**Squiggle - A Glyph Recognizer for Gesture Input**

Jeremy Lee

The Unorthodox Engineers
jeremy@unorthodox.com.au

**Abstract.** Squiggle is a template-based glyph recognizer in the lineage of “$1 Recognizer”[1] and “Protractor”[2]. It seeks a good fit linear affine mapping between the input and template glyphs which are represented as a list of milestone points along the glyph path. The algorithm can recognize input glyphs invariant of rotation, scaling, skew, and reflection symmetries. In practice the algorithm is fast and robust enough to recognize user-generated glyphs as they are being drawn in real time, and to project 'shadows' of the matching templates as feedback.

**1 Introduction**

Glyphs are the individual marks made during writing, from when the pen touches down to when it rises again. They can be as simple as the single dot above an “i”, or as complex as an entire signature. Glyphs are the natural basic units involved in gestural computer input systems such as digitizer tablets, touch screens, and mouse input. The challenge for a glyph recognizer is to take the resulting digital path and guess what it means.

This is a deep problem: not only do we need to search the computational space using an appropriate numeric representation of the glyph, but our result has to match up with what a human being expects.

I present a novel algorithm called Squiggle which accepts an input glyph and compares it against a library of template glyphs. Glyphs are compared through an affine map that projects the templates onto the input using simple linear matrix operations which can be performed efficiently on current hardware. The overlaid glyphs are geometrically compared, and the best match returned. The algorithm cleanly distinguishes between 0-D taps, 1-D strokes and full 2-D glyphs. Some orientations and reflections can optionally be excluded per-glyph.

The raw recognition rates of a template-based system are surprisingly irrelevant when its full capabilities are revealed to the user; to see and reprogram the templates. Success rates exceed 95% on published data, but in practise can asymptote towards 100% as the user adjusts templates that are consistently mis-recognized. Because the templates are projected onto the input glyph there is a natural ability to display the process to the user in their original geometry, which helps them learn the characteristics of the recognizer.

![Fig. 1. The 33 templates of the Squiggle recognizer prototype, including glyphs that cause other template recognizers trouble such as the center-start circle and infinity symbols, and the single/double/triple-stroke lines.](image1)

![Fig. 2. The prototype real-time glyph recognition software draws ‘shadows’ of the best template alignment matches underneath the input pen path. Selective symmetry detection at early stages of the algorithm prevent ‘S’ and ‘Z/N’ domain overlap, but allows ‘P’ and ‘b’ to be recognized as one glyph despite rotation and reflection.](image2)
Out of the infinity of possible affine maps connecting the two spaces, we can use internal numeric features of the glyph geometry to quickly pare that down to just a handful of candidates. We depend on a couple of key insights:

- Affine maps between two co-ordinate systems can be defined by drawing a triangle (with labelled points) in each co-ordinate system, and then asserting they are the ‘same triangle’.
- The glyph path provides a series of points between which triangles can be constructed. No other points really matter, not even the origin.
- We are not interested in comparing arbitrary triangles between glyphs, only the same logical triangle. If we choose the first, eighth, and fourteenth points as indexes, that triangle exists in all glyphs with enough points.
- If two glyphs are very similar, then choosing any logical triangle for the affine map will result in a good match.
- The best triangles for the candidate affine map are the biggest, because vertex jitter will distort them the least. Conversely the smallest triangles are the worst, because they are often degenerate.

The combination of these insights is that we first analyze the input glyph to find its biggest internal triangles, then we pick out the matching logical triangles from the template glyphs for the other side of the affine map. These triangle pairs represent our best chance at a good alignment. If the glyphs are similar, this will project one straight onto the other, resulting in good scores. If they are not, this arbitrary choice will create a projection mess, and bad scores.

**Fig.3.** The path is reduced to 16 milestone points, (equidistant along the original path) and it’s largest internal triangles are found.

**Fig.4.** Each template is projected onto the input glyph via affine transforms defined by the triangles. Some otherwise good matches (like ‘Z’) are excluded because they disallow reflections. The line glyphs are not even tested, because the input glyph is not degenerate.

We essentially assume that the glyphs are identical and can be mapped directly onto each other through corresponding triangles. We perform the operation, and then see how badly it went. We don’t attempt to correct bad assumptions, merely rank them.

All the templates are therefore judged against the same assumption. Although it’s possible for the wrong template glyph to win due to a poor choice of projection, it is surprisingly unlikely. (Just getting the 16 points in the right half of a box by accident is a-priori $1 : 2^{16}$, as a drastic simplification)

If the glyphs are very similar, choosing a different triangle to perform the mapping will have very little effect: the affine map will be more or less the same. So if we have a good fit there is little point searching the triangles for a better one. If we have a bad fit, there is likewise little point broadening our criteria for one side of the map. Why bother? We’re just going to blur the edges of the recognition domains. If finding a good fit is hard, it’s obviously not the correct glyph.

The affine map intrinsically accounts for rotation, scaling, translation, skew, and reflection symmetries. (Although some symmetries such as rotation and reflection may be excluded by individual glyphs) It directly
projects templates into ‘screen’ co-ordinates, not an intermediate ‘normalized’ co-ordinate system. This is important not just because the choice of co-ordinate system drastically affects the geometric difference metric; Points that are close together in one system may be stretched further apart by the affine map. Therefore it also makes sense to perform this comparison in the co-ordinate system that is ‘closest’ to the user, to match their conception of distance and geometry.

2 Algorithm Details

The input glyph and template glyph paths are reduced to a fixed number of milestone points along each path, typically 16, using the methods described in the previous papers. The two lists of points will be named $g$ for the input glyph and $h$ for the template glyph. Individual points from the path are nominated with a subscript index.

$$g = [g_0 \ldots g_{n-1}] \quad g_i \in \mathbb{R}^2$$

$$h = [h_0 \ldots h_{n-1}] \quad h_i \in \mathbb{R}^2$$

$$n \approx 16$$

Each of these sets of points is currently in its own co-ordinate system. Our task is to create an affine map $\hat{A}$ which connects three points (equivalent to an origin point and two axis vectors, or the points of a triangle) across both co-ordinate systems.

It helps to know the total length $\lambda$ of the glyph path. If a direct record of the original path length is not available, you can just sum the segment lengths. This calculation is a lower bound, because these downsampled points are the milestones along a more detailed path that had curvature. Fortunately this number isn’t critical, as long as it’s a good approximation.

$$\lambda(p) \geq \sum_{i=0}^{n-2} |p_{i+1} - p_i|$$

Next we construct our three-dimensional array of all possible triangles $[a,b,c]$ (indexed by the choice of corner points) and fill each entry with a small 2x2 matrix made from the column vectors $(p_b - p_a)$ and $(p_c - p_a)$. We are interested only in the combinations (not permutations) of triangles made from the path points, so our 0-based indices are constrained to increase in order.

$$M(p)_{abc} = [p_b - p_a, p_c - p_a] \quad 0 \leq a < b < c < n \in \mathbb{Z}$$

We take the determinant of each 2x2 matrix, giving us twice the area of the triangle it represents. This is divided by the square of half the total glyph path length (the largest triangle that can be formed by a path is a 90-degree “L” shape with sides of half the total length) to give ‘normalized’ determinant entries in a three dimensional matrix of scalars:

$$D(p)_{abc} = \frac{\text{det}(M(p)_{abc})}{\frac{1}{2} \lambda(p))^2}$$

This particular equation is so important it’s worthwhile expanding it again and showing the full matrix form:

$$D(p)_{abc} = \frac{4}{\lambda(p)^2} \begin{vmatrix} p_b.x - p_a.x & p_c.x - p_a.x \\ p_b.y - p_a.y & p_c.y - p_a.y \end{vmatrix}$$

Each triangle has therefore been given a score which expresses how large it is, relative to the scale of the entire glyph. ‘Small’ and ‘large’ triangles now have consistent absolute values, in the range $0...1$. More usefully, we can choose a constant area $\epsilon$ below which triangles are considered ‘too degenerate’ to be used as half of the affine map. This choice is not particularly critical, and experimentally was set to about $10^{-8}$. (Less than the size of a single pixel, if the glyph occupied the entire screen)

For the algorithm to proceed, we need to fully construct the normalized determinant matrix $G_{abc}$ from the input glyph points $g_i$. If generated from 16 input points, this 3-D matrix will contain 560 entries.

$$G_{abc} = D(g)_{abc}$$
The matrix $H_{abc}$ is similarly constructed from the template glyph points $h_i$ and can either be stored with the template, or (since we only need the occasional sparse entry) we could just store the total path length and compute them on-the-fly.

$$H_{abc} = D(h)_{abc}$$  \hfill (6)

It’s very useful to remember the maximum and minimum entry values of the input glyph matrix $G_{abc}$, as one of them corresponds to the top entry: the largest triangle. (Remember that negative values indicate mirror image triangles, so we need to pick the maximum absolute value in the matrix, not just the most positive.)

$$\top(G) = \text{MAX}_{abc}(\text{abs}(G))$$  \hfill (7)

$$[a, b, c] = \text{triangle}(\top(G))$$  \hfill (8)

The above is a terrible abuse of notation, since of course the top entry and the triangle it corresponds to are found together during one iteration over the matrix, possibly during construction. Also, one should not believe a function that tries to turn a scalar value back into an index triplet.

We use this ‘biggest triangle’ result to make a critical decision: are we dealing with a line glyph, in which the entire glyph has essentially been drawn along a single line. (with allowances for wandering pen) Common examples are single *strokes* or *flicks* in various directions, or lines that double back. Any pointer input system needs to gracefully distinguish these 1-dimensional glyphs, just as they distinguish *taps* or *clicks* as a separate class of 0-dimensional glyph.

Line glyphs are problematic because almost any affine map built from one of its triangles will collapse the other glyph down to single line as well, resulting in false positive matches against glyph templates that were never designed to be used that way. Flatten a “U” enough, (or almost any character) and it looks just like a dash. The opposite problem happens when line glyphs are provided as templates: they can get artificially over-stretched so that a tiny kink in the middle might be exaggerated into the letter “V”.

Note that there is no problem when comparing two glyphs which are both line glyphs. In that case, degenerately collapsing one line onto another line is exactly what we expect, and the algorithm works correctly. It’s only when you mix the classes that grief ensues.

Fortunately line glyphs are detected easily now we know the largest triangle that can be made from its constituent points. If it has very little area, then the glyph points are mostly in a single line. This is identical to the degeneracy test but with a much larger $\epsilon$ (about 0.002…0.004, compared with $10^{-8}$) which was settled on through experimentation and testing to gauge what humans consider ‘straight-ish’.

Importantly, once the algorithm detects that it is in line glyph mode it stops caring about keeping degenerate triangles out of the affine map. We now expect that both glyphs will contain mostly low-area degenerate triangles. This is the only difference between the modes; degenerate triangles on the template side of the map were a criteria for immediate rejection when matching against 2-D glyphs.

Note that only by treating the 1-D glyph as a 2-D glyph *first* (trying to compute triangle areas) can we tell that it’s not. This is in contrast with 0-D taps and clicks which are trivially self-evident.

In order to give us more than one chance to properly match up the glyph geometries, we select the biggest $m$ triangles from the matrix for testing. Various $m$ have been experimentally tried, and values from 8-16 seem to be perfectly adequate. The algorithm continues to function with only a partial loss of discrimination even if you reduce this to 2 or even 1! (Which scarcely seems possible, but if the glyphs are very similar then any triangle will do)

Computationally, constructing the triangle determinant matrix takes about 1,680 MACs (Multiply-Accumulate operations) to turn the 16 point path into a normalized determinant matrix. Each geometric projection takes about 64 MACs, and the geometric distance test another 32 MACs, so testing with the ten biggest triangles ($m = 10$) over a library of 32 template glyphs would consume 30,720 MACs. This is trivial on modern processors.

This neglects the *sorting time* needed to find the biggest $m$ triangles in the input glyph matrix. Beware using a naive sorting algorithm (like shell sort) at this step, which could easily consume $O(n^2)$ CAS-equivalent operations (313,600 for our 560 entry matrix) taking *emphthen* times longer than everything else combined, just to return the top few triangles.

Since we don’t need the result set in order, we can do a *pivot select* where we pick a value, count how many matrix entries it would include and then adjust the pivot value until it selects just the $m$ entries we’re
after, which we collect on a final sweep. It takes about \( O(\log n) = 9 \) passes on average for our matrix, with 560 Compares per pass, so about the same number of read/compares as the merge sort: 5,020 on average) but no memory writes until the final \( m \) are collected to the result list. Also each step compares the constant pivot value against each entry, rather than comparing two entries, so the read bandwidth is halved.

The pivot test selection can fail (or infinite loop) if there isn’t a valid pivot value between the \( m \)th and \((m + 1)\)th ranked entries, such as when they’re tied. An epsilon value stops the recursion when the range becomes too narrow.

Now that we have our set of candidate triangles on the input glyph (selected because they give us the greatest chance of matching against any template) we go through our library of template glyphs and perform the affine mapping of the template into the coordinate system of the input glyph so that the corresponding triangle of the template is forced to fit exactly over the input triangle. This is the point at which we can do selective symmetry and orientation detection, to exclude undesirable matches. If the determinants of the input and template triangles are opposite in sign, then we are about to test glyphs invariant of mirror symmetry.

Time to abuse some more tensor notation: \( \hat{p}_{abc} \) means the affine transformation matrix generated from the triangle with its points at indices \([a, b, c]\) on the path \( p \). Each transform is a 3x2 matrix made from three column vectors built almost directly from the points of the triangle.

\[
\hat{p}_{abc} = \begin{bmatrix}
 p_b - p_a & p_c - p_a & p_a
\end{bmatrix}
\]

\( 0 \leq a < b < c < n \in \mathbb{Z} \) (9)

Or if you prefer to see it expressed as the full matrix:

\[
\hat{p}_{abc} = \begin{bmatrix}
 p_b.x - p_a.x & p_c.x - p_a.x & p_a.x \\
 p_b.y - p_a.y & p_c.y - p_a.y & p_a.y \\
 0 & 0 & 1
\end{bmatrix}
\] (10)

2-dimensional points to be run through the transform are put into a 3-vector row with 1 in the last entry and then multiplied by the transformation matrix to produce a column vector \([x', y']\). This is also equivalent to a standard vector / matrix multiplication if we pretend the 3x2 matrix is actually 3x3, with \([0,0,1]\) in the bottom row, and we assume every vector has ‘1’ in its third element.

The slightly older way of defining an affine transform was to use a 2x2 matrix plus a vector addition so that only 2-vectors had to be involved. I use the 3x2 construction because it is the standard form used by modern computer graphics software and hardware.

We will continue to use lower case letters with a single index to denote an individual point \( p_i \) that exists in a path. We should also be distinguishing row and column vectors using something like bra-ket notation:

\[
(p_i.x, p_i.y, 1) \times \hat{t}_{abc} = (q_i.x, q_i.y)
\] (11)

Or we can just abuse the notation a little more for convenience and clarity and say, to indicate the transformation of a point:

\[
p_i \times \hat{t}_{abc} = q_i
\] (12)

Since we generally transform entire paths at once, we could also say:

\[
p \times \hat{t}_{abc} = q
\] (13)

Each transform has an inverse:

\[
q \times \hat{t}_{abc}^{-1} = p
\] (14)

In this case we’re not dealing with a complete 3x3 matrix, so the inverse is quite a bit easier to construct using Laplace’s formula:

\[
\begin{bmatrix}
 a & c & e \\
 b & d & f \\
 0 & 0 & 1
\end{bmatrix}^{-1} = \begin{bmatrix}
 d & -b & a \\
 -e & ad - cb & cf - de \\
 -f & af - bc & ad - cb \\
 0 & 0 & 1
\end{bmatrix}
\] (15)

Now that we can turn triangles in each coordinate system into affine transforms, we can finally declare the affine map \( \hat{A} \) for our choice of triangles by linking the two affine transformations together.

\[
\hat{A} : \hat{r}_{abc} \rightarrow \hat{g}_{abc} \quad g_i \in \mathbb{R}_g^2 \quad h_i \in \mathbb{R}_h^2
\] (16)
We use this to project all the points $h_i$ into a new set of points $r_i$ in the co-ordinate system of $g$. These are the points we are going to ‘recognize’ against the input glyph. Numerically we do this by multiplying $h_i$ by the matrix inverse of $h_{abc}$ to ‘leave’ that co-ordinate system, and then multiplying by $g_{abc}$ to ‘enter’ the new co-ordinate system to create a new set of points $r_i$.

$$h_i \times h_{abc}^{-1} \times g_{abc} = r_i \quad h_i \in \mathbb{R}_h^2 \quad r_i \in \mathbb{R}_g^2$$

(17)

If you are careful, you can pre-multiply the two matrices together and thereby perform the affine mapping in a single computational step.

$$\hat{t}_{abc} = h_{abc}^{-1} \times g_{abc}$$

(18)

$$h_i \times \hat{t}_{abc} = r_i$$

(19)

We can drop the indexes and apply the transform to entire paths, in this case to turn the template into a recognizer path to be compared against the input glyph:

$$h \times \hat{t}_{abc} = r$$

(20)

The geometric distance calculation we perform between the input and recognizer glyphs is a sum of the errors between the paired-off points.

$$\text{metric}(g, r) = \sum_{i=0}^{n-1} (g_i.x - r_i.x)^2 + (g_i.y - r_i.y)^2$$

(21)

Fig. 5. Input glyphs, their largest triangles, and the projections of template glyphs onto them. Some individual templates (in grey) specifically disallow reflection symmetries, detected by triangle pairs which have determinants of opposite sign.
Note that this is not quite the same as 2-D Euclidean distance, as we have omitted taking the square root. We are free to meddle with the metric, because there is no purely mathematical ‘best’ function: we are trying to choose a behaviour that lines up with human expectations of path closeness, and this is more subtle than pure geometry.

3 Orientation Detection

In order to make some meaningful comparisons with the $1 recognizer and Protractor algorithms, an ‘orientation detector’ was added with the intention of limiting glyphs matches to within 45° of their templates.

Because of the affine mapping, Squiggle does not have a simple conception of ‘glyph angle’, and there are no special orientation vectors that would remain undistorted for 2-D glyphs. Line glyphs do have a natural axis, but such glyphs are not handled by the other two algorithms.

To address this, a triangle similarity test was invented which sums the cosine angles between the corresponding triangle edges of $g_{abc}$ and $h_{abc}$.

$$\text{similarity}_{abc} = \cos(\theta_{ab}) + \cos(\theta_{bc}) + \cos(\theta_{ca})$$

where

$$\cos(\theta_{mn}) = \frac{(g_m - g_n) \cdot (h_m - h_n)}{\|g_m - g_n\| \|h_m - h_n\|}$$

This score ranges from $-3...3$. A right isocoles triangle (with equal sides surrounding the right angle) rotated by 45° produces a similarity score of 2.14. This rapidly drops to 0 at 90°, and then goes negative. This score is invariant to scaling, but is affected by skew.

4 Experimental comparisons with $1 and Protractor

Wobbrock, Wilson and Li’s excellent paper “Gestures without Libraries, Toolkits or Training: A $1 Recognizer for User Interface Prototypes” provided much of the inspiration and framework for the development of Squiggle. Their published set of templates and input gesture data was used for comparative testing.

This data set comprises of 5280 input gestures representing 16 glyphs collected from 11 users at 3 different drawing speeds. The method of collection is described in detail in their paper, but can be summed up as: Eleven different subjects were presented with an image of the template and asked to redraw the glyph on a pen-based computer. Each user drew the glyphs ten times at three different speeds; fast, medium, and slow. Some of the gestures are quite ‘messy’ and would be difficult even for a human being to correctly classify.

Since the original template data was not available, template glyphs were extracted from the code for their example web application. This created a small problem as one glyph (the question mark) had been replaced for the on-line version by a new ‘zig-zag’ template. Therefore, both the zig-zag template and question mark gestures were omitted from the testing, giving 4950 input gestures for 15 template glyphs.

This data set is in many ways a worst case scenario for Squiggle in such a comparison. The low number of templates means Squiggle’s high input glyph analysis costs are not amortized over a large set. The template set is quite ‘square’, and does not contain any line glyphs which the other two algorithms are unable to handle. All glyphs begin and end far from the center, avoiding the singularity that would confuse their ‘indicative angle’ calculation but to which Squiggle is immune. No use is made of mirror symmetries, and some glyphs are...
essentially affine transforms of others. Despite these disadvantages Squiggle performs admirably, producing an accuracy rate within 0.5% of the $1 Recognizer on this data.

The three recognizers were compared against the input data, and also against each other’s conclusions. All three algorithms were run within the same execution environment from the same in-memory data set. Run times include only the core recognition processing time, and omit template setup and other start-up costs. Times were measured over five trials, and the three most consistent times were averaged.

The recognition accuracy rates, runtimes, and correlation between the outputs of these three algorithms are summarized in the tables below. The most important result is the near-parity in accuracy rates between $1 and Squiggle.

| Algorithm     | Accuracy | Runtime |
|---------------|----------|---------|
| $1 Recognizer | 95.56%   | 2.038s  |
| Squiggle      | 95.09%   | 3.292s  |
| Protractor    | 92.87%   | 0.254s  |

Table 2. Correlation of recognized glyphs between pairs of algorithms.

| Algorithm Pair                     | Correlation |
|------------------------------------|-------------|
| Protractor vs. $1 Recognizer       | 96.28%      |
| Squiggle vs. $1 Recognizer         | 94.85%      |
| Squiggle vs. Protractor            | 92.77%      |

It is interesting to break down the recognition match counts for these two algorithms, to see where the mistakes are being made. The following table summarizes how each algorithm has classified the input gestures. There is a visible difference in the pattern of errors.

| Input Gesture | Squiggle | $1 Recognizer |
|---------------|----------|---------------|
| arrow         | 285      | 44            |
| caret         | -330     | -             |
| check         | -304     | 1 -           |
| circle        | 305      | 4 -           |
| delete mark   | -328     | -             |
| left curly    | -299     | 31            |
| left bracket  | -2       | 2 328         |
| pigtail       | 1 8      | -321          |
| rectangle     | 28       | 295           |
| right curly   | -        | -274 56      |
| right bracket | -        | 330           |
| star          | 1        | -329          |
| triangle      | -4       | 325           |
| v             | -5       | -325          |
| x             | -1       | 329           |

Squiggle consistently mis-recognizes arrows as carets, (but oddly never carets as arrows) checks as ‘v’s, and rectangles as circles, whereas $1 does not. This is explained by the affine mapping that Squiggle applies. These glyphs in particular can be transformed almost exactly onto each other through an appropriate affine map with significant skew, which is the symmetry that $1 is not invariant to.

The other interesting result is that Squiggle consistently mis-recognizes rectangles as circles but much less often circles as rectangles, perhaps indicating that subjects are ‘rounding their rectangles’ more than vice-versa. (A brief examination of the data confirmed that users are indeed often drawing rectangles more like squares.)

Both algorithms have difficulty distinguishing the square and curly brackets as one might expect, but Squiggle performs this relatively hard task much better than $1. In fact anywhere that a triangular gesture is compared against a square template, Squiggle almost never makes a mistake; in contrast to $1 often mistaking carets for right brackets, and triangles for rectangles.
5 Code

The following is not pseudocode, but the working JavaScript/ECMAScript used to implement the prototype system and perform tests. It is optimized for readability, and the semantics of the language are well defined. This approach should avoid transcription errors and be useful to implementers. The entire algorithm and all supporting functions is included, providing a complete example that hopefully avoids any ambiguity.

```javascript
// test a glyph against templates using alignment triangles
function match_glyph(g, align) {
    var best_glyph, best_index, best_score;
    // are the glyphs the same line type?
    if(g.nm.ln==h.nm.ln) {
        // go through the alignment triangles
        for(var j in align) {
            var tri = align[j];
            if(!g.nm.line) {
                // if the fraction is essentially zero...
                var npi = (p.count-1)*i/(count-1);
                var npi_int = Math.floor(npi);
                var npi_frac = npi - npi_int;
                // get the normalized determinants of the triangles
                var nd1 = ntm.entry(g.nm,tri[0],tri[1],tri[2]);
                var nd2 = ntm.entry(h.nm,tri[0],tri[1],tri[2]);
                // is either essentially zero?
                if( (Math.abs(nd1)<0.00000001) ||
                    (Math.abs(nd2)<0.00000001) ) test = false;
            } else {
                // interpolate between path points
                var p1 = path[npi_int]; var p2 = path[npi_int+1];
                var p = ip; // the next point is outside the distance. interpolate
                var inter = (dist - d) / (nd - d); // 0..1
                p[0] = p[0]*(1-inter) + np[0]*inter;
                p[1] = p[1]*(1-inter) + np[1]*inter;
            }
            // optional orientation test
            test = test && tri_similarity(g.path,h.path,tri) > 2.12;
        }
        // break path into small regular segments to remove pen jitter
        function path_regularize(path, dist) {
            var count = path.length;
            if(count<=2) return path;
            // result points array
            var r = [];
            var error = 0; var dist2 = dist*dist;
            // add the start point
            var p = path[0].slice(0,2); r.push(p);
            var d = 0; var i = 0;
            while(i<count) {
                var mp = path[i]; // next point along path
                var dp = [ np[0]-p[0], np[1]-p[1] ]; // delta vector
                var d2 = dp[0]*dp[0] + dp[1]*dp[1]; // distance squared
                // how far off? (matters for the last one).
                var error2 = d2-dist2;
                if(error2>0) {
                    if(mp[0]==p[0] && mp[1]==p[1]) return;
                    // if the triangle is essentially zero...
                    if(nm.entry(mp,-1)==0) test = false;
                } else {
                    // the point is within the required distance.
                    r.push(path[count-1].slice(0,2)); // add the final point
                    r.push({
                        points: r,
                        error: dist - Math.sqrt(error2+dist2)
                    });
                }
            }
            return r;
        }
        // triangle orientation similarity between two glyphs
        function tri_similarity(g,h,tri) {
            // get the edge vectors for both paths
            var ge = tri_edges(g.path,tri);
            var he = tri_edges(h.path,tri);
            // sum the cosine similarity
            var s = 0; for(var i=0; i<count; ++i) s+=cos_angle(ge[i],he[i]);
            return s;
        }
        // extract the edge vectors for a path function
        function tri_edges(path, tri) {
            return 
        }
    }
}
```
As well as the obvious code optimizations that would improve speed, several complications can help the algorithm in practice. When recognizing glyphs as they are being input in real-time, we can exploit work done previously:

6 Further Improvements
Append to the regularized path as new input is received, rather than rebuilding it from scratch. Just remember that the regularized path segments are exactly the same distance apart, which means they are not pixel-aligned.

If an established path is only extended by a few pixels, it’s NTM (and therefore it’s largest triangle ordering) will be largely unchanged. The previous sort order/pivot point is likely to be close to correct, and is a good starting point for an incremental algorithm.

The pivot search can be improved in dozens of ways: such as reducing the list during some iterations, (by sorting the list into ‘above’ and ‘below’ lists, counting them, and sequestering one side) taking averages or histograms to use as pivots instead of pure recursive approximation, or even random sampling.

If the triangle matrices become infeasible, then we can get very good approximations by dropping to a lower resolution. We might even use the low-res information to sparsely render a high-resolution matrix.

7 Observations

Why does it work so well? For that reason, why does it work at all? We seem to spend a lot of computation time looking at the input glyph for triangles, and very little effort comparing glyphs geometrically, which would seem to be the whole point.

There is a statistical method called “Principal Component Analysis” (PCA) which takes a cloud of point samples and assumes it is a (usually Gaussian) distribution around a mean in the form of a squashed ovoid. (In contrast, The usual statistical mean and standard deviation tests assume a simple ‘spherical’ distribution around the average.) A 2-D PCA analyzes the cloud of samples and returns two vectors which represent the cloud’s long axis and short axis, both in direction and size. Once these principal components are known, the sample cloud can be mapped onto other set for comparison.

One method of calculating the PCA is performing singular value decomposition to deconstruct the full covariance matrix as a series of simpler transformations. The algorithm essentially tries to force the squashed ovoid back into a standard distribution through a series of affine transforms which are generated from internal features of the sample points. The transform which does this best naturally contains the principal axes.

This sounds very similar in concept to Squiggle, and for good reason. The largest-area triangle search is looking for the biggest pair of mostly-orthogonal axes which can be made from the input points; it’s essentially looking for principal components, as important and essential a numeric property as the average or standard deviation. The results are therefore useful for comparing against all templates; we don’t even have to know what they are yet!

8 Conclusion

Squiggle fulfills all the requirements to be a practical glyph recognizer that can handle large dynamic libraries of arbitrary user-programmed template glyphs. The major computational step of the algorithm is fixed time and is it’s own natural approximation at lower resolutions, so performance tradeoffs can make the algorithm suitable for a wide range of hardware. The algorithm is fast, robust, numerically stable, and handles it’s own worst cases elegantly. There are no bad glyphs that give the recognizer trouble, and it can gracefully tune it’s approximations to cope with changes in available resources. It’s characteristics make it almost ideal as a part of a real-time computer input system using pen-like pointer devices.

References

1. Wobbrock, J.O., Wilson, A., Li, Y.: Gestures without libraries, toolkits or training: a $1 recognizer for user interface prototypes. UIST’07 (2007) 159–168
2. Li Y.: Protractor: A Fast and Accurate Gesture Recognizer. CHI 2010: ACM Conference on Human Factors in Computing Systems. (2010) 2169–2172