Application of Singular Value Decomposition (SVD) for Fingerprint Matching in Biometric Systems

Aliya Husna Fayyaza - 13523062
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
[1]13523062@std.stei.itb.ac.id, [2]aliyahfayyaza@gmail.com

***Abstract*— Fingerprint recognition is a widely used biometric technology that relies on extracting and analyzing unique features called minutiae. This paper explores an approach to fingerprint recognition, starting with preprocessing steps such as gray scaling and skeletonizing the image then extractint the minutiae points using a crossing number algorithm. The minutiae points are then standardized and turned into a covariance matrix. Singular Value Decomposition (SVD), using the QR algorithm, is applied to derive eigenvectors and project the data into a reduced-dimensional feature space. To evaluate the similarity between a query image and the database, the query is processed through the same pipeline and compared using cosine similarity to measure the angular distance between the query and the images from the dataset. This method ensures an efficient fingerprint recognition by retaining essential features while reducing computational complexity.**

***Keywords*— fingerprint recognition, minutiae extraction, singular value decomposition, QR algorithm**

I. Introduction

Fingerprint recognition has become one of the most widely used biometric techniques for identification and authentication, from daily corporate staff presence to forensic analytics. Each person has a unique fingerprint pattern, making it a reliable tool to identify someone accurately. However, the fingerprint recognition process requires a highly accurate algorithms to extract and compare the small features differences of the fingerprint patterns.

One of the popular approaches for fingerprint feature extraction is minutiae-based algorithms, which has the ability to extract little details from fingerprint patterns. This project use Crossing Number algorithm minutiae extraction method and Singular Value Decomposition method to compress data while retaining important information. The program use SVD to construct a fingerprint library where each fingerprint is represented as a projection vector. To evaluate similarity, the program use Cosine Similarity, which compares query vectors against vectors in the library.

II. Basic Theory

A. Eigenvalue and Eigenvectors

Eigenvectors show how a matrix transforms certain directions, and eigenvalues indicate how much the vector is stretched or shrunk. The word "eigen" itself originates from German, meaning "characteristic" or "inherent". For an n×n matrix A, a vector x (where x is not 0) in space Rn is called the eigenvector of A if the result of multiplying A by x is equivalent to multiplying x by a scalar λ. The scalar λ is the eigenvalue of A and x is the corresponding trivial solution. The equation above could be manipulated using the identity matrix I to get the characteristic equation.

$$A\mathbf{x} = \lambda\mathbf{x}$$

$$IA\mathbf{x} = \lambda I\mathbf{x}$$

$$A\mathbf{x} = \lambda I\mathbf{x}$$

$$(\lambda I - A)\mathbf{x} = 0 \quad (1)$$

To ensure a non-zero solution, the matrix should be singular or has the determinant of zero. and the solutions to this equation are the eigenvalues of matrix A. As shown in this figure below,

$$\det(\mathrm{A} - \lambda \mathrm{I}) = 0 \quad (2)$$

The corresponding eigenvectors can be determined by substituting these eigenvalues. If λ is positive, the vector keeps its direction; if λ is negative, it flips direction.

Eigenvectors and eigenvalues have many practical uses in different sectors because they help simplify complex problems and show important properties of transformations. In data science, they are used in Principal Component Analysis (PCA) to reduce the size of large datasets. Here, eigenvectors show the main directions of data, while eigenvalues tell how much information each direction contains. In engineering, they are used to study vibrations and stability of structures in physics, especially quantum mechanics. These examples show how eigenvectors and eigenvalues help solve real-world problems more easily.

B. QR Algorithm

Increasing computational efficiency and ensuring numerical stability are some of the key reasons the QR Algorithm is chosen as the method for finding eigenvectors of a matrix. The QR Algorithm is an iterative procedure specifically designed to approximate the eigenvalues and eigenvectors of an n×n matrix A. This method utilizes the QR decomposition, where the matrix A is factored into Q and R, with Q being an orthogonal matrix (its columns are orthonormal vectors) and R being an upper triangular matrix. The

decomposition process can be achieved through methods such as the Gram-Schmidt orthogonalization, which is illustrated in the figure below.

$$\begin{aligned}
\mathbf{u}_1 &= \mathbf{v}_1, & \mathbf{e}_1 &= \frac{\mathbf{u}_1}{\|\mathbf{u}_1\|} \\
\mathbf{u}_2 &= \mathbf{v}_2 - \operatorname{proj}_{\mathbf{u}_1}(\mathbf{v}_2), & \mathbf{e}_2 &= \frac{\mathbf{u}_2}{\|\mathbf{u}_2\|} \\
\mathbf{u}_3 &= \mathbf{v}_3 - \operatorname{proj}_{\mathbf{u}_1}(\mathbf{v}_3) - \operatorname{proj}_{\mathbf{u}_2}(\mathbf{v}_3), & \mathbf{e}_3 &= \frac{\mathbf{u}_3}{\|\mathbf{u}_3\|} \\
\mathbf{u}_4 &= \mathbf{v}_4 - \operatorname{proj}_{\mathbf{u}_1}(\mathbf{v}_4) - \operatorname{proj}_{\mathbf{u}_2}(\mathbf{v}_4) - \operatorname{proj}_{\mathbf{u}_3}(\mathbf{v}_4), & \mathbf{e}_4 &= \frac{\mathbf{u}_4}{\|\mathbf{u}_4\|} \\
&\vdots & &\vdots \\
\mathbf{u}_k &= \mathbf{v}_k - \sum_{j=1}^{k-1} \operatorname{proj}_{\mathbf{u}_j}(\mathbf{v}_k), & \mathbf{e}_k &= \frac{\mathbf{u}_k}{\|\mathbf{u}_k\|}.
\end{aligned}$$

Fig 1. Gram-Schmidt orthogonalization (source: https://informatika.stei.itb.ac.id/~rinaldi.munir/AljabarGeometri/2024-2025/Algeo-23b-Dekomposisi-QR-2024.pdf)

In this decomposition, v_k represents the column vector of A, e_k is the column vector of Q and $e_k \times v_k$ forms the column vector of R. Following the QR decomposition, the matrix A is updated using the relation A' = RQ. This updated matrix A' remains similar to the original matrix A, meaning it still have same eigenvalues. By repeating this step iteratively, the matrix $A^{(k+1)} = R^{(k)}Q^{(k)}$ is constructed, where $A^{(k)}$ progressively converges towards a quasi-diagonal form. In this quasi-diagonal (a matrix that is similar to diagonalization, but it accommodates cases where the matrix is not fully diagonalizable) form, the eigenvalues of the original matrix appear on the diagonal.

The process of finding eigenvectors is carried out simultaneously by accumulating the successive Q matrices obtained during each iteration. The product of these accumulated matrices, expressed as $V=Q^{(1)}Q^{(2)}\ldots Q^{(k)}$, yields the matrix V, where its columns correspond to the eigenvectors of the original matrix A. However, it is important to note that this result provides an approximation of the eigenvectors rather than their exact values. The accuracy of this approximation depends on the number of iterations k, which is typically limited and can be adjusted based on computational constraints. Despite these limitations, the QR Algorithm remains a powerful and widely used tool for efficiently finding eigenvalues and eigenvectors in various applications.

C. Singular Value Decomposition

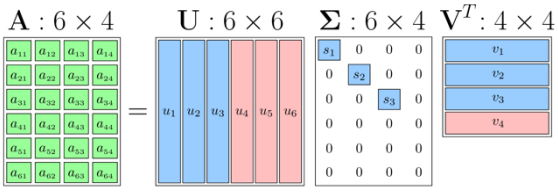

Fig 2. Example of Singular Value Decomposition (source: https://truetheta.io/concepts/linear-algebra/svd-and-the-fundamental-theorem-of-linear-algebra/)

This method is used to decomposed a matrix A with the dimension m × n into U, E and V^t. Matrix U is the left singular vectors with the dimension m × m , E is the singular values of A as its main diagonal with the dimension m × n (note that E is not a square matrix, so the main diagonal is defined as the line starting from the top-left corner and extending downward as far as possible within the matrix) and V^t is the right singular vectors with the dimension n × n.

Firstly, matrix A × A^t is counted, then QR algorithm is used to find the eigenvalues and eigenvectors of matrix A × A^t, and the matrix U is found by normalizing the sorted eigenvectors (based on the corresponding eigenvalues). Secondly, the V^t is found by the same steps but it is A^t × A instead of A × A^t, and lastly, U is found by rooting the eigenvalues of A × A^t. In this project, the focus will only be on matrix U that U provides the principal directions in the input space of A, making it essential for understanding the structure of the data and performing tasks like dimensionality reduction, compression, and feature extraction.

There is also another method to find the singular variance decomposition of a matrix. In this method, instead of determining the left and right singular vectors, this method only count for the right singular vectors that are corresponding to the eigenvalues of the matrix $A^t \times A$. These vectors are normalized by dividing each component by the vector's magnitude, forming the matrix V. Then, transpose V to obtain V^t. The rank of A, denoted as k is equal to the number of non-zero eigenvalues of matrix $A^t \times A$. SVD is used in various way like image and video compression, image processing, machine learning, computer vision, and digital watermarking.

D. Covariance Matrix

The covariance matrix is a mathematical tool that captures the variance and relationships between different variables in a dataset. The diagonal elements of the covariance matrix represent the variance of individual variables, while the non-diagonal elements capture the covariance between pairs of variables, indicating how strongly they are related. To calculate the covariance matrix, the first step is to center the data by subtracting the mean of each feature, ensuring that the data is unbiased by its scale. Once centered, the covariance matrix is computed using the formula

$$\text{Cov}(X) = \frac{1}{n} X^T X \tag{3}$$

where n is the number of samples, X is the data matrix, and X^t is its transpose. This results in a symmetric matrix that summarizes the variance and covariances of the features in the dataset.

The covariance matrix is often used in Singular Value Decomposition (SVD) because it encapsulates the variance structure of the data, making it ideal for dimensionality reduction and feature extraction. SVD applied to the covariance matrix allows for dimensionality reduction by focusing on the largest singular values and their associated vectors, which represent the principal components of the data. This process not only reduces the complexity of the data but also provides numerical stability.

D. Cosine Similarity

Cosine similarity is a mathematical measure used to evaluate the similarity between two non-zero vectors in a multidimensional space. It determines how aligned two vectors are by measuring the cosine of the angle between them. The cosine similarity between two vectors A and B is calculated as:

$$\cos(\theta) = \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\|\|\mathbf{B}\|} = \frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2} \sqrt{\sum_{i=1}^{n} B_i^2}} \quad (4)$$

where A·B is the dot product of A and B, ‖A‖ is the magnitude (or norm) of A, ‖B‖ is the magnitude (or norm), and θ is the angle between the two vectors.

E. Fingerprints Minutiae

There are two common features of fingerprints that are used for identification and authentication. The first one is called ridge ending, a point where a ridge abruptly stops. The second one is called bifurcation, a point where a ridge forks into two separate ridges. These features are collectively called minutiae, and are the most common minutiae used in fingerprint recognition.

Fingerprint minutiae are unique because they represent the distinct patterns and specific points of a fingerprint that vary from one individual to another. The arrangement of minutiae, such as ridge endings, bifurcations, enclosures, and dots, is determined by genetic and environmental factors during fetal development. Even identical twins, who share the same DNA, have unique fingerprint minutiae due to the randomness in how the ridges form.

The uniqueness of minutiae lies not only in their types but also in their relative positions, orientations, and distances within the fingerprint. This combination of spatial and structural uniqueness makes minutiae a reliable feature for distinguishing fingerprints. Fingerprint recognition programs analyze these minutiae points, creating a "map" of their positions, which is then used to match identities accurately. This inherent uniqueness ensures that no two fingerprints are identical, providing a secure basis for biometric authentication systems.

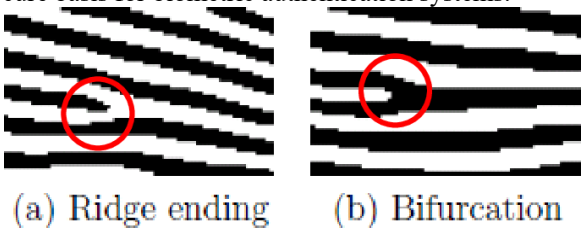

Fig 3. Example of ridge endings and ridge bifurcations (source: https://www.researchgate.net/figure/Example-of-ridge-endings-and-ridge-bifurcations_fig2_268041882)

F. Fingerprint Minutiae Extraction Algorithm Using Crossing Number

Before entering the main algorithm, the images of fingerprints have to be gray scaled and skeletonized. Gray scaling is done by using the Luminance method, showed in the picture below.

$$Y = 0.299R + 0.587G + 0.114B \quad (5)$$

The coefficients represent the contributions of each color channel to perceived brightness.

Then, skeletonizing is done by converting the gray scaled images into binary image where ridges becomes 1 and the background becomes 0, this is separated based on the number of pixel as the threshold. Then, the preprocessed images enter the Crossing Number algorithm that is used to detect minutiae from the skeletonized images, the ridge ending and the bifurcation. The iteration is applied on every pixel of the skeletonized images except of the edges, then the program will check only the ridge pixel (valued at 1) and ignore the backgrounds (valued at 0). Then, neighbors array formed by 8 pixels surrounding the processed pixel (3x3 kernel).

After that, the crossing number is counted by iteratively counts the transition of 0 to 1 across the neighbors then adding the transition of the last neighbors to the first one (to create a loop). If the crossing number is 1, it will be identified as ridge ending, if the crossing number is 3, it will be identified as bifurcation, and others with crossing number other than 1 and 3 does not identified as minutiae.

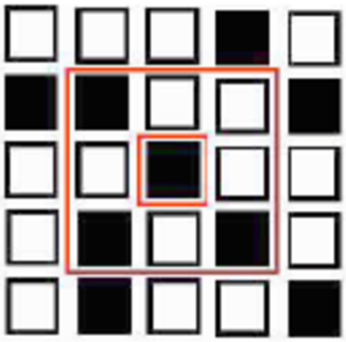

Fig 4. Crossing number visualization (source: https://www.researchgate.net/figure/Crossing-Number-and-Type-of-Minutiae_fig2_220485747)

Then, the minutiae points will be converted into fixed-size binary vector. This uniform representation enables the application of mathematical techniques, such as Singular Value Decomposition (SVD), to identify principal components and reduce the dimensionality of the data while retaining essential features by extracting the mean, eigenvectors, and projections to the SVD space of the covariance of the standardized data.

III. Implementation

A. Data Processing

The process starts by loading the dataset then standardizing the size and format of images, involving preprocessing steps like gray scaling and resizing, which are handled in the *load_dataset* function. The target size of the resized image can be set here.

```python
def load_dataset(folder_path, target_size=TARGET_SIZE):
    data = []
    image_names = []
    for filename in os.listdir(folder_path):
        if filename.endswith(('.jpg', '.jpeg', '.png', '.bmp')):
            image_path = os.path.join(folder_path, filename)
            image = Image.open(image_path).convert("L")
            image = image.resize(target_size)
```

Fig 5. load_dataset function
(source: author's source code)

Then, the minutiae points of the fingerprint image are extracted based on the crossing number of each pixel in the image, identifying key features such as ridge endings and bifurcations. This process is implemented in the *extract_minutiae* function. The image is first skeletonized using the operation *skeleton = (image > 100).astype(np.uint8)*, which applies a threshold of 100 to differentiate the ridges from the background. Pixels with values greater than 100 are set to 1, while others are set to 0, creating a binary image. The *np.uint8* data type is used to ensure that the binary image consists of 8-bit unsigned integers, which are either 0 or 1.

Once the image is skeletonized, the function iterates through each pixel using nested loops over the dimensions of the image. For each pixel with a value of 1 (indicating a ridge), its eight neighboring pixels are analyzed. These neighbors are stored in a list called *neighbors*, representing the surrounding ridge structure.

If a pixel has a crossing number equals to 1 or 3, its coordinates are added to the *minutiae_points* list, capturing the location of the minutiae. This approach ensures that only the most critical ridge features are identified and the list will contains all the detected key points, which are crucial for the matching and recognition stages.

```python
def extract_minutiae(image):
    skeleton = (image > 100).astype(np.uint8)
    minutiae_points = []
    rows, cols = skeleton.shape
    for i in range(1, rows - 1):
        for j in range(1, cols - 1):
            if skeleton[i, j] == 1:
                neighbors = [
                    skeleton[i-1, j], skeleton[i-1, j+1], skeleton[i, j+1], skeleton[i+1, j+1],
                    skeleton[i+1, j], skeleton[i+1, j-1], skeleton[i, j-1], skeleton[i-1, j-1]
                ]
                cn = sum((neighbors[k] - neighbors[k-1]) == 1 for k in range(8))
                + (neighbors[0] - neighbors[-1] == 1)
                if cn == 1 or cn == 3:
                    minutiae_points.append((i, j))
    return minutiae_points
```

Fig 6. extract_minutiae function
(source: author's source code)

Then, the minutiae points will be converted into vectors so they can be processed further, as shown in this *minutiae_points_to_vector* function below. The function works by mapping the coordinates of each minutiae point into a fixed-size vector using the formula index *(x * 16 + y) % vector_size*. The factor 16 in the formula is related to the resolution or grid size of the fingerprint image, where each row corresponds to 16 columns, ensuring that the x coordinate is properly scaled to maintain uniqueness when combined with y. The modulo operation *% vector_size* ensures that the resulting index fits within the bounds of the vector. This step allows the function to create a binary vector where each position set to 1 represents the presence of a minutiae point. This vectorized representation is made so it can be processed further.

```python
def minutiae_points_to_vector(minutiae_points, vector_size=120):
    vector = np.zeros(vector_size)
    for x, y in minutiae_points:
        index = (x * 16 + y) % vector_size
        vector[int(index)] = 1
    return vector
```

Fig 7. minutiae_points_to_vector function
(source: author's source code)

After that, the minutiae vectors will be standardized by subtracting their mean vector, which is calculated using *np.mean(data, axis=0)*. This ensures that the data is centered around zero, which is a necessary step for constructing an accurate covariance matrix. The standardized data is then used to compute the covariance matrix, implemented as *np.dot(standardized_data.T, standardized_data)/ len(data)*. The eigenvectors are found using the Singular Value Decomposition (SVD) method, specifically through the QR algorithm. In the code, this algorithm is implemented by looping through each column of the matrix A, applying the Gram-Schmidt process to orthogonalize it with respect to previously computed vectors.

During this process, a threshold value of 1e-10 is used to ensure numerical stability. If the norm of a column vector is greater than 1e-10, the vector is normalized to have unit length. However, if the norm falls below this threshold, the vector is replaced with a zero vector using *np.zeros_like(col)*. This prevents division by very small numbers, which could lead to instability or computational errors. Then, the matrix A is updated iteratively as RQ, and this process is repeated 10 times until the eigenvalues and eigenvectors converge. The eigenvectors are normalized in the *svd_matching* function using *np.linalg.norm(eigenvectors, axis=0),* ensuring they have unit length.

Finally, the standardized data is projected into the eigenvector space using *np.dot(standardized_data, eigenvectors)*, reducing the dimensionality while preserving the most meaningful features. The function returns the mean vector, eigenvectors, and projections, providing all necessary components for further processing in the fingerprint recognition pipeline.

```python
def svd_matching(data):
    mean_vector = np.mean(data, axis=0)
    standardized_data = data - mean_vector

    covariance = np.dot(standardized_data.T, standardized_data) / len(data)

    eigenvectors = primitifmatriks.svd(covariance)
    eigenvectors /= np.linalg.norm(eigenvectors, axis=0)

    projections = np.dot(standardized_data, eigenvectors)
    return mean_vector, eigenvectors, projections
```

Fig 8. svd_matching function
(source: author's source code)

```
def QR(matrix, iterations=10):
    n = len(matrix)
    A = np.array(matrix, dtype=np.float64)
    eigenvectors = np.eye(n)
    for _ in range(iterations):
        Q = []
        for i in range(n):
            col = A[:, i]
            for prev in Q:
                scale = np.dot(col, prev)
                col -= scale * prev
            norm = np.linalg.norm(col)
            if norm > 1e-10:
                col = col / norm
            else:
                col = np.zeros_like(col)
            Q.append(col)
        Q = np.array(Q).T
        R = np.dot(Q.T, A)
        A = np.dot(R, Q)
        eigenvectors = np.dot(eigenvectors, Q)

    eigenvalues = np.diag(A)
    return eigenvalues, eigenvectors.tolist()
```

Fig 9. QR function
(source: author's source code)

Lastly, the mean vector, eigenvectors, and the projections of data into the eigenvector space are stored in a csv (the directory could be set as the *OUTPUT_FILE*) to later used as the database to find similarity with a query image. This is done in the function *save_to_csv*.

```
def save_to_csv(mean_vector, eigenvectors, projections, image_names,
                output_file=OUTPUT_FILE):
    with open(output_file, mode="w", newline="") as file:
        writer = csv.writer(file)

        writer.writerow(["Mean_Vector"] + mean_vector.tolist())

        writer.writerow(["Eigenvectors"])
        for eigenvector in eigenvectors:
            writer.writerow([""] + eigenvector.tolist())

        writer.writerow(["Projections"])
        for name, projection in zip(image_names, projections):
            writer.writerow([name] + projection.tolist())
```

Fig 10 save_to_csv function
(source: author's source code)

B. Query Processing

The query image will go through the same processing steps as the database images, including standardization until the projection into the eigenvector space derived from the database. This ensures that the query image is represented within the same reduced-dimensional feature space as the database projections, allowing for consistent and accurate similarity comparisons between the query and the stored images. This part will be processed in the function shown below.

```
def query_minutiae(query_vector, mean_vector, eigenvectors,
                   projections, image_names):
    standardized_query = query_vector - mean_vector
    print(f"Standardized query vector: {standardized_query}")

    projected_query = np.dot(standardized_query, eigenvectors)
    print(f"Projected query vector: {projected_query}")

    similarities = []
    for name, projection in zip(image_names, projections):
        similarity = cosine_similarity(projection, projected_query)
        similarities.append((name, similarity))

    similarities.sort(key=lambda x: x[1], reverse=True)
    return similarities
```

Fig 11. query_minutiae function
(source: author's source code)

whereas the *mean_vector*, *eigenvectors*, *projections*, *image_names* data will be collected from the *library.csv* through this function *load library.*

```
def load_library(csv_path):
    with open(csv_path, mode='r') as file:
        reader = csv.reader(file)
        rows = list(reader)

    mean_vector = np.array([float(x) for x in rows[0][1:]])
    eigenvectors = np.array([[float(x) for x in row[1:]] for row in rows[2:122]])
    eigenvectors /= np.linalg.norm(eigenvectors, axis=0)
    projections = np.array([[float(x) for x in row[1:]] for row in rows[133:]])
    image_names = [row[0] for row in rows[133:]]

    return mean_vector, eigenvectors, projections, image_names
```

Fig 12. load_library function
(source: author's source code)

C. Similarity Evaluation

In this evaluation, cosine similarity algorithm is used to calculate the angular distance between the query projection and each projection in the database. This is implemented in the *cosine_similarity* function. The function begins by calculating the dot product of the two input vectors, *vec1* (the query projection) and *vec2* (a database projection), using *np.dot(vec1, vec2)*. The dot product measures the degree to which the vectors point in the same direction.

Next, the function computes the norms of each vector, which calculate the euclidean lengths of the vectors. To prevent division by zero or instability when the vector norms are extremely small, a small value, epsilon=1e-8, is added to each norm. This ensures numerical stability and avoids undefined behaviour. Then, the cosine between the two vectors are calculated by dividing the dot product by the product of the magnitude of both vectors.

```
def cosine_similarity(vec1, vec2, epsilon=1e-8):
    dot_product = np.dot(vec1, vec2)
    norm_vec1 = np.linalg.norm(vec1) + epsilon
    norm_vec2 = np.linalg.norm(vec2) + epsilon
    return dot_product / (norm_vec1 * norm_vec2)
```

Fig 13. cosine_similarity function
(source: author's source code)

IV. Testing

The database used in this testing model consists of 801 fingerprint images, collected from Kaggle. Author tested the model by using one of the fingerprint as the query. The query is shown below with the name of "00008_54.bmp", then the program shows 5 most similar fingerprints along as the similarity score, such as shown below.

```
Top matches:
00008_54.bmp: 1.0000
00008_39.bmp: 0.5677
00008_57.bmp: 0.5462
00009_49.bmp: 0.5317
00000_24.bmp: 0.4992
```

Fig 14. Similarity result of the query 00008_54.bmp
(source: author's archive)

It shows that the cosine similarity between the same fingerprint is 1, meaning that the vectors goes in the exact

same direction, or in another word, the two fingerprints matches. For reference, images displayed below are the images referred in the top matches, ordered from top-left to top-right, then bottom-left to bottom-right.

Fig 15. Corresponding fingerprints of query 00008_54.bmp similarity result (source: author's archive)

As observed, the similarity score drops significantly from the most similar fingerprint (with a similarity score of 1.000) to the second most similar (with a similarity score of 0.5677). This aligns with the theory that every fingerprint is unique, distinguished by the minutiae characteristics of each individual fingerprint. Although the images may appear visually similar, the minutiae characteristics of "00008_39.bmp," "00008_57.bmp," "00009_49.bmp," and "00000_24.bmp" are very different from the query image. However, these fingerprints are equally dissimilar to the query, resulting in similar similarity scores.

This similar result also appeared in a different query that author tested. The query is shown below with the name of "00008_05.bmp", then the program shows 5 most similar fingerprints along as the similarity score, , such as shown below.

```
Top matches:
00008_05.bmp: 1.0000
00008_35.bmp: 0.5411
00009_47.bmp: 0.5147
00000_64.bmp: 0.5104
00008_06.bmp: 0.4829
```

Fig 16. Similarity result of the query 00008_05*.bmp (source: author's archive)*

Fig 17. Corresponding fingerprints of query 00008_05.bmp similarity result (source: author's archive)

V. Conclusion

The application of Singular Value Decomposition (SVD) and minutiae-based feature extraction has proven effective for fingerprint recognition in biometric systems. Using the Crossing Number algorithm for minutiae extraction and SVD for dimensionality reduction enables efficient processing and storage of fingerprint data while preserving essential features. Cosine similarity, as the matching metric, ensures reliable and accurate identification, with clear distinctions in similarity scores between identical and non-identical fingerprints.

The approach reduces computational complexity and demonstrates scalability for large databases, making it suitable for real-world applications. The results confirm that unique fingerprint minutiae can be analyzed and compared effectively, providing accurate identification for purposes such as authentication, security, and forensic analysis.

VI. Appendix

Github:
https://github.com/aliyahusnaf/AlgeoPaper2024.git
Bonus video: https://youtu.be/cNtBS8fsE9A

VII. Acknowledgment

Author expresses gratitude to God Almighty for His blessings and grace, which have provided the strength to write this paper titled "Application of Singular Value Decomposition (SVD) for Fingerprint Matching in Biometric Systems" successfully. The author would also like to extend heartfelt thanks to Dr. Ir. Rinaldi Munir, M.T., the lecturer of the Linear Algebra and Geometry course for the first semester of 2023/2024, Class 02, for imparting valuable knowledge that served as a foundation for this paper. The author also wishes to thank all the reference sources utilized in the preparation of this paper. Lastly, the author sincerely apologizes for any errors that may be present in this paper.

REFERENCES

[1] Munir, Rinaldi. 2023. "Nilai Eigen dan Vektor Eigen (Bagian I)". https://informatika.stei.itb.ac.id/~rinaldi.munir/AljabarGeometri/2023-2024/Algeo-19-Nilai-Eigen-dan-Vektor-Eigen-Bagian1-2023.pdf. Accessed 20 December 2024, 7:30 AM.

[2] Munir, Rinaldi. 2023. "Nilai Eigen dan Vektor Eigen (Bagian 2)". https://informatika.stei.itb.ac.id/~rinaldi.munir/AljabarGeometri/2023-2024/Algeo-20-Nilai-Eigen-dan-Vektor-Eigen-Bagian2-2023.pdf. Accessed 20 December 2024, 8:30 AM.

[3] Munir, Rinaldi. 2023. "Singular Value Decomposition (SVD) (Bagian I). https://informatika.stei.itb.ac.id/~rinaldi.munir/AljabarGeometri/2023-2024/Algeo-21-Singular-value-decomposition-Bagian1-2023.pdf. Accessed 20 December 2024, 11:00 AM.

[4] Munir, Rinaldi. 2023. "Singular Value Decomposition (SVD) (Bagian II). https://informatika.stei.itb.ac.id/~rinaldi.munir/AljabarGeometri/2023-2024/Algeo-22-Singular-value-decomposition-Bagian2 2023.pdf. Accessed 20 December 2024, 12:00 AM.

[5] Munir, Rinaldi. 2024. "Dekomposisi QR)". https://informatika.stei.itb.ac.id/~rinaldi.munir/AljabarGeometri/2024-2025/Algeo-23b-Dekomposisi-QR-2024.pdf. Accessed 20 December 2024, 15:00 AM.

[6] Watkins, David . 2008. "The QR Algorithm Revisited". https://www.researchgate.net/publication/228355060_The_QR_Algorithm_Revisited. Accessed 22 December 2024, 11:00 AM.

[7] Tomar, Peeyush. 2021. "Cascade-based Multimodal Biometric Recognition System with Fingerprint and Face". https://onlinelibrary.wiley.com/doi/abs/10.1002/masy.202000271. Accessed 22 December 2024, 03:00 PM.

[8] Kasban, H. 2015. "Fingerprints verification based on their spectrum". https://www.sciencedirect.com/science/article/abs/pii/S0925231215010103. Accessed 22 December 2024, 07:00 PM.

[9] R, Venugopal. 2010. "Fingerprint Recognition Using Minutia Score Matching". https://www.researchgate.net/publication/220485747_Fingerprint_Recognition_Using_Minutia_Score_Matching. Accessed 22 December 2024, 09:00 PM.

[10] Rich, DJ. 2017. "Singular Value Decomposition and the Fundamental Theorem of Linear Algebra". source: https://truetheta.io/concepts/linear-algebra/svd-and-the-fundamental-theorem-of-linear-algebra/. Accessed 24 December 2024, 09:00 PM.

[11] Varun. 2020. "Cosine similarity: How does it measure the similarity, Maths behind and usage in Python**".** source: https://towardsdatascience.com/cosine-similarity-how-does-it-measure-the-similarity-maths-behind-and-usage-in-python-50ad30aad7db. Accessed 24 December 2024, 09:30 PM.

[12] Van leuken, Rene. 2005. "A Fast and Low Cost SIMD Architecture for Fingerprint Image Enhancement MSc THESIS". source: https://www.researchgate.net/publication/268041882_A_Fast_and_Low_Cost_SIMD_Architecture_for_Fingerprint_Image_Enhancement_MSc_THESIS. Accessed 25 December 2024, 01:30 PM.

PERSONAL STATEMENT

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

Bandung, December 27 2024

Aliya Husna Fayyaza/13523062