# $P Point-Cloud Gesture Recognizer Pseudocode

Radu-Daniel Vatavu
University Stefan cel Mare of Suceava
Suceava 720229, Romania
vatavu@eed.usv.ro

Lisa Anthony
UMBC Information Systems
1000 Hilltop Circle
Baltimore MD 21250
lanthony@umbc.edu

Jacob O. Wobbrock
Information School | DUB Group
University of Washington
Seattle, WA 98195-2840 USA
wobbrock@uw.edu

In the following pseudocode, POINT is a structure that exposes $x$, $y$, and *strokeId* properties. *strokeId* is the stroke index a point belongs to (1, 2, ...) and is filled by counting pen down/up events. POINTS is a list of points and TEMPLATES a list of POINTS with gesture class data.

**Recognizer main function.** Match *points* against a set of *templates* by employing the Nearest-Neighbor classification rule. Returns a normalized score in [0..1] with 1 denoting perfect match.

```
$P-RECOGNIZER (POINTS points, TEMPLATES templates)
n ← 32 // number of points
NORMALIZE(points, n)
score ← ∞
for each template in templates do
    NORMALIZE(template, n) // should be pre-processed
    d ← GREEDY-CLOUD-MATCH(points, template, n)
    if score > d then
        score ← d
        result ← template
score ← MAX((2.0 − score)/2.0, 0.0) // normalize score in [0..1]
return ⟨result, score⟩
```

**Cloud matching function.** Match two clouds (*points* and *template*) by performing repeated alignments between their points (each new alignment starts with a different starting point index $i$). Parameter $\epsilon \in [0..1]$ controls the number of tested alignments ($n^\epsilon \in \{1, 2, ...n\}$). Returns the minimum alignment cost.

```
GREEDY-CLOUD-MATCH (POINTS points, POINTS template, int n)
ϵ ← .50
step ← ⌊n^(1−ϵ)⌋
min ← ∞
for i = 0 to n step step do
    d1 ← CLOUD-DISTANCE(points, template, n, i)
    d2 ← CLOUD-DISTANCE(template, points, n, i)
    min ← MIN(min, d1, d2)
return min
```

**Distance between two clouds.** Compute the minimum-cost alignment between *points* and *tmpl* starting with point *start*. Assign decreasing confidence *weights* $\in [0..1]$ to point matchings.

```
CLOUD-DISTANCE (POINTS points, POINTS tmpl, int n, int start)
matched ← new bool[n]
sum ← 0
i ← start // start matching with points_i
do
    min ← ∞
    for each j such that not matched[j] do
        d ← EUCLIDEAN-DISTANCE(points_i, tmpl_j)
        if d < min then
            min ← d
            index ← j
    matched[index] ← true
    weight ← 1 − ((i − start + n) MOD n)/n
    sum ← sum + weight · min
    i ← (i + 1) MOD n
until i == start // all points are processed
return sum
```

The following pseudocode addresses gesture preprocessing (or normalization) which includes resampling, scaling with shape preservation, and translation to origin. The code is similar to $1 and $N recognizers[1,2] and we repeat it here for completeness. We highlight minor changes.

**Gesture normalization.** Gesture points are resampled, scaled with shape preservation, and translated to origin.

```
NORMALIZE (POINTS points, int n)
points ← RESAMPLE(points, n)
SCALE(points)
TRANSLATE-TO-ORIGIN(points, n)
```

**Points resampling.** Resample a *points* path into $n$ evenly spaced points. We use $n = 32$.

```
RESAMPLE (POINTS points, int n)
I ← PATH-LENGTH(points) / (n − 1)
D ← 0
newPoints ← points_0
for each p_i in points such that i ≥ 1 do
    if p_i.strokeId == p_{i−1}.strokeId then
        d ← EUCLIDEAN-DISTANCE(p_{i−1}, p_i)
        if (D + d) ≥ I then
            q.x ← p_{i−1}.x + ((I − D)/d) · (p_i.x - p_{i−1}.x)
            q.y ← p_{i−1}.y + ((I − D)/d) · (p_i.y - p_{i−1}.y)
            q.strokeId ← p_i.strokeId
            APPEND(newPoints, q)
            INSERT(points, i, q) // q will be the next p_i
            D ← 0
        else D ← D + d
return newPoints
```

```
PATH-LENGTH (POINTS points)
d ← 0
for each p_i in points such that i ≥ 1 do
    if p_i.strokeId == p_{i−1}.strokeId then
        d ← d + EUCLIDEAN-DISTANCE(p_{i−1}, p_i)
return d
```

**Points rescale.** Rescale *points* with shape preservation so that the resulting bounding box will be $\subseteq [0..1] \times [0..1]$.

```
SCALE (POINTS points)
x_min ← ∞, x_max ← 0, y_min ← ∞, y_max ← 0
for each p in points do
    x_min ← MIN(x_min, p.x)
    y_min ← MIN(y_min, p.y)
    x_max ← MAX(x_max, p.x)
    y_max ← MAX(y_max, p.y)
scale ← MAX(x_max − x_min, y_max − y_min)
for each p in points do
    p ← ((p.x − x_min)/scale, (p.y − y_min)/scale, p.strokeId)
```

**Points translate.** Translate *points* to the origin (0, 0).

```
TRANSLATE-TO-ORIGIN (POINTS points, int n)
c ← (0, 0) // will contain centroid
for each p in points do
    c ← (c.x + p.x, c.y + p.y)
c ← (c.x/n, c.y/n)
for each p in points do
    p ← (p.x - c.x, p.y - c.y, p.strokeId)
```

[1] http://depts.washington.edu/aimgroup/proj/dollar/index.html
[2] http://depts.washington.edu/aimgroup/proj/dollar/ndollar.html

This pseudocode is modified slightly from that which appears in the original ACM ICMI 2012 publication by Vatavu, Anthony, and Wobbrock (http://dx.doi.org/10.1145/2388676.2388732) to better highlight the use of the *strokeId* property and to provide a normalized matching score. It also contains more comments to assist the implementation. This algorithm's logic remains unchanged.