I. Technique Overview

One powerful technique to analyze photographs in the field of digital photo forensics focuses on shadows and its light source. A point light source will lie on a single line with a point on a shadow and its corresponding point on the object casting that shadow due to the fact that light travels in a straight line. The light source will always lie on a line connecting every point on a shadow with its corresponding point on the object. In any photograph or image, the projection of the lines formed by these points should always intersect that the two-dimensional projection of the light source. Thus, a forgery or and edited image can be determined by the presence of an inconsistent shadow. Such a shadow would be defined to contain a point and its corresponding point on the object casting said shadow that lie on a line, which when projected, does not cross through the light source determined by the projection of the lines formed by points on other shadows that are considered to be legitimate within the image. Note that it is possible to have multiple inconsistent shadows.

II. Program Overview

This previously mentioned characteristic lends itself to the forensic analysis tool developed in this program, which will be referred to as Shad. Shad focuses on computing the vanishing point, which represents the light source within a user provided image, based on the user-selected lines, each of which are specified by two selected points. Given three or more lines, Shad will attempt to determine the vanishing point by applying the least squares method to better fit the user-selected lines in order to achieve a well defined vanishing point that may otherwise not appear with the initial lines due to noise or pixel inaccuracies. The program will optimize a function that minimizes the deviation of all pairs of line intersections from their center of mass, while minimizing the deviation of the user-selected points from their initial values.

III. Code and Solution

The program, which is written as a function, takes a user input image name as its input argument. After displaying the image, the program will prompt the user to indicate the number of lines the user wish to input. The user will then select two points to specify each line of interest. The function ginput() is used to retrieve the coordinates of the points selected. Each point selected is then copied into either an X-coordinate or Y-coordinate matrix. With respect to the Y-coordinates, the value is negated before being copied into the matrix because the Y values become increasingly positive as you go south of the top-left point, which will be treated as the origin (0,0). The negation of the Y-coordinate will treat the entire image as the fourth quadrant of a regular Cartesian coordinate system.

After the coordinates of each line are copied, the slope and intercept are calculated copied to their respective matrices. The slope matrix (M) is an nx2 matrix and the intercept matrix (B) is an nx1 matrix. These two matrices are passed as one matrix through the MATLAB fminsearch() function, which will optimize the objective function opti(). The nested opti() function will split the single matrix back into the two separate slope and intercept matrices. The function then uses these two matrices along with an 2x1 matrix (X) representing the unknown common intersection (x,y) in a linear system that will calculate the intersection using the lines as constraints. The linear system is derived from the following statement: Given n lines, the lines will intersect if for some (x,y), there exists $(m_i)(x)+(b_i)=y$, for all i values from 1 to n. In order to calculate the intersection point, the X vector, the function will solve for X in the following equation: (M)(X)=(B). Once the X vector, representing the intersection point, is computed, the function will calculate the sum of the residual errors (vertical distances from the point to each line). It is this sum that fminsearch() will attempt to minimize in the optimization of the opti() function.

The fminsearch() function will return a new set of slopes and intercepts, which will be then used to plot the "wiggled" lines along with the user-selected lines. Shad will plot the new lines as well as the user-selected lines using the refline() function. The new lines will visibly intersect at a single point and the user can determine whether the newly fit lines are appropriate, given their previously selected lines. This solves the problem of the highly unlikely chance of an intersection when there are three or more lines.

IV. Examples

There are several examples to display the program in action. The three figures displayed below are from Image 1.

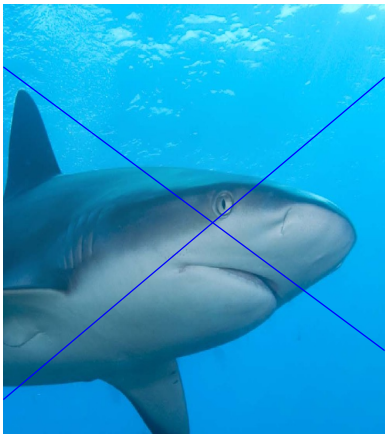

Figure 1. Two lines.

Figure 2. Four lines.

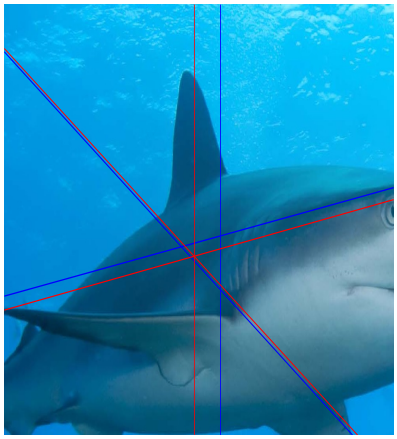

Figure 3. Edge case: vertical line

Figure 1 displays the simple case when the user selects two lines. With two lines, the user can easily see the intersection between the two lines. However, when the user selects three or more lines, it is unlikely that all the lines intersect. Figure 2 shows the result of the optimization of the objective function opti(). The four user lines are colored blue and the new "wiggled" lines are colored red. The blue lines do not intersect at any common point, but the red lines converge upon a single point while maintaining close distance from their reference user lines. Figure 3 displays the edge case involving a vertical line. There are three user-selected lines (in blue) with

one line being a vertical line. The resulting red lines determined after the optimization converge upon a common point while the original user-selected lines do not. The resulting lines, of course, maintain a close distance with their reference lines.

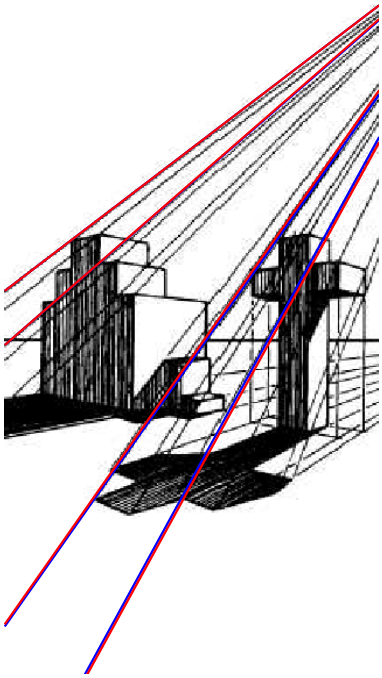

Figure 4. Four user-selected lines

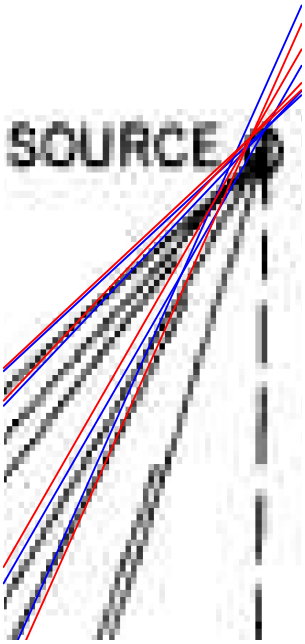

Figure 5. Intersection at light source

Figures 4 and 5 come from Image 2, an image with pre-drawn lines containing a point on the shadow, its corresponding point on the object, and the light source. Both Figures are the result of a same run of the program. The red lines represent the new, “wiggled” lines while the blue lines represent the user-selected lines. While it may be difficult to observe in Figure 4, the two top red lines are actually overlapping the user-selected blue lines because the user-selected lines are so aligned with the pre-drawn lines. The bottom two red lines, however, are at a miniscule distance from the user-selected lines. Figure 5 shows the lines intersecting at the light source that is indicated and pre-drawn in the image. While the blue, user-selected lines do not all intersect at a common point, the red lines do, and do so with very great accuracy to the light source. While miniscule, there is still a difference between the user-selected lines and the

optimized lines, and this can be more easily observed in Figure 5 compared to the previous figure.

V. Edge Cases

There are several edge cases to consider with this program. One such edge case would involve the user selecting the same point in order to specify a line. By choosing the same point twice, the user would have specified a point rather than a line. If such an event occurs, Shad will exit and display the appropriate error message. Another edge case to consider is parallel lines. Multiple parallel lines are allowed, as long as there is another user-selected line that is not parallel in order to create an intersection for the lines. If all the lines are parallel within an image, then there is either no vanishing point or the point light source is infinitely far away. In the case that all of the user-selected lines are parallel, the program will exit and display the relevant error message.

Two important edge cases manage horizontal and vertical lines as inputs. In the case with horizontal lines, the slope is zero. Thus, when optimizing through the least squares method, the distance between the calculated point and the line will be the intercept of the line subtract the y-coordinate of the point of interest. In this case, our least squares method will still function. However, with vertical lines, the slope and intercept are undefined, but the vertical line, X, can be represented in the form $x=x_n$. To include the vertical line in the computation of the common intercept, simply use the form $x=x_n$ in the linear system. Originally, the linear system will form lines by the following equation: $(-m_i)(x)+(1)(y)=(b_i)$. To accommodate the change, set the appropriate $(-m_i)$ value to 1, change the 1 in the i^{th} row, second column of the slope matrix to zero, and set the corresponding value in the intercept matrix to the value of the vertical line (x_n), resulting in $(1)(x)+(0)(y)=x_n$ or $x=x_n$, which is the vertical line. Now the vertical line can be

incorporated into the linear system to calculate the common intercept. However, when calculating the distance between the line and the intersection point, the vertical distance is not used, but rather, the horizontal distance between the lines and the intersection point. Appropriate measures have been taken within the program to determine vertical lines from the user-selected points, change the appropriate values in the slope and intercept matrices, and plot the vertical lines.