# Group 13's Building Blocks

A. Loh, S. Hu, *Department of Computer Science and Information Engineering, National Taiwan University*

***Abstract*— Through a combination of contour detection, color recognition, and length measurement, it is possible to program a robotic arm to autonomously detect and stack blocks into a stable structure.**

## I. Introduction

One of the common applications of robotic arms is the assembly of parts. However, rather than manually control the arm to do this, it would be a much more efficient use of manpower to program a level of autonomy into the arm to handle certain tasks by itself. One way to do this would be to use a camera to first detect the location of objects, and then feed the coordinates to the robotic arm to process. In addition, by finding the perimeter of each detected object and also doing color segmentation in the captured images, a level of object discernment can be achieved that will allow for some manner of flexibility in how the robotic arm can differentiate each object.

As far as work division goes, Alan Loh handled the contour detection and color segmentation part of the project, while Simon Hu took care of the camera calibration and robotic arm control.

## II. Method

The methodology employed in this project can be divided into several parts: camera calibration, image capture and contour detection, color segmentation, centroid and principal angle calculation, coordinate conversion, and robotic arm control.

### A. *Camera Calibration*

For intrinsic camera calibration, we followed the tutorial included in OpenCV's documentation [1]. First, we captured several pictures of a chessboard, and modified the pathname of the image list and the parameters in the configuration file. Then, we ran the sample code and got the camera matrix and distortion coefficient from the output file. Finally, we hard coded the matrices into our project, and undistorted the image.

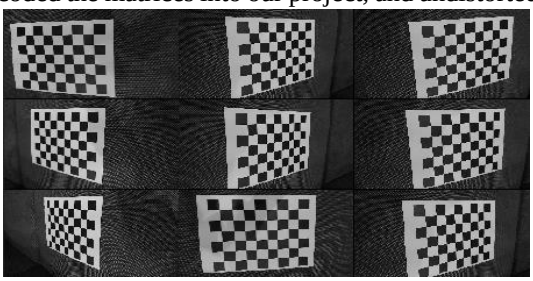


**Figure 1 – The images of chessboard captured**

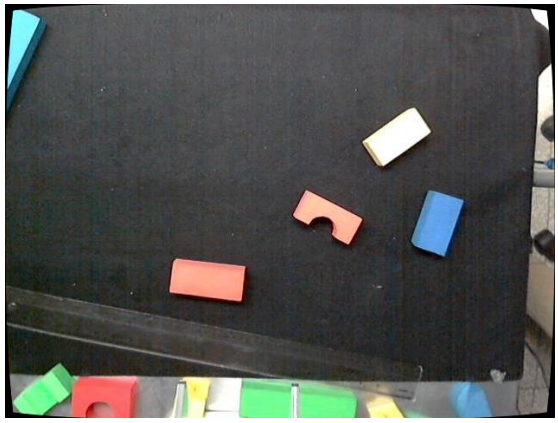


**Figure 2 – The calibrated image**

### B. *Color Segmentation*

Color segmentation is done through the HSV color spectrum. Images captured by the camera are first converted from RGB format to HSV. The images are then filtered through a threshold, where the saturation and value/lightness ranges are adjusted to reduce noise in order to get as clear a capture of the target objects as possible. The color segmentation part comes in when setting the hue value range to single out the four sets of colored blocks that were being used. A little research was done to find that the hue values of the four basic colors being used were 75-130 for blue, 160-179 for red, 22-38 for yellow, and 38-75 for green. Due to the lighting conditions in the setting used for the experiment, the actual values used were a little different.

After color segmentation, the resulting image would show white on areas of the segmented color and black everywhere else, making contour detection easier to accomplish.

### C. *Contour Detection*

Contour detection is largely done through existing OpenCV's library of functions. In particular, Canny is used to detect edges within the image, which is then processed through findContours() to get the actual contours [4].

To explain Canny without going into too much detail, according to OpenCV's documentation, noise is first filtered from the image using a Gaussian filter. An intensity gradient is then made from the image, where non-maximum suppression is applied to remove pixels that are not considered to be part of an edge. Finally, an upper and lower threshold is applied on the resulting pixels, where if the pixel is below the lower threshold or not connected to a pixel above the upper threshold, it gets rejected.

findContours() takes the results and finds any contours that can be drawn among the remaining pixels. This is done, according to OpenCV's documentation, through a border following process written by Suzuki, S. and Abe, K [5].

### *D. Calculations*

For each contour, its moment is calculated through OpenCV's built-in function moments(), which uses Green's theorem's formula for doing so [5]. By finding the moments of each contour, the centroid can also be calculated by finding the x-coordinate by dividing the contour's moment at the coordinates (1, 0) with the moment at (0, 0), and the y-coordinate by dividing the moment at (0, 1) with (0, 0).

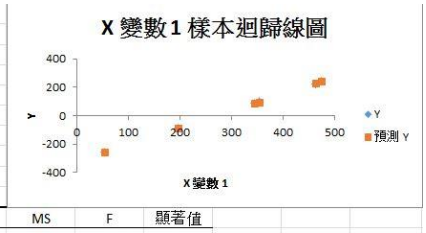
Having found the centroid of each object within camera view, every object outside of the robotic arm's field of operation gets filtered out by defining the coordinates of the four corners of the field and ignoring everything that lies outside of it.

The coordinates that get fed to the robotic arm is determined by manually positioning the robotic arm above the center of the object as seen in Figure 1. The x and y coordinates of the gripper's position are then retrieved from the software controlling the robotic arm and recorded as training points. These training points are used to find the correlation between the camera coordinates and the real world coordinates used by the arm, and are recorded into an Excel sheet to calculate the line of best for X and Y, as seen in Figure 2. The resulting equations for the X and Y coordinates are then applied directly into the program code itself to handle the conversion from centroids of detected objects on camera to actual coordinates that are fed into the robotic arm to maneuver to. The Z coordinate is found by simply manually recording the values used when positioning the robotic arm a certain distance above the table top used in the experiment.

**Figure 3 – Gripper positioning over block's centroid**

摘要輸出

| 迴歸統計 | |
|---|---|
| R 的倍數 | 0.999948 |
| R 平方 | 0.999895 |
| 調整的 R 平方 | 0.999869 |
| 標準誤 | 2.203656 |
| 觀察值個數 | 6 |




ANOVA

| | 自由度 | SS | MS | F | 顯著值 |
|---|---|---|---|---|---|
| 迴歸 | 1 | 185847.1 | 185847.1 | 38270.85 | 4.1E-09 |
| 殘差 | 4 | 19.4244 | 4.856101 | | |
| 總和 | 5 | 185866.5 | | | |

| | 係數 | 標準誤 | t 統計 | P-值 | 下限 95% | 上限 95% | 下限 95.0% | 上限 95.0% |
|---|---|---|---|---|---|---|---|---|
| 截距 | -321.102 | 2.101869 | -152.77 | 1.1E-08 | -326.938 | -315.266 | -326.938 | -315.2663 |
| X 變數 1 | 1.182683 | 0.006046 | 195.6294 | 4.1E-09 | 1.165898 | 1.199468 | 1.165898 | 1.199468 |

殘差輸出

| 觀察值 | 預測 Y | 殘差 |
|---|---|---|
| 1 | 240.5541 | 2.445907 |
| 2 | 226.9532 | -2.95324 |
| 3 | 86.33225 | -0.53225 |
| 4 | 97.3312 | 1.668803 |
| 5 | -89.5327 | -1.1673 |
| 6 | -258.538 | 0.538079 |

**Figure 4 – Calculating correlation between**

The principal angle is found by performing PCA analysis on the original image and the set of contour points to find the eigenvectors of each contours, which is then plugged into the atan2() function to find the resulting angle.

### *E. Robotic Arm Control*

The main issue in manipulating the robotic arm is controlling the joint angles and calculating the pose of endpoint. However, the ITRI RAS provides API which takes the pose of endpoint, so we do not actually have to implement inverse kinematics to get the joint angles. The only thing left is to get the position and orientation of the endpoint. The position can be calculated from camera intrinsic calibration and extrinsic calibration. However, the camera can only provide 1-D rotation information, which is about the Z axis, and the RAS document does not explicitly describes the angle system[2][3].

To find the correct representation of the Euler angles provided by the API, some trial and error was needed. First, we built elemental rotation matrices for the three axis, then we tried all the possible orders of rotation matrices and parameters, and with that we checked if the outcomes match the orientation of the robot arm. Finally, we decide if it is yaw-pitch-roll convention, or intrinsic Z-Y'-X'', or extrinsic X-Y-Z.

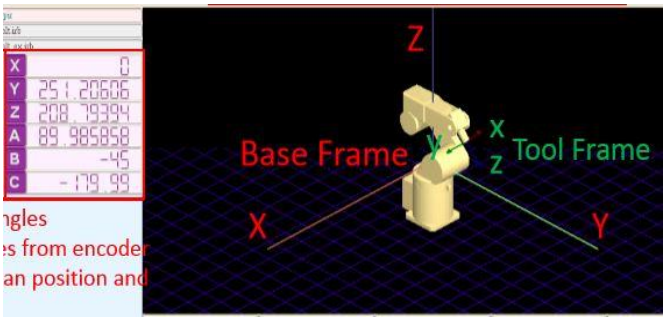



**Figure 5 – The base coordinate and gripper coordinate provided in the manual**

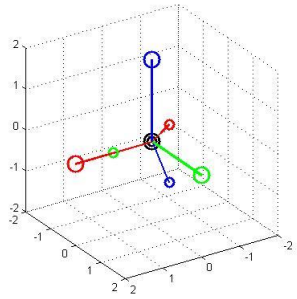


**Figure 6 – The simulation of base and gripper coordinate using Matlab**

To get the parameters of orientation for ITRI's API, we also have to obtain the Euler angles from the rotation matrix. Having solved the conventions before, we used algebra similar to those used for solving inverse kinematic in Homework 2. Details are in the appendix.

The API is able to takes pose of an endpoint as input through two different methods: MOVP and MOVL. MOVL uses linear interpolation in 3D space, while MOVP does not. MOVL is better most of the time, because it guarantees the endpoint moves in straight line, while MOVP could end up traversing through large curve because it just interpolates joint angles. However, when the robotic arm hits a singularity point, MOVL will cause the movement stop because of the undefined behavior of how to cross singularity point. Because some manner of delicacy is needed in the gripper's movement, which may lead to singularity problem, so we chose to use MOVP instead.

### *F. Stacking Algorithm*

We were supposed to find an algorithm which can fill a 3D model with heterogeneous blocks. However, due to time constraint, we implement a simplified version based on color segmentation and the total length of blocks of each color. First, we find the order and the position of the objects to grep. This involves finding the length of each object under each color filter, and summing them up to the four total lengths. The total lengths get sorted in descending order, and each object gets queued into a grepping queue that follows that order. We then calculate the stacking position as follows:

$$Y_{i,j} = Y_0$$

$$Z_{i,j} = Z_0 + j \times block\_height$$

$$X_{i,j}$$

$$= X_0 + \sum_{k=0}^{n(j)} length(j,k)/2 - \sum_{k=0}^{i} length(j,k)$$

$$- length(j,i)/2$$

j = the order (level) of this color.
i = the order of blocks in color j.
n(j) = the number of blocks in color j.

This results in the blocks being stacked in order total length based on color, with the longest at the bottom. This would create a stable castle-like structure, as seen in Figure 7.

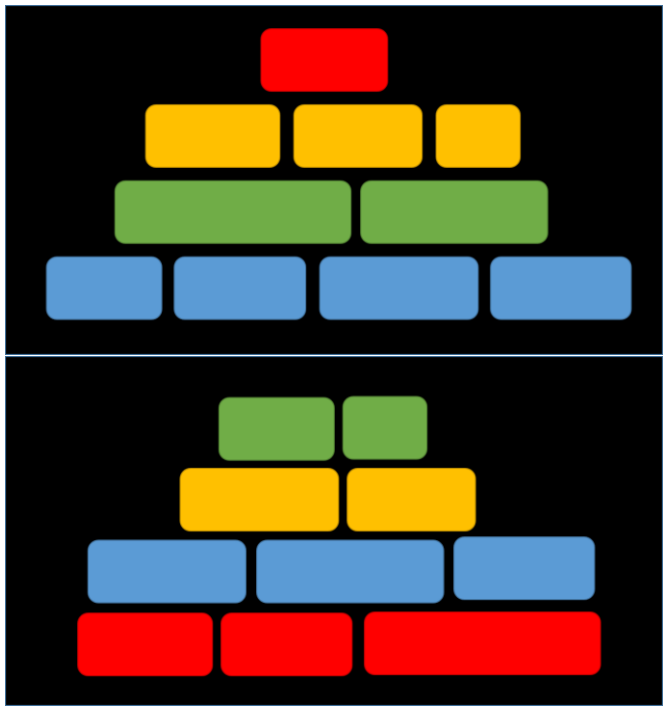


**Figure 7 – Two possible combinations of output structure**

## III. Experiment And Results

Overall, with a few shortcomings and limitations, the robotic arm was able to accomplish the task of stacking a stable structure autonomously.

### *A. Color Segmentation*

The main crux of the problem lied in detecting the colored blocks after color segmentation. Using the camera image shown in Figure 8 as an example, it was found that a saturation value range of 1 to 255, value/lightness range of 100 to 255, and a X and Y border from 0 to 600 and 500, respectively, produces an optimal image for color and object detection, as seen in Figure 9.

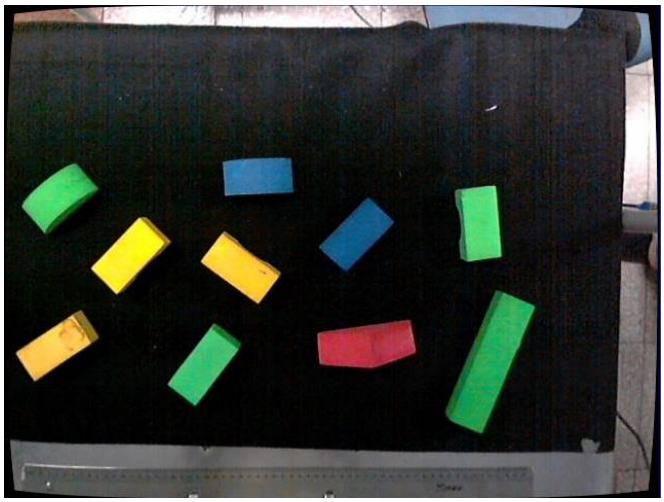


Figure 8 - Original Camera Image

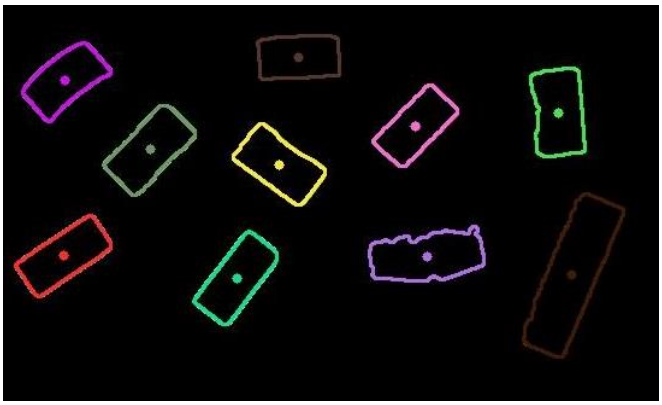


Figure 9 – Detected Contours

However, when it came to setting the hue ranges, it was initially found that the lighting of the room made it difficult to detect certain colors due to the light's reflection. This was compensated by using a plastic mat to reduce the glare casted on the area picked up by the camera.

Since only the colors yellow, green, blue, and red were used for the experiment, the hue value ranges were set to be 10 to 40, 44 to 95, 95 to 114, and 140 to 179, respectively for each color. The resulting color segmented images can be seen in Figures 10 through 12.

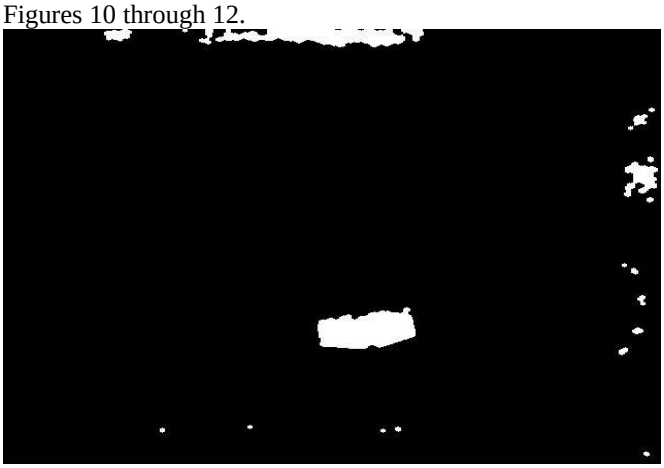


Figure 10 – Color Segmented For Red

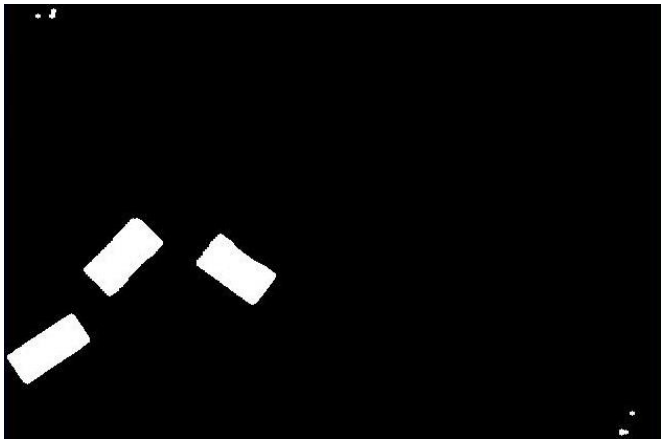


Figure 11 - Color Segmented For Yellow

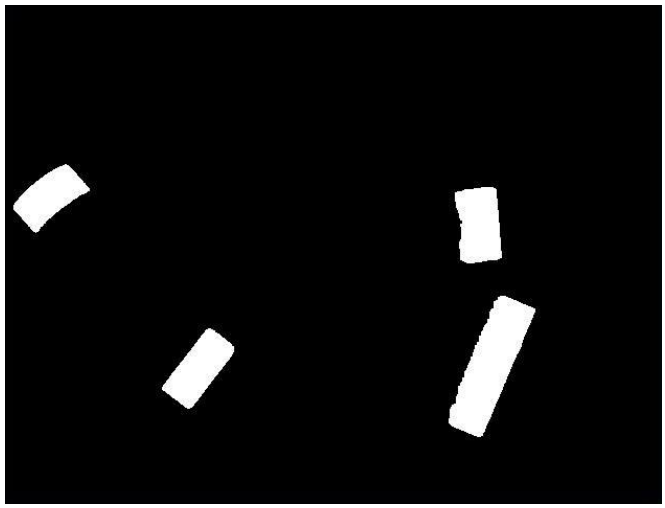


Figure 12 - Color Segmented For Green

Figure 13 - Color Segmented For Blue

Even then, there were still problems detecting the color red, as seen in the imperfect contour drawn for the red triangle in Figure 9 and the color segmentation done in Figure 10 compared to the rest. The problem seemed to be that since red lies at the extreme end of the HVS color spectrum, the hue value range for it actually extends past 179 and overlaps back to 0, seeing how the camera was able to perfectly pick up red

blocks when viewed under the full 0 to 179 spectrum. Limitations in OpenCV's coding library and time constraint prevented a solution to be made to include two ranges of hue values to be used during the segmentation process, and since a specific shade of red could still be detected despite these shortcomings, only blocks of that shade were used to help alleviate the problem.

### *B. Robotic Arm Control*

We confronted some inaccuracy in the experiments, as can be seen in some of the tests we made.

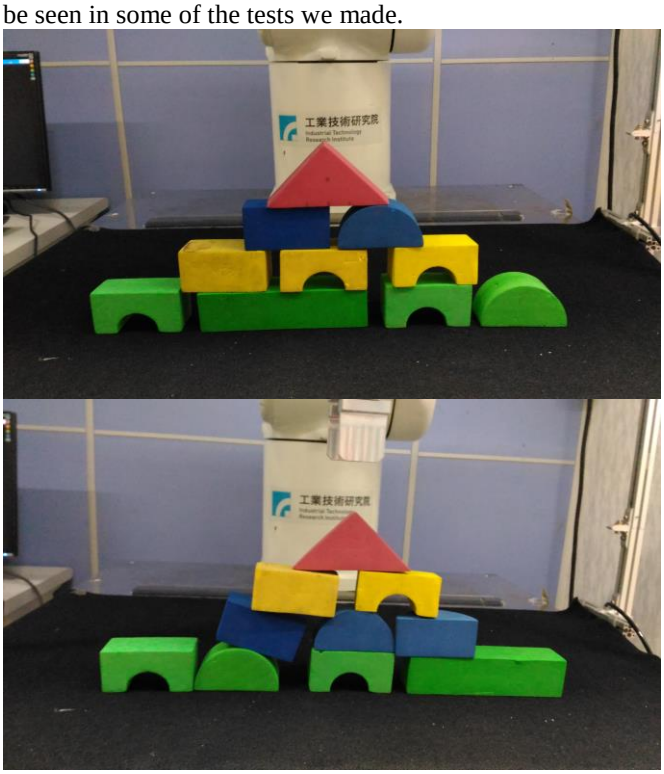


**Figure 14 – Two of the results**

One of the problem is that there are some inaccuracies in placement in the Y direction caused by a slight bounce in the object when released by the gripper. This phenomenon is probably caused by the tension of the foam built from the pressure of the gripper, and when opened too quickly, the tension transforms into momentum.

Another major issue is that the gap in the X direction is not uniform. This is caused from error in camera calibration and perspective transformation, as the program used for this project is very sensitive to any changes in the environment with regards to the positioning of the robotic arm and the camera. If we had more time and the camera remained untouched throughout the project, we believe collecting more data for training could improve the precision of the coordinates that get fed to the robotic arm.

## IV. Conclusion

Through this project, it can be concluded that it is possible to program a level of intelligence into a robotic arm to be able to assemble or build a model or structure in an arbitrary situation. Despite shortcomings and falling short of the initial goal of being able to stack blocks according to a model passed into the program, being able to build a stable structure based on color and block length can be considered a solid first step in achieving that.

## Appendix

Here is the process of finding Euler angles. First, we list the element rotation matrices, and combine them together in yaw-pitch-roll convention.

$$R_x(\theta) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\theta) & -\sin(\theta) \\ 0 & \sin(\theta) & \cos(\theta) \end{bmatrix}$$

$$R_y(\theta) = \begin{bmatrix} \cos(\theta) & 0 & \sin(\theta) \\ 0 & 1 & 0 \\ -\sin(\theta) & 0 & \cos(\theta) \end{bmatrix}$$

$$R_z(\theta) = \begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$R = R_z(\theta_z) \cdot R_y(\theta_y) \cdot R_x(\theta_x)$$

$$= \begin{bmatrix} -s_y c_z & s_x s_y c_z - c_x s_z & c_x s_y c_z + s_x s_z \\ c_y s_z & s_x s_y s_z + c_x c_z & c_x s_y s_z - s_x c_z \\ -s_y & s_x c_y & c_x c_y \end{bmatrix}$$

Here we use $\theta_x$, $\theta_y$, $\theta_z$ as the angle of each axis, and $s_a$, $c_a$ to represent $\sin(\theta_a)$, $\cos(\theta_a)$.
If $\cos(\theta_y) \neq 0$, we can get $\theta_x$ from atan2, and then $\theta_y$:

$$\theta_x = atan2(s_x c_y, c_x c_y) = atan2(R(3,2), R(3,3))$$

$$\theta_y = atan2(s_y, c_y)$$

$$= \begin{cases} atan2(-R(3,1), R(3,2)/s_x), & if\ s_x \neq 0 \\ atan2(-R(3,1), R(3,3)/c_x), & otherwise \end{cases}$$

When we have $\theta_x$, $\theta_y$, we can get $R_z$, and then $\theta_z$:

$$R_z = R \cdot {R_x}^{-1} \cdot {R_y}^{-1} = R \cdot {R_x}^{T} \cdot {R_y}^{T}$$

$$\theta_z = atan2(R_z(2,1), R_z(1,1))$$

So far we have done the case when $\cos(\theta_y) \neq 0$.
Now we have to consider when $\cos(\theta_y) = 0$:
$\theta_y = atan2(s_y, c_y) = atan2(-R(3,1), 0)$
Here we can simplify the rotation matrix:

$$R = \begin{bmatrix} -s_y c_z & s_x s_y c_z - c_x s_z & c_x s_y c_z + s_x s_z \\ 0 & s_x s_y s_z + c_x c_z & c_x s_y s_z - s_x c_z \\ -s_y & 0 & 0 \end{bmatrix}$$

If $\sin(\theta_y) = 1$, or $R(3,1) = -s_y = -1$:

$$R = \begin{bmatrix} -c_z & s_x c_z - c_x s_z & c_x c_z + s_x s_z \\ 0 & s_x s_z + c_x c_z & c_x s_z - s_x c_z \\ -1 & 0 & 0 \end{bmatrix}$$

$$= \begin{bmatrix} -c_z & s_{x-z} & c_{x-z} \\ 0 & -c_{x-z} & -s_{x-z} \\ -1 & 0 & 0 \end{bmatrix}$$

$$\theta_x - \theta_z = atan2(R(1,2), R(1,3))$$

But we do not have $s_z$, so we have use $acos(\theta_z)$, and regulate the range of $\theta_z$ by human knowledge:

$$\theta_z = \begin{cases} acos(-\mathrm{R}(1,1)), & if\ aco\,s\big(-\mathrm{R}(1,1)\big) \geq 0 \\ -aco\,s\big(-\mathrm{R}(1,1)\big) + \pi, & otherwise \end{cases}$$

Which returns the $\theta_z$ in [0, π]

$$\theta_x = atan2\big(R(1,2), R(1,3)\big) + \theta_z$$

Similarly, if $\sin(\theta_y) = -1$, or $\mathrm{R}(3,1) = 1$:

$$R = \begin{bmatrix} c_z & -s_x c_z - c_x s_z & -c_x c_z + s_x s_z \\ 0 & -s_x s_z + c_x c_z & -c_x s_z - s_x c_z \\ 1 & 0 & 0 \end{bmatrix}$$

$$= \begin{bmatrix} c_z & -s_{x+z} & -c_{x+z} \\ 0 & c_{x+z} & -s_{x+z} \\ 1 & 0 & 0 \end{bmatrix}$$

$$\theta_x + \theta_z = atan2(-R(1,2), -R(1,3))$$

$$\theta_z = \begin{cases} acos(\mathrm{R}(1,1)), & if\ aco\,s\big(\mathrm{R}(1,1)\big) \geq 0 \\ -aco\,s\big(\mathrm{R}(1,1)\big) + \pi, & otherwise \end{cases}$$

$$\theta_x = atan2\big(R(1,2), R(1,3)\big) - \theta_z$$

Now we have done all the cases including $\cos(\theta_y) \neq 0$ and $\cos(\theta_y) = 0$.

## REFERENCES


[1] http://docs.opencv.org/2.4/doc/tutorials/calib3d/camera_calibration/camera_calibration.html
[2] https://en.wikipedia.org/wiki/Euler_angles
[3] https://en.wikipedia.org/wiki/Rotation_matrix
[4] http://docs.opencv.org/2.4/doc/tutorials/imgproc/shapedescriptors/find_contours/find_contours.html
[5] http://docs.opencv.org/2.4/modules/imgproc/doc/structural_analysis_and_shape_descriptors.html?highlight=findcontours#findcontours