NC State University
Department of Electrical and Computer Engineering
ECE 785: Embedded Linux
Spring 2019
Project #3B: Image Stabilization Gimbal

by

SALONI SHAMBHUWANI
200266197
sdshambh@ncsu.edu

AVANTIKA CHAUDHURI
200261258
achaudh8@ncsu.edu

OVERVIEW:

Stabilization is the act of keeping your camera as still as possible while you capture a scene. There are many different ways we can stabilize our images, without lens or in-camera help. Depending on the make and model of your camera or lens, image stabilization helps. In this project, we are creating image stabilization gimbal using webcam connected to Beaglebone Black and openCV to identify the location of two reference marks in an image, and then calculate pan, tilt and roll error. Let us understand a bit more detail about the hardware involved.

Beaglebone Black is the main backbone of this project. A single-board computer is pretty much what it sounds like—all the hardware that we would expect to find in a desktop or laptop computer, shrunken down and soldered to a single circuit board. A processor, memory, and graphics acceleration are all present as chips on the board.

To this powerful board, we attach Logitech webcam C920 as an USB peripheral. This will be used to capture an image with two yellow colored reference objects and later on using openCV, we will be processing on the image.

Main purpose of processing on the image is to detect the two yellow objects and find pan, tilt and roll error. These basic measured parameters help us built a pre-system to a servo motor based on the run image stabilizer. Now let us dive into further details of the project.

INITIAL SETUP:

1. **OPENCV installation:**

Our first step towards the project was to install openCV library successfully. After a vast research on net, reading through Derek Molloy guides and youtube videos and referring second edition BeagleBone Black textbook by Derek Molloy, we started following steps given on this website: https://solarianprogrammer.com/2014/04/21/opencv-beaglebone-black-ubuntu/

However, soon into the downloading, we ran out of space. And then after few days of struggle, we kick-started project by loading image onto SD card, then creating partition on SD card to utilize 8GB worth of space and downloaded openCV library from above web source.

After 5 hours of compiling of full code, we ran into an unexpected error while compiling the last module. Again after intensive research, there were few things to be taken care before directly jumping into compiling openCV library.

Finally after almost 3 days of struggle just to setup openCV, we got it to running on board. The website had given basic few codes to test if openCV libraries were correctly installed. The following code was used to test:

```
1  // Test to convert a color image to gray
2  // Build on Linux with:
3  // g++ test_2.cpp -o test_2 -lopencv_core -lopencv_imgproc -lopencv_h
4
5  #include <opencv2/opencv.hpp>
6  #include <iostream>
7
8  int main() {
9      // Load the image file and check for success
10     cv::Mat input = cv::imread("lena.jpg", 1);
11     if(!input.data) {
12         std::cout << "Unable to open the image file" << std::endl;
13         return -1;
14     }
15
16     // Convert the input file to gray
17     cv::Mat gray_image;
18     cvtColor(input, gray_image, cv::COLOR_BGR2GRAY);
19
20     // Save the result
21     cv::imwrite("lena_gray.jpg", gray_image);
22
23     return 0;
24 }
```

Initially it gave error while compile time. This was because of the version of OpenCV which we are using in our code. So now to compile our code run the below command:

g++ project3b_SA.cpp -o project3b_SA `pkg-config opencv --cflags —libs`

After getting the basic openCV code to work, we started understanding the various commands and openCV inbuilt functions which used in our project are given below:

Creating MAT object: *Mat* is a class with two data parts: the matrix header (containing information such as the size of the matrix, the method used for storing, at which address is the matrix stored, and so on) and a pointer to the matrix containing the pixel values (taking any dimensionality depending on the method chosen for storing) . The matrix header size is constant, however the size of the matrix itself may vary from image to image and usually is larger by orders of magnitude. When we want to create image with various functionalities of OpenCV we will end up creating multiple copies of same Image. This is where MAT container helps. It just creates the pointer to the image for each cope and while we write the image it attaches it's pixel information with it. Below is the code that shows how to create a MAT container in OpenCV

```
Mat A, C;
```

Image reading: To allocate the matrix to the MAT container we use imread function. The input parameter to this function is the path of the image(`argv[1]`) and the color space in which the image needs to be read.

```
A = imread(argv[1], IMREAD_COLOR);
```

Image writing: To save the image in the specified folder or directory we use imwrite() function. This is used to see the final result of the image. Also we can use imshow() to see the resulting image on screen. Below is the code snippet for the same:

```
imwrite("output_image.jpg", C);
```

Convert color of Image: OpenCV has multiple color spaces defined in the library including BGR, HSV, gray, YCrCb, Bayer RGB, etc. In our test code we choose gray color and convert the original image into gray color image and save it using imwrite() function. The function to convert into different color space is defined in OpenCV libraries. Below is the code snippet for the same:

```
cvtColor(C, C, COLOR_BGR2HSV);
```

STEPS PERFORMED:

Step 1: Capturing Image and reading it in C++ file

To understand this in depth, we first made a stand alone code known as *capture_photo.cpp*. Below is the snapshot of code:

```
#include <opencv2/opencv.hpp>

using namespace cv;

int main(int argc, char** argv)
{
    VideoCapture cap;
    // open the default camera, use something different from 0 otherwise;
    // Check VideoCapture documentation.
    if(!cap.open(0))
        return 0;
    for(;;)
    {
          Mat frame;
          cap >> frame;
          if( frame.empty() ) break; // end of video stream
          imwrite("capture.jpg", frame);
          if( waitKey(30) >=0 ) break;
    }
    // the camera will be closed automatically upon exit
    // cap.close();
    return 0;
}
```

As you can see, cap is type VideoCapture that captures image from webcam and places it in frame which is of type MAT. Then using imwrite, frame is written into a jpg file and exit the code. This saves the image on board desktop successfully.

After the image has been saved, we read the image from the board itself. Code snippet attached:

```
,
cv::Mat src = cv::imread("input_img.jpg", 1);
if(!src.data)
{
    std::cout << "Unable to open the image file" << std::endl;
    return -1;
}
```

Step 2: Blurring the Image

OpenCV has function blur() to perform smoothing with this filter. We specify 4 arguments (more details, check the Reference):

- src: Source image
- dst: Destination image
- Size(w, h): Defines the size of the kernel to be used (of width w pixels and height h pixels)
- Point(-1, -1): Indicates where the anchor point (the pixel evaluated) is located with respect to the neighborhood. If there is a negative value, then the center of the kernel is considered the anchor point.

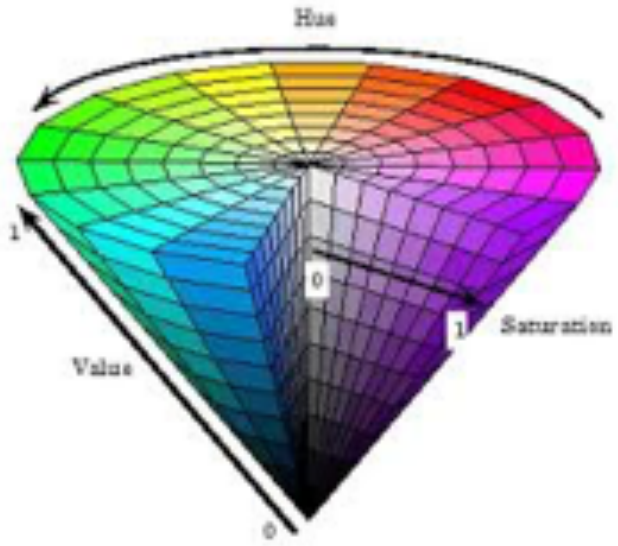
The blurring function allowed us to form the circles around our areas of concern. Also the matrix size for MAT objects reduced. This helped in narrowing down to pixels we want to work with. Below is the code snippet of the implementation in our project:

```
for ( int i = 1; i < MAX_KERNEL_LENGTH; i = i + 2 )
{
  cv::blur( src, dst, cv::Size( i, i ), cv::Point(-1,-1) );
}
```

Step 3: Detecting yellow references: Converting Image to HSV to identify yellow color

The main motive to convert the image into HSV color is to perform thresholding in the range of yellow color pixel. The inrange() function in OpenCV helps in detecting the pixels in the range specified in the function (in our case pixels for yellow). HSV color space models the color in the format similar to BGR color space. It doesn't change the image just adjust the pixel values using hue, saturation, value. Below is the code snippet for converting image into HSV color space. (This works exactly same as specified above in OpenCV test section):

```
cvtColor(dst, dst, COLOR_BGR2HSV);
inRange(dst, Scalar(low_H, low_S, low_V), Scalar(high_H, high_S, high_V), dst_HSV);
```



Thresholding:

- Detect an object based on the range of pixel values in the HSV colorspace.
- In the above figure,the highlighted area in black shows the range for yellow. So we selected that range in low and high HSV variables.
- Below is range specification for yellow color (from code):

```
int low_H = 15, low_S = 65, low_V =175;
int high_H = 180, high_S = 255, high_V = 255;
```

In the above code, snippet we see that the threshold inrange() function takes these values makes that area white(as the area of concern) and makes the rest of the image as dark. This is how we detect the yellow color references and make them our area of operation.

Step 4:Detecting edges and Finding contours

From the image formed from Step 3, we detect the edges (i.e. the white area- the yellow color area)using canny function and centre of that area by using moments and contour functions of OpenCV.

Canny: Canny Edge Detection is a edge detection algorithm performing following steps: Noise reduction, finding intensity gradient of image, Non-maximum suppression, Hysteresis thresholding. This gives us the binary image with thin edges.

Moments & Contours: Contours can be explained simply as a curve joining all the continuous points (along the boundary), having same color or intensity. The contours are a useful tool for shape analysis and object detection and recognition. Using this property we found the curve of all the area filled with yellow color pixels.

Image Moment is a particular weighted average of image pixel intensities, with the help of which we can find some specific properties of an image, like radius, area, centroid etc. To find the moments in the image we need the image in binary format. So we converted the resulting image into binary format and found the moment of the image using the moment function defined in OpenCV. Using moments we could find the points and centre of the reference areas. Also using the contour curves we drew circle around the yellow color area.

Below is the code snippet for the steps explained:

```
// detect edges using Canny
Canny( dst_HSV, canny_output, 50, 150, 3 );

// find contours
findContours( canny_output, contours, hierarchy, RETR_TREE, CHAIN_APPROX_SIMPLE, Point(0, 0) );

//get contours and centroid of figures
vector<Moments> mu(contours.size());
vector<Point2f> mc(contours.size());
for( int i = 0; i<contours.size(); i++ )
{
    mu[i] = moments( contours[i], false );
    mc[i] = Point2f( mu[i].m10/mu[i].m00 , mu[i].m01/mu[i].m00 );
}

Mat drawing(canny_output.size(), CV_8UC3, Scalar(255,255,255));
for( int i = 0; i<contours.size(); i++ )
{
    Scalar color = Scalar(4,255,248); // B G R values
    drawContours(drawing, contours, i, color, 2, 8, hierarchy, 0, Point());
    circle( drawing, mc[i], 4, color, -1, 8, 0 );
}
```

Step 5: Finding centre of yellow reference points

While finding counters, we used function Point2f() which is used to determine and store x and y coordinate of center of each contour. However, it determines coordinates of ends of the center circle, as center is not a dot. Therefore, I am taking both end points of

diameter and determining coordinate of center of circle. And using the center of both objects, I am determining midpoint of line joining them. Code snippet attached below:

```
centerPoint points[2];
int j = 0;
for( int i = 0; i<contours.size(); i+=2 )
{
    points[j].x = (mc[i].x + mc[i+1].x)/2;
    points[j].y = (mc[i].y + mc[i+1].y)/2;
    j++;
}

centerPoint midpoint, joint;
midpoint.x = (points[0].x + points[1].x)/2;
midpoint.y = (points[0].y + points[1].y)/2;
```

Step 6: Finding Image centre

Using size(), we can obtain height and width of any MAT format image. Then simply dividing width and height by 2, we determine coordinate of center of the image.

```
centerPoint imgCentre;
imgCentre.x = (src.size().width)/2;
imgCentre.y = (src.size().height)/2;
```

Step 7: Distance and Error calculations

As per our understanding and diagram provided by Dr. Dean,

Pan error is parallel distance between x axis through center of screen and x-axis through midpoint of line joining 2 yellow object.

Tilt Error is parallel distance between y axis through center of screen and y-axis through midpoint of line joining 2 yellow object.

For ease of understanding and work, we made a triangle with 3 vertices as center of screen, calculated mid-point and joint. Now when we carefully notice, this triangle is always a right-angled triangle. And joint is intersection of x-axis through mid-point and y-axis through center of screen. Therefore, coordinates of joint vertices is (x value of midpoint, y value of center of image).

Then using distance formula,

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

We measure pan and tilt error distance.

Roll error is angle between closet axis and line joining 2 yellow object, which is found by inverse tangent function of slope of line joining 2 yellow objects.

Code snippet attached below:

```
//PAN and TILT error
double pan_diff = pow((imgCentre.x - joint.x),2) + pow((imgCentre.y - joint.y),2);
double pan_error = sqrt(pan_diff);

double tilt_diff = pow((midpoint.x - joint.x),2) + pow((midpoint.y - joint.y),2);
double tilt_error = sqrt(tilt_diff);

cout << "Pan Error: " << pan_error << endl;
cout << "Tilt Error: " << tilt_error << endl;

//ROLL error
float m2 = (points[1].y - points[0].y) / (points[1].x - points[0].x);
float m1 = 0.0;
float m = (m2 - m1)/(1 + m1*m2);

float theta = atan(m2)*pi_val;
if(theta < 0)
{
    theta = 90 + theta;
}
cout << "Roll Error (in degrees): " << theta << endl;
```

Step 8: Generation of output image

As per requirement of project spec, we have to generate an output image with circles around reference marks and line between marks.

To use this, we played around with various openCV inbuilt functions.

For instance, to draw circle around reference marks, we initially used circle() to draw a static circle. However, that was a static circle with fixed radius and not covering reference mark. So to do so, we went ahead with the following code. Code snippet attached below:

```
vector<vector<Point> > contours_poly( contours.size() );
vector<Rect> boundRect( contours.size() );
vector<Point2f>centers( contours.size() );
vector<float>radius( contours.size() );

for( int i = 0; i<contours.size(); i++ )
{
    approxPolyDP( contours[i], contours_poly[i], 3, true );
    minEnclosingCircle( contours_poly[i], centers[i], radius[i] );
    circle( src, centers[i], (int)radius[i], Scalar( 255, 0, 0 ), 2 );
}
line( src, Point( points[0].x, points[0].y ), Point( points[1].x, points[1].y), Scalar( 255, 0, 0 ),  2, 8 );
```

In this case, we are using apronxPolyDP that determines approximate size of contour. Then we use minEnclosingCircle() to determine radius of a circle around contour and making it using circle(). This leads to creation of more dynamic circle as per size of contour.

Followed by using center points of contour to draw line between 2 yellow objects.

And then writing MAT file onto *output_image.jpg*.

TIMING INFORMATION:

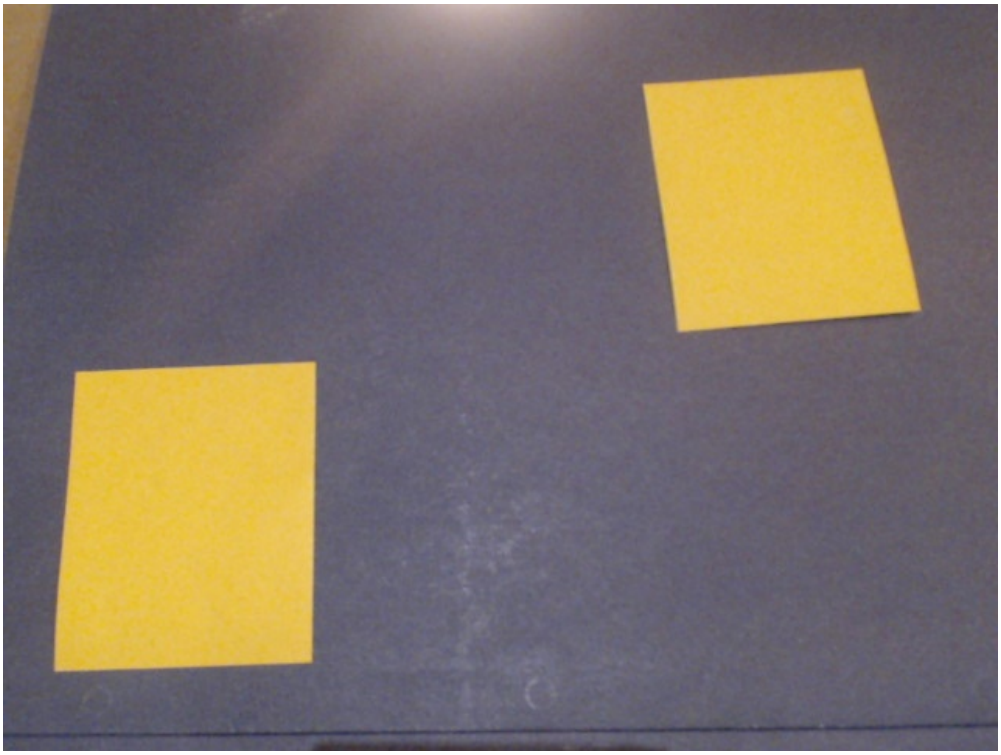
In order to determine total time taken by code, we added duration using inbuilt clock() function. **Total time taken: 2.23723 ms**.

FINAL OUTPUT:

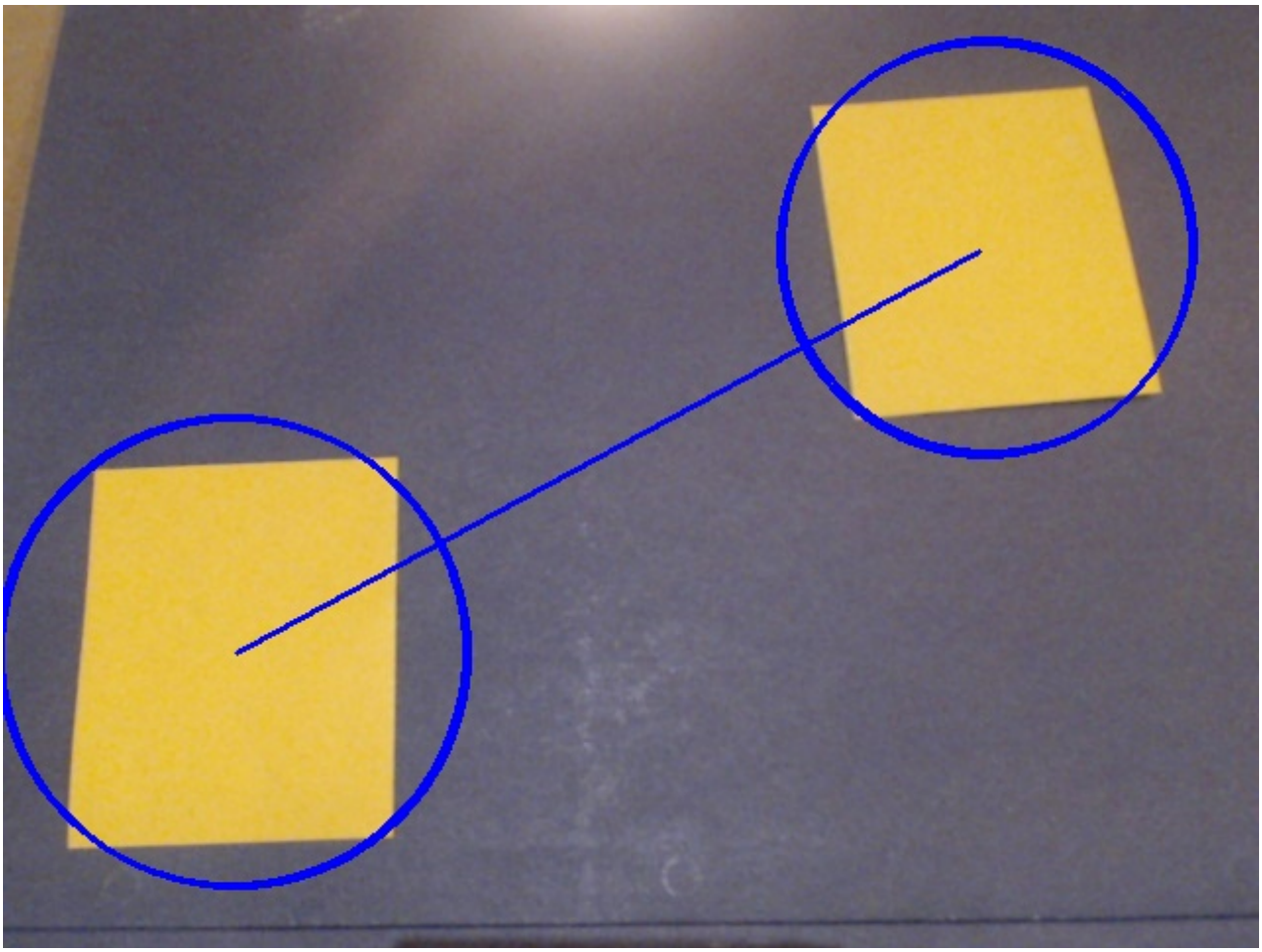
Putty output:

```
debian@beaglebone:~/opencv_Avantika$ ./project3b_SA
Pan Error: 11.5943
Tilt Error: 12.2252
Roll Error (in degrees): 61.6543
Program-Run Time: 2.23723
debian@beaglebone:~/opencv_Avantika$
```

Input image:



Output image:



PICTURE REFERENCE:

1. AFTER APPLYING HSV

2. AFTER INRANGE FUNCTION //DEFINES CONTOUR

CHALLENGES FACED:

1. openCV library compiling and running on board took almost 3 days. Initially ran out of space. Then error while compiling and finally worked after 3 whole days.
2. Setting of threshold value took time as there is no definite rage for yellow. With loads of trial and error, it finally worked.
3. Lighting for webcam mattered mostly. In dim light, we received erroneous output as contouring were differing every time.

REFERENCE:

https://solarianprogrammer.com/2014/04/21/opencv-beaglebone-black-ubuntu/

https://docs.opencv.org/3.4/d6/d6d/tutorial_mat_the_basic_image_container.html

https://docs.opencv.org/3.4/db/da5/tutorial_how_to_scan_images.html

https://docs.opencv.org/3.4/d3/d96/tutorial_basic_geometric_drawing.html

https://docs.opencv.org/3.4/df/d61/tutorial_random_generator_and_text.html

https://docs.opencv.org/3.4/dc/dd3/tutorial_gausian_median_blur_bilateral_filter.html

https://www.learnopencv.com/color-spaces-in-opencv-cpp-python/

https://docs.opencv.org/3.4/da/d97/tutorial_threshold_inRange.html