Skew Symmetry Detection and Reconstruction

Seminar in Applied Geometry with Alfred Bruckstein
Final report by Eric Yudin

Introduction and Motivation
In this course we have studied symmetry as a property of a set of points on a plane that map only onto each other after going through some isometric (distance-preserving) transformation. The symmetries we have seen include reflective, rotational, translational, and glide-reflective symmetry.

A well-known problem in computer vision and applied geometry is the detection of such symmetries given a set of points. In fact, many algorithms exist that are able to accomplish this task. The subject of the paper Skew-Symmetry Detection Via Invariant Signatures goes one step further: can we detect symmetries in objects that have undergone a non-isometric transformation? The authors present a procedure for doing so, as well as instructions for reconstructing the original shape up to a Euclidean transformation.

The analysis of this paper has culminated in the implementation of an application that demonstrates the results. This report along with the application therefore acts as a summary of the authors’ findings, as well as a description of the development process of the application.

To ease the development process, most of the concepts were first coded in MATLAB. This proved very useful as mathematical errors can be easily debugged when a working prototype is readily available for comparison. The final version was written in the Java programming language, and can be accessed on the internet at the following address: http://cs.technion.ac.il/~eyudin/applet.html.
Background: Non-Isometric Transformations
Throughout the course of the seminar we dealt mainly with isometric transformations. These mappings are of the form
\[
\begin{bmatrix}
    x' \\
    y'
\end{bmatrix} = \begin{bmatrix}
    \cos \theta & -\sin \theta \\
    \sin \theta & \cos \theta
\end{bmatrix} \begin{bmatrix}
    1 & 0 \\
    0 & \pm 1
\end{bmatrix} \begin{bmatrix}
    x \\
    y
\end{bmatrix} + \begin{bmatrix}
    t_x \\
    t_y
\end{bmatrix}
\]
and include rotations (expressed by the leftmost matrix), reflections (middle matrix), translations (rightmost vector), and glide reflections (a combination of reflections and translations). The isometric property implies that the Euclidean distance between any two given points will remain the same after undergoing this transformation. We can see that no distances are modified by looking at the determinant of the above expression: it equals unity (or negative unity). Since the determinant dictates the scaling ordered by the mapping, we see in this case that there is no scaling. Because of this, all of the points maintain their Euclidean distances with respect to each other.

In this case we concern ourselves with transformations that don’t preserve distance. Two examples of these non-isometric transformations include affine and projective.

Affine
Affine transformations are of the form
\[
\begin{bmatrix}
    x' \\
    y'
\end{bmatrix} = A \begin{bmatrix}
    x \\
    y
\end{bmatrix} + \begin{bmatrix}
    t_x \\
    t_y
\end{bmatrix}, \text{ where } A = \begin{bmatrix}
    \cos \theta & -\sin \theta \\
    \sin \theta & \cos \theta
\end{bmatrix} \begin{bmatrix}
    a_x & 0 \\
    0 & a_y
\end{bmatrix} \begin{bmatrix}
    1 & s
\end{bmatrix}.
\]
We can see here that there are six degrees of freedom that affect the mapping: one for rotation (theta), two for scaling (ax, ay), one for skew in the x-direction (s) and two for translation (tx, ty). Note that a parameter for skew in the y-direction is not necessary, as we can always equivalently express a transformation involving y-skew in the above form. Figure 1 shows the effects of an affine transformation.

Figure 1 - Left: Original polygon. Right: After affine transformation with rotation = 15 degrees, scale = (0.5, 0.5) skew = -0.9, translation = (20, 20).
A well-known property of affine transformations is that they preserve parallel lines. More generally, affine transformations preserve ratios of distances along a given line. We can explain this by noticing that the determinant of the above expression is $a_xa_y$; that is, these transformations scale each of $x$ and $y$ by a constant factor. This important result will prove very useful in the skew symmetry detection process.

**Projective**

Projective transformations are more general than affine ones; they follow the form

$$
\begin{bmatrix}
x' \\
y' \\
1
\end{bmatrix} = \frac{1}{1+w_xx+w_yy} A \begin{bmatrix} x \\
y \\
z
\end{bmatrix} \begin{bmatrix} x_x \\
y_y \\
0
\end{bmatrix},
$$

with the same matrix $A$ as in the affine case. Here we have an extra factor known as the *projective divide* that multiplies $A$. The values $w_x$ and $w_y$ are known as tilt parameters, and changing them mathematically changes (or visually, tilts) the plane onto which our expression is projecting the points. Figure 2 demonstrates the effect of tilt visually.

![Figure 2](image)

Note that projective transformations are a superclass of affine transformations: by setting tilt to zero we arrive back at the affine expression. Also, note that lines do not remain proportionally distant to one another, meaning that lines that were once parallel may now intersect. This result gives rise to the phenomenon of ‘vanishing points’ in projective photography and illustration, where all parallel lines (i.e. having the same slope) in true space converge to the same point on the projective plane.
Symmetry and Skew Symmetry Detection

The skew symmetry detection method is based on a previously-developed technique for finding symmetries of non-skewed shapes. The fundamental idea is to assign a characteristic signature to each vertex based on some aspect of the geometry at that vertex. Analyzing a cyclic string of these signatures allows the discovery of symmetries: if the string is cyclically palindromic, there is reflective symmetry. Moreover, if the string has one substring worth of values that repeats itself, there is rotational symmetry of degree equaling the number of repetitions of the substring. For example, in Figure 3(b), for side lengths \(d_n\) and angles \(a_n\), since \(d_1 = d_3 = d_5\), \(a_1 = a_3 = d_5\), and \(d_2 = d_4 = d_6\), then the string of values \(\{d_1a_1d_2\}\) will repeat itself three times in the overall string, indicating the existence of rotational symmetry of degree 3.

![Figure 3 - Examples of assigning geometry-based signature to a vertex. (a) reflective symmetry. (b) rotational symmetry.](image)

Drawing upon this, we can infer that if we can find signatures whose values are invariant to certain transformations, then these palindromic or substring-cyclical patterns existing in a polygon’s signature string indicates that there is a corresponding untransformed version of that polygon bearing reflective or rotational symmetry.

Transformation-invariant signatures

We now discuss what signatures we could use that would be invariant to our transformations of interest.

For the case of affine transformations, we draw on our previous result that affine transformations scale each of \(x\) and \(y\) uniformly. If a square of side 1 is put through an affine transformation with scale factors \(ax\) and \(ay\), then the new area enclosed by the points is now \(axay\). Moreover, any shape will be scaled by this factor. We see from this that the ratio of the areas of any two shapes remains invariant under an affine transformation.

Knowing this, we construct for each vertex a pair of shapes (usually triangles), and take the ratio of their areas as the signature. The paper defines the following metrics:
where $\Delta(a, b, c)$ represents the area of the triangle enclosed by vertices $a$, $b$, and $c$. Note that these signatures are not symmetric in themselves; that is, $\rho^L(Q_{i-1}) \neq \rho^L(Q_{i+1})$. However, $\rho^L(Q_{i-1}) = \rho^R(Q_{i+1})$, so we can use both signatures together to compare vertices equidistant from the vertex of interest ($Q_i$ in the above diagrams). Other variations are possible as well. In developing the application at least one other signature was tested, albeit with mixed results, as will be discussed.

For the projective case, we can use the cross ratio, as defined by

$$CR(A, B, C, D) = \frac{\|A, C\| \cdot \|B, D\|}{\|A, D\| \cdot \|B, C\|}$$

Figure 4 graphically illustrates two sets of points (in blue) with the same cross ratio.

The invariance of the cross ratio to projective transformations is proven elsewhere.

Similarly to the affine case, we construct a way to calculate the signature for each point, using its surrounding geometry. Figure 5 is a diagram from the paper illustrating one way to do so.
Armed with the above invariants, we can proceed with the original method and discover symmetries in our objects. While this is a simple process (and concavity/convexity should not matter), there are a few points to keep in mind. First, if the metric is in any way asymmetric with respect to the vertex of interest, care must be taken to always use points and lines in the correct order as specified by the signature. Second, there are cases where the metric may evaluate to infinity (in fact, this occurred during testing; see next section). Finally, the metrics discussed in the paper all involved absolute values; however, in some cases it may be beneficial to include signed areas. There may be cases where a convexity on one side of a shape corresponds to a concavity on the other side, and the only way to differentiate them is with a signed signature.

Both Figure 1 and Figure 2 demonstrate the application’s detection of affine and projective symmetries respectively. The green lines in the right-hand pane represent the axes of symmetry computed by the algorithm.

Development: Symmetry Detection
The prototype in MATLAB begins with a script that starts with a set of points known to be symmetric, and then transforms them with an affine transformation (projection would come later). All variables are exposed to allow testing of different transformations, and several different shapes were set up as test cases. Both sets of points are then run through a symmetry detection algorithm based on the above theory. Since the signatures derived should be affine-invariant, both sets of points should reveal the same signatures. A similar, separate script was written for projective transformations.

Several functions were instrumental in this process, the most important of which was

\texttt{GetSignaturesAffine()}. This works on a set of points and obtains the affine-invariant signatures at each vertex. Originally the simplest one was used, defined as

\[
\rho^1 = \frac{||\Delta(Q_{i-1}, Q_i, Q_{i+1})||}{||\Delta(Q_{i-2}, Q_i, Q_{i+2})||}
\]
as seen in the paper. This signature is convenient as it is symmetric and easy to program. However, it is not perfect as it is prone to infinities if surrounding vertices are lined up properly. Figure 6 shows a MATLAB plot of a cast where two of the axes are missed. This occurs because the triangle area $\Delta(Q_{i-2}, Q_i, Q_{i+2})$ evaluates to zero at the appropriate vertices (here in the corners), and so the signature evaluates to infinity.

![Figure 6](image_url) Note that the algorithm missed some symmetry axes

In fact the asymmetric metrics $\rho^L(i), \rho^R(i)$ suffer from a similar problem: the lines $\{Q_{i-1}Q_{i+1}\}$ and $\{Q_iQ_{i+2}\}$ become parallel to each other, so their intersection point is at infinity, and the area diverges. Another metric altogether was tested:

$$\gamma^L = \frac{\|\Delta(Q_{i-1}, Q_i, Q_{i+1})\|}{\|\Delta(Q_i, Q_{i+1}, Q_{i+2})\|} \quad \gamma^R = \frac{\|\Delta(Q_{i-1}, Q_i, Q_{i+1})\|}{\|\Delta(Q_i, Q_{i+1}, Q_{i-2})\|}$$

This worked nicely for the cross example in Figure 6. In the end however, the asymmetric $\rho^L(i)$ and $\rho^R(i)$ were used since they work in the majority of cases and for the purposes of demonstration worked well.

For the projective case, function `GetSignaturesProj()` was implemented. This function evaluates the cross ratio for each point via the method illustrated in Figure 5. Finally, after signature generation, `FindCyclicPalindrome()` finds all axes of symmetry. This function iterates through the vertices, testing each for vertex and edge symmetry via the formulas mentioned in the paper:

Vertex symmetric: $\rho^R(i - k) = \rho^L(i + k), \rho^L(i - k) = \rho^R(i + k), \ i = 1 \ldots N$

Edge symmetric: $\rho^R(i - k - 1) = \rho^L(i + k), \rho^L(i - k - 1) = \rho^R(i + k), \ i = 1 \ldots N$

This algorithm is $O(N^2)$, but for the application at hand I was not very concerned with time complexity optimization. The implementation in Java easily runs in real-time.
Reconstruction

After we have determined the axes of symmetry for a skewed shape, the task arises to reproduce its appearance to its pre-transformation configuration. We find that it is possible to do so, albeit only up to a Euclidean-distance-preserving transformation. In fact, the only parameter we learn from knowing the axes of symmetry is the skew parameter for affine transformations, and additionally the x-tilt parameter for projective transformations. Thus for each of the affine and projective cases, there are five and six degrees of freedom respectively, worth of non-skewed shapes that correspond to our input.

The procedure for building one of these non-skewed shapes breaks down into four cases, comprised of the combination of transformation type, and whether the axis is symmetric about a vertex or an edge. In each case, we choose a set of initial conditions under some constraints, and these fully constrain what the rest of the shape should like. Figure 7 shows a skewed polygon and an arbitrarily chosen form for its reconstruction.

![Figure 7 - Left: Skewed polygon with symmetry axes. Right: Reconstructed polygon.](image)

Affine Reconstruction

With six degrees of freedom, affine transformations require three 2D points as initial conditions. In both cases we will place three points, but impose a constraint that will account for the one degree of freedom we lose for symmetrizing around the axis. For the vertex symmetric case, the three points must lie in the form of an isosceles triangle, as any other configuration contains skew. For edge symmetry, the three points line an isosceles trapezoid. That is, the edges between the initial condition points do not have to be the same lengths, but the position of the point on the base must be selected carefully: the values of the signatures constrain its position. Essentially we must choose the point on the base such that the values of the signatures on the rest of the shape will require the other base point to complete an isosceles trapezoid. For our \( \rho^R \) and \( \rho^L \) signatures the distance \( x \) as illustrated in Figure 8 is given by \( x = d (1 - 2\lambda)/2\lambda \), where \( d \) is the length of the roof, and is \( \lambda \) the value of the affine signature at the current vertex.
Projective Reconstruction

For the projective case, eight degrees of freedom imply that four points must be chosen. As noted previously, two degrees of freedom (corresponding to skew and x-tilt) are fixed to ensure reflective symmetry. For the case of edge symmetry, the four points must simply take the form of any isosceles trapezoid. For the case of vertex symmetry, the four points line four sides of a pentagon. The top of the pentagon should form an isosceles triangle. In addition, like the edge-symmetric affine case, we must choose the last (base) point such that the signatures will constrain the vertex on the other side of the base to a mirror-symmetric position.

Development: Reconstruction

To implement reconstruction of affine-transformed shapes, the skewed shape is first brought to a normalized form that imposes the necessary constraints, such as \( \{Q_1, Q_2, Q_3\} = \{(-10,0), (0,10), (10,0)\} \), and then transforms from there with the allowed 5 degrees of freedom. In the case of edge-symmetry, the roof of the isosceles trapezoid is fixed, and then the base points are calculated using the formula above.

To bring the shape to the normal form, each parameter is solved sequentially, and ‘peeled off’ of the transformation stack. That is, knowing that the transformation order is: skew, scale,
rotation and then translation, we can first cancel translation by translating all points to the origin. We can then solve for rotation by taking the cross product of the vector (1,0) with the base of the triangle or trapezoid; the arcsine of the magnitude of the resulting vector then represents the rotation. With this, we can rotate back to the origin and continue onto scale and skew. The process is straightforward and very fast.

Implementation of reconstruction in the projective case begins with configuration of the normalized form of the initial conditions. For edge symmetry, this is the trapezoid found by the points \((-20, 0), (-10, 10), (10, 10), (20, 0)\). For vertex symmetric points, the first three points are fixed at \((-10, 10), (0, 15), (10, 10)\), and the last is chosen on the x-axis. To find this last point, the application scans the x-axis for the point that produces a cross ratio closest to the signature of the symmetry vertex. When the target cross ratio is passed, the search reverses direction and the increment is halved. This process continues until the difference is below a threshold. The algorithm is fast, and with an increment starting at 5.0, converges within 10-15 iterations.

Once the normalized form is produced, the application must transform the vertices of interest on the skewed polygon to conform to these initial conditions. This, however, posed problems. Because of the homogeneous division factor, the above ‘peeling’ process is much more difficult as x- and y-tilt are not easily separable. To get around this, a homography solver was employed, found at. The homography solver accepts two sets of four pairs of points: pre- and post-transformation. These points should allow the solver to output the matrix defining the projective transformation between them. Unfortunately, the results were less than satisfactory. It is possible there is a bug in the software, or even that some underlying assumption is incorrect. It was decided to leave this as a future problem and placed on the list of ‘known issues’ for the program; for now, reconstruction for the vertex-symmetric projective case is not supported.
Application: The Skew Symmetry Detector

The Skew Symmetry Detector is an application written in the Java programming language in the form of an applet. The program consists of three sections: Input, Skew Detection, and Reconstruction.

Column one: Input

In the leftmost column, the user defines the points to send to the skew symmetry detection algorithm. Because it is very difficult to draw anything but trivial shapes in skewed form that would have symmetries, xSkews asks the user to input points in the untransformed domain, and then transforms them with user-specified parameters. Points may be input manually by specifying x- and y-coordinates, by clicking on the plot, or, by selecting a preset configuration from the drop-down menu and clicking ‘Preset.’

Extra features of the window include allowing dragging of points, and also clicking an existing point to select it, which will highlight its coordinates in the list box. Since order of point creation is significant, the user can select a point (either in the plot, or in the list box) to ensure that the next point will be added immediately after it.

The warp parameters include all eight degrees of freedom described thus far. Every time one of these values is changed, the entire application updates and re-evaluates in real time. In this way, the application becomes a very instructive and enjoyable interactive experience.

Column two: Skew detection

The middle column represents the skew detection algorithm. Every time a change is detected on the input, the plot view updates by drawing the skewed shape, and any axes of symmetry it might have. In addition, the status message indicates how many axes of symmetry were found, and whether they are affine or projective. The results of the symmetry detector may also be overlayed for comparison on the input plot view by clicking the ‘Overlay’ checkbox in the first column.

Column Three: Reconstruction

The final column demonstrates the reconstruction process. Here, the asymmetric input shape is transformed into a shape without skew, conforming to the initial conditions. The initial conditions are based around the origin, and the user can transform them with parameters manipulating the allowed degrees of freedom that will preserve symmetry (i.e., all but skew and x-tilt). A drop-down menu has also been provided to allow the user to select the symmetry axis around which to reconstruct. Each end of each axis is presented as a separate option. Once selected, the application automatically detects how many initial condition points it needs based on the type of transformation detected, and sets up their configuration based on whether it should be vertex- or edge-symmetric.
Possible Improvements
The following is a list of suggestions that would improve the interactivity as well as the instructiveness of the application.

1. If a polygon is found to have more than one axis of symmetry, we can use these to reduce degrees of freedom and come closer to finding the original shape. For example, preset Shape 04 has horizontal, vertical and diagonal axes of symmetry. Reconstructing them individually only constrains their own direction and the perpendicular direction, leaving the final reconstruction still potentially skewed from the original shape in some other axis. We can use the other discovered axes of symmetry to more fully recover the original shape up to a Euclidean transformation (minus scale).

2. Allow the user to try different signatures

3. Allow the user to modify the error tolerance, which determines at which degree of accuracy two signatures are equal. A higher error threshold finds more symmetry axes, but introduces the wobble seen during reconstruction.

4. Show/hide axes of symmetry from input panel.

5. Find axes of rotational symmetry. This had been prototyped in MATLAB but never made its way into the final application.

6. Even if a symmetry axis is not found, we can use the invariant signature technique to discover a closest match for a symmetric version of the shape. For a given set of points and an assumed axis of symmetry, signatures to one side of the axis dictate what signatures on the other side of the axis should be. The sum of the square of these deviations from ideally symmetric gives us an error metric that can be minimized to suggest the best candidates for axes of symmetry. Once candidates are confirmed, we simply copy the signatures from the ‘ideal’ side of the axis to the other side, and reconstruct. More than one ‘potential’ axis of symmetry (i.e. with a low value for the symmetry metric), preferably not perpendicular to each other, can add confidence to each other’s claim as an axis of symmetry. This notion can be useful in discovering symmetries despite corruption of a polygon.

7. Allow the user to save and load polygons.

Known Issues
While the application successfully demonstrates several cases of the skew detection and reconstruction process, it is not without issues. The following is a list of deviations from the ideal incarnation of this application.

1. Currently, reconstruction of projective skew symmetries is somewhat error prone. As the distortion values are changed, some wobble in the reconstruction is visible. This is probably due to numerical errors.

2. Reconstruction for vertex symmetric symmetries after projection transformations is not currently supported.
3. Shape 03 with only x- or y-tilt escapes projective symmetry detection. Adding tilt on the other axis or varying any other parameter away from identity repairs this.

4. Projective reconstruction should not include y-tilt explicitly, as tilt is applied after rotation, and can therefore make the reconstruction asymmetric. The proper parameter should be a ratio that indicates the proportionality between the two trapezoid bases in the case of edge symmetry, and for the case of vertex symmetry, between the height of the base trapezoid and the height of the isosceles triangle sitting atop. This will ensure that the reconstructed polygon remains symmetric while still allowing a form of y-tilt into the transformation.

References:

Anis Zaman, Keith O’Hara, Homography.java, code link.