Touchscreen Gesture Recognition Using Cosine Similarity

Buege Mahara Putra - 13523037[1]
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
[1]buege.putra@gmail.com, 13523037@std.stei.itb.ac.id

***Abstract*—Touchscreen gesture recognition is an integral component of modern human-computer interaction, providing an intuitive mechanism for executing commands on devices. This study evaluates the feasibility of using a lightweight gesture recognition system based on cosine similarity, offering an alternative to computationally intensive machine learning models. The proposed system preprocesses gesture images into binary grids, extracts feature vectors by analyzing grid densities, and classifies gestures by comparing feature vectors with template gestures using cosine similarity. Extensive experimentation on varying grid sizes reveals that mid-range configurations, such as 30 × 20, achieve optimal accuracy of 78.57%. Despite the simplicity and computational efficiency of the approach, the recognition accuracy remains suboptimal for practical deployment.**

***Keywords*—Cosine similarity, feature vectors, gesture recognition, grid-based preprocessing**

I. Introduction

The rapid development of touchscreen technology has transformed how users interact with devices. Early touchscreen systems, first seen in the 1960s and 1970s, were limited in precision and responsiveness, primarily relying on resistive touch technology. E. A. Johnson invented the first finger-driven touchscreen in 1965 at the Royal Radar Establishment in Malvern, United Kingdom. Johnson's work laid the foundation for capacitive touchscreens, commonly used today in high-end smartphones and tablets [1].

With advancements in capacitive touchscreens during the late 2000s, popularized by devices like the Apple iPhone, touch interfaces became faster, more accurate, and capable of supporting multi-touch gestures. This evolution paved the way for the widespread adoption of touchscreen devices, making gestures an indispensable tool for interaction across consumer, industrial, and medical applications. As the technology matured, the demand for efficient gesture recognition systems grew, especially in areas requiring real-time performance and adaptability to diverse user inputs.

Touchscreen gestures have become a fundamental component of human-computer interaction, providing users with an intuitive and efficient way to navigate systems and execute commands. From swipes and taps to intricate gesture patterns, this modality has become essential in devices such as smartphones, tablets, and embedded systems. Despite the widespread use of gesture-based interfaces, creating an effective and reliable gesture recognition system remains a challenging task. Variability in user input, environmental noise, and differences in device sensitivity all contribute to the complexity of the problem.

Many modern gesture recognition systems leverage machine learning, particularly deep learning models, to achieve high accuracy. While these methods excel in handling complex and diverse input, they often come with significant computational requirements and the need for extensive training datasets. This makes them impractical for resource-constrained environments, such as low-power embedded systems or real-time applications with limited computational budgets. Consequently, there is a need to explore simpler, more efficient methods that can balance accuracy and computational cost.

In this paper, we explore the possibility of using a lightweight gesture recognition system based on cosine similarity, a straightforward measure of similarity between two vectors. Our focus is not on introducing a novel algorithm but rather on evaluating whether a simple, feature-based approach can serve as an effective alternative to more complex machine learning models. The goal is to understand how well such methods perform under various conditions and to assess their potential for real-world applications.

By focusing on simplicity, interpretability, and efficiency, this study aims to provide insights into the trade-offs between computational complexity and recognition accuracy. It seeks to highlight the potential of classical methods for solving practical problems, especially in scenarios where high computational resources are unavailable or unnecessary.

II. Theoretical Basis

A. Gestures

Gesture refers to a physical movement or position that conveys meaning or intent. In the context of touchscreen devices, gestures are intentional finger movements, such as swipes, taps, pinches, and complex patterns, used to perform specific commands or actions. These gestures act as a bridge between human intent and machine interpretation, enabling seamless interaction with digital interfaces [2].

Gesture recognition is the process of identifying and interpreting gestures from input data. In touchscreen systems, this typically involves analyzing the trajectory of finger movements or shapes drawn on the screen. The process often

consists of two stages:

1. *Feature Extraction*
   Converting raw input (e.g., images or coordinates) into a mathematical representation.
2. *Classification*
   Comparing the extracted features against predefined gesture templates to determine the best match.

B. Matrix

A matrix is a two-dimensional array of numbers arranged in rows and columns. Formally, a matrix M with m rows and n columns is represented as:

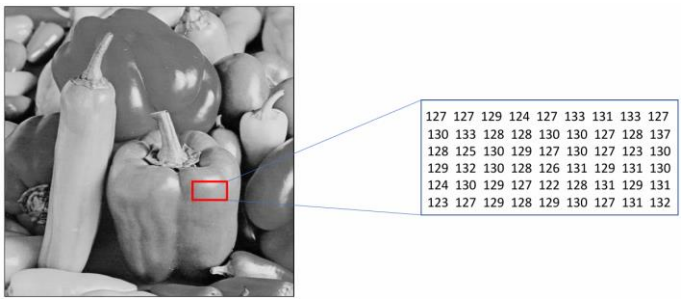
$$M = \begin{bmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m,1} & a_{m,2} & \cdots & a_{m,n} \end{bmatrix} \quad (1)$$

Matrix can be manipulated by operations similar to algebra. One of the operation relevant to this paper is scalar multiplication. Matrix scalar multiplication involves multiplying each element of the matrix by a scalar value. Given a matrix M and a scalar k, the resulting matrix is:

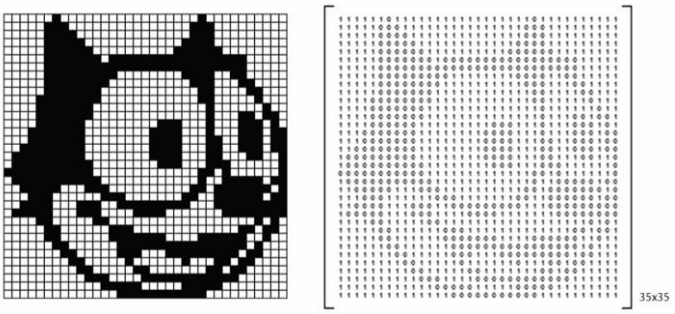
$$k \cdot M = \begin{bmatrix} k \cdot a_{1,1} & k \cdot a_{1,2} & \cdots & k \cdot a_{1,n} \\ k \cdot a_{2,1} & k \cdot a_{2,2} & \cdots & k \cdot a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ k \cdot a_{m,1} & k \cdot a_{m,2} & \cdots & k \cdot a_{m,n} \end{bmatrix} \quad (2)$$

Matrices are widely used to represent various types of data, including images and transformations, in both mathematical and computational contexts.

In the case of images, each element of a matrix corresponds to a pixel, with its value representing the intensity of the pixel. In grayscale images, this intensity value typically ranges from 0 (black) to 255 (white), with intermediate values representing varying shades of gray. For binary images, where the only colors present are black and white, pixel values are limited to 0 (black) and 1 (white), simplifying the representation.



(a) Grayscale image



(b) Binary image

Fig. 1. Pixel values in images represented as matrix
Source: [3]

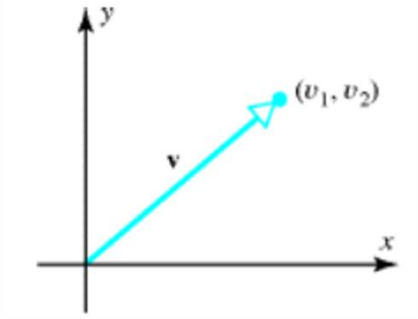
When resizing images, techniques like bilinear interpolation are commonly used to preserve image quality. Bilinear interpolation resamples the image by estimating the value of a pixel in the resized image through a weighted average of the four nearest pixels in the original image. This approach helps to ensure smooth transitions and maintain the shape and structure of the gesture, which is crucial during preprocessing for gesture recognition.

C. Vector

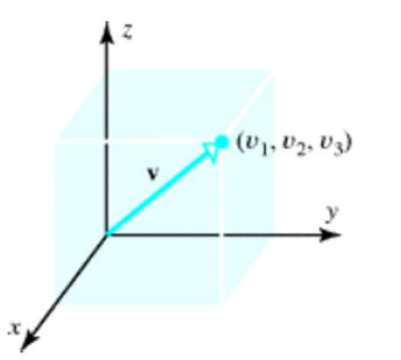
A vector is a mathematical object that represents both magnitude and direction. It is expressed as an ordered list of numbers, called components, each of which corresponds to a specific direction in a given space. In n-dimensional space (R^n), a vector $\mathbf{v}$ is written as:

$$\mathbf{v} = [v_1, v_2, \ldots, v_n] \quad (3)$$

Each component v_i is a scalar that indicates the magnitude of the vector in the direction of the corresponding axis in the space. For example, in a 2D space, a vector $\mathbf{v} = [v_1, v_2]$ represent a direction in the x-axis and y-axis, respectively. Similarly, in a 3D space, a vector $\mathbf{v} = [v_1, v_2, v_3]$ represent a direction in the x-axis, y-axis, and z-axis, respectively.



(a) Vector in R^2 *space*



(b) Vector in R^3 *space*
Fig. 2. Vector in Euclidean spaces
Source: [4]

Vectors exist within a vector space, which is a set of vectors that can be added together and multiplied by scalars while still remaining within the same space. A vector space V must satisfy certain properties [5]:

1. *Closure*
   The result of addition and scalar multiplication of vectors must be within the same space. If $\mathbf{u}, \mathbf{v} \in V$ and k is a scalar, then:
   $$\mathbf{u} + \mathbf{v} \in V \quad (4)$$
   $$k\mathbf{u} \in V \quad (5)$$

2. *Commutativity*
   Vector addition must be commutative, meaning for all $\mathbf{u}, \mathbf{v} \in V$:
   $$\mathbf{u} + \mathbf{v} = \mathbf{v} + \mathbf{u} \quad (6)$$

3. *Associativity*
   Vector addition must be associative, meaning for all $\mathbf{u}, \mathbf{v}, w \in V$:
   $$\mathbf{u} + (\mathbf{v} + \mathbf{w}) = (\mathbf{u} + \mathbf{v}) + \mathbf{w} \quad (7)$$

4. *Identity*
   For all $\mathbf{u} \in V$, there exist $\mathbf{0}$ identity vector and scalar 1 such that:
   $$\mathbf{u} + \mathbf{0} = \mathbf{0} + \mathbf{u} = \mathbf{u} \quad (8)$$
   $$1\mathbf{u} = \mathbf{u} \quad (9)$$

5. *Inverse*
   For each $u \in V$, there exists $-u \in V$, such that:
   $$\mathbf{u} + (-\mathbf{u}) = (-\mathbf{u}) + \mathbf{u} = \mathbf{0} \quad (10)$$

6. *Distributivity*
   Vector addition and scalar multiplication must hold distributive properties. For all $\mathbf{u}, \mathbf{v}, \mathbf{w} \in V$ and k, m is scalars, then:
   $$k(\mathbf{u} + \mathbf{v}) = k\mathbf{u} + k\mathbf{v} \quad (11)$$
   $$(k + m)\mathbf{w} = k\mathbf{w} + m\mathbf{w} \quad (12)$$
   $$k(m\mathbf{u}) = (km)\mathbf{u} \quad (13)$$

One of the fundamental operations in vector is scalar multiplication, which involves multiplying each component of a vector by a scalar. Given a vector $\mathbf{v} = [v_1, v_2, \dots, v_n]$ and a scalar k, the operation results in a new vector:

$$k \cdot \mathbf{v} = [k \cdot v_1, k \cdot v_2, \dots, k \cdot v_n] \quad (14)$$

This operation scales the vector, changing its magnitude without affecting its direction. Scalar multiplication is crucial for transforming vectors or normalizing them to unit length in various applications.

Another fundamental vector operation is the dot product (or scalar product) of two vectors, which provides a measure of their similarity. It is defined as the sum of the products of their corresponding components. For two vectors $\mathbf{A} = [a_1, a_2, \dots, a_n]$ and $\mathbf{B} = [b_1, b_2, \dots, b_n]$ the dot product is given by:

$$\mathbf{A} \cdot \mathbf{B} = \sum_{i=1}^{n} a_i b_i \quad (15)$$

The dot product has several important properties. It is commutative, meaning $\mathbf{A} \cdot \mathbf{B} = \mathbf{B} \cdot \mathbf{A}$, and distributive over vector addition. It is also bilinear, meaning it is linear in each of its arguments.

Geometrically, the dot product of two vectors can also be expressed as:

$$\mathbf{A} \cdot \mathbf{B} = \|\mathbf{A}\|\|\mathbf{B}\| \cos\theta \quad (16)$$

Where $\|\mathbf{A}\|$ and $\|\mathbf{B}\|$ are the magnitudes (or norms) of the vectors, and θ is the angle between them. If the vectors are parallel, the dot product is maximized. If they are orthogonal, the dot product is zero, indicating no similarity.

The norm of a vector, also known as the magnitude or length, is a measure of its size. The most commonly used type of norm is the Euclidean norm, which is defined as:

$$\|\mathbf{v}\| = \sqrt{\sum_{i=1}^{n} v_i^2} \quad (17)$$

This norm calculates the straight-line distance from the origin to the point represented by the vector in the space. The norm of two vectors follow triangle inequality, meaning $\|\mathbf{v} + \mathbf{w}\| \leq \|\mathbf{v}\| + \|\mathbf{w}\|$ for any vectors v and w.

D. Information Retrieval System

Information retrieval (IR) system is a framework designed to retrieve relevant information from a dataset in response to a query. Examples include search engines and document matching systems. In the context of gesture recognition, the task of classifying gestures is analogous to information retrieval: the input gesture serves as the "query," and the system matches it to the most relevant template from a database.

A key concept in IR systems is the measurement of similarity between the query and the available documents. One common

metric used to compare the similarity of two vectors is cosine similarity. For query vector **Q** and document vector **D**, it is defined as [6]:

$$\text{sim}(\mathbf{Q}, \mathbf{D}) = cos\,\theta = \frac{\mathbf{Q} \cdot \mathbf{D}}{\|\mathbf{Q}\|\|\mathbf{D}\|} \quad (18)$$

This metric quantifies how closely the two vectors align in terms of their direction. The value ranges from -1 to 1, where values closer to 1 indicates greater similarity, and values closer to -1 indicates greater dissimilarity.

A value of 1 indicates that the vectors are identical in direction, meaning they have maximum similarity. A value of 0 suggests that the vectors are orthogonal, meaning there is no similarity between them. Conversely, a value of -1 indicates that the vectors point in opposite directions, signifying maximum dissimilarity.

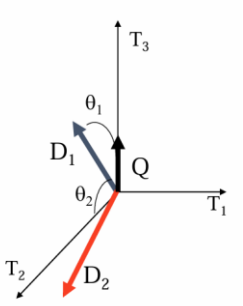


Fig. 3. A query vector **Q** *compared to two document vectors* $\mathbf{D_1}$ *and* $\mathbf{D_2}$
Source: [6]

Thus, cosine similarity provides a useful measure for comparing vectors in a variety of applications, including gesture recognition, where it helps assess how similar a query gesture is to predefined gesture templates.

III. METHODOLOGY

The proposed gesture recognition system presented in this paper aims to classify touchscreen gestures using cosine similarity as the core metric for matching. The methodology involves a series of systematic stages, including dataset preparation, preprocessing, feature extraction, and classification, each contributing to the overall recognition process. These stages are carefully designed to ensure the system accurately identifies input gestures by matching them with the most relevant templates in the database.

A. Dataset Preparation

The foundation of any recognition system lies in its dataset. For this study, the dataset are comprised of black-and-white images, where the white pixels trace the path of the gesture drawn on a touchscreen. Each gesture is recorded as a PNG image with a resolution of 240 × 150 pixels and an 8-pixel-wide gesture path. This resolution and path size were chosen to strike a balance between accurately representing the gesture drawn and minimizing storage requirements. Standardizing the image size eliminates inconsistencies that could arise from varying dimensions, ensuring uniformity across the dataset. For simplicity and proof of concept, all gesture images in this study were created using Microsoft Paint, providing a controlled environment for generating the dataset.

B. Preprocessing

Once the dataset is prepared, the next stage is preprocessing, which transforms raw gesture images into a structured format suitable for subsequent analysis. In this stage, each gesture image in the dataset is converted into a binary image matrix, where the white pixels are represented with the value of 1 and the black pixels are represented with the value of 0. This binary representation ensures simplicity and efficiency in processing. To further structure the image, the matrix is divided into equal-sized grid cells. This grid-based segmentation is essential for the next stage, as it facilitates the extraction of spatial information by dividing the image into smaller, manageable regions.

C. Feature Extraction

Feature extraction is the stage where the meaningful characteristics of the gestures are distilled into a numerical representation. Using the grid-based representation from preprocessing, the system calculates the proportion of white pixels within each grid cell. This process transforms the binary image matrix into a grid of density values that capture the spatial distribution of the gesture. The grid is then flattened into a feature vector, where each component maps to a specific region of the gesture image. This high-dimensional vector preserves the general spatial structure of the gesture, allowing the system to compare gestures effectively. By condensing the gesture's details into a density format, the system reduces the complexity of raw images while retaining the information necessary for accurate classification.

D. Classification

With the feature vectors obtained from the feature extraction stage, the system proceeds to classify input gestures by comparing their feature vectors to those of the template gestures in the database. The core of this classification process lies in cosine similarity, a metric that measures the angular similarity between two vectors. For each input gesture, cosine similarity evaluates the alignment between its feature vector and the feature vectors of all template gestures. A similarity score is produced for each comparison, ranging from -1 to 1, with score closer to 1 indicates a high degree of similarity, while a score closer to -1 indicates significant dissimilarity. The template gesture with the highest similarity score is selected as the predicted class. To account for poorly matched gestures, a threshold is applied. If the highest similarity score falls below this threshold, the input gesture is classified as "unrecognized." This mechanism helps to prevent erroneous classifications in cases where the input gesture does not closely resemble any of the stored templates.

E. Evaluation

The final stage of the methodology involves evaluating the performance of the system. Accuracy is assessed by comparing

the predicted gesture labels against the ground truth labels for a diverse set of test gestures. The evaluation dataset includes variations in gesture styles and drawing consistency to simulate real-world scenarios and test the system's ability to generalize. This evaluation shows the system's effectiveness and its potential for practical applications.

IV. Implementation

The implementation of the gesture recognition system is developed using Python, using libraries such as OpenCV for image processing and NumPy for numerical computations. The system is modular, comprising several key functions that perform preprocessing, feature extraction, classification, and evaluation.

A. Preprocessing Gesture Images into Grids

Preprocessing begins by converting gesture images into a binary format and dividing them into grids. The function **`preprocess_image_into_grids`** takes an image filepath and a specified grid size as inputs. It loads the image in grayscale, binarizes it (assigning pixel values of 0 or 1), and divides it into a 4D array of grid cells. This process ensures a consistent representation of gesture images for subsequent feature extraction. This function ensures uniformity in image dimensions and prepares the data for extracting spatial features.

```
def preprocess_image_into_grids(filepath,
                                grid_size):
  image = cv2.imread(filepath,
                     cv2.IMREAD_GRAYSCALE)
  if image is None:
    raise FileNotFoundError(f"Image not found
                            at {filepath}")
  _, binary_image = cv2.threshold(image, 127, 1,
                                  cv2.THRESH_BINARY)
  height, width = binary_image.shape
  if (height % grid_size[0] != 0 or width %
      grid_size[1] != 0):
    raise ValueError(f"Image dimensions({height}
                     , {width}) must be divisible
                     by grid size {grid_size}")
  cell_height = height // grid_size[0]
  cell_width = width // grid_size[1]
  grid = binary_image.reshape(cell_height,
                              grid_size[0],
                              cell_width,
                              grid_size[1])
  return grid
```

Fig. 4. Code snippet for the preprocessing function

B. Extracting Features from Gesture Grids

The function **`extract_features`** converts the preprocessed grid into a feature vector. It computes the sum of white pixels in each grid cell and flattens the resulting 2D array into a 1D feature vector. This vector represents the spatial structure of the gesture and is crucial for classification.

```
def extract_features(grid):
  density_grid = grid.sum(axis=(1, 3))
  vector = density_grid.flatten()
  return vector
```

Fig. 5. Code snippet for feature extraction function

By summarizing pixel densities within grid cells, this method reduces the complexity of raw image data while retaining essential spatial information.

C. Gesture Recognition Using Cosine Similarity

To classify gestures, the function **`cosine_similarity`** computes the similarity between two feature vectors. It calculates the cosine of the angle between the vectors, with values closer to 1 indicating higher similarity.

```
def cosine_similarity(vector1, vector2):
  dot_product = np.dot(vector1, vector2)
  magnitude = np.linalg.norm(vector1) *
              np.linalg.norm(vector2)
  return dot_product / magnitude
```

Fig. 6. Code snippet for cosine similarity function

The **`recognize_gesture`** function uses this metric to compare the feature vector of an input gesture with template vectors stored in a dictionary. The gesture with the highest similarity score is identified as the recognized gesture.

```
def recognize_gesture(input_filepath,
                      gesture_vectors,
                      grid_size):
  grid = preprocess_image_into_grids(
         input_filepath, grid_size)
  input_vector = extract_features(grid)
  best_match = None
  best_score = -1
  scores = []
  for gesture_id, template_vector in
      gesture_vectors.items():
    score = cosine_similarity(input_vector,
                              template_vector)
    scores.append(float(score.round(2)))
    if score > best_score:
      best_match = gesture_id
      best_score = score
  return best_match, best_score, scores
```

Fig. 7. Code snippet for input gesture recognition

D. Loading Gesture Templates

The **load_gesture_vectors** function preprocesses template gestures stored in a directory. It converts each gesture image into a feature vector and maps it to a unique gesture ID. This function creates a database of gesture templates, enabling efficient comparison during recognition.

```
def load_gesture_vectors(data_dir, grid_size):
  gesture_vectors = {}
  for filename in os.listdir(data_dir):
    if filename.endswith(".png"):
      gesture_id = os.path.splitext(filename)[0]
      filepath = os.path.join(data_dir, filename)
      grid=preprocess_image_into_grids(filepath,
                                       grid_size)
      feature_vector = extract_features(grid)
      gesture_vectors[gesture_id]=feature_vector
  return gesture_vectors
```

Fig. 8. Code snippet for loading gesture templates function

E. Evaluating System Performance

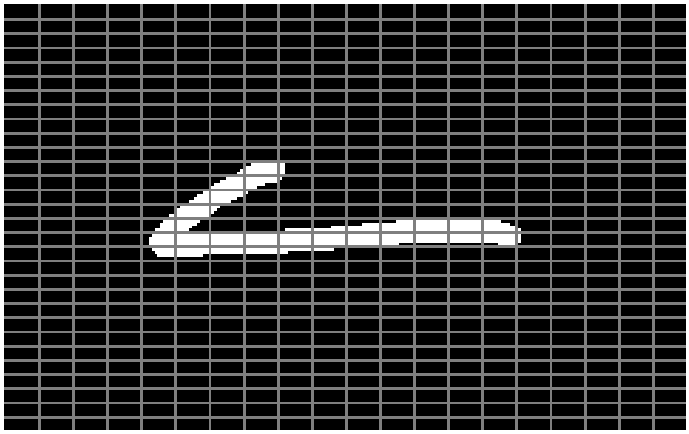
To evaluate the system, a loop iterates through grid sizes, preprocesses gestures from input and template directories, and calculates accuracy for each grid size. Recognition accuracy is determined by comparing the predicted gesture IDs with the ground truth.

```
accuracies = []
for i in range(1, 151):
  if 150 % i != 0:
    continue
  for j in range(1, 241):
    if 240 % j != 0:
      continue
    grid_size = (i, j)
    data_dir = "./template"
    input_dir = "./input"
    gesture_vectors = load_gesture_vectors(
                        data_dir, grid_size)
    total, correct = 0, 0
      for filename in os.listdir(input_dir):
      input_filepath = os.path.join(input_dir,
                                    filename)
      recognized_gesture, similarity, scores =
        recognize_gesture(input_filepath,
                          gesture_vectors,
                          grid_size)
      print(f"Input {filename} recognized as
            gesture {recognized_gesture}.
            (Similarity score:
            {similarity:.2%})", end="\t")
      for score in scores:
        print(score, end="\t")
      print()
      total += 1
      if filename[0] == recognized_gesture:
        correct += 1
    accuracy = correct / total
    print(f"Accuracy: {accuracy:.2%}
          ({correct}/{total}) for grid size
          {grid_size[0]}x{grid_size[1]}")
    accuracies.append((grid_size, accuracy))
```

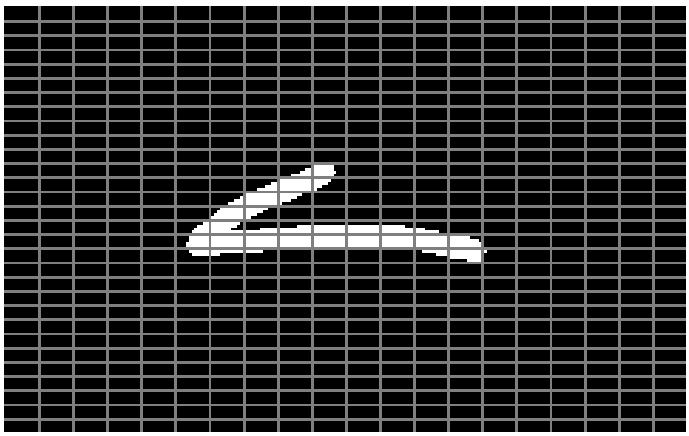
Fig. 9. Code snippet for the main driver

V. Analysis

The system rely heavily on the preprocessing stage to transform raw input into a format that allows for accurate feature extraction and classification. To achieve this, the input gesture images are first binarized and divided into grids of specified size. Fig. 10. is an excerpt from the preprocessing pipeline, showcasing a sample gesture template and its corresponding input gesture used for evaluation.



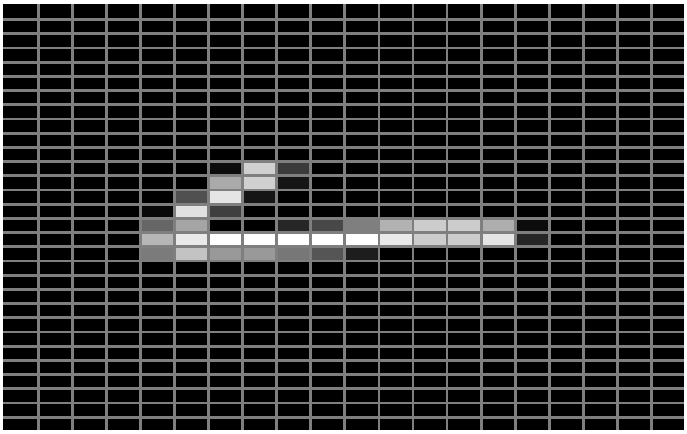
(a) Template path



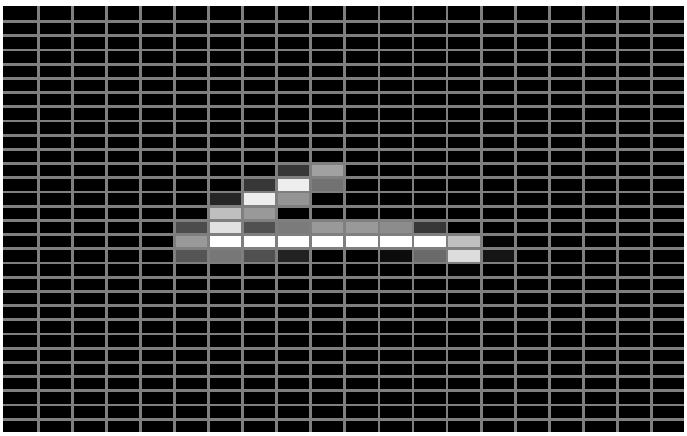
(b) Input path

Fig. 10. A pair of template and input gesture image, subdivided by 30 × 20 grid after preprocessing pipeline

After preprocessing, the gesture image undergoes feature extraction, where each grid cell is summed to compute its pixel density. This step outputs a compact feature vector representing the gesture, capturing both the structural and spatial characteristics necessary. The resulting feature vector is then compared to a database of precomputed gesture templates using cosine similarity as the matching metric. Fig. 11. demonstrates the image post-feature extraction using grid size of 30 × 20, where the grid cell intensities reflect the distribution of the path pixels across the image.



(a) Template path



(b) Input path

Fig. 11. A pair of template and input gesture image converted into density matrix after feature extraction pipeline

To determine the optimal grid size for maximizing recognition accuracy, every combination of grid sizes is tested extensively. Grid configurations range from highly granular and small sizes, such as 1 × 1 and 2 × 2, to coarse and large sizes like 150 × 240. Each configuration generates a unique feature vector representation, impacting the system's ability to distinguish between gestures. Below is the table of 5 top-performing and worst-performing grid sizes, along with graph displaying every combination performance, which displays the balance between granularity and accuracy.

Table 1. 5 top-performing and worst-performing grid sizes

Grid Size	**Correct (out of 56 inputs)**	**Accuracy (%)**
Top-performing		
30 x 20	44	78.57
30 x 24	43	76.79
30 x 48	43	76.79
15 x 20	40	71.43
25 x 20	40	71.43
Worst-performing		
75 x 120	19	33.93
150 x 120	17	30.36
150 x 60	15	26.79
75 x 240	11	19.64
150 x 240	8	14.29

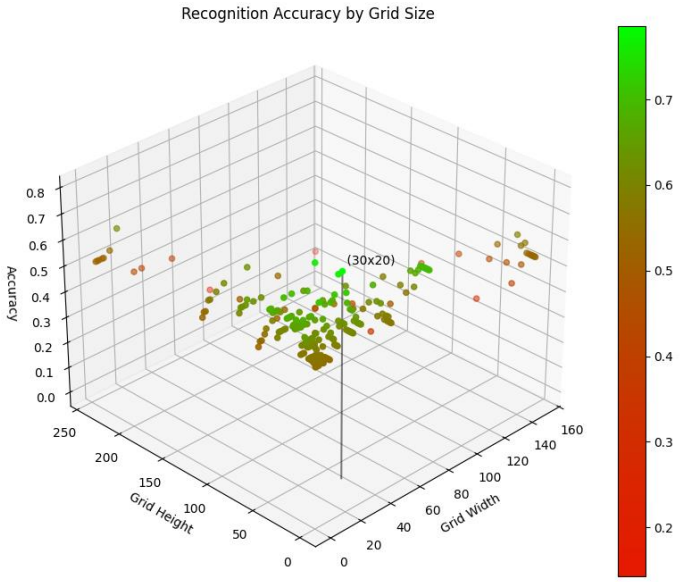


Fig. 12. Scatter plot of various grid sizes and its resulting accuracies

From Table 1 and Fig. 12, it is evident that grid sizes in the small-to-mid range such as 30 × 20, achieving an accuracy of 78.57%, outperform both overly coarse and excessively granular configurations. This success can be attributed to their ability to preserve crucial structural details without introducing segmentation noise. In comparison, overly coarse grids, such as 150 × 240, fail to capture sufficient detail, leading to worse accuracy. Small and granular grid sizes provide decent accuracy, despite resulting in over-segmentation and amplifying noise.

VI. Conclusion

Despite the promising results observed with certain grid sizes, the overall accuracy levels highlight that the system, in its current state, is not reliable enough for consistent, day-to-day usage. Even the best-performing grid size, 30 × 20, achieves an accuracy of only 78.57%, which falls short for practical

applications requirement where high reliability is essential. Without further enhancements to both preprocessing and classification methodologies, the system risks frequent misclassifications, making it unsuitable for environments that demand precision and consistency.

VII. Acknowledgment

The author sincerely expresses gratitude towards the lecturers of IF2123 Geometric and Linear Algebra, particularly Ir. Rila Mandala, M.Eng., Ph.D., the lecturer for class K-01, for his continuous guidance and expertise throughout the semester. The author also expresses gratitude to their friends and family for their endless support during the period of writing this paper.

References

[1] Q. Dang, "Touchscreen: An engineered harmony between humans and machines - USC viterbi school of engineering," USC Viterbi School of Engineering - USC Viterbi School of Engineering, https://illumin.usc.edu/touchscreen-an-engineered-harmony-between-humans-and-machines/ (accessed Jan. 1, 2025 at 17.11 WIB).

[2] "Gestures," Apple Developer Documentation, https://developer.apple.com/design/human-interface-guidelines/gestures (accessed Jan. 1, 2025 at 18.42 WIB).

[3] R. Munir, "Review Matriks", https://informatika.stei.itb.ac.id/~rinaldi.munir/AljabarGeometri/2023-2024/Algeo-01-Review-Matriks-2023.pdf (accessed Dec. 30, 2024 at 12.44 WIB).

[4] R. Munir, "Vektor di Ruang Euclidean (bagian 1)", https://informatika.stei.itb.ac.id/~rinaldi.munir/AljabarGeometri/2024-2025/Algeo-11-Vektor-di-Ruang-Euclidean-Bag1-2024.pdf (accessed Dec. 30, 2024 at 13.30 WIB).

[5] R. Munir, "Ruang Vektor Umum (bagian 1)", https://informatika.stei.itb.ac.id/~rinaldi.munir/AljabarGeometri/2023-2024/Algeo-15-Ruang-vektor-umum-Bagian1-2023.pdf (accessed Dec. 30, 2024 at 14.10 WIB).

[6] R. Munir, "Aplikasi Dot Product pada Sistem Temu-balik Informasi (Information Retrieval System)", https://informatika.stei.itb.ac.id/~rinaldi.munir/AljabarGeometri/2023-2024/Algeo-14-Aplikasi-dot-product-pada-IR-2023.pdf (accessed Dec. 30, 2024 at 17.58 WIB).

Statement

I hereby declare that the paper I have written is my own work, not an excerpt or translation of someone else's paper, and is not plagiarized.

Bandung, 2 January 2025

Buege Mahara Putra
13523037