Final Summary of Pose Estimation Project

Andrew Ho
Aerospace Robotics and Control Laboratory
California Institute of Technology

July 13, 2018

1 Introduction

The goal of the image processing stage is to extract line segments which correspond to true edges or at least to edges available in the spacecraft model. For our project, we aim to use the following approach to detect line segments within an image, in order to detect larger polygons:

1. Low Pass Filtering
2. Edge Detection
3. Edge Smoothing
4. Straight Edge filtering
5. Polygon Extraction

Low Pass Filtering reduces contrast within image, effectively blurring out less important edges. This allows for higher accuracy when we apply edge detection algorithms.

Edge Detection is the first stage in which preliminary edge detection occurs. The best method would be to use Canny Edge Detection.

Edge Smoothing algorithms such as Contour Approximation are applied, which aim to eliminate insignificant edges produced by the Edge Detection

step and extract vertices of significant edges.

Straight Edge filtering takes the smoothed edges from Edge Smoothing and finds straight edges by splitting up detected edges into separate lines. Overlapping straight edges are filtered out.
Polygon extraction takes the filtered straight edges and builds polygons using conditions of collinearity, parallelism, and adjacency, as discussed in the Simone D?Amico.

2 The issues with Hough Line Transform and other Straight Line Detection Algorithms

Hough Line Transform and other popular straight line detection algorithms is fantastic for images with many straight edges, and for detecting a majority of edges in an image. However, it values guaranteeing straight edges over checking all potential straight edges. As a result, Hough Line Transform is not ideal for detecting objects.

3 Proposed New Line Detection Algorithm

We propose a new algorithm that guarantees all significant straight edges in an image are detected. This algorithm utilizes several libraries from OpenCV, which will be referenced below.

1. We start by loading the source image into a Mat object, which is standard practice with OpenCV.

2. We then apply a Gaussian filter to blur pixels. This first pass removes soft noise and prepares the image for thresholding. Gaussian filter has produced the best results with this algorithm.??

3. Next, apply a thresholding function on blurred image, which increases contrast between pixels, strengthening detected edges. Note that if a gaussian filter had not been applied, noise could have been amplified, which would throw off the Canny Edge Detection in the next step.??

4. Apply Canny Edge detection (taken from OpenCV) on thresholded image to outline object. This retrieves all pixels in an image that lie on an edge.??

5. Apply approximate contour detection (taken from OpenCV) to get edges from canny map. This gives us the actual set of points that are on the Canny Edge detection in a coordinate system. Often times, due to the nature of Bresenham's line algorithm, implemented by approximate contour detection, significant points come in pairs. This works nicely with the next section.??

6. From all contours, filter out edges with less than 20 points. All contours with less than 20 points are 99% likely to be fringe edges that we do not want??

7. From remaining contours, detect all straight edges (straights). The algorithm uses the Pearson Correlation Coefficient to detect the probability a set of points forms a line.

   The process is as follows:

   We first create a new C++ vector called "straight_contours". This holds all of our detected straight contours.

   We then iterate through each contour from the contour map.

   Testing 6 points at a time, we calculate the Pearson Correlation Coefficient (r value) through these 6 points. The higher the r value of a set of points, the more likely they are to be on the same line. If the r value does not pass a certain threshold (we take the threshold to be 0.85), then we remove the first two points and add the next two points from the contour map.

   However, if the r value through these 6 points is higher than the threshold, then we add the next two points from the contour map. If the r value increases, then we know we have detected a straight edge.

   We continue to add two points at a time, in order to take pairs of significant points as described in the previous section. in addition, this speeds up the process of checking every point. As long as the r value continues to increase, we know that the points added are supposed to be on the line.

Once we add points that decrease the r value of the set of points, we check whether we remove both or only one of the points. We then take the first point added and the last point added to mark the straight contour, and add it to straight_contours.

Iterating through all contours in the contour map returns all straight edges in the image.

8. The method outlined in 7) is especially vulnerable to thick edges, since it will detect the same edge twice: once for the outer edge and once for the inner edge. As such, we need to filter out edges that appear to be from the same edge.

   We sort the resulting straight edges by their slope, from negative to positive. If the slopes are similar, then we check the distances from end points. If the distances between end points are small enough, then we know that these lines are overlapping. We then eliminate the shorter edge.

   After filtering, we have only edges that we want.

9. We can now apply conditions outlined by Simone D'Amico

   First, we test for lines that should be collinear. If two lines are meant to be unified as a single line, then we take the two furthest endpoints as the endpoints of the new line.

   After connecting all line segments that should be connected, we need to test to see if any resulting line segments are overlapping. The reason we check twice is because if we were to connect collinear edges before initially checking for overlapping edges, there would be a much higher chance of accidentally connecting lines that were not meant to be added together.

   Next, we check for pairs of parallel lines. If we find significant pairs of parallel lines, then we know we have identified parts of a significant polygon. These pairs are stored for later.

   Finally, we check if any non-collinear line segments should be connected.

   The result of this step gives us all significant connected edges.

10. We can now search within the detected lines whether a polygon exists.

If two pairs of parallel lines were connected then a rectangle has been detected guaranteed.

If two pairs of parallel lines are connected and connected by a line, then draw the other line with the unconnected endpoints of the parallel lines

If no pairs of parallel lines were detected, then if any two lines were connected, draw parallelogram.

If no lines were connected, then no polygons have been detected. This end case is one that should not result.

4 Current issues and Next Steps

Currently, the code is unable to properly implement the Stanford conditions. The program currently outputs significant straight edges found in the original image using the new line detection algorithm. Further steps need to be taken to rectify the incorrect implementation of collinear, parallel, and adjacency detection.

A proposed solution is to implement a directed graph data structure, saving each corner as a node. This would allow us to test whether pairs of parallel lines are in fact connected by a single edge. If not, then we know that an edge that does not currently exist must be drawn.

The complete polygon detection algorithm must be completed. However, after edge cases are dealt with, we then look to implement tracking algorithms. After performing object detection on the first frame, we can then track the detected objects in faster time that relying solely on polygon detection for every frame. However, to continually check that new structures are not detected within each frame, we will continue to apply object detection to every frame.