$Q: A Super-Quick, Articulation-Invariant Stroke-Gesture Recognizer for Low-Resource Devices

Radu-Daniel Vatavu
MintViz Lab | MANSiD Center
University Stefan cel Mare of Suceava
Suceava 720229, Romania
vatavu@eed.usv.ro

Lisa Anthony
Department of CISE
University of Florida
Gainesville, FL 32611, USA
lanthony@cise.ufl.edu

Jacob O. Wobbrock
Information School | DUB Group
University of Washington
Seattle, WA 98195, USA
wobbrock@uw.edu

ABSTRACT

We introduce $Q, a *super-quick*, *articulation-invariant* point-cloud stroke-gesture recognizer for mobile, wearable, and embedded devices with low computing resources. $Q ran up to 142× faster than its predecessor $P in our benchmark evaluations on several mobile CPUs, and executed in less than 3% of $P's computations without any accuracy loss. In our most extreme evaluation demanding over 99% user-independent recognition accuracy, $P required 9.4 s to run a single classification, while $Q completed in just 191 ms (a 49× speed-up) on a Cortex-A7, one of the most widespread CPUs on the mobile market. $Q was even faster on a low-end 600-MHz processor, on which it executed in only 0.7% of $P's computations (a 142× speed-up), reducing classification time from two minutes to less than one second. $Q is the next major step for the "$-family" of gesture recognizers: articulation-invariant, extremely fast, accurate, and implementable on top of $P with just 30 extra lines of code.

ACM Classification Keywords

H.5.2. Information Interfaces and Presentation (e.g., HCI): User Interfaces – *Input devices and strategies.*

Author Keywords

Gesture recognition; stroke recognition; $1; $P; $Q; $-family; point-cloud recognizer; mobile devices; low-resource devices.

INTRODUCTION

In recent years, we have seen the tremendous proliferation of small electronic devices of all kinds [40,41,52,54]. Smartphones are the most prevalent of these, but other, smaller devices are now widespread, such as smartwatches, activity-monitoring wristbands, augmented reality glasses, Bluetooth earbuds, memory sticks, GPS trackers, nano-projectors, and others. All these smart devices embed CPUs that need to execute code quickly, especially for applications that process human input, such as gesture interfaces. As small as these devices are, new smaller devices emerge regularly, and continued miniaturization seems inevitable for some time [14]. Amazingly, ARM has shipped 86 billion chips to date (such as the CPU architectures referred to in Figure 1) that are used to power 95% of today's smartphones, 80% of digital cameras, and 35% of all electronic devices [4].

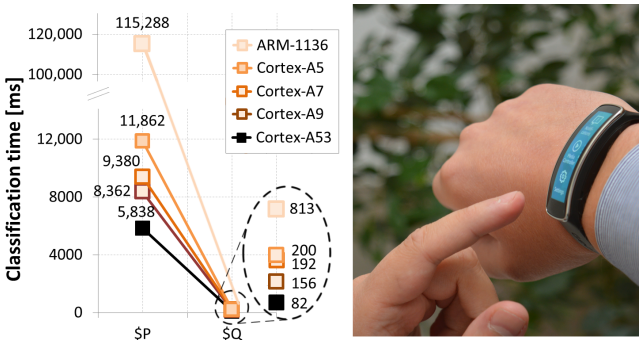

Figure 1. Classification times for $P and $Q (left) on five CPU architectures common for today's mobile, wearable, and embedded platforms for a set of 16 distinct stroke-gestures with *user-independent accuracy requirements of >99%*. Speed-ups of 46× on average and up to 142× over $P [50] make $Q very fast and suitable for next generation miniaturized low-resource devices, like the Gear Fit smartwatch shown at right.

Input on such small devices has largely been restricted to one or two buttons. Some devices, such as smartwatches, have touch-sensitive screens, but sophisticated input like text entry is still challenging due to a dearth of screen space. Research efforts have looked at novel input schemes such as tapping rhythms [52] to extend the range of input possibilities, but the opportunity remains to make interaction with these devices better. The challenge is exacerbated by the fact that the smallest devices are often also "low-resource devices," meaning they lack the processing power and memory that larger devices have. New input schemes are needed that are suitable for tiny low-resource devices, but more expressive than merely pressing a button.

One type of input that has seen significant uptake in recent years is gesture recognition. Although gestures come in many forms [30], stroke-gestures have gone mainstream, appearing in every touch screen-based product as swipes and flicks, atop virtual keyboards like Swype (www.swype.com) or ShapeWriter [25,56], and in the "insert special characters" feature in Google Docs, to name a few. A decade ago, the $1 gesture recognizer [53] provided simple, easy-to-build stroke-

MobileHCI '18, September 3–6, 2018, Barcelona, Spain.

ACM ISBN 978-1-4503-5898-9/18/09...$15.00.
https://doi.org/10.1145/3229434.3229465

gesture recognition that not only programmers could implement, but also interaction designers, rapid prototypers, and students from a variety of design and engineering disciplines. The $1 gesture recognizer inspired a series of follow-ons, such as Protractor [27], $N [2,3], $P [49,50], $3 [24], 1¢ [19], CircGR [10], Quick$ [38], Penny Pincher [43,44], and Jackknife [45], among others.

Arguably the most flexible and powerful of the "$-family" of gesture recognizers is $P [50] due to its unique feature of representing gestures as point clouds. This representation makes $P an *articulation-invariant* recognizer, *i.e.*, it classifies gestures correctly no matter how users articulate them in terms of the number of strokes, stroke types and directions, or stroke orderings.To achieve this performance, $P relies on solving a point-to-point correspondence problem between two point clouds, which is an example of the classic "assignment problem" [11,15], adapted to and reformulated in the context of stroke gesture representation, matching, and classification. However, $P is not well suited to low-resource devices, not having been implemented for that purpose.

In order to bring powerful stroke-gesture recognition to low-resource devices, we introduce $Q, a major advance over the $P recognizer. $Q is a super-Quick highly-optimized recognizer that *uses a mere 3% of the computations of $P without losing any recognition accuracy*, which makes it computationally affordable on low-resource devices. We implemented $Q on five CPU architectures representative of today's low to high-end mobile, wearable, and embedded platforms (see Figure 1) and found that $Q achieves speed-ups of 46× on average and up to 142×, compared to $P. For instance, $Q evaluated over 1,000 templates in just 82 ms on the Cortex-A53 mobile CPU, compared to $P's 5.8 seconds.

With this work we contribute both theoretical and practical advances to the state-of-the-art in stroke-gesture recognition:

1. We perform the first thorough evaluation of $P's classification time performance, showing that $P [50], the most flexible gesture recognizer today, is simply unsuited for efficient operation on low-resource devices. Moreover, we show that standard code optimization techniques are not enough to make $P suited for such devices, *e.g.*, the optimized $P took 2.7 s to compute on our fastest CPU. Waiting nearly three seconds for each gesture to be recognized is clearly infeasible for interactive applications.
2. We designed $Q informed by key observations about efficient point-cloud matching that we transcoded into clear, theoretically-sound algorithmic formalisms implementable on any platform. In this process, we introduced the first theoretical lower bound for matching point clouds.
3. We conducted a thorough evaluation of $Q by considering user-dependent and user-independent scenarios, various configurations of the template set, and multiple CPU architectures, which showed massive speed improvements of $Q over $P: $Q is up to 142× faster and requires only 3% of $P's computations without any accuracy loss.
4. We provide clear pseudocode for $Q to enable implementation on any mobile, wearable, or low-resource gadget.

$Q is the next major evolution for the $-family: fast, accurate, and enabling gesture recognition on a host of current and future devices with little or limited computing power.

RELATED WORK

We discuss in this section prior work on gesture recognition. For the purposes of this work, we define a "gesture" to be any path traced onto a touch-sensitive surface by a user with a finger or stylus [53,55]. We thereby differentiate this type of gesture from the body of work exploring 3-D, spatial, mid-air, whole-body, and whole-hand gestures.

Stroke-Gesture Recognition

Gesture classification algorithms for touch input have ranged from designs specifically focused on recognizing individual letters and shapes [2,3,27,28,39,50,53] to complex fully sketched diagrams [16,17,20,21]. Stroke-gestures have been explored not only for smartphones and smartwatches [27,54], but also for new wearable prototypes enabling input through clothing [40,41] or even directly on the skin [18].

The area of gesture recognition most relevant to our work is the so-called "$-family" of stroke-gesture recognition algorithms. In the nearly ten years since Wobbrock *et al.* [53] introduced their $1 unistroke gesture recognizer, there has been a growing trend in the use of template-based matching algorithms for recognition of touch gestures, such as extensions of $1 to multistroke recognition in $N [2], closed-form speed-ups like Protractor and $N-Protractor [3,27], and articulation-invariant approaches, such as $P [50]. The $-family has also inspired other researchers to use similar naming schemes to denote a likeminded philosophy of providing formerly hard-to-implement capabilities in a more prototype-friendly form; *e.g.*, the Quick$ [38] and 1¢ [19] recognizers, the 1€ filter [12], the 1F filter [47], Penny Pincher [43,44], and Jackknife [45]. Efforts to optimize gesture recognition have considered identifying the lowest resampling rate and bit depth that still allow for accurate recognition [46,48], or selecting the best fit templates for training the recognizer, such as Pittman *et al.*'s [33] metaheuristics-inspired approach to prototype selection. This line of work on $-family recognizers has formed a tangent from state-of-the-art machine learning based approaches, such as Hidden Markov Models [42], deep neural networks [13], or feature-based statistical classifiers [20]. The $-family algorithms have remained relevant due to their simplicity of implementation, making them easy to port to new platforms, and their ability to perform very accurately with few templates.

Gesture Recognition on Low-Resource Devices

Protractor [27] was the first template-matcher proposed specifically for low-resource device demands. The algorithm extended $1 [53] to use a closed-form matching step, whereas $1 used a more computationally expensive Golden Section Search [34] to iteratively find the best angular alignment between two stroke-gestures during their comparison. Li evaluated Protractor's performance on a low-resource device (a 528-MHz ARM-11 CPU) and reported a nearly 60-fold performance improvement over $1 [27].

Other prior work introduced methods to improve the computational complexity of the $1 recognizer: the Quick$ recognizer [38], 1¢ [19], the 1F filter [47], or Penny Pincher [43,44].

Quick$ [38] sacrificed some implementation simplicity by employing offline hierarchical clustering of the template gestures and used a branch and bound search strategy to match candidate gestures to templates. The 1¢ recognizer [19] used a rotation-invariant 1-D representation of stroke-gestures that, just like Protractor, removed the need for the search strategy used by $1. The 1F filter [47] focused on pruning the template set using a gesture feature to approximate the best match, and only required a local search rather than a full search of all possible templates. Penny Pincher [43,44] was characterized by high accuracy and rapid execution, especially in the case of time-limited recognition, although it was less flexible than $P, *i.e.*, *"it is not intrinsically a multistroke recognizer"* [43] (p. 201). A more recent recognizer, Jackknife [45], proved very accurate with few templates but, because of its foundation on Dynamic Time Warping [23,32], it addressed mostly unistrokes and could not recognize multi-stroke gestures with the articulation-invariant flexibility of $P.

Unlike other approaches, $P can handle any combination of templates and candidates: $P can recognize gestures accurately no matter how users wish to articulate them [50]. Thus, in this work, from all possible gesture classification approaches, we focus on $P, the current most flexible gesture classifier.

The $P Point-Cloud Recognizer

The recognizer with the highest flexibility for the widest range of stroke-gestures so far is $P, introduced by Vatavu *et al.* [50]. In this section, we overview the principles of point-cloud gesture recognition, and we reproduce the $P pseudocode from Vatavu *et al.* [50] (p. 280) that serves as the starting point for our next-generation $-family recognizer, $Q.

$P represents stroke-gestures as clouds of points and discards any timestamp information associated with the individual points. This specific representation enables $P to deliver accurate classifications no matter how users choose to produce gestures in terms of different numbers of strokes, stroke orderings, and stroke directions. To evaluate the dissimilarity between two gestures, a candidate C and a template T, $P computes an approximate optimal alignment between their point clouds using the following formula [50] (p. 279):

$$\$P(C,T) = \sum_{i=0}^{n-1} \left(1 - \frac{i}{n}\right) \cdot \min_{j(\star)} \left\| C_i - T_j \right\| \qquad (1)$$

where point C_i from the candidate has been matched with point T_j from the template under the constraint that point T_j is closest to C_i and has not been matched before with any other point from C ($\star$ in Eq. 1). The Euclidean distances between points ($\left\|C_i - T_j\right\|$) are weighted by a factor normalized in $[0..1]$ that shows the confidence of a particular matching: weights decrease linearly from 1 to $1/n$, where n is the number of points in each cloud. (Note that the number of points in each cloud are the same because of a point-resampling procedure [25,46,53].)

To produce a classification result, the $P recognizer computes dissimilarity scores between the candidate gesture C and each template T from the template set and assigns C to the class of its closest template, as follows:

$$C \in \text{ class of } T^\star \text{ where } T^\star = \text{argmin}_T \left\{\$P(C,T)\right\} \qquad (2)$$

Figure 2 reminds the reader of the main parts of $P for easy reference when following our discussions about $Q.

$P-RECOGNIZER (POINTS *points*, TEMPLATES *templates*)

```
1: n ← 32
2: NORMALIZE(points, n)
3: score ← ∞
4: for each template in templates do
5:     NORMALIZE(template, n)
6:     d ← GREEDY-CLOUD-MATCH(points, template, n)
7:     if d < score then
8:         score ← d
9:         result ← template
10: return ⟨result, score⟩
```

GREEDY-CLOUD-MATCH (POINTS *points*, POINTS *template*, int *n*)

```
1: ε ← .50
2: step ← ⌊n^(1−ε)⌋
3: min ← ∞
4: for i = 0 to n − 1 step step do
5:     d1 ← CLOUD-DISTANCE(points, template, n, i)
6:     d2 ← CLOUD-DISTANCE(template, points, n, i)
7:     min ← MIN(min, d1, d2)
8: return min
```

CLOUD-DISTANCE (POINTS *points*, POINTS *tmpl*, int *n*, int *start*)

```
1: matched ← new bool[n]
2: sum ← 0
3: i ← start // start matching with points_i
4: do
5:     min ← ∞
6:     for each j such that not matched[j] do
7:         d ← EUCLIDEAN-DISTANCE(points_i, tmpl_j)
8:         if d < min then
9:             min ← d
10:            index ← j
11:    matched[index] ← true
12:    weight ← 1 − ((i − start + n) MOD n)/n
13:    sum ← sum + weight · min
14:    i ← (i + 1) MOD n
15: until i == start // all points are processed
16: return sum
```

EUCLIDEAN-DISTANCE (POINT *a*, POINT *b*)

```
1: return ((a_x − b_x)^2 + (a_y − b_y)^2)^0.5
```

Figure 2. Pseudocode of the main parts of the $P recognizer (classification algorithm and cloud distance) from Vatavu *et al.* [50] (p. 280), which we use in this work as the starting point for $Q.

Summary

Out of all stroke-gesture recognition approaches available to practitioners, we focus on $P due to its high accuracy and flexibility: $P delivers $> 99\%$ recognition rates with just a few templates [50], adapts easily to various contexts of use and user groups [1,31,49] and, most importantly, enables users to articulate gestures as they wish and still be recognized, a feature difficult to match by other recognizers. Unfortunately, $P is quite costly for low-resource devices, as we are about to show in the next section, because of its quadratic complexity [50] (p. 278). Consequently, there is still need for improvement in the state-of-the-art for gesture recognition algorithms of the $-family type. In this work, we focus on making point cloud gesture recognition *fast* and *robust* on the miniaturized, low-resource devices of today and tomorrow.

EVALUATION OF $P ON LOW-RESOURCE DEVICES

In this section, we conduct the first thorough evaluation of $P's time performance on several mobile CPU architectures and we show that even a carefully code-optimized version of $P is not enough for efficient operation on such devices. First, we introduce our methodology for evaluating classification performance, which we use throughout this paper.

Evaluation Methodology

We evaluate classification time, user-dependent, and user-independent recognition rates using the following procedures.

User-Dependent Recognition Rates

We compute user-dependent recognition rates as a function of the number of samples per gesture type (T) available in the recognizer's template set. The procedure to compute user-dependent rates is as follows: T samples for each gesture type are selected at random for a given user (T varies from 1 to 8) and one additional sample, different from the first T, is selected as the candidate gesture to be submitted for classification. This procedure is repeated 100 times for each T=1 to 8 and each user, and results are averaged into the user-dependent recognition rate, expressed as a percentage. We adopted this procedure from the $-family gesture recognizers [2,3,50,53].

User-Independent Recognition Rates

We compute user-independent recognition rates as a function of the number of participants (P) from which templates are collected and the number of samples per gesture type (T) from each participant. P participants are selected at random for gathering templates (P varies from 1 to 8), and one additional participant, different from the first P, is selected for testing. T samples are selected at random for each gesture type from each of the P training participants (T varies from 1 to 8). One sample for each gesture type is selected at random from the testing participant and submitted for classification. This procedure is repeated 100 times for each P and 100 times for each T, and results are averaged into the user-independent recognition rate, expressed as a percentage. This same procedure was employed to evaluate the user-independent accuracy rate of the $P gesture recognizer [50].

Experiment Design

We framed our evaluation process as a controlled experiment with four independent variables:

1. RECOGNIZER, nominal variable, representing various gesture recognition approaches. In this work, we compare $P with a code-optimized version of $P (described in the next section), and $P with $Q, respectively.
2. T, the number of templates per gesture type. In this work, we evaluate four conditions for T starting from 1 template up to 8 templates per gesture type following a geometric progression with common ratio 2, *i.e.*, with each new condition, T doubles from 1 to 2, 4, and 8 templates, respectively.
3. P, the number of participants from which gesture samples are gathered as templates. Just as for T, we evaluate four conditions for P as well: 1, 2, 4, and 8 training participants.
4. CPU, the architecture on which we evaluate the classification speed of $P and $Q. CPU is an ordinal variable with four conditions: *high-end* (Cortex-A53), *midrange* (Cortex-A9), *entry-level* (Cortex-A7), and *low-end* (Cortex-A5).

P and T determine the size of the template set and, consequently, affect the classification speed of our recognizers for a given CPU. Our *worst-case scenario* is $P \times T = 8 \times 8 = 64$ templates per gesture type that corresponds to the slowest speed, but also to the highest (user-independent) recognition accuracy (>99%), as we are about to show.

We evaluate the performance of gesture recognizers with the following dependent variables:

1. CLASSIFICATION-TIME represents the time, in milliseconds, necessary for a RECOGNIZER to compute the classification result. We evaluate classification time as a function of the number of samples in the template set by considering all the P×T combinations (1, 2, 4, 8, 16, 32, and 64), which correspond to sets of 16 to 1024 templates.
2. TIME-SAVINGS, the percentage of CPU time saved by a fast recognizer, *e.g.*, $Q, compared to the baseline $P:

$$\left(1 - \frac{\text{CLASSIFICATION-TIME}(\$Q)}{\text{CLASSIFICATION-TIME}(\$P)}\right) \cdot 100\% \quad (3)$$

3. RECOGNITION-RATE represents the recognition accuracy (user-dependent or user-independent) for a given training configuration, reported as a percentage.

Benchmark Devices and CPU Architectures

To reach a thorough understanding of $P and $Q's classification time performance in practice, we measured the dependent variables on several CPU architectures designed for a wide range of mobile devices, from low-power wearables up to high-end, high-performance smartphones:

1. Cortex-A53 (high-end) is a CPU architecture on 64 bits (ARMv8-A) applicable for devices that require high performance in power-constrained environments, such as high-end smartphones, low-power servers, and Smart TVs [7]. We evaluated the Cortex-A53 architecture with the Samsung Galaxy A3 smartphone (model number SM-A300FU, Android v6.0.1) that includes this CPU running at 1.2 GHz (Quad-Core) and 1.5 GB RAM.
2. Cortex-A9 (midrange) is a CPU architecture on 32 bits (ARMv7-A) applicable for low-power, cost-sensitive consumer devices [9]. We evaluated the Cortex-A9 with the ASUS Transformer Pad tablet (model number TF300T, Android v4.0) that includes this CPU running at 1.2 GHz (Quad-Core) and 1 GB RAM.
3. Cortex-A7 (entry-level) is a CPU architecture on 32 bits (ARMv7-A) designed for a wide range of devices demanding a balance between power and performance, particularly rich UI based wearables, with over a billion units shipped in production [8]. We evaluated the Cortex-A7 architecture with the AllView VIVA C7 tablet (Android v4.4.2) that includes this CPU running at 1.0 GHz (Dual-Core) and 512 MB RAM.
4. Cortex-A5 (low-end) is a CPU architecture on 32 bits (ARMv7) designed for devices that require an extremely low-power, low-area profile, such as feature phones and

wearables [6]. We evaluated the Cortex-A5 architecture with the Samsung Galaxy Young smartphone (model number GT-S6310N, Android v4.1.2) that includes this CPU running at 1.0 GHz (Single-Core) and 768 MB RAM.

These CPU architectures have already been deployed in many types of mobile devices, home and consumer devices, and embedded designs. In fact, the ARM-based CPU market share covers 95% of the smartphone market and as of this writing, over 86 billion chips with ARM cores have been produced [4]. Also, these architectures are representative of trends in CPU technology for mobile, wearable, and embedded platforms, such as shifting from single-core to multi-core computing, from 32-bit to 64-bit apps, and improving instruction sets. Android Studio was used to compile a Java implementation of $P for the recommended Android API level of each device.

Gesture Dataset

We evaluate classification time performance on the MMG gesture dataset of Anthony and Wobbrock [3]. The dataset comprises 16 distinct gesture types performed by 20 participants with 10 executions per gesture type. Overall, the dataset contains 9596 gestures performed using the finger and the stylus on a tablet device. The dataset is available from the "$N Multistroke Recognizer" web page [3] and is the same dataset used in the original $P paper [50]. All gestures were preprocessed before recognition by following the recommendations from the $-family literature [2,3,50,53]: gestures were uniformly resampled into n=64 points, scaled down to the unit box with shape preservation, and translated to the origin.

$P's Classification Time on Mobile CPUs

Figure 5 illustrates the user-dependent and user-independent recognition rates delivered by the $P recognizer as a function of the number of training templates (T) and participants (P) in the template set. $P delivers 99% user-dependent accuracy with T=4 templates per gesture type (Figure 5a), reaches 99.1% user-independent accuracy starting from P×T = 32 templates, and goes up to 99.5% for P×T = 64 templates (Figure 5b). Figure 6 shows the influence of the number of templates per gesture type (P×T) on $P's classification time for each CPU architecture. To reach >99% user-independent accuracy, $P needs about 6 s on the fastest CPU (Cortex-A53), more than 8 s on Cortex-A9 and Cortex-A7, and about 12 s on the low-end Cortex-A5 CPU. These results show that $P is not suited for interactive applications on mobile platforms. The next section shows how $P can be made faster using standard textbook code optimization techniques but, even still, $P remains inefficient for operation on low-resource devices.

A Code-Optimized Version of $P

We looked at several ways to improve the classification time of $P on our mobile CPUs by resorting to code optimization techniques accessible to any programmer [22,29]. As it will be seen, however, despite improvements in $P's execution speed, known optimization techniques do not give $P enough speed-up for interactive use on low-resource devices.

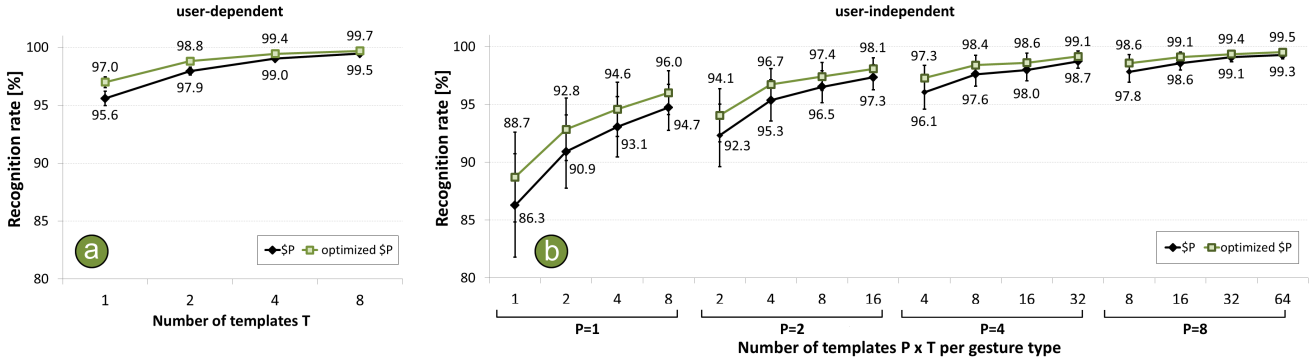

Figure 5. Recognition rates for user-dependent (a) and user-independent (b) template sets for optimized $P compared to original $P. Notes: Higher is better. Error bars represent 95% confidence intervals.

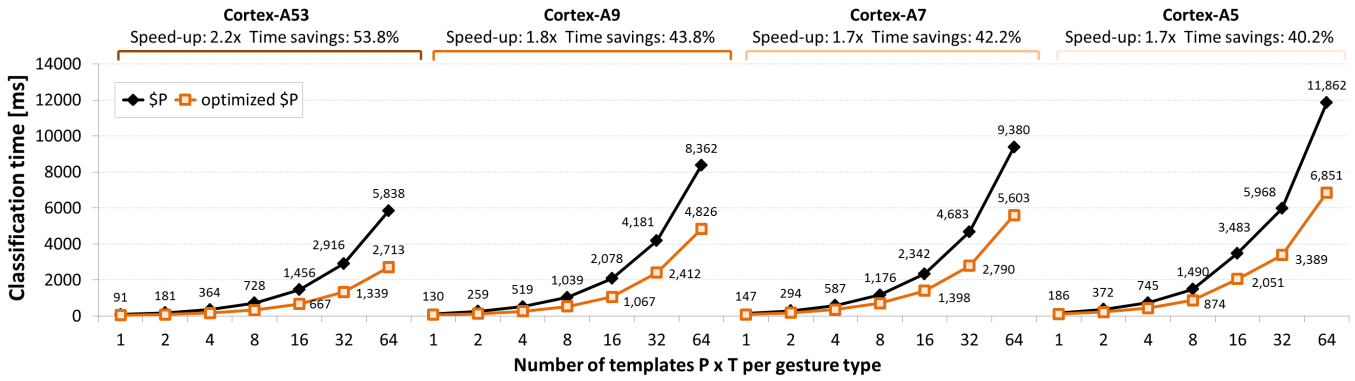

Figure 6. Classification times for optimized $P compared to original $P. Notes: Lower is better. Error bars are too small to be visible.

```
CLOUD-DISTANCE (POINTS points, POINTS template, int n, int start)
 1: unmatched ← {0,1,2,...,n−1}   // indices of unmatched points from template
 2: i ← start      // start the matching from this index in the points cloud
 3: weight ← n     // weights decrease from n to 1
 4: sum ← 0        // computes the cloud distance between points and template
 5: do
 6:     min ← ∞
 7:     for each j in unmatched do
 8:         d ← SQR-EUCLIDEAN-DISTANCE(points[i], template[j])
 9:         if d < min then
10:             min ← d
11:             index ← j
12:     REMOVE(unmatched, index)   // implementable in O(1)
13:     sum ← sum + weight · min
14:     weight ← weight − 1   // weights decrease from n to 1
15:     i ← (i+1) MOD n     // advance to the next point in points
16: until i == start
17: return sum

SQR-EUCLIDEAN-DISTANCE (POINT a, POINT b)
1: return (a_x − b_x)^2 + (a_y − b_y)^2
```

Figure 7. Code-level optimizations in the $P's CLOUD-DISTANCE function reduce CPU time by 42.1% on average compared to original $P. Highlighted text shows changes with respect to the $P recognizer.

First, we noted that unnecessary computations are performed in the original CLOUD-DISTANCE function (Figure 2) when searching for points from the second cloud that have not been matched yet to points from the first cloud. This part has been implemented by $P with the *matched* array, which stores at index j a boolean value indicating whether point j from the second cloud has been matched or not (see line 6 in the original CLOUD-DISTANCE function, Figure 2). With this implementation, j has to loop through all the points of the second cloud when searching for an unmatched point, which represents a very naive search strategy. A faster approach would have j only enumerate the points that have not been matched yet. Therefore, we refactored the *matched* array of boolean values into a list of integers, *unmatched*, which we initialize with values from 0 to $n-1$. As points from the second cloud are matched, their indices are removed from the list and are not considered by the next iterations. This refactoring saves $n(n+1)/2$ conditional tests from being executed.[1] Modified lines are highlighted in light gray in the new $P pseudocode illustrated in Figure 7.

Second, the square root of the Euclidean distance represents computational overhead, because only the *relative* order of distances is needed and not their *absolute* magnitudes when searching for the closest point j from the second cloud to match point i from the first cloud (lines 7 to 10 in the original CLOUD-DISTANCE function, Figure 2). Also, dividing *all* the weights by n (Figure 2, line 12) represents unnecessary computational overhead as well. These two observations can help save precious floating point operations.[2] Lines affected by these two changes are highlighted in yellow in Figure 7.

[1] If-statements affect the efficiency of built-in branch prediction strategies of modern CPUs [22].

[2] For low-resource CPUs, the Floating Point Unit (FPU) comes as an optional feature [5,6] and, thus, may not always be available to speed up floating-point arithmetic.

Time savings effected by these code optimizations were considerable: 45.0% on average and 53.8% for the fastest CPU (Cortex-A53); see Figure 6 on the previous page. The optimized $P produced a classification result in 2713 ms versus 5838 ms for the most demanding scenario (P×T=8×8=64, corresponding to a user-independent >99% requirement of classification accuracy) on the Cortex-A53; it took 4826 ms versus 8362 ms on Cortex-A9, 5603 ms versus 9380 ms on Cortex-A7, and 6851 ms versus 11,862 ms on the Cortex-A5. Figure 5 shows recognition rates for optimized $P versus original $P. Removing the square root of the Euclidean distance also caused a small increase in recognition accuracy, from 98.0% to 98.7% for user-dependent training and from 95.7% to 96.7% for user-independent training. The slight increase in accuracy is due to the sum of squared distances placing progressively greater weight on points farther apart.

Nevertheless, although speed-ups are considerable with the optimized $P, they are still far from real-time responsive. Thus, we pursued a new recognizer for low-resource devices, $Q.

THE $Q GESTURE RECOGNIZER

In this section, we present $Q, our next-generation, fast and accurate $-family gesture recognizer. First, we outline the design guidelines that we adopted for $Q. Second, we introduce new algorithmic techniques for articulation-invariant point-cloud gesture matching that reduce >97% of the computations needed by $P. Overall, our new algorithmic improvements make $Q run 46.4× faster on average and up to 142× faster than original $P on common mobile CPU architectures.

$Q Design Philosophy

We adopted the following guidelines for our algorithmic design and analysis process to improve gesture classification time performance on low-resource devices:

1. ***Simplicity.*** The algorithmic design of $Q must be in line with the general *raison d'être* of the $-family [2,3,19,24,27, 38,43,44,50,53], which is to provide practitioners with recognizers that are *"easy to understand, build, inspect, debug, and extend"* with just a few lines of code [53]. Because of the diversity of professions involved in Human-Computer Interaction (developers, user interface designers, interaction designers, usability specialists, etc.), the new $Q recognizer must be implementable on top of $P without much additional effort even by novice programmers.
2. ***Accuracy and flexibility.*** $Q must not compromise the high recognition accuracy and flexible articulation-invariant matching of its predecessor point-cloud recognizer, $P. It is important not to trade off accuracy for speed gains, as previous research has shown that even a few percent increase in recognition errors can affect user satisfaction [26]. As it turns out, $Q is actually slightly *more* accurate than $P.
3. ***Efficiency on any platform.*** $Q must be fast, anywhere. Note that speed-ups (usually of about 20%) are easily obtained with faster hardware. For example, changing the target platform and running our implementation of $P on a Quad-Core 1.2 GHz CPU (Cortex-A9) instead of a Single-Core 1.0 GHz CPU (Cortex-A5) improved classification

time by 24.4%. In this work, we target algorithmic improvements that are *hardware-independent* and, therefore, improve classification time on any platform.

$Q Design and Implementation

$P implements Nearest-Neighbor classification [51] (pp. 93-105) by comparing the candidate gesture with each template and, with each comparison, updating an internal reference to the template with the lowest dissimilarity from the candidate (see variable *min*, lines 8-10 in the original CLOUD-DISTANCE function, Figure 2). However, because the dissimilarity score is computed in the form of a sum (Eq. 1), the value of this sum can become, at some point during its computation, already larger than the minimum score found until that point. From that moment on, completing the matching process between the candidate and the template represents unnecessary computation, because we already know that the minimum score cannot be improved by the current sum. Therefore, it makes sense to stop the computation of Eq. 1 at that point. Rakthanmanon *et al.* [35,36] employed such a strategy to speed up the execution time of Dynamic Time Warping (DTW) for querying very large time series. $Q similarly implements early abandonment during the point-cloud matching process. Preliminary evaluations (a thorough evaluation of $Q is presented next in this section) showed a reduction in CPU workload by 85.7% on average compared to original $P. The extra time savings compared to the optimized version of $P was 43.6%.

We can further save valuable computation time by calling the CLOUD-DISTANCE function only when needed. For example, if we could know in advance that the cloud distance between the candidate and a template cannot be less than a lower bound (LB) and that the minimum (best so far) distance is already smaller than LB, then calling CLOUD-DISTANCE would simply be a waste of computing resources. Lower bounding has been employed in the data mining community [35,36] for speeding up DTW [23,37] but, so far, has not been utilized for matching point clouds. In the following, we define a lower bound for $Q as any function LB that satisfies:

$$\mathrm{LB}(C,T) \leq \$\mathrm{Q}(C,T) \text{ for all gestures } C \text{ and } T \qquad (4)$$

where $\$\mathrm{Q}(C,T)$ evaluates similarly to Eq. 1, except that weights are no longer divided by the size of the point cloud, n, and Euclidean distances are now squared Euclidean distances:

$$\$Q(C,T) = \sum_k w_k \cdot \min_{j(\star)} \left\|C_k - T_j\right\|^2 \qquad (5)$$

where w_k are weights that decrease from $n-1$ to 1.

We can now introduce a lower bound for the cloud distance by removing the constraint of having strict 1-to-1 point matchings and allowing any point from C to be matched to any point from T, even if they were matched before. The dissimilarity score will be smaller than the exact cloud distance, because there are more options to choose pairs of points from C and T. Assuming starting point i, the lower bound is:

$$\mathrm{LB}_i(C,T) = \sum_k w_k \cdot \min_{j=0,n-1} \left\|C_k - T_j\right\|^2 \qquad (6)$$

CLOUD-MATCH (POINTS *points*, POINTS *template*, int *n*, int *min*)

```
1: ε ← 0.5
2: step ← ⌊n^(1−ε)⌋
3: //compute lower bounds for both matching directions between points and template
4: LB1 ← COMPUTE-LOWER-BOUND(points, template, step, template.LUT)
5: LB2 ← COMPUTE-LOWER-BOUND(template, points, step, points.LUT)
6: for i ← 0 to n − 1 step step do
7:     if LB1[i/step] < min then
8:         min ← MIN(min, CLOUD-DISTANCE(points, template, n, i, min))
9:     if LB2[i/step] < min then
10:        min ← MIN(min, CLOUD-DISTANCE(template, points, n, i, min))
11: return min
```

CLOUD-DISTANCE (POINTS *points*, POINTS *template*, int *n*, int *start*, float *minSoFar*)

```
1: unmatched ← {0,1,2,...,n−1}  // indexes of unmatched points from template
2: i ← start      // start the matching from this index in the points cloud
3: weight ← n     // weights decrease from n to 1
4: sum ← 0        // computes the cloud distance between points and template
5: do
6:     min ← ∞
7:     for each j in unmatched do
8:         d ← SQR-EUCLIDEAN-DISTANCE(points[i], template[j])
9:         if d < min then
10:            min ← d
11:            index ← j
12:    REMOVE(unmatched, index)  // implementable in O(1)
13:    sum ← sum + weight · min
14:    if sum ≥ minSoFar then
15:        return sum            // early abandoning of computations
16:    weight ← weight − 1       // weights decrease from n to 1
17:    i ← (i + 1) MOD n         // advance to the next point in points
18: until i == start
19: return sum
```

Figure 8. $Q introduces key algorithmic improvements to point-cloud matching. Highlighted text shows changes with respect to $P's pseudocode. LUT stands for "look-up table" and is described in the paper.

Note that multiple lower bounds need to be computed: one for each starting point i (because of the coefficients w_k used in Eq. 6 that apply, in order, starting from point i) and for each matching direction (because of the asymmetric nature of matching, if T_j is the closest point for C_k, it doesn't necessarily follow that C_k is the closest point from T_j); see Figure 8. Also, lower bounds need to evaluate fast to be effective. (A naïve implementation of Eq. 6 requires $O(n^2)$ time to compute.) Therefore, we introduce a new look-up point-matching technique that implements searching for the closest point in $O(1)$ time and the lower bound in $O(n)$.

Let $m \times m$ be a 2-D grid of equally-spaced points superimposed on top of the point cloud of a gesture C (Figure 9). For each point (x,y) in the grid, we compute its closest point from C, say C_i, and store index i in the LUT. When we need to compute the closest point from a template to point C_i, we just approximate the x and y coordinates of the template point to coordinates in the grid, for which the look-up table returns the closest point from C, reducing an $O(n)$ search to a mere $O(1)$ arithmetic operation. A lower bound (Eq. 6) can thus be computed in $O(n)$ time and all lower bounds in $O(n^{1.5})$. Because $Q matches gestures in both directions, both the LUTs of the candidate and template gestures must be available. However, while the LUT of the candidate needs to be computed online, the LUT of each template gesture can be computed offline and stored together with the gesture points in the template set. Moreover, lower bounds for consecutive starting points can be derived directly from previous computations, as indicated by

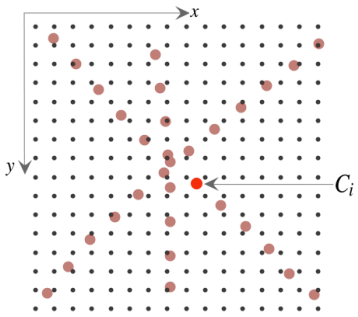

Figure 9. A grid of points superimposed on the point cloud C of an "asterisk" gesture. The look-up table of cloud C will contain at location x, y the index of the point C_i from C closest to (x, y), *i.e.* LUT[x,y]= i.

the following recurrence relation:

$$LB_{[i/step]} = LB_{[0]} + i \cdot SAT_{[n-1]} - n \cdot SAT_{[i-1]} \quad (7)$$

where SAT implements a summed area table for fast access to cumulative summations, with $SAT_{[j]}$ $(j = 0..n-1)$ being the sum of Euclidean distances between points 0 to j of the first cloud to their closest points in the second cloud:

$$SAT_{[j]} = \sum_{t=0}^{j} \min_{k=0,n-1} \|C_t - T_k\| \quad (8)$$

and i goes from 0 to $n-1$ in steps of $step$; see the pseudocode from Figure 8. Equations 7 and 8 help compute *all* the lower bounds in just $O(n)$ time instead of $O(n^{1.5})$; see the Appendix.

Gesture points need to be rescaled to match the size of the grid, but this operation can be done during preprocessing (see the NORMALIZE and SCALE functions in the full pseudocode of $Q at the end of this paper). Our experimentation showed that grids of 64×64 points deliver a good compromise between extra memory demands (4KB per gesture) and recognition accuracy.[3] The time complexity to compute look-up tables is $O(n \cdot m^2)$, but LUTs can be precomputed and loaded together with gesture templates from the template set, thus removing the dependency on m^2 and resulting in $O(n)$ complexity.

Figure 10 shows user-dependent and user-independent recognition rates for $Q versus original $P. Recognition rates increased from 98.0% to 98.7% on average for user-dependent training (gain +0.7%) and from 95.7% to 96.7% for user-independent training (gain +1.0%), respectively, due to the square root effect explained above in the paper. However, time savings are now massive: $Q completed in 82 ms versus $P's 5838 ms for our high recognition performance test case on the Cortex-A53 architecture (71.5× speed-up); it took 156 ms versus 8362 ms on Cortex-A9 (53.6× speed-up), 192 ms versus 9380 ms on Cortex-A7 (48.9× speed-up), and computed in just 200 ms versus $P's required time of 11,862 ms on Cortex-A5 (59.3× speed-up); see Figure 11. Overall, $Q executed in only 3% of $P's computations on all our CPU architectures.

[3] This result is in accordance with Vatavu [48], who showed that 4-5 bits per channels x and y are sufficient for recognizers to discriminate gestures accurately. So, $m = 64$ means 6 bits for x and y.

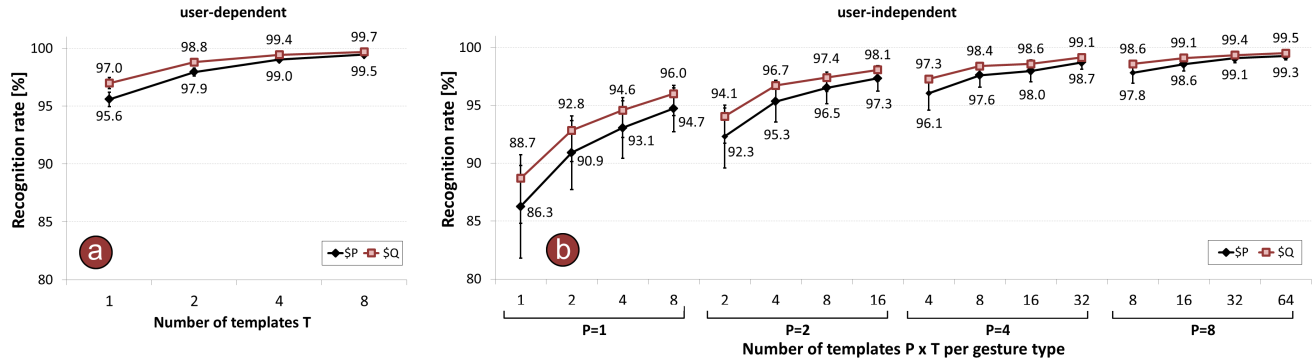

Figure 10. Recognition rates for user-dependent (a) and user-independent (b) template sets for $Q compared to original $P. Notes: Higher is better. Error bars represent 95% confidence intervals.

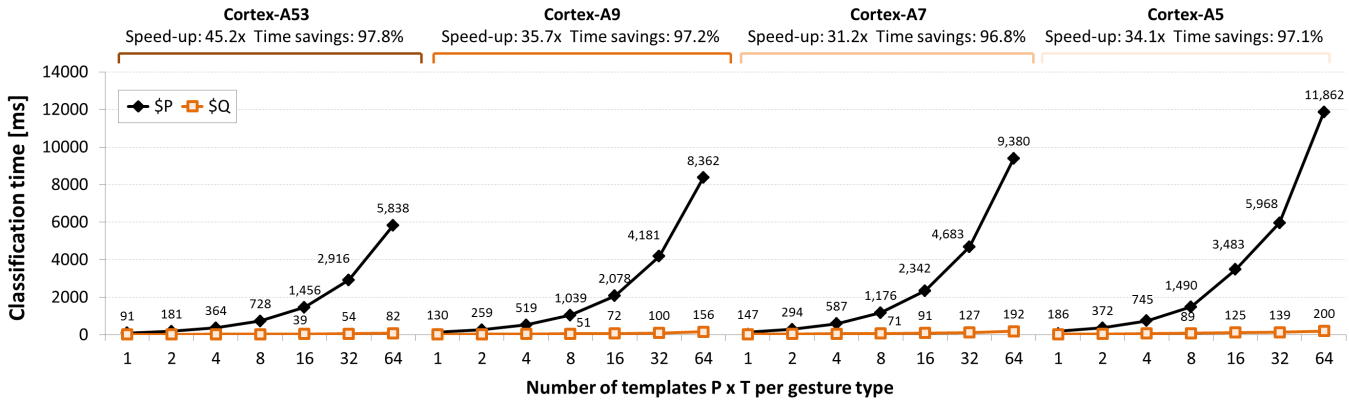

Figure 11. Classification times for $Q compared to original $P. Notes: Lower is better. Error bars are too small to be visible.

THE ALGORITHMIC COMPLEXITY OF $Q

The $P recognizer is known to have a polynomial time complexity of $O(\tau \cdot n^{2.5})$, where n is the size of the point cloud and τ the size of the template set. However, the time complexity of $Q is difficult, if not impossible, to calculate exactly, because its execution is not affected only by the size of the input data (n and τ, respectively), but also by the gestures themselves, as the efficiency of the lower bounding and early abandoning filters at any given step of the algorithm depend on what kind of data they have seen up to that point. Nevertheless, we want to get an understanding of $Q's theoretical complexity, at least in approximate form. To this end, we computed classification times for $Q for gestures sampled in n points going from 10 to 90, in steps of 10, and for template sets of size τ increasing from 160 to 1440, corresponding to a number of templates per gesture type from 10 to 90, in steps of 10.[4]

In the following, we want to find a reasonable functional approximation $f(n,\tau)$ for the classification time of $Q. We know that $f(n,\tau)$ is at least $a_0 \cdot n \cdot \tau + a_1 \cdot \tau + a_2 \cdot n^{0.5} \cdot \tau$, because $Q needs to compute the lower bounding function for each template in $O(n \cdot \tau)$, calculate the minimum matching score across all templates in $O(\tau)$, while lower bounds are compared with the minimum score for all templates and all $n^{0.5}$ starting points in $O(n^{0.5} \cdot \tau)$ time; a_0, a_1, and a_2 are constants hidden in the $O(\cdot)$ notations that reflect how fast these steps run on various CPUs. Because our experimental results showed that lower bounding and early abandoning reduce about 97% of the computations of $P, we expect that the number of runs of the point cloud matching procedure with the complexity $O(n^{2.5})$ will be limited to only few templates from the template set. We also expect that the rest of the templates, who bypass the lower bounding step, will only require a few point matchings until the early abandoning comes into effect, making point-cloud matchings execute in $O(n)$. With these considerations, our functional form estimate for the runtime complexity of $Q is:

$$f(n,\tau) = a_0 \cdot n \cdot \tau + a_1 \cdot \tau + a_2 \cdot n^{0.5} \cdot \tau + a_3 \cdot n^{2.5} + a_4$$

We found that this model showed a very good fit (R^2=.996, $F_{(4,76)}$=4736.118, p<.001) to our classification time data, producing a_0=0.0014, a_1=0.0176, a_2=−0.0072, a_3=0.0009, and a_4=−0.0065 for the Cortex-A53 CPU.[5] Based on these findings, we can estimate a time complexity of $O(\tau \cdot n + n^{2.5})$ for $Q, which reduces to a *linear growth* in n and τ, $O(\tau \cdot n)$, for large training sets ($\tau \gg n^{1.5}$). The speed gain of $Q over $P gets larger with more samples available in the template set.

$Q SUPER-QUICK RUNTIME ON A VERY LOW-END CPU

We wanted to learn how much extra performance $Q can deliver on a very low-end CPU. To this end, we did one more experiment, running $Q on an ARM processor released before the Cortex-A series. Our choice was the ARM-1136, a 600-MHz CPU implementing the ARMv6 architecture [5]. Classification times measured on ARM-1136 showed that original $P took 33.3 s on average to classify a candidate gesture when the number of templates per gesture type (T) varied from 1 to 64. Also, $P needed almost 2 minutes (115.3 s) to perform a single classification when T was 64. However, $Q took less than one second (813 ms) to produce the same classification result in the worst-case scenario, with a speed-up of 142× and a time savings of 99.3%. On average, $Q classified a gesture on the ARM-1136 in just 389 ms, representing a speed-up of 86× and 98.8% time savings compared to $P's 33.3 seconds!

CONCLUSION

We have shown a massive speed-up of the already-powerful $P recognizer [50], resulting in the new $Q, a super-quick articulation-invariant gesture recognizer for low-resource devices. $Q uses less than 3% of the computations of $P, making it capable of interactive speeds on a variety of architectures with limited computing capabilities. We achieved this performance by applying successive optimization layers to $P, from standard coding tricks that reduced CPU time by 42%, to an early abandoning filter that reduced another 44% with just one extra line of code (86% overall gain), to a more complex, yet very efficient lower-bounding filter that achieved a total reduction of 97% CPU time compared to $P.

$Q can speed up classification performance on high-end devices as well, such as notebooks and touch tables, not just on low-resource devices. Thus, $Q can be used effectively in "time-budget" scenarios, where the gesture recognizer is given just a limited CPU time to run [43]. For high-end devices, only some of $Q's speed-up layers would probably be more than enough to boost $P's classification performance considerably. Conveniently, these layers can be activated independently. For example, the standard code optimization and the early abandoning layers are easily implementable with changes affecting a maximum of 7 lines of $P code, yet will likely deliver significant speed-ups on a wide range of high-end devices.

Complete pseudocode of the $Q stroke-gesture recognizer is provided in the Appendix. $Q is implementable on top of $P with only 30 extra lines of code and, overall, $Q has only 100 lines of platform-independent code. We also provide C#, Java (Android), and JavaScript implementations of $Q at http://depts.washington.edu/madlab/proj/dollar/qdollar.html.

$Q sets the new standard in classification speed and recognition accuracy for articulation-invariant gesture recognizers, being capable of supporting numerous future applications on devices of all capabilities and types. We expect $Q to accelerate innovations in gesture input for mobile, wearable, and embedded scenarios, enabling the community to implement very fast and accurate gesture recognition for such platforms.

ACKNOWLEDGMENTS

R.D. Vatavu acknowledges support from the project PN-III-P2-2.1-PED-2016-0688 (209PED/2017), UEFISCDI, Romania. L. Anthony acknowledges partial support from the NSF Grant Award #IIS-1552598. J.O. Wobbrock acknowledges a Google Faculty Award. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect these agencies' views.

[4] 16 distinct gesture types × 90 templates per gesture type = 1440.

[5] As mentioned, these constants change with the CPU, as they reflect the various speed of running basic operations. For instance, values were $a_0 = 0.0031$, $a_1 = 0.0407$, $a_2 = -0.0166$, $a_3 = 0.0015$, and $a_4 = 0.7326$ for the Cortex-A9 CPU ($R^2 = .997$).

REFERENCES

1. L. Anthony, Q. Brown, J. Nias, and B. Tate. 2015. Children (and Adults) Benefit from Visual Feedback During Gesture Interaction on Mobile Touchscreen Devices. *Int. J. Child-Comp. Interact.* 6 (2015), 17–27. DOI:http://dx.doi.org/10.1016/j.ijcci.2016.01.002
2. Lisa Anthony and Jacob O. Wobbrock. 2010. A Lightweight Multistroke Recognizer for User Interface Prototypes. In *Proc. of GI '10*. 245–252. http://dl.acm.org/citation.cfm?id=1839214.1839258
3. Lisa Anthony and Jacob O. Wobbrock. 2012. $N-protractor: A Fast and Accurate Multistroke Recognizer. In *Proc. of GI '12*. 117–120. http://dl.acm.org/citation.cfm?id=2305276.2305296
4. ARM. 2016. Humanizing the Digital World: The world's leading semiconductor intellectual property (IP) supplier. (2016). http://www.arm.com/about/company-profile/
5. ARM. 2017a. ARM1136 processor. (2017). https://www.arm.com/products/processors/classic/arm11/arm1136.php
6. ARM. 2017b. Cortex-A5 processor. (2017). http://www.arm.com/products/processors/cortex-a/cortex-a5.php
7. ARM. 2017c. Cortex-A53 processor. (2017). http://www.arm.com/products/processors/cortex-a/cortex-a53-processor.php
8. ARM. 2017d. Cortex-A7 processor. (2017). https://www.arm.com/products/processors/cortex-a/cortex-a7.php
9. ARM. 2017e. Cortex-A9 processor. (2017). http://www.arm.com/products/processors/cortex-a/cortex-a9.php
10. R. Balcazar, F.R. Ortega, K. Tarre, A. Barreto, M. Weiss, and N.D. Rishe. 2017. CircGR: Interactive Multi-Touch Gesture Recognition Using Circular Measurements. In *Proc. of ISS '17*. ACM, New York, NY, USA, 12–21. DOI:http://dx.doi.org/10.1145/3132272.3134139
11. Rainer Burkard, Mauro Dell'Amico, and Silvano Martello. 2009. *Assignment Problems*. Society for Industrial and Applied Mathematics, Philadelphia, PA.
12. G. Casiez, N. Roussel, and D. Vogel. 2012. 1€Filter: A Simple Speed-based Low-pass Filter for Noisy Input in Interactive Systems. In *Proc. of CHI '12*. 2527–2530. DOI:http://dx.doi.org/10.1145/2207676.2208639
13. D.C. Cireşan, U. Meier, L.M. Gambardella, and J. Schmidhuber. 2010. Deep, Big, Simple Neural Nets for Handwritten Digit Recognition. *Neural Computation* 22, 12 (Dec. 2010), 3207–3220. DOI: http://dx.doi.org/10.1162/NECO_a_00052
14. Tim Cross. 2016. After Moore's Law. *The Economist*. (March 2016). http://www.economist.com/technology-quarterly/2016-03-12/after-moores-law
15. Jack Edmonds. 1965. Paths, trees, and flowers. *Canadian Journal of Mathematics* 17 (1965), 449–467. http://dx.doi.org/10.4153/CJM-1965-045-4
16. T.A. Hammond and R. Davis. 2009. Recognizing Interspersed Sketches Quickly. In *GI '09*. 157–166. http://dl.acm.org/citation.cfm?id=1555880.1555917
17. Tracy Hammond and Brandon Paulson. 2011. Recognizing Sketched Multistroke Primitives. *ACM TiiS* 1, 1, Article 4 (Oct. 2011), 34 pages. DOI: http://dx.doi.org/10.1145/2030365.2030369
18. Chris Harrison, Desney Tan, and Dan Morris. 2011. Skinput: Appropriating the Skin As an Interactive Canvas. *Commun. ACM* 54, 8 (Aug. 2011), 111–118. DOI: http://dx.doi.org/10.1145/1978542.1978564
19. J. Herold and T. F. Stahovich. 2012. The 1¢Recognizer: A Fast, Accurate, and Easy-to-implement Handwritten Gesture Recognition Technique. In *Proc. of SBIM '12*. http://dl.acm.org/citation.cfm?id=2331067.2331074
20. L.B. Kara and T.F. Stahovich. 2004. Hierarchical Parsing and Recognition of Hand-sketched Diagrams. In *Proc. of UIST '04*. ACM, 13–22. DOI: http://dx.doi.org/10.1145/1029632.1029636
21. L.B. Kara and T.F. Stahovich. 2005. An Image-based, Trainable Symbol Recognizer for Hand-drawn Sketches. *Comput. Graph.* 29, 4 (2005), 501–517. DOI: http://dx.doi.org/10.1016/j.cag.2005.05.004
22. Kris Kaspersky. 2003. *Code Optimization: Effective Memory Usage*. A-List Publishing.
23. Eamonn Keogh. 2002. Exact Indexing of Dynamic Time Warping. In *Proc. of VLDB '02*. 406–417. http://dl.acm.org/citation.cfm?id=1287369.1287405
24. Sven Kratz and Michael Rohs. 2010. A $3 Gesture Recognizer: Simple Gesture Recognition for Devices Equipped with 3D Acceleration Sensors. In *Proc. of IUI '10*. ACM, New York, NY, USA, 341–344. DOI: http://dx.doi.org/10.1145/1719970.1720026
25. Per-Ola Kristensson and Shumin Zhai. 2004. SHARK2: A Large Vocabulary Shorthand Writing System for Pen-based Computers. In *Proc. of UIST '04*. ACM, 43–52. DOI:http://dx.doi.org/10.1145/1029632.1029640
26. Mary LaLomia. 1994. User Acceptance of Handwritten Recognition Accuracy. In *Companion CHI '94*. 107–108. DOI:http://dx.doi.org/10.1145/259963.260086
27. Yang Li. 2010. Protractor: A Fast and Accurate Gesture Recognizer. In *Proc. of CHI '10*. 2169–2172. DOI: http://dx.doi.org/10.1145/1753326.1753654
28. Hao Lü, James A. Fogarty, and Yang Li. 2014. Gesture Script: Recognizing Gestures and Their Structure Using Rendering Scripts and Interactively Trained Parts. In *Proc. of CHI '14*. ACM, 1685–1694. DOI: http://dx.doi.org/10.1145/2556288.2557263
29. Steve McConnell. 2004. *Code Complete: A Practical Handbook of Software Construction, 2nd Ed.* Microsoft Press. http://www.cc2e.com/Default.aspx
30. David McNeill. 1992. *Hand and Mind: What Gestures Reveal about Thought*. University of Chicago Press.
31. Martez E. Mott, Radu-Daniel Vatavu, Shaun K. Kane, and Jacob O. Wobbrock. 2016. Smart Touch: Improving Touch Accuracy for People with Motor Impairments with Template Matching. In *Proc. of CHI '16*. 1934–1946. DOI:http://dx.doi.org/10.1145/2858036.2858390

32. C.S. Myers and L.R. Rabiner. 1981. A comparative study of several dynamic time-warping algorithms for connected-word recognition. *The Bell System Technical Journal* 60, 7 (1981), 1389–1409. DOI: http://dx.doi.org/10.1002/j.1538-7305.1981.tb00272.x

33. Corey Pittman, Eugene M. Taranta II, and Joseph J. LaViola, Jr. 2016. A $-Family Friendly Approach to Prototype Selection. In *Proc. of IUI '16*. ACM, 370–374. DOI:http://dx.doi.org/10.1145/2856767.2856808

34. William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. 1992. *Numerical Recipes in C (2nd Ed.): The Art of Scientific Computing*. Cambridge University Press, New York, NY, USA.

35. T. Rakthanmanon, B. Campana, A. Mueen, G. Batista, B. Westover, Q. Zhu, J. Zakaria, and E. Keogh. 2012. Searching and Mining Trillions of Time Series Subsequences Under Dynamic Time Warping. In *Proc. of KDD '12*. 262–270. DOI: http://dx.doi.org/10.1145/2339530.2339576

36. T. Rakthanmanon, B. Campana, A. Mueen, G. Batista, B. Westover, Q. Zhu, J. Zakaria, and E. Keogh. 2013. Addressing Big Data Time Series: Mining Trillions of Time Series Subsequences Under Dynamic Time Warping. *ACM Trans. Knowl. Discov. Data* 7, 3 (2013), 10:1–10:31. DOI:http://dx.doi.org/10.1145/2500489

37. Chotirat Ann Ratanamahatana and Eamonn Keogh. 2005. Three Myths about Dynamic Time Warping. In *Proc. of SDM '05*. 506–510.

38. J. Reaver, T. F. Stahovich, and J. Herold. 2011. How to Make a Quick$: Using Hierarchical Clustering to Improve the Efficiency of the Dollar Recognizer. In *Proc. of SBIM '11*. ACM, New York, NY, USA, 103–108. DOI: http://dx.doi.org/10.1145/2021164.2021183

39. Dean Rubine. 1991. Specifying Gestures by Example. In *Proc. of SIGGRAPH '91*. ACM, New York, NY, USA, 329–337. DOI:http://dx.doi.org/10.1145/122718.122753

40. T. Scott Saponas, Chris Harrison, and Hrvoje Benko. 2011. PocketTouch: Through-fabric Capacitive Touch Input. In *Proc. of UIST '11*. ACM, 303–308. DOI: http://dx.doi.org/10.1145/2047196.2047235

41. Stefan Schneegass and Alexandra Voit. 2016. GestureSleeve: Using Touch Sensitive Fabrics for Gestural Input on the Forearm for Controlling Smartwatches. In *Proc. of ISWC '16*. ACM, 108–115. DOI:http://dx.doi.org/10.1145/2971763.2971797

42. Tevfik Metin Sezgin and Randall Davis. 2005. HMM-based Efficient Sketch Recognition. In *Proc. of IUI '05*. ACM, New York, NY, USA, 281–283. DOI: http://dx.doi.org/10.1145/1040830.1040899

43. Eugene M. Taranta, II and Joseph J. LaViola, Jr. 2015. Penny Pincher: A Blazing Fast, Highly Accurate $-family Recognizer. In *Proc. of GI '15*. 195–202. http://dl.acm.org/citation.cfm?id=2788890.2788925

44. Eugene M. Taranta, II, Andrés N. Vargas, and Joseph J. LaViola. 2016. Streamlined and Accurate Gesture Recognition with Penny Pincher. *Comput. Graph.* 55, C (April 2016), 130–142. DOI: http://dx.doi.org/10.1016/j.cag.2015.10.011

45. Eugene M. Taranta II, Amirreza Samiei, Mehran Maghoumi, Pooya Khaloo, Corey R. Pittman, and Joseph J. LaViola Jr. 2017. Jackknife: A Reliable Recognizer with Few Samples and Many Modalities. In *Proc. of CHI '17*. ACM, 5850–5861. DOI: http://dx.doi.org/10.1145/3025453.3026002

46. Radu-Daniel Vatavu. 2011. The Effect of Sampling Rate on the Performance of Template-based Gesture Recognizers. In *Proc. of ICMI '11*. ACM, 271–278. DOI: http://dx.doi.org/10.1145/2070481.2070531

47. Radu-Daniel Vatavu. 2012a. 1F: One Accessory Feature Design for Gesture Recognizers. In *Proc. of IUI '12*. ACM, New York, NY, USA, 297–300. DOI: http://dx.doi.org/10.1145/2166966.2167022

48. Radu-Daniel Vatavu. 2012b. Small Gestures Go a Long Way: How Many Bits Per Gesture Do Recognizers Actually Need?. In *Proc. of DIS '12*. ACM, 328–337. DOI:http://dx.doi.org/10.1145/2317956.2318006

49. Radu-Daniel Vatavu. 2017. Improving Gesture Recognition Accuracy on Touch Screens for Users with Low Vision. In *Proc. of CHI '17*. 4667–4679. DOI: http://dx.doi.org/10.1145/3025453.3025941

50. Radu-Daniel Vatavu, Lisa Anthony, and Jacob O. Wobbrock. 2012. Gestures As Point Clouds: A $P Recognizer for User Interface Prototypes. In *Proc. of ICMI '12*. ACM, New York, NY, USA, 273–280. DOI: http://dx.doi.org/10.1145/2388676.2388732

51. Andrew Webb. 2002. *Statistical Pattern Recognition, 2nd Ed.* John Wiley & Sons, West Sussex, England.

52. Jacob O. Wobbrock. 2009. TapSongs: Tapping Rhythm-based Passwords on a Single Binary Sensor. In *Proc. of UIST '09*. ACM, New York, NY, USA, 93–96. DOI:http://dx.doi.org/10.1145/1622176.1622194

53. Jacob O. Wobbrock, Andrew D. Wilson, and Yang Li. 2007. Gestures Without Libraries, Toolkits or Training: A $1 Recognizer for User Interface Prototypes. In *Proc. of UIST '07*. ACM, 159–168. DOI: http://dx.doi.org/10.1145/1294211.1294238

54. Chao Xu, Parth H. Pathak, and Prasant Mohapatra. 2015. Finger-writing with Smartwatch: A Case for Finger and Hand Gesture Recognition Using Smartwatch. In *Proc. of HotMobile '15*. ACM, 9–14. DOI: http://dx.doi.org/10.1145/2699343.2699350

55. Shumin Zhai, Per Kristensson, Caroline Appert, Tue Andersen, and Xiang Cao. 2012. Foundational Issues in Touch-Surface Stroke Gesture Design: An Integrative Review. *Found. Trends HCI* 5, 2 (Feb. 2012), 97–205. DOI:http://dx.doi.org/10.1561/1100000012

56. Shumin Zhai and Per-Ola Kristensson. 2003. Shorthand Writing on Stylus Keyboard. In *Proc. of CHI '03*. ACM, New York, NY, USA, 97–104. DOI: http://dx.doi.org/10.1145/642611.642630

PSEUDOCODE FOR THE $Q RECOGNIZER

We provide complete pseudocode for $Q. In the following, POINT is a structure that exposes x, y, and *strokeId* properties; *strokeId* is the stroke index a point belongs to and is filled by counting touch down/touch up events; x and y are integers in the set $\{0, 1, ..., m-1\}$, where m is the size of the look-up table. POINTS is a list of points and TEMPLATES a list of POINTS with associated gesture class data. The pseudocode assumes that templates have already been preprocessed (when loading the template set, for instance).

$Q-RECOGNIZER (POINTS *points*, TEMPLATES *templates*)

```
n ← 32, m ← 64  // defaults for cloud size (n) and size of the look-up table (m)
NORMALIZE(points, n, m)  // templates have already been normalized
score ← ∞
for each template in templates do
    d ← CLOUD-MATCH(points, template, n, score)
    if d < score then
        score ← d
        result ← template
return ⟨result, score⟩  // the closest template from the set and the smallest score
```

CLOUD-MATCH (POINTS *points*, POINTS *template*, int *n*, int *min*)

```
step ← ⌊n^0.5⌋
// compute lower bounds for both matching directions between points and template
LB1 ← COMPUTE-LOWER-BOUND(points, template, step, template.LUT)
LB2 ← COMPUTE-LOWER-BOUND(template, points, step, points.LUT)
for i ← 0 to n − 1 step step do
    if LB1[i/step] < min then
        min ← MIN(min, CLOUD-DISTANCE(points, template, n, i, min))
    if LB2[i/step] < min then
        min ← MIN(min, CLOUD-DISTANCE(template, points, n, i, min))
return min
```

CLOUD-DISTANCE (POINTS *points*, POINTS *template*, int *n*, int *start*, float *minSoFar*)

```
unmatched ← {0, 1, 2, ..., n − 1}  // indices of unmatched points from template
i ← start  // start the matching from this index in the points cloud
weight ← n  // weights decrease from n to 1
sum ← 0  // computes the cloud distance between points and template
do
    min ← ∞
    for each j in unmatched do
        d ← SQR-EUCLIDEAN-DISTANCE(points[i], template[j])
        if d < min then
            min ← d
            index ← j
    REMOVE(unmatched, index)  // implementable in O(1)
    sum ← sum + weight · min
    if sum ≥ minSoFar then
        return sum  // early abandoning of computations
    weight ← weight − 1  // weights decrease from n to 1
    i ← (i + 1) MOD n  // advance to the next point in points
until i == start
return sum
```

COMPUTE-LOWER-BOUND (POINTS *points*, POINTS *template*, int *step*, int[,] *LUT*)

```
LB ← new float[n/step + 1]  // multiple lower bounds, one for each starting point
SAT ← new float[n]  // summed area table for fast computations (see text)
// first, compute the lower bound for starting point index 0
LB[0] ← 0
for i ← 0 to n − 1 do
    index ← LUT[points[i].x, points[i].y]
    d ← SQR-EUCLIDEAN-DISTANCE(points[i], template[index])
    SAT[i] ← (i == 0) ? d : SAT[i−1] + d
    LB[0] ← LB[0] + (n − i) · d
// compute the lower bound for the other starting points (see formula in the text)
for i ← step to n − 1 step step do
    LB[i/step] ← LB[0] + i · SAT[n−1] − n · SAT[i−1]
return LB
```

The following pseudocode implements gesture preprocessing: resampling, translation to origin, rescaling into the $m \times m$ grid, and computation of the look-up table. Except for the new COMPUTE-LUT function and changes in the SCALE function, this pseudocode is practically the same as for $P [50] (p. 280).

NORMALIZE (POINTS *points*, int *n*, int *m*)

```
points ← RESAMPLE(points, n)
TRANSLATE-TO-ORIGIN(points, n)
SCALE(points, m)
LUT ← COMPUTE-LUT(m, points, n)
```

RESAMPLE (POINTS *points*, int *n*)

```
I ← PATH-LENGTH(points) / (n − 1)
D ← 0
newPoints ← {points[0]}
for i ← 1 to n − 1 do
    if points[i].strokeId == points[i−1].strokeId then
        d ← EUCLIDEAN-DISTANCE(points[i−1], points[i])
        if (D + d) ≥ I then
            q.x ← points[i−1].x + (I − D)/d · (points[i].x - points[i−1].x)
            q.y ← points[i−1].y + (I − D)/d · (points[i].y - points[i−1].y)
            APPEND(newPoints, q)
            INSERT(points, i, q) // q will be the next points[i]
            D ← 0
        else D ← D + d
return newPoints
```

TRANSLATE-TO-ORIGIN (POINTS *points*, int *n*)

```
c ← (0, 0)  // will compute the centroid of the points cloud
for each p in points do
    c ← (c.x + p.x, c.y + p.y)
c ← (c.x/n, c.y/n)
for each p in points do
    p ← (p.x − c.x, p.y − c.y)
```

SCALE (POINTS *points*, int *m*)

```
x_min ← ∞, x_max ← −∞, y_min ← ∞, y_max ← −∞
for each p in points do
    x_min ← MIN(x_min, p.x)
    y_min ← MIN(y_min, p.y)
    x_max ← MAX(x_max, p.x)
    y_max ← MAX(y_max, p.y)
s ← MAX(x_max − x_min, y_max − y_min)/(m − 1)  // scale factor
for each p in points do
    p ← ((p.x − x_min)/s, (p.y − y_min)/s)  // p.x and p.y are now integers in 0...m − 1
```

COMPUTE-LUT (POINTS *points*, int *n*, int *m*)

```
LUT ← new int[m, m]
for x ← 0 to m − 1 do
    for y ← 0 to m − 1 do
        min ← ∞
        for i ← 0 to n − 1 do
            d ← SQR-EUCLIDEAN-DISTANCE(points[i], new POINT(x, y))
            if d < min then
                min ← d
                index ← i
        LUT[x, y] ← index
return LUT
```

PATH-LENGTH (POINTS *points*)

```
d ← 0  // will compute the path length
for i ← 1 to n − 1 do
    if points[i].strokeId == points[i−1].strokeId then
        d ← d + EUCLIDEAN-DISTANCE(points[i−1], points[i])
return d
```

SQR-EUCLIDEAN-DISTANCE (POINT *a*, POINT *b*)

```
return (a.x − b.x)^2 + (a.y − b.y)^2  // much faster to compute without the sqrt()
```

EUCLIDEAN-DISTANCE (POINT *a*, POINT *b*)

```
return sqrt(SQR-EUCLIDEAN-DISTANCE(a, b))
```