

Handwriting Recognition with SQL

and a tiny bit of web stuff

Noah Doersing

noah.doersing@student.uni-tuebingen.de

ABSTRACT

A basic handwriting recognition algorithm conceived in the 60's has been implemented, 50 years later, on top of a modern relational database system. We discuss the origins of the system, introduce the algorithm and its SQL implementation, provide a web app that captures pen strokes and sends them off to the database for processing, evaluate the approach and provide further context.

1 INTRODUCTION

Throughout the history of computing, human-computer interaction has been an ever-evolving avenue for research. The results of computations, in the early days indicated by blinking light bulbs, have more recently been presented on screens of ever-increasing resolution, with different output formats such as voice being explored continuously. On the input side of the equation, punch cards have given way to command prompts fed by keyboards, which in turn have largely been superseded touch- or voice-based user interfaces.

Both need to operate in lockstep: A fine meshing of input and output methods is required if an interface is to feel natural to users. Today's touch-based input devices, when compared to now-old-fashioned mouse input, exhibit clear usability upsides. Whenever the user wishes to manipulate a thing, she can simply touch it, instead of negotiating a mouse pointer towards it or entering keyboard shortcuts.

Current as this may be, it is not fundamentally new. More than 50 years ago, researchers at RAND Corporation came up with a pen-based input paradigm intended, according to GUI pioneer Alan Kay, "for economists and other non-computer specialists [...] who said 'Hey, none of us can type, can't you do something about that?'" [1] As part of this system, a basic handwriting recognition algorithm was developed. [2, 5]

Outline. Before delving into its details – and how a simplified variant of it, only able to recognize a single-stroke letter at a time (see figure 1), has been implemented as a SQL query – we will take a look at the RAND tablet and the GRAIL system it powers. After discussing the implementation, we will consider its performance and some related ideas.

2 GRAIL AND THE RAND TABLET

Developed by RAND in the 1960s, the Graphic Input Language (GRAIL) was a flowchart-based programming system. [9] As described by its creators, it enables a user to "draw flowchart symbols,

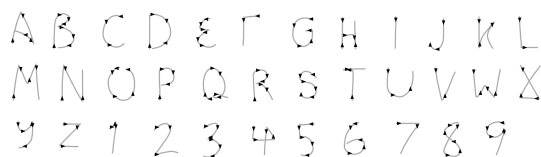


Figure 1: How to draw the simplified characters.

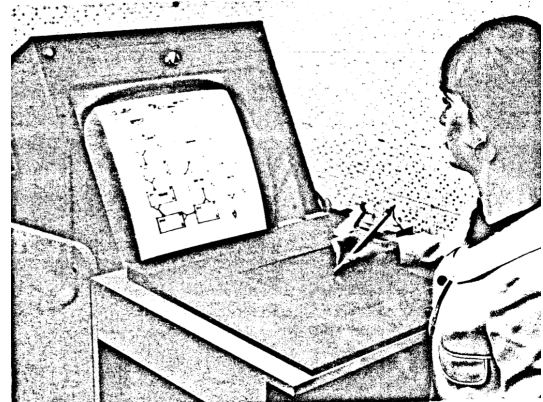


Figure 2: A scan [9] of a printout of a photograph of a user interacting with GRAIL.

edit and rearrange them on the display surface, and connect them appropriately to form a meaningful program. [The user can] execute the program while controlling its execution rate and the amount and content of information presented to him. The system interprets, in real-time, the [user's] hand-drawn figures, characters, and other [...] gestures to provide germane responses on the display surface." [8] Alan Kay, in a talk where he demonstrates GRAIL, summarizes it as "the world's first modeless system" and notes that it is "where Macintosh window control came from" [1].

The user operated the GRAIL system using a stylus connected to a tablet as shown in figure 2. Its "writing surface is a 10"x10" area with a resolution of 100 lines per inch in both x and y [...], allowing the user to 'write' in a natural manner. [...] To maintain 'naturalness' of the pen device, a pressure-sensitive switch in the tip of the stylus indicates 'stroke' [...] to the computer" [6].¹

The original "IBM/360 assembly language program for online graphical-input character recognition using the RAND Tablet" [3] is publicly available, but was not consulted when writing the SQL query – it is too low-level to be of much utility.

2.1 Web App

Testing the SQL query necessitates a simple simulation of the RAND tablet, which was given along with the assignment in the form of a D3.js-based web app that prints a list of coordinates once the user draws a character.

In order to present the algorithm succinctly, this was extended with a server component that, with the help of some Python code, dynamically executes the handwriting recognition query on a Postgres instance and returns the result to the user.²

¹The authors of this memo describe the function of the tablet in great detail. Another memo takes a deep dive into the innards of GRAIL. [10]

²This is given in `code/index.html`. The backend setup is a can of worms, so it's recommended to just use the frontend, which yields a JSON version of the pen stroke.

```

1 WITH RECURSIVE
2 tablet(pos, x, y) -- ... (conversion of JSON list to rows)
3 smooth(pos, x, y) AS (
4   SELECT pos, x :: real, y :: real FROM tablet WHERE pos = 1
5   UNION ALL
6   SELECT
7     t.pos,
8     (:smoothingfactor * s.x + (1.0 - :smoothingfactor) * t.x) :: real AS x,
9     (:smoothingfactor * s.y + (1.0 - :smoothingfactor) * t.y) :: real AS y
10  FROM smooth s, tablet t
11  WHERE t.pos = s.pos + 1
12 ),
13 thin(pos, x, y) AS (
14   SELECT * FROM smooth WHERE pos = 1
15   UNION ALL
16   SELECT *
17   FROM (
18     SELECT s.pos, s.x, s.y
19     FROM thin t, smooth s
20     WHERE s.pos > t.pos
21     AND :thinningsize < || (s.x - t.x)^2 + (s.y - t.y)^2
22     ORDER BY s.pos
23     LIMIT 1
24   ) AS _
25 ),
26 curve(pos, x, y, direction) AS (
27   SELECT pos, x, y,
28     COALESCE(degrees(-atan2(y - lag(y) OVER (ORDER BY pos),
29                               -x + lag(x) OVER (ORDER BY pos))
30       ) + 180, 90)
31 FROM thin
32 ),
33 cardinal(pos, direction) AS (
34   SELECT pos,
35     (enum_range(NULL :: cardinal_direction))[ (direction / 90) :: int % 4 + 1 ]
36 FROM curve
37 ),
38 cardinal_change(pos, direction) AS (
39   SELECT pos, direction
40   FROM (SELECT pos, direction,
41         COALESCE(lag(direction, 2) OVER win <> lag(direction) OVER win,
42               true)
43        AND lag(direction) OVER win = direction
44        FROM cardinal
45        WINDOW win AS (ORDER BY pos)) AS _(pos, direction, is_new)
46 WHERE is_new
47 ),
48 corner(pos, x, y) AS (
49   SELECT pos, x, y, (
50     angdiff(lag(direction, 2) OVER win, lag(direction) OVER win) < 22.5
51     AND angdiff(lag(direction) OVER win, direction) > :cornerangle
52     AND angdiff(direction, lead(direction) OVER win) < 22.5
53   ) OR ( -- Immediate direction change OR one-segment turn.
54     angdiff(lag(direction, 3) OVER win, lag(direction, 2) OVER win) < 22.5
55     AND angdiff(lag(direction, 2) OVER win, direction) > :cornerangle
56     AND angdiff(direction, lead(direction) OVER win) < 22.5
57   ) AS is_corner
58 FROM curve
59 WINDOW win AS (ORDER BY pos)) AS _(pos, x, y, is_corner)
60 WHERE is_corner
61 ),
62 aabb(xmin, xmax, ymin, ymax, aspect, width, height, centerx, centery) -- ...
63 start_grid(n) AS (
64   SELECT gridpos(a.width, a.height, a.xmin, a.ymin, s.x, s.y)
65   FROM smooth s, aabb a
66   ORDER BY s.pos
67   LIMIT 1
68 ),
69 stop_grid(n) -- ... (analogous to start_grid)
70 corner_grid(pos, n) -- ... (also similar)
71 features(directions, start, stop, corners, width, height, aspect, center) AS (
72   SELECT (SELECT array_agg(direction ORDER BY pos)
73         FROM cardinal_change),
74         (TABLE start_grid),
75         (TABLE stop_grid),
76         (SELECT COALESCE(array_agg(n ORDER BY pos), '{}')
77         FROM corner_grid),
78         width, height, aspect, point(centerx, centery)
79 FROM aabb
80 ),
81 character(character) AS (
82   SELECT l.character
83   FROM features f, lookup_candidates c, lookup_bestfit l
84   WHERE f.directions[1:4] = c.first_four_directions
85         AND c.candidate_characters = l.candidate_characters
86         AND (l.start IS NULL OR f.start = l.start)
87         AND (l.stop IS NULL OR f.stop = l.stop)
88         AND (l.corners IS NULL OR f.corners = l.corners)
89         AND (l.aspect_range IS NULL OR l.aspect_range @> f.aspect :: numeric)
90 )
91 TABLE character;

```

Figure 3: Slightly condensed SQL implementation. Some custom functions have been omitted.

3 IMPLEMENTATION

The web app outputs a JSON value of the form [{"x": 37, "y": 39}, {"x": 38, "y": 43, ...}]. It is passed to the Postgres³ command-line client psql and converted to a table with an sequential identifier column in CTE tablet (see figure 3).

3.1 Smoothing & Thinning

The RAND tablet provides a stream (“the recognition scheme is notified of [the pen] position every 4 msec” [2]) of pen positions as integer x, y coordinate pairs.

As a preprocessing step, this digital representation pen stroke is smoothed to counteract quantization effects (“noise due to the discreteness of the pen location as measures by the tablet” [2], which can negatively affect later steps of the algorithm) as shown in figure 4. Smoothing is performed by computing a weighted average of the most recently smoothed point (weight 75%; the first point of the stroke is assumed to be smooth already) and the next incoming point (25%). In SQL, this is accomplished by the recursive CTE smooth where the weight is stored in the psql variable smoothingfactor.

The second stage of preprocessing drops points that are in close proximity to a previous point and thus contribute little or no additional information. This mainly serves to ease the computing effort required during the next stages. Groner uses a set of inequalities that compare x and y positions, effectively superimposing a square over each point and removing any other points that fall into this square. This is less “correct” than pruning based on the Euclidean distance as implemented in the SQL query (line 20), but computationally more efficient. The seemingly superfluous wrapping for the recursive part of the thin CTE in a SELECT * FROM (...) AS _ is required because Postgres does not permit ORDER BY directly in a recursive query.

The degree to which these steps affect the raw data is controlled using the psql variables smoothingfactor and thinningsize. Choosing sensible values is important: As smoothingfactor approaches 1, corners disappear and the end of the stroke is pulled closer to the start. This effect is visible in figure 4 where ● extend further along the x axis than ●. Similarly, a large thinningsize can obscure corners or bends in the stroke.

³Postgres 9.4 or newer is required.

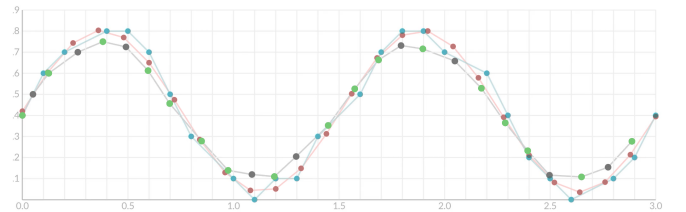


Figure 4: The ● pen stroke drawn by the user, the corresponding ● quantized signal received by the computer, the ● smoothed intermediate result and ● the points left after thinning.

3.2 Curvature

“Curvature is the most obvious characteristic which is independent of position and size, and yet which describes the [stroke]’s shape. [...] Four directions, used in conjunction with other features, provide sufficient description for recognition, yet result in [less effort] than do [more] directions.” [2] Only changes of these cardinal directions the pen stroke moves toward as each character is drawn are kept.

Groner extracts this feature by comparing positional differences of point pairs using a set of inequalities. Because more fine-grained curvature information is helpful in detecting corners later on, the SQL implementation (see CTE curve) instead computes the precise angle between the line connecting each pair of points and the x axis as shown in figure 5.

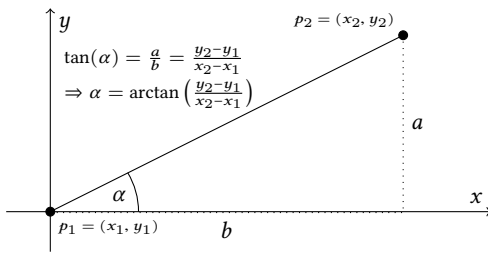


Figure 5: Deriving the angle α between two points p_1 and p_2 .

In CTE cardinal it is then quantized to determine the cardinal directions, denoted by an ENUM type comprising the Unicode symbols $\blacktriangleleft, \blacktriangleright, \blacktriangleup, \blacktriangledown$. To accomplish this, the `enum_range(anyenum)` function is used, which returns all values of the input enum type as an array.⁴

“If the same direction occurs twice in succession, and is not the same as the last direction [...], then it is added to the [result], otherwise it is discarded.” [2] Thus the CTE cardinal_change takes care of discarding sequential duplicates.

TODO because window functions cannot be used in where, an awkward subquery todo “They are forbidden elsewhere, such as in GROUP BY, HAVING and WHERE clauses. This is because they logically execute after the processing of those clauses.”⁵

3.3 Corners

Cardinal directions are sufficient to discern between many characters, but in some cases – see figure 6a – other features such as the presence or absence of corners are required. “A corner is detected whenever the pen moves in the same (± 1) 16-direction for at least two segments [$\blacktriangleright\blacktriangleright$], changes direction by at least 90° , and then proceeds along the new direction (± 1) for at least two segments [$\blacktriangleright\blacktriangleright$]. The change in direction must take place either immediately or through a one-segment turn [$\blacktriangleright\blacktriangledown$].” [2] (The triangles correspond to the lower half of figure 6a.)

Instead of 16-directions (i.e., the division of 360° into 16 slices), the CTE corner of the SQL implementation uses absolute angles, making sure that the two line segments directly before a corner point within $\frac{360^\circ}{16} = 22.5^\circ$ of each other, the two next line segments also point within 22.5° of each other and the angle at the corner

point is $> 90^\circ$. The window function `lag()` is used to access previous and next points – rather, the angles the associated line segments point towards – and a custom `angdiff()` function computes the `delat`

$$\arctan\left(\frac{\sin(\alpha - \beta)}{\cos(\alpha - \beta)}\right)$$

between a pair of angles.⁶ The output of the CTE is a table of corner positions.

3.4 Additional Features

In addition to cardinal directions and corners, Groner’s algorithm determines a number of additional features: “the symbol’s height and width [...], its aspect ratio [...], and its center relative to the tablet origin.” [2] They are of limited use to the simplified single-stroke alphanumeric characters supported by the SQL implementation; e.g., height and width aid in discerning parentheses from commas.

Nevertheless, they are extracted in the CTE `aabb`, which trivially derives an axis-aligned bounding box around the character drawn by the user and returns two x, y pairs defining it, along with width, height, center and aspect ratio.

Grid. The algorithm then “divides the rectangular area defined by the symbol into a 4×4 grid. The starting (pen-down) and ending (pen-up) points, as well as the corner locations, are then each encoded as lying in one of these 16 areas, thereby locating them relative to the symbol.” [2] Figure 6b shows an example.

The SQL implementation uses a custom `gridpos(width, height, xmin, ymin, x, y)` function which “desugars” to an arithmetic expression mapping x and y , given an AABB defined by minimum x , y , width and height to grid positions.⁷

Finally, the CTE features combines all extracted features into a single-row table.

⁶See <https://stackoverflow.com/a/2007279>. The formula $180 - |\alpha - \beta| - 180$ is equivalent and cheaper to compute, but the one given in the text elegantly combines `sin`, `cos` and `arctan`.

⁷Global state would be convenient here to avoid passing the unchanging AABB on each call, but creating an extra temporary table for storing it seems like overkill.

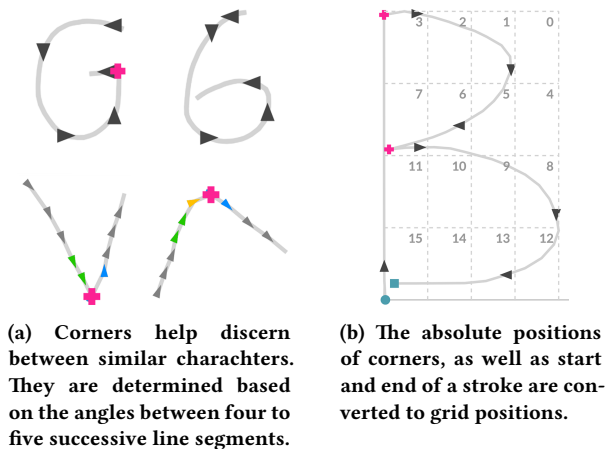


Figure 6: Cardinal directions, corners and the grid.

⁴See <https://www.postgresql.org/docs/8.3/static/functions-enum.html>.

⁵See <https://www.postgresql.org/docs/10/static/tutorial-window.html>.

3.5 Mapping Features to Characters

Once the pen stroke has been sufficiently analyzed, Groner’s algorithm “[looks] up the first four direction segments” in a table. For the letter B as shown in figure 6b, this table entry is $\blacktriangle\blacktriangleright\blacktriangledown\blacktriangleleft$, which corresponds to 2, 3, 8, B, D, P, or R. “Further tests, based on this particular set of possibilities, result in a single identification.” [2]

In the original implementation, these tests take the form of a custom decision tree for each entry of the initial lookup table. A direct transliteration to SQL is difficult, requiring a rat’s tail of nested CASE expressions or a similarly convoluted approach. However, it turned out that the simplified alphabet used in the context of this work suggests a more idiomatic two-stage approach that, while less flexible, delivers acceptable results:

- (1) An array comprising the first four cardinal directions is mapped to an array of possible characters using lookup table T_1 .
- (2) Another lookup table T_2 is consulted that maps the potential characters and specific feature values – some of which can be marked as “don’t care” by setting them to NULL (hence the IS NULL checks in lines 86 through 89) – to a concrete result character. For characters which are wholly determined by the first lookup, all features are set to NULL.

Finding a satisfactory representation for this decision mechanism was one of the trickier aspects of this work: A reasonable balance between flexibility and maintainability needed to be struck while ideally still resembling Groner’s ideas.

Alternate approach. The two-stage approach implemented in CTE character is not ideal: The array-to-array mapping of T_1 does not conform to 1NF. An alternate representation⁸ was explored where the rows of T_1 were “expanded” to map to single characters from direction arrays (which can occur multiple times in the table) and T_2 is modified accordingly by chopping off the array column. This, however, complicates CTE character and complicates the ability to discern some pairs of characters: Deciding whether a stroke resembles 7 or 1 requires different features than the choice between 7 and 3, say. This difficulty could be patched over on the query level, however.

4 EVALUATION

Basing his analysis on the original IBM/360 assembly program running on a Model 40, Groner writes that the preprocessing and feature extraction stage “takes up about 40 percent of the available computing time in the limiting case [...]. Smoothing and thinning requires about 20 percent of this analysis time, [drawing each incoming segment to the screen] 30 percent, and corner detection 16 percent. The remainder of the analysis time is for computing quantized directions, updating x and y extremes, and bookkeeping.” He goes on to outline potential improvements that could cut the computing time down to “only about 15 percent” [2] but does not mention the cost of walking the decision tree – supposedly it’s minimal. Groner also presents the results of user testing.

The SQL implementation takes 10 – 15 ms to process a typical pen stroke generated by the web app. Taking a look at the EXPLAIN ANALYZE output of the query shown in figure 3, none of the CTEs

⁸Its implementation can be found in the `more relational` branch of the Git repository.

take an unexpectedly high amount of actual time – the computing effort seems to be relatively evenly spread, without any major departures from Groner’s observations.

5 RELATED & FUTURE WORK

Related work. GRAIL is a successor to Sketchpad, one of the first systems that “[make] it possible for a man and a computer to converse rapidly through the medium of line drawings” but relied on a keyboard for text input. [7]

Because Chinese and Korean characters are drawn based on a rigidly defined stroke order, algorithms like the one presented here are well-suited for these writing systems. Indeed, Groner and Heafner worked on “classification of handprinted Chinese characters as a translation aid” [4] in the same timeframe.

Palm later developed Graffiti, a text input system based on single-stroke characters and commands. Among other devices, it was deployed on Apple’s Newton and Windows Mobile.⁹ The SQL implementation could conceivably be modified to recognize Graffiti’s gestures by rewriting the lookup tables.

Dynamic Time Warping may be used for character recognition, given an effective way of converting stroke information to time series.¹⁰

Future work. Recognizing characters composed of multiple strokes would bring the SQL implementation to basic feature parity with the original system.

Groner’s decision tree could be represented more effectively – building the character recognition scheme on top of a graph database might lend itself to an implementation that’s more true to the original.

Because modern tablets feature resolutions significantly higher than the RAND tablet, the smoothing step is less crucial now than it was when the RAND tablet was used. Similarly, the time savings brought about by thinning are barely worth the computing effort on modern, faster systems.

REFERENCES

- [1] Alan Kay. Doing with Images Makes Symbols: Communicating with Computers. <http://archive.org/details/AlanKay01987?start=1439.5>, 1987 (uploaded to archive.org in 2002, accessed August 10, 2018).
- [2] Gabriel F. Groner. *Real-Time Recognition of Handprinted Text*. RAND Corporation, Santa Monica, CA, 1966.
- [3] Gabriel F. Groner. *Real-Time Recognition of Handprinted Text: Program Documentation*. RAND Corporation, Santa Monica, CA, 1968.
- [4] Gabriel F. Groner, J. F. Heafner, and Thomas W. Robinson. *On-Line Computer Classification of Handprinted Chinese Characters as a Translation Aid*. RAND Corporation, Santa Monica, CA, 1967.
- [5] Jack Schaedler. Back to the Future of Handwriting Recognition. <https://jackschaedler.github.io/handwriting-recognition/>, 2016 (accessed August 10, 2018).
- [6] Malcolm Davis and T. O. Ellis. *The RAND Tablet: A Man-Machine Graphical Communications Device*. RAND Corporation, Santa Monica, CA, 1964.
- [7] I. E. Sutherland. Sketchpad: A man-machine graphical communication system. Technical Report UCAM-CL-TR-574, University of Cambridge, Computer Laboratory, Sept. 2003.
- [8] T. O. Ellis, J. F. Heafner, and W. L. Sibley. *The GRAIL Language and Operations*. RAND Corporation, Santa Monica, CA, 1969.
- [9] T. O. Ellis, J. F. Heafner, and W. L. Sibley. *The Grail Project: An Experiment in Man-Machine Communications*. RAND Corporation, Santa Monica, CA, 1969.
- [10] T. O. Ellis, J. F. Heafner, and W. L. Sibley. *The GRAIL System Implementation*. RAND Corporation, Santa Monica, CA, 1969.

⁹See [https://en.wikipedia.org/wiki/Graffiti_\(Palm_OS\)](https://en.wikipedia.org/wiki/Graffiti_(Palm_OS)).

¹⁰Ask the guy in B312 about this.