This lab assignment should be done in teams of two. Go through the exercises below and show me your results. If you don’t finish by the end of today’s class, you may show me during the next class.

The task for today is to use OpenCV to detect a target composed of 5 contrasting concentric circles (CCCs) in images from a live camera, or video, and compute the pose of the target. The dimensions (inches) of the target pattern are shown in the figure.

Create an OpenCV project. If you have an existing OpenCV project, the easiest thing is to copy the entire folder of that project; otherwise follow the steps from the previous lecture to set up a new project.

Finding the Target Pattern

Enter the following code for the main program. Note – after you copy and paste code, it is a good idea to “clean up” the formatting. You can do this by going to Edit->Advanced->Format Document.

```c++
#include <iostream>
#include <opencv2/opencv.hpp>

std::vector<cv::Point2d> findTargets(cv::Mat Image);
std::vector<cv::Point2d> orderTargets(std::vector<cv::Point2d> allTargets);

int main(int argc, char* argv[]) {
    cv::VideoCapture cap(0);// open the camera
    //cv::VideoCapture cap("fiveCCC.avi"); // or a video
    if (!cap.isOpened()) { // check if we succeeded
        printf("error - can't open the camera\n");
        system("PAUSE");
    }
```
return EXIT_FAILURE;
}
printf("Hit ESC key to quit ...
\n");

// Loop until no more images are available, or person hits "ESC" key.
while (true) {
    cv::Mat imageInput;
    cap >> imageInput; // get image from camera or video file
    if (imageInput.empty()) break;

    // Convert to gray.
    cv::Mat imageInputGray;
    cvtColor(imageInput, imageInputGray, cv::COLOR_BGR2GRAY);

    // Find all CCC targets.
    std::vector<cv::Point2d> allTargets;
    allTargets = findTargets(imageInputGray);

    // Draw dots on all targets found.
    for (unsigned int i = 0; i < allTargets.size(); i++) {
        cv::circle(imageInput,
                   allTargets[i], // center
                   3, // radius
                   cv::Scalar(0, 0, 255), // color
                   -1); // negative thickness=filled
    }

    // Find the 5-CCC target pattern, and put the targets in the order
    // 1  2  3
    // 4  5
    std::vector<cv::Point2d> orderedTargets;
    orderedTargets = orderTargets(allTargets);

    // Optionally label targets in the image.
    for (unsigned int i = 0; i < orderedTargets.size(); i++) {
        char szLabel[50];
        sprintf_s(szLabel, "%d", i);
        putText(imageInput, szLabel, orderedTargets[i],
                cv::FONT_HERSHEY_PLAIN, // font face
                2.0, // font scale
                cv::Scalar(0, 255, 0), // font color
                2); // thickness
    }

    imshow("Image", imageInput);

    // Wait for xxx ms (0 means wait until a keypress)
    if (cv::waitKey(1) == 27) break; // hit ESC (ascii code 27) to quit
}
return EXIT_SUCCESS;
Add two additional functions to your project. To do this, go to the “Solution Explorer” panel, right click on “Source Files”, and select Add->New Item. Name one file “findTargets.cpp” and the other “orderTargets.cpp”. The first function finds all CCC targets in the image, using the methods described in the lecture. The other function puts the found targets into the correct order; namely:

```
0 1 2
3 4
```

The code for these two functions are given at the end of this document. Copy and paste the code into the files. Run the program and verify that it finds the target pattern and labels the targets in the right order. The output should look like the image below.

![Image of targets labeled](image.png)

Finding the Pose

Enter the following code in your main loop, after the point where you label the targets, but before displaying the image. Certain places in the text have been overwritten by $$$ ... you will have to figure out what should go there.

```cpp
if (orderedTargets.size() == 5) {
    // Calculate the pose.
    cv::Mat rotVec, transVec;
    bool foundPose = cv::solvePnP($$$, // model points
                                $$$,  // image points
                                K,    // intrinsic camera parameter matrix
                                dist, // distortion coefficients
                                rotVec, transVec); // output rotation and translation

    if (foundPose) {
        std::vector<cv::Point3d> pointsAxes;
        std::vector<cv::Point2d> pointsImage;

        // Draw the xyz coordinate axes on the image.
        pointsAxes.push_back(cv::Point3d(0, 0, 0)); // origin
```
Enter the following code prior to the main loop. Certain places in the text have been overwritten by $$$ ... you will have to figure out what should go there.

```cpp
// Create 3D object model points, in the order
// 0 1 2
// 3 4
std::vector<cv::Point3d> pointsModel;
pointsModel.push_back(cv::Point3d(-3.7, $$$, 0));
pointsModel.push_back(cv::Point3d($$$, $$$, $$$));
pointsModel.push_back(cv::Point3d($$$, $$$, $$$));
pointsModel.push_back(cv::Point3d($$$, $$$, $$$));
pointsModel.push_back(cv::Point3d($$$, $$$, $$$));

// Camera intrinsic matrix
double K_[3][3] =
    {{ 675, 0, 320 },
     { 0, 675, 240 },
     { 0, 0, 1 });
cv::Mat K = cv::Mat(3, 3, CV_64F, K_).clone();
cv::Mat dist = cv::Mat::zeros(5, 1, CV_64F); // distortion coeffs
```

Run the program and verify that it displays the coordinate axes correctly. The output should look like the image below. Show the instructor your running program.
Code for the function “findTargets”:

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

// Parameters for finding CCC targets.
#define DPIXEL 3.0 // max distance between centroids

// Find all CCC targets on the image. Return a vector (list) of their locations.
std::vector<cv::Point2d> findTargets(cv::Mat Image) {
    std::vector<cv::Point2d> targets;
    cv::Mat imageThresh;

    // Do Otsu global thresholding - is faster than adaptive thresholding.
    cv::threshold(Image, imageThresh, 0, 255, cv::THRESH_OTSU | cv::THRESH_BINARY_INV); // threshold_type - invert

    // Do adaptive threshold ... this compares each pixel to a local mean of the neighborhood. The result is a binary image, where dark areas of the original image are now white (1's).
    //adaptiveThreshold(Image,
    // imageThresh, 255, cv::ADAPTIVE_THRESH_GAUSSIAN_C, cv::THRESH_BINARY_INV, 91, 0); // a constant to subtract from mean

    // Apply morphological operations to get rid of small (noise) regions.
    cv::Mat structuringElmt = cv::getStructuringElement(cv::MORPH_ELLIPSE, cv::Size(3, 3));
    cv::Mat imageOpen;
    morphologyEx(imageThresh, imageOpen, cv::MORPH_OPEN, structuringElmt);
    cv::Mat imageClose;
    morphologyEx(imageOpen, imageClose, cv::MORPH_CLOSE, structuringElmt);

    // Now find contours.
    std::vector<std::vector<cv::Point>> contours;
    std::vector<cv::Vec4i> hierarchy;
    cv::findContours(imageClose, contours, hierarchy, cv::CHAIN_APPROX_NONE);

    // Analyze components and find CCCs.
    for (unsigned int i1 = 0; i1 < (int)contours.size(); i1++) {
        int i2 = hierarchy[i1][2];
        if (i2 < 0) continue; // See if it has a child inside

        // Compute centroids of inner and outer regions.
        cv::Moments mu1 = cv::moments(contours[i1]);
        cv::Point2d x1(mu1.m10 / mu1.m00, mu1.m01 / mu1.m00);
    }
```
cv::Moments mu2 = cv::moments(contours[i2]);
cv::Point2d x2(mu2.m10 / mu2.m00, mu2.m01 / mu2.m00);

// Check if centroids coincide.
if (norm(x1 - x2) > DPIXEL) continue;

// Check the "circularity" ratio of the outer region, which is
// the ratio of area to perimeter squared: R = 4*pi*A/P^2.
// R is 1 for a circle, and pi/4 for a square.
double P1 = arcLength(contours[i1], true);
double A1 = contourArea(contours[i1]);
if (4 * 3.1415*A1 / (P1*P1) < 3.1415 / 4)
    // Let's say that we want our region to be at least as round as a square.
    continue;

// This must be a valid target; add it to the output list.
targets.push_back(x1);
}

return targets;
Code for the function “orderTargets”:

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

// This function tries to find the 5-target pattern that looks like this
// 1 2 3
// 4 5
// It puts the targets in that order, and returns them.
std::vector<cv::Point2d> orderTargets(std::vector<cv::Point2d> allTargets)
{
    std::vector<cv::Point2d> orderedTargets;
    unsigned int i1, i2, i3, i4, i5;
    unsigned int nCCC = allTargets.size();

    // Find 3 CCCs that are in a line.
    double dMin = 1e9; // distance from a CCC to the midpt between points 1,3
    double d13 = 1; // the distance between points 1,3
    for (unsigned int i = 0; i < nCCC; i++) {
        for (unsigned int j = i + 1; j < nCCC; j++) {
            // Get the mid point between i,j.
            cv::Point2d midPt = (allTargets[i] + allTargets[j]) * 0.5;

            // Find the CCC that is closest to this midpoint.
            for (unsigned int k = 0; k < nCCC; k++) {
                if (k == i || k == j) continue;
                double d = norm(allTargets[k] - midPt); // distance from midpoint
                if (d < dMin) {
                    // This is the minimum found so far; save it.
                    dMin = d;
                    i1 = i;
                    i2 = k;
                    i3 = j;
                    d13 = norm(allTargets[i] - allTargets[j]);
                }
            }
        }
    }

    // If the best distance from the midpoint is < 30% of the distance between
    // the two other points, then we probably have our colinear set.
    if (dMin / d13 > 0.3) return orderedTargets; // return an empty list

    /*
    We have found 3 colinear targets: p1 -- p2 -- p3.
    Now find the one closest to p1; call it p4.
    */
    dMin = 1e9;
    for (unsigned int i = 0; i < nCCC; i++) {
        if (i != i1 && i != i2 && i != i3) {
            double d = norm(allTargets[i] - allTargets[i1]);
            if (d < dMin) {
                dMin = d;
                i4 = i;
            }
        }
    }
    if (dMin > 1e7) return orderedTargets; // return an empty list
```
/ * Now find the one closest to p3; call it p5. */
 double dMin = 1e9;
 for (unsigned int i = 0; i < nCCC; i++) {
     if (i != i1 && i != i2 && i != i3 && i != i4) {
         double d = norm(allTargets[i] - allTargets[i3]);
         if (d < dMin) {
             dMin = d;
             i5 = i;
         }
     }
 }
 if (dMin > 1e7) return orderedTargets;  // return an empty list

 // Now, check to see where p4 is with respect to p1,p2,p3. If the
 // signed area of the triangle p1-p3-p4 is negative, then we have
 // the correct order; ie
 //  1  2  3
 //  4  5
 // Otherwise we need to switch the order; ie
 //  3  2  1
 //  5  4

 // Signed area is the determinant of the 2x2 matrix [ p4-p1, p3-p1 ]
    cv::Vec2d p41 = allTargets[i4] - allTargets[i1];
    cv::Vec2d p31 = allTargets[i3] - allTargets[i1];
    double m[2][2] = { { p41[0], p31[0] }, { p41[1], p31[1] } };
    double det = m[0][0] * m[1][1] - m[0][1] * m[1][0];

    // Put the targets into the output list.
    if (det < 0) {
        orderedTargets.push_back(allTargets[i1]);
        orderedTargets.push_back(allTargets[i2]);
        orderedTargets.push_back(allTargets[i3]);
        orderedTargets.push_back(allTargets[i4]);
        orderedTargets.push_back(allTargets[i5]);
    }
    else {
        orderedTargets.push_back(allTargets[i3]);
        orderedTargets.push_back(allTargets[i2]);
        orderedTargets.push_back(allTargets[i1]);
        orderedTargets.push_back(allTargets[i5]);
        orderedTargets.push_back(allTargets[i4]);
    }

    return orderedTargets;