Stop Detection:
Using Computer Vision Algorithms and OpenCV to Identify Stop Signs in Images

By Benjamin Carman
Spring 2019

A Final Project for CS 2981T
A Tutorial in Computer Vision
Dr. David Chelberg

Stop Detection:
Using Computer Vision Algorithms and OpenCV to Identify Stop Signs In Images

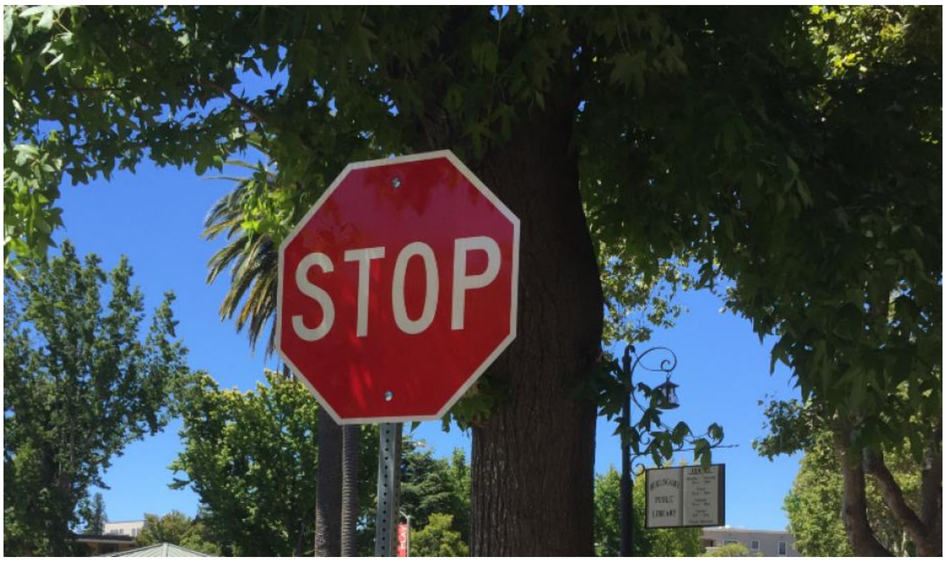
***Abstract*—Computer vision is a field that aims to help computers "see" and draw conclusions about the contents of digital images. This is an extremely important field in applications like self-driving vehicles which need to detect the signs of our current traffic system. This project aims to explore techniques that can make this happen using stop signs as an example. The techniques explored and employed here to detect stop signs can be easily transferred to most any other road sign. Some of these said techniques include line detection, corner detection, perspective transformations, and more. These techniques have been implemented in C++, using OpenCV as a resource. The implemented algorithms are successful in detecting many stop sign features, but various future improvements for robustness are also considered.**

Keywords—computer vision, OpenCV, sign detection, thresholding, perspective transformation, corner detection, line detection, strip trees

I. Introduction

Computer vision is a field that aims to use computers to draw conclusions and understandings from digital images. Ultimately, it aims to help computers "see" so that some sort of analysis can be performed on these images—which to a computer, are simply matrices of various pixels and their associated color values. In this project, computer vision is used for the simple purpose of detecting stop signs in images. This is an extremely relevant application as self-driving vehicles are positioned to play a significant role in the future of transportation. As our traffic system exists today, these vehicles need to be able to detect when and where a stop sign is present in order to obey traffic laws and avoid a collision. This goes for all other road signs as well—of which the techniques used in this project could be directly applied under slight modification. As this project aims to ultimately determine whether a stop sign is present in an image, a number of computer vision and image processing algorithms are considered in the process. Some of these include: Thresholding, finding connected components, eliminating noise, boundary detection, corner detection, perspective transformations, and line detection using both strip trees and the Hough line transform.

As we consider each of these operations, examples will be provided based on the following input image:

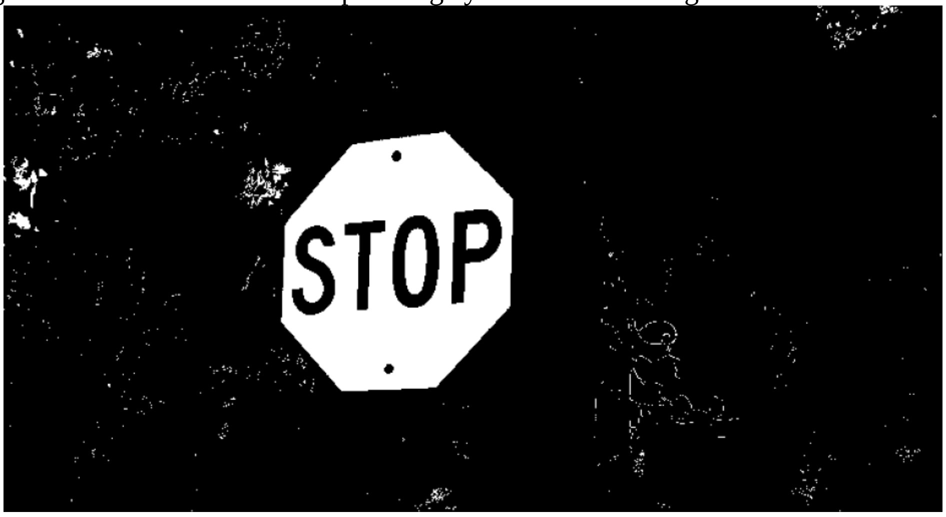


II. Thresholding

First and foremost, this project uses thresholding to simplify the input image and pull out candidate regions for a stop sign based on a color range similar to that of a stop sign. First the image was converted from being represented by blue, green, and red values (BGR) to being represented by hue, saturation, and value (HSV). It was found through empirical testing that thresholding on the image in HSV produced far more refined results. The actual operation of thresholding itself was simple and already in the OpenCV library where the pixels that matched a predetermined range of HSV values were made fully white, and the rest were made black. The optimal HSV values for thresholding on a red stop sign were experimentally found to be: Hue (0 - 179), Saturation (159 - 255), and Value (93 - 183). An excerpt of the code to perform this operation is as follows:

```
//Initialize HSV image
cvtColor(r.bgr_img, r.hsv_img, COLOR_BGR2HSV);

//Threshold for stop sign on HSV image
inRange(r.hsv_img, Scalar(LOW_H, LOW_S, LOW_V), Scalar(HIGH_H, HIGH_S, HIGH_V),
        r.hsv_img_thresholded);
```

Performing the above code on our example image yields the following:

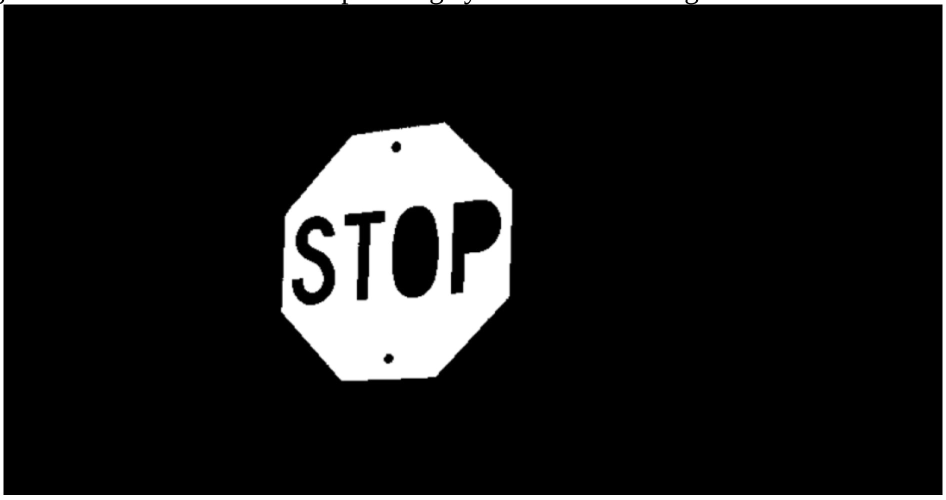


III. Removing Noise with Connected Components

After thresholding with the above values, we are left with an image that has a clear stop sign popping out to the naked eye, but also a great deal of noise due to other areas of the image that had similar HSV values. These areas will make the later parts of the stop sign detection nearly impossible, so they need to be removed. This can be easily done using the connectedComponentsWithStats() function in OpenCV. This function will label all the connected regions of the image and we can then simply loop through every pixel in the image and only keep the ones that are a part of a connected region of a size that we would expect of a stop sign. An excerpt of code to do just this is as follows and thresholds for regions of a size between 2000 and 30000:

```
//Get connected region
Mat labels, stats, centroids;
int num_components = connectedComponentsWithStats(r.hsv_img_thresholded, labels,
                                                  stats, centroids, 4);

r.noise_free = Mat::zeros(r.hsv_img_thresholded.rows, r.hsv_img_thresholded.cols,
                          CV_8U);
for(int i = 0; i < r.noise_free.rows; i++) {
  for(int j = 0; j < r.noise_free.cols; j++) {
    int label = labels.at<int>(i,j);
    if (stats.at<int>(label, CC_STAT_AREA) > 2000 &&
        stats.at<int>(label, CC_STAT_AREA) < 30000)
    {
      r.noise_free.at<unsigned char>(i,j) = 255;
    }
  }
}
```

Performing the above code on our example image yields the following:



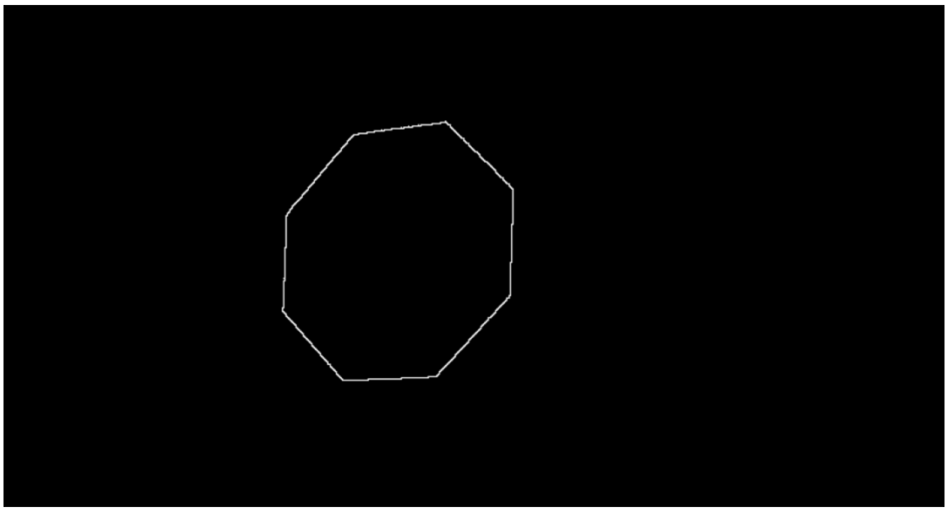
IV. Boundary Detection

The next step in determining that this region of the image is in fact a stop sign, is to get the boundary of the region for further processing (i.e. corner points, boundary lines, etc). This is done by going through each pixel in the image until an edge pixel is found. This is defined by a white pixel that is bordered by a black pixel on one of its sides. After the edge pixel is found, we can walk around the boundary of the region by visiting all the other edge pixels connected to the original. Doing this produces a set of points which form the boundary of the region. In this project, the code to do this is as follows:

```
bool isEdge(Mat &image, int i, int j);
void drawBoundary(Mat &source, Mat &target, int i, int j, std::vector<Point>
&boundary_points);

//In the main - Get stop sign boundary
  r.boundary = Mat::zeros(r.noise_free.rows, r.noise_free.cols, CV_8U);

  std::vector<Point> boundary_points;

  //Find a Boundary
  bool done = false;
  for(int i = 0; i < r.noise_free.rows; i++) {
    if (done) break;
    for(int j = 0; j < r.noise_free.cols; j++) {
      if (isEdge(r.noise_free, i, j))
      {
        drawBoundary(r.noise_free, r.boundary, i, j, boundary_points);
        done = true;
        break;
      }
    }
  }
```

```
//Outside of main
bool isEdge(Mat &image, int i, int j)
{
  if (image.at<unsigned char>(i,j) == 255)
  {
    if (i > 0 && j > 0 && image.at<unsigned char>(i-1,j-1) == 0) return true;
    if (i > 0 && image.at<unsigned char>(i-1,j) == 0) return true;
    if (i > 0 && j < image.cols && image.at<unsigned char>(i-1,j+1) == 0)
        return true;
    if (j > 0 && image.at<unsigned char>(i,j-1) == 0) return true;
    if (j < image.cols && image.at<unsigned char>(i,j+1) == 0) return true;
    if (i < image.rows && j > 0 && image.at<unsigned char>(i+1,j-1) == 0)
        return true;
    if (i < image.rows && image.at<unsigned char>(i+1,j) == 0) return true;
    if (i < image.rows && j < image.cols && image.at<unsigned char>(i+1,j+1) == 0)
        return true;
  }

  return false;
}

void drawBoundary(Mat &source, Mat &target, int i, int j, std::vector<Point>
                  &boundary_points)
{
  Point p;
  p.x = j;
  p.y = i;
  boundary_points.push_back(p);
  target.at<unsigned char>(i,j) = 255;
  if (isEdge(source, i - 1, j) && target.at<unsigned char>(i - 1,j) == 0)
        drawBoundary(source, target, i - 1, j, boundary_points);
  if (isEdge(source, i - 1, j + 1) && target.at<unsigned char>(i - 1, j + 1) == 0)
        drawBoundary(source, target, i - 1, j + 1, boundary_points);
  if (isEdge(source, i, j + 1) && target.at<unsigned char>(i, j + 1) == 0)
        drawBoundary(source, target, i, j + 1, boundary_points);
  if (isEdge(source, i + 1, j + 1) && target.at<unsigned char>(i + 1, j + 1) == 0)
        drawBoundary(source, target, i + 1, j + 1, boundary_points);
  if (isEdge(source, i + 1, j) && target.at<unsigned char>(i + 1, j) == 0)
        drawBoundary(source, target, i + 1, j, boundary_points);
  if (isEdge(source, i + 1, j - 1) && target.at<unsigned char>(i + 1, j - 1) == 0)
        drawBoundary(source, target, i + 1, j - 1, boundary_points);
  if (isEdge(source, i, j - 1) && target.at<unsigned char>(i, j - 1) == 0)
        drawBoundary(source, target, i, j - 1, boundary_points);
  if (isEdge(source, i - 1, j - 1) && target.at<unsigned char>(i - 1, j - 1) == 0)
        drawBoundary(source, target, i - 1, j - 1, boundary_points);
}
```

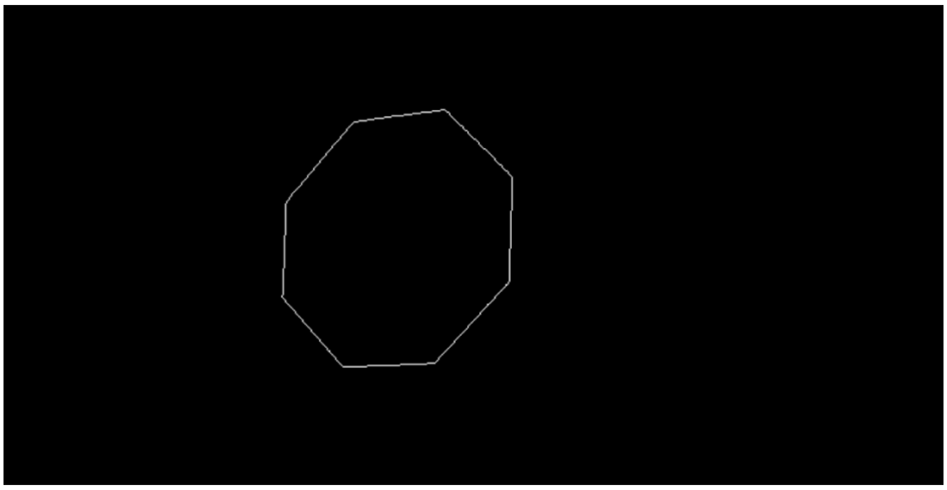
Performing the above code on our example image yields the following:



The next step after obtaining this boundary is to thin it to make sure it is no more than a pixel thick all the way around. This will allow for more accurate calculations when determining corners and lines by not having additional stray pixels along the boundary. This algorithm works by thinning from the top, right, bottom, and left until no changes can be made. It chooses pixels to be thinned if they exist along the respective boundary and deleting them does not change the connectedness of the boundary with respect to 8-connectivity (i.e. every pixel along the border must maintain the invariant that it is bordered by another pixel in one of the 8 directions surrounding that pixel). The code written to this is shown below:

```
bool changesConnectivity(Mat &image, int i, int j);

//IN THE MAIN -- Generate thinned boundary
r.thinned_boundary = r.boundary.clone();
thin(r.thinned_boundary);

Mat dummy = Mat::zeros(r.thinned_boundary.rows, r.thinned_boundary.cols, CV_8U);
std::vector<Point> thinned_boundary_points;

//Find thinned boundary points
done = false;
for(int i = 0; i < r.thinned_boundary.rows; i++) {
  if (done) break;
  for(int j = 0; j < r.thinned_boundary.cols; j++) {
    if (isEdge(r.thinned_boundary, i, j))
    {
      drawBoundary(r.thinned_boundary, dummy, i, j, thinned_boundary_points);
      done = true;
      break;
    }
  }
}
//END MAIN

void thin(Mat &image)
{
  bool was_changed = true;
  int count = 0;
  while (was_changed)
  {
    was_changed = false;
    //Thin from top
    for (int j = 0; j < image.cols; j++)
    {
      for (int i = 0; i < image.rows; i++)
      {
        if (image.at<unsigned char>(i,j) == 255)
        {
          count++;
          if (!changesConnectivity(image, i, j))
          {
            image.at<unsigned char>(i,j) = 0;
            was_changed = true;
          }
          break;
        }
      }
    }

    //Thin from right
    for (int i = 0; i < image.rows; i++)
    {
      for (int j = image.cols - 1; j >= 0; j--)
      {
        if (image.at<unsigned char>(i,j) == 255)
        {
          if (!changesConnectivity(image, i, j))
          {
            image.at<unsigned char>(i,j) = 0;
            was_changed = true;
          }
          break;
        }
      }
    }

    //Thin from bottom
    for (int j = 0; j < image.cols; j++)
    {
      for (int i = image.rows - 1; i >= 0; i--)
      {
        if (image.at<unsigned char>(i,j) == 255)
        {
          if (!changesConnectivity(image, i, j))
          {
            image.at<unsigned char>(i,j) = 0;
            was_changed = true;
          }
          break;
        }
      }
    }

    //Thin from left
    for (int i = 0; i < image.rows; i++)
    {
      for (int j = 0; j < image.cols; j++)
      {
        if (image.at<unsigned char>(i,j) == 255)
        {
          if (!changesConnectivity(image, i, j))
          {
            image.at<unsigned char>(i,j) = 0;
            was_changed = true;
          }
          break;
        }
      }
    }
  }
}

bool changesConnectivity(Mat &image, int i, int j)
{
  int num_neighbors = 0;
  Mat local_area = Mat::zeros(3,3, CV_8U);

  for (int n = 0; n < 3; n++)
  {
    for (int m = 0; m < 3; m++)
    {
      //Keep the center a 0
      if (!(n == 1 && m == 1))
      {
        local_area.at<unsigned char>(n,m) = image.at<unsigned char>(i - 1 + n,
                                                                     j - 1 + m);

        if (local_area.at<unsigned char>(n,m) == 255)
        {
          num_neighbors++;
        }
      }
    }
  }

  if (num_neighbors >= 1)
  {
    Mat labels;
    int num_components = connectedComponents(local_area, labels, 8);
    if (num_components == 2) //Only a foreground and background
    {
      return false;
    }
  }
  return true;
}
```

Performing the above code on our example image yields the following:

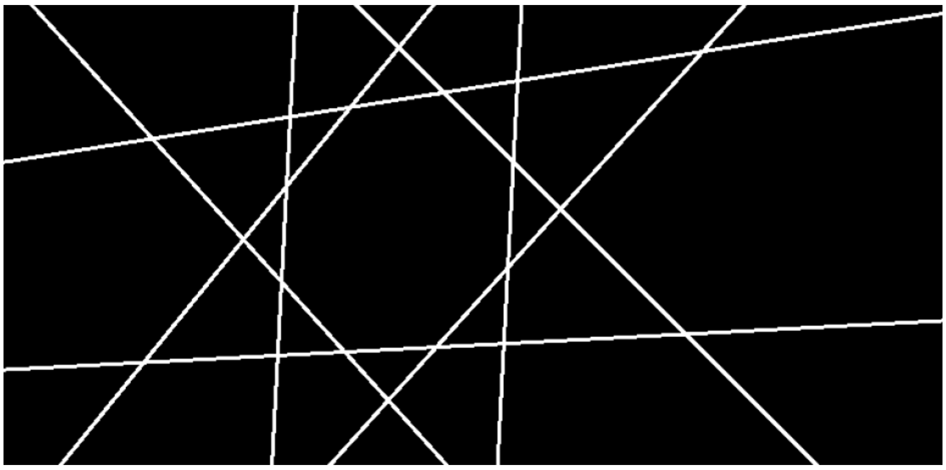


V. Line Detection

After obtaining a thinned boundary of the stop sign, we can obtain the bordering lines of the stop sign. There are a number of ways to do this, and two primary methods are considered here. One, is to use OpenCV's built in Hough Line Transform method. The following code was written to employ this OpenCV method and it essentially calls the HoughLines() method to get the boundary lines and then chooses the 8 unique lines we would expect from an octagonal stop sign:

```
//Generate boundary lines using Hough Lines
int line_length = 25;

std::vector<Vec2f> lines;
std::vector<Vec2f> unique_lines;
HoughLines(r.boundary, lines, 1, 3*CV_PI/180, line_length);

int num_unique = 0;
for (size_t i = 0; i < lines.size(); i++)
{
  if (is_unique(lines[i], unique_lines))
  {
    unique_lines.push_back(lines[i]);
    num_unique++;
  }

  if (num_unique == 8) break;
}

r.final = r.boundary.clone();
for (size_t i = 0; i < num_unique; i++)
  {
    float rho = unique_lines[i][0], theta = unique_lines[i][1];
    //std::cout << rho << " " << theta << std::endl;
    Point pt1, pt2;
    double a = std::cos(theta), b = std::sin(theta);
    double x0 = a*rho, y0 = b*rho;
    pt1.x = cvRound(x0 + 2000*(-b));
    pt1.y = cvRound(y0 + 2000*(a));
    pt2.x = cvRound(x0 - 2000*(-b));
    pt2.y = cvRound(y0 - 2000*(a));
    line(r.final, pt1, pt2, Scalar(255,0,0), 2);
  }
```

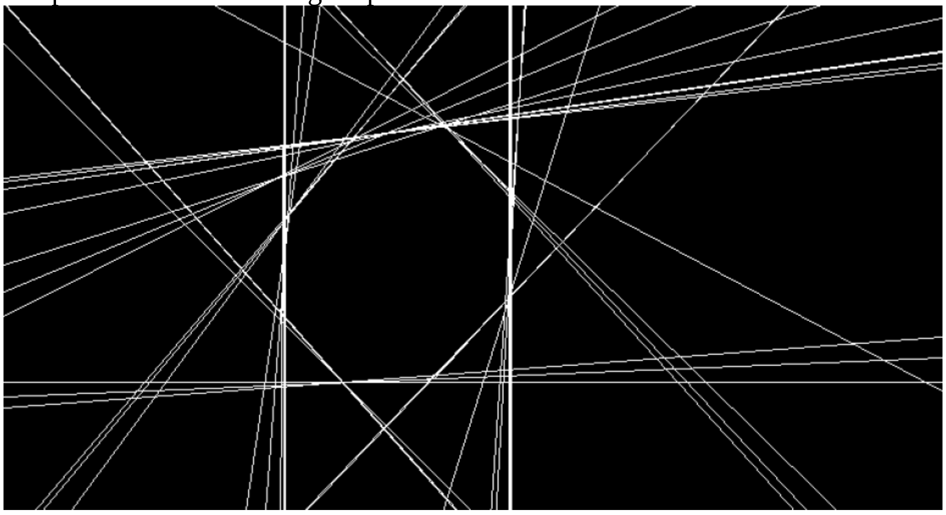
Using HoughLines produces the following result:



There are also other good algorithms for finding boundary lines more manually than OpenCV’s HoughLines method. One such algorithm is by using strip trees. This method involves the use of OpenCV’s fitline() method to fit a line using least squares to the boundary region. The boundary region is then split where the line intersects the region. These intersections are found by mapping the boundary points onto the line using the dot product. After the split is made, the strip tree algorithm proceeds recursively by fitting a line to each separated region of the boundary. The point in the region that is furthest from the line perpendicularly marks the next split. This point is found by mapping all the boundary points of the region onto the line perpendicular to the least square fit line. These splits continue until a line is found where every point is within some threshold of distance from the line. The code to implement this algorithm is shown below:

```
std::vector<std::vector<Point> > split_at_intersection_points(std::vector<Point>
&boundary_points, Vec4f &major, Mat &image);

void split_at_extreme_points(std::vector<std::vector<Point> > &boundary_lines,
const std::vector<Point> &boundary_section, Mat &image);

std::vector<std::vector<Point> > get_lines(std::vector<Point> &boundary_points,
Mat &image);

//In Main -- Generate boundary lines using strip trees
r.strip_tree = r.thinned_boundary.clone();
std::vector<std::vector<Point> > boundary_lines =
get_lines(thinned_boundary_points, r.strip_tree);
//END Main

std::vector<std::vector<Point> > get_lines(std::vector<Point> &boundary_points,
Mat &image)
{
  Vec4f major;
  cv::fitLine(boundary_points, major, cv::DIST_L2, 1, 0.001, 0.001);

  std::vector<std::vector<Point> > split_boundary =
        split_at_intersection_points(boundary_points, major, image);

  std::vector<std::vector<Point> > boundary_lines;
  split_at_extreme_points(boundary_lines, split_boundary[0], image);
  split_at_extreme_points(boundary_lines, split_boundary[1], image);

  for (int i = 0; i < boundary_lines.size(); i++)
  {
    Vec4f major;
    cv::fitLine(boundary_lines[i], major, cv::DIST_L2, 1, 0.001, 0.001);

    Point pmajor1, pmajor2, pminor1, pminor2;
    pmajor1.x = major[2] - 2000 * major[0];
    pmajor1.y = major[3] - 2000 * major[1];
    pmajor2.x = major[2] + 2000 * major[0];
    pmajor2.y = major[3] + 2000 * major[1];

    cv::line(image, pmajor1, pmajor2, Scalar(255,0,0), 1);
  }

  return boundary_lines;
}


std::vector<std::vector<Point> > split_at_intersection_points(std::vector<Point>
&boundary_points, Vec4f &major, Mat &image)
{
  std::vector<std::vector<Point> > split_boundary;

  //Get fitted line vector direction
  double vx = major[0];
  double vy = major[1];

  //Get origin fitted vector is respect to for line
  double ox = major[2];
  double oy = major[3];

  int intersection1;
  int intersection2;
  double min = 1000;
  double max = -1000;
  for (int i = 0; i < boundary_points.size(); i++)
  {
    //Get vector to current point from origin
    double px = boundary_points[i].x - ox;
    double py = boundary_points[i].y - oy;

    //Normalize vector
    double length = sqrt(px * px + py * py);
    px /= length;
    py /= length;

    double dot_product = px *  vx + py * vy;
    if (dot_product < min)
    {
      intersection1 = i;
      min = dot_product;
    }
    if (dot_product > max)
    {
      intersection2 = i;
      max = dot_product;
    }
  }

  //Generate separate vectors of points and put into split_boundary
  if (intersection1 < intersection2)
  {
    std::vector<Point> set1 (boundary_points.begin() + intersection1,
                             boundary_points.begin() + intersection2);

    std::vector<Point> set2 (boundary_points.begin() + intersection2,
                             boundary_points.end());
    for (int i = 0; i < intersection1; i++)
    {
      set2.push_back(boundary_points[i]);
    }

    split_boundary.push_back(set1);
    split_boundary.push_back(set2);
  }
  else
  {
    std::vector<Point> set1 (boundary_points.begin() + intersection2,
                             boundary_points.begin() + intersection1);

    std::vector<Point> set2 (boundary_points.begin() + intersection1,
                             boundary_points.end());
    for (int i = 0; i < intersection2; i++)
    {
      set2.push_back(boundary_points[i]);
    }

    split_boundary.push_back(set1);
    split_boundary.push_back(set2);
  }
  return split_boundary;
}

void split_at_extreme_points(std::vector<std::vector<Point> > &boundary_lines,
const std::vector<Point> &boundary_section, Mat &image)
{
  if (boundary_section.size() < 9)
  {
    boundary_lines.push_back(boundary_section);
    return;
  }

  Vec4f major;

  cv::fitLine(boundary_section, major, cv::DIST_L2, 1, 0.001, 0.001);

  Point pmajor1, pmajor2, pminor1, pminor2;
  pmajor1.x = major[2] - 2000 * major[0];
  pmajor1.y = major[3] - 2000 * major[1];
  pmajor2.x = major[2] + 2000 * major[0];
  pmajor2.y = major[3] + 2000 * major[1];

//Get fitted line vector direction PERPENDICULAR
  double vx = major[1] * -1;
  double vy = major[0];

  //Get origin fitted vector is respect to for line
  double ox = major[2];
  double oy = major[3];

  std::list<std::pair<double,int> > dot_products;
  for (int i = 0; i < boundary_section.size(); i++)
  {
    //Get vector to current point from origin
    double px = boundary_section[i].x - ox;
    double py = boundary_section[i].y - oy;

    //Normalize vector
    double length = sqrt(px * px + py * py);
    px /= length;
    py /= length;

    double dot_product = px *  vx + py * vy;
    std::pair<double,int> current_dot_prod (abs(dot_product),i);
    dot_products.push_back(current_dot_prod);
  }

  dot_products.sort();

  if (dot_products.back().first < 0.8)
  {
    boundary_lines.push_back(boundary_section);
    return;
  }
  else
  {
    int split_location = dot_products.back().second;
    std::vector<Point> set1 (boundary_section.begin(),
                             boundary_section.begin() + split_location);
    std::vector<Point> set2 (boundary_section.begin() + split_location,
                             boundary_section.end());

    split_at_extreme_points(boundary_lines, set1, image);
    split_at_extreme_points(boundary_lines, set2, image);
  }
}
```

This code then produces the following output of lines:

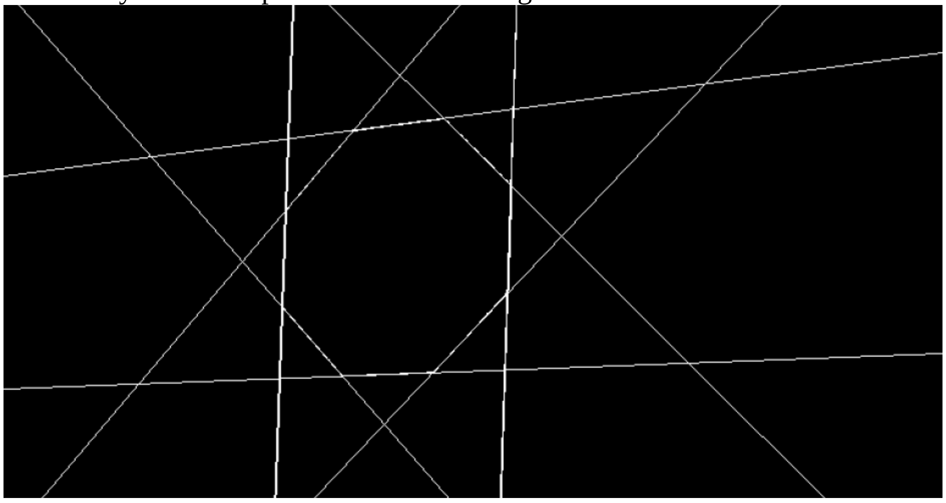


Finally, we proceed by merging lines with similar slopes to leave us with only those lines unique to each edge of the stop sign. This is done using the following code:

```
void merge_lines(std::vector<std::vector<Point> > &boundary_lines, Mat &image);
```

```
//In Main -- Generate merged boundary lines
r.merged_strip_tree = r.thinned_boundary.clone();
merge_lines(boundary_lines, r.merged_strip_tree);
//END Main

void merge_lines(std::vector<std::vector<Point> > &boundary_lines, Mat &image)
{
  for (int i = 0; i < boundary_lines.size() - 1; i++)
  {
    if (boundary_lines[i].size() == 0) continue;
    Vec4f line1;
    cv::fitLine(boundary_lines[i], line1, cv::DIST_L2, 1, 0.001, 0.001);
    double vx1 = line1[0];
    double vy1 = line1[1];
    double ox1 = line1[2];
    double oy1 = line1[3];

    for (int j = i + 1; j < boundary_lines.size(); j++)
    {
      if (boundary_lines[j].size() == 0) continue;
      Vec4f line2;
      cv::fitLine(boundary_lines[j], line2, cv::DIST_L2, 1, 0.001, 0.001);
      double vx2 = line2[0];
      double vy2 = line2[1];
      double ox2 = line2[2];
      double oy2 = line2[3];
      double dvx = abs(vx2 - vx1);
      double dvy = abs(vy2 - vy1);
      double dox = abs(ox2 - ox1);
      double doy = abs(oy2 - oy1);

      if (dvx < 0.5 && dvy < 0.5 && dox < 65 && doy < 70)
      {
        for (int k = 0; k < boundary_lines[j].size(); k++)
        {
          boundary_lines[i].push_back(boundary_lines[j][k]);
        }
        boundary_lines[j].clear();
      }
    }
  }

  for (int i = 0; i < boundary_lines.size(); i++)
  {
    if (boundary_lines[i].size() == 0) continue;

    Vec4f major;
    cv::fitLine(boundary_lines[i], major, cv::DIST_L2, 1, 0.001, 0.001);

    Point pmajor1, pmajor2, pminor1, pminor2;
    pmajor1.x = major[2] - 2000 * major[0];
    pmajor1.y = major[3] - 2000 * major[1];
    pmajor2.x = major[2] + 2000 * major[0];
    pmajor2.y = major[3] + 2000 * major[1];

    cv::line(image, pmajor1, pmajor2, Scalar(255,0,0), 1);
  }
}
```

Merging the boundary lines then produces the following result:

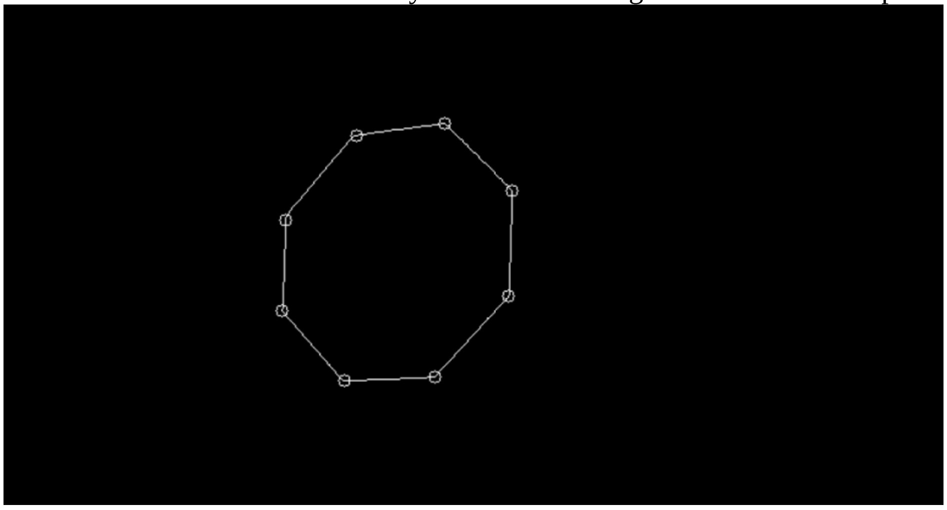


VI. Corner Detection

Similar to line detection, given the boundary points of the stop sign, corners can also be detected. Naturally, corners should occur at the points of the boundary that have the greatest change in slope. Specifically, taking the derivative across the points of the boundary should give the slope around the boundary and taking the second derivative should yield the changes in slope around the boundary. By simply walking the boundary and computing the slope at each point given its close neighbors and then by iterating through each of the slopes we can compute this second derivative and choose the points on the boundary where this change in slope is the greatest, i.e. the corner points. One key factor to note is that when calculating the slope at each location, one must use angular slope with the atan2() function in order to avoid bias. Thus, calculating slope is as simple as fitting a line and using the vector direction in the atan2 function to get angular slope. Then computing the difference in slope we had to take into consideration wrap around at -pi/2 and pi/2 so that these two angles weren't considered different slopes. An implementation that does just this is given below:

```
double get_difference(double angle1, double angle2);

void get_corners(std::vector<Point> boundary_points, std::vector<Point>
&corner_points, Mat &image, double threshold, int buffer);

//IN THE MAIN -- Find boundary corners
r.boundary_corners = r.thinned_boundary.clone();
std::vector<Point> corner_points;
get_corners(thinned_boundary_points, corner_points, r.boundary_corners, 0.09, 5);
//END MAIN

double get_difference(double angle1, double angle2)
{
  double difference1 = abs(angle1 - angle2);

  //If angles are in different quadrants
  if (angle1 * angle2 < 0)
  {
    double difference2;
    if (angle1 < 0)
    {
      difference2 = (CV_PI/2.0 + angle1) + (CV_PI/2.0 - angle2);
      return min(difference1, difference2);
    }
    else
    {
      difference2 = (CV_PI/2.0 + angle2) + (CV_PI/2.0 - angle1);
      return min(difference1, difference2);
    }
  }
  return difference1;
}

void get_corners(std::vector<Point> boundary_points, std::vector<Point>
&corner_points, Mat &image, double threshold, int buffer)
{
  std::vector<double> slopes;

  //Get all out slopes from orgigin +- 6 pixels
  for (size_t i = 0; i < boundary_points.size(); i++)
  {
    if (i < buffer)
    {
      std::vector<Point> pts (boundary_points.begin(),
                              boundary_points.begin() + (i + buffer));

      for (int j = boundary_points.size() - (buffer - i);
           j < boundary_points.size(); j++)
      {
        pts.push_back(boundary_points[j]);
      }

      Vec4f line; //use l2
      fitLine(pts, line, cv::DIST_L2, 1, 0.01, 0.01);

      double slope = atan2(line[1], line[0]);

      slopes.push_back(slope);
    }
    else if (i >= boundary_points.size() - buffer)
    {
      std::vector<Point> pts (boundary_points.begin() + (i - buffer),
                              boundary_points.end());

      for (int j = 0; j < (buffer + 1) - (boundary_points.size() - i); j++)
      {
        pts.push_back(boundary_points[j]);
      }

      Vec4f line;
      fitLine(pts, line, cv::DIST_L2, 1, 0.01, 0.01);

      double slope = atan2(line[1], line[0]);

      slopes.push_back(slope);
    }
    else
    {
    std::vector<Point> pts (boundary_points.begin() + (i - buffer),
                            boundary_points.begin() + (i + buffer));

    Vec4f line;
    fitLine(pts, line, cv::DIST_L2, 1, 0.01, 0.01);

    double slope = atan2(line[1], line[0]);

    slopes.push_back(slope);
    }
  }

  //Compute slope changes (gradient)
  std::vector<double> slope_change;

  for (int i = 0; i < slopes.size() - 1; i++)
  {
    double change = get_difference(slopes[i+1], slopes[i]);
    slope_change.push_back(change);
  }
  slope_change.push_back(get_difference(slopes[0], slopes[slopes.size() - 1]));

  std::vector<std::pair<Point,double> > corner_candidates;
  for (int i = 0; i < slope_change.size(); i++)
  {
    if (slope_change[i] >= threshold)
    {
      corner_candidates.push_back(std::make_pair(boundary_points[i],
                                                 slope_change[i]));
    }
  }

  std::vector<bool> checked (corner_candidates.size(), false);

  for (int i = 0; i < corner_candidates.size(); i++)
  {
    if (checked[i]) continue;
    checked[i] = true;

    double max = corner_candidates[i].second;
    int index = i;

    for (int j = 0; j < corner_candidates.size(); j++)
    {
      if (checked[j]) continue;

      if (abs(corner_candidates[j].first.x - corner_candidates[i].first.x) < 15
         && abs(corner_candidates[j].first.y - corner_candidates[i].first.y) < 15)
      {
        checked[j] = true;

        if (corner_candidates[j].second > max)
        {
          max = corner_candidates[j].second;
          index = j;
        }
      }
    }

    corner_points.push_back(corner_candidates[index].first);
    circle(image, corner_candidates[index].first, 5, Scalar(255, 0, 0));
  }
}
```

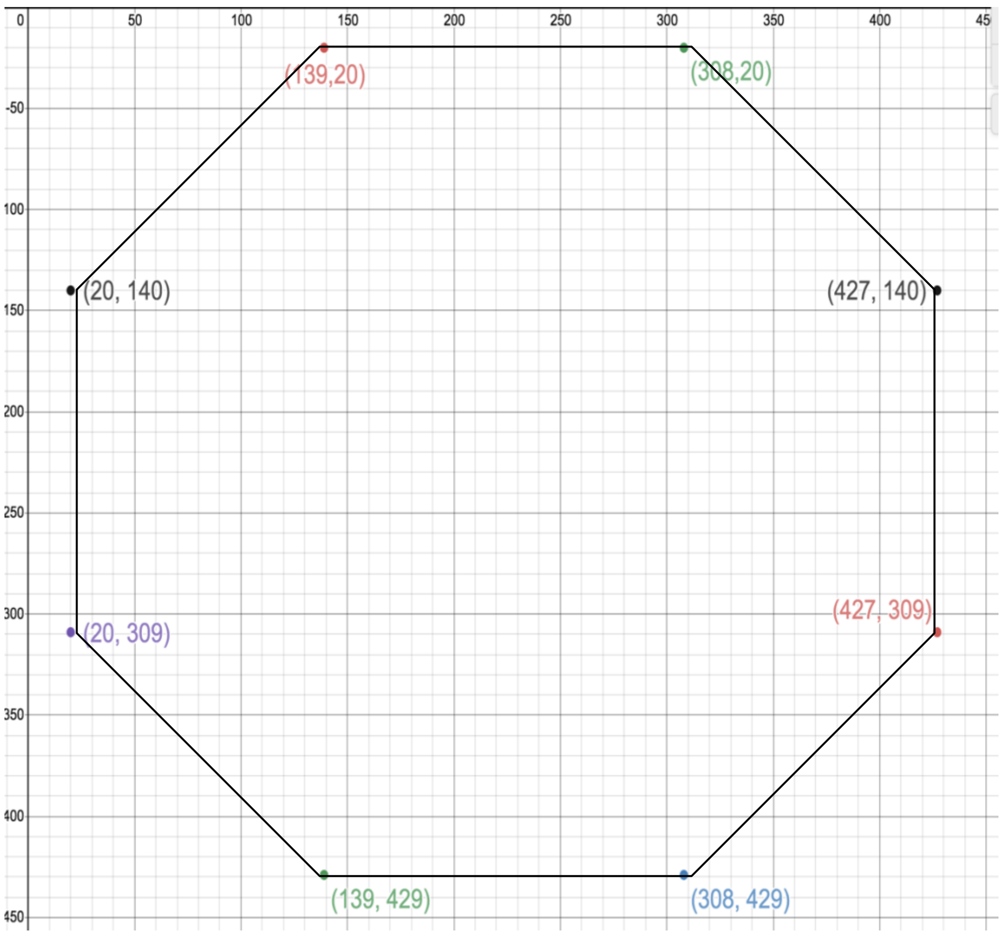
This implementation of corner detection then yields the following results on our example image:



VII. Perspective Transformation

Now that we have done all of this image processing and computer vision analysis to obtain the corner points of the region, the most important thing left is for our algorithm to determine whether or not the region is in fact a stop sign. One of the most important ways to determine this is to decide whether the region in the image is in fact that of a regular octagon. Now that we know where the corners are, we can use them to generate a perspective transform of the stop sign onto the points of a regular octagon. If the resulting transformation fits such that the edge lines are of relatively equal length and the corners have similar angles, it is safe for our algorithm to say that the image likely contains a stop sign. In order to obtain this perspective transform we have the option of using OpenCV’s getPerspectiveTransform function or its findHomography function. The former only allows us to map for of the stop sign’s corner points while with findHomography we can map all 8. In order to create this mapping however, a key problem must be solved—that is, where the corners of our destination image should be.

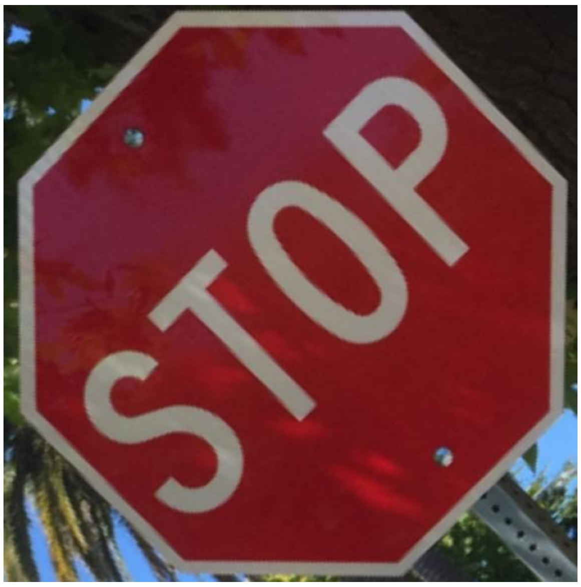
Naturally, the result of the transform if a stop sign is present, should be a regular octagon. This implies sides of all equal length. In order to obtain sides of equal length in a discrete digital image, one must consider carefully how long to make each side. Because some of the sides will have to exist at a slope that is not perfectly horizontal or vertical, it is impossible to make them all the same length without a slight skewing of the interior octagonal angles. In order to minimize this skewing, we want to make sure the diagonal lines of the octagon form as close to an isosceles right triangle as possible with respect to the x and y axes. Of course, achieving an isosceles right triangle is impossible while maintaining integer values for the length of each side. This means that we must carefully choose a Pythagorean triple that is *nearly* isosceles. The triple used in our implementation is 119 : 120 : 169. By making sure this is nearly isosceles, we are ensuring that the diagonal sides of the octagon are as close to having a slope of 1 or -1 as possible. With this triple chosen, the precise corner points for are transformation can be picked naturally. Our implementation also accounts for a 20 pixel buffer from each axis so that the resulting image has space around it. The points chosen in this implementation are as follows:

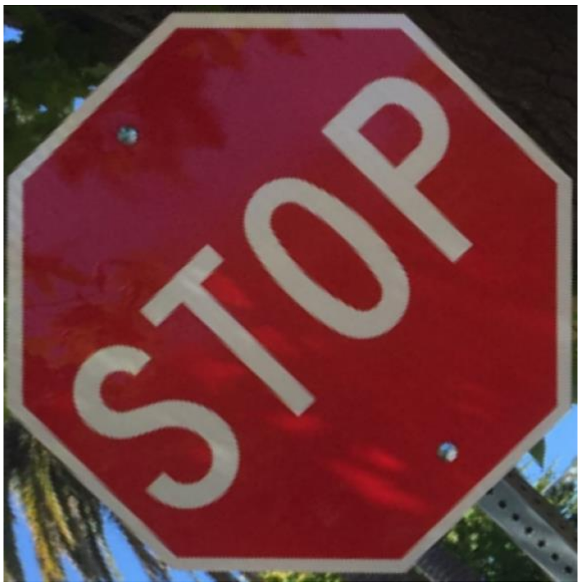


Using these points, it is simple to compute and apply the perspective transform. For precision sake, the implementation for creating a transform using all 8 corner points is shown here. It uses findHomography to compute the transform and then warpPerspective to apply the transform to the source image:

```
void change_perspective8(std::vector<Point> corner_points, const Mat
&source_image, Mat &dst_image);

//In the main -- Change perspective of bgr image using 4 points
r.perspective4_bgr = r.bgr_img.clone();
change_perspective4(corner_points, r.bgr_img, r.perspective4_bgr);

//Change perspective of boundary using 4 points
r.perspective4_boundary = r.thinned_boundary.clone();
change_perspective4(corner_points, r.thinned_boundary, r.perspective4_boundary);
//END MAIN

void change_perspective8(std::vector<Point> corner_points, const Mat
&source_image, Mat &dst_image)
{
  std::vector<Point2f> use_points;
  for (int i = 0; i < 8; i++)
  {
    use_points.push_back(Point2f(corner_points[i]));
  }

  std::vector<Point2f> dst;
  dst.push_back(Point2f(139, 20));
  dst.push_back(Point2f(308, 20));
  dst.push_back(Point2f(427, 140));
  dst.push_back(Point2f(427, 309));
  dst.push_back(Point2f(308, 429));
  dst.push_back(Point2f(139, 429));
  dst.push_back(Point2f(20, 309));
  dst.push_back(Point2f(20, 140));

  Mat s(use_points);
  Mat d(dst);

  Mat transform = findHomography(s, d);
  warpPerspective(source_image, dst_image, transform, Size(447, 449));
}
```

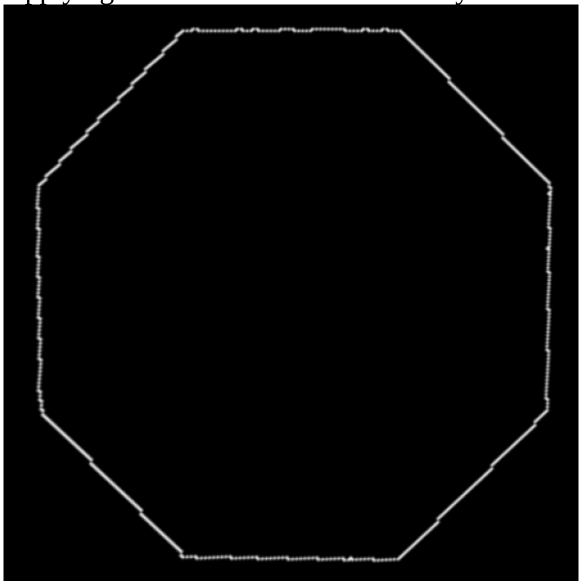
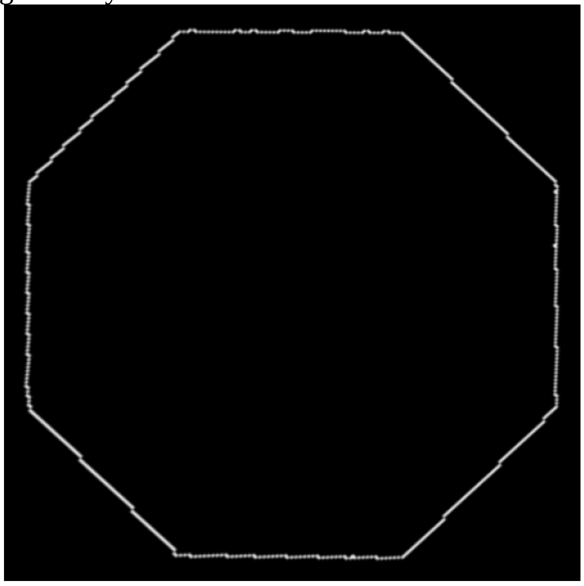
Using this and the very similar method of getting a perspective transform from only 4 points, we have the following results:





Left: Used 4 points and getPerspectiveTransform, Right: Used 8 points and findHomography

Applying the transform to the boundary of the image alone yields:

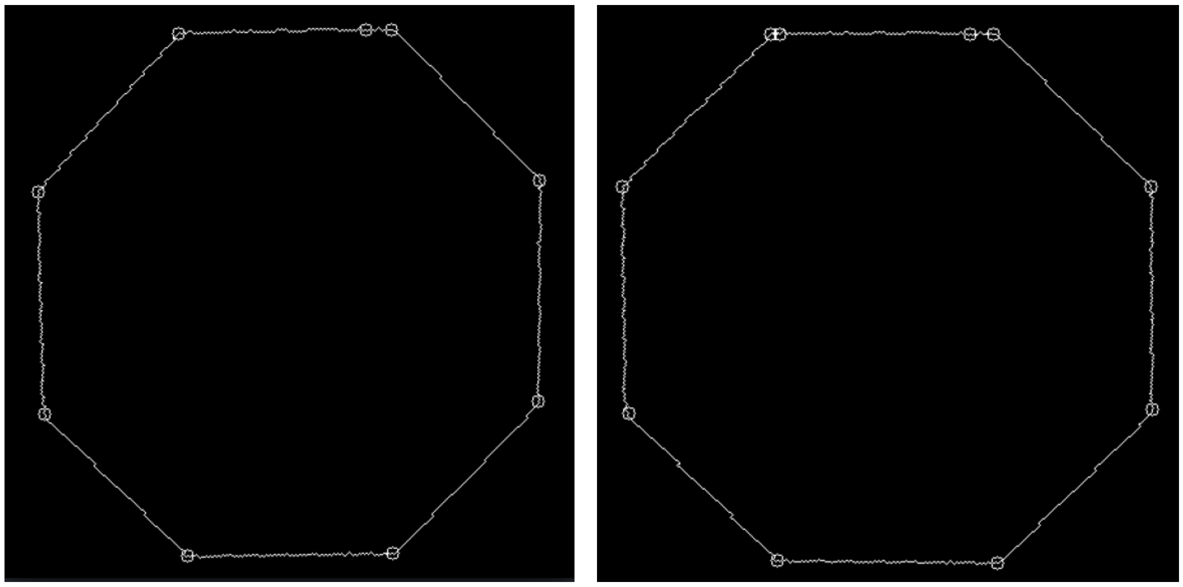
Left: Used 4 points and getPerspectiveTransform, Right: Used 8 points and findHomography

VIII. Operating On Transformed Region

Finally, having the boundary region of the image, we can apply the same previously used techniques to thin the boundary and relocate its apparent corner points. Ultimately this should allow us to cross reference invariants based on the sign's lines and corner points against that of what we would expect to be the case in a normal octagon. An implementation to thin and relocate corner points is shown:

```
void get_perspective_corners(Mat &image);

//IN THE MAIN
get_perspective_corners(r.perspective4_corners);
get_perspective_corners(r.perspective8_corners);
//END MAIN

void get_perspective_corners(Mat &image)
{
  //Make boundary all same value
  for (int j = 0; j < image.cols; j++)
  {
    for (int i = 0; i < image.rows; i++)
    {
      if (image.at<unsigned char>(i,j) != 0)
      {
        image.at<unsigned char>(i,j) = 255;
      }
    }
  }

  //Generate thinned boundary
  thin(image);

  Mat dummy = Mat::zeros(image.rows, image.cols, CV_8U);
  std::vector<Point> bps;

  //Find thinned boundary points
  bool done = false;
  for(int i = 0; i < image.rows; i++) {
    if (done) break;
    for(int j = 0; j < image.cols; j++) {
      if (isEdge(image, i, j))
      {
        drawBoundary(image, dummy, i, j, bps);
        done = true;
        break;
      }
    }
  }

  //Find boundary corners
  std::vector<Point> corner_points;
  get_corners(bps, corner_points, image, 0.1, 7);
}
```

The result of these operations yielded:


Left: Used 4 points and getPerspectiveTransform, Right: Used 8 points and findHomography

Clearly, this produced imperfect results for finding the corner points as the jagged edges caused an extra point or two to be identified. The corners we would expect have been found, but the corner detection algorithm needs to be made more robust so that only the best corners are chosen in order for invariants to be analyzed. This is expected to be first priority in a future extension of this project.

IX. FUTURE EXTENSIONS

This project was clearly successful in exploring a number of computer vision and image processing algorithms to identify features pertinent to detecting a stop sign in an image. However, since this application pertains to something as sensitive as self-driving vehicles, in practice robustness of these algorithms must be key. This algorithm works extremely well on many stop sign images, however, it is not perfect. One of the key extensions must be in optimizing the corner detection algorithm to only find corners we would expect on a stop sign and ignore any that arise from a slightly jagged side. These corners could even be identified using the highly precise line detection algorithms previously discussed and determining where edge lines intersect. This will allow for perspective invariants to finally be analyzed, which the project ultimately sought to do in order to make the final classification of the image being a stop sign.

Furthermore, many other tools can be utilized to further *verify* the presence of a stop sign. A natural one is in character recognition and detecting that the letters S, T, O, and P, do in fact exist in the image. With a few simple additions this could be a very quick module that would further extend the project and increase robustness.

Overall, the project would benefit from an abundance of testing, with an emphasis on robustness so each module explored herein will work the same way on a variety of images (including those that don’t include stop signs at all). With more robust corner detection, and greater perspective invariant analysis, this project could perform at very high accuracy in classifying images on whether or not they do in fact contain a stop sign.

X. WORKS CITED

[1] Rosebrock, Adrian. “4 Point OpenCV getPerspectiveTransform Example”. *PyImageSearch,* PyImageSearch.com.

[2] “OpenCV 2.4.13.7 Documentation”. *OpenCV Dev Team,* docs.OpenCV.org.