Mohamad Al jazaery
Dr. Guodong Guo
CS 691E - Computer Vision
3-16-2016

Robust Estimation for Circular Shape Detection Using RANSAC and Hough Transform

The Algorithm:

1- Normalize the image size to around 500 px using pyrUp and pyrDown. Sharp the image with 2D filter

2- Change the circle votes threshold until we find no more than 2 circles

Note : If votes threshold is bigger then we find less semicircle and more full circle.

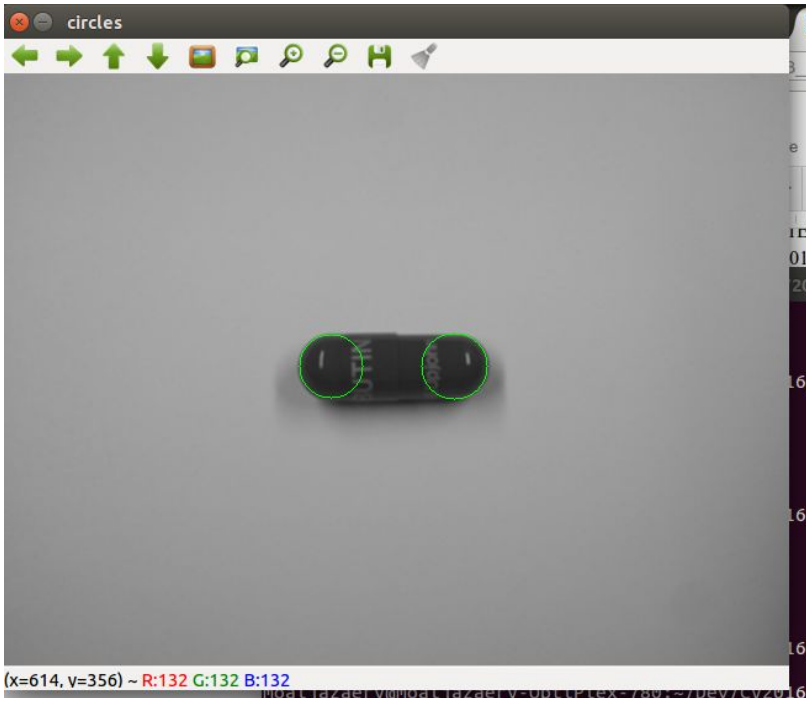
3 - While we have more than 2 circles increase the votes threshold and Run RANSAC with 100 000 iterations.

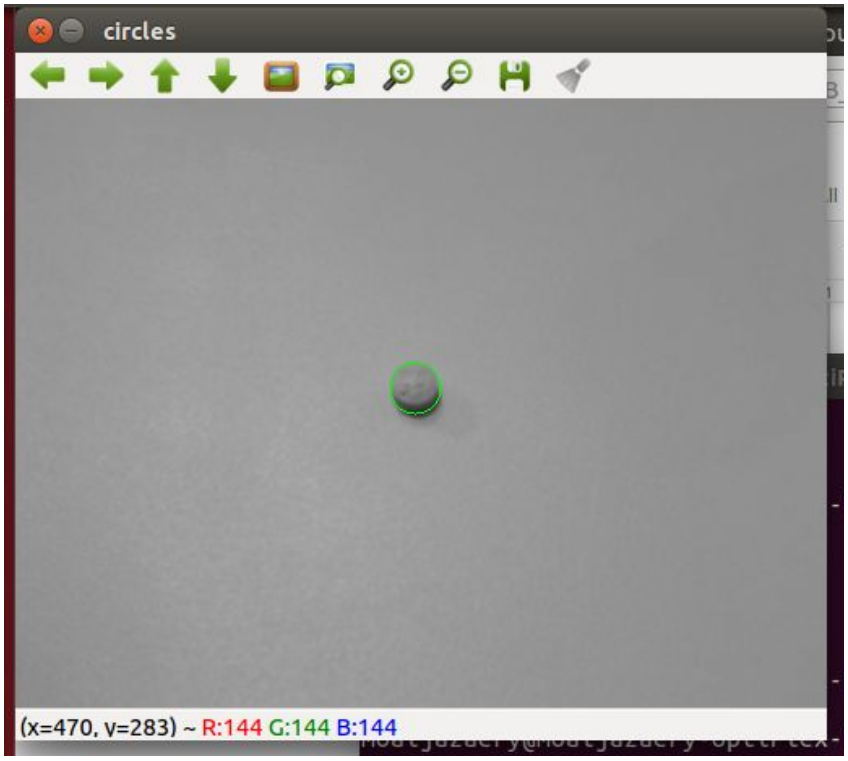
RANSAC Algorithm with Hough transform:

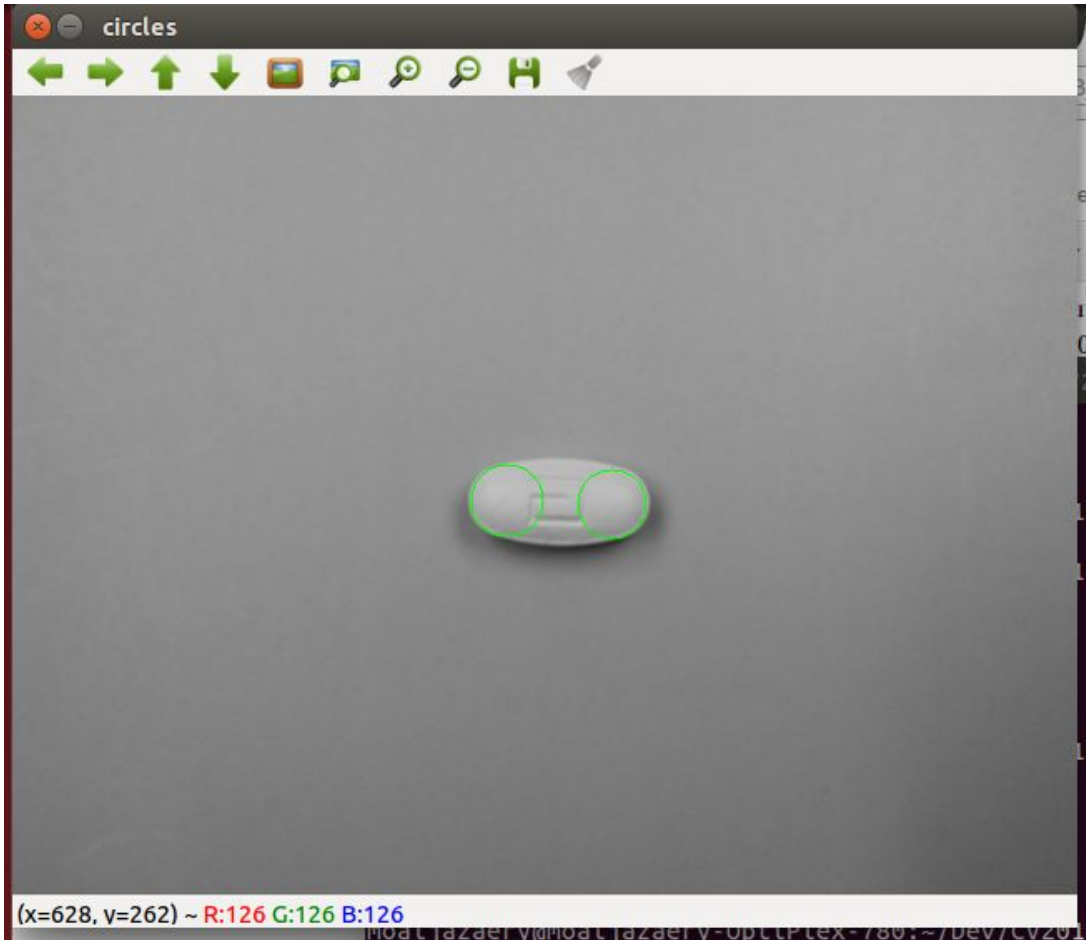
- **Find edge points using Canny edge detection . Canny_threshold=60 - is higher canny threshold, lower is set to canny_threshold / 2**
- **Choose 4 random points**
  - **Check if those points are not on the same line colinear_tolerance = 1**
  - **Check if the points are far enough from each other min_point_separation = 10**
- **Use 3 points to find the center and radius**
- **Check if the radius is greater than the minimum acceptable radius (15)**
- **Check if the 4th point is on the circle with reasonable error "radius_tolerance=1"**
- **Check how many points of the remaining points are on the circle with reasonable error "radius_tolerance=1"**

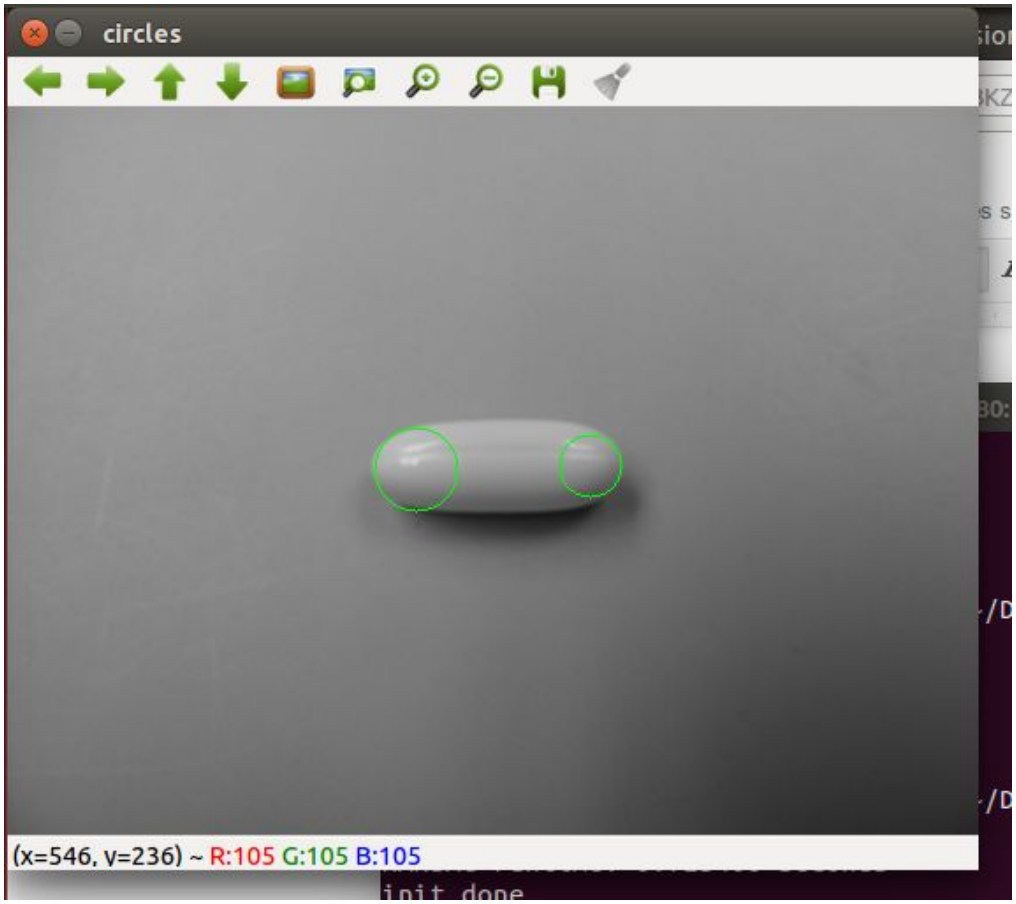
- **If the number of the votes is greater than circle votes threshold**
    - **Add circle to result**
    - **Remove points from the set so they can't vote on multiple circles**
- **Stop RANSAC if there are few points left points_threshold else go to next iteration**

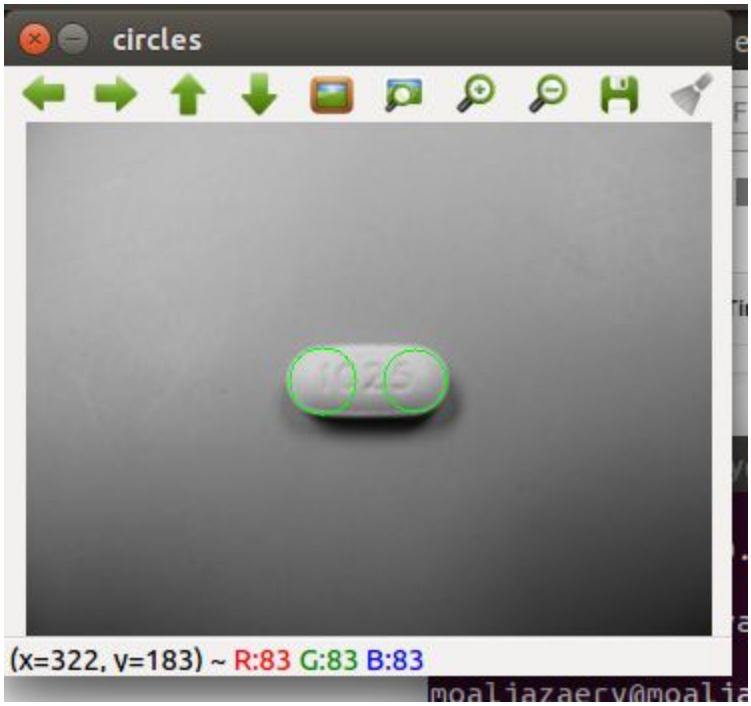
Results:

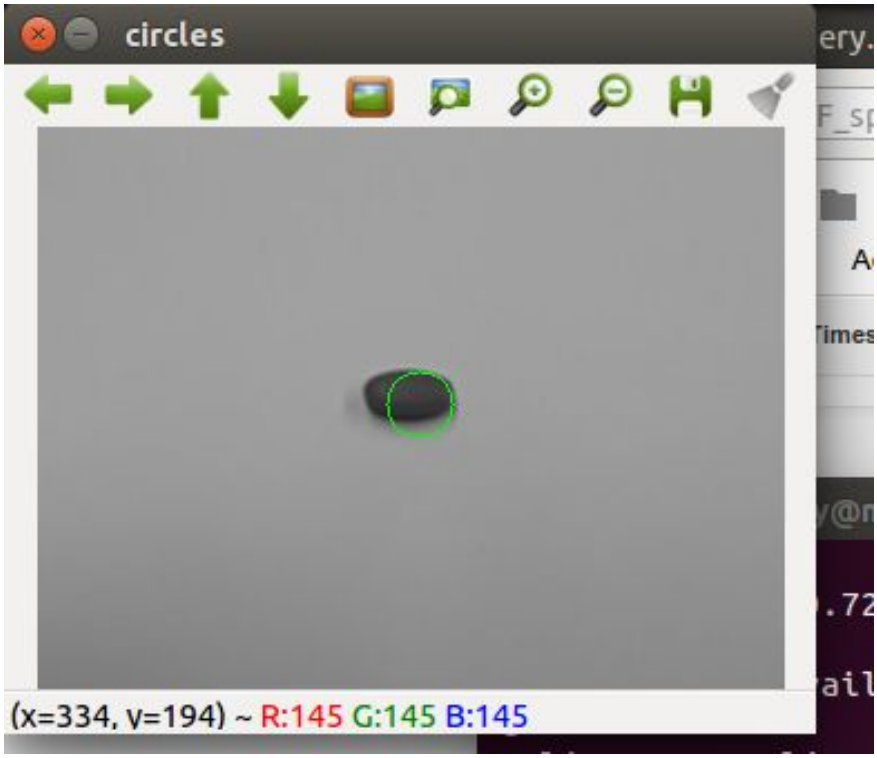


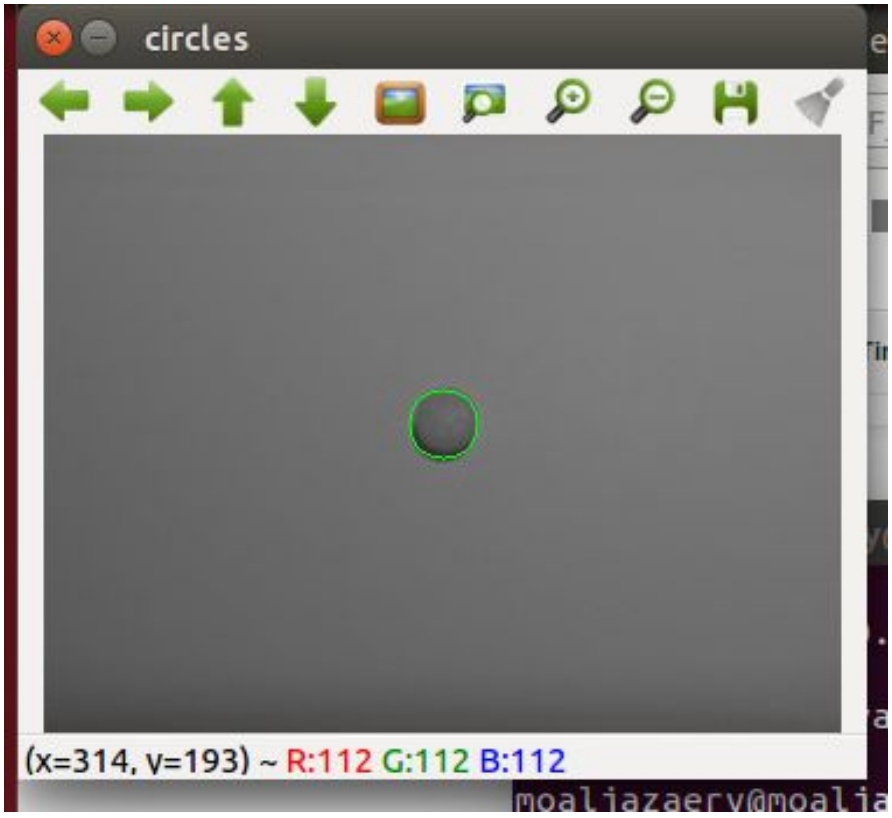

circles
(x=470, y=283) ~ R:144 G:144 B:144


circles
(x=628, y=262) ~ R:126 G:126 B:126


circles
(x=546, y=236) ~ R:105 G:105 B:105
init done


circles
(x=322, y=183) ~ R:83 G:83 B:83


circles
(x=334, y=194) ~ R:145 G:145 B:145


circles
(x=314, y=193) ~ R:112 G:112 B:112

CODE:

```
#include <opencv2/core/core.hpp>
#include <opencv2/imgproc/imgproc.hpp>
#include <opencv2/highgui/highgui.hpp>

#include <iostream>
#include <vector>
#include <string>

#include <time.h>

using namespace cv;
using namespace std;

// circleRANSAC
//
// input:
//		image - either CV_8UC1 or CV_8UC3
//		circles - return vector of Vec3f (x,y,radius)
//		canny_threshold - higher canny threshold, lower is set to canny_threshold / 2
//		circle_threshold - value between 0 and 1 for the percentage of the circle that needs to vote for it to be accepted
//		numIterations - the number of RANSAC loops, the function will quit if there is no points left in the set
//
void circleRANSAC(Mat &image, vector<Vec3f> &circles, double canny_threshold, double circle_threshold, int numIterations);
bool debug=false;
int main(int argc, char *argv[])
{

   // init with best result
   string filename = argv[1];
   double canny_threshold = 60;
   double circle_threshold = 0.4;
   int iterations = 100000;
   debug = bool(atof(argv[2]));

   Mat image = imread(filename,cv::IMREAD_GRAYSCALE);

   //scale images arround 500 px
   while(image.rows >500)
	pyrDown(image,image);
```

```
//Sharp the image
	Mat blurimage=image.clone();
	cv::GaussianBlur(image, blurimage, cv::Size(5, 5),2);
	cv::addWeighted(image, 1.5, blurimage, -0.5,0,image);
	cv::GaussianBlur(image, image, cv::Size(3, 3), 3,3);

vector<Vec3f> circles;

const clock_t start = clock();

//find semi Circles with 40% threshold
circle_threshold=0.4;
circleRANSAC(image, circles, canny_threshold, circle_threshold, iterations);

//change the threshold to find best 2 circles using RANSAC
while(int(circles.size()) > 2)
	{

	circleRANSAC(image, circles, canny_threshold, circle_threshold, iterations);

	circle_threshold+=0.1;

	}


	//Print Result
clock_t end = clock();

cout << "Found " << (int)circles.size() << " Circles." << endl;

double time = ((double)(end - start)) / (double)CLOCKS_PER_SEC;
std::cout << "RANSAC runtime: " << time << " seconds" << std::endl;

// Draw Circles
cvtColor(image,image,CV_GRAY2RGB);
for(int i = 0; i < (int)circles.size(); i++)
{
	 int x = circles[i][0];
	 int y = circles[i][1];
	 float rad = circles[i][2];

	 circle(image, Point(x,y), rad, Scalar(0,255,0));
}

imshow("circles", image);
waitKey();

return 0;
```

```
}

void circleRANSAC(Mat &image, std::vector<Vec3f> &circles, double canny_threshold, double circle_threshold,
int numIterations)
{
   CV_Assert(image.type() == CV_8UC1 || image.type() == CV_8UC3);
   circles.clear();

   // Edge Detection
   Mat edges;
   Canny(image, edges, MAX(canny_threshold/2,1), canny_threshold, 3);

   // Create point set from Canny Output
   std::vector<Point2d> points;
   for(int r = 0; r < edges.rows; r++)
   {
         for(int c = 0; c < edges.cols; c++)
         {
               if(edges.at<unsigned char>(r,c) == 255)
               {
                     points.push_back(cv::Point2d(c,r));
               }
         }
   }

   // 4 point objects to hold the random samples
   Point2d pointA;
   Point2d pointB;
   Point2d pointC;
   Point2d pointD;

   // distances between points
   double AB;
   double BC;
   double CA;
   double DC;

   // varibales for line equations y = mx + b
   double m_AB;
   double b_AB;
   double m_BC;
   double b_BC;

   // varibles for line midpoints
   double XmidPoint_AB;
   double YmidPoint_AB;
   double XmidPoint_BC;
   double YmidPoint_BC;
```

```
// variables for perpendicular bisectors
double m2_AB;
double m2_BC;
double b2_AB;
double b2_BC;

// RANSAC
cv::RNG rng;
int min_point_separation = 10; // change to be relative to image size?
int colinear_tolerance = 1; // make sure points are not on a line
int radius_tolerance = 1; // change to be relative to image size?
int points_threshold = 8; //should always be greater than 4
//double min_circle_separation = 10; //reject a circle if it is too close to a previously found circle
//double min_radius = 10.0; //minimum radius for a circle to not be rejected

int x,y;
Point2d center;
double radius;

// Iterate
for(int iteration = 0; iteration < numIterations; iteration++)
{
        //std::cout << "RANSAC iteration: " << iteration << std::endl;

        // get 4 random points
        pointA = points[rng.uniform((int)0, (int)points.size())];
        pointB = points[rng.uniform((int)0, (int)points.size())];
        pointC = points[rng.uniform((int)0, (int)points.size())];
        pointD = points[rng.uniform((int)0, (int)points.size())];

        // calc lines
        AB = norm(pointA - pointB);
        BC = norm(pointB - pointC);
        CA = norm(pointC - pointA);
        DC = norm(pointD - pointC);

        // one or more random points are too close together
        if(AB < min_point_separation || BC < min_point_separation || CA < min_point_separation || DC <
min_point_separation) continue;

        //find line equations for AB and BC
        //AB
        m_AB = (pointB.y - pointA.y) / (pointB.x - pointA.x + 0.000000001); //avoid divide by 0
        b_AB = pointB.y - m_AB*pointB.x;

        //BC
        m_BC = (pointC.y - pointB.y) / (pointC.x - pointB.x + 0.000000001); //avoid divide by 0
```

```
b_BC = pointC.y - m_BC*pointC.x;


//test colinearity (ie the points are not all on the same line)
if(abs(pointC.y - (m_AB*pointC.x + b_AB + colinear_tolerance)) < colinear_tolerance) continue;

//find perpendicular bisector
//AB
//midpoint
XmidPoint_AB = (pointB.x + pointA.x) / 2.0;
YmidPoint_AB = m_AB * XmidPoint_AB + b_AB;
//perpendicular slope
m2_AB = -1.0 / m_AB;
//find b2
b2_AB = YmidPoint_AB - m2_AB*XmidPoint_AB;

//BC
//midpoint
XmidPoint_BC = (pointC.x + pointB.x) / 2.0;
YmidPoint_BC = m_BC * XmidPoint_BC + b_BC;
//perpendicular slope
m2_BC = -1.0 / m_BC;
//find b2
b2_BC = YmidPoint_BC - m2_BC*XmidPoint_BC;

//find intersection = circle center
x = (b2_AB - b2_BC) / (m2_BC - m2_AB);
y = m2_AB * x + b2_AB;
center = Point2d(x,y);
radius = cv::norm(center - pointB);
if(radius<15) continue;

//check if the 4 point is on the circle
if(abs(cv::norm(pointD - center) - radius) > radius_tolerance) continue;

// vote
std::vector<int> votes;
std::vector<int> no_votes;
for(int i = 0; i < (int)points.size(); i++)
{
	double vote_radius = norm(points[i] - center);

	if(abs(vote_radius - radius) < radius_tolerance)
	{
		votes.push_back(i);
	}
	else
	{
```

```
                        no_votes.push_back(i);
                }
        }

        // check votes vs circle_threshold
        if( (float)votes.size() / (2.0*CV_PI*radius) >= circle_threshold )
        {
                circles.push_back(Vec3f(x,y,radius));

                // remove points from the set so they can't vote on multiple circles
                std::vector<Point2d> new_points;
                for(int i = 0; i < (int)no_votes.size(); i++)
                {
                        new_points.push_back(points[no_votes[i]]);
                }
                points.clear();
                points = new_points;
        }

        // stop RANSAC if there are few points left
        if((int)points.size() < points_threshold)
                break;
    }

    return;
}
```