Simeneh S. Gulelat 20140931

During the final two weeks of preparation, we carried out several demo trials to recalibrate the working conditions of the vehicle inside the demo room environment. The parameters that needed retuning included:

i) The view angle of the top camera to make sure the entire region is within the camera field view. Using trial and error, we found that 70 degrees for the first hinge and 90 degrees for the second gives the maximum view angle for the camera at the farthest i.e. 5 meters.



90o

70o

ii) Lighting conditions inside the demo room, since it affects the reflectivity of the balls, and

iii) Regarding the visual system, we were having problems with the fitting algorithm we used for more precision. For balls located beyond a certain distance, our fitting method resizes the radius of the balls. Therefore we update it to have a linear correction instead of constant correction.

We can incrementally correct fitting circle radius for decreasing distance or increasing pixel radius. To implement this method, we first created a correction table based on rough estimates of effective radius reduction by using trials at different positions of the ball.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Ball position  (m) | 5 | 4 | 3 | 2 | 1 |
| Fraction of maximum pixel correction | 1/5 | 2/5 | 3/5 | 4/5 | 5/5 |
| Optimal pixel correction | 0.2\*10 | 0.4\*10 | 0.6\*10 | 0.8\*10 | 1\*10 |

Since radius = K/ distance, for some constant K, to implement the above method, we expressed the method mathematically as:

pixel\_correction = which simplifies into,

pixel\_correction = 12 – 2\*dist = 12 – 2\* (K/radius)

Since, K = 23.54 based on the instrinsic camera parameters and the real size of the ball, the correction term will be

pixel\_correction = 12 – (47.07788/radius)

Therefore, in the OpenCV code, we convert

if (radius[i] > i\_Min\_detecting\_radius + 9){

radius[i] = radius[i] - 9 ;

}

into,

#include <math.h>

if (radius[i] > i\_Min\_detecting\_radius){

float pixel\_correction = 12 - (47.0788/radius[i]);

radius[i] = radius[i] - roundf(pixel\_correction);

}

Moreover, we improved the location of the lower camera so that red balls, which should be ignored during the picking up process, can be allowed to slip under the vehicle. In this case, there will be no need to avoid the balls themselves, but we will simply target the blue balls and collect them, all the while ignoring the red balls.

Finally, regarding the dropping phase, we tried to create a simple technique where we detect the positions of the two green balls and then average their x-coordinate values to obtain the center coordinates of the basket. Then, the ROS subsystem takes the coordinate information and enters a dropping state by approaching the basket perpendicularly and then moving sideways to drop the balls as the wire is made loose.

Progress Report 6 (5/28 -6/3)

Simeneh S. Gulelat 20140931

In the concluding stages, we finalized the overall structure of Namsaeng-2 and fully integrated the heat management and vibration reduction system. The heat management system is integrated with heat sensors that allow the sensors to detect high temperature rises (eg. above 30 degrees Celsius). As the temperature in myRio, or power converter or battery spikes above the threshold, the fans in the respective subsystem are activated for cooling.

In regards to the dropping stage, we came up with a new mechanism to initiate the dropping of the collected blue balls. When the Namseang-2 is on the opposite side of the demo court from the basket, the green ball detecting algorithm is less robust. While performing practice demos, we faced the problem of having the last blue ball on the opposite side of the court. Due to the large distance, the pixel-wise dimensions of the basket are too small to estimate distance. Therefore, the new method solves this problem by moving the vehicle to the center of the court before initiating dropping phase. We used the RP Lidar to obtain distances along every degree and average the results to detect whether the distances between opposite walls match. From the center, it is possible to efficiently detect the location and distance of the green balls for approaching the basket and dropping the balls.

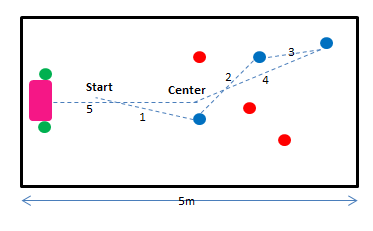


Fig. Example of possible trajectory for the vehicle in a case where the third blue balls is situated far from the basket where the OpenCV code is less robust. To avoid such mishaps, we made the vehicle rendez vous to the center of the demo court following the completion of the ball collecting phase. From the center, the OpenCV part is once again activated and the location of the green balls is found.