Machete: Easy, Efficient, and Precise Continuous Custom Gesture Segmentation

EUGENE M. TARANTA II, COREY R. PITTMAN, MEHRAN MAGHOUMI, MYKOLA MASLYCH, YASMINE M. MOOLENAAR, and JOSEPH J. LAVIOLA JR., University of Central Florida

We present Machete, a straightforward segmenter one can use to isolate custom gestures in continuous input. Machete uses traditional continuous dynamic programming with a novel dissimilarity measure to align incoming data with gesture class templates in real time. Advantages of Machete over alternative techniques is that our segmenter is computationally efficient, accurate, device-agnostic, and works with a single training sample. We demonstrate Machete's effectiveness through an extensive evaluation using four new high-activity datasets that combine puppeteering, direct manipulation, and gestures. We find that Machete outperforms three alternative techniques in segmentation accuracy and latency, making Machete the most performant segmenter. We further show that when combined with a custom gesture recognizer, Machete is the only option that achieves both high recognition accuracy and low latency in a video game application.

CCS Concepts: • **Human-centered computing → Gestural input**;

Additional Key Words and Phrases: Continuous, custom, gesture, segmentation, CDP, DTW

ACM Reference format:
Eugene M. Taranta II, Corey R. Pittman, Mehran Maghoumi, Mykola Maslych, Yasmine M. Moolenaar, and Joseph J. Laviola Jr. 2021. Machete: Easy, Efficient, and Precise Continuous Custom Gesture Segmentation. *ACM Trans. Comput.-Hum. Interact.* 28, 1, Article 5 (January 2021), 46 pages.
https://doi.org/10.1145/3428068

1 INTRODUCTION

The same natural and non-verbal articulations we use to communicate with one another also allow individuals to interact with software systems when leveraged in user interface design. Indeed gestures have permeated every edge of the user interface boundary having been exploited through interactive displays, smart watches, handheld game controllers, and passive sensor tracking systems; and it may be that this trend ceases only when and if brain–computer interfaces obtain the same level of maturity that today's alternative low cost input devices enjoy. As such, researchers continue to investigate how to maximize gesture utility as well as how best to recognize gesture patterns. One such branch of research fueled by continued success and demand addresses issues that surface at the intersection between customization and simplicity. We have researchers, designers,

Authors' addresses: E. M. Taranta II, C. R. Pittman, M. Maghoumi, M. Maslych, Y. M. Moolenaar, and J. J. LaViola Jr., University of Central Florida, 4000 Central Florida Blvd, Orlando, FL, 32816; emails: etaranta@gmail.com, cpittman@knights.ucf.edu, mehran@cs.ucf.edu, {maslychm, ymmoolenaar}@knights.ucf.edu, jjl@eecs.ucf.edu.

1073-0516/2021/01-ART5 $15.00
https://doi.org/10.1145/3428068

and users who will train their software to recognize unique and idiosyncratic custom gesture patterns so long as their system is content with only one or two demonstrations of a particular gesture. For this reason, we require methods that work well with very little training data. Further, we have researchers, designers, and practitioners who by trade are neither mathematicians, computer scientists, nor pattern matching experts but want customization in their interface and will implement or integrate recognizers that are sufficiently easy to comprehend. It has been a non-trivial effort addressing these requirements, but the community has made great progress over the last decade through the advancement of $-family recognizers, including $1 [89], $N [3], Protractor [46], $N-Protractor [4], $P [78], $P+ [77], and $Q [79], as well as those recognizers inspired by them, such as Penny Pincher [76], Jackknife [74], $3 [41], and Protractor 3D [42], among others [12, 26, 28]. These recognizers enable customization using purposefully straightforward algorithms and mathematics.

Yet, a number of common input device types, including those that use passive sensors, provide continuous input sampled from human motion, and while there are a number of approaches for localizing gesture end points over continuous streams, only a few are $-family principled. Of these, window-based segmentation is perhaps the most common approach, where on a per frame basis one evaluates the most recent fixed-length set of input samples using an ordinary recognizer [34, 48, 59, 69]. Other approaches correlate discontinuities in an input signal with gesture boundaries [17, 35, 36], whereas others just use simple heuristics [40, 43]. Needless to say, each method has its own set of drawbacks relating to computational inefficiency, imprecision, or both, and this motivates us to present a new solution we call Machete, a novel, continuous dynamic programming (CDP) [58] based solution tailored for fast and efficient custom gesture segmentation.

Machete offers two primary advantages over alternative methods. First and foremost, our technique is computationally efficient, whereas other techniques such as energy-based segmentation frequently invoke an underlying robust recognizer to evaluate segmented gesture candidates, we are able to prune away most candidates, which is critical for resource strapped platforms and real-time recognition systems. Second, we do not require a static boundary such as with a sliding window, which means Machete is able to support a greater temporal variability in gesture production. To evaluate our approach, we collected continuous high activity (HA)[1] datasets from 30 participants using 3 input devices, 1 of which we split into 2 subsets to create a total of 4 unique and varying datasets. We then compared Machete to three alternative segmentation techniques and demonstrate that across four different error measures, our technique is the only method that consistently achieves high accuracy across all conditions. Specifically, in three cases, Machete outperforms all segmenters; and in the fourth case, the most accurate method not only requires excessive computation but is the least accurate method across the prior three conditions. Further, Machete is significantly more efficient, reducing processing time by as much as 98%, and in our view these reasons make Machete the best option for a variety of continuous custom gesture recognition problems. Thus, the contributions of this article are:

(1) A novel segmentation technique based on CDP for custom gestures that is simple yet highly effective. Specifically, we introduce a new local cost function that improves CDP's ability to match patterns. We also introduce a new measure that improves the Douglas–Peucker (DP) line simplification algorithm [21] we use to train our segmenter.
(2) Four HA datasets that mix gestures with direct manipulation and puppeteering like interactions. One can download our datasets at https://github.com/ISUE/Machete.

[1]By high activity we mean that participants were continuously moving without breaks in a way that combined gesture with non-gesture activities.

(3) An evaluation of Machete that demonstrates competitive segmentation accuracy and superior computational performance. In a first test, we look at the effect of four segmentation approaches on our HA datasets. In a second test, we look at the effect of segmentation on recognizer accuracy and latency in a Unity-based application.

2 MOTIVATION

In this section, we define the language of this manuscript and motivate our decision to focus on customization using straightforward techniques. To begin, by *gesture* we understand a neuromuscular response resulting from an intentional communication that is captured via an *input device* that periodically samples the environment. A gesture *recognizer* is able to analyze an input device signal and determine to which gesture *class* a given sequence of motion most likely belongs. When a person is unable to delineate gesture end points through mechanical mechanisms, such as a trigger toggle or touch event, then the signal is continuous and the system must algorithmically localize end points to either *segment* the input, or engage in gesture *spotting*. Segmentation and spotting are often used interchangeably, but by spotting we mean that the system only needs to recognize gesticulations as they occur, not identify where within time their end points localize. To train a system, one can specify gestures through high-level language descriptions [44, 62] or by example [63]. In the latter case, example motions sampled from gesticulating individuals are used to train class *models*, from which the system learns one more representations of a target class. In general, more examples correlate with better performance, but to facilitate end-user customization, the system must work well with limited training data. An extreme case of this is *one shot* learning, where the system is given only one example of each gesture. Our goal in this work is to deliver fast and robust custom gesture segmentation using only straightforward, $-family principled techniques. In the remainder of this section, we further describe the importance of customization, simplicity, and speed.

2.1 The Virtues of Gesture Customization

Nacenta et al. [52] describe four gesture set types: stock gestures, pre-designed gestures, user-elicited gestures,[2] and user-defined gestures. Generic system-wide available actions such as pinch, rotate, and swipe are stock gestures that are widely available to designers for use in their own software. Stock options are likely limited and therefore least expressive, as their shoehorned inclusion into a user interface may result in rather ambiguous associations, e.g., pinch to open a jogging playlist. Alternatively, designers who leverage their expertise and domain knowledge can pre-design gesture sets that are in their view meaningful, memorable, and well separated in a recognizer's feature space. Or, when time permits and sufficient resources are available, designers may instead choose to conduct an elicitation study so as to determine what is an appropriate gesture set for their interface. Finally, as it pertains to customization, designers can let users define their own gestures. Herein, we are concerned primarily with this latter type. We believe that support for gesture customization in user interface design inherently possesses functional, practical, and cognitive virtue.

As a functional virtue: The need for customization becomes apparent in designs where predefined gestures, user-elicited gestures, and stock gestures are inadequate or unworkable. For instance, one may envision a supernatural game where players create their own wand spells using an HTC Vive controller, or a system where meaningful shortcuts are required for an unspecified number of referents. Custom gestures are further useful in security and authentication, such as

[2]Nacenta et al. actually filed user-elicited gestures under pre-designed gestures, but for reasons soon explained, we keep them separate.

to increase password entropy and complicate shoulder surfing attacks [54]. Or as a final example, those with motor impairments are sometimes unable to interact with a given device using typical able-bodied articulations [2]. Customization can overcome this issue by allowing impaired individuals to define gesture sets that accommodate their impairment.

As a practical virtue: Once a designer decides that stock gestures are inadequate, he or she may consider using pre-designed gestures, yet designers often get it wrong [51]. We are often unsure of how users want to interact with our software; or what seems appropriate to us as experts turns out to be awkward and unfamiliar to others. However, we can overcome this issue with iterative design and elicitation studies. Norton et al. [55], for instance, conducted a Wizard-of-Oz study to explore full body interaction in a parkour game environment. Wobbrock et al. [88] conducted an elicitation study to learn how non-technical users gesture on interactive tabletops. Chan et al. [14] performed a similar study for single-hand microgestures (SHMG). Such investigations yield workable gesture sets and recommendations for user interface design within their respective domains that, while being quite useful, have two limitations. First, established findings are specific to their domain, e.g., an SHMG consensus set is unlikely to be of value to one who is designing a Vive game. In other words, if prior knowledge is unavailable or without a direct application to one’s interface, he or she will still need to conduct their own study, which, to the second point, can be time and cost prohibitive. Finding sufficient time, participants, and capital to run a user study is sometimes no small feat. These reasons make customization an attractive *practical* alternative, especially to the indie developer, prototyper, hobbyist, and student.

As a cognitive virtue: It should first be understood that gestures are personal. When creating custom gesture sets, users tend to prioritize the intuitive, natural, and obvious first; the simple and easy second; and the familiar third [57]. But the result is still something unique; that is, we each have our own ideas about which actions ought to map to which functions based on internal conceptualizations, associations, and perceptions. This may be why we never see perfect agreement for all tasks in elicitation studies, and why custom gesture sets require less concentration and are easier to remember [52].

These virtues compel us to focus on customization and consequently techniques that work with very little training data. Because if we wish to support gesture customization, we must also ensure we do not burden the user [46]. Yet how we choose to go about designing such a system is of equal importance, which leads us next to $-family principles.

2.2 $-family Ideology and Practice

Human–computer interaction (HCI) researchers often engage in rapid prototyping to iteratively design, evaluate, and refine user interfaces. Some researchers further incorporate custom gestures into their software so as to facilitate natural and fluid interactions. Since HCI researchers vary in skill, it is common to find people in the field who are familiar with neither machine learning nor the advanced mathematics employed by pattern matching techniques that enable gesture recognition. $-family recognizers[3] serve these individuals by offering straightforward solutions to complex pattern matching problems, so that researchers can incorporate custom gesture recognition into their interfaces with comparatively little effort. Wobbrock et al. introduced the first $-family recognizer, $1 [89], after which a long and varied series of work followed. It is because of their reliability, relatability, simplicity, and adherence to the fundamental principles outlined below that $-family recognizers have been widely adopted by the community.[4]

[3]For simplicity, we refer to the core recognizers [3, 4, 46, 77–79, 89] and those inspired by as the $-family collectively.
[4]Impact of the $-family is discussed at http://depts.washington.edu/acelab/proj/dollar/impact.html.

We too are motivated by the $-family design philosophy in order to provide a robust segmentation solution that is accessible to a large and varied talent pool, ranging in background from developer to designer, and in skill from neophyte to expert. In our interpretation of the $-family literature, tenets of the philosophy stand on:

(1) *Independence*: One can implement a proposed technique from scratch without requiring external library support, thereby enabling gesture recognition on new platforms or in new programming languages as they come into existence, where popular libraries do not already exist. In other words, if necessary, one can implement all required machinery without difficulty.
(2) *Foundational Mathematics*: Computations involve only straightforward algebra, geometry, and statistics that most learn early in their academic careers.
(3) *Relatable Representations*: How a method internally represents a class model is straightforward, utilizing constructs that are easy to understand and visualize.
(4) *Basic Algorithms*: Techniques employ algorithms that can be understood by those who have completed their first computer science course or have equivalent experience.
(5) *Competitive Accuracy*: Recognizers achieve high accuracy with limited training data so as to avoid burdening users and not push integrators toward more complex solutions.
(6) *Customizability*: New gesture classes and examples can be added to the system at runtime without significant effort or delay.

Adherence to these goodness criteria bestow additional benefits onto developers in that one can quickly understand, implement, and debug such recognizers. And because most variants come with reference code, pseudocode, or both, efforts are further hastened.

Although, there is yet no objective measure one can use to evaluate whether a proposed solution is $-family compatible, one can at least compare and contrast with non-compliant recognizers to gain an intuitive notion of what the community considers orthodox. To give a taste, many advanced approaches use statistical features to describe and model gesture classes that are hand-engineered [7, 20, 30, 31, 67] and derived from domain or problem-specific knowledge. One of the more popular and easier to understand solutions developed by Rubine [63] uses linear discriminate analysis, which constructs and inverts a covariance matrix. Poor feature selection can lead to poor performance or singular matrices—two issues that are hard to conceptualize and debug without prior experience. To evaluate the similarity of query data to class models, advanced recognition techniques may use naive-Bayes [87], hidden Markov models (HMM) [91], support vector machines (SVM) [83], random forests [75], or deep learning [49], to name a few. $-family recognizers, on the other hand, typically use simple point or vector data representations with nearest neighbor pattern matching, which is said to be the simplest instance-based learning method [50]. Techniques that have been employed as part of the data preparation and measurement include uniform spatial resampling, min-max scaling, the DP line simplification algorithm [21], dynamic time warping (DTW) [74], golden section search [89], Euclidean distance (ED), angular cosine distance, stochastic resampling [72], nonlinear embedding [32, 33], lower bounding [74, 79], point cloud matching [78], and the like.

2.3 The Need for Speed

Since $1's inception, researchers have put a concerted effort into optimizing recognizer performance for three important reasons:

(1) *Real-time Performance*: User interface design often requires low latency feedback in order to maintain a fluid interaction. For instance, it has been shown that users perceive, plan,

and react to millisecond delays [27]. And although numerous applications do not require real-time performance, we still prefer responsive feedback. Further, processing queries should not delay other ongoing operations, as software does much more than just analyze input—it must share hardware resources with other system processes [76].

(2) *Low-Resource Devices*: Mobile, wearable, and embedded devices employ low-resource computing technology to preserve power and lower cost, and to enable gesture recognition on such devices, we require highly efficient techniques [79].

(3) *Quantity*: In general, recognizer accuracy improves with training set size, whereas responsiveness declines under an increased workload. Intuitively, algorithmically and computationally efficient techniques are able to process more templates per unit time without compromising runtime performance requirements [76].

For these reasons, we place special emphasis on computational performance. As will be shown, our approach is exceptionally fast, yet highly accurate.

2.4 Effect of Segmentation on User Experience

The techniques we explore in this work use a hierarchical approach where a segmenter first identifies the temporal boundaries of a gesture candidate and then a recognizer classifies the candidate. Gesture candidates may be rejected or mapped to a known gesture class. In a customization context, especially when using nearest neighbor pattern matching, the probability that a recognizer correctly classifies a true positive gesture candidate increases with segmentation accuracy. To illustrate, we discuss various error measures in Section 8.3 and, in particular, we highlight real segmentation errors in Figure 13. Visual inspection reveals that poor segmentation accuracy yields significant truncation or extension of gesture candidates, leading to higher recognizer dissimilarity scores. Improper segmentation, therefore, may lead to gesture candidate rejection when such scores are too high. One can raise the rejection threshold to combat this effect, but doing so will lead to an increase in false positives. Segmentation accuracy, therefore, plays an important role in user experience given that recognition errors can frustrate users, require greater effort to use, and reduce confidence [22, 37].

Although we discussed computational performance in the previous subsection, it is worth discussing efficiency further in the context of user experience. Arguably, window-based segmentation is the most popular method given that windowing generally yields high accuracy and is easy to implement. One issue we often face, however, is that windowing also has high computational costs where for every frame we must extract and evaluate one or more sequences from the input stream. As we demonstrate in our evaluation, this can result in a significant drop in frame rates, sometimes to below interactive rates (10–17.5 Hz depending on use case [16]). Since prior work in video games has shown that frame rates and player performance as well as perceived quality are related [19], any latency that results in frame rate reductions can be harmful to user experience. In virtual reality, higher frame rates result in smoother motion and increased presence, whereas low frames rates can also lead to motion sickness [85]. These results have led Oculus to recommend applications run at the Rift display refresh rate (90 frames per second (FPS)) or faster in their best practices report [56]. So although high segmentation accuracy may be possible, we must also consider its impact on overall software performance.

3 RELATED WORK

Gesture spotting and segmentation is said to be as difficult as gesture recognition itself [86], and like with recognition, there are countless approaches people use to solve these problems. Example questions we can ask follow: What is the probability that a given frame is a gesture boundary, regardless of class? Given all history leading up to a moment in time, to what gesture class does the present frame most likely belong? What subsequence ending at the present frame maximizes the probability that it belongs to a particular gesture class? These subtly different questions impact how one approaches segmentation.

Escalera et al.'s survey paper [24] on multi-modal gesture recognition discusses segmentation and identifies two main approaches: direct and indirect. A second survey by Weinland et al. [86] uses a slightly different organization: boundary detection, sliding windows, and grammar concatenation. Between the surveys, direct methods entail boundary detection and windowing, whereas indirect methods correspond to higher-level grammars. Therefore, we report on direct and indirect segmentation strategies, though our goal here is to inform the reader about commonly employed techniques and discuss their relation to our design goals. For more information, please refer to the surveys.

3.1 Direct Methods

Direct methods use heuristics or examine physical characteristics of motion such as velocity, acceleration, and poses to localize endpoints, but these methods tend to be unreliable in the presence of complex interactions [86]. For example, to detect gesture boundaries in dance sequences, Kahol et al. [35] hierarchically track the velocity, acceleration, and mass of various segments that are coalesced into a single force, kinetic energy, and momentum result. They then look for minima in the total body force signal to guide subsequent analysis in deciding boundaries. Kang et al. [36] similarly look for abnormal velocities, static poses, and severe curvatures to help detect candidate gesture boundaries in video games. Ye et al. [93] use curvature to segment a single stroke gesture into constituent strokes. Chen et al. [17] used energy and root mean squared (RMS) as their metric to segment real-time electromyography signals. RMS is often measured when using electromyography. Arn et al. [5] proposed the use of generalized curvature analysis to segment user gestures from continuous Kinect skeletal data. It is able to segment main motions from transitions. Regions of high curvature are transition phases, whereas main motions have lower curvature. This can be seen as the inverse of energy based segmentation, because lower regions of energy are considered segmentation points. Other examples include [66, 82].

Perhaps the most prevalent technique used for segmentation is the sliding window, which continuously evaluates a fixed-size subset of the most recent input. Windowing is attractive because of its simplicity, as one only needs to keep a small buffer of history that is continuously fed to an underlying recognizer. Sliding windows have been used in data glove [48], body-worn inertial sensors [34], and vision-based body and tracking [69] gesture recognition systems. One problem with windowing, however, is that one must decide on an appropriate window size. This issue can be resolved by learning appropriate window sizes with training data [48] or by using multiple windows at the expense of greater computation.

Keogh et al. [38] presented an overview of several techniques for time series segmentation, including windowing, top-down, and bottom up. Top-down segmentation processes an entire time series at once and splits it repeatedly based on a given measure (such as energy) until either it reaches a set number of segments or no further splits are possible. The bottom-up technique conversely groups sequences together that have similar properties until a certain criteria is met. Both top-down and bottom-up are traditionally offline approaches to segmentation, which limits their utility compared to sliding windows, despite being generally more accurate. Keogh et al., however, propose the sliding window after bottom-up (SWAB) technique, which builds small segments that may be merged in the future.

Different from most other direct approaches, Vatavu et al. [80] proposed a hierarchical recognition approach based on the integral absolute curvature of two-dimensional (2D) gestures, which they proved to be scale invariant. With a modest amount of training data, one can determine a representative band of curvatures and quickly reject unfit segments.

Simple, heuristic (rule-based) segmentation is also common, whereby a system uses easy to detect criteria in order to delineate gestures from other user activities. Using a Kinect, Kristensson et al. [43] defined an “input zone” for hand gestures where users could interact with their application. Movement into and out of the zone therefore segmented the continuous input stream. Gesture Watch [40] instead uses a hardware sensor to detect when one raises their wrist, at which time the system begins to record input from other sensors until he or she again lowers their wrist, thereby trigging recognition.

3.2 Indirect Methods

Indirect methods are like mini-recognizers that score continuous data until a possible gesture boundary is detected, many of which require extensive training data. One example is Alon et al.’s spatiotemporal HMM recognizer [1] as well as Yin and Davis’s three part HMM [94]. This latter approach divides a gesture into three parts, the pre-stroke (setup), nucleus (main activity), and post-stroke, and learns models for each. Other examples include the use of artificial neural networks [53] for hand gestures and conditional random fields for sign language [92]. Yin and Davis [94] trained an HMM to detect hand gestures from salience maps extracted from RGB-D images. Wang et al. [84] make use of an HMM for segmenting atomic gestures from streams of human actions in recorded videos.

For a more recent example, consider uDeepGRU [11], which took first place in the 2019 3D shape retrieval contest (SHREC’19) for continuous hand gesture recognition. uDeepGRU is an end-to-end deep neural network designed for recognizing gestures in an unsegmented stream of data. With gated recurrent units [18] as the main building block, uDeepGRU takes as input per-frame raw direction vector features and outputs class conditional probabilities. Feature extraction and implicit segmentation are performed jointly without relying on the data of future time steps. These properties make uDeepGRU suitable for online recognition of gestures, but nontrivial to understand and implement.

Conversely, one approach suitable of custom gesture recognition is CDP [58], which relaxes a DTW constraint that makes continuous recognition possible. Similarly, Sakurai et al. [65] use a modified DTW algorithm, called SPRING, to continuously segment single or multidimensional data streams. The two modifications they proposed were star-padding, which reduces the number of matrices one needs to calculate per candidate template, and sub-sequence time warping, which adds additional information to each individual cell of the DTW matrix. These two modifications dramatically reduce spatial and temporal complexity of the problem of time series matching. In our view, SPRING and CDP are very similar.

Most similar to our work, Tang et al. [70] proposed Structured Dynamic Time Warping (SDTW) for hand gestures, where like Machete, they utilize direction vectors between consecutive points to segment input. Specifically, their local cost function measures the exact angle between model and input direction vectors, rather than the squared ED between point data. When the cumulative score reaches a predetermined threshold, SDTW then localizes gesture end points based on a sliding window SVM over position and velocity data. While direction vectors are a corner stone of our work, we propose a unique local cost function and initial boundary condition that makes Machete viable for customization, whereas SDTW requires sufficient training data to learn an SVM model. Further, we designed Machete for a variety of input, not just hand gestures.

3.2.1 Accept-Reject Criteria. A continuous gesture recognizer evaluates input as it arrives, and for each new sample, the recognizer must decide whether or not to inform the system of a possible gesticulation. This decision comprises the *accept-reject criteria*—the set of conditions one must satisfy in order to send notification. Typically, scores that fall below a per class *rejection threshold* are discarded, whereas sufficiently high scores result in further analysis. What happens thereafter is nuanced, application specific, and rarely discussed in the literature, despite having a significant impact on recognition accuracy. When descriptions are provided, they often lack detail, appear ad-hoc, or seem to fit the data at hand [36, 90, 94]. To illustrate with a few examples, some approaches require that a duration of time passes between subsequent detections as a way to remove unintended non-gesture movements and potential noise [10, 59]. Similarly, a recognizer may require that an evolving pattern scores well over multiple consecutive frames prior to acceptance, so as to circumvent arbitrary spikes from noisy data [35]. Recognizers may further purge [1, 9] or keep [8] prior data and internal state information upon detection, which impacts subsequent signal processing. One may use a low sampling rate to stabilize results or interpolate across multiple frames [9]. In this work, we do not propose an accept-reject criteria per se, but we do describe a related pruning technique that eliminates most candidate gestures.

3.3 Putting It All Together

We are interested in finding a fast and accurate segmentation approach suitable for gesture customization that also conforms to $-family standards. After surveying the state-of-the-art, we found three candidate approaches. Standard energy-based segmentation showed promise because of computational efficiency; however, there are at least two shortcomings. First, boundaries are not gesture-specific, which implies we have to evaluate all templates for each boundary pair. Second, energy alone does not appear to be a reliable boundary indicator and one may have to learn additional criteria from training data to use this direct method.

Windowing also shows promise, having a long history of success. It is simple, direct, and we expect windowing to do well when temporal variability is low given that prototype-based window sizes will precisely fit incoming data. Otherwise, with greater variance, the optimal size has to be learned, multiple windows have to be used, or both. The most critical issue we see is that there is also no potential for pruning, which means that we must run a full evaluation every frame and incur high computational costs.

We also considered SWAB [38] and the integral absolute curvature approach [80]. We were concerned that SWAB would require domain-specific knowledge and extensive tuning to use effectively, or suffer from the same issues that plague other direct methods. For this reason, we decided to leave SWAB for future work. On the other hand, the integral absolute curvature approach requires at least a small training set size to learn valid curvature intervals. We require that our segmenter works well with a single training example per gesture class (one-shot learning).

Our final choice is CDP, which is fast, and in our opinion, $-family friendly. However, CDP suffers from several issues that we address. Specifically, we introduce a new local cost function that is time, scale, and position invariant. Instead of measuring ED between corresponding points normalized by the warping path length, we measure the *weighted* squared inner product between corresponding direction vectors normalized by *gesture path length*. To ensure proper segmentation with our new local cost function, we also introduce a sink node cost that controls how the first element of two sequences are matched. We further propose a simple automatic pruning technique that eliminates the majority of segmented gesture candidates. Finally, we improve DP resampling by including angular information in its point selection logic, which leads to better gesture representations and templates. The combination of these new techniques is collectively called Machette.

4 MACHETE

We depict a typical custom gesture recognition pipeline that incorporates Machete in Figure 1 below. An input device periodically measures its sensors at a fixed sampling rate f_s and outputs a

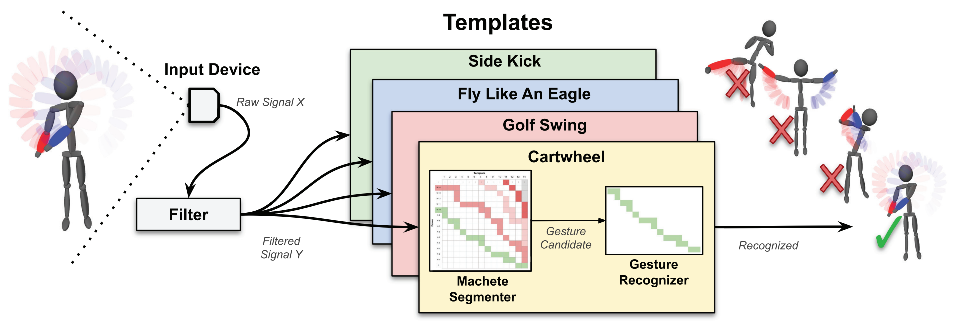

Fig. 1. Gesture recognition pipeline for continuous input. An input device samples human motion and outputs noisy data that we clean with a low-pass filter. For each example gesture provided by the user during training, Machete uses a CDP based technique to find gesture candidates in the continuous input. In this illustration, matrix rows represent frames and columns represent template elements that must be matched. Identified candidates are thereafter passed to a gesture recognizer for further analysis. If the candidate satisfies the recognizer's critiera, the application is informed a gesture is recognized.

discrete signal X. This signal contains high-frequency noise and other corruptions that the system removes with a low-pass filter so as to let through the most informative frequencies, i.e., those that embed human motion. Filtered signal Y is then fed into the main recognition subsystem where we measure its likeness to each gesture class. Specifically, Machete first determines on a per frame basis how well the incoming filtered data matches a given class template suitable for segmentation, but not recognition. When its dissimilarity score is sufficiently low, Machete localizes the end points and passes the segmented gesture candidate $Y[start : end]$ to an underlying robust recognizer for further analysis. The recognizer processes the candidate and generates a new score that is more reliable than Machete's initial estimate. With the recognizer's output in hand, an accept-reject machine analyzes the current state-of-affairs, e.g., context in combination with other ongoing recognition results, to decide whether the sequence is without meaning or that it indeed maps to the specified gesture class.

In this section, we describe Machete in detail. We start with preprocessing but quickly proceed to DTW so as to gain an intuitive understanding of how its elastic matching process works. Thereafter, we turn to CDP [58], which generalizes DTW by relaxing one constraint that thereby enables continuous processing. In our CDP discussion, we introduce three novel modifications that facilitate gesture segmentation. We thereafter discuss the DP line simplification algorithm [21] and our modification thereto, which we use to convert training samples into Machete templates. Last, we describe our strategy for pruning most candidate gestures that boosts overall performance. Note, pseudocode for all methods discussed in this section can be found in Appendix A.

4.1 Preprocessing

Machete works on direction vectors, which means we must first transform input signal X into a series of deltas as follows:

$$\vec{X} = \left(\vec{\boldsymbol{x}}_i = \boldsymbol{x}_{i+1} - \boldsymbol{x}_i \,\middle|\, i = 1 \ldots X_N - 1\right). \tag{1}$$

During this transformation, we discard new input samples that would produce invalid vectors. For instance, we discard samples that are less than one pixel away from the last accepted position when working with mouse data. It is also critical that one smooths the input signal with a low-pass filter so as to remove jitter and variabilities in the direction vector signal. Small differences in position over time can result in significant fluctuations between consecutive direction vectors. And

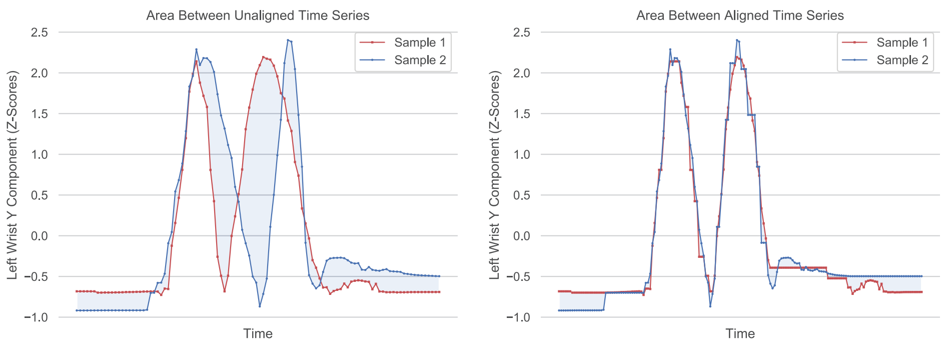

Fig. 2. Visualization of two time series before and after DTW alignment. The shaded regions between them represent the amount of dissimilarity measured by ED. Although the unaligned series are of a shorter duration, the area between them after alignment is significantly improved.

although we account for this issue via a per input sample weighting scheme, filtering nonetheless further improves performance.

4.1.1 Training. Given demonstration sample X belonging to gesture class $g_i \in \mathbb{G}$, we must convert the sample into a Machete template. To do this, we first find a suitable low resolution representation of X using a modified DP line simplification algorithm, after which we then define template T as the normalized direction vectors between DP points Y:

$$T = \left(\vec{t}_i = \frac{\boldsymbol{y}_{i+1} - \boldsymbol{y}_i}{||\boldsymbol{y}_{i+1} - \boldsymbol{y}_i||} \middle| i = 1 \ldots Y_N - 1 \right). \tag{2}$$

4.2 Dynamic Time Warping

DTW is an elastic matching dissimilarity measure that finds the optimal alignment between two sequences based on a given local cost function. Although squared ED is the most common cost function, Jackknife [74] utilizes the inner product of gesture path direction vectors, which provides translation and scale invariance. We adopt this approach herein not just because of its simplicity, but because we are unaware of any alternative computationally efficient data representation or technique that can handle position and scale variance on a per time-step basis.

To understand how DTW works at a high level, consider Figure 2 above. We present two sequences produced by the same individual who twice in a row raises their arm up high. In the left graph, notice that the sequences are temporally out of sync because of differences in production speed. The shaded area between each sequence measures the squared ED, i.e., $\sum (x_i - y_i)^2$ between them, which reveals major dissimilarities. Contrast this with DTW which is able to resolve temporal differences as shown in the right graph, where we see that DTW eliminates most of the shaded area. How does this work? Internally, DTW constructs an optimal *warping path* W that specifies how each element from query Q maps to template T. Warping path W is defined as:

$$W = \left(\boldsymbol{w}_i = (a, b) \middle| 1 \leq a \leq |Q|, \;\; 1 \leq b \leq |T| \right), \tag{3}$$

and is used by DTW in the following way:

$$DTW(Q, T) = \sum_{i=1}^{|W|} d(\boldsymbol{q}_a, \boldsymbol{t}_b) \tag{4}$$
$$a = \boldsymbol{w}_{i_a}$$
$$b = \boldsymbol{w}_{i_b}$$

where $d(\boldsymbol{q}_a, \boldsymbol{t}_b)$ is a local cost function that measures the dissimilarity between query element $\boldsymbol{q}_a$ and template element $\boldsymbol{t}_b$.

Next we need to understand how DTW constructs the warping path. Let us suppose we have partial warping path $W[1 : N]$ and the last element therein $\boldsymbol{w}_N = (i,\ j)$ maps query $\boldsymbol{q}_i$ to template $\boldsymbol{t}_j$. From here, we have three options for how we can proceed—we can set:

- $\boldsymbol{w}_{N+1} = (i,\ j+1)$ Repeat query $\boldsymbol{q}_i$ so that it is matched with template $\boldsymbol{t}_{j+1}$
- $\boldsymbol{w}_{N+1} = (i+1,\ j)$ Repeat template $\boldsymbol{t}_j$ so that it is matched with query $\boldsymbol{q}_{i+1}$
- $\boldsymbol{w}_{N+1} = (i+1,\ j+1)$ Repeat neither and match query $\boldsymbol{q}_{i+1}$ with template $\boldsymbol{t}_{j+1}$

It is by repeating elements in either sequence that DTW is able to align the sequences. Referring back to Figure 2 (right), one can see example short plateaus in the aligned curves where elements are repeated.

One might note that there are many possible warping paths that align two sequences, but DTW selects the optimal solution via an elegant dynamic programming approach. DTW constructs a $|Q| \times |T|$ matrix M, where each element (i, j) holds the optimal warping path score that aligns $Q[1 : i]$ with $T[1 : j]$. In other words, once two subsequences are aligned, element (i, j) is the measure between them. The matching decision described above can then be decided using the following recurrence relation:

$$M_{i,j} = d(\boldsymbol{q}_i,\ \boldsymbol{t}_j) + min\begin{cases} M_{i,j-1} & \text{Repeat query } \boldsymbol{q}_i \\ M_{i-1,j} & \text{Repeat template } \boldsymbol{t}_j \\ M_{i-1,j-1} & \text{Repeat neither,} \end{cases} \tag{5}$$

with boundary conditions:

$$M_{0,0} = 0 \tag{6}$$

$$M_{i,0} = \infty \tag{7}$$

$$M_{0,j} = \infty. \tag{8}$$

In code, one can fill in the matrix one row at a time from left to right, starting at $(1, 1)$. Then for each subsequent element (i, j), one evaluates Equation (5). Once this process reaches the bottom right corner $(|Q|, |T|)$, the optimal path (and score) through the matrix is known. Two important DTW properties worth noting are that every warping path through M forces $\boldsymbol{q}_1$ to be matched with $\boldsymbol{t}_1$ (as a consequence of the boundary conditions), and similarly $\boldsymbol{q}_{|Q|}$ is matched with $\boldsymbol{t}_{|T|}$.

Pathological warping occurs when a small section of one sequence is aligned with a large section of a second sequence due to unbound warping [61]. To prevent pathological warping, one may impose a warping path constraint, such as a Sakoe–Chiba band [64]. One may also use lower bounding [39, 74, 96] along with other techniques [60] to optimize the alignment process, but these optimizations do not apply to this work.

4.3 Continuous Dynamic Programming

CDP [58] extends DTW so as to be able to match patterns embedded in unsegmented sequences, including gestures embedded in continuous signals. Specifically, CDP replaces the initial matching constraint imposed on DTW by Equation (7) with a new condition $M_{i,0} = 0$. In this way, CDP introduces the start of a new warping path every frame that extends on through matrix M only when its cumulative distance is sufficiently low. We present an example of this process in Figure 3. When the system samples a new Kinect pose $\boldsymbol{q}_i$ at time i, CDP establishes the start of a new path by matching pose $\boldsymbol{q}_i$ with template $\boldsymbol{t}_1$. CDP then processes the remainder of the row as standard DTW would. At the end of the update, $M_{i,|T|}$ holds the optimal warping path score ending at time i, and by monitoring this score we can speculate about whether the user gesticulated. We illustrate

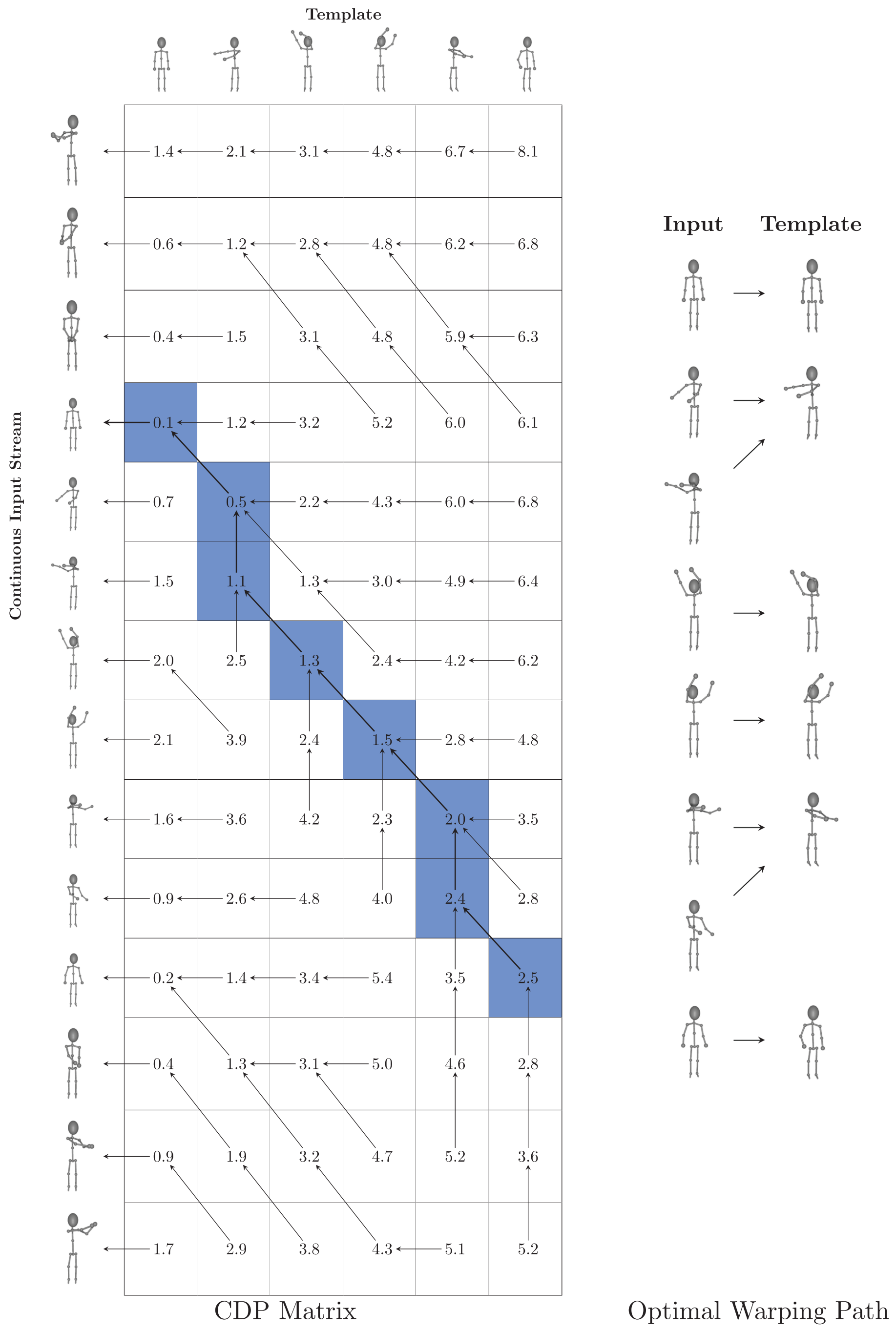

Fig. 3. Visualization of a CDP matrix. Each row is the result of processing one new input sample from a continuous stream, and each element holds the accumulated warping path score. Arrows indicate the direction of each warping path through the matrix. The blue warping path highlights the best result, showing which input poses where matched with which template poses.

this idea in the previous figure where we highlight the optimal warping path associated with the best score found in the last column and visualize the eight matched elements. Our system after having found a sufficiently low score can extract the subsequence and pass it to the underlying recognizer for further analysis.

Since DTW has been used in prior work that meets our design criteria [47, 74], and CDP is a straightforward extension of DTW as just described, we decided to use CDP in Machete. However, standard CDP suffers from scale and position invariance under the ED measure, which we address in our work. Specifically, in the following subsections we (1) develop a new local cost function, (2) define CDP_{ip}, and (3) introduce a new initial matching constraint.

4.3.1 Local Cost Function. The squared ED local cost function one typically employs in DTW and CDP is inappropriate for several continuous gesture recognition problems. Namely, squared ED is neither position nor scale invariant. One way around this issue is to use a running z-score normalization scheme [60], but finding an appropriate lag in the presence of large discontinuities (such as when a person jumps to a new location on an interactive display) may prove challenging. Further, we want to avoid scaling noise to unit variance which can happen when idle components do not change position during gesticulation, e.g., the y-component in a minus sign. A simpler approach is to measure the inner product between direction vectors, which is invariant to position and scale and has already been used successfully to recognize gestures across varying input device types [74, 76]. We specifically leverage the inner product between direction vectors as follows:

$$d\left(\vec{q}_i, \vec{t}_j\right) = \left(1 - \frac{\left\langle \vec{q}_i, \vec{t}_j \right\rangle}{||\vec{q}_i||}\right)^2. \tag{9}$$

In words, the inner product of normalized direction vectors fall in the interval $[-1, 1]$. Since template elements are already normalized, we only need to divide $< \vec{q}_i, \vec{t}_j >$ by the query's vector length $|\vec{q}_i|$. To transform this product into a dissimilarly measure, we subtract it from one, and as a consequence, the score falls in the interval $[0, 2]$. We further square the sum to allow for a greater variability in low angular differences, and to further penalize severely diverging trajectories. This difference can be seen in Figure 4. Certainly one can use other powers, but squaring the sum is computationally efficient as it does not involve an expensive power function call, and we found that cubing the sum did not improve performance.

4.3.2 CDP with Direction Vectors. Given Equation (9) as the local cost function, we define CDP_{ip} as the *weighted* sum of the optimal warping path through M, which corresponds to the recurrence blow. Note, our unique contributions are, first, how we weight the local cost function with $||\vec{q}_i||$ and, second, how we track and normalize the result with path length $L_{i,j}$:

$$M_{i,j} = \frac{1}{L_{i,j}} \left[||\vec{q}_i|| \cdot d\left(\vec{q}_i, \vec{t}_j\right) + L_{k,l} \cdot min \begin{cases} M_{i,j-1} \\ M_{i-1,j} \\ M_{i-1,j-1} \end{cases} \right] \tag{10}$$

$$L_{i,j} = L_{k,l} + ||\vec{q}_i||. \tag{11}$$

L is an auxiliary matrix that stores the sum of query vector lengths along each warping path. And indices (k, l) correspond to the selected matrix element $M_{k,l}$ (being one of $M_{i,j-1}$, $M_{i-1,j}$ or $M_{i-1,j-1}$). The effect of Equation (10) on the overall cost is that the contribution of each warping path element is weighted by the query vector's length. This treatment ensures that short vectors occurring on cusps and corners have less impact on the final score than do vectors that define the main body of motion. It is worth noting that we informally tested CDP_{ip} without the normalization factor and saw a significant drop in performance.

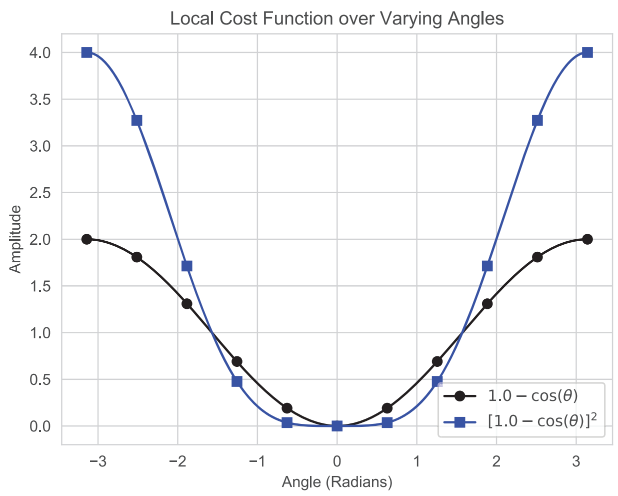

Fig. 4. Amplitude of unweighted local cost functions over varying angles. The squared variant ensures that a wider variability of similar angles are mapped to lower costs, whereas dissimilar angles are more severely penalized.

We designed our warping path formula with $-family principles in mind. The local cost function is weighted by the input direction vector length, a simple geometric operation that has been used in prior work [72, 74, 76]; and we track the full gesture path length through the warping path, which is merely an accumulator. Similarly, in code, it is trivial to define a warping path object that stores the start frame, unnormalized weighted cumulative score, and cumulative path length. In this way, one can calculate $M_{i,j}$ on the fly by normalizing the score and extend a warping path by propagating these values to the next element, as shown in our pseudocode in Appendix A.

4.3.3 Boundary Conditions. One issue with matching direction vectors is that once a gesture begins, a subsequent vector may better match the first template element. When this happens, the new warping path will overtake the prior path rather than extend it. Our solution to overcome this problem is to modify the initial matching condition so that CDP_{ip} only starts a new warping path when the present path passing through the first column is worse than a threshold θ. Formerly, CDP_{ip}'s boundaries conditions are:

$$\begin{aligned} M_{0,0} &= 0 & L_{0,0} &= 0 \\ M_{i,0} &= (1 - cos(\theta))^2 & L_{i,0} &= 0 \\ M_{0,j} &= \infty & L_{0,j} &= 1 \end{aligned}$$

Note that because $L_{i,0} = 0$, the transition from column zero to one, which starts a new warping path, carries zero cost. With this modification, vectors matching the first template element are now grouped together. In Section 5 we discuss how we select the θ parameter in further detail.

4.4 Correction Factors

One limitation of Machete is that we are unable to localize gesture end points that occur within a straight line. For full body gestures, this issue is less problematic, but straight lines are common for 2D gestures. Our sink node weighting scheme enables Machete to latch onto the start point when the gesture boundary occurs on a curve, but we also need a similar mechanism for end points. To address end point localization, we adopt the correction factors concept [6, 74].

$$CDP'_{ip} = CDP_{ip} \times cf_{openness} \times cf_{f2l}. \tag{12}$$

The adjusted CDP'_{ip} score is the standard score inflated by two correction factors that measure dissimilarity information not captured by CDP_{ip}. Specifically, we look at the relationship between the first and last points of a gesture, $cf_{openness}$. When they are far apart, the gesture is open, and conversely, the gesture is closed when these points are collocated. Second, we also consider the angle made by the direction vector between them, cf_{f2l}. Since the direction vector may be unreliable when a gesture is closed but provide useful information about correct gesture production when open, we use a weighting scheme to balance these two factors, which we describe shortly. A correction factor resolves to one (zero inflation) when there is no difference between the query and template, and differences between them cause the factors to scale upward, though we limit each factor's maximum value to two (not shown in the math below).

We limit the design of our new correction factors to the same set of operations used in prior rapid prototyping work [72, 74, 76] to ensure compliance with our design goals; namely, we use scalars, vectors, ratios, weighting, min-max functions, and inner products. In greater detail, let us define the first to last point (abbreviated $f2l$) direction vector as follows:

$$\vec{\boldsymbol{x}}_{f2l} = \boldsymbol{x}_l - \boldsymbol{x}_f, \tag{13}$$

where f and l, respectively, index the gesture's start and end points. From $\vec{\boldsymbol{x}}_{f2l}$, we derive the openness of a gesture as the length of its direction vector over the gesture's path length:

$$openness = \frac{\|\vec{\boldsymbol{x}}_{f2l}\|}{\mathcal{L}}. \tag{14}$$

Here, $\mathcal{L}$ is the total path length $X[f:l]$, which we measure directly from training data or estimate with $L_{l,|T|}$ for queries. We use these values because of their fast computation—one can index the points from a circular buffer and $L_{l,|T|}$ is readily available. Intuitively, from Equation (14), we see a five point star gesture will score low in openness, whereas a gesture separated by the length of its boundary will score highest. With training sample's $\vec{\boldsymbol{t}}_{f2l}$ direction vector, we also decide how much to weight the $cf_{openness}$ and cf_{f2l} factors. The direction vector length relative to the gesture's bounding box diagonal length $diag$ gives us a sense of the gesture's openness, where a value close to one means the gesture is fully open. Note, we cannot use the diagonal length in the above definition of openness because we found the per component boundaries were too expensive to carry forward through time. However, to calculate the bounding box during training is okay. With this in mind, we calculate the weights with:

$$w_{f2l} = \min\left(1,\ 2\frac{\|\vec{\boldsymbol{t}}_{f2l}\|}{diag}\right) \tag{15}$$

$$w_{closedness} = 1 - \frac{\|\vec{\boldsymbol{t}}_{f2l}\|}{diag}. \tag{16}$$

By these definitions, we give preference to w_{f2l} when the gesture is more open and $w_{closedness}$ when the gesture is closed.

With the direction vectors, openness, and weights, we last define the correction factors as follows:

$$cf_{f2l} = 1 + w_{f2l}\frac{1}{2}\left(1.0 - \left\langle \frac{\vec{\boldsymbol{q}}_{f2l}}{\|\vec{\boldsymbol{q}}_{f2l}\|}, \frac{\vec{\boldsymbol{t}}_{f2l}}{\|\vec{\boldsymbol{t}}_{f2l}\|}\right\rangle\right) \tag{17}$$

$$cf_{openness} = 1 + w_{closedness}\left(\frac{\max\left(openness(Q), openness(T)\right)}{\min\left(openness(Q), openness(T)\right)} - 1.0\right). \tag{18}$$

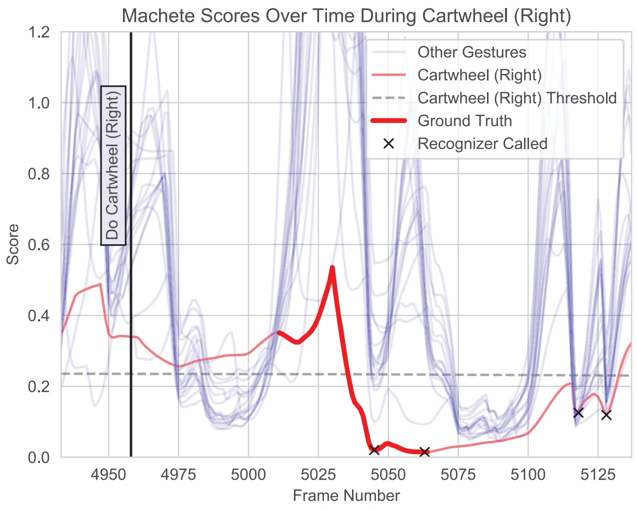

Fig. 5. Example Machete segmentation scores over time as a participant performs the cartwheel right gesture. A total of 17 templates are loaded. Per frame Machete scores are rendered red for the cartwheel gesture and blue for the remaining sixteen gestures. Ground truth segmentation is in bold red. Based on our pruning strategy, the four minimum values below the template threshold (horizontal dashed black line) are where candidate gestures are passed from Machete to an underlying recognizer for further evaluation. Notice that in approximately 200 frames, the recognizer is invoked only four times to analyze the cartwheel gesture.

Without weighting, the cf_{f2l} factor is manipulated so that it evaluates to one when the query and template direction vectors match, whereas cf_{f2l} evaluates to two when they perfectly differ. The fraction in $cf_{openess}$ is a ratio greater than one that describes the difference in openness. Similarly, $cf_{openness}$ is one when their openness matches, but scales to higher values as their openness diverges.

These correction factors together help compensate for situations where gestures end on a long line. In our evaluation, we only apply these factors to mouse data, where we have several such gestures. For full body interactions, however, we do not leverage this mechanism, and it is a goal of our future work to derive a more general solution.

4.5 Pruning

Machete generates a new gesture candidate every frame, but in order to reduce the computational burden of calling an underlying recognizer too frequently, we wish to prune as many candidates as possible. Figure 5 illustrates how several CDP_{ip} template scores progress over time during a cartwheel right gesture. We observe that most (though not all) curves stay relatively high while the user swings their arms, whereas the cartwheel template score drops close to zero. We anticipate this drop for two reasons. First, we expect well defined gestures to stand out from other human motion, which implies that the input will diverge from most template patterns at least enough to drive the score high on average. Second, because we square the inner product, sufficiently similar patterns ought to approach zero. Based on these assumptions, we are able to derive a relatively efficient pruning strategy.

For each template, we track the running average and automatically reject any candidate that is above half the mean. We believe this is a reasonable conservative estimate given that scores are expected to approach zero during gesticulation. To further improve pruning, we also assume that gesture end points correlate with minima. Based on this idea, we also prune any candidate that does

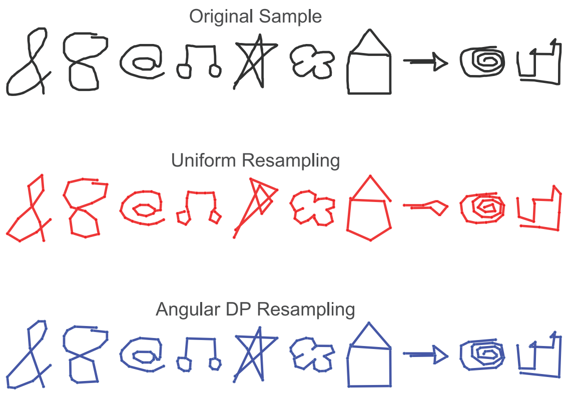

Fig. 6. Comparison of resampling techniques. Each original sample is resampled using angular DP and then uniformly resampled using the same number of points.

not occur on a minimum. This rule results in the automatic rejection of most remaining candidates. As shown in Figure 5, only a few candidate gestures satisfy these requirements, whereas the vast majority are discarded.

4.6 Angular DP

Our final point of order is to describe how Machete transforms a given training sample into a template. Many recognizers uniformly resample the sample's trajectory to a fixed number of points that are equidistantly spaced along its arc length. However, uniform resampling may cut corners, express the gesture's shape with too many points, or both. Instead, we hope to reduce computation by representing a pattern with the least number of points possible, but while preserving important features. One technique that does both is the DP line simplification algorithm [21], which has been successfully used in gesture recognition [26].

The algorithm proceeds as follows. Given line segment $\overline{\boldsymbol{x}_a\boldsymbol{x}_b}$, we calculate the distance of each point $\boldsymbol{x}_i, a < i < b$ to the segment. We then select the furtherest point $\boldsymbol{x}_i$ from the segment and subdivide it into two halves: $\overline{\boldsymbol{x}_a\boldsymbol{x}_i}$ and $\overline{\boldsymbol{x}_i\boldsymbol{x}_b}$. DP recurses into both halves and continues on as such until either the furthest distance falls below a predetermined threshold ϵ or the segment is indivisible. All selected points are then combined to describe a simplified version of the original trajectory. This approach is generally effective, but because Machete works on direction vectors, we make a small modification. Instead of using distance directly, we weight it by the normalized angle it forms with the end points:

$$d'_i = d_i \cdot \arccos\left(\frac{\langle \vec{\boldsymbol{v}}_1, \vec{\boldsymbol{v}}_2 \rangle}{\|\vec{\boldsymbol{v}}_1\| \cdot \|\vec{\boldsymbol{v}}_2\|}\right)\frac{1}{\pi} \tag{19}$$

$$\vec{\boldsymbol{v}}_1 = \boldsymbol{x}_i - \boldsymbol{x}_a \tag{20}$$

$$\vec{\boldsymbol{v}}_2 = \boldsymbol{x}_b - \boldsymbol{x}_i, \tag{21}$$

where d_i is the distance from $\boldsymbol{x}_i$ to $\overline{\boldsymbol{x}_a\boldsymbol{x}_b}$ and d'_i is the scaled distance. Please see our pseudocode in Appendix A for more information.

We chose to weight the distance of a point by its associated angle in order to help DP select those points where the trajectory changes direction, e.g., on corners and cusps. We believed this would lead to better simplified gesture representations using fewer points, which we later confirm. For now, we illustrate the impact of Angular DP on resampling in Figure 6. For each sample, we first resample the trajectory using Angular DP with $\epsilon = 0.01$. We then uniformly resample the original trajectory to the same number of points selected by Angular DP. By inspecting their shapes side by side, one will notice that angular DP provides a better description of each gesture shape, often using relatively few points.

5 PARAMETER SELECTION

Our system requires three parameters, a low-pass filter cutoff frequency f_c, an angular DP termination threshold ϵ (Section 4.6), and an initial sink node cost θ (Section 4.3.3). We turn to the former first, but before we can select a cutoff frequency, we must choose a low-pass filter. There are several options compatible with $-family principles, including the moving average, exponential moving average, double exponential moving average [45], and 1€ [13] filters. We decided to use a multiple-pass centered moving average (MCMA) because of its simplicity, optimal properties [68], and because we wanted the filtered response to correspond directly to the original input to facilitate analysis with our development tools. The central moving average (CMA) filter has a single parameter M, the number of points per average, which we set to three:

$$\boldsymbol{y}_i = \frac{\boldsymbol{x}_{i-1} + \boldsymbol{x}_i + \boldsymbol{x}_{i+1}}{3}, \tag{22}$$

where signal Y is the filtered response over input signal X. MCMA builds on this by passing the signal through a CMA filter R times. Algebraic manipulation of basic MCMA facts using our window size leads to:

$$R = \frac{3}{2}\left(\frac{f_s}{2\pi f_c}\right)^2. \tag{23}$$

The sampling rate f_s can be determined from training data or at runtime, but the cutoff frequency f_c is less trivial. In general, most human motion falls below 10 Hz [95], but this varies over different types of motion. Although there are automatic low-pass filter calibration procedures for pointing tasks [71] which balance precision and lag, we are unaware of similar procedures for gestures. So to help us decide appropriate cutoff values, we examine two public datasets that are representative of the gesture types used in our evaluation. The first is a Kinect dataset [23] comprising sixteen full body parkour gestures collected from sixteen participants, totaling 1,280 samples. The second, $1-GDS [89], is a pen and touch gesture dataset comprising 16 gestures articulated at 3 different speeds by ten participants, totaling 4,800 samples.

We conducted power spectral density (PSD) analysis on each sample and combined their cumulative powers into a single distribution. These results are shown in Figure 7. Our analysis reveals that information embedded in full body gestures fall below 3 Hz, which we select as the cutoff frequency f_c for our Kinect and Vive input described later. For pen and touch gestures, we see that most information falls below 5 Hz, which we select as the cutoff frequency f_c for mouse input. In addition to PSD data, we also illustrate MCMA's frequency response for each cutoff, which shows the reduction in signal strength over varying frequencies. Specifically, this shows what ratio of each frequency will remain in the signal after being filtered. The strength of low frequencies are left mostly unaltered, whereas high frequencies are almost entirely removed. Recall that our main objective with filtering is to remove high frequency noise that jitters the direction vectors without distorting the gesture's trajectory. In this regard, one will note that MCMA has a slow rolloff; so even if the cutoff is not optimal, there is still sufficient wiggle room and the gesture shape will largely remain in tact.

Next we must determine an appropriate threshold ϵ that balances precision and computation. Our goal is then to find ϵ that will adequately describe complex trajectories. To do this, we compare the intraclass and interclass score distributions of a sufficiently complex dataset. Specifically, for

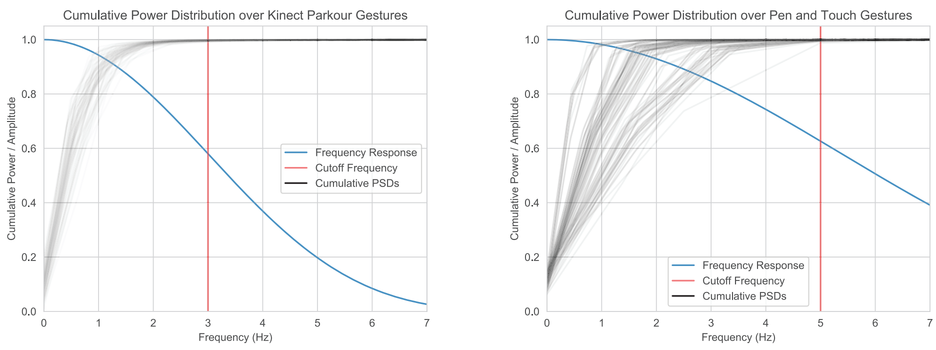

Fig. 7. Cumulative power spectral distribution for parkour Kinect (left) as well as pen and touch (right) gesture signals. Vertical red lines denotes our proposed cutoff frequency for full body and hand gestures, respectively.

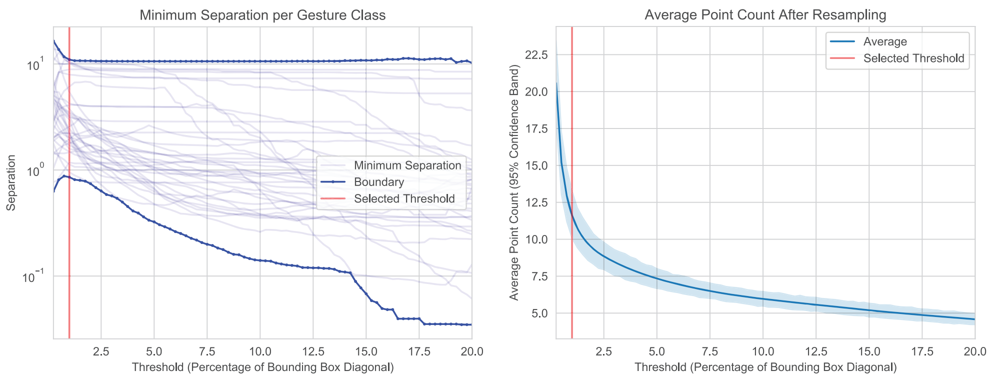

Fig. 8. *Left*: Minimum separation between each EDS [81] gesture class and its nearest neighbor as measured by Machete over varying ϵ thresholds. *Right*: Average resampling rate N per threshold.

a given dataset and threshold ϵ, we compute the pairwise dissimilarity of each sample to every other sample in the dataset using Machete—between two samples, one is used as a template and the other is treated as continuous input. For each class g_1 and g_2, we find g_1's intraclass mean score and the interclass mean score from g_1 to g_2, and then we save their ratio. A ratio greater than one means the classes are perfectly separated, whereas a ratio less than one indicates there may be confusion between the classes under Machete, and to have the best separation possible, we select threshold ϵ that maximizes the ratio between the closest classes.

We carried out our analysis over two pen datasets combined into one whole [81]. The Execution Difficulty Sets (EDS) were created to model and test the perceived difficulty of articulating unistroke pen gestures. In aggregate, there were 38 gestures articulated 20 times, with 18 gestures being produced by 14 participants and the remaining 20 by 11, for a grand total of 9,440 samples. We chose to use EDS because compared to other options, they varied in complexity. With a smaller dataset, we run the risk of selecting a poor threshold because classes are easily separated and not representative of all motions possible in its associated input space.

Using the described procedure on EDS, we collected the minimum separation ratio between classes over varying thresholds. In Figure 8, we plot the result for each of the 36 classes as well as the average resampling rate N for each threshold. We find that the best separation occurs at $\epsilon = 0.75\%$, which we round up to 1% since the difference is not substantial. At this level, we also

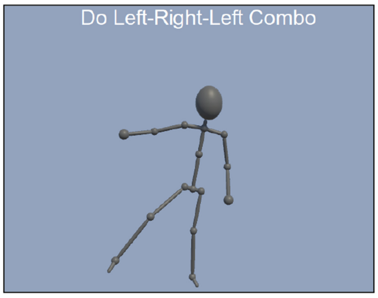

Fig. 9. An example FTL screenshot: The leader and participant are standing on one foot, as if to maintain balance on a tightrope, when a gesture command pops up. At this time, the participant will immediately stop following and execute the command, after which they will return to following the leader.

note that the resampling rate is low compared to typical values used by \$-family recognizers, e.g., \$Q recommends using $N = 32$. Although lower rates are possible, we see that the separation between classes also quickly drops, and so we are compelled to remain at $\epsilon = 1\%$.

With a ϵ threshold in hand, we were able to next decide on the initial sink node cost θ. For this parameter, we simply resampled all of the samples of a given dataset using angular DP, and thereafter measured the pairwise angles between each sample within the same class. We again used EDS for hand gestures, and analysis revealed the average angle to be $\mu = 20.8$ ($\sigma = 32.4$), which we rounded to 20 degrees. Informal testing on our mouse dataset revealed this was a good choice as θ values outside this locality produces worse results. For fully body gestures, using the same procedure over the Kinect parkour dataset, we found that θ jumped to nearly $\mu = 65.02$ ($\sigma = 27.0$) degrees. Informal testing revealing that although results were reasonable, this threshold was more liberal than necessary. When we lowered the parameter to $\theta = 40$ degrees, performance slightly improved and so we used this value throughout our evaluation for full body gestures.

6 DATA COLLECTION

To evaluate the effectiveness of Machete against alternative techniques, we collected HA data from 30 participants over 3 input device types (10 participants per device).

6.1 Data Collection

We designed Machete to spot custom gestures among other HA interactions. However, because most public datasets were captured with a single device and comprise presegmented sequences, low-activity interactions, or both, we decided to collect four new HA datasets using three unique input device types via a game-based data collection protocol. Our goal was to ensure that participants engaged in a range of activities that vary beyond just gestures, such as to puppeteer an avatar or directly manipulate objects in a virtual environment. To this end, we designed a simple "Follow The Leader" (FTL) game that forces players to remain active between gesticulations in a way that is consistent with our objective. Through the interface, we present a continuously active *leader* who performs various random movements that participants must mimic to their best ability. At random times throughout the game, we further present text commands that require a player to break solidarity and gesticulate, see Figure 9. This strategy results in a rich stream of complex, high-activity motions that (unlike a typical one-at-a-time data collection protocol) captures participants easing into and out of gesticulations from other activities. Another reason we chose this approach is because gesture productions may differ when a participant's attention is divided. For instance, several studies show a significant difference in recognizer accuracy when

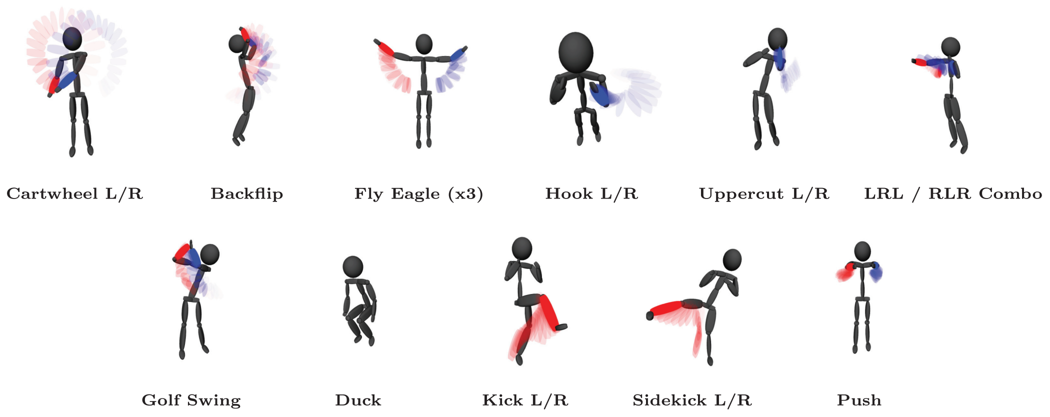

Fig. 10. High activity gesture dataset for Kinect and Vive, where L and R denote left and right respectively. For example, there is a left upper cut gesture and a right uppercut gesture. All seventeen gestures were used in our Kinect evaluation, where only the first eleven (top row and golf swing) were used for Vive.

comparing game data to homogenous cross validated data [15, 75, 76], which we believe is also likely to make segmentation an even more difficult problem.

6.1.1 Apparatuses. We chose to leverage the Kinect, Vive, and mouse for a number of reasons. Namely, they cover a wide range of input device types, sensor qualities, and data representations. Concerning Kinect, we recorded 21 three-dimensional (3D) skeletal joint positions using an XBOX ONE Kinect, for a total of $m = 63$ components per data point. However, since the hand and foot data were especially noisy, we zeroed out this part of the signal. The HTC Vive reports two 3D controller positions and orientations via quaternions, which we break into two separate datasets, for a total of $m = 6$ and $m = 8$ components per data point, respectively. Last, a mouse reports 2D position data, thereby yielding $m = 2$ components per data point (since we do not sample button state signals). We also chose to evaluate mouse data because it is challenging to perform complex trajectories with this input device due to issues with friction and range, whereby participants must constantly reposition their arm, hand, and device.

6.1.2 Follow the Leader. We developed FTL in Unity[5] as a means to procure Kinect, HTC Vive controller, and mouse data. FTL has two modes, which are the demonstration and game modes. In demonstration mode, we record five custom samples of each gesture class in a randomized order so as to increase within class variability[6]. For Kinect and Vive, we first display text that specifies what gesture the participant must perform, and once he or she indicates their readiness, we start a countdown timer. When this timer reachers zero, we commence recording and conclude the record via a keyboard stroke after the gesture is complete. For mouse data, a participant instead toggles the left mouse button to achieve the same effect. We replay each sample to verify correctness and ensure the participant is satisfied with their result. When a participant performs a gesture incorrectly or a tracking error occurs, we simply discard the faulty sample and try again. In total, we collected three datasets comprising 17 full-body Kinect, 11 HTC Vive controller, and 10 mouse gestures. See Figures 10 and 11 for a depiction of each.

[5]Unity is a popular, real-time game development platform, see https://unity.com/.
[6]Gestures repeated back-to-back may have less variation.

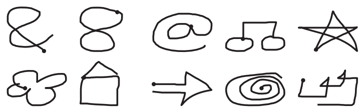

Fig. 11. The ten mouse gestures used in our evaluation, taken from [81, 89].

In FTL's game mode, we render a leader[7] on screen and participants are told to follow its motion as closely as possible. All avatar movements are non-gesture actions designed to mimic the direct manipulation and puppeteering kind of interactions that may occur in a video game, however, at unknown intervals we display a text command instructing the participant to perform a specific gesture. During gesticulation, we visually confirmed that the participant correctly produced the gesture, as there were some instances when a participant forgot a gesture's form or confused left with right. If we observed an error, the instance was re-queued to be retried at a later time. We collected three correct instances of every gesture throughout each game session. Before we began the main session, however, we first allowed each participant to complete a practice round, where each gesture was executed one time, thereby allowing a participant to become familiar with FTL's mechanics. Each practice session lasted approximately 2 minutes.

6.1.3 Subjects. We recruited thirty students (17 male, 13 female) from a local university to participate in our study. Their ages ranged from 18 to 29 with a median age of 22. All but one participant owned a game system, and none of the participants had any mobility problems. For each participant, we first explained the study's purpose and then demonstrated each gesture they would have to perform, after which we started to collect data. Each session lasted about 30 minutes and participants were allowed to take a break if they felt tired.

7 EVALUATION OF ANGULAR DP

To understand whether angular DP is a viable alternative to standard DP and uniform resampling, we examined the effect of resampling rate on all training data under an inner product measure. For a given training sample and rate N, we spatially resampled the training sample to N points using DP, angular DP, and uniform resampling. Then to measure differences between a training sample and its resampled variants, we again uniformly resampled all four trajectories to a high resolution ($N = 1024$), and measured the inner product between corresponding normalized direction vectors:

$$f\left(\vec{X}, \vec{Y}\right) = \sum_{i=1}^{1024} \left\langle \vec{\boldsymbol{x}}_i, \vec{\boldsymbol{y}}_i \right\rangle . \tag{24}$$

Dissimilarity measure f informs us of the resampled trajectory's likeness to its original form. We next relate the three methods to each other by measuring the percentage improvement of angular DP and standard DP over uniform resampling. This improvement is equivalent to the percentage decrease in dissimilarity:

$$\text{Improvement} = \left(1 - \frac{\text{DP}}{\text{Uniform}}\right) \times 100\%, \tag{25}$$

where DP is the dissimilarity measure for a given DP method, and Uniform is the same measure under uniform resampling.

We finally averaged together all results per dataset per N as shown in Figure 12. Across all conditions, angular DP showed an average improvement of 44% ($\sigma = 24\%$) over uniform resampling, whereas standard DP only achieved an average improvement of 37% ($\sigma = 24\%$). Further, on

[7]With Kinect and Vive, the leader is a skeleton model, whereas with mouse, the leader is a long mouse trail.

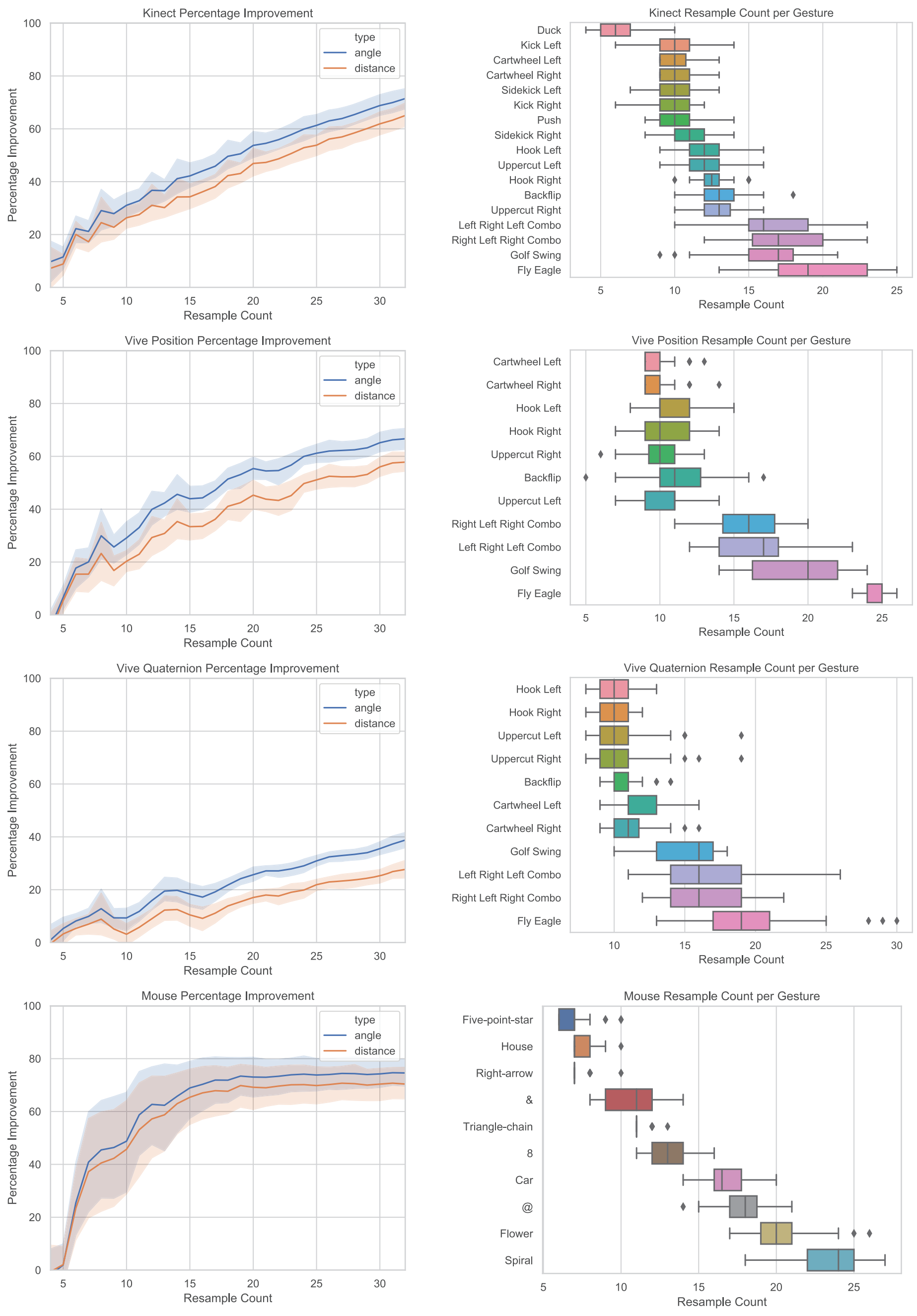

Fig. 12. Left: Percentage improvement in self similarity measure relative to uniform resampling for standard DP using distance only and our angular DP variant (higher is better). Right: Resample count distribution per gesture, sorted by median.

visual inspection, we see the direction of this difference is consistent for all values of N, across all datasets, except for especially low resampling rates where all three methods are approximately equivalent. In our view, these results justify our minor modification to DP when working with direction vectors.

7.1 Automatic Parameter Selection

As part of our evaluation, we also recorded the resampling rate N selected by our automatic parameter selection algorithm for angular DP (Section 4.6) for each training sample. These results are shown in Figure 12, which are grouped by gesture class and sorted by median. Across all conditions, the average resampling rate was $N = 13$ ($\sigma = 4.6$), whereas the minimum and maximum values were respectively 4 and 30. Compared with common rates found in the literature (e.g., the recent $Q uses $N = 32$), these results are quite low, which is expected to boost Machete's computational performance. Further, results are consistent with our intuition that longer and more complex gestures require higher N to adequately describe its shape.

8 EVALUATION OF MACHETE

We compare Machete against three alternative segmentation approaches to determine which technique is most performant in both accuracy and computation across all four datasets. We assume that a given segmenter is paired with a robust recognizer having a reliable accept–reject criteria. In this way, the segmenter may frequently pass candidate gestures to the recognizer without concern for generating false positives, and the recognizer in turn will evaluate each candidate, accepting only those queries that are most like their gesture class while rejecting any query that does not match. In practice, the recognizer may incorrectly accept or reject certain candidates or notify an application of gesticulation before it is complete, even as a better solution simultaneously unfolds. However, for the purpose of our first evaluation, we focus mainly on how well our segmenter is able to localize a gesture's end points.

Specifically, we pair each segmenter with Jackknife [74], a general purpose, device-agnostic custom gesture recognizer. For a given participant and training sample, we train both the segmenter and Jackknife with the specified sample, after which we replay the session one frame at a time through the system. After each call, the segmenter returns a flag indicating whether or not it has identified a gesture candidate. If so, the candidate is fed to the recognizer as a query where it is scored. When playback is within the boundaries of a section where FTL told a participant to perform a gesture, we log the best score and segmentation result.

8.1 The Contenders

As described in our Related Work section, we identified three segmentation approaches that we believe are compliant with $-family principles: an energy-based method, windowing, and CDP. The former identifies gesture boundaries by examining a time series's energy profile. Given input time series X, we define its energy profile as:

$$E = (e_i \mid i = 1 \ldots |X| - 1) \tag{26}$$

$$e_i = \frac{1}{2}||\boldsymbol{x}_{i+1} - \boldsymbol{x}_i||^2. \tag{27}$$

We treat each minimum in E as a candidate gesture boundary. As such, when we observe a new boundary in a continuous stream, we assume it correlates with the end of a gesture, and its beginning can be any previous boundary that falls within 1.5 times the training sample's length. We then pass each start-end pair as a candidate gesture to the underlying recognizer for further analysis. It may be possible for us to reduce the number of candidate gestures by rejecting boundaries whose energies are too high, but this requires learning a threshold from training data, which is difficult to do in a customization context.

Windowing is perhaps the most popular method and was included because of its ubiquity and effectiveness. In our evaluation, we assume that the training sample length provides the best approximation of duration in practice. Therefore, we match our window length to the training sample. One can instead use multiple windows to improve segmentation, but this will increase computational costs, which is contrary to our objective.

Finally, since Machete is based on CDP, we also wanted to evaluate standard CDP. One issue, however, is that CDP measures ED, which is neither position nor scale invariant and consequently inappropriate for our evaluation. For this reason, we replace its local cost function with an inner product measure that works on normalized direction vectors, though the result is not squared as it is with Machete. Further, we normalize the accumulated cost by warping path length, not distance, as we do for Machete. Note, without these two changes, we found CDP was unusable with our HA datasets.

8.2 Preprocessing and Ground Truth

Since a training sample is likely to contain idle data at its tails, we trim each Kinect and Vive sample as part of a preprocessing step. This step was especially helpful in controlling the sliding window length, which will have otherwise been too long. Specifically, to remove unwanted frames, we computed the cumulative energy over time and removed all frames that fall outside of the inner 98%; 1% of energy was cut from each tail. What remains is a valid pattern that a given segmenter must match. We chose to use this process given that it is sufficiently straightforward and anyone can use it without difficulty in practice. Second, because mouse data may contain hooks,[8] after angular DP processing, we heuristically cut the first and last vector when their length was less than 20% of their neighboring vectors. Further, because there is a great deal of variability in mouse gesture production, we additionally train the system with samples rotated ±15 degrees—thus, each training sample is converted into three templates for all segmenters.

For each gesture instance in a session, we must establish ground truth segmentation and identify where in time the gesture begins and ends. To accomplish this task, we use a two step process. First, we conduct a naive brute force search in the region of a session where an instance is known to reside. Specifically, if we expect to find a gesture instance between times t_1 and t_2, we search for the start s and end e times that minimize the dissimilarly between subsequence $X_{s,e}$ and a given training sample Y under measure f:

$$s, e = \underset{t_1 \leq s \leq e \leq t_2}{\text{argmin}} \; f(X_{s,e}, Y), \tag{28}$$

where f is a DTW-based measure that varies per device type and yields a reasonable segment approximation. Since there are five training samples per gesture class, we further take the median of the aggregate start and end results as an estimate of ground truth. Finally, in step two, we examine each result from step one. If we observe an obvious error, we manually search for the correct solution and log the updated result. However, we try to limit such corrections as this second step is highly subjective.

8.3 Error Measures

Given our evaluation protocol, we obtained one segmentation result per instance per training sample that in aggregate form 7,350 results—10 participants × 5 training samples per participant

[8]Changes in direction due to imprecision that sometimes occur at the start or end of a stroke.

$\times$ 3 instances per session $\times$ (17 Kinect + 11 Vive position + 11 Vive quaternion + 10 mouse) gesture classes. We examined each result to determine if the given segmenter correctly detected the associated gesture instance. For each miss, we removed the offending result along with the remaining three corresponding segmentation results, so that the population remained identical between methods. With those results that remained, we proceeded to extract four error measures. The first three of these were the signed start, signed end, and total segmentation errors. Let G_s and G_e, respectively, denote the ground truth start and end times, and let S_s and S_e similarly denote the segmenter candidate boundaries. The three errors are then defined as:

$$\text{Start Error} = S_s - G_s, \tag{29}$$

$$\text{End Error} = S_e - G_e, \tag{30}$$

$$\text{Total Error} = |\text{Start Error}| + |\text{End Error}| \,. \tag{31}$$

These measures are important in understanding how well a method temporally segments gestures, but alone the result can be misleading. When one gesticulates quickly, the difference between two frames can have a significant impact on the gesture's shape, whereas this difference is less dramatic at lower speeds. For this reason, we also measure the arc length error:

$$\text{Arc Length Error} = \frac{\mathcal{L}\left(X\left[\min(S_s, G_s) : \max(S_s, G_s)\right]\right) + \mathcal{L}\left(X\left[\min(S_e, G_e) : \max(S_e, G_e)\right]\right)}{\mathcal{L}\left(X[G_s : G_e]\right)}, \tag{32}$$

where X is the session time series, and $\mathcal{L}$ is the arc length:

$$\mathcal{L}(X) = \sum_{i=2}^{|X|} \|\boldsymbol{x}_i - \boldsymbol{x}_{i-1}\| \,. \tag{33}$$

As one can see, we relate clipped and extended boundaries to the gesture's total arc length via a ratio. If the start and end errors for a given gesture instance are respectfully one and zero frames, but this difference represents 5% of total gesture's arc length, then the arc length error is 5%, which we believe better represents how well a segmenter captured the gesture's shape, compared to time-based error measures. To gain an intuition of this measure, we visualize varying arc length error magnitudes in Figure 13. Roughly, with errors that are 10% or less, we see that the gesture shape is well preserved. Beyond this point, clipping and extensions begin to fundamentally change gesture shapes, which can lead to an increase in false negatives when the underlying recognizer is unable to identify an otherwise well articulated gesture.

8.3.1 Design and Analysis. We conducted one experiment per dataset, where each experiment was a 1-factor repeated measures design. The nominal factor was *segmenter*, for which there were four levels: Machete, CDP, Window, and Energy. The outcome variables were those just discussed—the start, end, total, and arc length errors. These errors were averaged over each participant into a single result. We then used Friedman testing [25] to drive our omnibus, after which we conducted post-hoc analysis with exact, two-tailed Wilcoxon signed rank tests, while protecting against multiple comparison type I errors by way of the Holm–Bonferroni step-down procedure [29].

8.4 Results

8.4.1 Temporal Variability. We first examined the temporal variability of ground truth results as shown in Figure 14. These results represent the pairwise interclass distribution of duration, defined as the ratio of maximum to minimum duration between each pair of gesture instances belonging to the same class and participant. We included these data specifically to give one a sense of how well we might expect the window method to do over the varying datasets. Vive gestures

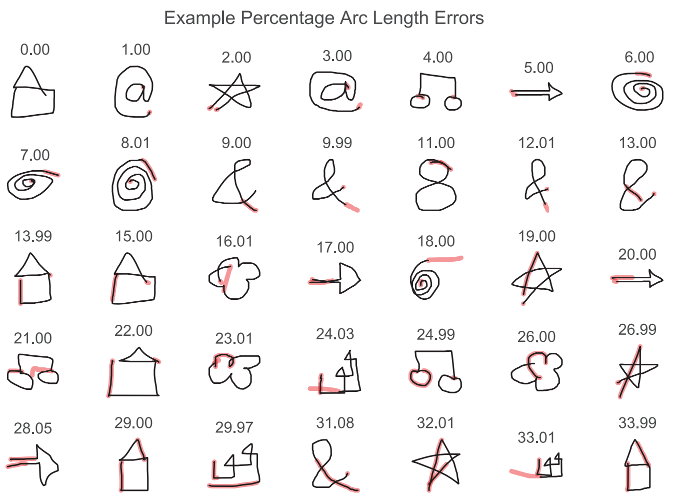

Fig. 13. Varying arc length errors over mouse data. We render ground truth as a thin black line and highlight errors in red where the segmenter truncated a gesture or extended beyond its temporal boundary.

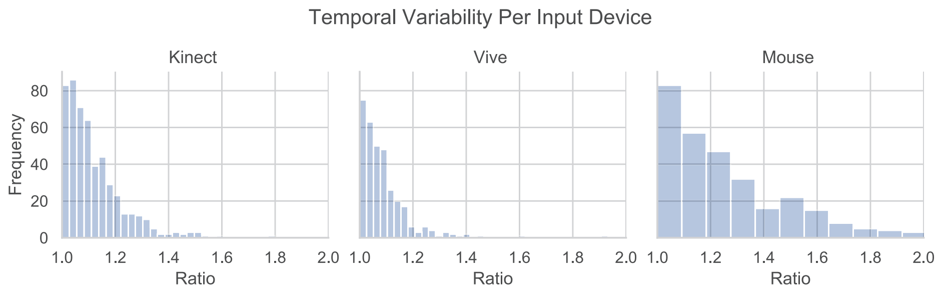

Fig. 14. Distribution of pairwise intraclass ground truth duration ratios.

Table 1. Average (μ) per Frame Processing Time in Nanoseconds and Standard Deviation (σ)

	Kinect		Vive Position		Vive Quaternion		Mouse	
Method	μ	(σ)	μ	(σ)	μ	(σ)	μ	(σ)
Machete	1.71	(0.48)	0.54	(0.18)	0.57	(0.16)	2.63	(0.73)
CDP	6.55	(0.86)	1.61	(0.31)	1.81	(0.34)	6.28	(0.97)
Window	98.44	(17.19)	44.58	(11.92)	53.94	(11.39)	123.84	(19.91)
Energy	13445.53	(2035.94)	6507.65	(2661.44)	9646.43	(3952.72)	6836.84	(1777.55)

have the least variation ($\mu = 1.08, \sigma = 0.08$), which is followed by Kinect ($\mu = 1.11, \sigma = 0.11$) and mouse ($\mu = 1.27, \sigma = 0.28$) gestures. Indeed we see by arc length error analysis (discussed soon), that the windowing segmenter's performance declines across the datasets in accordance with this variability. In most cases, performance is still acceptable, but it could be improved by using a multiple window scheme.

8.4.2 Computational Performance. In addition to segmentation errors, we also recorded the average processing time per frame needed to run each segmenter, which includes any calls to the underlying Jackknife recognizer. Average nanosecond times are presented in Table 1. We see that

Table 2. Percentage Decrease in Computation Time per Segmenter Invocation Using Window as the Baseline

Method	Kinect	Vive Position	Vive Quaternion	Mouse
Machete	98.27	98.80	98.95	97.88
CDP	93.35	96.39	96.65	94.93
Window	0.00	0.00	0.00	0.00
Energy	−13558.13	−14498.28	−17782.85	−5420.48

Table 3. Average (μ) Segmentation Error in Frames and Standard Deviation (σ) for Kinect Position Data, as well as the Percentage Path Length Error

	Start Error		End Error		Total Error		Length Error	
Method	μ	(σ)	μ	(σ)	μ	(σ)	μ	(σ)
Machete	2.41	(0.65)	3.45	(1.05)	5.86	(1.68)	7.45	(1.75)
CDP	4.50	(0.77)	3.70	(1.15)	8.21	(1.89)	12.48	(1.56)
Window	3.42	(0.84)	4.94	(1.26)	8.36	(1.74)	8.17	(1.92)
Energy	4.42	(1.21)	6.86	(1.80)	11.28	(2.48)	10.23	(2.18)

Machete and CDP are extremely fast compared to window and energy-based segmentation. The latter is especially slow because it calls Jackknife numerous times when it identifies a local minimum in the energy profile. Since we believe that windowing is the present first-choice method for segmentation, we use this technique as a baseline to calculate the percentage decrease in processing time as presented in Table 2. Machete reduces the computational burden by at least 98% across all experiments. CDP performs similarly well, but because its accuracy is subpar (as will be shown), we do not consider CDP a practical option. In contrast, energy-based segmentation runs thousands of times worse than window based segmentation, which implies one is better off running multiple sliding windows in parallel.

In this work, we are concerned primarily with computational performance because as discussed in Section 2, fast processing allows us to process more templates per unit time, enable segmentation on low resource devices, and free up system resources; and as shown, Machete is remarkably efficient. However, with only one template loaded, we cannot afford to sacrifice segmentation accuracy, and as we see next, Machete works well under this condition.

8.5 Segmentation Accuracy

8.5.1 Kinect Segmentation. Friedman tests revealed a significant difference between the four segmenters across all error measures. On average, Machete achieved the lowest error as shown in Table 3. Concerning start segmentation error, Machete was significantly different from CDP and energy, but not window, per Table 4. This was true for the remaining measures as well. Window was also different from energy, and energy from CDP across the end segmentation and total error measures. These results in sum suggest that Machete and window-based segmentation are both reasonable approaches, though Machete may be a better option because of its significantly lower total segmentation error.

8.5.2 Vive Position Segmentation. Results for Vive position data are very similar to Kinect. Friedman tests revealed a significant difference between the four segmenters across all error measures. On average, Machete again achieved the lowest error as shown in Table 5. On the start, total,

Table 4. Kinect Segmentation Post-hoc Results Using the Exact, Two-tailed Wilcoxon Signed Rank Test

	Start Error			End Error			Total Error			Length Error		
	$\left(\chi^2_3 = 20.0, p < .001\right)$			$\left(\chi^2_3 = 22.2, p < .001\right)$			$\left(\chi^2_3 = 22.0, p < .001\right)$			$\left(\chi^2_3 = 22.6, p < .001\right)$		
Pair	$W+$	p	r	$W+$	p	r	$W+$	p	r	$W+$	p	r
Machete - Window	4	0.05	—	6	0.05	—	3	<.05	0.9 L	16	0.28	—
Machete - CDP	55	<.05	1.0 L	55	<.05	1.0 L	55	<.05	1.0 L	55	<.05	1.0 L
Machete - Energy	0	<.05	1.0 L	0	<.05	1.0 L	0	<.05	1.0 L	3	<.05	0.9 L
Window - CDP	49	0.08	—	9	0.06	—	26	0.92	—	55	<.05	1.0 L
Window - Energy	47	0.10	—	54	<.05	1.0 L	52	<.05	0.9 L	46	0.15	—
CDP - Energy	35	0.49	—	0	<.05	1.0 L	5	<.05	0.8 L	47	0.15	—

Friedman test results are also listed under each column header.

Table 5. Average (μ) Segmentation Error in Frames and Standard Deviation (σ) for Vive Position Data, as well as the Percentage Path Length Error

	Start Error		End Error		Total Error		Length Error	
Method	μ	(σ)	μ	(σ)	μ	(σ)	μ	(σ)
Machete	8.95	(2.45)	10.65	(2.87)	19.60	(4.56)	6.10	(1.33)
CDP	16.61	(2.32)	11.68	(3.05)	28.29	(3.83)	13.93	(1.16)
Window	10.83	(3.50)	13.11	(2.90)	23.95	(5.84)	6.93	(0.89)
Energy	17.00	(3.60)	22.93	(5.43)	39.93	(7.64)	8.31	(1.94)

Table 6. Vive Position Segmentation Post-hoc Results Using the Exact, Two-tailed Wilcoxon Signed Rank Test

	Start Error			End Error			Total Error			Length Error		
	$\left(\chi^2_3 = 22.4, p < .001\right)$			$\left(\chi^2_3 = 24.2, p < .001\right)$			$\left(\chi^2_3 = 23.5, p < .001\right)$			$\left(\chi^2_3 = 22.4, p < .001\right)$		
Pair	$W+$	p	r	$W+$	p	r	$W+$	p	r	$W+$	p	r
Machete - Window	14	0.39	—	1	<.05	1.0 L	6	0.05	—	9	0.13	—
Machete - CDP	55	<.05	1.0 L	48	0.07	—	55	<.05	1.0 L	55	<.05	1.0 L
Machete - Energy	0	<.05	1.0 L	0	<.05	1.0 L	0	<.05	1.0 L	4	<.05	0.9 L
Window - CDP	53	<.05	0.9 L	12	0.13	—	49	0.05	—	55	<.05	1.0 L
Window - Energy	55	<.05	1.0 L	55	<.05	1.0 L	55	<.05	1.0 L	46	0.13	—
CDP - Energy	24	0.77	—	0	<.05	1.0 L	1	<.05	1.0 L	54	<.05	1.0 L

Friedman test results are also shown under each column header.

and arc length error, Machete was significantly different from CDP and energy, but not window, per Table 6; but Machete was not different from CDP in end error. Like with Kinect, these results suggest that Machete and window-based segmentation are both reasonable approaches, though Machete may be a better option because of its significantly lower end segmentation error. We also note that their difference in the end error measure was marginal.

8.5.3 Vive Quaternion Segmentation. Friedman tests revealed a significant difference between the four segmenters across all error measures. On average, Machete for the third time achieved the lowest error as shown in Table 7, although there were less significant differences (see Table 8). In start and total error, Machete differed from CDP and energy, and on arc length error, Machete differed from CDP. Across the four measures, Machete did not differ from window-based segmentation, but we note that the end and total errors are marginal. Again, like with Kinect and Vive, these results suggest that Machete and window-based segmentation are both reasonable approaches, but unlike in prior cases, Machete does not have a clear edge over windowing in error.

Table 7. Average (μ) Segmentation Error in Frames and Standard Deviation (σ) for Vive Quaternion Data, as well as the Percentage Path Length Error

	Start Error		**End Error**		**Total Error**		**Length Error**	
Method	μ	(σ)	μ	(σ)	μ	(σ)	μ	(σ)
Machete	9.02	(1.83)	10.72	(2.10)	19.75	(2.75)	7.32	(0.76)
CDP	12.94	(2.70)	11.27	(2.14)	24.20	(3.31)	12.15	(1.47)
Window	12.71	(5.13)	14.09	(3.94)	26.80	(7.96)	7.85	(1.29)
Energy	17.82	(4.58)	17.17	(3.46)	34.99	(6.01)	8.51	(1.11)

Table 8. Vive Quaternion Segmentation Post-hoc Results Using the Exact, Two-tailed Wilcoxon Signed Rank Test

	Start Error			**End Error**			**Total Error**			**Length Error**		
	$\left(\chi^2_3 = 19.1, p < .001\right)$			$\left(\chi^2_3 = 17.8, p < .001\right)$			$\left(\chi^2_3 = 20.3, p < .001\right)$			$\left(\chi^2_3 = 21.0, p < .001\right)$		
Pair	$W+$	p	r	$W+$	p	r	$W+$	p	r	$W+$	p	r
Machete - Window	7	0.11	—	4	0.05	—	5	0.06	—	16	0.55	—
Machete - CDP	54	<.05	1.0 L	46	0.13	—	55	<.05	1.0 L	55	<.05	1.0 L
Machete - Energy	0	<.05	1.0 L	0	<.05	1.0 L	0	<.05	1.0 L	6	0.08	—
Window - CDP	33	0.62	—	7	0.11	—	26	0.92	—	55	<.05	1.0 L
Window - Energy	49	0.11	—	45	0.13	—	49	0.06	—	39	0.55	—
CDP - Energy	7	0.11	—	0	<.05	1.0 L	0	<.05	1.0 L	55	<.05	1.0 L

Friedman test results are also listed under each column header.

Table 9. Average (μ) Segmentation Error in Frames and Standard Deviation (σ) for Mouse Position Data, as well as the Percentage Path Length Error

	Start Error		**End Error**		**Total Error**		**Length Error**	
Method	μ	(σ)	μ	(σ)	μ	(σ)	μ	(σ)
Machete	19.86	(8.68)	24.90	(4.38)	44.76	(7.68)	5.61	(1.31)
CDP	35.35	(8.73)	25.27	(3.87)	60.62	(9.66)	13.98	(2.12)
Window	28.75	(7.72)	33.57	(4.88)	62.31	(9.14)	10.08	(2.46)
Energy	25.72	(7.77)	26.84	(3.51)	52.56	(8.46)	4.73	(1.36)

8.5.4 Mouse Segmentation. Friedman tests revealed a significant difference between the four segmenters across all error measures. For the first time, energy achieved the lowest arc length error, which was significantly different from all other methods (see Tables 9 and 10). However, Machete still achieved the lowest average start, end, and total segmentation errors, and on all measures Machete was significantly different from window-based segmentation. Although energy based segmentation performed better on arc length error, we note that both Machete and energy perform well, where both are better than 6% (the lowest averages for all datasets). On the other hand, window-based segmentation performed at its worst, which may be due to the increased temporal variability in this dataset.

8.6 Machete in Practice

To better understand the viability of Machete in practice, we implemented the custom gesture recognition pipeline shown in Figure 1. We used Jackknife [74] to classify segmented gesture candidates. Since automatic rejection threshold selection for custom gestures is yet an unsolved problem, and because we are primarily concerned with the effect of segmentation on recognition and

Table 10. Mouse Segmentation Post-hoc Results Using the Exact, Two-tailed Wilcoxon Signed Rank Test

	Start Error			**End Error**			**Total Error**			**Length Error**		
	$\left(\chi^2_3 = 19.6, p < .001\right)$			$\left(\chi^2_3 = 16.9, p < .001\right)$			$\left(\chi^2_3 = 22.4, p < .001\right)$			$\left(\chi^2_3 = 27.8, p < .001\right)$		
Pair	$W+$	p	r	$W+$	p	r	$W+$	p	r	$W+$	p	r
Machete - Window	2	<.05	0.9 L	0	<.05	1.0 L	0	<.05	1.0 L	0	<.05	1.0 L
Machete - CDP	55	<.05	1.0 L	32	0.70	—	55	<.05	1.0 L	55	<.05	1.0 L
Machete - Energy	6	0.05	—	14	0.39	—	0	<.05	1.0 L	51	<.05	0.9 L
Window - CDP	53	<.05	0.9 L	1	<.05	1.0 L	22	0.62	—	54	<.05	1.0 L
Window - Energy	15	0.23	—	0	<.05	1.0 L	4	<.05	0.9 L	0	<.05	1.0 L
CDP - Energy	53	<.05	0.9 L	12	0.39	—	51	<.05	0.9 L	55	<.05	1.0 L

Friedman tests results are also shown under each column header.

latency, we used a grid search to find optimal rejection thresholds for each condition discussed below.

Based on results reported in the prior section, we further decided to evaluate both Machete-based and Window-based segmentation on Kinect input. Both techniques are comparable in start, end, and path length error, showing no statistically significant difference in accuracy when one selects an appropriate window size. Insufficient accuracy prohibits the use of CDP, which also impacts energy-based segmentation, as does computational performance. Further, since windowing is a popular technique, it is useful to understand where Machete stands with respect to window-based segmentation.

One challenge with windowing is that one must select an appropriate window size. Too short and gesture candidates will be truncated. Too large and the window will contain data from temporally adjacent activities. In both case, a recognizer may reject the result. In our prior test, we set the window size to that of its associated training sample length to understand the potential of window-based segmentation. In a practical application, one must balance accuracy with latency and use only a minimum number of windows. When gesture lengths are homogeneous, one can use a single window and expect reasonable segmentation. In our case, gesture length varies between short and long gestures; so it is unclear what window size we should use. For this reason, we evaluate five options: a single window using the (1) minimum, (2) middle, and (3) maximum sample length; (4) two windows using the minimum and maximum training sample length; and (5) all three together.

Since our goal with this evaluation is to understand how segmentation may impact user experience, we measure recognition accuracy and latency. See Section 2.4 for a discussion on the importance of these measures. Latency results were collected with an Intel Core i7-7700K with 24 GB 2400 MHz DRAM and an Nvidia GeForce GTX 1080, running Windows 10.

8.6.1 Recognition Accuracy. To measure the effect of segmentation on recognition accuracy in a realistic scenario, we use the following procedure: For a given participant P, we randomly select T training samples per gesture class. We train both the segmenter and Jackknife with said random sample, and then replay participant P's session through our pipeline, one frame at a time. During replay, we record all true positive tp, false positive fp, and false negative fn classification results—gesture candidates output by the segmenter that are not rejected by Jackknife. Results are then combined into an F_1-score, a commonly used accuracy measure balancing precision and recall:

$$F_1 = \frac{tp}{tp + \frac{1}{2}\left(fp + fn\right)}. \tag{34}$$

Recognizers achieve a perfect score when all performed gestures are correctly classified, but degrade as misses and misclassifications increase. In our evaluation, a true positive occurs when a participant performs the specified gesture and our pipeline correctly observes the correct pattern.

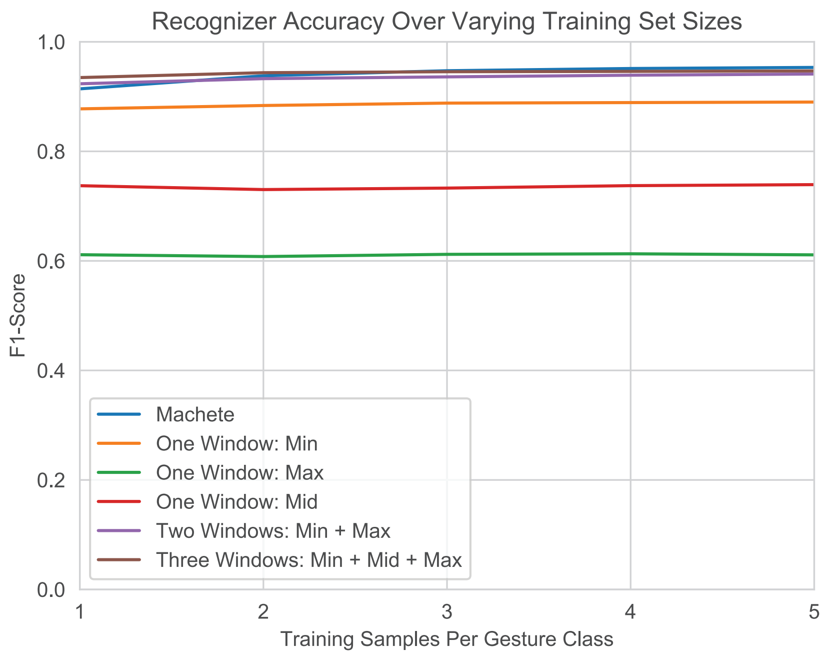

Fig. 15. F_1-score accuracy for varying training set sizes for different segmentation strategies. Minimum refers to the length of the minimum length training sample, whereas maximum refers to the maximum length training sample and mid is their average.

Otherwise, missed patterns are treated as false negatives, and missclasficiations as false positives. For each participant and level of T, we repeat the process 10 times and average all F_1 score results into a single accuracy measure per individual. Training count T varies from 1 to 5.

Results are shown in Figure 15. We observe that poor window choices lead to poor recognition accuracy. Using only the maximum or mid window length, windowing achieves only a 61% and 74% accuracy, respectfully. By segmenting with the minimum length, accuracy increases to 88% for $T = 1$ and 89% for $T = 3$, but there is no further gain. Machete and multiple window segmentation significantly outperform single window segmentation, and are comparable to each other. Using two windows (minimum and maximum), accuracy starts at 92% and increases to 94% as T increases, and three window segmentation performs similarly. Machete-based segmentation also performs well, has the sharpest incline, and ends with the highest result, starting at 91% and ending at 95%.

As we saw earlier (Figure 14), a number of Kinect gestures share similar temporal variation, which is likely why the minimum sized window works best of all single window options. The remaining gestures are sufficiently long and captured by the maximum window. For gesture sets that possess greater variability than ours, selecting appropriate window lengths will prove to be more challenging. Further, we find that poor segmentation leads to suboptimal results, evidenced by a substantial reduction in recognition accuracies achieved by the single window methods. These lower results force one to use multiple windows, leading to observable latency degradations as we will soon see.

8.6.2 Latency. To evaluate the effect of segmentation on latency, we implemented our pipeline in a Unity based video game called Sleepy Town, Figure 16 (see [73] for a full description). It presents a simple geometric 3D environment in which citizens wander between random points without purpose. We chose to use this simple platform because it uses many features common

Fig. 16. Screenshot of Sleepy Town [73] environment.

among modern games, including path planning, behavior scripts, collision detection and response, physics, scene management, and rendering. Our pipeline must, therefore, share system resources with other various ongoing processes. This means that gesture recognition overhead may directly impact frame rate and system responsiveness.

Due to COVID-19, we were unable to directly collect participant data. For this reason, we decided to select an exemplar participant from our previous FTL data collection procedure and replay their session data through our pipeline as if it were read directly from a Kinect input device. We repeated this procedure for varying training set sizes and segmenter approaches as discussed in the previous section. During this time, we tracked frame rate and recognition accuracies (which were confirmed to match our prior results).

Results are shown in Figure 17. Sleepy Town running without gesture recognition maintains approximately 137 FPS. The single window segmenters drop down to near 60 FPS with one template per gesture class loaded, and performance continues to drop as more templates are loaded, ultimately hitting 32–35 FPS. Two window segmentation starts at 58 FPS and drops to 19 FPS. Three window segmentation drops from 40 to 13 FPS. Machete outperforms all window options, starting at close to maximum performance, 126 FPS, and dropping down to 83 FPS.

As expected, we find there is significant overhead when using window-based segmentation. Since recognition accuracy may be too low with one window in many cases, we expect one will prefer to use two windows, if not more (for instance, Multiwave [59] used five windows to handle temporal variability issues). We also note that all methods degrade as the number of templates increase, though Machete is the only method that continues to operate in an acceptable range with 85 templates loaded [16] (17 gestures × 5 per class). This increase in performance will benefit user experience in both desktop and virtual reality applications, where high latency can degrade performance and presence, as well as increase the probability of inducing motion sickness.

9 DISCUSSION

Our experiments reveal that Machete is the only segmentation method that consistently achieves high accuracy across all datasets, and in most cases, Machete is the best performing method. Window-based segmentation is a close second on three datasets, but performed poorly on mouse

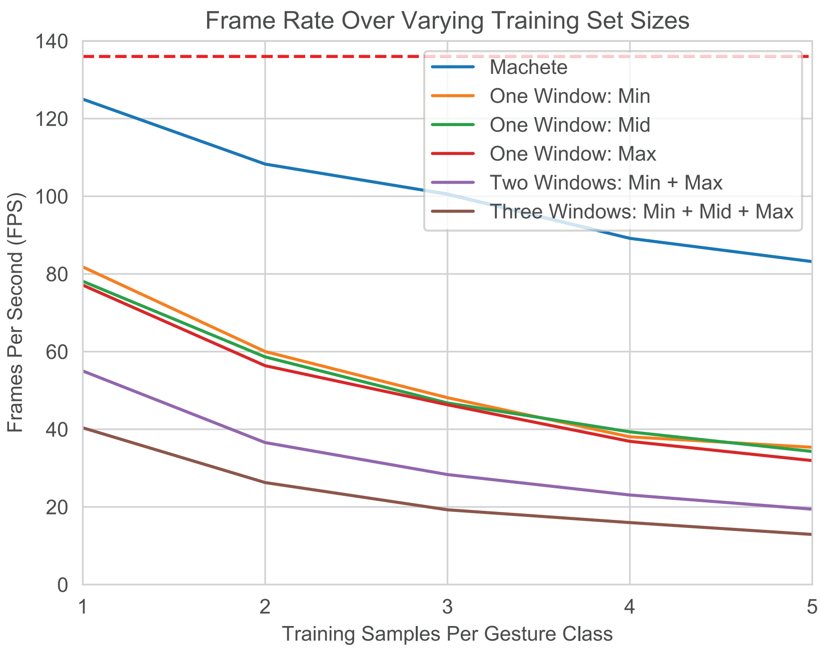

Fig. 17. Frame rate over varying training set sizes for the various segmentation configurations. Curves are in approximate order of the legend labels. The horizontal red dashed line is the frame rate of Sleepy Town without gesture recognition.

data. Although there was not a statistically significant difference between Machete and windowing across all conditions, the direction of their differences remained constant. When we look at computational performance, only CDP can compete, but because of its poor accuracy, CDP is not a viable option. Machete's combined accuracy, speed, scalability, and simplicity makes it a clearly capable segmenter.

In more detail, we found that windowing works relatively well in the presence of minor temporal variabilities, but falters as variability increases. We can recover performance by employing multiple sliding windows, where we evaluate each window every frame and take the best result. A related practical issue with our evaluation is that we fit our window size to the training data, so each template uses its own window. In practice, one may cluster templates into groups that share a common window size, or one might select just a few sizes that all templates share. The downside of using multiple windows is obvious though, more windows means more overhead means more processing, and there is still no guarantee a particular size will fit a given gesture instance. Alternatively, one can use a single long sliding window to segment input when and if gesture boundaries are delineated by periods of inactivity between actions. For instance, a two second gesture is adequately described by a four second window when a user holds their pose before and after gesticulation, and when the underlying recognizer is time invariant. This approach may be appropriate for low-activity interactions but not for HA interactions where users are engaged in constant motion. Machete does not suffer from these issues, which is why our technique outperformed window-based segmentation in every test.

We found that energy-based segmentation was often less accurate and its computational performance was also low. Although we were able to quickly calculate the input signal's energy profile, performance dropped because of an overabundance in minima, each of which resulted in multiple calls to the recognizer. We believe with more work we can develop a thresholding scheme that rejects certain minima and coalesces collocated minima to reduce the number of candidate gestures. However, based on work by Kahol et al. [35], we fear a workable solution will be complicated, device specific, or both. Interestingly, the method did very well on mouse data. There are often clear low-energy start and stop points in this dataset where the segmenter could latch, which is why accuracy is high in this case. In a related way, this also may be why the segmenter is less successful on full body gestures. Recall that gestures comprise pre-stroke, nucleus, and post-stroke activities [94], where the nucleus defines the main action. It may be that the nucleus is clearly delineated in mouse data but not in full body gestures. Regardless, because of the segmenter's inconsistent performance and high overhead, we believe it is not a good general first-choice technique for custom gesture recognition.

CDP was the worst performing segmenter in our evaluation. We noticed that CDP had consistently high start errors, and visual inspection of the associated distributions reveals a bias toward late segmentation. This issue is likely because of CDP's zero sink node cost, an issue we addressed in Machete. With respect to computational performance, CDP did very well, though because we were unable to find a reliable rejection threshold, it did not perform as well as Machete. This difference in part is due to Machete's improved local cost function. In other words, our modifications to CDP have a profound impact on performance.

Another issue with both windowing and energy-based segmentation is that these methods do not prune gesture candidates. The latter technique will identify potential gesture boundaries and invoke the recognizer with a query that it must score against each template. Other than duration and application-specific context, there is no a-priori information available that the recognizer can use to discard unlikely matches. Windowing has a similar problem. Whether templates share a window or are independent, the recognizer must evaluate each per frame. Another advantage of Machete is that the recognizer only evaluates one query-template pair per invocation, which further reduces the workload.

Advantages of Machete over other segmenters were also found in our recognition and latency evaluations. As expected, with better segmentation, we find gesture recognition accuracy improves. Otherwise, gesture sequences that are truncated or extended beyond their natural boundaries may be rejected by the underlying recognizer. In our case, to match Machete's accuracy, we were required to use two windows, but not without sacrificing significant computational resources. Frame rates in our Sleepy Town video game fell to below 20 FPS with two window segmentation as we scaled the training set size up from one to five samples per gesture class (85 samples total). In the worst case, Machete remained above 80 FPS. Low frame rates are known to impact perceived quality and performance in video games [16], which made windowing a suboptimal option. Informally, when running with window-based segmentation, we were also able to observe responsiveness degradation. Moreover, we had observed this issue in prior research and it became part of what motivated us to invent Machete.

We also note that performance can be impacted by gesture class similarity, where similar gestures may produce more candidates relative to when there is greater separation between gesture classes. Our Kinect dataset has several gestures that are related such as the left hook, left uppercut, left-right-left combo, push, and backflip, as well as the right handed variants. All of these require significant forward left arm movement. Yet, despite their similiarity, we find that Machete maintains a high level of performance throughout all of our evaluations. For these reasons, we believe practitioners and designers engaged in continuous custom gesture recognition work will be able to take advantage of our segmenter to improve both recognition accuracy and latency.

9.1 Does Machete Adhere to $-family Principles

Machete uses many techniques that have already been vetted by the community. To ensure that Machete adheres to those design guidelines described early on, we only need to look at our specific changes. We first spatially resample training samples using angular DP. DP line simplification had already been used in prior rapid prototyping work for gesture recognition [26]. Internally, DP decides on split points by a distance measure, which we now weight by the angle formed by the split. This is a straightforward geometry exercise handled with very little code.

Next, Machete uses CDP to identify candidate gesture patterns in continuous data. CDP is identical to DTW except that the first column of each new matrix row is constant, rather than infinite. In this work, we provide reasonable values one can use for various input devices types, and DTW has already been used in Jackknife [74]. That is, we incur no additional complexity in this regard. We also introduced a new local cost function that weights the squared inner product by the input vector length. Direction vectors and length are nothing new in the $-family literature [72, 74, 76], though. Perhaps the most complex aspect of our approach is that we normalize the accumulated warping path cost by the total gesture path length. However, this only means that one must propagate accumulated path lengths through the matrix as paths are extended. Last, the correction factors we introduce are, in our opinion, no more difficult to understand or implement than those introduced by Jackknife [74] and may be customized according to designer needs.

Machete also uses a straightforward data representation—direction vectors. Although raw input is often not spatial data, we can easily think about trajectories through multidimensional spaces. For example, one uses quaternions to represent 3D orientation with four components, but quaternions are not always considered to be an intuitive construct. Despite this issue, we can still understand trajectories through a four dimensional space and a trajectory's change in position over, over time. This approach is why Jackknife works with accelerometer, quaternion, and binned sound wave data, and is why we chose to use direction vectors in Machete.

9.2 Limitations and Future Work

Presently, Machete is unable to localize end points that occur within the middle of long straight lines. If one produces a right arrow with an exceptionally long shaft, Machete will incorrectly estimate its end points as being near deviations from the template. Often such deviations naturally occur on gesture boundaries, but not always. Our correction factors help localize end points relative to the associated start, but when this estimate is wrong, the error may propagate to the end point as well. One solution may be to include a post processing step that refines segmentation, but we leave this to future work. Another issue is that Machete is not orientation invariant, though we do not intend to address this limitation. Because Machete is sufficiently fast, one can simply add rotated samples to the training set, or depending on the target environment, orientation can be handled elsewhere within the application.

Another important question is how can we improve Machete so that it does not depend on an underlying recognizer, because an obvious benefit in independence is that we reduce overall system complexity. We found that our approach is very effective at segmenting continuous input, but its measurement strategy cannot sufficiently separate gesture classes, and this is why we require external validation. We believe the underlying issue relates to elasticity. Recognizers like Jackknife that work on segmented input are able to put sequences into direct correspondence over both space and time because they resample the input and limit warping, whereas Machete does neither. As part of our future work, we plan to address this situation, perhaps by using correction factors that evolve over time and help prevent pathological warping.

We have also argued based on prior work that Machete adheres to \$-family principles and that it can be used for rapid prototyping. We acknowledge that our view is subjective and some techniques in this as well as other work may be more complex than we first thought, or that such techniques are only accessible to those with a few years of experience in computer science, engineering, mathematics, or other related fields. To understand what rapid prototyping appropriate means will require an in depth study of techniques presented to those of varying backgrounds in the HCI community, and perhaps the development of a new measure, which we leave for future work.

9.3 What's In a Name?

A machete is a coarse cutting tool often used in tropical settings to remove unwanted overgrowth and shrubbery. It requires little training to use: one simply swings forcefully at a target and is left with clean-cut greenery. In much the same way, our segmentation method requires little background knowledge or gesture recognition experience. A practitioner can simply swing Machete at their continuous data and be left with well-segmented data that any high precision recognizer can thereafter evaluate.

10 CONCLUSION

We have presented Machete, a reliable and fast segmentation technique for custom gestures. Through an extensive evaluation involving difficult HA data across a variety of input devices, we found that Machete's accuracy is competitive with other commonly used approaches and often performs better. However, Machete especially shines with respect to performance. Because of Machete's discriminatory power and simple pruning rules, it reduces the computational burden by as much as 98%. Further, our technique is straightforward, being a CDP variant with several novel improvements, which allows for quick integration and rapid prototyping. Though more importantly, because of its high accuracy, one can use Machete to support custom interface design for continuous input device types. As such, we believe Machete will serve a need for those who require fast, custom gesture segmentation.

APPENDIX

A PSEUDOCODE

In this section, we provide pseudocode for Machete. Note that indexing starts at zero, and $\langle\cdot\rangle$ and $||\cdot||$, respectively, denote the inner product and vector length operators. Throughout our pseudocode, *template* refers to a single object created by Initialize-Template that we use to cache template, segmentation, and culling data. In practice, the input buffer instantiated and used by this object is shared between all instances. Further, in our testing, $\epsilon = 0.01$, $\theta = 20$ degrees for mouse gestures, and $\theta = 40$ for full body gestures.

```
INITIALIZE-TEMPLATE($samplePts$, $\theta$, $\epsilon$)
/*** Initialize house keeping data. Note that $row$ is a circular buffer over two CDP$_{ip}$ rows. ***/
$template.buffer \leftarrow []$
$template.row \leftarrow [[], []]$
$template.\vec{T} = []$
$template.startFrame \leftarrow 0$
$template.endFrame \leftarrow 0$
$template.currRowIdx \leftarrow 0$
$template.s1 \leftarrow 0$
$template.s2 \leftarrow 0$
$template.s3 \leftarrow 0$

/*** Resample using Angular DP and build template and create initial CDP$_{ip}$ matrix. ***/
$\boldsymbol{pts} \leftarrow$ ANGULAR-DP ($samplePts$, $\epsilon$)
$N \leftarrow |\boldsymbol{pts}|$
for $i \leftarrow 0$ to $N - 1$ do
    $elem.startFrame \leftarrow -1$
    $elem.endFrame \leftarrow -1$
    $elem.cumulativeCost \leftarrow 0$
    $elem.cumulativeLength \leftarrow 0$
    $elem.score \leftarrow \infty$
    if $i = 0$ then $elem.score \leftarrow (1 - \cos(\theta))^2$

    PUSH-BACK($template.row[0]$, $elem$)
    PUSH-BACK($template.row[1]$, $elem$)

    if $i > 0$ then
        $\vec{\boldsymbol{v}} \leftarrow \boldsymbol{pts}[i] - \boldsymbol{pts}[i-1]$
        PUSH-BACK$\left(template.\vec{T},\ \frac{\vec{\boldsymbol{v}}}{\|\vec{\boldsymbol{v}}\|}\right)$

/*** Calculate correction factor values and weights. ***/
$\vec{\boldsymbol{f2l}} \leftarrow \boldsymbol{pts}[N-1] - \boldsymbol{pts}[0]$
$diagLength \leftarrow$ DIAGONAL-LENGTH($\boldsymbol{pts}$)
$length \leftarrow$ PATH-LENGTH($\boldsymbol{pts}$)
$template.\vec{\boldsymbol{f2l}} \leftarrow \frac{\vec{\boldsymbol{f2l}}}{\|\vec{\boldsymbol{f2l}}\|}$
$template.openness \leftarrow \frac{\|\vec{f2l}\|}{length}$
$template.w_{closedness} \leftarrow 1 - \frac{\|\vec{\boldsymbol{f2l}}\|}{diagLength}$
$template.w_{f2l} \leftarrow \min\left(1,\ 2\frac{\|\vec{\boldsymbol{f2l}}\|}{diagLength}\right)$
return $template$
```

```
ANGULAR-DP ($trajectory$, $\epsilon$)
/*** Determine threshold that stops recursion. ***/
$diagLength \leftarrow$ DIAGONAL-LENGTH($trajectory$)
$\epsilon \leftarrow diagLength \times \epsilon$

/*** Recursively find most descriptive points. ***/
$newPts \leftarrow \{\}$
$N \leftarrow |trajectory|$
PUSH-BACK($newPts$, $trajectory[0]$)
ANGULAR-DP-RECURSIVE($trajectory, 0, N-1, newPts, \epsilon$)
PUSH-BACK($newPts$, $trajectory[N-1]$)
return $newPts$
```

```
ANGULAR-DP-RECURSIVE ($trajectory, start, end, newPts, \epsilon$)
/*** Base case.                                                                  ***/
if $start + 1 \geq end$ then return

/*** Calculate distance from every point in [start+1, end-1]                      ***/
/*** to the line defined by trajectory[start] to trajectory[end].                 ***/
$\vec{AB} \leftarrow trajectory[end] - trajectory[start]$
$denom \leftarrow \left\langle \vec{AB}, \vec{AB} \right\rangle$
if $denom = 0$ then return

$largest \leftarrow \epsilon$
$selected \leftarrow -1$

for $idx \leftarrow start + 1$ to $end - 1$ do
    /*** Project point onto line segment AB.                                      ***/
    $\vec{AC} \leftarrow trajectory[idx] - trajectory[start]$
    $numer \leftarrow \left\langle \vec{AB}, \vec{AC} \right\rangle$
    $d2 \leftarrow \left\langle \vec{AC}, \vec{AC} \right\rangle - (numer^2 / denom)$

    /*** Get vectors made by end points and this point.                           ***/
    $\vec{v1} \leftarrow trajectory[idx] - trajectory[start]$
    $\vec{v2} \leftarrow trajectory[end] - trajectory[idx]$
    $l1 \leftarrow \|v1\|$
    $l2 \leftarrow \|v2\|$
    if $l1 \times l2 = 0$ then continue

    /*** Calculate weighted distance and save if it's the best so far.            ***/
    $d \leftarrow \left\langle \vec{v1}, \vec{v2} \right\rangle / (l1 \times l2)$
    $distance \leftarrow d2 \times \text{ACOS}(d) / \pi$
    if $distance \geq largest$ then
        $largest \leftarrow distance$
        $selected \leftarrow idx$

if $selected = -1$ then return

/*** If we split the subsequence, then recurse into each half. Also save the split point.   ***/
ANGULAR-DP-RECURSIVE($trajectory, start, selected, newPts, \epsilon$)
PUSH-BACK($newPts, trajectory[selected]$)
ANGULAR-DP-RECURSIVE($trajectory, selected, end, newPts, \epsilon$)
```

```
CALCULATE-CORRECTION-FACTORS($template, cdpElem$)
/*** Calculate the first-to-last vector, then the closeness and first-to-last correction factors.   ***/
$\vec{f2l} \leftarrow template.buffer[cdpElem.endFrame] - template.buffer[cdpElem.startFrame]$
$f2lLength \leftarrow \|\vec{f2l}\|$
$openness \leftarrow f2lLength \,/\, cdpElem.cumulativeLength$

$cf_{openness} \leftarrow 1 + template.w_{closedness} \times \left( \frac{\max(openness,\ template.openness)}{\min(openness,\ template.openness)} - 1 \right)$
$cf_{openness} = \min\left(2,\ cf_{openness}\right)$

$cf_{f2l} \leftarrow 1 + \frac{1}{2} \times template.w_{f2l} \times \left( 1 - \left\langle \frac{\vec{f2l}}{f2lLength},\ template.f2l \right\rangle \right)$
$cf_{f2l} = \min\left(2,\ cf_{f2l}\right)$
return $\left(cf_{openness} \times cf_{f2l}\right)$
```

```
CONSUME-INPUT ($template, \boldsymbol{x}, frameNumber$)
/*** Add input to buffer.                                                                  ***/
PUSH-BACK($template.buffer,\ \boldsymbol{x}$)

/*** Filter out inconsequential movement. We use $\delta = 1$ for mouse data and zero otherwise.   ***/
$length \leftarrow \|\boldsymbol{x} - template.\boldsymbol{prev}\|$
if $length < \delta$ then return

/*** Convert to direction vector.                                                          ***/
$\vec{\boldsymbol{x}} \leftarrow (\boldsymbol{x} - template.\boldsymbol{prev})\ /\ length$
$template.\boldsymbol{prev} \leftarrow \boldsymbol{x}$

/*** We store two rows of matrix data, accessed as a circular buffer.                      ***/
$prevRow \leftarrow template.row\,[template.currRowIdx]$
$template.currRowIdx \leftarrow (template.currRowIdx + 1) \mod 2$
$currRow \leftarrow template.row\,[template.currRowIdx]$
$currRow\,[0]\,.startFrame \leftarrow frameNumber$

/*** Update current row with new input.                                                    ***/
$\vec{T} \leftarrow template.\vec{T}$
$T_N \leftarrow \left|\vec{T}\right|$

for $col \leftarrow 1$ to $T_N$ do

    /*** Determine which one of three paths to extend.                                     ***/
    $best \leftarrow currRow\,[col - 1]$
    $path2 \leftarrow prevRow\,[col - 1]$
    $path3 \leftarrow prevRow\,[col]$

    if $path2.score \leq best.score$ then $best \leftarrow path2$
    if $path3.score \leq best.score$ then $best \leftarrow path3$

    /*** Extend selected path through current column.                                      ***/
    $localCost \leftarrow length \times \left(1 - \left\langle \vec{\boldsymbol{x}}, \vec{\boldsymbol{t}}[col-1] \right\rangle\right)^2$
    $currRow[col].startFrame \leftarrow best.startFrame$
    $currRow[col].endFrame \leftarrow frameNumber$
    $currRow[col].cumulativeCost \leftarrow best.cumulativeCost + localCost$
    $currRow[col].cumulativeLength \leftarrow best.cumulativeLength + length$
    $currRow[col].score \leftarrow currRow[col].cumulativeCost\ /\ currRow[col].cumulativeLength$

$cf \leftarrow$ CALCULATE-CORRECTION-FACTORS($template,\ currRow\,[T_N]$)
$correctedScore \leftarrow cf \times currRow\,[T_N]\,.score$

/*** Determine if we should call underlying recognizer.                                    ***/
$template.doCheck \leftarrow false$
$template.total \leftarrow template.total + currRow\,[T_N]\,.score$
$template.n \leftarrow template.n + 1$
$template.s1 \leftarrow template.s2$
$template.s2 \leftarrow template.s3$
$template.s3 \leftarrow correctedScore$

/*** If new low, save segmentation information.                                            ***/
if $template.s3 < template.s2$ then
    $template.startFrame = currRow\,[T_N]\,.startFrame$
    $template.endFrame = currRow\,[T_N]\,.endFrame$
    return

/*** If previous frame is a minimum below the threshold, trigger check                     ***/
$\mu = template.total\ /\ (2 \times template.n)$
$template.doCheck \leftarrow (template.s2 < \mu$ and $template.s2 < template.s1$ and $template.s2 < template.s3)$
```

REFERENCES

[1] Jonathan Alon, Vassilis Athitsos, Quan Yuan, and Stan Sclaroff. 2009. A unified framework for gesture recognition and spatiotemporal gesture segmentation. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 31, 9 (2009), 1685–1699.

[2] Lisa Anthony, YooJin Kim, and Leah Findlater. 2013. Analyzing user-generated YouTube videos to understand touchscreen use by people with motor impairments. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems.* ACM, 1223–1232.

[3] Lisa Anthony and Jacob O. Wobbrock. 2010. A lightweight multistroke recognizer for user interface prototypes. In *Proceedings of the 2010 Graphics Interface (GI'10).* Canadian Information Processing Society, Toronto, Ontario, Canada, 245–252.

[4] Lisa Anthony and Jacob O. Wobbrock. 2012. $N-protractor: A fast and accurate multistroke recognizer. In *Proceedings of the 2012 Graphics Interface (GI'12).* Canadian Information Processing Society, Toronto, Ontario, Canada, 117–120.

[5] Robert Arn, Pradyumna Narayana, Bruce Draper, Tegan Emerson, Michael Kirby, and Chris Peterson. 2018. Motion segmentation via generalized curvatures. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 41, 12 (2018), 2919–2932.

[6] Gustavo E. A. P. A. Batista, Xiaoyue Wang, and Eamonn J. Keogh. 2011. A complexity-invariant distance measure for time series. In *Proceedings of the 11th SIAM International Conference on Data Mining.* 699–710.

[7] Rachel Blagojevic, Samuel Hsiao-Heng Chang, and Beryl Plimmer. 2010. The power of automatic feature selection: Rubine on steroids. In *Proceedings of the 7th Sketch-Based Interfaces and Modeling Symposium.* Eurographics Association, 79–86.

[8] Aaron Bobick and James Davis. 1996. Real-time recognition of activity using temporal templates. In *Proceedings of the 3rd IEEE Workshop on Applications of Computer Vision.* IEEE, 39–42.

[9] Aaron F. Bobick and Andrew D. Wilson. 1995. A state-based technique for the summarization and recognition of gesture. In *Proceedings of the 5th International Conference on Computer Vision.* IEEE, 382–388.

[10] Lee W. Campbell and Aaron F. Bobick. 1995. Recognition of human body motion using phase space constraints. In *Proceedings of the 5th International Conference on Computer Vision.* IEEE, 624–630.

[11] F. M. Caputo, S. Burato, G. Pavan, T. Voillemin, H. Wannous, J. P. Vandeborre, M. Maghoumi, E. M. Taranta, A. Razmjoo, J. J. LaViola Jr, F. Manganaro, S. Pini, G. Borghi, R. Vezzani, R. Cucchiara, H. Nguyen, M. T. Tran, and A. Giachetti. 2019. Online gesture recognition. In *Proceedings of the Eurographics Workshop on 3D Object Retrieval.*

[12] Fabio Marco Caputo, Pietro Prebianca, Alessandro Carcangiu, Lucio D. Spano, and Andrea Giachetti. 2017. A 3 cent recognizer: Simple and effective retrieval and classification of mid-air gestures from single 3D traces. In *Proceedings of the Conference on Smart Tools and Apps for Graphics.* Eurographics Association.

[13] Géry Casiez, Nicolas Roussel, and Daniel Vogel. 2012. 1€ filter: A simple speed-based low-pass filter for noisy input in interactive systems. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems.* ACM, 2527–2530.

[14] Edwin Chan, Teddy Seyed, Wolfgang Stuerzlinger, Xing-Dong Yang, and Frank Maurer. 2016. User elicitation on single-hand microgestures. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems.* ACM, 3403–3414.

[15] Salman Cheema, Michael Hoffman, and Joseph J. LaViola Jr. 2013. 3D gesture classification with linear acceleration and angular velocity sensing devices for video games. *Entertainment Computing* 4, 1 (2013), 11–24.

[16] Jessie Y. C. Chen and Jennifer E. Thropp. 2007. Review of low frame rate effects on human performance. *IEEE Transactions on Systems, Man, and Cybernetics-Part A: Systems and Humans* 37, 6 (2007), 1063–1076.

[17] Yineng Chen, Xiaojun Su, Feng Tian, Jin Huang, Xiaolong Luke Zhang, Guozhong Dai, and Hongan Wang. 2016. Pactolus: A method for mid-air gesture segmentation within EMG. In *Proceedings of the 2016 CHI Conference Extended Abstracts on Human Factors in Computing Systems.* ACM, 1760–1765.

[18] Kyunghyun Cho, Bart van Merriënboer, Caglar Gülçehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. 2014. Learning phrase representations using RNN encoder–decoder for statistical machine translation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP'18).* Association for Computational Linguistics, Doha, Qatar, 1724–1734. Retrieved from http://www.aclweb.org/anthology/D14-1179.

[19] Mark Claypool, Kajal Claypool, and Feissal Damaa. 2006. The effects of frame rate and resolution on users playing first person shooter games. In *Multimedia Computing and Networking 2006.* Vol. 6071. International Society for Optics and Photonics, 607101.

[20] M. Devanne, H. Wannous, S. Berretti, P. Pala, M. Daoudi, and A. Del Bimbo. 2015. 3-D human action recognition by shape analysis of motion trajectories on riemannian manifold. *IEEE Transactions on Cybernetics* 45, 7 (July 2015), 1340–1352.

[21] David H. Douglas and Thomas K. Peucker. 1973. Algorithms for the reduction of the number of points required to represent a digitized line or its caricature. *Cartographica: The International Journal for Geographic Information and Geovisualization* 10, 2 (1973), 112–122.

[22] Abdallah El Ali, Johan Kildal, and Vuokko Lantz. 2012. Fishing or a Z? Investigating the effects of error on mimetic and alphabet device-based gesture interaction. In *Proceedings of the 14th ACM International Conference on Multimodal Interaction.* 93–100.

[23] Chris Ellis, Syed Zain Masood, Marshall F. Tappen, Joseph J. Laviola, Jr., and Rahul Sukthankar. 2013. Exploring the trade-off between accuracy and observational latency in action recognition. *International Journal of Computer Vision* 101, 3 (Feb. 2013), 420–436.

[24] Sergio Escalera, Vassilis Athitsos, and Isabelle Guyon. 2017. Challenges in multi-modal gesture recognition. In *Gesture Recognition.* Springer, 1–60.

[25] Milton Friedman. 1937. The use of ranks to avoid the assumption of normality implicit in the analysis of variance. *Journal of the American Statistical Association* 32, 200 (1937), 675–701.

[26] Vittorio Fuccella and Gennaro Costagliola. 2015. Unistroke gesture recognition through polyline approximation and alignment. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems.* ACM, 3351–3354.

[27] Wayne D. Gray and Deborah A. Boehm-Davis. 2000. Milliseconds matter: An introduction to microstrategies and to their use in describing and predicting interactive behavior. *Journal of Experimental Psychology: Applied* 6, 4 (2000), 322.

[28] James Herold and Thomas F. Stahovich. 2012. The 1¢ recognizer: A fast, accurate, and easy-to-implement handwritten gesture recognition technique. In *Proceedings of the International Symposium on Sketch-Based Interfaces and Modeling.* Eurographics Association, 39–46.

[29] Sture Holm. 1979. A simple sequentially rejective multiple test procedure. *Scandinavian Journal of Statistics* 6, 2 (1979), 65–70.

[30] J. Hu, W. Zheng, J. Lai, and J. Zhang. 2017. Jointly learning heterogeneous features for RGB-D activity recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 39, 11 (Nov 2017), 2186–2200. DOI: https://doi.org/10.1109/TPAMI.2016.2640292

[31] Jian-Fang Hu, Wei-Shi Zheng, Lianyang Ma, Gang Wang, and Jianhuang Lai. 2016. Real-time RGB-D activity prediction by soft regression. In *Proceedings of the European Conference on Computer Vision (ECCV'16)*, Bastian Leibe, Jiri Matas, Nicu Sebe, and Max Welling (Eds.). Springer International Publishing, Cham, 280–296.

[32] Yoonho Hwang, Bohyung Han, and Hee-Kap Ahn. 2012. A fast nearest neighbor search algorithm by nonlinear embedding. In *Proceedings of the 2012 IEEE Conference on Computer Vision and Pattern Recognition (CVPR'12).* 3053–3060.

[33] Yoonho Hwang and Hee kap Ahn. 2011. Convergent bounds on the Euclidean distance. In *Proceedings of the Advances in Neural Information Processing Systems*, J. Shawe-taylor, R. S. Zemel, P. Bartlett, F. C. N. Pereira, and K. Q. Weinberger (Eds.). 388–396.

[34] Holger Junker, Oliver Amft, Paul Lukowicz, and Gerhard Tröster. 2008. Gesture spotting with body-worn inertial sensors to detect user activities. *Pattern Recognition* 41, 6 (2008), 2010–2024.

[35] Kanav Kahol, Priyamvada Tripathi, and Sethuraman Panchanathan. 2004. Automated gesture segmentation from dance sequences. In *Proceedings of the 6th IEEE International Conference on Automatic Face and Gesture Recognition.* IEEE, 883–888.

[36] Hyun Kang, Chang Woo Lee, and Keechul Jung. 2004. Recognition-based gesture spotting in video games. *Pattern Recognition Letters* 25, 15 (2004), 1701–1714.

[37] Maria Karam and M. C. Schraefel. 2006. Investigating user tolerance for errors in vision-enabled gesture-based interactions. In *Proceedings of the Working Conference on Advanced Visual Interfaces.* 225–232.

[38] Eamonn Keogh, Selina Chu, David Hart, and Michael Pazzani. 2004. Segmenting time series: A survey and novel approach. In *Data Mining in Time Series Databases.* World Scientific, 1–21.

[39] Eamonn Keogh and Chotirat Ann Ratanamahatana. 2005. Exact indexing of dynamic time warping. *Knowledge and Information Systems* 7, 3 (2005), 358–386.

[40] Jungsoo Kim, Jiasheng He, Kent Lyons, and Thad Starner. 2007. The gesture watch: A wireless contact-free gesture based wrist interface. In *Proceedings of the 2007 11th IEEE International Symposium on Wearable Computers.* IEEE, 15–22.

[41] Sven Kratz and Michael Rohs. 2010. The $3 recognizer: Simple 3D gesture recognition on mobile devices. In *Proceedings of the 15th International Conference on Intelligent User Interfaces.* ACM, 419–420.

[42] Sven Kratz and Michael Rohs. 2011. Protractor3D: A closed-form solution to rotation-invariant 3D gestures. In *Proceedings of the 16th International Conference on Intelligent User Interfaces.* ACM, 371–374.

[43] Per Ola Kristensson, Thomas Nicholson, and Aaron Quigley. 2012. Continuous recognition of one-handed and two-handed gestures using 3D full-body motion tracking sensors. In *Proceedings of the 2012 ACM International Conference on Intelligent User Interfaces.* ACM, 89–92.

[44] Eyal Krupka, Kfir Karmon, Noam Bloom, Daniel Freedman, Ilya Gurvich, Aviv Hurvitz, Ido Leichter, Yoni Smolin, Yuval Tzairi, Alon Vinnikov, and Aharon Bar Hillel. 2017. Toward realistic hands gesture Interface: Keeping it simple for developers and machines. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*. ACM, 1887–1898.

[45] Joseph J. LaViola. 2003. Double exponential smoothing: An alternative to Kalman filter-based predictive tracking. In *Proceedings of the Workshop on Virtual Environments 2003*. ACM, 199–206.

[46] Yang Li. 2010. Protractor: A fast and accurate gesture recognizer. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI'10)*. ACM, New York, NY, 2169–2172.

[47] Jiayang Liu, Lin Zhong, Jehan Wickramasuriya, and Venu Vasudevan. 2009. uWave: Accelerometer-based personalized gesture recognition and its applications. *Pervasive and Mobile Computing* 5, 6 (2009), 657–675.

[48] Granit Luzhnica, Jorg Simon, Elisabeth Lex, and Viktoria Pammer. 2016. A sliding window approach to natural hand gesture recognition using a custom data glove. In *Proceedings of the 2016 IEEE Symposium on 3D User Interfaces*. IEEE, 81–90.

[49] Mehran Maghoumi and Joseph J. LaViola. 2019. DeepGRU: Deep gesture recognition utility. In *Proceedings of the International Symposium on Visual Computing*. 16–31.

[50] Thomas M. Mitchell. 1997. *Machine Learning* (1 ed.). McGraw-Hill, Inc., New York, NY.

[51] Meredith Ringel Morris, Jacob O. Wobbrock, and Andrew D. Wilson. 2010. Understanding users' preferences for surface gestures. In *Proceedings of the 2010 Graphics Interface*. Canadian Information Processing Society, 261–268.

[52] Miguel A. Nacenta, Yemliha Kamber, Yizhou Qiang, and Per Ola Kristensson. 2013. Memorability of pre-designed and user-defined gesture sets. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 1099–1108.

[53] Pedro Neto, Dário Pereira, J. Norberto Pires, and A. Paulo Moreira. 2013. Real-time and continuous hand gesture spotting: An approach based on artificial neural networks. In *Proceedings of the 2013 IEEE International Conference on Robotics and Automation*. IEEE, 178–183.

[54] Yuan Niu and Hao Chen. 2011. Gesture authentication with touch input for mobile devices. In *Proceedings of the International Conference on Security and Privacy in Mobile Information and Communication Systems*. Springer, 13–24.

[55] Juliet Norton, Chadwick A. Wingrave, and Joseph J. LaViola Jr. 2010. Exploring strategies and guidelines for developing full body video game interfaces. In *Proceedings of the 5th International Conference on the Foundations of Digital Games*. ACM, 155–162.

[56] LLC Oculus VR. 2017. *Best Practices*. Thorlabs.

[57] Uran Oh and Leah Findlater. 2013. The challenges and potential of end-user gesture customization. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 1129–1138.

[58] Ryuichi Oka. 1998. Spotting method for classification of real world data. *Computer Journal* 41, 8 (1998), 559–565.

[59] Corey Pittman, Pamela Wisniewski, Conner Brooks, and Joseph J. LaViola Jr. 2016. Multiwave: Doppler effect based gesture recognition in multiple dimensions. In *Proceedings of the 2016 CHI Conference Extended Abstracts on Human Factors in Computing Systems*. ACM, 1729–1736.

[60] Thanawin Rakthanmanon, Bilson Campana, Abdullah Mueen, Gustavo Batista, Brandon Westover, Qiang Zhu, Jesin Zakaria, and Eamonn Keogh. 2012. Searching and mining trillions of time series subsequences under dynamic time warping. In *Proceedings of the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 262–270.

[61] Chotirat Ann Ratanamahatana and Eamonn Keogh. 2005. Three myths about dynamic time warping data mining. In *Proceedings of the 2005 SIAM International Conference on Data Mining*. SIAM, 506–510.

[62] Dean Rubine. 1991. *The Automatic Recognition of Gestures*. Ph.D. Dissertation. Citeseer.

[63] Dean Rubine. 1991. Specifying gestures by example. *SIGGRAPH Computer Graphics* 25, 4 (July 1991), 329–337.

[64] Hiroaki Sakoe and Seibi Chiba. 1978. Dynamic programming algorithm optimization for spoken word recognition. *IEEE Transactions on Acoustics, Speech, and Signal Processing* 26, 1 (1978), 43–49.

[65] Yasushi Sakurai, Christos Faloutsos, and Masashi Yamamuro. 2007. Stream monitoring under the time warping distance. In *Proceedings of the 2007 IEEE 23rd International Conference on Data Engineering*. IEEE, 1046–1055.

[66] Junjie Shan and Srinivas Akella. 2014. 3D human action segmentation and recognition using pose kinetic energy. In *Proceedings of the 2014 IEEE Workshop on Advanced Robotics and Its Social Impacts (ARSO'14)*. IEEE, 69–75.

[67] Rim Slama, Hazem Wannous, Mohamed Daoudi, and Anuj Srivastava. 2015. Accurate 3D action recognition using learning on the Grassmann Manifold. *Pattern Recognition* 48, 2 (Feb. 2015), 556–567. DOI: https://doi.org/10.1016/j.patcog.2014.08.011

[68] Steven W. Smith. 1997. *The Scientist and Engineer's Guide to Digital Signal Processing*. California Technical Publishing, San Diego.

[69] Yale Song, David Demirdjian, and Randall Davis. 2012. Continuous body and hand gesture recognition for natural human-computer interaction. *ACM Transactions on Interactive Intelligent Systems* 2, 1 (2012), 5.

[70] Jingren Tang, Hong Cheng, Yang Zhao, and Hongliang Guo. 2018. Structured dynamic time warping for continuous hand trajectory gesture recognition. *Pattern Recognition* 80 (2018), 21–31. http://www.sciencedirect.com/science/article/pii/S0031320318300621.

[71] Eugene M. Taranta II, Seng Lee Koh, Brian M. Williamson, Kevin P. Pfeil, Corey R. Pittman, and Joseph J. LaViola Jr. 2019. Pitch Pipe: An automatic low-pass filter calibration technique for pointing tasks. In *Proceedings of the 45th Graphics Interface Conference on Proceedings of Graphics Interface 2019*. Canadian Human-Computer Communications Society, 1–8.

[72] Eugene M. Taranta II, Mehran Maghoumi, Corey R. Pittman, and Joseph J. LaViola, Jr. 2016. A rapid prototyping approach to synthetic data generation for improved 2D gesture recognition. In *Proceedings of the 29th Annual Symposium on User Interface Software and Technology (UIST'16)*. ACM, New York, NY, 873–885.

[73] Eugene M. Taranta II, Corey R. Pittman, Jack P. Oakley, Mykola Maslych, Mehran Maghoumi, and Joseph J. LaViola Jr. 2020. Moving toward an ecologically valid data collection protocol for 2D gestures in video games. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*. 1–11.

[74] 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 *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems (CHI'17)*. ACM, New York, NY, 5850–5861.

[75] Eugene M. Taranta II, Thaddeus K. Simons, Rahul Sukthankar, and Joseph J. Laviola Jr. 2015. Exploring the benefits of context in 3D gesture recognition for game-based virtual environments. *ACM Transactions on Interactive Intelligent Systems* 5, 1 (2015), 1.

[76] Eugene M. Taranta II, Andrés N. Vargas, and Joseph J. LaViola Jr. 2016. Streamlined and accurate gesture recognition with Penny Pincher. *Computers & Graphics* 55 (2016), 130–142. http://www.sciencedirect.com/science/article/pii/S0097849315001788.

[77] Radu-Daniel Vatavu. 2017. Improving gesture recognition accuracy on touch screens for users with low vision. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems (CHI'17)*. ACM, New York, NY, 4667–4679.

[78] Radu-Daniel Vatavu, Lisa Anthony, and Jacob O. Wobbrock. 2012. Gestures as point clouds: A $P recognizer for user interface prototypes. In *Proceedings of the 14th ACM International Conference on Multimodal Interaction (ICMI'12)*. ACM, New York, NY, 273–280.

[79] Radu-Daniel Vatavu, Lisa Anthony, and Jacob O. Wobbrock. 2018. $ Q: A super-quick, articulation-invariant stroke-gesture recognizer for low-resource devices. In *Proceedings of the 20th International Conference on Human-Computer Interaction with Mobile Devices and Services*. ACM, 23.

[80] Radu-Daniel Vatavu, Laurent Grisoni, and Stefan-Gheorghe Pentiuc. 2009. Multiscale detection of gesture patterns in continuous motion trajectories. In *Proceedings of the International Gesture Workshop*. Springer, 85–97.

[81] Radu-Daniel Vatavu, Daniel Vogel, Géry Casiez, and Laurent Grisoni. 2011. Estimating the perceived difficulty of pen gestures. In *Proceedings of the 13th International Conference on Human-computer Interaction - Volume Part II (INTERACT'11)*. Springer, Berlin, 89–106.

[82] Duc-Hoang Vo, Huu-Hung Huynh, Thanh-Nghia Nguyen, and Jean Meunier. 2016. Automatic hand gesture segmentation for recognition of Vietnamese sign language. In *Proceedings of the 7th Symposium on Information and Communication Technology*. ACM, 368–373.

[83] Pei Wang, Chunfeng Yuan, Weiming Hu, Bing Li, and Yanning Zhang. 2016. Graph based skeleton motion representation and similarity measurement for action recognition. In *Proceedings of the European Conference on Computer Vision*. Springer, 370–385.

[84] Tian-Shu Wang, Heung-Yeung Shum, Ying-Qing Xu, and Nan-Ning Zheng. 2001. Unsupervised analysis of human gestures. In *Proceedings of the Pacific-Rim Conference on Multimedia*. Springer, 174–181.

[85] Séamas Weech, Sophie Kenny, and Michael Barnett-Cowan. 2019. Presence and cybersickness in virtual reality are negatively related: A review. *Frontiers in Psychology* 10 (2019), 158. https://www.frontiersin.org/article/10.3389/fpsyg.2019.00158.

[86] Daniel Weinland, Remi Ronfard, and Edmond Boyer. 2011. A survey of vision-based methods for action representation, segmentation and recognition. *Computer Vision and Image Understanding* 115, 2 (Feb. 2011), 224–241.

[87] J. Weng, C. Weng, and J. Yuan. 2017. Spatio-temporal Naive-Bayes nearest-neighbor (ST-NBNN) for skeleton-based action recognition. In *Proceedings of the 2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR'17)*. 445–454. DOI: https://doi.org/10.1109/CVPR.2017.55

[88] Jacob O. Wobbrock, Meredith Ringel Morris, and Andrew D. Wilson. 2009. User-defined gestures for surface computing. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 1083–1092.

[89] Jacob O. Wobbrock, Andrew D. Wilson, and Yang Li. 2007. Gestures without libraries, toolkits or training: A $1 recognizer for user interface prototypes. In *Proceedings of the 20th Annual ACM Symposium on User Interface Software and Technology (UIST'07)*. ACM, New York, NY, 159–168.

[90] Di Wu, Fan Zhu, and Ling Shao. 2012. One shot learning gesture recognition from RGBD images. In *Proceedings of the 2012 IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops (CVPRW'12)*. IEEE, 7–12.

[91] L. Xia, C. Chen, and J. K. Aggarwal. 2012. View invariant human action recognition using histograms of 3D joints. In *Proceedings of the 2012 IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops*. 20–27. DOI: https://doi.org/10.1109/CVPRW.2012.6239233

[92] Hee-Deok Yang, Stan Sclaroff, and Seong-Whan Lee. 2009. Sign language spotting with a threshold model based on conditional random fields. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 31, 7 (2009), 1264–1277.

[93] Yina Ye and Petteri Nurmi. 2015. Gestimator: Shape and stroke similarity based gesture recognition. In *Proceedings of the 2015 ACM on International Conference on Multimodal Interaction*. ACM, 219–226.

[94] Ying Yin and Randall Davis. 2013. Gesture spotting and recognition using salience detection and concatenated hidden Markov models. In *Proceedings of the 15th ACM on International Conference on Multimodal Interaction*. ACM, 489–494.

[95] Hansong Zeng and Yi Zhao. 2011. Sensing movement: Microsensors for body motion measurement. *Sensors* 11, 1 (2011), 638–660.

[96] Yaodong Zhang and James R. Glass. 2011. An inner-product lower-bound estimate for dynamic time warping. In *Proceedings of the 2011 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP'11)*. IEEE, 5660–5663.

Received May 2019; revised August 2020; accepted October 2020