Gestures without Libraries, Toolkits or Training: A $1 Recognizer for User Interface Prototypes

Jacob O. Wobbrock
The Information School
University of Washington
Mary Gates Hall, Box 352840
Seattle, WA 98195-2840
wobbrock@u.washington.edu

Andrew D. Wilson
Microsoft Research
One Microsoft Way
Redmond, WA 98052
awilson@microsoft.com

Yang Li
Computer Science & Engineering
University of Washington
The Allen Center, Box 352350
Seattle, WA 98195-2350
yangli@cs.washington.edu

ABSTRACT

Although mobile, tablet, large display, and tabletop computers increasingly present opportunities for using pen, finger, and wand gestures in user interfaces, implementing gesture recognition largely has been the privilege of pattern matching experts, not user interface prototypers. Although some user interface libraries and toolkits offer gesture recognizers, such infrastructure is often unavailable in design-oriented environments like Flash, scripting environments like JavaScript, or brand new off-desktop prototyping environments. To enable novice programmers to incorporate gestures into their UI prototypes, we present a "$1 recognizer" that is easy, cheap, and usable almost anywhere in 100 lines of code. In a study comparing our $1 recognizer, Dynamic Time Warping, and the Rubine classifier on user-supplied gestures, we found that $1 obtains over 97% accuracy with only 1 loaded template and 99% accuracy with 3+ loaded templates. These results were nearly identical to DTW and superior to Rubine. In addition, we found that medium-speed gestures, in which users balanced speed and accuracy, were recognized better than slow or fast gestures for all three recognizers. We also discuss the effect that the number of templates or training examples has on recognition, the score falloff along recognizers' *N*-best lists, and results for individual gestures. We include a detailed pseudocode listing of $1 to aid development, inspection, extension, and testing.

Author Keywords

Gesture recognition, unistrokes, strokes, marks, symbols, recognition rates, statistical classifiers, Rubine, Dynamic Time Warping, user interfaces, rapid prototyping.

ACM Classification Keywords

H.5.2. [**Information interfaces and presentation**]: User interfaces – *Input devices and strategies*. I.5.2. [**Pattern recognition**]: Design methodology – *Classifier design and evaluation*. I.5.5. [**Pattern recognition**]: Implementation – *Interactive systems*.

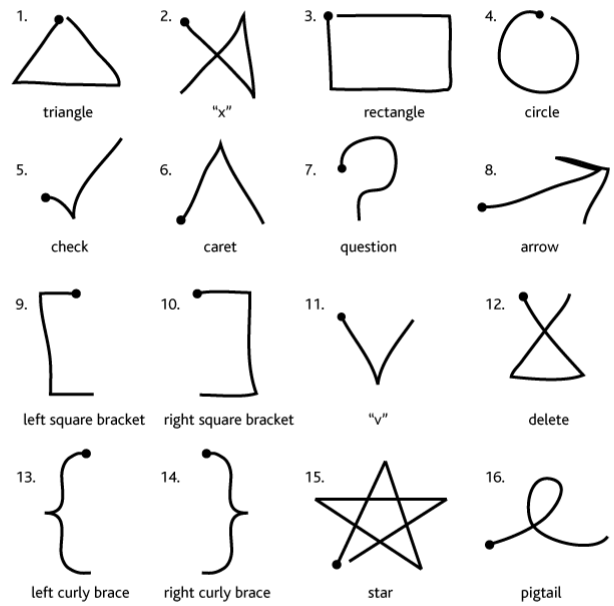

Figure 1. Unistroke gestures useful for making selections, executing commands, or entering symbols. This set of 16 was used in our study of $1, DTW [18,28], and Rubine [23].

INTRODUCTION

Pen, finger, and wand gestures are increasingly relevant to many new user interfaces for mobile, tablet, large display, and tabletop computers [2,5,7,10,16,31]. Even some desktop applications support mouse gestures. The Opera Web Browser, for example, allows gestures to navigate and manage windows.[1] As new computing platforms and new user interface concepts are explored, the opportunity for using gestures made by pens, fingers, wands, or other path-making instruments is likely to grow, and with it, interest from user interface designers and rapid prototypers in using gestures in their projects.

However, along with the naturalness of gestures comes inherent ambiguity, making gesture recognition a topic of interest to experts in artificial intelligence (AI) and pattern matching. To date, designing and implementing gesture recognition largely has been the privilege of experts in these fields, not in human-computer interaction (HCI),

---

[1] http://www.opera.com/products/desktop/mouse/

whose primary concerns are usually not algorithmic, but interactive. This has perhaps limited the extent to which novice programmers, human factors specialists, and user interface prototypers have considered gesture recognition a viable addition to their projects, especially if they are doing the algorithmic work themselves.

As an example, consider a sophomore computer science major with an interest in user interfaces. Although this student may be a capable programmer, it is unlikely that he has been immersed in Hidden Markov Models [1,3,25], neural networks [20], feature-based statistical classifiers [4,23], or dynamic programming [18,28] at this point in his career. In developing a user interface prototype, this student may wish to use Director, Flash, Visual Basic, JavaScript or a brand new tool rather than an industrial-strength environment suitable to production-level code. Without a recognition library for these tools, his options for adding gestures are rather limited. He can dig into pattern matching journals, try to devise an ad-hoc algorithm of his own [4,19,31], ask for considerable help, or simply choose not to have gestures.

We are certainly not the first to note this issue in HCI. Prior work has attempted to provide gesture recognition for user interfaces through the use of libraries and toolkits [6,8,12, 17]. However, libraries and toolkits cannot help where they do not exist, and many of today's rapid prototyping tools may not yet have them available.

On the flip side, ad-hoc recognizers also have their drawbacks. By "ad-hoc" we mean recognizers that use heuristics specifically tuned to a predefined set of gestures [4,19,31]. Implementing ad-hoc recognizers can be challenging if the number of gestures is very large, since gestures tend to "collide" in feature-space [14]. Ad-hoc recognition also prevents application end-users from defining their own gestures at runtime, since new heuristics would need to be added.

To facilitate the incorporation of gestures into user interface prototypes, we present a *$1 recognizer* that is easy, cheap, and usable almost anywhere. The recognizer is very simple, involving only basic geometry and trigonometry. It requires about 100 lines of code for both gesture definition and recognition. It supports configurable rotation, scale, and position invariance, does not require feature selection or training examples, is resilient to variations in input sampling, and supports high recognition rates, even after only one representative example. Although $1 has limitations as a result of its simplicity, it offers excellent recognition rates for the types of symbols and strokes that can be useful in user interfaces.

In order to evaluate $1, we conducted a controlled study of it and two other recognizers on the 16 gesture types shown in Figure 1. Our study used 4800 pen gestures provided by 10 subjects on a Pocket PC. Some of the questions we address in this paper are: How well does $1 recognize user interface gestures compared with two more complex algorithms used in HCI? How does recognition improve as the number of templates or training examples increases? How do gesture articulation speeds affect recognition? How do recognizers' scores degrade as we move down their *N*-best lists? Which gestures do users prefer?

Along with answering these questions, the contributions of this paper are:

1. To present an easy-to-implement gesture recognition algorithm for use by UI prototypers who may have little or no knowledge of pattern recognition. This includes an efficient scheme for rotation invariance;
2. To empirically compare $1 to more advanced, theoretically sophisticated algorithms, and to show that $1 is successful in recognizing certain types of user interface gestures, like those shown in Figure 1;
3. To give insight into which user interface gestures are "best" in terms of human and recognizer performance, and human subjective preference.

We are interested in recognizing paths delineated by users interactively, so we restrict our focus to unistroke gestures that unfold over time. The gestures we used for testing (Figure 1) are based on those found in other interactive systems [8,12,13,27]. It is our hope that user interface designers and prototypers wanting to add gestures to their projects will find the $1 recognizer easy to understand, build, inspect, debug, and extend, especially in design-oriented environments where gestures are typically scarce.

RELATED WORK

Various approaches to gesture recognition were mentioned in the introduction, including Hidden Markov Models (HMMs) [1,3,25], neural networks [20], feature-based statistical classifiers [4,23], dynamic programming [18,28], and ad-hoc heuristic recognizers [4,19,31]. All have been used extensively in domains ranging from on-line handwriting recognition to off-line diagram recognition. Space precludes a full treatment. For in-depth reviews, readers are directed to prior surveys [21,29].

For recognizing simple user interface strokes like those shown in Figure 1, many of these sophisticated methods are left wanting. Some must be trained with numerous examples, like HMMs, neural networks, and statistical classifiers, making them less practical for UI prototypes in which application end-users define their own strokes. These algorithms are also difficult to program and debug. Even Rubine's popular classifier [23] requires programmers to compute matrix inversions, discriminant values, and Mahalanobis distances, which can be obstacles. Dynamic programming methods are computationally expensive and sometimes *too* flexible in matching [32], and although improvements in speed are possible [24], these improvements put the algorithms well beyond the reach of most UI designers and prototypers. Finally, ad-hoc methods scale poorly and usually do not permit adaptation or definition of new gestures by application end-users.

Previous efforts at making gesture recognition more accessible have been through the inclusion of gesture recognizers in user interface toolkits. Artkit [6] and Amulet [17] support the incorporation of gesture recognizers in user interfaces. Amulet's predecessor, Garnet, was extended with Agate [12], which used the Rubine classifier [23]. More recently, SATIN [8] combined gesture recognition with other ink-handling support for developing informal pen-based UIs. Although these toolkits are powerful, they cannot help in most new prototyping environments because they are not available.

Besides research toolkits, some programming libraries offer APIs for supporting gesture recognition on specific platforms. An example is the *Siger* library for Microsoft's Tablet PC [27], which allows developers to define gestures for their applications. The Siger recognizer works by turning strokes into directional tokens and matching those tokens using regular expressions and heuristics. As with toolkits, libraries like *Siger* are powerful; but they are not useful where they do not exist.

THE $1 GESTURE RECOGNIZER

In this section, we describe the $1 gesture recognizer. A pseudocode listing of the algorithm is given in Appendix A.

Characterizing the Challenge

A user's gesture results in a set of *candidate points* C, and we must determine which set of previously recorded *template points* T_i it most closely matches. Candidate and template points are usually obtained through interactive means by some path-making instrument moving through a position-sensing region. Thus, candidate points are sampled at a rate determined by the capabilities of the sensing hardware and the input software. These factors and human variability mean that points in similar C and T_i will rarely "line up" so as to be easily comparable. Consider the two pairs of gestures made by the same subject in Figure 2.

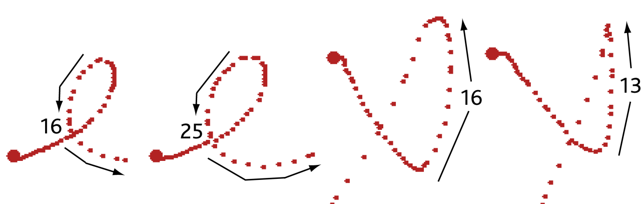

Figure 2. Two pairs of fast (~600 ms) gestures made by one subject with a stylus on a Pocket PC. The number of points in corresponding sections are labeled. Clearly, a 1:1 comparison of points is insufficient.

In examining these pairs of "pigtail" and "x", we see that they are different sizes and contain different numbers of points. This distinction presents a challenge to recognizers. Also, under rotation and scale invariance, the pigtails can be made similar to the "x" gestures using a 90° clockwise turn. Reflecting on these issues and on our desire for simplicity, we formulated the following criteria for our $1 recognizer. The recognizer must:

1. be resilient to variations in sampling due to movement speed or sensing;
2. support optional and configurable rotation, scale, and position invariance;
3. require no advanced mathematical techniques (e.g., matrix inversions, derivatives, integrals);
4. be easily written in few lines of code;
5. be fast enough for interactive purposes (no lag);
6. allow developers and application end-users to "teach" it new gestures with only one example;
7. return an N-best list with sensible [0..1] scores that are independent of the number of input points;
8. provide recognition rates that are competitive with more complex algorithms previously used in HCI to recognize the types of gestures shown in Figure 1.

With these goals in mind, we describe a *$1 recognizer* in the next section. The recognizer has four steps, which correspond to those offered as pseudocode in Appendix A.

A Simple Four-Step Algorithm

Raw input points, whether those of gestures meant to serve as templates, or those of candidate gestures attempting to be recognized, are initially treated the same: they are resampled, rotated once, scaled, and translated. Candidate points C are then scored against each set of template points T_i over a series of angular adjustments to C that find its optimal angular alignment to T_i. Each of these steps is explained in more detail below.

Step 1: Resample the Point Path

As noted in the previous section, gestures in user interfaces are sampled at a rate determined by the sensing hardware and input software. Thus, movement speed will have a clear effect on the number of input points in a gesture (Figure 3).

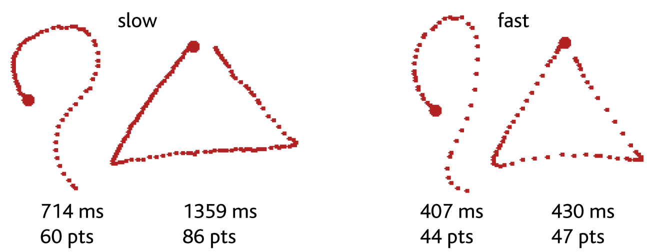

Figure 3. A slow and fast question mark and triangle made by subjects using a stylus on a Pocket PC. Note the considerable time differences and resulting number of points.

To make gesture paths directly comparable even at different movement speeds, we first resample gestures such that the path defined by their original M points is defined by N equidistantly spaced points (Figure 4). Using an N that is too low results in a loss of precision, while using an N that is too high adds time to path comparisons. In practice, we found $N = 64$ to be adequate, as was any $32 \leq N \leq 256$.

Although resampling is not particularly common compared to other techniques (e.g., filtering), we are not the first to use it. Some prior handwriting recognition systems have also resampled stroke paths [21,29]. Also, the *SHARK*[2] system resampled its strokes [11]. However, SHARK[2] is not fully rotation, scale, and position invariant, since gestures are defined atop the soft keys of an underlying stylus keyboard, making complete rotation, scale, and position invariance undesirable. Interestingly, the original SHARK system [32] utilized Tappert's elastic matching technique [28], but SHARK[2] discontinued its use to improve accuracy. However, in mentioning this choice, the SHARK[2] paper [11] provided no specifics as to the comparative performance of these techniques. We now take this step, offering an evaluation of an elastic matching technique (DTW) and our simpler resampling technique ($1), extending both with efficient rotation invariance.

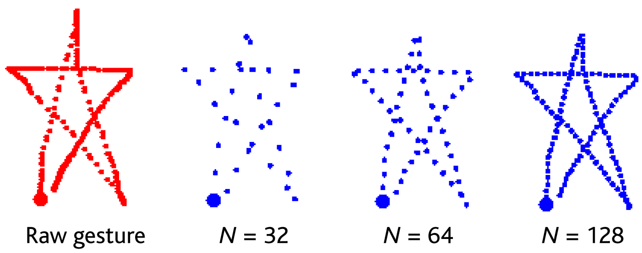

Figure 4. A star gesture resampled to N = 32, 64, and 128 points.

To resample, we first calculate the total length of the M-point path. Dividing this length by (N–1) gives the length of each increment, I, between N new points. Then the path is stepped through such that when the distance covered exceeds I, a new point is added through linear interpolation. The RESAMPLE function in Appendix A gives a listing.

At the end of this step, the candidate gesture and any loaded templates will all have exactly N points. This will allow us to measure the distance from $C[k]$ to $T_i[k]$ for k = 1 to N.

Step 2: Rotate Once Based on the "Indicative Angle"

When given two paths of ordered points, there is no simple closed-form solution for determining the angle to which one set of points should be rotated to best align with the other [9]. Although there are complex techniques based on *moments*, these are not made to handle ordered points and sometimes fail [26]. Our $1 algorithm therefore searches over the space of possible angles for the best alignment between two point-paths. Although for many complex recognition algorithms an iterative process is prohibitively expensive [9], our simple $1 algorithm is fast enough to make iteration useful. In fact, even naïvely rotating the candidate gesture by +1° for all 360° is fast enough for interactive purposes with 30 templates loaded. However, we can improve upon this brute force scheme with a "rotation trick" that makes finding the optimal angle faster.

First, we find a gesture's *indicative angle*, which we define as the angle formed between the centroid of the gesture $(\bar{x},\bar{y})$ and the gesture's first point. Then we rotate the gesture so that this angle is at 0° (Figure 5). The ROTATE-TO-ZERO function in Appendix A gives a listing. An analysis of $1's rotation invariance scheme is discussed in the next section.

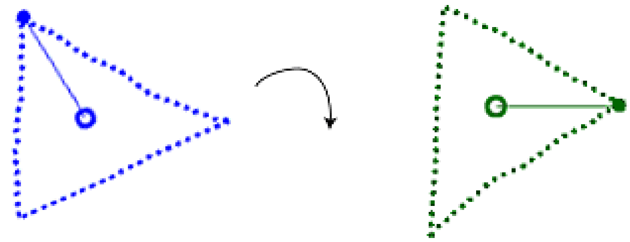

Figure 5. Rotating a triangle so that its "indicative angle" is at 0° (straight right). This approximates finding the best angular match.

Step 3: Scale and Translate

After rotation, the gesture is scaled to a *reference square*. By scaling to a square, we are scaling non-uniformly. This will allow us to rotate the candidate about its centroid and safely assume that changes in pairwise point-distances between C and T_i are due only to rotation, not aspect ratio. Of course, non-uniform scaling introduces some limitations, which will be discussed in a future section. The SCALE-TO-SQUARE function in Appendix A gives a listing.

After scaling, the gesture is translated to a reference point. For simplicity, we choose to translate the gesture so that its centroid $(\bar{x},\bar{y})$ is at (0,0). The TRANSLATE-TO-ORIGIN function gives a listing in Appendix A.

Step 4: Find the Optimal Angle for the Best Score

At this point, all candidates C and templates T_i have been treated the same: resampled, rotated once, scaled, and translated. In our implementations, we apply the above steps when templates' points are read in. For candidates, we apply these steps after they are drawn. Then we take Step 4, which actually performs the recognition. RECOGNIZE and its associated functions give a listing in Appendix A.

Using Equation 1, a candidate C is compared to each stored template T_i to find the average distance d_i between corresponding points:

$$d_i = \frac{\sum_{k=1}^{N} \sqrt{\left(C[k]_x - T_i[k]_x\right)^2 + \left(C[k]_y - T_i[k]_y\right)^2}}{N} \qquad (1)$$

Equation 1 defines d_i, the *path-distance* between C and T_i. The template T_i with the least path-distance to C is the result of the recognition. This minimum path-distance d_i^* is converted to a [0..1] score using:

$$score = 1 - \frac{d_i^*}{\frac{1}{2}\sqrt{size^2 + size^2}} \qquad (2)$$

In Equation 2, *size* is the length of the reference square to which all gestures were scaled in Step 3. Thus, the denominator is half of the length of the bounding box diagonal, which serves as a limit to the path-distance.

When comparing C to each T_i, the result of each comparison must be made using the best angular alignment of C and T_i. In Step 2, rotating C and T_i once using their indicative angles only *approximated* their best angular alignment. However, C may need to be rotated further to find the least path-distance to T_i. Thus, the "angular space" must be searched for a global minimum, as described next.

An Analysis of Rotation Invariance

As stated, there is no simple closed-form means of rotating C into T_i such that their path-distance is minimized. For simplicity, we take a "seed and search" approach that minimizes iterations while finding the best angle. This is simpler than the approach used by Kara and Stahovich [9], which used polar coordinates and had to employ weighting factors based on points' distances from the centroid.

After rotating the indicative angles of all gestures to 0° (Figure 5), there is no guarantee that two gestures C and T_i will be aligned optimally. We therefore must fine-tune C's angle so that C's path-distance to T_i is minimized. As mentioned, a brute force scheme could rotate C by +1° for all 360° and take the best result. Although this method is guaranteed to find the optimal angle to within 0.5°, it is unnecessarily slow and could be a problem in processor-intensive applications (e.g., games).

We manually examined a stratified sample of 480 similar[2] gesture-pairs from our subjects, finding that there was always a global minimum and no local minima in the graphs of path-distance as a function of angle (Figure 6a). Therefore, a first improvement over the brute force approach would be hill climbing: rotate C by ±1° for as long as C's path-distance to T_i decreases. For our sample of 480 pairs, we found that hill climbing always found the global minimum, requiring 7.2 (SD=5.0) rotations on average. The optimal angle was, on average, just 4.2° (5.0°) away from the indicative angle, indicating that the indicative angle was indeed a good approximation of angular alignment for similar gestures. (That said, there *were* a few matches found up to ±44° away.) The path-distance after the single indicative angle rotation was only 10.9% (13.0%) higher than optimal.

However, although hill climbing is efficient for similar gestures, it is not efficient for dissimilar ones. In a second stratified sample of 480 dissimilar gesture-pairs, we found that the optimal angle was an average of 63.6° (SD=50.8°) away from the indicative angle. This required an average of 53.5 (45.7) rotations using hill climbing. The average path-distance after just rotating the indicative angle was 15.8% (14.7%) higher than optimal. Moreover, of the 480 dissimilar pairs, 52 of them, or 10.8%, had local minima in their path-distance graphs (Figure 6b), which means that hill climbing would not be guaranteed to succeed. However, local minima alone are not concerning, since suboptimal scores for dissimilar gestures only *decrease* our chances of getting unwanted matches. The issue of greater concern is the number of iterations required, especially if many templates are loaded.

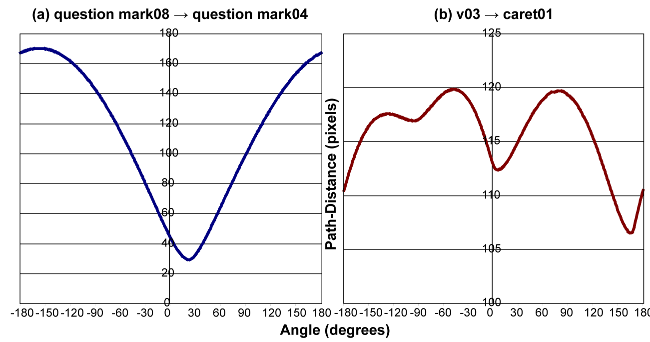

Figure 6. Path-distance as a function of angular rotation away from the 0° indicative angle (centered y-axis) for **(a)** similar gestures and **(b)** dissimilar gestures.

Since there will be many more comparisons of C to dissimilar T_i than to similar T_i, we chose to use a strategy that performs slightly worse than hill climbing for similar gestures but far better for dissimilar ones. The strategy is *Golden Section Search (GSS)* [22], a simple, efficient algorithm that finds a minimum value in a range by dividing that range using the Golden Ratio $\varphi = 0.5(-1 + \sqrt{5})$. In our sample of 480 similar gestures, no match was found outside ±45° from the indicative angle, so we use GSS bounded by ±45° and a 2° threshold. This guarantees that GSS will finish after exactly 10 iterations, regardless of whether two gestures are similar or dissimilar. For our 480 similar gesture-pairs, the distance returned by GSS was, on average, within 0.2% (0.4%) of the optimal, while the angle returned was within 0.5°. Furthermore, although GSS loses |10.0 – 7.2| = 2.8 iterations to hill climbing for similar gestures, it gains |10.0 – 53.5| = 43.5 iterations for dissimilar ones. Thus, in a recognizer with 10 templates for each of 16 gesture types (160 templates), GSS would require 160×10=1600 iterations to recognize a candidate, compared to 7.2×10 + 53.5×150=8097 iterations for hill climbing—an 80.2% savings. (Incidentally, with the aforementioned brute force algorithm, 160×360=57,600 iterations would be needed.) The DISTANCE-AT-BEST-ANGLE function in Appendix A implements GSS.

Limitations of the $1 Recognizer

Simple techniques often have limitations, and the $1 recognizer is no exception. The $1 recognizer is a geometric template matcher, which means that candidate strokes are compared to previously stored templates, and the result produced is the closest match in 2-D Euclidean space. To facilitate pairwise point comparisons, the default $1 algorithm is rotation, scale, and position invariant. While this provides tolerance to gesture variation, it means that $1 cannot distinguish gestures whose identities depend on specific orientations, aspect ratios, or locations. For example, separating squares from rectangles, circles from ovals, or up-arrows from down-arrows is not possible

[2] By "similar," we mean gestures subjects intended to be the same.

without modifying the algorithm. Furthermore, horizontal and vertical lines are abused by non-uniform scaling; if these are to be recognized, their bounding box can be tested to see if its minor dimension exceeds a threshold. If it does not, the line can be scaled uniformly so that its major dimension matches the reference square. Finally, the $1 algorithm does not utilize timing information, so gestures cannot be differentiated on the basis of speed. Prototypers wishing to differentiate gestures on these bases will need to understand and modify the $1 algorithm. For example, if scale invariance is not desired, the candidate C can be resized to match each *unscaled* template T_i before comparison. Or if rotation invariance is not desired, C and T_i can be compared without rotating to the indicative angle. Such exceptions can be made on a per gesture (T_i) basis.

Accommodating gesture variability is a key property of any recognizer. Feature-based recognizers, like the Rubine classifier [23], can capture properties of a gesture that matter for recognition if the features are properly chosen. Knowledgeable users can add or remove features to distinguish troublesome gestures, but because of the difficulty in choosing good features, it is usually necessary to define a class by its summary statistics over a set of examples. In Rubine's case, this has the undesirable consequence that there is no guarantee that even the training examples themselves will be correctly recognized if they are entered as candidates. Such unpredictable behavior may be a serious limitation for $1's audience.

In contrast, to handle variation in $1, prototypers or application end-users can define new templates that exemplify the variation they desire using a single name. For example, different types of arrows can be recognized together as "arrow" with just a few templates (Figure 7). This aliasing is a simple and direct means of accommodating variation among gestures in a way that users can understand. If a user finds that a new arrow he makes is not properly recognized, he can simply add that arrow as a new template of type "arrow" and it will be recognized from then on. Of course, the success of this approach will depend on what other templates are loaded.

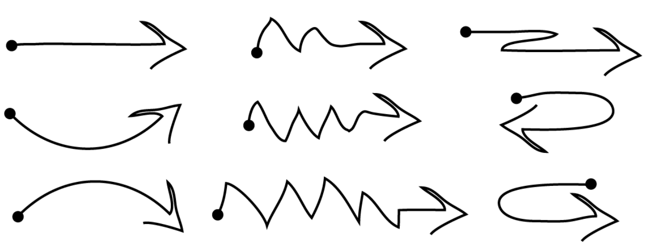

Figure 7. Defining multiple instances of "arrow" allows variability in the way candidate arrows can be made and matched. Note that orientation is not an issue, since $1 is rotation invariant.

EVALUATION

To compare the performance of our $1 recognizer to more complex recognizers used in HCI, we conducted an evaluation using 4800 gestures collected from 10 subjects.

Method

Subjects

Ten subjects were recruited from the local community. Five were students. Eight were female. Three had technical degrees in science, engineering, or computing. The average age was 26.1 (*SD*=6.4).

Apparatus

Using an HP iPAQ h4355 Pocket PC with a 2.25"×3.00" screen, we presented the gestures shown in Figure 1 in random order to subjects. The gestures were based on those used in other user interface systems [8,12,13,27]. Subjects used a pen-sized plastic stylus measuring 6.00" in length to enter gestures on the device. Our Pocket PC application (Figure 8) logged all gestures in a simple XML format containing (x,y) points with millisecond timestamps.

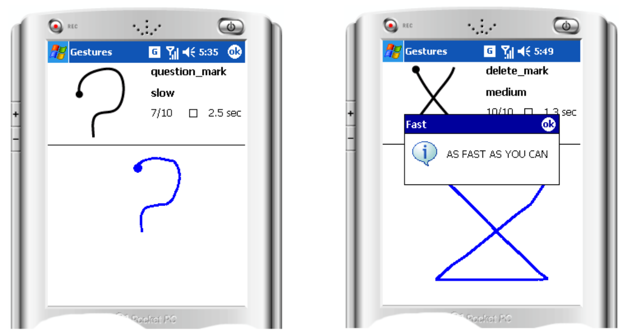

Figure 8. The Pocket PC application used to capture gestures made by subjects. The right image shows the reminder displayed when subjects began the fast speed for the "delete_mark" gesture.

Procedure: Capturing Gestures

For each of the 16 gesture types from Figure 1, subjects entered one practice gesture before beginning three sets of 10 entries at slow, medium, and fast speeds. Messages were presented between each block of slow, medium, and fast gestures to remind subjects of the speed they should use. For slow gestures, they were asked to "be as accurate as possible." For medium gestures, they were asked to "balance speed and accuracy." For fast gestures, they were asked to "go as fast as they can." After entering 16×3×10=480 gestures, subjects were given a chance to rate them on a 1-5 scale (1=disliked a lot, 5=liked a lot).

Procedure: Recognizer Testing

We compared our $1 recognizer to two popular recognizers previously used in HCI. The Rubine classifier [23] has been used widely (e.g., [8,13,14,17]). It relies on training examples from which it extracts and weights features to perform statistical matching. Our version includes the *gdt* [8,14] routines for improving Rubine on small training sets.

We also tested a template matcher based on Dynamic Time Warping (DTW) [18,28]. Like $1, DTW does not extract features from training examples but matches point-paths. Unlike $1, however, DTW relies on dynamic programming, which gives it considerable flexibility in how two point sequences may be aligned.

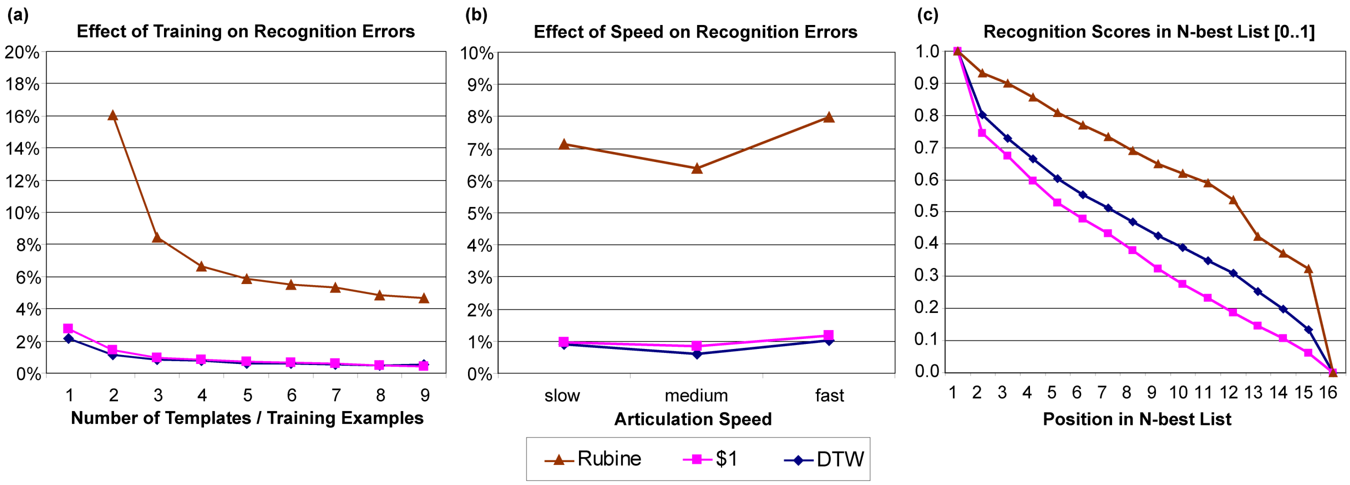

Figure 9. (a) Recognition error rates as a function of templates or training (lower is better). **(b)** Recognition error rates as a function of articulation speeds (lower is better). **(c)** Normalized gesture scores [0..1] for each position along the *N*-best list at 9 training examples.

We extended Rubine and DTW to use $1's rotation invariance scheme. Also, the gestures for Rubine and DTW were scaled to a standard square size and translated to the origin. They were not resampled, since these techniques do not use pairwise point comparisons. Rubine was properly trained *after* these adjustments to gestures were made.

The testing procedure we followed was based on those used for testing in machine learning [15] (pp. 145-150). Of a given subject's 16×10=160 gestures made at a given speed, the number of training examples *E* for each of the 16 gesture types was increased systematically from *E*=1 to 9 for $1 and DTW, and *E*=2 to 9 for Rubine (Rubine fails for *E*=1). In a process repeated 100 times per level of *E*, *E* training examples were chosen randomly for each gesture category. Of the remaining 10–*E* untrained gestures in each category, one was picked at random and tested as the candidate. Over the 100 tests, incorrect outcomes were averaged into a recognition error rate for each gesture type for that subject at that speed.

For a given subject at a given speed, there were 9×16×100=14,400 recognition tests for $1 and DTW, and 8×16×100=12,800 tests for Rubine. These 41,600 tests were done at 3 speeds, for 124,800 total tests per subject. Thus, with 10 subjects, the experiment consisted of 1,248,000 recognition tests. The results of every test were logged, including the entire *N*-best lists.

Design and Analysis

The experiment was a 3-factor within-subjects repeated measures design, with nominal factors for *recognizer* and *articulation speed*, and a continuous factor for *number of training examples*. The outcome measure was *mean recognition errors*. Since errors were rare, the data were highly skewed toward zero and violated ANOVA's normality assumption, even under transformation. However, Poisson regression [30] was well-suited to these data and was therefore used. The overall model was significant ($\chi^2_{(22,N=780)}$=3300.21, p<.0001).

Results

Overall Recognition Performance

$1 and DTW were very accurate overall, with 0.98% (*SD*=3.63) and 0.85% (3.27) recognition errors, respectively. (Equivalently, recognition rates were 99.02% and 99.15%, respectively.) Rubine was less accurate, with 7.17% (10.60) errors. These differences were statistically significant ($\chi^2_{(2,N=780)}$=867.33, p<.0001). $1 and DTW were significantly more accurate than Rubine ($\chi^2_{(1,N=780)}$=668.43, p<.0001), but $1 and DTW were not significantly different from each other ($\chi^2_{(1,N=780)}$=0.13, n.s.).

Effect of Number of Templates / Training Examples

The number of templates or training examples had a significant effect on recognition errors ($\chi^2_{(1,N=780)}$=125.24, p<.0001). As shown in Figure 9a, $1 and DTW improved slightly as the number of templates increased, from 2.73% (*SD*=2.38) and 2.14% (1.76) errors with 1 template to 0.45% (0.64) and 0.54% (0.84) errors with 9 templates, respectively. Rubine's improvement was more pronounced, from 16.03% (5.98) errors with 2 training examples to 4.70% (3.03) errors with 9 training examples. However, this difference only produced a marginal *recognizer*×*training* interaction ($\chi^2_{(2,N=780)}$=4.80, p=0.09).

Effect of Gesture Articulation Speed

Subjects' average speeds for slow, medium, and fast gestures were 1761 (*SD*=567), 1153 (356), and 668 (212) milliseconds. Speed had a significant effect on errors ($\chi^2_{(2,N=780)}$=24.56, p<.0001), with slow, medium, and fast gestures being recognized with 2.84% (4.07), 2.46% (4.09), and 3.22% (4.44) errors, respectively (Figure 9b). All three recognizers were affected similarly, so a *recognizer*×*speed* interaction was not significant ($\chi^2_{(4,N=780)}$=4.52, n.s.).

Scores Along the N-Best List

In recognizing a candidate, all three recognizers produce an *N*-best list containing scores at each position. The actual "result" of the recognition is the head of this list. It is

Gesture	Millseconds			NumPts			$1.00	DTW	Rubine	Subjective
	Slow	Medium	Fast	Slow	Medium	Fast	(error % with 9 training examples)			(1-5)
arrow	1876	1268	768	90	76	61	**0***	**0***	3.70	3.0
caret	1394	931	452	70	59	43	0.33	**0***	2.87	4.0
check	**1028***	**682***	393	**58***	**49***	**37***	0.97	0.93	3.97	4.1
circle	1624	936	496	91	70	50	0.40	0.40	3.13	4.0
delete_mark	1614	1089	616	84	71	55	**0***	**0***	**0.33***	3.2
left_curly_brace	1779	1259	896	81	70	63	**1.67†**	**2.20†**	2.10	**2.0†**
left_square_bracket	1591	1092	678	74	62	51	**0***	**0***	1.17	3.2
pigtail	1441	949	540	87	72	52	**0***	**0***	2.83	**4.4***
question_mark	1269	837	523	70	60	48	1.37	1.83	6.40	2.7
rectangle	2497	1666	916	117	96	70	**0***	**0***	12.87	3.0
right_curly_brace	2060	1429	**1065†**	81	73	66	0.33	0.50	5.47	2.1
right_square_bracket	1599	1044	616	75	62	52	**0***	**0***	5.90	3.4
star	**3375†**	**2081†**	998	**139†**	**110†**	**75†**	**0***	**0***	0.40	3.7
triangle	2041	1288	706	99	78	58	1.03	0.73	**14.80†**	3.6
v	1143	727	**377***	65	53	38	0.83	1.70	6.40	4.1
x	1837	1162	640	91	73	55	0.23	0.40	2.80	3.7
Mean	**1760.5**	**1152.5**	**667.5**	**85.8**	**70.9**	**54.6**	**0.45**	**0.54**	**4.70**	**3.4**
StdDev	**567.3**	**356.3**	**211.6**	**20.1**	**15.2**	**10.7**	**0.55**	**0.75**	**4.06**	**0.7**

Table 1. Results for individual gestures: times (ms), number of points, recognition error rates (%), and subjective ratings (1=dislike a lot, 5=like a lot). For times, number of points, and error rates, minimum values in each column are marked with (*); maximum values are marked with (†). For subjective ratings, the best is marked with (*); the worst is marked with (†). For readability, extra zeroes are omitted for error rates that are exactly 0%.

interesting to examine the falloff that occurs as we move down the *N*-best list, as this gives us a sense of the relative competitiveness of results as they vie for the top position. In general, we prefer a rapid and steady falloff, especially from position 1 to 2, indicating a good separation of scores. Such a falloff would make it easier to set a non-recognition threshold and thus improve overall recognition robustness.

Figure 9c shows the normalized *N*-best falloff for all three recognizers using 9 templates or training examples. The first and last results are defined as scores 1.0 and 0.0, respectively. We can see that $1 falls off the fastest, DTW second, and Rubine third. Note that $1 shows the greatest falloff from position 1 to position 2.

Recognizer Execution Speed

We found that DTW runs noticeably slower than the other techniques. On average, DTW took a whopping 128.26 (*SD*=60.02) minutes to run the 14,400 tests for a given subject's 160 gestures made at a given speed. In contrast, $1 only took 1.59 (0.04) minutes, while Rubine took 2.38 (0.60) minutes. This difference in speed is explained by the fact that DTW's runtime is quadratic in the number of samples. Thus slowly-made gestures are much slower to recognize. As noted, there are procedures to accelerate DTW [24], but these make it a more complicated algorithm, which runs counter to our original motivation for this work.

Differences Among Gestures and Subjective Ratings

Table 1 shows results for individual gestures. Here we can see that "check" and "v" were fast gestures at all speeds, and that "star" and "right_curly_brace" were slow. The "check" had the fewest points at all speeds, while the "star" had the most. With 9 templates or training examples loaded for each gesture type, $1 and DTW had perfect rates for 7 and 8 of 16 gestures, respectively, while Rubine had none. Recognizing the "left_curly_brace" gesture was hardest for $1 and DTW, while for Rubine it was the "triangle". Rubine fared best on "delete_mark" and "star".

Qualitative results show that subjects liked "pigtail", "check", and "v", all fairly fast gestures. They disliked the curly braces and "question_mark". Subjects' comments as to why they liked certain gestures included, "They were easiest to control," and "They were all one fluid motion." Comments on disliked gestures included, "The curly braces made me feel clumsy," and "Gestures with straight lines or 90° angles were difficult to make, especially slowly."

Discussion

From our experiment, it is clear that $1 performs very well for user interface gestures, recognizing them at more than 99% accuracy overall. DTW performed almost identically, but with much longer processing times. Both algorithms did well even with only 1 loaded template, performing above 97% accuracy. With only 3 loaded templates, both algorithms function at about 99.5% of the accuracy they exhibit at 9 templates. This means that designers and application end-users can define gestures using only a few examples and expect reliable recognition. Although DTW's flexibility gave it an edge over $1 with few templates, with 9 templates, that same flexibility causes DTW to falter while $1 takes a small lead. This finding resonates with Kristensson and Zhai's decision to abandon elastic matching due to unwanted flexibility [11].

Another interesting finding is that $1 performs well even without using Golden Section Search. $1's overall error rate after only rotating the indicative angle to 0° was 1.21% (3.88), just 1.21–0.98=0.23% higher than using GSS to search for the optimal angular alignment.

At its best, Rubine performed at about 95% accuracy using 9 training examples for each of the 16 gesture types. This result is comparable to that reported by Rubine himself, who showed 93.5% accuracy on a set of 15 gesture types with 10 training examples per type [23]. Our result may be better due to our use of rotation invariance. Of course, Rubine would improve with more training examples that capture more gesture variability [23].

Although articulation speed significantly affected errors, this is most evident for Rubine. It is interesting that the medium speed resulted in the best recognition rates for all three recognizers. This may be because at slow speeds, subjects were less fluid, and their gestures were made too tentatively; at fast speeds, their gestures were sloppier. At medium speeds, however, subjects' gestures were neither overly careful nor overly sloppy, resulting in higher recognition rates. Subjective feedback resonates with this, where fluid gestures were preferred.

The falloff during $1's *N*-best list is a positive feature of the algorithm, since scores are better differentiated. DTW is nearly the same, but Rubine showed a clear disadvantage.

Recognizers, Recorders, and Gesture Data Set

To facilitate the recording and testing of gestures, we implemented $1, DTW, and Rubine in C#. Each uses an identical XML gesture format, which is also the format written by our Pocket PC recorder (Figure 8). In addition, we implemented a JavaScript version of $1 for use on the web.[3] This version recognizes quite well, even with only 1 template defined. When it does err, the misrecognized gesture can be added immediately as a new template, increasing recognition rates thereafter. In addition to these implementations, we have made our XML gesture set available to other researchers for download and testing.

FUTURE WORK

Although we demonstrate the strengths of a simple $1 recognizer, we have not yet validated its programming ease for novice programmers. A future study could give many recognition algorithms to user interface prototypers to see which are easiest to build, debug, and comprehend.

An interactive extension would be to allow users to correct a failed recognition result using the *N*-best list, and then have their articulated gesture morph some percentage of the way toward the selected template until it *would have been* successfully recognized. This might aid gesture learning.

Further empirical analysis may help justify some algorithmic choices. For example, we currently compute the indicative angle to the *first* point in the gesture, but the first point in a stroke is probably not the most reliable. Is there another point along gesture sequences that generates more consistent estimates of the best angular alignment?

CONCLUSION

We have presented a simple $1 recognizer that is easy, cheap, and usable almost anywhere. Despite its simplicity, it provides optional rotation, scale, and position invariance, and offers 99+% accuracy with only a few loaded templates. It requires no complex mathematical procedures, yet competes with approaches that use dynamic programming and statistical classification. It also employs a rotation invariance scheme that is extensible to other algorithms like DTW and Rubine. It is our hope that this work will support the addition of gestures in mobile, tablet, large display, and tabletop systems, particularly by user interface prototypers who may have previously felt gesture recognition was beyond their reach.

[3] http://faculty.washington.edu/wobbrock/proj/dollar/

REFERENCES

1. Anderson, D., Bailey, C. and Skubic, M. (2004) Hidden Markov Model symbol recognition for sketch-based interfaces. *AAAI Fall Symposium.* Menlo Park, CA: AAAI Press, 15-21.
2. Cao, X. and Balakrishnan, R. (2003) VisionWand: Interaction techniques for large displays using a passive wand tracked in 3D. *Proc. UIST '03*. New York: ACM Press, 173-182.
3. Cao, X. and Balakrishnan, R. (2005) Evaluation of an on-line adaptive gesture interface with command prediction. *Proc. Graphics Interface '05.* Waterloo, Ontario: CHCCS, 187-194.
4. Cho, M.G. (2006) A new gesture recognition algorithm and segmentation method of Korean scripts for gesture-allowed ink editor. *Information Sciences 176* (9), 1290-1303.
5. Guimbretière, F., Stone, M. and Winograd, T. (2001) Fluid interaction with high-resolution wall-size displays. *Proc. UIST '01*. New York: ACM Press, 21-30.
6. Henry, T.R., Hudson, S.E. and Newell, G.L. (1990) Integrating gesture and snapping into a user interface toolkit. *Proc. UIST '90*. New York: ACM Press, 112-122.
7. Hinckley, K., Ramos, G., Guimbretiere, F., Baudisch, P. and Smith, M. (2004) Stitching: Pen gestures that span multiple displays. *Proc. AVI '04*. New York: ACM Press, 23-31.
8. Hong, J.I. and Landay, J.A. (2000) SATIN: A toolkit for informal ink-based applications. *Proc. UIST '00.* New York: ACM Press, 63-72.
9. Kara, L.B. and Stahovich, T.F. (2004) An image-based trainable symbol recognizer for sketch-based interfaces. *AAAI Fall Symposium.* Menlo Park, CA: AAAI Press, 99-105.
10. Karlson, A.K., Bederson, B.B. and SanGiovanni, J. (2005) AppLens and LaunchTile: Two designs for one-handed thumb use on small devices. *Proc. CHI '05*. New York: ACM Press, 201-210.
11. Kristensson, P. and Zhai, S. (2004) SHARK2: A large vocabulary shorthand writing system for pen-based computers. *Proc. UIST '04*. New York: ACM Press, 43-52.
12. Landay, J. and Myers, B.A. (1993) Extending an existing user interface toolkit to support gesture recognition. *Adjunct Proc. CHI '93*. New York: ACM Press, 91-92.
13. Lin, J., Newman, M.W., Hong, J.I. and Landay, J.A. (2000) DENIM: Finding a tighter fit between tools and practice for web site design. *Proc. CHI '00*. New York: ACM Press, 510-517.
14. Long, A.C., Landay, J.A. and Rowe, L.A. (1999) Implications for a gesture design tool. *Proc. CHI '99*. New York: ACM Press, 40-47.
15. Mitchell, T.M. (1997) *Machine Learning*. New York: McGraw-Hill.
16. Morris, M.R., Huang, A., Paepcke, A. and Winograd, T. (2006) Cooperative gestures: Multi-user gestural interactions for co-located groupware. *Proc. CHI '06*. New York: ACM Press, 1201-1210.
17. Myers, B.A., McDaniel, R.G., Miller, R.C., Ferrency, A.S., Faulring, A., Kyle, B.D., Mickish, A., Klimovitski, A. and Doane, P. (1997) The Amulet environment: New models for effective user interface software development. *IEEE Trans. Software Engineering 23* (6), 347-365.
18. Myers, C.S. and Rabiner, L.R. (1981) A comparative study of several dynamic time-warping algorithms for connected word recognition. *The Bell System Technical J. 60* (7), 1389-1409.
19. Notowidigdo, M. and Miller, R.C. (2004) Off-line sketch interpretation. *AAAI Fall Symposium.* Menlo Park, CA: AAAI Press, 120-126.
20. Pittman, J.A. (1991) Recognizing handwritten text. *Proc. CHI '91*. New York: ACM Press, 271-275.
21. Plamondon, R. and Srihari, S.N. (2000) On-line and off-line

handwriting recognition: A comprehensive survey. *IEEE Trans. Pattern Analysis & Machine Int. 22* (1), 63-84.

22. Press, W.H., Teukolsky, S.A., Vetterling, W.T. and Flannery, B.P. (1992) *Numerical Recipes in C.* Cambridge Univ. Press.
23. Rubine, D. (1991) Specifying gestures by example. *Proc. SIGGRAPH '91*. New York: ACM Press, 329-337.
24. Salvador, S. and Chan, P. (2004) FastDTW: Toward accurate dynamic time warping in linear time and space. *3rd Wkshp. on Mining Temporal and Sequential Data, ACM KDD '04.* Seattle, Washington (August 22-25, 2004).
25. Sezgin, T.M. and Davis, R. (2005) HMM-based efficient sketch recognition. *Proc. IUI '05.* New York: ACM Press, 281-283.
26. Stojmenović, M., Nayak, A. and Zunic, J. (2006) Measuring linearity of a finite set of points. *Proc. CIS '06.* Los Alamitos, CA: IEEE Press, 1-6.
27. Swigart, S. (2005) Easily write custom gesture recognizers for your Tablet PC applications. *Tablet PC Technical Articles.*
28. Tappert, C.C. (1982) Cursive script recognition by elastic matching. *IBM J. of Research & Development 26* (6), 765-771.
29. Tappert, C.C., Suen, C.Y. and Wakahara, T. (1990) The state of the art in online handwriting recognition. *IEEE Trans. Pattern Analysis & Machine Int. 12* (8), 787-808.
30. Vermunt, J.K. (1997) *Log-linear Models for Event Histories.* Thousand Oaks, CA: Sage Publications.
31. Wilson, A.D. and Shafer, S. (2003) XWand: UI for intelligent spaces. *Proc. CHI '03.* New York: ACM Press, 545-552.
32. Zhai, S. and Kristensson, P. (2003) Shorthand writing on stylus keyboard. *Proc. CHI '03.* New York: ACM Press, 97-104.

APPENDIX A – $1 GESTURE RECOGNIZER

Step 1. Resample a *points* path into *n* evenly spaced points.

RESAMPLE(*points*, *n*)
1 $I \leftarrow$ PATH-LENGTH(*points*) / $(n-1)$
2 $D \leftarrow 0$
3 $newPoints \leftarrow points_0$
4 **foreach** point p_i for $i \geq 1$ in *points* **do**
5 $d \leftarrow$ DISTANCE($p_{i\text{-}1}$, p_i)
6 **if** $(D + d) \geq I$ **then**
7 $q_x \leftarrow p_{i\text{-}1_x} + ((I - D) / d) \times (p_{i_x} - p_{i\text{-}1_x})$
8 $q_y \leftarrow p_{i\text{-}1_y} + ((I - D) / d) \times (p_{i_y} - p_{i\text{-}1_y})$
9 APPEND(*newPoints*, *q*)
10 INSERT(*points*, *i*, *q*) // *q* will be the next p_i
11 $D \leftarrow 0$
12 **else** $D \leftarrow D + d$
13 **return** *newPoints*

PATH-LENGTH(*A*)
1 $d \leftarrow 0$
2 **for** *i* **from** 1 **to** $|A|$ **step** 1 **do**
3 $d \leftarrow d +$ DISTANCE($A_{i\text{-}1}$, A_i)
4 **return** *d*

Step 2. Rotate *points* so that their "indicative angle" is at 0°.

ROTATE-TO-ZERO(*points*)
1 $c \leftarrow$ CENTROID(*points*) // computes $(\bar{x}, \bar{y})$
2 $\theta \leftarrow$ ATAN $(c_y - points_{0_y}, c_x - points_{0_x})$ // for $-\pi \leq \theta \leq \pi$
3 $newPoints \leftarrow$ ROTATE-BY(*points*, $-\theta$)
4 **return** *newPoints*

ROTATE-BY(*points*, θ)
1 $c \leftarrow$ CENTROID(*points*)
2 **foreach** point *p* in *points* **do**
3 $q_x \leftarrow (p_x - c_x)\ \mathrm{COS}\ \theta - (p_y - c_y)\ \mathrm{SIN}\ \theta + c_x$
4 $q_y \leftarrow (p_x - c_x)\ \mathrm{SIN}\ \theta + (p_y - c_y)\ \mathrm{COS}\ \theta + c_y$
5 APPEND(*newPoints*, *q*)
6 **return** *newPoints*

Step 3. Scale *points* so that the resulting bounding box will be of $size^2$ dimension; then translate *points* to the origin. BOUNDING-BOX returns a rectangle according to (min_x, min_y), (max_x, max_y). For gestures serving as templates, Steps 1-3 should be carried out on the raw points once. For candidates, Steps 1-4 should be used right after the candidate is articulated.

SCALE-TO-SQUARE(*points*, *size*)
1 $B \leftarrow$ BOUNDING-BOX(*points*)
2 **foreach** point *p* in *points* **do**
3 $q_x \leftarrow p_x \times (size / B_{width})$
4 $q_y \leftarrow p_y \times (size / B_{height})$
5 APPEND(*newPoints*, *q*)
6 **return** *newPoints*

TRANSLATE-TO-ORIGIN(*points*)
1 $c \leftarrow$ CENTROID(*points*)
2 **foreach** point *p* in *points* **do**
3 $q_x \leftarrow p_x - c_x$
4 $q_y \leftarrow p_y - c_y$
5 APPEND(*newPoints*, *q*)
6 **return** *newPoints*

Step 4. Match *points* against a set of *templates*. The *size* variable on line 7 of RECOGNIZE refers to the *size* passed to SCALE-TO-SQUARE in Step 3. The symbol φ equals $\frac{1}{2}(-1 + \sqrt{5})$. We use $\theta = \pm 45°$ and $\theta_\Delta = 2°$ on line 3 of RECOGNIZE. Due to using RESAMPLE, we can assume that *A* and *B* in PATH-DISTANCE contain the same number of points, i.e., $|A| = |B|$.

RECOGNIZE(*points*, *templates*)
1 $b \leftarrow +\infty$
2 **foreach** template *T* in *templates* **do**
3 $d \leftarrow$ DISTANCE-AT-BEST-ANGLE(*points*, *T*, $-\theta$, θ, θ_Δ)
4 **if** $d < b$ **then**
5 $b \leftarrow d$
6 $T' \leftarrow T$
7 $score \leftarrow 1 - b / 0.5\sqrt{(size^2 + size^2)}$
8 **return** $\langle T', score \rangle$

DISTANCE-AT-BEST-ANGLE(*points*, *T*, θ_a, θ_b, θ_Δ)
1 $x_1 \leftarrow \varphi\theta_a + (1 - \varphi)\theta_b$
2 $f_1 \leftarrow$ DISTANCE-AT-ANGLE(*points*, *T*, x_1)
3 $x_2 \leftarrow (1 - \varphi)\theta_a + \varphi\theta_b$
4 $f_2 \leftarrow$ DISTANCE-AT-ANGLE(*points*, *T*, x_2)
5 **while** $|\theta_b - \theta_a| > \theta_\Delta$ **do**
6 **if** $f_1 < f_2$ **then**
7 $\theta_b \leftarrow x_2$
8 $x_2 \leftarrow x_1$
9 $f_2 \leftarrow f_1$
10 $x_1 \leftarrow \varphi\theta_a + (1 - \varphi)\theta_b$
11 $f_1 \leftarrow$ DISTANCE-AT-ANGLE(*points*, *T*, x_1)
12 **else**
13 $\theta_a \leftarrow x_1$
14 $x_1 \leftarrow x_2$
15 $f_1 \leftarrow f_2$
16 $x_2 \leftarrow (1 - \varphi)\theta_a + \varphi\theta_b$
17 $f_2 \leftarrow$ DISTANCE-AT-ANGLE(*points*, *T*, x_2)
18 **return** MIN(f_1, f_2)

DISTANCE-AT-ANGLE(*points*, *T*, θ)
1 $newPoints \leftarrow$ ROTATE-BY(*points*, θ)
2 $d \leftarrow$ PATH-DISTANCE(*newPoints*, T_{points})
3 **return** *d*

PATH-DISTANCE(*A*, *B*)
1 $d \leftarrow 0$
2 **for** *i* **from** 0 **to** $|A|$ **step** 1 **do**
3 $d \leftarrow d +$ DISTANCE(A_i, B_i)
4 **return** $d / |A|$