

# Relatório do Projeto de LAPR1

## Reconstrução e Identificação de Imagens Utilizando EigenFaces

**Grupo: 02**

**Equipa: TechTitans**

**Turmas: 1DA E 1DB**

1230350, Luís Teixeira  
1241286, Rafael Vieira  
1240729, Rita Oliveira  
1240720, Alexandre Henrique

**12 de janeiro de 2025**

Professora Claudia Lima (CAL)  
Professor Fernando Duarte (FJD)  
Professora Teresa Ferro (TMF)  
Professor Carlos Ferreira (CGF)

## ÍNDICE

<b>1. INTRODUÇÃO .....</b>	<b>II</b>
<b>2. METODOLOGIA DE TRABALHO.....</b>	<b>III</b>
2.1 SCRUM NO DESENVOLVIMENTO DO PROJETO.....	III
2.2 PLANEAMENTO E DISTRIBUIÇÃO DE TAREFAS .....	III
2.3 REFLEXÃO CRÍTICA SOBRE A DINÂMICA DA EQUIPA.....	III
<b>3. EIGENFACES .....</b>	<b>IV</b>
3.1. DECOMPOSIÇÃO PRÓPRIA .....	IV
3.2. EIGENFACES .....	V
3.3. MATRIZ DE COVARIÂNCIA .....	VI
3.4. RECONSTRUÇÃO/REPRESENTAÇÃO DE IMAGENS UTILIZANDO EIGENFACES .....	VI
3.5. IDENTIFICAÇÃO DE IMAGENS UTILIZANDO EIGENFACES.....	VII
<b>4. DESENVOLVIMENTO E IMPLEMENTAÇÃO DA APLICAÇÃO .....</b>	<b>IX</b>
4.1 MENU – MODO INTERATIVO E NÃO INTERATIVO .....	IX
4.2 FUNCIONALIDADE 1 - DECOMPOSIÇÃO PRÓPRIA DE UMA MATRIZ SIMÉTRICA .....	X
4.3 FUNCIONALIDADE 2 - RECONSTRUÇÃO DE IMAGENS A PARTIR DAS EIGENFACES .....	XIII
4.4 FUNCIONALIDADE 3 - IDENTIFICAÇÃO DA IMAGEM MAIS SIMILAR DENTRO DE UMA BASE DE IMAGENS .....	XVIII
4.5 FUNCIONALIDADE 4 - GERAÇÃO DE NOVAS IMAGENS UTILIZANDO EIGENFACES. ....	XX
<b>5. RESULTADOS.....</b>	<b>XXII</b>
5.1 APRESENTAÇÃO DOS RESULTADOS.....	XXII
5.1.1 FUNCIONALIDADE 1 .....	XXII
5.1.2 FUNCIONALIDADE 2 .....	XXIII
5.1.3 FUNCIONALIDADE 3 .....	XXIV
5.1.4 FUNCIONALIDADE 4 .....	XXV
5.2 DISCUSSÃO DOS RESULTADOS .....	XXVI
<b>6 CONCLUSÃO.....</b>	<b>XXVII</b>
<b>REFERÊNCIAS .....</b>	<b>XXVII</b>

## Índice de Figuras

Figura 1 - Modo interativo.....	ix
Figura 2 - Método DecomposeSymmetricMatrix.....	xi
Figura 3 - Métodos getEigenVectors e getEigenValues.....	xi
Figura 4 - Método getValuesAndIndexArray.....	xii
Figura 5 - Método calculateFunction2.....	xiii
Figura 6 - Método getCSVFileNames .....	xiv
Figura 7 - Método getMatricesFromCsvFolder.....	xiv
Figura 8 - Método populateLinearizedImages .....	xiv
Figura 9 - Método calculateMeanVector .....	xv
Figura 10 - Método centralizeImages.....	xv
Figura 11 - Método createSubMatrix.....	xvi
Figura 12 - Método calculateWeights.....	xvi
Figura 13 - Método reconstructImagesWithEigenfaces.....	xvi
Figura 14 - Método calculateMAE .....	xvii
Figura 15 - Método calculateFunction3.....	xviii
Figura 16 - Método identifyClosestImage .....	xix
Figura 17 - Método calculateEuclidianDistance .....	xix
Figura 18 - Método checkCloserVector.....	xx
Figura 19 - Método generateNewImage .....	xxi
Figura 20 - Método creationImage.....	xxi

## 1. Introdução

A unidade curricular **Laboratório/Projeto 1** tem como objetivo consolidar e expandir os conhecimentos adquiridos nas disciplinas do primeiro semestre.

Neste contexto, o projeto desenvolvido ao longo do presente ano letivo teve como objetivo criar uma aplicação informática para a instituição **European Intelligence Agency (EIA)**, centrada na representação e identificação de indivíduos utilizando **Eigenfaces** (vectores próprios), conforme proposto por Turk e Pentland (1991). A EIA dispõe de uma vasta base de dados de imagens de rostos humanos, que pretende compactar e distribuir pelas suas instituições afiliadas em diversos países europeus, de forma a viabilizar a identificação de indivíduos em circulação.

O sistema proposto deve ser capaz de decompor imagens de faces humanas em valores e vetores próprios, permitindo uma representação compacta da base de dados da EIA, de modo a viabilizar a sua distribuição e uso por diferentes instituições europeias. O projeto contempla **quatro funcionalidades principais**: a decomposição própria de uma matriz simétrica, a reconstrução de imagens a partir das Eigenfaces, a identificação da imagem mais similar dentro de uma base de dados e geração de novas imagens utilizando Eigenfaces. Para além disso, foi realizado um estudo de sensibilidade que avalia o impacto do número de Eigenfaces selecionadas no desempenho do sistema, contribuindo para otimizar a representatividade e a eficiência do modelo proposto.

## 2. Metodologia de Trabalho

A metodologia de trabalho adotada para o desenvolvimento do projeto foi fundamentada na **metodologia Scrum**.

O Scrum é uma metodologia ágil que visa otimizar a gestão de projetos especialmente nas áreas de desenvolvimento de software. É baseado em uma abordagem iterativa e incremental, o trabalho é organizado em ciclos curtos chamados **sprints**, permitindo que as equipas entreguem resultados de forma contínua e ajustem o seu curso conforme as necessidades do cliente ou mudanças no mercado. Esta estrutura promove a colaboração entre os membros da equipa, transparência nas decisões e melhoria contínua, contribuindo para um processo de trabalho dinâmico. O Scrum define papéis claros, como o **Product Owner**, o **Scrum Master** e o **Development Team**, e uma série de eventos e artefactos que garantem a eficiência e o foco.

### 2.1 Scrum no desenvolvimento do Projeto

Durante o projeto, a equipa foi organizada em torno de três papéis principais: o Product Owner, que foi responsável por definir as prioridades do backlog; o Scrum Master, com um papel rotativo, que facilitou o processo, removendo obstáculos e assegurando a aplicação das práticas do Scrum; e os Desenvolvedores, que se concentraram na execução das tarefas.

Os trabalhos foram divididos em ciclos curtos de 1 a 2 semanas, chamados sprints, durante os quais as tarefas eram abordadas de forma colaborativa e revisadas no final de cada ciclo. A cada sprint, fizemos reuniões de revisão e retrospectiva para avaliar o progresso, discutir desafios, distribuir tarefas e identificar melhorias no processo, o que garantiu um fluxo constante de desenvolvimento e ajustes rápidos, sempre com foco no produto final.

### 2.2 Planeamento e distribuição de tarefas

Para a gestão do trabalho, recorremos às ferramentas Scrum disponíveis no GitHub, nomeadamente os projetos, de modo a garantir uma distribuição equitativa da carga laboral entre todos os membros da equipa. Adotámos uma abordagem iterativa, dividindo o trabalho em etapas pequenas e bem definidas, assegurando o foco e a progressão gradual do projeto.

Inicialmente, procedemos à distribuição clara de responsabilidades, incluindo a definição de tarefas como o desenvolvimento da interface de interação do utilizador com a aplicação, abrangendo o tipo de menu e os inputs necessários. Seguidamente, avançámos para a implementação da primeira funcionalidade principal. Cada membro da equipa ficou responsável por desenvolver componentes independentes dessa funcionalidade, promovendo o trabalho autónomo. Após a conclusão destas partes, integramos todos os contributos de forma colaborativa na branch principal do projeto ("main"), assegurando que todas as decisões e implementações foram supervisionadas e aprovadas consensualmente por todos os colegas.

Todas as funções seguintes seguiram este mesmo esquema de trabalho, pois este revelou ser bastante efetivo.

### 2.3 Reflexão crítica sobre a dinâmica da equipa

De uma forma geral, a equipa demonstrou um desempenho muito positivo ao longo do projeto. Mantivemos uma comunicação constante e eficaz entre todos os membros, o que facilitou significativamente a execução e a conclusão das tarefas propostas. Sempre que surgiam momentos em

que algum elemento se sentia desmotivado ou sem tarefas atribuídas, a equipa atuava de forma colaborativa para resolver a situação, garantindo o envolvimento e a produtividade de todos.

## 3. Eigenfaces

### 3.1. Decomposição Própria

A **Decomposição Própria** de uma matriz é um processo matemático utilizado para representar uma matriz como o produto de três matrizes:

1. **Matriz de vetores próprios** ( $P$ ): contém os vetores próprios da matriz original como colunas.
2. **Matriz diagonal de valores próprios** ( $D$ ): contém os valores associados aos vetores próprios na diagonal principal.
3. **Matriz transposta de vetores próprios** ( $P^T$ ): a transposta da matriz de vetores próprios.

Formalmente, para uma matriz quadrada  $A$ , a decomposição própria é expressa como:

$$A = PDP^T$$

Este processo é usado em áreas como por exemplo:

- **Compressão de Imagens:** Reduz a dimensão das imagens armazenando apenas os componentes mais significativos.
- **Resolução de Sistemas Lineares:** Permite encontrar soluções eficientes em sistemas dinâmicos
- **Modelação de Sistemas Dinâmicos:** Facilita a análise de estabilidade e comportamento de sistemas complexos.

No caso da nossa aplicação foi solicitado que o utilizador deva escolher a quantidade de vetores próprios a utilizar na decomposição. O motivo da escolha é a performance. Ao escolher menores quantidades de vetores próprios, a performance é melhor, mas a “qualidade” da Decomposição Própria é menor.

A fórmula utilizada é modificada para a seguinte forma:

$$A_k = P_k D_k P_k^T$$

Onde o  $k$  é o número de vetores e valores próprios a utilizar. Os vetores e valores próprios selecionados são os de maior valor.

#### Introdução aos vetores próprios e os seus valores associados (valores próprios)

Os vetores próprios e os seus valores associados são definidos para uma matriz  $A$  como se segue:

- Um vetor  $v$  é chamado de vetor próprio de  $A$  se satisfizer a equação:

$$Av = \lambda v$$

onde  $\lambda$  é o valor associado correspondente ao vetor próprio. (Matos, 2024, p28)

Propriedades de matrizes simétricas relevantes:

1. Todos os valores associados de uma matriz simétrica real são reais.
2. Os vetores próprios de uma matriz simétrica real são ortogonais entre si.
3. Matrizes simétricas podem ser diagonalizadas.

### Exemplo Ilustrativo

Considere a matriz simétrica:

$$A = \begin{bmatrix} 4 & 1 \\ 1 & 3 \end{bmatrix}$$

1. Calcular os valores associados ( $\lambda$ ) resolvendo os valores de  $\lambda$  que satisfazem:

$$\det(A - \lambda I) = 0$$

Resultado:  $\lambda_1 = 5$ ,  $\lambda_2 = 2$ .

2. Determinar os vetores próprios ( $v$ ) para cada valor associado, resolvendo  $(A - \lambda I)v = 0$

Resultado:

$$P = \begin{bmatrix} 1 & -1 \\ 1 & 1 \end{bmatrix}, \quad D = \begin{bmatrix} 5 & 0 \\ 0 & 2 \end{bmatrix}$$

### 3.2. Eigenfaces

As Eigenfaces aplicam o conceito de decomposição própria no reconhecimento facial, utilizando a **Decomposição Própria** para processar e identificar imagens faciais.

#### Funcionamento das Eigenfaces

1. **Recolha de Dados:** Criação de um conjunto de treino com imagens faciais sob condições controladas.
2. **Cálculo da Imagem Média:** Determina-se a média de todas as imagens do conjunto.
3. **Matriz de Covariância:** Com base nas diferenças entre cada imagem e a imagem média, calcula-se a matriz de covariância.
4. **Cálculo dos Vetores Próprios:** Os vetores próprios da matriz de covariância correspondem às Eigenfaces.
5. **Projeção em Espaço Reduzido:** Cada nova imagem é projetada neste espaço reduzido e comparada com os pesos das imagens conhecidas.

#### Propriedades e Benefícios

- Representação compacta: Reduz o tamanho dos dados ao armazenar apenas os padrões principais.
- Robustez: Minimiza o impacto de pequenas variações faciais.
- Eficiência: Reduz o custo computacional em operações de reconhecimento.

### 3.3. Matriz de Covariância

A **matriz de covariância** é um elemento central para calcular as Eigenfaces. Ela descreve como os diferentes pixels de uma imagem variam conjuntamente entre si.

#### Definição

Para um conjunto de  $N$  imagens representadas como vetores  $I_i$ :

1. Calcular a imagem média  $\mu$ :

$$\mu = \frac{1}{N} \sum_{i=1}^N I_i$$

2. Calcular o peso de cada Imagem:  $\Phi_i = I_i - \mu$
3. Construir a Matriz de pesos  $A$ :
4. A matriz de covariância  $C$  é dada por:

$$C = \frac{1}{N} A A^T$$

#### Exemplo

Para duas imagens  $I_1 = [1, 2]$  e  $I_2 = [3, 4]$ :

1.  $\mu = [2, 3]$ .
2. Subtração da média:  $\Phi_1 = [-1, -1]$ ,  $\Phi_2 = [1, 1]$ .
3. Construção da matriz de pesos:  $A = \begin{bmatrix} -1 & 1 \\ -1 & 1 \end{bmatrix}$
4. Calcular a matriz de covariância:

$$C = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}$$

Esta matriz é utilizada para calcular os vetores próprios que formam as Eigenfaces.

### 3.4. Reconstrução/Representação de Imagens utilizando Eigenfaces

A reconstrução de imagens utilizando Eigenfaces baseia-se na utilização de um subconjunto de vetores próprios (Eigenfaces) para representar ou aproximar uma imagem original. Este método é utilizado devido à sua simplicidade e eficácia, reduzindo o tempo de processamento ao trabalhar num espaço dimensional inferior.

#### Passos para Reconstrução:

1. **Transformar cada imagem da base de imagens, com  $N$  imagens, em vetor;**  
Cada imagem é convertida para um formato vetor  $I_i$ , permitindo a sua manipulação matricial.
2. **Construir uma matriz de imagens,  $M$ , em que cada coluna da matriz é uma imagem da base de imagens**
3. **Calcular a Média das Imagens**  
Determina-se o vetor médio  $\mu$ , que é a média de todas as imagens na base de dados
4. **Calcular a Diferença entre Cada Imagem e a Média**  
Para cada imagem  $I_i$ , calcula-se  $\Phi_i = I_i - \mu$



5. **Construir a matriz dos pesos A**

$$A = [\Phi_1, \Phi_2, \dots, \Phi_N]$$

6. **Construir a Matriz de Covariâncias C**

$$C = \frac{1}{N} AA^T$$

Not : Na construção da matriz de covariância como  $AA^T$  e  $A^T A$  têm os mesmos valores próprios podemos utilizar (para a otimização o cálculo, trabalhando com uma matriz de menor dimensão)  $A^T A$  em vez de  $AA^T$

7. **Normalização dos vetores próprios de C**

8. **Reconstruir a Imagem**

A reconstrução de uma imagem  $I_i$  é realizada utilizando um subconjunto de  $k$  Eigenfaces (vetores próprios normalizados) e os respetivos pesos  $w_j$ :

$$I_i = \mu + \sum_{j=1}^n u_j w_j$$

- $\mu$  é o vetor médio,
- $u_j$  são os vetores próprios (Eigenfaces),
- $w_j$  é o peso calculado para cada eigenface  $u_j$ , dado por:  $w_j = u_j^T \Phi_i$

**Avaliação da Qualidade da Reconstrução:**

Para medir a qualidade da reconstrução, utiliza-se uma métrica como o **Erro Absoluto Médio (EAM)** entre a imagem original  $A$  e a reconstruída  $A_k$ :

$$EAM(A, A_k) = \frac{1}{MN} \sum_{i=1}^M \sum_{j=1}^N |A(i, j) - A_k(i, j)|$$

### 3.5. Identificação de Imagens utilizando Eigenfaces

A identificação de imagens utilizando Eigenfaces baseia-se na comparação de pesos calculados para as Eigenfaces de uma imagem alvo com os pesos correspondentes das imagens armazenadas numa base de dados. Este método tira partido da redução dimensional oferecida pela decomposição própria, permitindo que apenas as características mais significativas sejam utilizadas no processo de reconhecimento.

**Processo Detalhado**

O algoritmo para identificação de imagens segue os passos descritos abaixo:

**1. Calcular  $\Phi_{\text{nova}}$  :**

Para a imagem alvo a ser identificada ( $I_{\text{nova}}$ ), subtrai-se o vetor médio ( $\mu$ ) calculado previamente da base de treino:

$$\Phi_{\text{nova}} = I_{\text{nova}} - \mu$$

Este passo centraliza a imagem alvo, garantindo que as diferenças em relação às imagens da base de dados sejam avaliadas de forma consistente.

## 2. Calcular os Pesos $w_{nova,j}$ :

Projeta-se  $\Phi_{nova}$  em cada eigenface  $u_j$  da base, calculando os pesos associados:

$$w_{nova,j} = u_j^T \Phi_{nova}$$

Os pesos representam a contribuição de cada eigenface na reconstrução da imagem alvo.

## 3. Construir os Vetores de Pesos:

Com os pesos calculados para  $\Phi_{nova}$  constrói-se o vetor:

$$\Omega_{nova} = [w_{nova,1}, w_{nova,2}, \dots, w_{nova,k}]$$

Da mesma forma, para cada imagem na base de dados, já previamente processada, temos:

$$\Omega_j = [w_{j,1}, w_{j,2}, \dots, w_{j,k}]$$

O número k representa o subconjunto de Eigenfaces utilizado, geralmente selecionado com base nos maiores valores próprios associados.

## 4. Calcular as Distâncias Euclidianas:

Para comparar  $\Omega_{nova}$  com os vetores de pesos  $\Omega_j$  de cada imagem da base, calcula-se a distância euclidiana:

$$d(\Omega_{nova}, \Omega_j) = \sqrt{\sum_{i=1}^k (w_{nova,i} - w_{j,i})^2}$$

Este valor reflete o grau de similaridade entre a imagem alvo e as imagens da base de dados.

## 5. Identificar a Imagem Correspondente:

A imagem da base associada ao vetor  $\Omega_j$  cuja distância a  $\Omega_{nova}$  seja a menor é devolvida como resultado.

### Uso de Subconjuntos de Eigenfaces

Nem todas as Eigenfaces disponíveis precisam ser utilizadas no cálculo dos pesos e na comparação. Para otimizar o desempenho, pode-se restringir o processo às Eigenfaces associadas aos maiores valores próprios, que são as mais significativas. Esta abordagem equilibra a precisão e o desempenho computacional, especialmente em bases de dados de grande dimensão.

Que no caso da nossa aplicação, o utilizador pode escolher a quantidade de k que deseja, decidindo assim o quão preciso ele quer que a identificação de imagens seja. [2] (Ferreira, 2024)

## 4. Desenvolvimento e Implementação da Aplicação

O objetivo da nossa aplicação era executar quatro funcionalidades principais:

- Funcionalidade 1 - Decomposição própria de uma matriz simétrica
- Funcionalidade 2 - Reconstrução de imagens a partir das Eigenfaces
- Funcionalidade 3 - Identificação da imagem mais similar dentro de uma base de imagens
- Funcionalidade 4 - Geração de novas imagens utilizando Eigenfaces.
- Extras – Opção 5 (conheça a equipa de desenvolvimento!) e Opção 0 (opção para que o utilizador decida quando encerrar a aplicação!).

Nesta secção vamos explorar a forma como o projeto foi desenvolvido mostrando as partes de código representativas das suas respetivas funcionalidades e como as mesmas funcionam.

### 4.1 Menu – modo interativo e não interativo

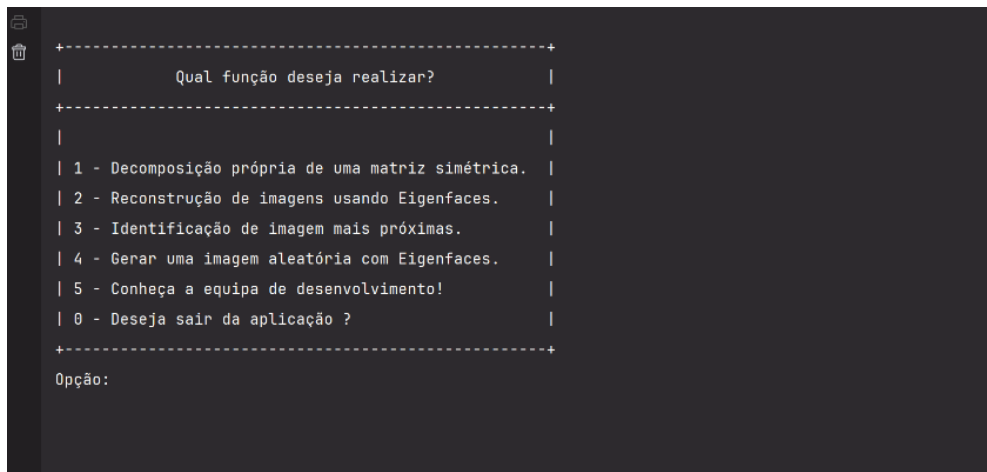
Assim que o programa é inicializado, ele logo verifica se foi inicializado com ou sem argumentos e a partir desta verificação, ele escolhe o modo que o utilizador desejou iniciar.

```
public static void main(String[] args) {  
    if (checkCorrectParametersStructure(args)) {  
        runNonInteractive(args);  
    } else {  
        runInteractive();  
    }  
    scanner.close();  
}
```

(Parte do código)

Para o utilizador da nossa aplicação poder interagir/usar a aplicação propriamente dita, a pedido da **European Intelligence Agency (EIA)** tivemos de criar 2 formas de interação:

- Modo interativo



```
+-----+  
|           Qual função deseja realizar?           |  
+-----+  
|  
| 1 - Decomposição própria de uma matriz simétrica. |  
| 2 - Reconstrução de imagens usando Eigenfaces.    |  
| 3 - Identificação de imagem mais próximas.         |  
| 4 - Gerar uma imagem aleatória com Eigenfaces.    |  
| 5 - Conheça a equipa de desenvolvimento!          |  
| 0 - Deseja sair da aplicação ?                    |  
+-----+  
Opção:
```

Figura 1 - Modo interativo

```
public static void runInteractive() {  
    int function;  
  
    // Roda enquanto a função for inválida  
    function = verifyFunction();  
  
    switch (function) {  
        case 1:
```

```

        function1(function);
        break;
    case 2:
        function2(function);
        break;
    case 3:
        function3(function);
        break;
    case 4:
        function4(function);
        break;
    case 5:
        devTeam();
        break;
    case 0:
        quitApplication();
        break;
    }
}

```

(Parte do código)

Após ser chamado pelo main, o método irá verificar a função escolhida pelo utilizador, permitindo que apenas as opções mostradas sejam escolhidas, para isso, existe um loop dentro do método verifyFunction() que faz o tratamento adequado para essa questão. Logo, modo interativo consiste em um menu em consola que mostra todas as informações necessárias para a escolha da função a ser realizada pela aplicação.

- Modo não interativo

Já o modo não interativo recebe essas mesmas informações como argumentos num único comando na linha de comandos, esse comando e os argumentos sendo da seguinte maneira:

*Java -cp 'nome da biblioteca utilizada' 'nome da classe'.java -f X -k Y -i Z -d W*

*Ex: java -cp "/code/lib/commons-math3-3.6.1.jar" ./code/src/LAPR1\_24\_25\_DAB\_02.java -f 1 -k 123 -i 'Input/Funcao1/exampleInputFunc1.csv' -j 'Input/Funcao2-3/csv'*

- [ -f ] : Este argumento recebe a função a executar.
- [ -k ] : Este argumento recebe a quantidade de vetores próprios a utilizar. (se o valor for -1 ou superior à quantidade de vetores próprios existentes então a quantidade de vetores próprios usados será o máximo).
- [ -i ] : Este argumento recebe a localização da matriz/imagem (em csv) de input a utilizar nas funcionalidades 1 e 3.
- [ -d ] : Este argumento recebe a localização da base de imagens (em csv) de input a utilizar nas funcionalidades 2, 3 e 4.

## 4.2 Funcionalidade 1 - Decomposição própria de uma matriz simétrica

```

public static void function1(int function) {
    int vectorNumbers = verifyVectorNumbers();
    String csvLocation = verifyCsvLocation(function);

    printHeaderFunction("Decomposição Própria de uma Matriz Simétrica:");

    decomposeSymmetricMatrix(vectorNumbers, csvLocation);

    System.out.println();
    System.out.println("Funcionalidade 1 finalizada, a retornar ao menu inicial.");

    runIterative();
}

```

(Parte do código)

Aqui é como a funcionalidade 1 se comporta após, quando executada via *INTERATIVA*.

Este método `fuction1()` é utilizado para dar uma melhor experiência ao utilizador, fazendo com que a aplicação seja bem interativa, mostrando o título da funcionalidade, mostrando opções, tratando possíveis erros/equívocos durante a execução do programa e voltando ao menu inicial quando a execução da funcionalidade 1 for finalizada, fazendo com que a aplicação seja recursiva.

### DecomposeSymmetricMatrix

```
public static void decomposeSymmetricMatrix(int vectorNumbers, String csvLocation) { // LuizGabrielTeixeira
    double[][] oneMatrixCsv = readCSVToMatrix(csvLocation);

    double[][] eigenVectors = getEigenVectors(oneMatrixCsv);
    double[][] eigenValues = getEigenValues(oneMatrixCsv);

    int vectorK = validateEigenVectors(oneMatrixCsv, vectorNumbers);

    double[][] valuesAndIndexArray = getValuesAndIndexArray(eigenValues, vectorK);
    double[][] newEigenVectorsK = createSubMatrix(eigenVectors, valuesAndIndexArray);
    double[][] newEigenValuesK = constructDiagonalMatrix(valuesAndIndexArray);
    double[][] newEigenVectorsTransposeK = transposeMatrix(newEigenVectorsK);
    double[][] matrixEigenFaces = multiplyMatrices(multiplyMatrices(newEigenVectorsK, newEigenValuesK), newEigenVectorsTransposeK);

    double maximumAbsoluteError = calculateMAE(oneMatrixCsv, matrixEigenFaces);
    adjustValue(maximumAbsoluteError);

    printFunction1(vectorK, newEigenValuesK, newEigenVectorsK, maximumAbsoluteError, matrixEigenFaces);
    saveMatrixToFile(matrixEigenFaces, csvLocation, outputFolderPath: "Output/Func1", printOrNot: 1);
}
```

Figura 2 - Método *DecomposeSymmetricMatrix*

Este é o método principal da funcionalidade 1. Nele foram implementadas as várias funções necessárias para calcular os vetores e valores próprios de uma matriz simétrica (decomposição própria), utilizando-os de forma parcial (k vetores) ou não, para reconstruir a matriz original e avaliar o erro absoluto médio (MAE).

```
public static double[][] getEigenVectors(double[][] matrix) { // LuizGabrielTeixeira
    EigenDecomposition eigenDecomposition = decomposeMatrix(matrix);
    RealMatrix eigenVectors = eigenDecomposition.getV();
    return eigenVectors.getData();
}

public static double[][] getEigenValues(double[][] matrix) { // LuizGabrielTeixeira
    EigenDecomposition eigenDecomposition = decomposeMatrix(matrix);
    RealMatrix eigenValues = eigenDecomposition.getD();
    return eigenValues.getData();
}
```

Figura 3 - Métodos *getEigenVectors* e *getEigenValues*

Após a introdução da matriz simétrica, o objetivo dos métodos `getVectors` e `getValues` é fazer a extração dos vetores e valores próprios. Os métodos recebem como parâmetro uma matriz simétrica de números reais `double[][]`.

O método começa por fazer a decomposição espectral da matriz fornecida, ou seja, a matriz fornecida é dividida em dois componentes principais, matriz dos vetores próprios *V* e a matriz dos valores próprios *D*. A partir dessa decomposição, extraímos os vetores ou valores próprios usando respetivamente os métodos `.getV()` e `.getD()`;

(O método `decomposeMatrix` é nos fornecido pela biblioteca `commons-math` da *Apache* na versão 3-3.6.1.).

```

public static double[][] getValuesAndIndexArray(double[][] eigenValuesArray, int eigenfaces) { // LuizGabrielTeixeira +1
    double[][] valuesAndIndexArray = new double[eigenfaces][2];

    for (int i = 0; i < valuesAndIndexArray.length; i++) {
        valuesAndIndexArray[i][0] = Double.MIN_VALUE;
    }

    for (int i = 0; i < eigenValuesArray.length; i++) {
        double absValue = Math.abs(eigenValuesArray[i][i]);
        for (int j = 0; j < valuesAndIndexArray.length; j++) {
            if (absValue > Math.abs(valuesAndIndexArray[j][0])) {
                // Joga os valores para a direita se encontrar um valor maior mais ao fim da matrix
                for (int l = valuesAndIndexArray.length - 1; l > j; l--) {
                    valuesAndIndexArray[l][0] = valuesAndIndexArray[l - 1][0];
                    valuesAndIndexArray[l][1] = valuesAndIndexArray[l - 1][1];
                }
                valuesAndIndexArray[j][0] = eigenValuesArray[i][i];
                valuesAndIndexArray[j][1] = i;
                break;
            }
        }
    }

    return valuesAndIndexArray;
}

```

Figura 4 - Método getValuesAndIndexArray

Este método é extremamente importante na escolha dos maiores K valores próprios da matriz diagonal D. Este foi criado por forma a seleccionar esses valores de uma maneira ordenada, permitindo que os vetores próprios correspondentes sejam utilizados na reconstrução parcial da matriz original, focando-se nos elementos mais significativos.

O método começa por percorrer os valores absolutos presentes na diagonal de D, comparando-os com os valores já armazenados num array bidimensional do tipo double[][] inicializado com os valores mais pequenos possíveis (Double.MIN\_VALUE). Neste array cada posição contém dois elementos: o valor próprio e o índice correspondente na matriz original.

Sempre que é encontrado um valor próprio maior do que o já armazenado, este é inserido no array na respetiva posição, garantindo que os valores fiquem ordenados de forma decrescente quanto à grandeza. Simultaneamente, os valores de menor relevância são deslocados ou removidos, assegurando que apenas os k maiores valores e os respetivos índices sejam preservados no array.

Por fim retorna o array que contém os maiores valores próprios da matriz D e os índices correspondentes, facilitando a seleção dos vetores próprios relevantes para as operações posteriores. Printando na consola as informações solicitadas pela EIA, a quantidade de Vetores K utilizados nessa execução, as matrizes de Vetores e de Valores próprios, a matriz reconstruída, o erro médio absoluto, a informação de onde a matriz resultante foi reconstruída e a informação que a funcionalidade 1 foi finalizada e que estará retornando ao menu inicial.

```

double[][]matrixEigenFaces=multiplyMatrices(multiplyMatrices(newEigenVectorsK,    newEigenValuesK),
newEigenVectorsTransposeK);

saveMatrixToFile(matrixEigenFaces, csvLocation, "Output/Func1", 1);

```

(Parte do código)

Por fim, a reconstrução da matriz utilizando os novos VetoresK, ValoresK e VetoresK transpostos, que sendo uma matriz simétrica, a fórmula:

$$A_k = P_k D_k P_k^T$$

é possível, onde  $A_k$  é a matriz decomposta utilizando K vetores.

### 4.3 Funcionalidade 2 - Reconstrução de imagens a partir das Eigenfaces

```
public static void function2(int function) {
    int vectorNumbers = verifyVectorNumbers();
    String dataBase = verifyDataBaseLocation();

    printHeaderFunction("Reconstrução de Imagens utilizando Eigenfaces");

    calculateFunction2(vectorNumbers, dataBase, function);

    System.out.println();
    System.out.println("Funcionalidade 2 finalizada, a retornar ao menu inicial.");

    runIterative();
}
```

(Parte do código)

Método semelhante ao function1(), apenas mostra e solicita ao utilizador os dados necessários para executar a função, printa informações básicas sobre a função e chama o método calculateFunction2().

```
public static void calculateFunction2(int vectorNumbers, String dataBase, int function) {
    String[] csvFiles = getCSVFileNames(dataBase);
    double[][][] allMatricesCsv = getMatricesFromCsvFolder(dataBase);

    double[][] linearizedImages = new double[allMatricesCsv[0].length * allMatricesCsv[0].length][allMatricesCsv.length];
    populateLinearizedImages(linearizedImages, allMatricesCsv);
    double[] averageVectors = calculateMeanVector(linearizedImages);
    double[][] phi = centralizeImages(linearizedImages, averageVectors);
    int vectorK = validateEigenVectors(linearizedImages, vectorNumbers);

    double[][] phiT = transposeMatrix(phi);
    double[][] phiTxPhi = multiplyMatrices(phiT, phi);
    double[][] eigenVectors = getEigenVectors(phiTxPhi);
    double[][] selectedColumnsK = getValuesAndIndexArray(eigenVectors, vectorK);
    double[][] newEigenVectorsK = createSubMatrix(eigenVectors, selectedColumnsK);
    double[][] expandedVectorsK = multiplyMatrices(phi, newEigenVectorsK);
    double[][] eigenfaces = normalize(expandedVectorsK);
    double[][] weightsMatrix = new double[eigenfaces[0].length][allMatricesCsv.length];

    populateWeightsMatrix(weightsMatrix, phi, eigenfaces);

    reconstructImagesWithEigenfaces(vectorK, csvFiles, averageVectors, eigenfaces, linearizedImages, weightsMatrix, allMatricesCsv, function, phiTxPhi);
}
```

Figura 5 - Método calculateFunction2

Este método tem como objetivo principal calcular e reconstruir imagens usando Eigenfaces, sendo dividido em 6 passos principais:

- **Preparação dos dados:** Carregamento das imagens do dataset e conversão para vetores linearizados.
- **Cálculo da imagem média:** Determinação do vetor médio a partir dos vetores linearizados.
- **Centralização das imagens:** Subtração da média de cada imagem para centralizar os dados.
- **Cálculo das eigenfaces:** Determinação dos vetores próprios (eigenfaces) a partir da matriz de covariância.
- **Normalização das eigenfaces:** Ajuste das eigenfaces para garantir que estejam normalizadas.
- **Reconstrução das imagens:** Uso das eigenfaces e dos pesos calculados para reconstruir ou identificar imagens.

```

public static String[] getCSVFileNames(String folderLocation) {
    File folder = new File(folderLocation);
    File[] csvFiles = folder.listFiles((dir, name) -> name.toLowerCase().endsWith(".csv"));
    if (csvFiles == null || csvFiles.length == 0) {
        errorGeneral("Nenhum arquivo CSV encontrado na pasta: " + folderLocation);
    }

    String[] fileNames = new String[csvFiles.length];
    for (int i = 0; i < csvFiles.length; i++) {
        fileNames[i] = csvFiles[i].getName();
    }

    return fileNames;
}

```

Figura 6 - Método `getCSVFileNames`

Este método tem como objetivo retornar um array de strings com os nomes de todos os arquivos CSV encontrados no diretório especificado. Filtra os arquivos com base na extensão **.csv**, em seguida, extrai e retorna o nome de cada arquivo encontrado.

```

public static double[][][] getMatricesFromCsvFolder(String folderLocation) {
    File folder = new File(folderLocation);
    File[] csvFiles = folder.listFiles((dir, name) -> name.toLowerCase().endsWith(".csv"));
    if (csvFiles == null || csvFiles.length == 0) {
        errorGeneral("Nenhum arquivo CSV encontrado na pasta: " + folderLocation);
    }

    for (int i = 0; i < csvFiles.length - 1; i++) {
        for (int j = i + 1; j < csvFiles.length; j++) {
            if (csvFiles[i].getName().compareTo(csvFiles[j].getName()) > 0) {
                File temp = csvFiles[i];
                csvFiles[i] = csvFiles[j];
                csvFiles[j] = temp;
            }
        }
    }

    double[][][] matrices = new double[csvFiles.length][][];

    for (int i = 0; i < csvFiles.length; i++) {
        File csvFile = csvFiles[i];
        matrices[i] = readCSVtoMatrix(csvFile.getPath());
    }

    return matrices;
}

```

Figura 7 - Método `getMatricesFromCsvFolder`

Este método tem a função de ler todos os arquivos CSV em um diretório especificado, ordená-los alfabeticamente pelo nome do arquivo e, em seguida, carregar cada CSV em uma matriz 2D (2 dimensões) que será armazenada numa matriz 3D (3 dimensões)

- `matrices[i]` é uma **matriz 2D** que representa os dados do arquivo CSV `i`.
- `matrices[i][j]` é uma **linha** da matriz 2D que corresponde a uma linha do arquivo CSV.
- `matrices[i][j][k]` é o **valor específico** na linha `j` e na coluna `k` da matriz 2D do arquivo CSV `i`.

```

double[][] linearizedImages = new double[allMatricesCsv[0].length * allMatricesCsv[0].length][allMatricesCsv.length];

```

```

public static void populateLinearizedImages(double[][] linearizedImages, double[][][] imageMatrices) {
    for (int img = 0; img < imageMatrices.length; img++) {
        double[] linearizedMatrix = matrixToArray1D(imageMatrices[img]);
        for (int i = 0; i < linearizedMatrix.length; i++) {
            linearizedImages[i][img] = linearizedMatrix[i];
        }
    }
}

```

Figura 8 - Método `populateLinearizedImages`

Este método tem como objetivo é preencher a matriz **linearizedImages** com dados extraídos da matriz **ImageMatrices**, convertendo cada matriz 2D de cada imagem em um vetor 1D e armazenando esses vetores em **linearizedImages**. O preenchimento desta matriz dá-se de como que cada linha represente o número total de pixels em uma imagem (o número de elementos em uma matriz 2D "linearizada") e as colunas sejam o número total de imagens.



```

public static double[] calculateMeanVector(double[][] linearizedImages) {
    int numPixels = linearizedImages.length;
    int numImages = linearizedImages[0].length;
    double[] meanVector = new double[numPixels];

    for (int i = 0; i < numPixels; i++) {
        double sum = 0;
        for (int j = 0; j < numImages; j++) {
            sum += linearizedImages[i][j];
        }
        meanVector[i] = sum / numImages;
    }
    return meanVector;
}

```

*Figura 9 - Método calculateMeanVector*

Este método calcula o vetor médio de um conjunto de imagens linearizadas. O objetivo é calcular o **valor médio** de cada pixel em todas as imagens, resultando em um vetor 1D **meanVector** onde cada elemento é o valor médio de um pixel ao longo de todas as imagens.

```

public static double[][] centralizeImages(double[][] images, double[] meanVector) { // LuizGabrielFeixeira +1
    int numPixels = meanVector.length;
    int numImages = images[0].length;
    if (images.length != numPixels) {
        errorGeneral("Para centralizar a imagem o número de pixels na matriz de imagens deve ser igual ao tamanho do vetor médio.");
    }

    double[][] phi = new double[numPixels][numImages];

    for (int i = 0; i < numPixels; i++) {
        for (int j = 0; j < numImages; j++) {
            phi[i][j] = images[i][j] - meanVector[i];
        }
    }

    return phi;
}

```

*Figura 10 - Método centralizeImages*

Este método tem como objetivo **subtrair o valor médio** de cada pixel (presente em meanVector) dos **valores dos pixels nas imagens** ( matriz Images).O método retorna a matriz **phi**, que contém as imagens centralizadas

Seguidamente ocorre a validação do número de EigenVectors da matriz linearizedImages através do método **validateEigenVectors**, cálculo de Phi transposto através do método **transposeMatrix**, e o calculo da matriz Phi transposto e phi através do método **multiplyMatrices**, cálculo dos EigenVectors pelo método **getEigenVectors** e calculo dos EigenValues pelo método **getEigenValues** da matriz PhiTxPhi.

```

public static double[][] createSubMatrix(double[][] eigenVectors, double[][] valuesAndIndexArray) {
    for (int i = 0; i < valuesAndIndexArray.length - 1; i++) {
        for (int j = 0; j < valuesAndIndexArray.length - 1 - i; j++) {
            if (valuesAndIndexArray[j][1] > valuesAndIndexArray[j + 1][1]) {
                double[] temp = valuesAndIndexArray[j];
                valuesAndIndexArray[j] = valuesAndIndexArray[j + 1];
                valuesAndIndexArray[j + 1] = temp;
            }
        }
    }

    double[][] submatrix = new double[eigenVectors.length][valuesAndIndexArray.length];

    for (int i = 0; i < eigenVectors.length; i++) {
        for (int j = 0; j < valuesAndIndexArray.length; j++) {
            submatrix[i][j] = eigenVectors[i][(int) valuesAndIndexArray[j][1]];
        }
    }

    return submatrix;
}

```

Figura 11 - Método createSubMatrix

Neste método é importante realçar a alteração feita à matriz valuesAndIndexArray(ordenação crescente por índice) de forma a manter, na submatrix, os valores nas posições a que originalmente pertenciam excluindo apenas os valores indesejados.

Seguidamente são calculados os **expandedVectorsK** com a multiplicação de **phi** e **newEigenVectorsK** através do método **multiplyMatrices**. Cálculo das **eigenfaces** com a normalização dos expandedVectors através do método **normalize**.

```

public static double[] calculateWeights(double[] phi, double[][] eigenfaces) {
    if (phi.length != eigenfaces.length) {
        errorGeneral("Para calcular os pesos o comprimento de 'phi' deve ser igual a quantidade de linhas da matriz 'eigenfaces'.");
    }

    double[] weights = new double[eigenfaces[0].length];

    for (int j = 0; j < eigenfaces[0].length; j++) {
        weights[j] = 0;
        for (int i = 0; i < eigenfaces.length; i++) {
            weights[j] += phi[i] * eigenfaces[i][j];
        }
    }

    return weights;
}

```

Figura 12 - Método calculateWeights

Método com o objetivo de calcular os pesos através da multiplicação do phi e das eigenfaces que recebe como parâmetro.

```

public static void reconstructImagesWithEigenfaces(int vectorNumbers, String[] csvFiles, double[] averageVectors, double[][] eigenfaces, double[][] linearizedImages,

    printVector("vectorName: \"Valores do vetor médio :\", averageVectors);
    printMatrix(AtxA, "matrixName: \"AtxA \");
    System.out.println("\nQuantidade de Eigenfaces utilizadas: " + vectorNumbers);

    for (int img = 0; img < linearizedImages[0].length; img++) {
        double[] columnWeights = getColumn(weightsMatrix, img);
        double[] reconstructedImage = reconstructImage(averageVectors, eigenfaces, columnWeights, vectorNumbers);
        double[][] reconstructedImageMatrix = array1DToMatrix(reconstructedImage, allMatricesCsv[img]);
        double maximumAbsoluteError = calculateMAE(allMatricesCsv[img], reconstructedImageMatrix);
        adjustValue(maximumAbsoluteError);
        System.out.print("\nPara a imagem: " + csvFiles[img]);
        printVector("vectorName: \", foi utilizado este vetor peso :", columnWeights);
        System.out.printf("O erro absoluto médio dessa imagem com sua original foi: %.3f\n", maximumAbsoluteError);
        saveImage(reconstructedImageMatrix, csvFiles[img], outputFolderPath: "Output/Func2/ImagensReconstruidas", function);
        saveMatrixToFile(reconstructedImageMatrix, csvFiles[img], outputFolderPath: "Output/Func2/Eigenfaces", printOrNot: 0);
    }
}

```

Figura 13 - Método reconstructImagesWithEigenfaces

Este método é responsável por reconstruir imagens a partir de um conjunto de k Eigenfaces e calcular o erro de reconstrução em comparação com as imagens originais. Além disso salva as imagens reconstruídas e as matrizes resultantes em arquivos.

```
public static double calculateMAE(double[][] originalMatrix, double[][] matrixEigenFaces) {
    int rows = originalMatrix.length;
    int columns = originalMatrix[0].length;
    double errorAbsMed = 0;
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < columns; j++) {
            errorAbsMed += Math.abs(originalMatrix[i][j] - matrixEigenFaces[i][j]);
        }
    }
    return errorAbsMed / (rows * columns);
}
```

*Figura 14 - Método calculateMAE*

Este método tem como objetivo calcular o erro absoluto médio entre a imagem original (**originalMatrix**) e a imagem reconstruída com os k eigenfaces (**matrixEigenFaces**)

O método **SaveImage** tem como objetivo guardar uma imagem a partir de uma matriz de valores (**ImageArray**).

A função **saveMatrixToFile** é responsável por salvar uma matriz (**matrix**) num arquivo CSV em um diretório específico. O arquivo gerado será um arquivo de texto no formato CSV, onde cada linha da matriz será escrita em uma linha do arquivo, com os valores separados por vírgula.

#### 4.4 Funcionalidade 3 - Identificação da imagem mais similar dentro de uma base de imagens

```
public static void function3(int function) {
    int vectorNumbers = verifyVectorNumbers();
    String csvLocation = verifyCsvLocation(function);
    String dataBase = verifyDataBaseLocation();

    printHeaderFunction("Identificação da imagem mais próxima utilizando Eigenfaces");

    calculateFunction3(vectorNumbers, csvLocation, dataBase, function);

    System.out.println();
    System.out.println("Funcionalidade 3 finalizada, a retornar ao menu inicial.");

    runIterative();
}
```

(Parte do código)

Semelhante às demais funções que solicitam e mostram ao utilizador as opções e as entradas solicitadas para a execução do método, porém, como é um método que demanda de demasiados cálculos, primeiro é chamado o método que realiza os cálculos para a funcionalidade 3, somente depois, será chamado e enviado para o método principal, o método `identifyClosestImage()`.

Para a realização desta função, o programa irá solicitar ao utilizador basicamente todas as informações possíveis, é o único método que abrange todas as variáveis *'Inputadas'* pelo utilizador presente no programa.

```
public static void calculateFunction3(int vectorNumbers, String csvLocation, String dataBase, int function) {
    String[] csvFiles = getCSVFileNames(dataBase);
    double[][][] allMatricesCsv = getMatricesFromCsvFolder(dataBase);
    double[][] oneMatrixCsv = readCSVToMatrix(csvLocation);

    double[][] linearizedImages = new double[allMatricesCsv[0].length * allMatricesCsv[0].length][allMatricesCsv.length];
    populateLinearizedImages(linearizedImages, allMatricesCsv);
    double[] averageVectors = calculateMeanVector(linearizedImages);
    double[][] phi = centralizeImages(linearizedImages, averageVectors);
    int vectorK = validateEigenVectors(linearizedImages, vectorNumbers);

    double[][] phiT = transposeMatrix(phi);
    double[][] phiTxPhi = multiplyMatrices(phiT, phi);
    double[][] eigenVectors = getEigenVectors(phiTxPhi);
    double[][] selectedColumnsK = getValuesAndIndexArray(eigenVectors, vectorK);
    double[][] newEigenVectorsK = createSubMatrix(eigenVectors, selectedColumnsK);
    double[][] expandedVectorsK = multiplyMatrices(phi, newEigenVectorsK);
    double[][] eigenfaces = normalize(expandedVectorsK);
    double[][] weightsMatrix = new double[eigenfaces[0].length][allMatricesCsv.length];

    populateWeightsMatrix(weightsMatrix, phi, eigenfaces);

    identifyClosestImage(vectorK, csvFiles, averageVectors, eigenfaces, oneMatrixCsv, weightsMatrix, allMatricesCsv, function);
}
```

Figura 15 - Método `calculateFunction3`

Após realizar todos os cálculos necessários para a execução da funcionalidade 3, ele chama o método `identifyClosestImage()`, que irá realizar os cálculos mais exclusivos desta funcionalidade e irá realizar todos os prints solicitados pela EIA.

```

public static void identifyClosestImage(int vectorNumbers, String[] csvFiles, double[] averageVectors, double[][] eigenfaces, double[][] oneMatrixCsv, double[][] weightsMatrix,
double[] linearizedPrincipalImage = matrixToArrayID(oneMatrixCsv);
double[] phiPrincipalImage = subtractionColumns(linearizedPrincipalImage, averageVectors);

double[] principalWeightsVector = calculateWeights(phiPrincipalImage, eigenfaces);

double[] distances = calculateEuclidianDistance(principalWeightsVector, weightsMatrix);
int[] closestImageIndex = checkCloseVector(distances);

System.out.println("O número de vetores próprios utilizados: " + vectorNumbers + "\n");
printVector(vectorName: "Novo vetor Omega (0 nova) :", principalWeightsVector);

int counter = 0;
for (int i = 0; closestImageIndex[i] != Integer.MAX_VALUE; i++) {
    counter++;
}

for (int i = 0; i < counter; i++) {
    double[] closestImageWeights = getColumn(weightsMatrix, closestImageIndex[i]);
    double[] reconstructedImage = reconstructImage(averageVectors, eigenfaces, closestImageWeights, vectorNumbers);
    double[][] reconstructedImageMatrix = arrayIDtoMatrix(reconstructedImage, allMatricesCsv[0]);

    printFunction(csvFiles, closestImageIndex[i], distances, counter, i, closestImageWeights, weightsMatrix);
    saveImage(reconstructedImageMatrix, csvFiles[closestImageIndex[i]], outputFolderPath + "Output/Func3/Identificacao", function);
}
}

```

Figura 16 - Método identifyClosestImage

```

public static double[] calculateEuclidianDistance(double[] principalVector, double[][] weightsMatrix) { // Luiz Gabriel
    if (principalVector.length != weightsMatrix.length) {
        errorGeneral("O comprimento do vetor principal não corresponde ao número de linhas da matriz de pesos.");
    }

    double[] result = new double[weightsMatrix[0].length];
    for (int i = 0; i < weightsMatrix[0].length; i++) {
        double sum = 0;
        for (int j = 0; j < weightsMatrix.length; j++) {
            sum += Math.pow(principalVector[j] - weightsMatrix[j][i], 2);
        }
        result[i] = Math.sqrt(sum);
    }
    return result;
}

```

Figura 17 - Método calculateEuclidianDistance

Este método tem como objetivo calcular a distância euclidiana entre um vetor principal, que representa uma imagem específica, e todos os vetores de pesos armazenados na matriz de pesos. Ele retorna um array unidimensional contendo as distâncias euclidianas calculadas. Essas distâncias são utilizadas para identificar as imagens mais próximas (com menor distância), permitindo determinar qual imagem ou conjunto de imagens compartilha maior semelhança com a entrada fornecida. Depois de retornar o array com as distâncias, o método identifyClosestImage() vai chamar o checkCloseVector() que receberá o array distâncias e calculará se existe mais de uma imagem com as mesmas distâncias.

```

public static int[] checkCloserVetor(double[] distances) {  // LuizGabrielTeixeira +1
    double minDistance = Double.MAX_VALUE;
    int[] closestImageIndex = new int[distances.length];
    fillArrayMax(closestImageIndex);

    int j = 0;
    for (int i = 0; i < distances.length; i++) {
        if (distances[i] < minDistance) {
            minDistance = distances[i];
            j = 1;
            closestImageIndex = new int[distances.length];
            fillArrayMax(closestImageIndex);
            closestImageIndex[0] = i;
        } else if (distances[i] == minDistance) {
            closestImageIndex[j] = i;
            j++;
        }
    }
    return closestImageIndex;
}

```

Figura 18 - Método checkCloserVector

Este método foi concebido de forma a selecionar qual a imagem ou imagens com a menor distância euclidiana. Ele retorna um array unidimensional com a posição de todos os vetores que compartilham a menor distância, sendo um ou mais de um vetor. Depois, como solicitado, o método identifyClosestImage() irá percorrer um ciclo conforme a quantidade de imagens encontradas com a mesma distância.

#### 4.5 Funcionalidade 4 - Geração de novas imagens utilizando Eigenfaces.

```

Public static void function4(int function) {
    int vectorNumbers = verifyVectorNumbers();
    String dataBase = verifyDataBaseLocation();

    printHeaderFunction("Gerar uma imagem aleatória com Eigenfaces");

    generateNewImage(vectorNumbers, dataBase, function);

    System.out.println();
    System.out.println("Funcionalidade 4 finalizada, a retornar ao menu inicial.");

    runIterative();
}

```

(Parte do código)

```

public static void generateNewImage(int vectorNumbers, String dataBase, int function) { 2 usages LuizGabrielTeixeira
    double[][][] allMatricesCsv = getMatricesFromCsvFolder(dataBase);
    double[][] linearizedImages = new double[allMatricesCsv[0].length * allMatricesCsv[0].length][allMatricesCsv.length];
    populateLinearizedImages(linearizedImages, allMatricesCsv);
    double[] meanVector = calculateMeanVector(linearizedImages);
    double[][] phi = centralizeImages(linearizedImages, meanVector);
    int vectorK = validateEigenVectors(linearizedImages, vectorNumbers);

    double[][] phiT = transposeMatrix(phi);
    double[][] phiTxPhi = multiplyMatrices(phiT, phi);
    double[][] eigenVectors = getEigenVectors(phiTxPhi);
    double[][] eigenValues = getEigenValues(phiTxPhi);
    double[][] selectedColumnsK = getValuesAndIndexArray(eigenValues, vectorK);
    double[][] newEigenVectorsK = createSubMatrix(eigenVectors, selectedColumnsK);
    double[][] newEigenValuesK = constructDiagonalMatrix(selectedColumnsK);
    double[][] expandedVectorsK = multiplyMatrices(phi, newEigenVectorsK);
    double[][] eigenfaces = normalize(expandedVectorsK);
    double[][] weightsMatrix = new double[eigenfaces[0].length][allMatricesCsv.length];

    populateWeightsMatrix(weightsMatrix, phi, eigenfaces);
    int dimension = meanVector.length;
    double[] newImage = creationImage(dimension, meanVector, vectorK, newEigenValuesK, eigenfaces);
    double[][] newImageMatrix = array1DToMatrix(newImage, allMatricesCsv[0]);
    System.out.println("A quantidade de Eigenfaces selecionadas para a variável K foi: " + vectorK + "\n");
    saveImage(newImageMatrix, inputCsvPath: "Input/Funcao2-3/csv", outputFolderPath: "Output/Func4", function);
}

```

Figura 19 - Método generateNewImage

Este método é responsável por criar imagens a partir de uma base de imagens já existente e de um conjunto de **k Eigenfaces**. Além disso salva as novas imagens em arquivos.

```

public static double[] creationImage(int dimension, double[] meanVector, int k, double[][] lambdas, double[][] eigenfaces) {
    double[] newImage = new double[dimension];
    for (int i = 0; i < dimension; i++) {
        newImage[i] = meanVector[i];
    }
    for (int i = 0; i < k; i++) {
        adjustPrecision(lambdas);
        double weightsImage = Math.random() * (2 * Math.sqrt(lambdas[i][i])) - Math.sqrt(lambdas[i][i]);
        for (int j = 0; j < dimension; j++) {
            newImage[j] += weightsImage * eigenfaces[j][i];
        }
    }

    return newImage;
}

```

Figura 20 - Método creationImage

Primeiramente, é criado um vetor com a mesma dimensão que o **meanVector**, que representa a média de todas as imagens da base processadas, e em seguida é inicializado com os seus valores. São depois ajustadas as precisões dos **k** valores próprios (**lambdas**), e com base neles são calculados pesos aleatórios, compreendidos entre a raiz negativa e positiva do valor próprio em questão.

Por fim esses pesos são multiplicados às componentes das **Eigenfaces** associadas e adicionados ao vetor da nova imagem.

## 5. Resultados

Nesta secção apresentam-se os testes realizados para quatro funcionalidades, usando como parâmetros de k para cada funcionalidade (os valores máximos e médios podem variar): k mínimo, k médio e k máximo. Tendo como objetivo identificar como cada funcionalidade opera e compreender como o parâmetro k influencia no resultado obtido.









### 5.1 Apresentação dos resultados

#### 5.1.1 Funcionalidade 1

<p><b>Input:</b></p> <p>123,45,67,89,12,34,56,78,90,21,32,43,54,65,76 45,234,78,90,23,45,67,89,12,34,56,78,90,21,32 67,78,200,55,34,56,78,90,23,45,67,89,12,34,56 89,90,55,180,45,67,89,12,34,56,78,90,23,45,67 12,23,34,45,150,78,90,21,43,65,87,12,34,56,78 34,45,56,67,78,140,65,87,23,45,67,89,12,34,56 56,67,78,89,90,65,130,43,54,76,87,21,43,65,87 78,89,90,12,21,87,43,120,32,54,76,98,12,34,65 90,12,23,34,43,23,54,32,110,65,87,12,45,67,89 21,34,45,56,65,45,76,54,65,125,98,32,54,76,12 32,56,67,78,87,67,87,76,87,98,135,43,65,87,21 43,78,89,90,12,89,21,98,12,32,43,145,34,56,78 54,90,12,23,34,12,43,12,45,54,65,34,155,67,89 65,21,34,45,56,34,65,34,67,76,87,56,67,165,12 76,32,56,67,78,56,87,65,89,12,21,78,89,12,175</p>	<p><b>Output:</b></p> <p><b>Com k = 1 (mínimo)</b></p> <p>55, 64, 64, 66, 52, 57, 67, 58, 48, 53, 68, 59, 48, 54, 62 64, 75, 74, 77, 60, 66, 78, 67, 56, 62, 79, 68, 56, 63, 72 64, 74, 74, 76, 60, 66, 77, 67, 56, 62, 78, 68, 56, 62, 71 66, 77, 76, 79, 62, 68, 80, 69, 58, 64, 81, 70, 58, 65, 74 52, 60, 60, 62, 48, 53, 63, 54, 45, 50, 64, 55, 45, 51, 58 57, 66, 66, 68, 53, 59, 69, 60, 50, 55, 70, 61, 50, 56, 64 67, 78, 77, 80, 63, 69, 81, 70, 58, 65, 82, 71, 58, 66, 75 58, 67, 67, 69, 54, 60, 70, 61, 51, 56, 71, 62, 51, 57, 65 48, 56, 56, 58, 45, 50, 58, 51, 42, 47, 59, 51, 42, 47, 54 53, 62, 62, 64, 50, 55, 65, 56, 47, 51, 66, 57, 47, 52, 60 68, 79, 78, 81, 64, 70, 82, 71, 59, 66, 84, 72, 59, 67, 76 59, 68, 68, 70, 55, 61, 71, 62, 51, 57, 72, 63, 51, 58, 66 48, 56, 56, 58, 45, 50, 58, 51, 42, 47, 59, 51, 42, 47, 54 54, 63, 62, 65, 51, 56, 66, 57, 47, 52, 67, 58, 47, 53, 60 62, 72, 71, 74, 58, 64, 75, 65, 54, 60, 76, 66, 54, 60, 69</p> <p><b>Erro Absoluto Médio: 25,74</b></p>
<p><b>Output:</b></p> <p><b>Com k = 8 (médio)</b></p> <p>121, 47, 65, 82, 3, 27, 56, 64, 90, 33, 48, 57, 46, 56, 87 47, 229, 80, 90, 14, 47, 64, 84, 6, 44, 65, 79, 94, 12, 37 65, 80, 197, 55, 34, 54, 83, 96, 22, 44, 61, 87, 9, 40, 57 82, 90, 55, 186, 48, 69, 93, 24, 39, 47, 65, 82, 25, 54, 62 3, 14, 34, 48, 136, 79, 100, 23, 46, 74, 87, 16, 43, 46, 77 27, 47, 54, 69, 79, 126, 59, 88, 25, 53, 72, 96, 4, 32, 65 56, 64, 83, 93, 100, 59, 117, 39, 64, 75, 93, 30, 49, 53, 79 64, 84, 96, 24, 23, 88, 39, 131, 46, 48, 68, 94, 23, 32, 57 90, 6, 22, 39, 46, 25, 64, 46, 110, 61, 73, 8, 55, 68, 77 33, 44, 44, 47, 74, 53, 75, 48, 61, 97, 109, 25, 45, 91, 11 48, 65, 61, 65, 87, 72, 93, 68, 73, 109, 126, 40, 52, 97, 29 57, 79, 87, 82, 16, 96, 30, 94, 8, 25, 40, 146, 36, 57, 69 46, 94, 9, 25, 43, 4, 49, 23, 55, 45, 52, 36, 150, 76, 84 56, 12, 40, 54, 46, 32, 53, 32, 68, 91, 97, 57, 76, 151, 18 87, 37, 57, 62, 77, 65, 79, 57, 77, 11, 29, 69, 84, 18, 182</p> <p><b>Erro Absoluto Médio: 6,31</b></p>	<p><b>Output:</b></p> <p><b>Com k =15 (máximo para este exemplo)</b></p> <p>123, 45, 67, 89, 12, 34, 56, 78, 90, 21, 32, 43, 54, 65, 76 45, 234, 78, 90, 23, 45, 67, 89, 12, 34, 56, 78, 90, 21, 32 67, 78, 200, 55, 34, 56, 78, 90, 23, 45, 67, 89, 12, 34, 56 89, 90, 55, 180, 45, 67, 89, 12, 34, 56, 78, 90, 23, 45, 67 12, 23, 34, 45, 150, 78, 90, 21, 43, 65, 87, 12, 34, 56, 78 34, 45, 56, 67, 78, 140, 65, 87, 23, 45, 67, 89, 12, 34, 56 56, 67, 78, 89, 90, 65, 130, 43, 54, 76, 87, 21, 43, 65, 87 78, 89, 90, 12, 21, 87, 43, 120, 32, 54, 76, 98, 12, 34, 65 90, 12, 23, 34, 43, 23, 54, 32, 110, 65, 87, 12, 45, 67, 89 21, 34, 45, 56, 65, 45, 76, 54, 65, 125, 98, 32, 54, 76, 12 32, 56, 67, 78, 87, 67, 87, 76, 87, 98, 135, 43, 65, 87, 21 43, 78, 89, 90, 12, 89, 21, 98, 12, 32, 43, 145, 34, 56, 78 54, 90, 12, 23, 34, 12, 43, 12, 45, 54, 65, 34, 155, 67, 89 65, 21, 34, 45, 56, 34, 65, 34, 67, 76, 87, 56, 67, 165, 12 76, 32, 56, 67, 78, 56, 87, 65, 89, 12, 21, 78, 89, 12, 175</p> <p><b>Erro Absoluto Médio: 00,00</b></p>



### 5.1.2 Funcionalidade 2

Input: Primeira e segunda imagem da base de imagens utilizada:		
		
Outputs: Com k = 1 (mínimo)		
		
Erro Absoluto Médio:	21,19	22,81
Com k = 20 (médio)		
		
Erro Absoluto Médio:	08,66	09,15
Com k = 40 (máximo)		
		
Erro Absoluto Médio:	03,27	00,11

### 5.1.3 Funcionalidade 3

Input:

Base de imagens:



Imagem a Identificar:



Output:

Com  $k = 1$  (mínimo)



Com  $k = 20$  (médio)



Com  $k = 40$  (máximo)



#### 5.1.4 Funcionalidade 4

Input:

Foi utilizada a mesma base de imagens mencionada na funcionalidade 3.

Output:

Com  $k = 1$  (mínimo)



Com  $k = 20$  (médio)



Com  $k = 40$  (máximo)



## **5.2 Discussão dos resultados**

Os testes realizados nas quatro funcionalidades do programa demonstraram claramente o impacto da escolha do parâmetro K na qualidade dos resultados gerados. Observou-se que:

### **5.2.1 Qualidade das Reconstruções:**

Valores mais altos de K resultaram em reconstruções de maior fidelidade à imagem original, capturando mais detalhes e características relevantes.

Valores baixos de K, por outro lado, levaram a reconstruções mais simplificadas, com perda significativa de detalhes, especialmente em características faciais mais sutis.

### **5.2.2 Balanceamento entre Performance e Qualidade:**

Enquanto K alto proporciona maior precisão, ele também aumenta os requisitos computacionais, tornando o programa um pouco mais lento.

Escolher um K ideal envolve um compromisso entre a qualidade esperada e a performance desejada.

### **5.2.3 Resultados Gerais:**

O programa demonstrou ser eficaz na aplicação dos métodos utilizando Eigenfaces, apresentando resultados consistentes e previsíveis conforme a variação do parâmetro K.

Utilizadores que priorizam alta qualidade devem optar por valores mais altos de K, enquanto aqueles que buscam maior eficiência podem utilizar valores menores com resultados ainda aceitáveis para aplicações menos exigentes.

Essas observações evidenciam a importância de configurar adequadamente o parâmetro K para atender às necessidades específicas de cada caso de uso.

## 6 Conclusão

Este trabalho por fim apresenta o desenvolvimento do projeto focado na implementação de uma aplicação para a European Intelligence Agency (EIA) baseada no conceito de Eigenfaces, conforme proposto por Turk e Pentland (1991).

A metodologia Scrum foi essencial para organizar o desenvolvimento e garantir a entrega contínua de resultados. A dinâmica da equipa foi marcada por boa comunicação, transparência e colaboração, o que contribuiu para a superação de desafios e o alcance dos objetivos estipulados.

Ao longo deste relatório, foram explicadas as quatro funcionalidades principais: a decomposição própria de matrizes simétricas, reconstrução de imagens, identificação de imagens mais próxima e a geração de novas imagens aleatórias.

Todas as funcionalidades demonstraram resultados coerentes e consistentes, comprovando a eficácia da aplicação no processamento e identificação de imagens faciais. Os principais resultados mostram que o número de Eigenfaces selecionado pelo utilizador (parâmetro  $K$ ) tem impacto direto na qualidade e desempenho do sistema. Valores mais altos de  $K$  resultaram em maior precisão na identificação e reconstrução de imagens, enquanto valores mais baixos proporcionaram maior eficiência computacional, mas com perda de detalhes.

O trabalho desenvolvido reflete a eficiência e qualidade do programa, também demonstra um grande potencial de aplicabilidade em sistemas reais para identificação facial, contribuindo para a compactação e distribuição de dados de forma eficiente e segura, alinhado aos objetivos da EIA e abrindo futuras portas para evoluções tecnológicas no ramo.

## Referências

- [1] Matos, J. (2024). *Espaços vectoriais: Álgebra linear e geometria analítica* (pp. 1–25). [Porto]. ISEP-Instituto Superior de Engenharia do Porto.
- [2] Ferreira, C. (2024). *Trabalho de LAPR1* (pp. 1–8). Porto: EIA-European Intelligence Agency.