Supplemental File 1: Graphomotor Feature Extraction

This document describes how we extracted features from graphomotor data collected during the Trail Making Test (TMT) parts A and B.

TMT-A and TMT-B graphomotor data were stored in individual .csv files for each subject. Each file included a time variable (measures were taken every 5 milliseconds), along with recorded x and y coordinates of the pen on the tablet, as well as pressure, azimuth, and elevation.

In Step 1, we pre-processed the data and calculated derivative variables. Step 2 involved detecting probable target (i.e., when participants likely reached one of the encircled numbers or letters) and movement segments. In Step 3, we assigned target identifiers to the target segments, before determining correct and incorrect sequences of targets. In Step 4, we marked zones of correct and incorrect targets and movements, as well as lifts and pauses. In Step 5, we analysed velocity profiles of drawing movements. Finally, in Step 6, we extracted features from the processed data.

All processing steps were performed using Python with a set of dedicated scripts. In the following segments, we explain the processing steps in detail. Most data manipulation was handled by the Pandas package whereas numerical operations were handled by NumPy.

Step 1: Data Preprocessing

Coregraph_data_preprocess.py implements time alignment, data interpolation and smoothing, computation of first-, second- and third-order temporal features (i.e., distance, velocity, acceleration, jerk, direction), and spatial resampling based on distance.

Time Alignment

The sampling rate was 5 milliseconds. Time alignment was performed by taking the time at which the pen first contacted the tablet (pen pressure > 0) as the reference point and extracting the value of this time point from all the time points. Consequently, the time when the pen first contacts the tablet was time 0, indicating the start of the task period, whereas all recorded timepoints prior to this moment obtained a negative sign and can be discarded from analysis.

Data Interpolation and Smoothing

Within the task period, instances where the pen was lifted off the tablet were identified through periods where the pen pressure was zero. Positional data during these non-contact periods were replaced by interpolated data, based on immediately adjacent time points, using the *pandas.Series.interpolate* function with method='linear' and *limit_direction='both'*. This ensured that gaps were filled based on the nearest surrounding data points. To reduce noise in the positional data, the *scipy.signal.savgol_filter* function was applied to the interpolated x and y coordinates, using a window length of 7 and a polynomial order of 3.

Spatial Resampling

We created a more uniform spatial distribution of points by sampling points separated by a distance of at least 50 pixels. This uniform spacing improved the performance of downstream clustering algorithms in the spatial domain.

First-, second-, and third-order temporal features computation

Distance. We calculated the Euclidean distance between consecutive points using smoothed x and y coordinates.

Velocity, Acceleration, Jerk. Velocity was computed as the numerical gradient of the smoothed position data with respect to time, using *numpy.gradient*. Final velocity units were pixels per second. Acceleration was calculated as the gradient of the velocity with respect to time. Jerk was determined as the gradient of the acceleration with respect to time.

Direction was composed of horizontal (x) and vertical (y) movement direction. We calculated the x and y coordinate differences between consecutive spatially resampled points and divided these differences by the Euclidean norm (when non-zero).

Step 2: Detection of Target and Movement Segments

To isolate cognitive task control and movement control components, we needed to separate the data into segments, implemented in *coregraph_clustering.py*. We first identified the movement segments, corresponding to the drawing movement to link two targets, and second, we identified potential targets as the midpoint between movements. Identification of these segments can rely on position, direction, and velocity data. Our algorithm used only position and direction data, and adequately detected movement segments.

Detecting Probable Movement Segments

The challenge was to identify datapoints that belong to the same drawing stroke, reflected by those points having similar albeit noisy direction of movement. To achieve this, we used a clustering algorithm that allowed us to identify these groups of points that belong together while accounting for noise. This algorithm detected the different clusters that corresponded to the different drawing strokes between pairs of targets (see Figure S1).

Clustering. We identified movement segments by clustering according to horizontal and vertical direction and position, using spatially resampled points. These variables were first scaled using the *StandardScaler* from the *sklearn.preprocessing* package. For clustering, we used the *Hierarchical Density-Based Spatial Clustering of Applications with Noise* algorithm (McInnes et al., 2017) from the *hdbscan* package, which requires setting only one parameter: minimum cluster size, which determines how many points need to be within the threshold distance of a given point (minimum cluster size is set at 20 points in our case), for that point to be considered part of the cluster. The maximum distance between two points for them to be considered members of the same cluster is computed by the *hdbscan* algorithm but can be set manually when using the *dbscan* algorithm instead. These clustering algorithms allowed to group spatially adjacent points with similar directions (e.g., those forming a movement), while simultaneously identifying spatially adjacent points with high directional variability as outliers or noise (e.g., often found in target segments). The algorithm performed well in detecting points that belong to the same movement (see Figure S1).

Cleaning. We resolved minor temporal overlaps between clusters by scanning the data in sliding windows of ten consecutive points. Within each window, we identified the most frequent cluster label and marked any point with a different label as NaN. This ensured that the clusters, which each represent a distinct drawing movement, were clearly and fully separated in time.

Probable Target Segments

Probable target locations were inferred at the starting point of the first cluster in chronological order, at the ending point of the last cluster, and at the midpoints of the trajectories between each pair of successive clusters (i.e., when one drawing stroke ends and another begins).

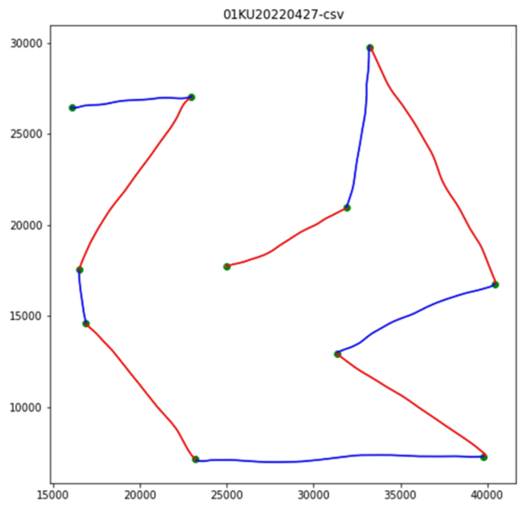


Figure S1: Trace of the pen for one participant on a TMT task. The alternation of blue and red colours indicates that the algorithm detects different clusters (i.e., points identified as belonging together). The green points are the midpoints between clusters.

We also implemented an alternative method for identifying probable targets by detecting dips in velocity (see Figure S2), available in coregraph_targets_velocity.py. This approach can be used in combination with the primary method when the latter does not yield satisfactory results. However, in our case, the primary method alone was sufficient. Local minima in velocity can be identified using the *find_peaks_cwt* function from the *scipy.signal* package, using the inverted velocity data.

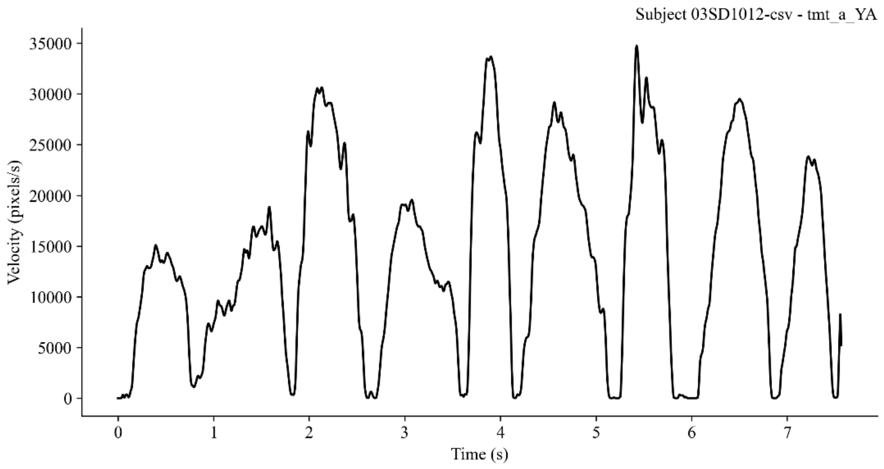


Figure S2: Velocity profile for a participant on a TMT task. Successive peaks represent consecutive movements between two targets, each involving phases of acceleration and deceleration. Dips in velocity indicated moments where targets are probably reached.

Step 3: Assigning Target Identifiers

Complexity arises from variations in paper placement on graphical tablets, which may result in positional inaccuracies and orientation variations. We implemented several steps that take care of such issues with *coregraph_identify_targets.py*.

Scaling Data. Participant and theoretical target position data were scaled using *MinMaxScaler* from *sklearn.preprocessing*. This standardization converted all measurements to a 0 to 1 scale, aligning participant data with the theoretical configuration of targets.

Fixing Reflection Issues. Misplaced sheets sometimes caused the recorded data to be flipped in orientation. To correct this, we tested all combinations of flipping the points along the x and y axes and identified the best match by comparing each configuration to the expected target locations. For each flipped version, we calculated the total squared distance between each participant target and the nearest theoretical target point (using *cdist* from *scipy.spatial.distance*). The configuration with the smallest total distance was selected as the correct orientation.

Identifying Target Candidates

One challenge in matching participant trajectories to the intended target sequence was that movement paths often pass near multiple targets. For example, a participant may move toward target 4 but pass close to target 8 on the way, potentially triggering an incorrect match. To address this, we allowed for multiple candidate matches per actual target (Figure 3 shows an example with two candidate matches for actual target 8), and later selected between them based on spatial and temporal context.

To generate these candidate matches, we first aligned the configuration of the actual targets on the task sheet with the participant's trajectory data. For each actual target, we searched for all probable target points in the participant's data within progressively larger elliptical thresholds. These elliptical zones account for different resolutions in the x and y axes after rescaling (using MinMaxScaler), and were defined in normalized coordinate space (range [0, 1] per axis). The threshold radii increased linearly using numpy.linspace, from 0 up to 0.10

(horizontal) and 0.15 (vertical) for TMT-A/B, and up to 0.06 and 0.09 for TMT-A-long/B-long, respectively. If no probable target was found within the maximum threshold (e.g., when a target was visited only indirectly during a straight movement), we also considered the closest point from all available data, as long as it remained within the maximum threshold.

This process yielded a data frame in which each actual target is linked to a list of candidate points in the participant's trajectory. These candidates were sorted by distance (from closest to furthest), allowing us to later select the most plausible match. This approach allowed us to prioritize matches that are spatially closest while preserving the information needed to resolve incorrect initial matches in a subsequent correction step.

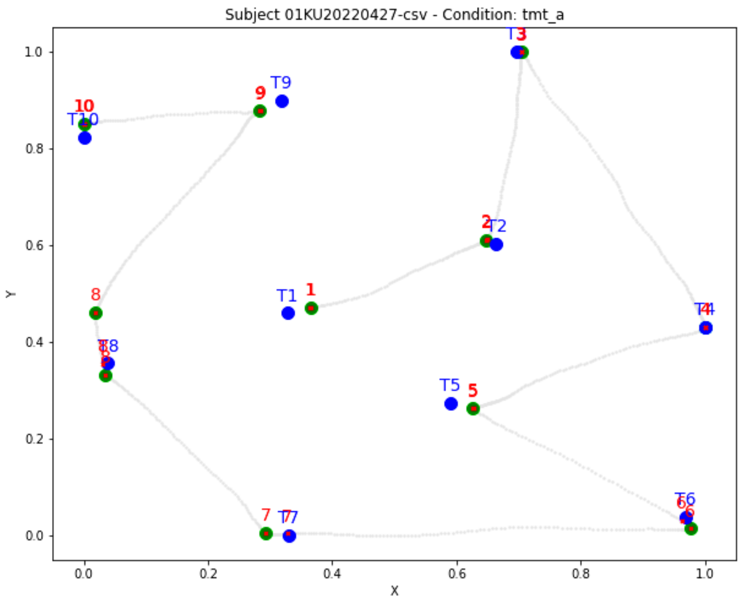


Figure S3: Blue dots indicate actual targets, while green dots indicate candidate matching targets in the participant data.

Assign and Correct Targets

The process of assigning and correcting target identifiers in participant data was implemented in *coregraph_targets_assign_correct.py* and involved two nested and iterative steps, as follows:

Assign Target Identifiers. The assignment step attributed all not-yet assigned target identifiers to the single most suitable target from the dataframe of potential targets. After each iteration of target identifier assignment, the sequence of assignments was verified in the *Correct Target Identifiers* step (see below). It was possible for this step to lead to some target identifiers being unassigned. The assign target identifiers process continued to iterate until all target identifiers were definitively assigned or until all potential targets were exhausted.

Correct Target Identifiers. This process examined each assigned target identifier and checked if it followed the expected correct sequence. That is, if the target identifier followed numerically on the previous target identifier (e.g., target 2 follows target 1), the target was marked as correct. If an assigned target identifier did not follow the expected sequence (e.g., target 10 does not follow target 8), we assumed that the participant made an error, and the target was marked as incorrect. However, it was first determined whether the incorrect target met any exceptions (i.e., in the longer versions of the TMT, exceptions could happen when a target was almost on the path between two targets, e.g., target 1- target 7- target 2, triggering a premature match). If an exception was met, the target identifier was unassigned rather than marked as incorrect, because in this case no participant error was made. Unassigned targets were to be assigned again in the Assign Targets step. See Figure S4 for an example of a correction of one correct and one incorrect sequence of target identifiers.

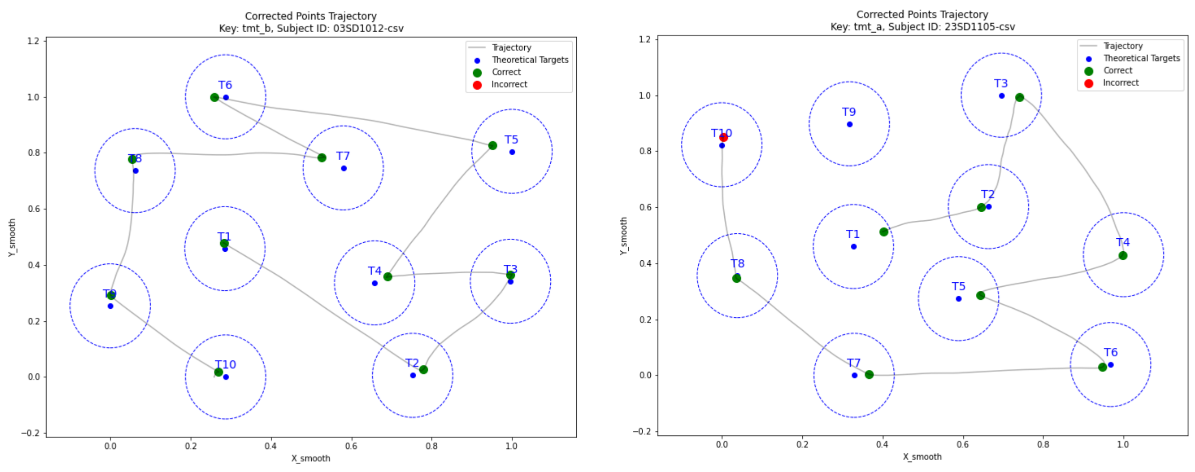


Figure S4: Correction of target identifiers with display of theoretical targets (blue), targets correctly visited by the participant (green), and targets incorrectly visited (red). On the right hand side, we see that the participant incorrectly visited target 10 after visiting target 8, thereby skipping target 9.

Step 4: Marking Zones

With the script *coregraph_zone_identification.py*, the data were divided in correct and incorrect target zones and movement zones, and lifts and pauses were marked.

Target zones. The target zone refered to the area surrounding datapoints that were assigned a target identifier. To define its boundaries, we looked at the datapoints before and after each target until a movement cluster was encountered in either direction. All points between these movement segments were included in the target zone. Each target zone was then labelled as correct or incorrect based on whether the target was correctly visited.

Movement zones. These zones were defined by first locating the datapoints labelled as part of a movement cluster. From these datapoints, the movement zone extended forward until the next target zone was reached. Each movement zone was then assigned the target identifier of the following target zone, along with a label indicating whether the movement was to a correct or an incorrect target.

Lifts. Rows where a lift was enacted were marked, identified by pressure changing from > 0 to 0.

Pauses. A pause was defined as a row with a velocity < 100 pixels per second. Pauses were marked separately for target and movement zones.

Step 5: Velocity Analysis

Velocity analysis was done with *coregraph_extract_features.py.*

We analysed the velocity profile for each drawing movement linking two targets. Figure S5 shows several examples of velocity profiles for a single drawing movement.

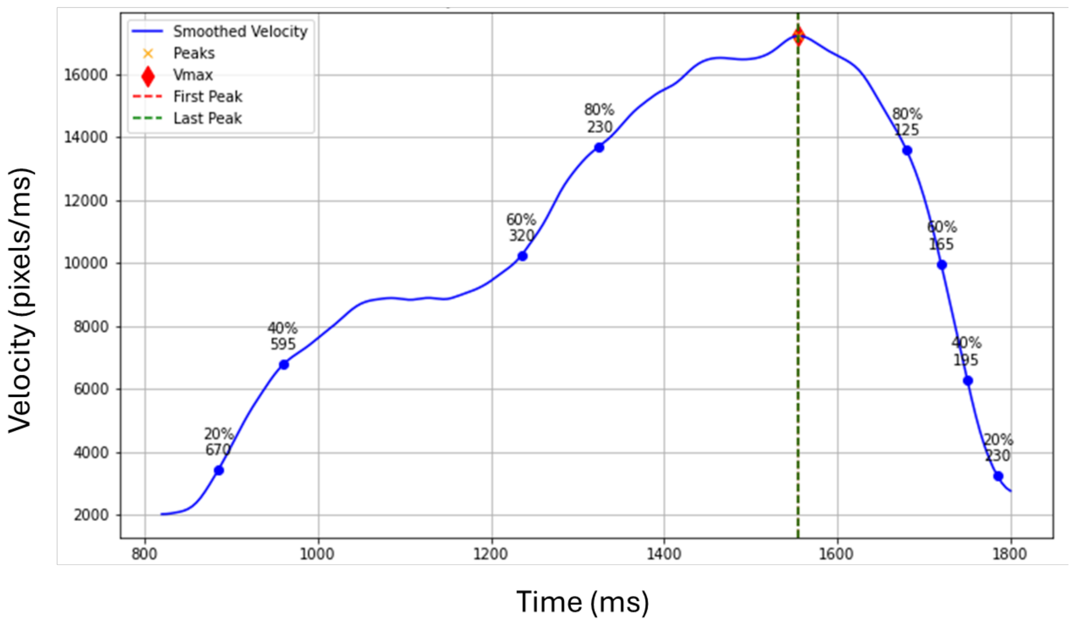
Smoothing Velocity Data. Velocity data were smoothed using a uniform filter with a window size of 20 to reduce noise and make peak detection more conservative. Larger window sizes would lead to more smoothing and remove more peaks.

Peak Detection. We identified peaks in the smoothed velocity data using the *find_peaks* function from the *scipy* library. Peaks were detected based on a specified minimum speed height (i.e., difference from 0) and prominence (i.e., difference from surrounding values) thresholds. The location of the peaks in the data was returned for each movement.

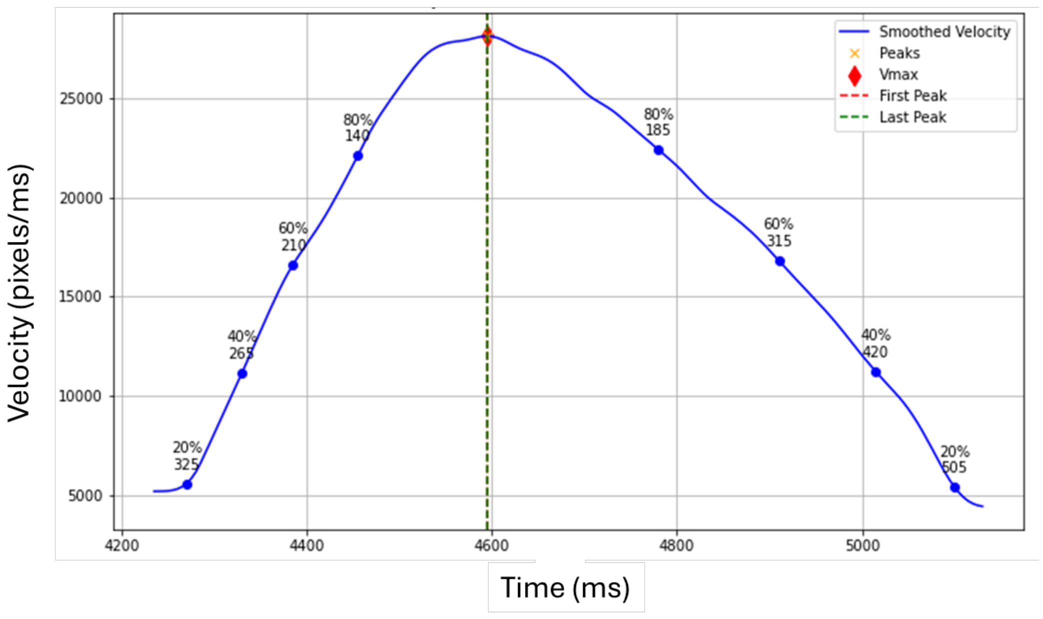
Maximum Velocity. We located the point (V$_{\_max}$) with maximum velocity together with the corresponding velocity value.

Peak Characteristics. We calculated several characteristics of the peak that contains VMAX. For this, we marked several velocity thresholds in the data, when the velocity was at 80%, 60%, 40% or 20% of VMAX, both in the acceleration phase leading up to VMAX and the deceleration phase following VMAX. We calculated the time delta between each of these threshold points and VMAX, as well as between the successive thresholds on each side. Figure 5 shows these thresholds marked on the velocity profile, with the time delta to VMAX.

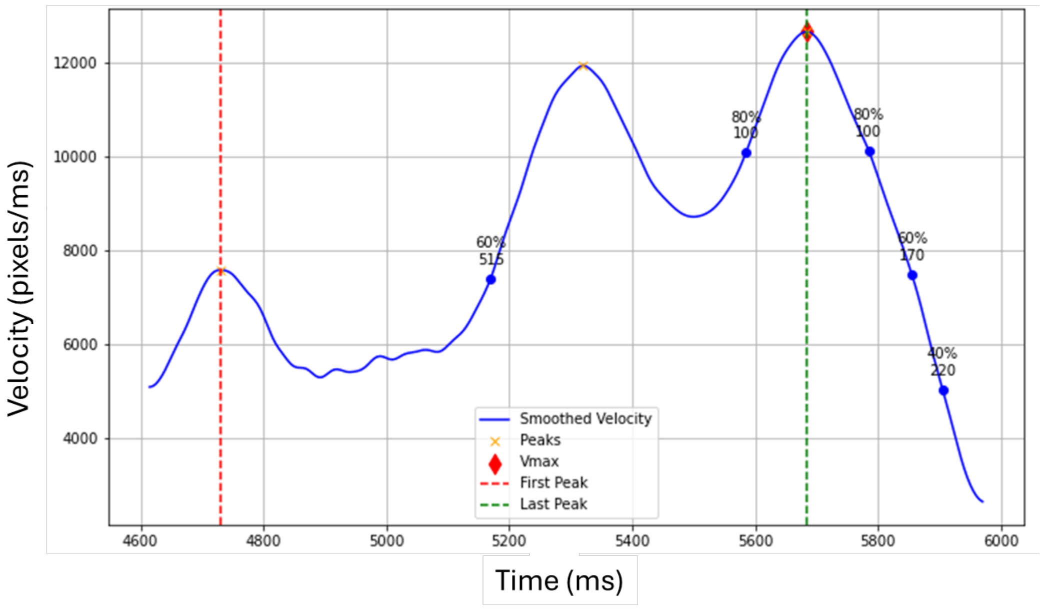
A. Acceleration Time > Deceleration Time



B. Deceleration Time > Acceleration Time



C. Multiple Peaks



D. Acceleration Time == Deceleration Time (Symmetry)

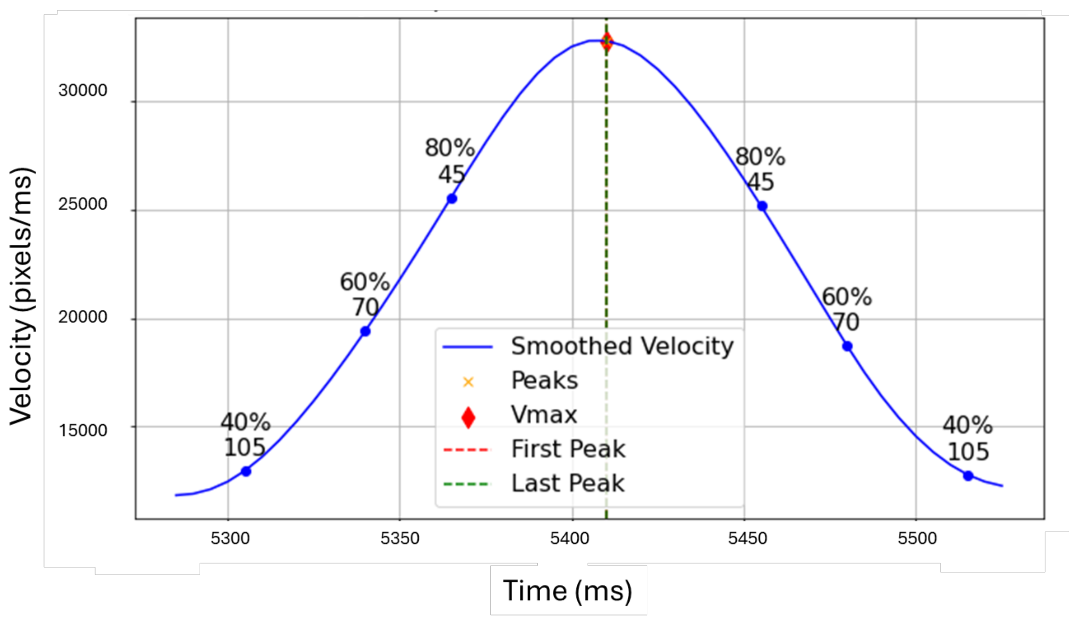


Figure S5: Examples of velocity profiles for one movement linking two targets, with indication of velocity peaks, and the point of maximum velocity.

Step 6: Feature Extraction

Feature extraction was performed using *coregraph_extract_features.py*.

Definitions

- **pause**: A datapoint with a velocity less than 100 pixels per second.
- **movement**: A drawing segment connecting two targets.
- **correct_target**: A visited target that correctly follows the previously visited target.
- **correct_movement**: A movement that connects two correct targets.
- **one_peak_movement**: A movement with a single velocity peak, indicating a ballistic movement.
- **path_efficiency**: The total amplitude of a movement (sum of distances between successive points) divided by the Euclidean distance between the movement's start and end points.

Extracted Features

** Indicates the features that were included in the analysis for the paper*

- **task_time**: The total time from when the pen touches the tablet at the beginning of the task until the pen leaves the tablet at the end of the task.
- **pretask_time**: Negative time before the first pen-down event. This may reflect preparation time or a delay between task launch and participant response, limiting its interpretability.
- **count_correct_targets**: Total number of correctly visited targets.
- ***error_count**: Number of errors, calculated as the total number of targets minus the number of correct targets.
- **lifts_count**: Number of pen lifts during the task (i.e., between the first and last contact with the tablet).
- ***avg_pause_between_targets**: Average pause duration during correct movements between targets.
- ***avg_pause_on_targets**: Average pause duration while on correct targets.
- ***avg_peaks_/movement**: Average number of velocity peaks per correct movement.
- **count_one_peak_movements**: Total number of correct movements with a single velocity peak.
- ***proportion_one_peak_movements**: Proportion of one-peak movements among all correct movements.
- ***maximum_velocity**: Average maximum velocity (VMAX) reached during correct one-peak movements.

- **avg_threshold_to_peak_ratio**. A measure of symmetry for correct one-peak movements, calculated from the peak characteristics. Specifically, for each velocity threshold, we divide the time delta before the peak by the time delta after the peak. The average of these ratios is taken as a global measure of symmetry across different thresholds. This global measure of symmetry is averaged across all correct one-peak movements.
  - Values > 1: longer acceleration phase
  - Values < 1: longer deceleration phase
  - Value = 1: symmetrical profile
- ***long_acceleration**: Average of avg_threshold_to_peak_ratio values > 1, indicating longer acceleration phases.
- ***long_deceleration**: Average of avg_threshold_to_peak_ratio values < 1, indicating longer deceleration phases.
- **avg_threshold_slopes_ratio**: A slightly different measure of symmetry for correct one-peak movements. It involves calculating the time delta between successive thresholds before and after the peak, giving a measure of slope on each side. The time deltas (slopes) before the peak are divided by the time deltas after the peak. The average of these ratios is taken as a global measure of symmetry.
  - Values < 1: steeper acceleration
  - Values > 1: steeper deceleration
  - Value = 1: symmetry
- **avg_path_efficiency**: Average path efficiency across all correct movements.
- **avg_pressure_during_line**: Average pen pressure during correct movements.
- **avg_sd_pressure_during_line**: Average standard deviation of pen pressure during correct movements.

References

McInnes, L., Healy, J., & Astels, S. (2017). hdbscan: Hierarchical density based clustering. *The Journal of Open Source Software*, 2(11), 205. https://doi.org/10.21105/joss.00205