Giulio Guilherme de Souza Simão

Uma Biblioteca em C++ e Python para Cálculo de Métricas de Distância Baseadas em Estatística

Florianópolis, SC-Brasil
19 de maio de 2021

Giulio Guilherme de Souza Simão

UMA BIBLIOTECA EM C++ E PYTHON PARA CÁLCULO DE MÉTRICAS DE DISTÂNCIA BASEADAS EM ESTATÍSTICA

Desenvolvimento de uma biblioteca que fornece suporte para o cálculo das distâncias de Mahalanobis, Mahalanobis polinomial e Bhattacharyya com enfoque em aplicações de visão computacional.
Orientador: Prof. Dr. rer.net. Aldo von Wangenheim

Florianópolis, SC-Brasil
19 de maio de 2021

,
Uma Biblioteca em C++ e Python para Cálculo de Métricas de Distância Baseadas em Estatística : / Giulio Guilherme de Souza Simão; orientador, Prof. Dr. rer.net. Aldo von Wangenheim. - Florianópolis, SC-Brasil 2021.
122 p.

- Universidade Federal de Santa Catarina, Centro Tecnológico - CTC. Curso de Bacharelado em Ciências da Computação.

Inclui Referências

1.Visão Computacional. 2. Processamento Digital de Imagens. 3. Distância de Mahalanobis. 4. Distância de Mahalanobis Polinomial. 5. Distância de Bhattacharyya. I. , . II. Universidade Federal de Santa Catarina. Curso de Bacharelado em Ciências da Computação. II. Uma Biblioteca em C++ e Python para Cálculo de Métricas de Distância Baseadas em Estatística.

SUMÁRIO

1 INTRODUÇÃO

A noção de distância é algo natural para os seres humanos e a usamos intuitivamente para medir diferenças entre vários tipos de coisas no dia-a-dia. Contudo, a definição matemática atual de distância foi definida há cerca de um século, e ela serviu de base para a definição de várias diferentes métricas para o cálculo de distâncias em diferentes contextos, como a distância de Watson-Crick, que mede a diferença entre sequências de DNA, e a distância cognitiva de Granstrand, que mede a distância entre perfis tecnológicos. [1]

Na área da visão computacional, para extrair informações de imagens buscando regiões que contém uma determinada semântica, utilizam-se técnicas do que se chama de segmentação de imagem, selecionando *pixels* cujos elementos compartilham de uma mesma característica, e agrupando-os em um mesmo segmento, que possui características diferentes dos demais. Para determinar esses segmentos, é comum abstrair-se as *features* de forma que possa ser calculada uma distância entre elas, agrupando-as com base na menor distância. [2] [3]

Em certas situações em que se trabalha com segmentação de imagens, buscar a similaridade entre os *pixels* pode ser mais complicado devido a uma distribuição irregular das características analisadas (e.g. valores RGB). Nesse sentido, a distância euclidana (uma das métricas mais simples e naturalmente mais usada) entre os *pixels* acaba sendo uma métrica não tão eficaz quanto desejado. [3] [4]

Possíveis soluções utilizadas exploram mais a fundo essa noção matemática de análise desses valores abstraídos geometricamente e aplicam outros tipos de distâncias nos algoritmos de segmentação. Algumas distâncias empregadas foram desenvolvidas usando fundamentos matemáticos estatísticos, interpretando os conjuntos de dados com os quais se é trabalhado como populações ou distribuições estatísticas. Três distâncias que são usadas para esse fim, as distâncias de Mahalanobis, de Mahalanobis polinomial e de Bhattacharyya, foram escolhidas dentre as utilizadas para serem exploradas nesse trabalho. [4]

1.1 JUSTIFICATIVA

Apesar de produzirem resultados mais satisfatórios em alguns contextos, não existe tanto suporte para o cálculo dessas medidas em termos de ferramentas de programação quanto se gostaria. Mesmo as implementações existente e disponíveis carecem de uma maior complexidade para um uso mais versátil, sólido e eficiente.

1.2 OBJETIVOS

1.2.1 Geral

Implementar uma biblioteca aberta para C++ e Python que implemente as distâncias de Mahalanobis, de Mahalanobis polinomial e de Bhattacharyya, focando no uso em aplicações de visão computacional.

1.2.2 Específicos

- Desenvolver software para cálculo das distâncias de Mahalanobis, de Mahalanobis polinomial e de Bhattacharyya
- Criar uma documentação para o software desenvolvido para instruir sobre o uso em aplicações que a utilizem
- Submissão de componentes de software e documentação em uma plataforma que a disponibilize abertamente para instalação por terceiros

1.3 CASOS DE USO

Neste capítulo, explicar-se-á exemplos de aplicações em problemas reais do campo da visão computacional das distâncias implementadas neste trabalho.

1.3.1 Distância de Mahalanobis

[5] apresenta um algoritmo para rastreamento de porcos em um alojamento de criação para abate. Ao receber uma entrada contendo os *pixels*

correspondendo aproximadamente a cada porco, cada conjunto de *pixels* correspondendo a um porco específico é armazenado como uma distribuição gaussiana de vetores de cinco dimensões (coordenadas em x e y e valores RGB de cada *pixel*). O programa é executado recebendo imagens em tempo-real do alojamento e consegue atualizar as distribuições calculando a distância de Mahalanobis entre cada pixel de um subconjunto do próximo frame (assumindo a localidade temporal dos porcos) e as distribuições atualmente armazenadas. Os *pixels* que estiverem com as menores distâncias serão escolhidos para formarem a nova distribuição correspondente àquele porco. As imagens, de [5], abaixo demonstram a proximidade de cada pixel à distribuição correspondente a um porco específico, usando a distância de Mahalanobis para as coordenadas (x, y), usando os valores *RGB* e os dois simultaneamente, respectivamente.

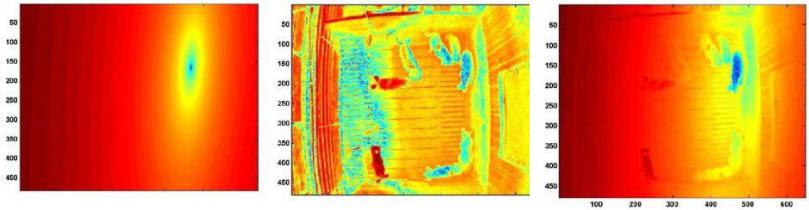

Figura 1 – imagens representando a distância de mahalanobis entre os *pixels* que representam o porco mais à direita, na parte superior, usando métricas espaciais, de cor e ambas, respectivamente (cores mais próximas do vermelho indicam maior distância)

1.3.2 Distância de Mahalanobis Polinomial

Uma aplicação da distância de Mahalanobis polinomial é para detecção de caminhos para robôs de movimento autônomo. [4] usou as imagens de um robô LAGR, que pode determinar alguns pontos correspondentes ao chão sobre o qual ele se desloca e usá-los para construir uma métrica de Mahalanobis polinomial. Assim, efetua-se um cálculo de distância de cada pixel da imagem até o ponto de referência da métrica, e determina-se o caminho a ser seguido como a porção da imagem com a menor distância. Como demons-

trado, a distância de Mahalanobis polinomial obteu um resultado muito mais claro que a distância euclidiana e a distância de Mahalanobis. A figura 2, de [4], mostra os resultados obtidos com aplicações das diferentes distâncias.

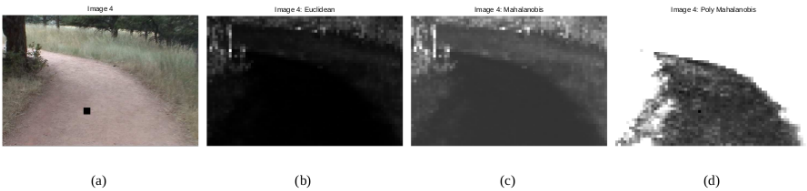

Figura 2 – imagens representando a distância de mahalanobis entre os *pixels* que representam o caminho, indicado em (a), usando distância euclidiana (b), distância de Mahalanobis (c) e distância de Mahalanobis polinomial (d) (*pixels* mais claros representam uma distância maior)

1.3.3 Distância de Bhattacharyya

Um trabalho apresentado em [6] usa distância de Bhattacharyya para detecção de comportamento anormal de multidões em imagens públicas para auxílio em questões de segurança pública. O trabalho usa as imagens de câmeras de segurança e detecta as pessoas presentes e a representa como pontos de interesse, agrupados usando k-médias. O algoritmo utilizado emprega a distância de Bhattacharyya normalizada para calcular a distância entre cada *cluster* de pontos de interesse de um frame das imagens de segurança e cada *cluster* do frame seguinte. Após calcular a média geométrica das distâncias para cada *cluster* e normalizá-la, armazenando-a numa variável chamada de ϕ_β, o algoritmo calcula um threshold, acima do qual o valor de ϕ_β indica um comportamento anormal de multidões.

As figuras abaixo, de [6], representam casos em que ocorreram situações anormais de movimentação de pessoas, que causaram picos nos valores extraídos pelo algoritmo, como indicados nos gráficos.

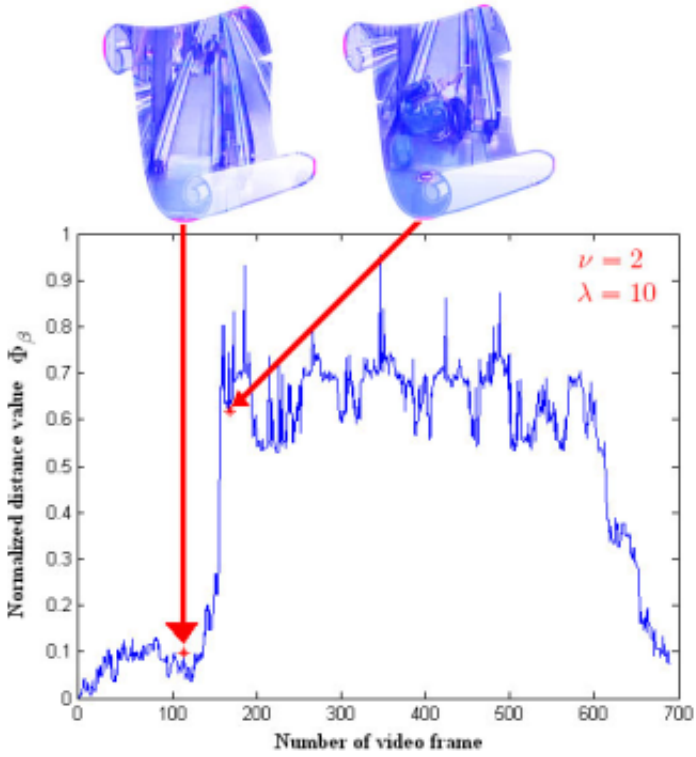

Figura 3 – pico de distância entre *clusters* de pontos de interesse entre frames de queda de escada rolante

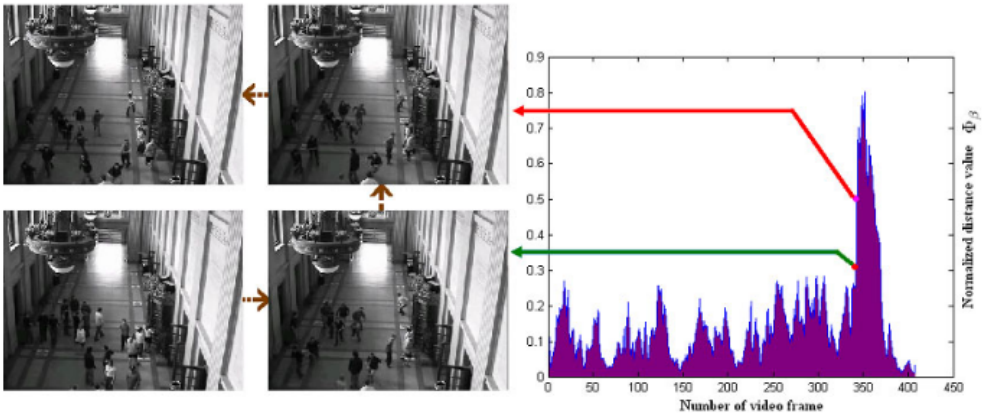

Figura 4 – pico de distância entre *clusters* de pontos de interesse entre frames de corrida

2 FUNDAMENTAÇÃO TEÓRICA

Neste capítulo, serão apresentadas definições matemáticas iniciais para as distâncias que são o foco deste trabalho. O capítulo de desenvolvimento se aprofundará nesses conceitos para apresentar sua solução.

2.1 DISTÂNCIA

Uma função d que mapeia cada par de elementos de um conjunto não-vazio A a um número real e que satisfaz as seguintes propriedades, para todo $x, y, z \in A$:

- $d(x,y) \geq 0$
- $d(x,y) = 0$ se, e somente se $x = y$
- $d(x,y) = d(y,x)$
- $d(x,y) \leq d(x,z) + d(y,z)$

é chamada uma função de distância em A ou métrica em A. [7]

2.2 DISTÂNCIA EUCLIDIANA

Sejam as sequências ordenadas de n números reais, em que n é um número inteiro positivo qualquer, $u_1, u_2, u_3, ..., u_n$ e $v_1, v_2, v_3, ..., v_n$ denotadas por u e v, respectivamente, a distância euclidiana $d(u,v)$ entre u e v é dada por:

$$d(u,v) = \sqrt{\sum_{i=1}^{n} (u_i - v_i)^2}$$

em que s_i é o i-ésimo elemento da sequência ordenada de números reais s.

A distância euclidiana é normalmente usada para medir distâncias espaciais entre objetos do mundo real, usando triplas ordenadas correspondentes a coordenadas tridimensionais de algum sistema de localização espacial, mas a distância euclidiana não é limitada a distâncias espaciais nem a sequências tridimensionais. [8]

2.3 DISTÂNCIA DE MAHALANOBIS

A distância de Mahalanobis pode ser definida como a segunte distância entre duas populações estatísticas: considerando μ_P como a matriz coluna correspondente ao vetor contendo as médias da população P em cada uma de suas dimensões, A e B duas populações estatísticas de mesmas dimensões, e C a matriz de covariância de A e B, tem-se que a distância de Mahalanobis $DM_{(A,B)}$ entre A e B é:

$$DM_{(A,B)} = \sqrt{(\mu_A - \mu_B)^T C^{-1} (\mu_A - \mu_B)}$$

A distância de Mahalanobis também pode ser usada para calcular a distância entre um ponto e uma distribuição estatística, assumindo uma das populações como unitária. [9] [10]

2.4 DISTÂNCIA DE MAHALANOBIS POLINOMIAL

A distância de Mahalanobis polinomial de grau l pode ser definida como uma distância de Mahalanobis entre duas populações, A e B, calculada usando vetores contendo todos os termos das expansões polinomiais de grau l dos elementos dos pontos dessas distribuições em vez de os pontos em si. Por exemplo, a distância de Mahalanobis polinomial de grau 2 entre as populações A e B, sendo $a = (x_1, y_1)$ e $b = (x_2, y_2)$ pontos quaisquer das populações A e B, respectivamente, poderia ser calculada como uma distância de Mahalanobis entre as populações A' e B', sendo $a' = (x_1, y_1, x_1 y_1, x_1^2, y_1^2)$ e $b' = (x_2, y_2, x_2 y_2, x_2^2, y_2^2)$ os pontos correspondentes a a e b nas populações A' e B', respectivamente. [4]

2.5 DISTÂNCIA DE BHATTACHARYYA

A distância de Bhattacharyya é uma métrica de divergência entre duas distribuições de probabilidade, definida a partir de um ângulo entre dois vetores de dimensão k, sendo k a cardinalidade da população sobre a qual essas distribuições são definidas. A distância de Bhattacharyya $DB_{(P,Q)}$ entre as distribuições discretas P e Q, representadas respectivamente pelos vetores p e q, pode ser definida, então, da seguinte maneira:

$$DB_{(P,Q)} = -ln(\sum_{i=1}^{k} \sqrt{p_i q_i})$$

sendo p_i e q_i os elementos da i-ésima dimensão dos vetores p e q, respectivamente. Em outras palavras, a probabilidade do i-ésimo elemento da população sobre a qual são definidas P e Q, dada uma ordem qualquer, sob essas distribuições. [11] [12] [13]

3 REVISÃO

Com esta revisão, espera-se descobrir que implementações estejam (ou não) atualmente disponíveis em bibliotecas abertas. Busca-se, com isso, entender o que é necessário para uma útil implementação das distâncias.

3.1 CRITÉRIOS DA REVISÃO SISTEMÁTICA

Nesta seção, será explicado o método seguindo o qual foi feita a revisão sistemática de literatura, buscando bibliotecas que devam ser analisadas por este trabalho.

3.1.1 Definição da Busca

Seguindo a metodologia de pesquisa adotada em [14], foram definidos os seguintes:

Pergunta: Quais são as implementações disponíveis atualmente de distância de Mahalanobis, distância de Mahalanobis polinomial e distância de Bhattacharyya?

População: Bibliotecas abertas para uso em código escrito em C++ ou em Python.

Intervenção: Análise das bibliotecas disponíveis quem contenham as implementações desejadas.

Resultado: Demonstração da necessidade ou desnecessidade de desenvolver uma biblioteca contendo implementações apropriadas das distâncias de Mahalanobis, de Mahalanobis polinomial e de Bhattacharyya.

Contexto: Veículos de busca Google e Google Scholar.

Para alcançar o objetivo da revisão, efetuou-se uma busca em veículos de busca por trabalhos e publicações em inglês de a partir de 2009. Como está sendo buscada qualquer biblioteca aberta, procurou-se não focar tanto em publicações acadêmicas.

Os termos pesquisados foram ”Mahalanobis distance”, ”Bhattacharyya distance”, ”C++ library” e ”Python library”, para encontrar menções a bibliotecas para essas linguagens em um mesmo contexto que essas distâncias. Para obtenção de um número maior de resultados de pesquisa, foram utiliza-

dos termos de busca menos estritos, que compreendessem diferentes arranjos desses termos em texto que não fossem idênticos às *strings* mencionadas anteriormente, mas que ainda fossem relevantes para a pesquisa. Sendo assim, foram adotados os seguintes critérios:

Inclusão: Menciona alguma biblioteca para alguma linguagem de programação.

Exclusão: A biblioteca mencionada não é para uso em código escrito em C++ ou Python.

3.2 EXECUÇÃO DA BUSCA

As buscas foram realizadas nos meses 6 e 7 do ano de 2019, nos veículos de busca Google e Google Scholar. Em ambos os veículos foi utilizada a seguinte *string* de busca: "mahalanobis | bhattacharyya distance" "c++ AROUND(30) library" | "python AROUND(30) library". As pesquisas foram refinadas para retornarem apenas resultados dos anos de 2009 a 2019. No caso da pesquisa feita utilizando o veículo Google, foram desconsiderados os resultados omitidos automaticamente por similaridade.

Como o objeto de interesse da revisão eram as bibliotecas, não os trabalhos em si, fez-se uma busca apenas pelas bibliotecas citadas, desconsiderando o conteúdo dos trabalhos. Procurou-se por termos em torno das palavras-chave *library* e *libraries* relacionados a bibliotecas para código escrito em C++ ou Python. A tabela abaixo mostra a quantidade de trabalhos ou links retornados por cada veículo, assim como quantos deles incluíam algo de relevante para a revisão e o número de bibliotecas citadas.

Tabela 1 – Links e bibliotecas retornadas por cada veículo de busca

Veículo	**Links retornados**	**Links aproveitados**	**Bibliotecas encontradas**
Google Scholar	384	352	303
Google	129	82	128

Com isso, o próximo passo foi determinar quais das bibliotecas citadas possuíam implementação de alguma(s) das métricas que se pretende implementar. Isso foi feito baixando o código-fonte das bibliotecas em um

computador e executando uma busca recursiva nos diretórios contendo essas bibliotecas, procurando pelas *strings* "bhat" e "maha", sem diferenciação entre letras maiúsculas e minúsculas. Após essa busca, catalogou-se as bibliotecas que continham implementações das métricas desejadas. Abaixo está uma tabela que mostra as bibliotecas escolhidas, as implementações que elas contêm e em que linguagens elas são usadas.

3.3 ANÁLISE DAS BIBLIOTECAS

Nesta seção estão apresentadas as implementações encontradas nas bibliotecas apresentadas na Tabela 2 que possuem relação direta com visão computacional. Discutir-se-á brevemente como foram feitas as implementações e como seu uso pode ser feito ou não, focando em aplicações de visão computacional.

3.3.1 Armadillo

Possui classes para aprendizado de modelos gaussianos e implementa a distância de Mahalanobis dentro de alguns métodos de aprendizado. Não permite, portanto, que o cálculo da distância seja usado para outros fins.

3.3.2 BGSLibrary

A BGSLibrary é uma biblioteca de suporte à OpenCV e serve para seleção de *backgrounds* de imagens. Entretanto, todas as implementações de distância de Mahalanobis dessa biblioteca estão contidas dentro de métodos de atualização de modelos e, portanto, não podem ser usadas dentro de outras aplicações.

3.3.3 GTSAM

A distância de Mahalanobis dessa biblioteca está implementada como um método de uma classe que representa uma distribuição gaussiana ou isotrópica e recebe um vetor de entrada para calcular sua distância a essa distribuição. No momento de sua instanciação, o objeto que representa essa distribuição recebe sua matriz de covariância.

Tabela 2 – Bibliotecas e suas implementações

Biblioteca	**Mahalanobis**	**Maha. Poli.**	**Bhattacharyya**	**Linguagem(ns)**
Armadillo	S	N	N	C++
BGSLibrary	S	N	N	C++
Bliff	S	N	N	C++
bnpy	S	N	N	Python
C++ SOM lib	S	N	N	C++
Cluslib	S	N	N	C++
DAAL	S	N	N	C++, Python
dBoost	S	N	N	Python
fastcluster	S	N	N	Python
FilterPy	S	N	N	Python
FST3	S	N	S	C++
GNU Radio	N	N	S	Python
GPdoemd	S	N	N	Python
GTSAM	S	N	N	C++
Insight Toolkit	S	N	N	C++
libmaxdiv	S	N	N	C++
libpointmatcher	S	N	N	C++
Metric-learn	S	N	N	Python
MLpack	S	N	N	C++
OpenCV	S	N	N	C++, Python
Open3D	S	N	N	C++, Python
Orange	S	N	S	Python
POT	S	N	N	Python
robot_localization	S	N	N	C++
ruptures	S	N	N	Python
PCL	S	N	N	C++
PyMVPA	S	N	N	Python
pyTorch	S	N	N	Python
Scikit-learn	S	N	N	Python
Scipy	S	N	N	Python
Shogun	S	N	N	C++, Python
Spectral	S	N	S	Python
STAIR Vision	N	N	S	C++
Statsmodels	S	N	N	Python
SyncPy	S	N	N	Python

3.3.4 Insight Toolkit

Possui uma classe que armazena uma matriz de covariância, sua inversa e uma média alteráveis, e possui um método de cálculo de distância

entre dois vetores de pontos e um vetor de pontos e sua média. Também possui implementadas classes que calculam pertencimento ou probabilidade e threshold de imagens usando a distância de Mahalanobis. É a implementação mais completa das bibliotecas analisadas.

3.3.5 libmaxdiv

Possui uma implementação da distância de Mahalanobis como um método de uma classe que implementa um estimador de densidade gaussiano, que contém distribuições estatísticas. O método calcula a distância entre dois vetores passados como parâmetros, a partir da distribuição escolhida também nos parâmetros.

3.3.6 libpointmatcher

Essa biblioteca não possui uma implementação propriamente dita da distância de Mahalanobis, mas converte uma das suas distâncias implementadas em distância de Mahalanobis usando um estimador de escala.

3.3.7 Metric-learn

Implementa uma classe que aprende uma métrica de Mahalanobis. Em outras palavras, a distribuição que serve de referência para calcular a distância entre dois pontos precisa ser aprendida e não pode ser construída diretamente pelo programador.

3.3.8 OpenCV

A OpenCV possui uma implementação simples da distância de Mahalanobis que funciona da mesma maneira em C++ e em Python: uma função recebe dois pontos de entrada e uma matriz de covariância e calcula de maneira simples a distância entre esses pontos a partir da matriz usada como referência a uma distribuição.

A distância de Bhattacharyya não é de fato implementada nessa biblioteca, tendo seu nome na verdade associado a uma implementação da distância de Hellinger.

3.3.9 Open3D

Sua implementação de distância de Mahalanobis se dá como um método de uma classe que implementa uma nuvem de pontos genérica, calculando de maneira simples a distância de Mahalanobis entre cada ponto da nuvem e a sua média; não permitindo o cálculo com pontos externos.

3.3.10 Orange

Tanto o cálculo da distância de Mahalanobis quanto o da de Bhattacharyya são implementados dentro de classes que representam seus modelos para uso genérico, mas que não armazenam distribuições ou pontos.

3.3.11 PCL

Nessa biblioteca, a distância de Mahalanobis é acessível através de uma variável que guarda seu valor, mas seu cálculo está dentro de um algoritmo mais complexo e abrangente e, portanto, não pode ser feito sozinho, o que deixa seu uso imprático.

3.3.12 pyTorch

Sua implementação consiste numa função que recebe um *batch* de distribuições e um *batch* de diferenças entre pontos e calcula a distância de Mahalanobis a partir desses.

3.3.13 Scikit-learn

Possui uma classe que implementa o cálculo da distância entre dois pontos em uma função que os recebe como parâmetros, e a classe em si armazena a matriz de covariância inversa da distribuição usada como referência. A classe pode ser construída apenas recebendo a matriz de covariância, calculando sua inversa no momento da instanciação, ou já recebendo a inversa.

3.3.14 STAIR Vision

Essa biblioteca possui uma função que recebe duas distribuições discretas monodimensionais como parâmetros e calcula de maneira simples a

distância entre as duas. A biblioteca, porém, não é atualizada há quase 10 anos.

3.4 CONCLUSÃO DA REVISÃO

Como visto pelo resultado da revisão, expresso na tabela 2, há um repertório considerável de bibliotecas que implementam algumas das distâncias consideradas neste trabalho. Entretanto, nenhuma delas possui todas as implementações desejadas, sendo que a maioria implementa apenas a distância de Mahalanobis; e poucas possuem uma implementação de uso prático para o cálculo das distâncias repetido várias vezes.

Assim, este trabalho visa complementar as implementações existentes e faltantes, desenvolvendo uma biblioteca que possua um suporte mais robusto para de métricas de distâncias de Mahalanobis, de Mahalanobis polinomial e de Bhattacharyya para extenso uso, com foco em aplicações de visão computacional.

4 IMPLEMENTAÇÃO

A biblioteca, batizada de libmahala, foi construída inteiramente em C++, usando a biblioteca OpenCV. Ela é dividida em uma classe para cada distância implementada, além de uma para uma interface gráfica que auxilia a extração de pontos.

4.1 DISTÂNCIA DE MAHALANOBIS

A classe de distância de Mahalanobis foi implementada com base na formulação de [4]. Ela armazena armazena a matriz de covariância inversa e as matrizes e valores necessários para calculá-la. Ela também dispõe de um conjunto de métodos de cálculo de distância, apropriados para diferentes situações.

4.2 DISTÂNCIA DE MAHALANOBIS POLINOMIAL

A classe de distância de Mahalanobis polinomial foi adaptada da implementação usada por [15], adequando seu uso para a biblioteca, tornando-o mais simples e intuitivo. Ela contém uma estrutura similar à da classe de distância de Mahalanobis linear, porém ligeiramente mais enxuta, devido à imutabilidade de seus atributos.

4.3 DISTÂNCIA DE BHATTACHARYYA

A classe de distância de Bhattacharyya foi construída sobre a implementação da distância de Hellinger da biblioteca OpenCV. Como uma pode ser calculada a partir da outra ([16]), a classe realiza o cálculo sobre o resultado da função da OpenCV para obter o valor da distância de Bhattacharyya. Ela também armazena valores usados pela OpenCV para gerar histogramas a partir de matrizes, usadas nesta biblioteca para representar imagens e multiconjuntos de pontos.

4.4 SUPORTE PARA PYTHON

A versão em Python da biblioteca é construída aplicando funções básicas da biblioteca pybind11. Como os tipos suportados no acesso a elementos de matrizes da biblioteca OpenCV são limitados, cada possível invocação dos métodos de *template* da biblioteca puderam ser explicitamente criados no *binding*. A fim de que os argumentos do tipo `cv::Mat` pudessem ser usados de forma transparente pela versão em Python, foram usados arquivos do repositório `https://github.com/edmBernard/pybind11_opencv_numpy`.

5 RESULTADOS

Alguns testes foram realizados para testar a eficiência e a utilidade das distâncias. O teste escolhido para as distâncias de Mahalanobis e Mahalanobis polinomial foi uma limiarização simples na imagem de resultado do cálculo das distâncias entre cada *pixel* e o centro de uma métrica. Para comparação de resultados, o precedimento também foi aplicado a uma imagem de distâncias euclidianas.

As imagens usadas foram providenciadas pelo Laboratório de Processamento de Imagens e Computação Gráfica (LAPIX) da Universidade Federal de Santa Catarina (UFSC). Foram usadas cinco diferentes imagens, medindo a distância de Mahalanobis linear e polinomial de ordens 2, 4 e 6. Os limiares escolhidos foram de 150, 3, 0.25, 1 e 1, respectivamente.

As figuras 5 a 9 mostram as imagens usadas no teste. As figuras 10 a 14, os pixels extraídos de cada imagem. 15 a 19, os resultados de binarização com distância Euclidiana. 20 a 24, distância de Mahalanobis linear. 25 a 29, Mahalanobis polinomial de ordem 2. 30 a 34, ordem 4. 35 a 39, ordem 6.

Para o teste da distância de Bhattacharyya, foram usadas imagens providenciadas por [17] de núcleos celulares coradas com método de Feulgen. O teste buscou selecionar segmentos das imagens que correspondiam a núcleos celulares. As imagens foram segmentadas com o algoritmo Mumford-Shah, e foi usada uma imagem de referência em que os segmentos foram escolhidos manualmente, e então, para cada segmento das outras imagens, foi calculada a distância de Bhattacharyya entre ele e cada segmento da imagem de referência. Se a menor distância fosse aquela a um segmento de núcleo, o segmento testado era escolhido. A figura 40 mostra a imagem escolhida como referência e a figura 41 os segmentos escolhidos dessa imagem. As figuras 42 a 60 mostram as imagens testadas e as figuras 43 a 61 mostram os segmentos escolhidos para essas imagens.

Figura 5 – Imagem A

Figura 6 – Imagem B

Figura 7 – Imagem C

Figura 8 – Imagem D

Figura 9 – Imagem E

Figura 10 – Imagem A

Figura 11 – Imagem B

Figura 12 – Imagem C

Figura 13 – Imagem D

Figura 14 – Imagem E

Figura 15 – Imagem A

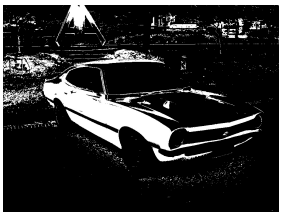

Figura 16 – Imagem B

Figura 17 – Imagem C

Figura 18 – Imagem D

Figura 19 – Imagem E

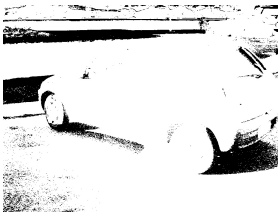

Figura 20 – Imagem A

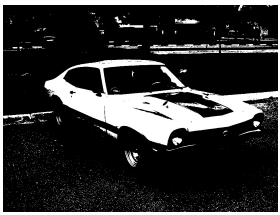

Figura 21 – Imagem B

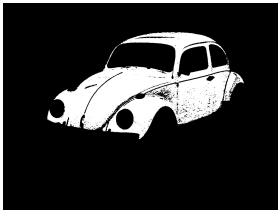

Figura 22 – Imagem C

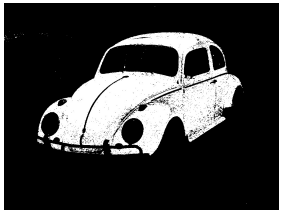

Figura 23 – Imagem D

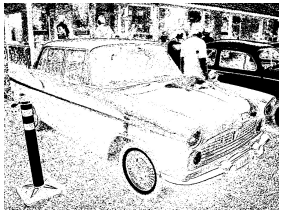

Figura 24 – Imagem E

Figura 25 – Imagem A

Figura 26 – Imagem B

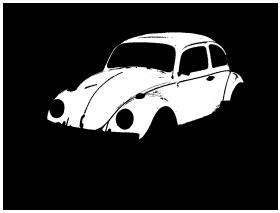

Figura 27 – Imagem C

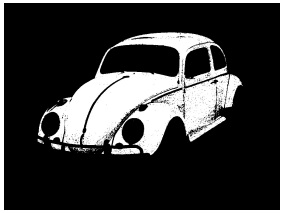

Figura 28 – Imagem D

Figura 29 – Imagem E

Figura 30 – Imagem A

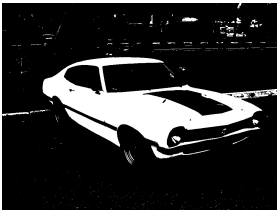

Figura 31 – Imagem B

Figura 32 – Imagem C

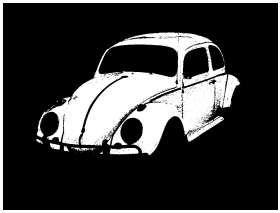

Figura 33 – Imagem D

Figura 34 – Imagem E

Figura 35 – Imagem A

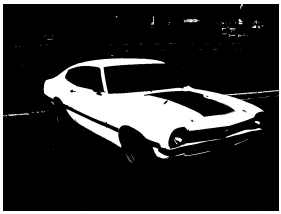

Figura 36 – Imagem B

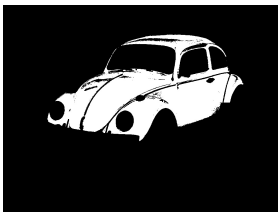

Figura 37 – Imagem C

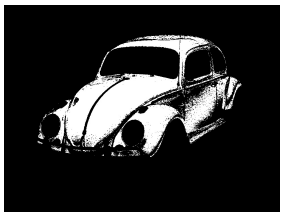

Figura 38 – Imagem D

Figura 39 – Imagem E

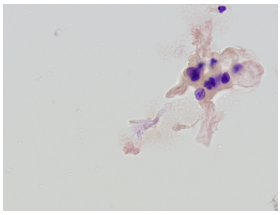

Figura 40 – Imagem de referência

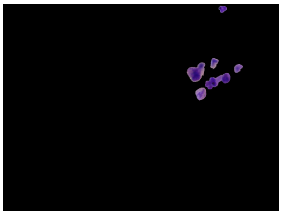

Figura 41 – Segmentos de referência

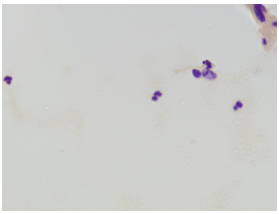

Figura 42 – Imagem testada 1

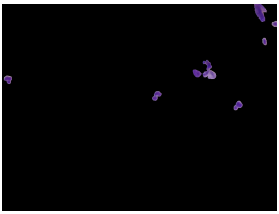

Figura 43 – Segmentos extraídos 1

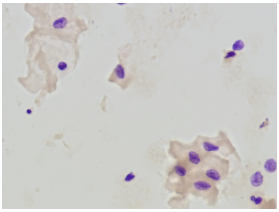

Figura 44 – Imagem testada 2

Figura 45 – Segmentos extraídos 2

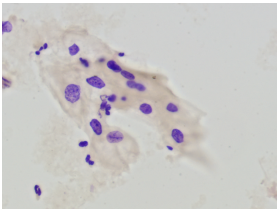

Figura 46 – Imagem testada 3

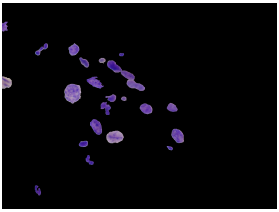

Figura 47 – Segmentos extraídos 3

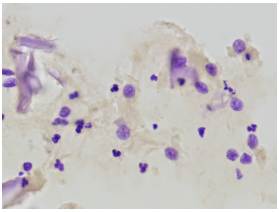

Figura 48 – Imagem testada 4

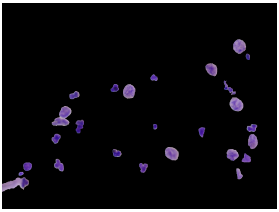

Figura 49 – Segmentos extraídos 4

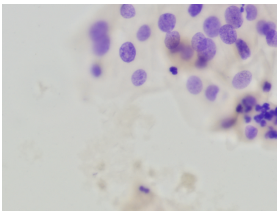

Figura 50 – Imagem testada 5

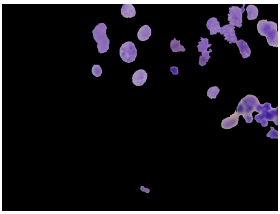

Figura 51 – Segmentos extraídos 5

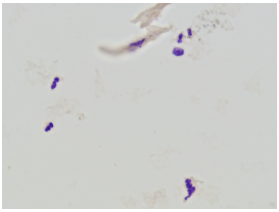

Figura 52 – Imagem testada 6

Figura 53 – Segmentos extraídos 6

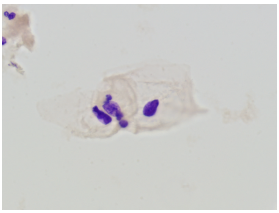

Figura 54 – Imagem testada 7

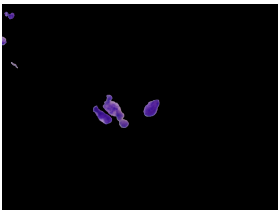

Figura 55 – Segmentos extraídos 7

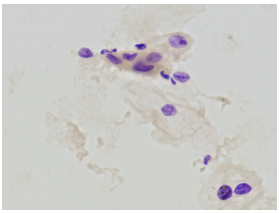

Figura 56 – Imagem testada 8

Figura 57 – Segmentos extraídos 8

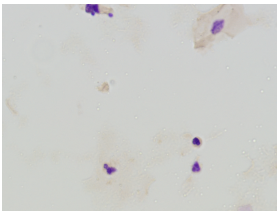

Figura 58 – Imagem testada 9

Figura 59 – Segmentos extraídos 9

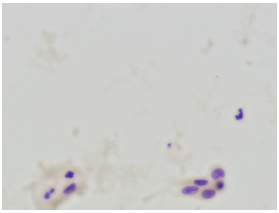

Figura 60 – Imagem testada 10

Figura 61 – Segmentos extraídos 10

6 CONCLUSÃO E TRABALHOS FUTUROS

O uso de diferentes tipos de distâncias é eminente em aplicações de visão computacional. Apesar disso, carece-se de boas bibliotecas que possuam implementações dessas distâncias de maneira que se possa usá-las de maneira em que se tem um maior controle sobre o algoritmo que se deseja implementar.

Este trabalho visou contribuir para o preenchimento dessa lacuna, trazendo a implementação de importantes distâncias na área da visão computacional. Ainda que se espere que essa biblioteca possa ajudar programadores no desenvolvimento de suas aplicações, ainda existe oportunidade para refinamento de suas funcionalidades e ampliação da sua gama de uso.

Espera-se de trabalhos futuros, incluindo futuras versões desta biblioteca, mais distâncias implementadas, mais funcionalidades para extração e manipulação de dados, melhor *performance* e possivelmente algoritmos em que se possa usar as distâncias diretamente.

REFERÊNCIAS

[1] Michel Deza and Elena Deza. *Encyclopedia of Distances* Berlin Heidelberg: Springer-Verlag, 2009.

[2] George Stockman and Linda G. Shapiro. *Computer Vision*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 2001.

[3] Pei-Gee Peter Ho. *Image Segmentation*. INTECH Open Access Publisher, Janeza Trdine 9, 51000 Rijeka, Croatia, 2011.

[4] Greg Grudic and Jane Mulligan. *Outdoor Path Labeling Using Polynomial Mahalanobis Distance*. Robotics: Science and Systems 2006, Philadelphia, PA, USA, August 16-19, 2006.

[5] Peter Ahrendt, Torben Gregersen, Henrik Karstoft. *Development of a real-time computer vision system for tracking loose-housed pigs*. Aarhus School of Engineering, Dalgas Avenue 2, 8000 Aarhus, Denmark, 2011.

[6] Md. Haidar Sharif, Sahin Uyaver and Chabane Djeraba. *Crowd Behavior Surveillance Using Bhattacharyya Distance Metric*. In: Barneva R.P., Brimkov V.E., Hauptman H.A., Natal Jorge R.M., Tavares J.M.R.S. (eds) Computational Modeling of Objects Represented in Images. CompIMAGE 2010. Lecture Notes in Computer Science, vol 6026. Springer, Berlin, Heidelberg.

[7] John Robilliard Giles. *Introduction to the Analysis of Metric Spaces*. Cambridge University Press, The Pitt Building, Trumpington Street, Cambridge CB2 1RP, 1987.

[8] Howard Anton. *Elementary Linear Algebra*. John Wiley & Sons, New York, 1987.

[9] Prasanta Chandra Mahalanobis. *On the generalised distance in statistics*. Proceedings of the National Institute of Sciences of India. 2 (1): 4955. Retrieved 2016-09-27.

[10] Geoffrey John McLachlan. *Mahalanobis Distance*. Resonance. 1999. 4. 20-26. 10.1007/BF02834632.

[11] Frank J. Aherne, Neil A. Thacker and Peter I. Rockett. *The Bhattacharyya metric as an absolute similarity measure for frequency coded data*. Kybernetika, 32(4), 17, 1997.

[12] Bidyut Kr. Patraa, Raimo Launonena, Ville Ollikainena and Sukumar Nandi. *A new similarity measure using Bhattacharyya coefficient for collaborative filtering in sparse data*. KNOSYS 3090. 2015. 82. 163-177. doi:10.1016/j.knosys.2015.03.001.

[13] Thomas Kailath. *The Divergence and Bhattacharyya Distance Measures in Signal Selection* Stanford University, Stanford, Calif., and Stanford Research Institute, Menlo Park, Calif. Recd 4/6/156; revised 10/14/66. Paper 19TP67-906. IEEE TRANS. ON CDMMUNICATION TECHNOLOGY, 15-1, February 1967, pp. 52–60.

[14] Barbara Kitchenham. *Procedures for Performing Systematic Reviews*. Keele, UK, Keele Univ.. 33. 2004.

[15] L. E. Carvalho, S. L. Mantelli Neto, A. C. Sobieranski, E. Comunello and A. von Wangenheim. *Improving Graph-Based Image Segmentation Using Nonlinear Color Similarity Metrics*. International Journal of Image and Graphics Vol. 15, No. 4 (2015) 1550018. DOI:10.1142/S0219467815500187.

[16] Roma Goussakov. *Hellinger Distance-based Similarity Measures for Recommender Systems*. Umeå University, Faculty of Social Sciences, Umeå School of Business and Economics (USBE), Statistics. 2020.

[17] Antonio Buschetto Macarini, L., Von Wangenheim, A., Perozzo Daltoé, F., Sherlley Casimiro Onofre, A., Botelho de Miranda Onofre, F., & Ricardo Stemmer, M. (2020). *Towards a Complete Pipeline for Segmenting Nuclei in Feulgen-Stained Images.* Anais do Computer on the Beach, 11(1), 169-175. doi:https://doi.org/10.14210/cotb.v11n1.p169-175

APÊNDICE A – MANUAL DE INSTALAÇÃO

Libmahala

libmahala is a library built upon OpenCV with support for C++ and Python and that offers means for calculating: * Mahalanobis distance * Polynomial Mahalanobis distance * Bhattacharyya distance

Currently, the library only has support for 64-bit architectures of Linux.

The documentation for the C++ version is at the documentation.pdf file in the repository. The architecture for the Python version is exactly the same as the one for the C++ version, with the appropriate type changes.

There is a short tutorial for all the functionalities of the library in the usage_tutorial.ipynb file, which contains a jupyter notebook with examples for each functionality of the library in the Python version.

Provided that the user is familiarized with the basics of the OpenCV library in both C++ and Python, they should be able to understand both the documentation and the tutorial and then use the library in their applications, no matter if it is written in C++ or Python.

The installation of libmahala is dependent on cmake and OpenCV, and the Python version requires Python3.8 or greater.

OpenCV can be installed by executing the following terminal commands:

```
$ sudo apt update && sudo apt install -y cmake g++ wget unzip
$ wget -O opencv.zip  https://github.com/opencv/opencv/archive/master.zip
$ unzip opencv.zip
$ mkdir -p build && cd build
$ cmake  -DOPENCV_GENERATE_PKGCONFIG=ON ../opencv-master
$ cmake --build .
```

Note that this will download the source code of the library in whatever directory you are executing the commands in. For more information about the installation of OpenCV, please check https://docs.opencv.org/master/d7/d9f/tutorial_linux_install.html.

With OpenCV installed, libmahala can be installed by executing the following terminal commands:

```
$ git clone https://github.com/ggsimao/libmahala
$ cd libmahala
$ git submodule update --init --recursive
$ mkdir build
$ cd build
$ cmake ..
$ make
$ sudo make install
```

As with OpenCV, this will download the contents of this repository into the directory in which you are executing the commands. If you don't want to install

the Python version, add the `-DWITH_PYTHON=0` flag in the `cmake ..` command. If you don't want to install the C++ version, add the `-DWITH_CPP=0` flag.

After installed the library, one can find the necessary headers in the `libmahala` directory and include them in their C++ code like so:

```
#include "libmahala/necessaryHeader.hpp"
```

The code can then be compiled with g++ by adding `` `pkg-config --cflags --libs opencv4 libmahala` `` to your compile command. For instance:

```
g++ main.cpp `pkg-config --cflags --libs opencv4 libmahala` -o program
```

In Python, the library can simply be imported with the `import libmahalapy` line.

APÊNDICE B – MANUAL DE USO

usage_tutorial

May 18, 2021

This is a tutorial for the libmahala library in Python. The usage of the library in C++ can be easily adapted from this tutorial, so long as the user has a basic knowledge of the OpenCV library in both languages.

As the library is aimed at helping developers and scientists in CV applications by giving them support in the process of extract the desired pixels of an image and then using them in the calculation of the implemented distances, this tutorial will exemplify a basic flow of usage for all the classes. The following pieces of code will serve to load images from the file system, extract pixels from them, and calculate the distances between images and points.

As this library is integrated with OpenCV, it is expected that images should be read using the appropriate OpenCV functions. For simply extracting its pixels as points, however, this is not necessary, as the PointCollector class can be instantiated with an image or by using it as one would use the cv2.imread() function. Note that the flags should be inserted as a number.

When using the PointCollector interface, one should use the left mouse button for collecting the points, and the right mouse button for collecting a single reference point. The collected points highlight can be toggles with the 's' key. The process can be restarted with the 'r' key. The color of the highlight can be changed with the 1-6 keys, plus the 'Shift' and 'Alt' keys for speeding up the color change. When all the desired points have been collected, the window can be closed with the 'Esc' key.

[ ]:
```
import libmahalapy as lm
import cv2

pointsCar1 = lm.PointCollector("images/PICT0019.JPG", 1)

car2 = cv2.imread("images/PICT0020.JPG", cv2.IMREAD_COLOR)
pointsCar2 = lm.PointCollector(img)
```

After collecting the desired points, they can be retrieved as such:

[ ]:
```
pixelsCar1 = pointsCar1.collectedPixels
pixelRefCar1 = pointsCar1.referencePixel
cooridnatesCar1 = pointsCar1.collectedCoordinates
cooridnateRefCar1 = pointsCar1.referenceCoordinate

pixelsCar2 = pointsCar2.collectedPixels
pixelRefCar2 = pointsCar2.referencePixel
cooridnatesCar2 = pointsCar2.collectedCoordinates
```

```
cooridnateRefCar2 = pointsCar2.referenceCoordinate
```

Here is a piece of code for checking the values:

[ ]:
```
print("pixelsCar1:")
pixelsCar1
print("pixelRefCar1:")
pixelRefCar1
print("cooridnatesCar1:")
cooridnatesCar1
print("cooridnateRefCar1:")
cooridnateRefCar1
print("pixelsCar2:")
pixelsCar2
print("pixelRefCar2:")
pixelRefCar2
print("cooridnatesCar2:")
cooridnatesCar2
print("cooridnateRefCar2:")
cooridnateRefCar2
```

With the images loaded and the points collected, we can calculate the distances between them. The following piece of code will construct Mahalanobis metrics, one linear and one polynomial of order 2, for each collection of points, and then calculate the distance between the metrics' centers, the population used to construct one and the center of the other, and the image used to collect the population of one and the center of the other. When instantiating the Mahalanobis distances, a small parameter can be added to guarantee that the covariance matrix will be inversible (smin for the Mahalanobis distance, eps_svd for the polynomial Mahalanobis distance), but it is important to note that it might alter the results.

[ ]:
```
mdCar1 = lm.MahalaDist(pixelsCar1, reference=PixelRefCar1)
mdCar2 = lm.MahalaDist(pixelsCar2, reference=PixelRefCar2)
mdCar1.build()
mdCar2.build()

distCenters1 = mdCar1.pointTo(pixelRefCar1, mdCar2.reference)
distPopCenter1 = mdCar1.pointsTo(pixelsCar1, mdCar2.reference)
car1 = cv2.imread("images/PICT0019.JPG", cv2.IMREAD_COLOR)
distImageCenter1 = mdCar1.imageTo_uchar(car1, mdCar2.reference)
distCenters2 = mdCar2.pointTo(mdCar2.reference, mdCar1.reference)
distPopCenter2 = mdCar2.pointsTo(pixelsCar2, mdCar1.reference)
distImageCenter2 = mdCar2.imageTo_uchar(car2, mdCar1.reference)

pmd2Car1 = lm.PolyMahalaDist(pixelsCar1, order=2, reference=PixelRefCar1)
pmd2Car2 = lm.PolyMahalaDist(pixelsCar2, order=2, reference=PixelRefCar2)

dist2Centers1 = pmd2Car1.pointTo(pmd2Car1.reference, pmd2Car2.reference)
dist2PopCenter1 = pmd2Car1.pointsTo(pixelsCar1, pmd2Car2.reference)
```

```
dist2ImageCenter1 = pmd2Car1.imageTo_uchar(car1, pmd2Car2.reference)
dist2Centers2 = pmd2Car2.pointTo(pmd2Car2.reference, pmd2Car1.reference)
dist2PopCenter2 = pmd2Car2.pointsTo(pixelsCar2, pmd2Car1.reference)
dist2ImageCenter2 = pmd2Car2.imageTo_uchar(car2, pmd2Car1.reference)
```

Here is a piece of code for checking the values:

[ ]:
```
print("distCenters1:")
distCenters1
print("distPopCenter1:")
distPopCenter1
print("distCenters2:")
distCenters2
print("distPopCenter2:")
distPopCenter2
print("dist2Centers1:")
dist2Centers1
print("dist2PopCenter1:")
dist2PopCenter1
print("dist2Centers2:")
dist2Centers2
print("dist2PopCenter2:")
dist2PopCenter2
```

The distance between image and point can be visualized as an image:

[ ]:
```
cv2.imshow("image1", distImageCenter1)
cv2.imshow("image2", distImageCenter2)
cv2.imshow("image3", dist2ImageCenter1)
cv2.imshow("image4", dist2ImageCenter2)

cv2.waitKey(0)
```

As an example of usage of these distances, we can segment the images with a simple inverse thresholding:

[ ]:
```
_, car1thresh = cv2.threshold(car1, 127, 255, cv2.THRESH_BINARY_INV)
_, car2thresh = cv2.threshold(car2, 127, 255, cv2.THRESH_BINARY_INV)

cv2.imshow("image1thresh", car1thresh)
cv2.imshow("image2thresh", car2thresh)

cv2.waitKey(0)
```

When it comes to the Bhattacharyya distance, the class can be instantiated similar to the cv2.calcHist() function. The results depend on the instantiation parameters, so tweaking them can render different results. Then, we can calculate the distance between collections of points, images and histograms. Note that, since the constructor parameters serve to calculate a histogram

from an image or a collection of points, the method to calculate the Bhattacharyya distance between histograms does not need an object of the class, and thus is static.

[ ]:
```
channels = [0, 1, 2]
histSize = [256, 256, 256]
ranges = [0, 256, 0, 256, 0, 256]
bd = lm.BhattaDist(channels, histSize, ranges)

distImgs = bd.calcBetweenImg(car1, car2)
distPoints = bd.calcBetweenPoints(pixelsCar1, pixelsCar2)

hist1 = cv2.calcHist([car1], channels, None, histSize, ranges)
hist2 = cv2.calcHist([car2], channels, None, histSize, ranges)
distHists = BhattaDist.calcBetweenHist(hist1, hist2)
```

Here is a piece of code for checking the values:

[ ]:
```
print("distImgs:")
distImgs
print("distPoints:")
distPoints
print("distHists:")
distHists
```

As with the cv2.calcHist() function, the calcBetweenImg() method can also receive masks as arguments to filter what pixels will be selected. In the following example, the thresholded segments from one of the previous examples will serve as masks:

[ ]:
```
distImgsMasks = bd.calcBetweenImg(car1, car2, car1thresh, car2thresh)
distImgsMasks
```

APÊNDICE C – ARTIGO

Revista de Informática Teórica e Aplicada - RITA - ISSN 2175-2745
Vol. XX, Num. XX (2018) 11-XX

RESEARCH ARTICLE

A Library for C++ and Python Based on OpenCV for Calculating the Mahalanobis Distance, the Polynomial Mahalanobis Distance, and the Bhattacharyya Distance

Uma Biblioteca para C++ e Python Baseada em OpenCV para o Cálculo da Distância de Mahalanobis, da Distância de Mahalanobis Polinomial e da Distância de Bhattacharyya

Giulio Guilherme de Souza Simão[1]*, Aldo von Wangenheim[1,4]

Abstract: In this paper, we propose a library that implements the Mahalanobis distance, the polynomial Mahalanobis distance, and the Bhattacharyya distance to aid in the development of computer vision applications. We present the importance of these distances in the context of the field of computer vision, and we attest the lack of tools for the use of these distances in computer vision applications through a systematic literature review, concluding that the library is necessary to complement the attested lack of tools. We developed the library in the C++ and Python programming languages, using the OpenCV library as a basis, due to the fact that it was the most mentioned library for both languages according to the conducted literature review. We perform a few simple experiments to demonstrate that possibilities of use of this library are in accord of what is expected.
Keywords: Computer Vision — Image Segmentation — Digital Image Processing — Mahalanobis Distance — Bhattacharyya Distance — Polynomial Mahalanobis Distance

Resumo: Neste artigo, propomos uma biblioteca que implementa as unções de distância de Mahalanobis, de Mahalanobis polinomial, e de Bhattacharyya para auxiliar no desenvolvimento de aplicações de visão computacional. Neste artigo, apresentamos a importância dessas distâncias no contexto da área de visão computacional, e atestamos a falta de ferramentas para o uso dessas distâncias em aplicações de visão computacional através de uma revisão sistemática de literatura. Desenvolvemos a biblioteca nas linguagens de programação C++ e Python, usando a biblioteca OpenCV como base por ser a biblioteca mais mencionada para ambas as linguagens escolhidas de acordo com a revisão de literatura realizada. Realizamos simples experimentos para demonstrar que as possibilidades de uso da biblioteca estão de acordo com o que é esperado.
Palavras-Chave: Visão Computacional — Segmentação de Imagens — Processamento Digital de Imagens — Distância de Mahalanobis — Distância de Bhattacharyya — Distância de Mahalanobis Polinomial

[1] *Department of Informatics and Statistics, Federal University of Santa Catarina, Brazil*
[4] *Brazilian Institute for Digital Convergence, Federal University of Santa Catarina, Brazil*
***Corresponding author**: cd_giulio@hotmail.com
DOI: http://dx.doi.org/10.22456/2175-2745.XXXX • **Received:** dd/mm/yyyy • **Accepted:** dd/mm/yyyy
CC BY-NC-ND 4.0 - This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License.

1. Introduction

The area of computer vision (CV) consists of making decisions about real and concrete objects and scenes based on sensed images. [1] In CV applications and experiments (referenced hereinafter as just applications), an important step is called image segmentation, in which an image is partitioned into segments whose pixels have similar features and are different from pixels in adjacent segments. [2]

In many algorithms for image segmentation, the notion of the pixel as a point in space is used for measuring the similarity between them, much like measuring the spatial distance between points, more specifically, the Euclidean distance. However, due to the fact that the Euclidean distance is uniformly distributed around a point, it can lead to undesired results when the pixels of a desired segment are not uniformly distributed. To correct this, distances that pull mathematical resources from Statistics can be used to achieve better results. [3] Among these, the Mahalanobis distance, the polynomial Mahalanobis distance, and the Bhattacharyya distance.

Despite their usefulness, there are few libraries aimed at CV applications that give support for extended usage of these distances. Most times, programmers and scientists have to implement their own version of these distances to use in their work, which leads to time that could be used to further

develop the work being wasted unnecessarily. The goal of this work is to help filling this void by creating a library that can be easily integrated into their applications. The chosen technologies to be used in this work are the C++ and Python programming languages, for their frequency of use in CV applications, and the OpenCV library, which is among the most used libraries for CV applications and has support for both of the aforementioned languages.

This paper is organized in the following way: section 2 gives a review of what already exists in terms of what this work proposes to do; section 3, a more in-depth explanation about the most important mathematical concepts used in this work; section 4, an overview of the implementation and architecture of the proposed library; section 5, a few simple examples of usage of the library; and section 6, the conclusions of this paper.

2. State of the Art

For evaluating the necessity of a contribution in the form of a library for calculating these distances in CV applications, a systematic literature review (SLR) was performed, following the methodology proposed by [12].

The review was conducted in 2019 and searched for mentions of libraries for the C++ programming language, libraries for the Python programming language, the Mahalanobis distance, and the Bhattacharyya distance, in the search engines Google and Google Scholar that were published from 2009 and 2019. It yielded 303 mentions to libraries from Google Scholar and 128 mentions from Google, of which only 35 were of relevance for this work. In other words, were open-sourced libraries for C++ and/or Python that included mentions to Mahalanobis or Bhattacharyya in some form. Of these libraries, only 14 were related to CV. One of the most mentioned library in this review was the OpenCV library, which was chosen as a basis for the library proposed by this work for both its usage and language support.

The majority of the 35 related libraries had some implementation of the Mahalanobis distance, a handful had some implementation of the Bhattacharyya distance, and none had an implementation of the polynomial Mahalanobis distance. However, even among those that had an implementation, these were usually employed exclusivelly in another functionality of the library, meaning that they couldn't be used alone or in a different algorithm. Of the libraries found, the one that is closest to what this proposes is the Scikit-learn library, for the Python language. It has a Mahalanobis distance class which can be instantiated with a covariance matrix as a parameter, or its inverse. Then, distances between points can be calculated using the instantiated object by calling one of its methods and only providing the points as parameters.

3. Mathematical Formulations

3.1 Distance Function

A distance function (also called a metric) d is a function $d : A \times A \to [0, \infty)$ which satisfies the following properties for every $x, y, z \in A$:

- $d(x,y) = 0 \iff x = y$
- $d(x,y) = d(y,x)$
- $d(x,y) \leq d(x,z) + d(y,z)$

These properties are called, respectively, identity of indiscernibles, symmetry, and triangle inequality. It is important to note that, while some functions may not satisfy one of these properties fully, they can still be useful for measuring similarities. [4]

3.2 Euclidean Distance

Let $u = (u_1, u_2, u_3, ..., u_n)$ and $v = (v_1, v_2, v_3, ..., v_n)$ be n dimensional column vectors such that $u, v \in \mathbb{R}^n$, the Euclidean distance $d(u,v)$ between u and v is defined as:

$$d(u,v) = \sqrt{(u-v)^T (u-v)}$$

The distance can be interpreted as the length of a line that starts at u and ends at v or vice-versa, and thus it is homogeneously distributed around a single point, as exemplified in the subsection 3.6.

Because of how the Euclidean distance is used to measure spatial distances in the real world, it is often used in contexts where objects can be represented as points in an n dimensional space, such as pixels. However, due to how it is spatially distributed, the Euclidean distance can render subpar results when trying to identify an object that doesn't have homogeneously distributed features. [5]

3.3 Mahalanobis Distance

Let A and B be two statistical populations with the same covariance matrix C and means μ_A and μ_B, respectively, the Mahalanobis distance $d_M(A,B)$ between A and B is defined as:

$$d_M(A,B) = \sqrt{(\mu_A - \mu_B)^T C^{-1} (\mu_A - \mu_B)}$$

This definition can be reformulated to an equivalent one similar to that of the Euclidean distance, taking column vectors u and v and a matrix C^{-1} as arguments:

$$d_M(u,v,C^{-1}) = \sqrt{(u-v)^T C^{-1} (u-v)}$$

This second formulation can be used more straightforwardly when calculating distances between singular objects, provided that there is a inversible matrix C that determines the distribution of the distance around a single point. This is not always the case, however.

In contrast to the Euclidean distance, the distribution of the Mahalanobis distance around a reference point changes according to the covariance matrix used. In a 2D plane, for example, the Euclidean distance takes the shape of a sphere, while the Mahalanobis distance takes the shape of an ellipse, whose eccentricity and orientation are determined by the covariance matrix, and roughly corresponds to the distribution of points represented by it. Subsection 3.6 exemplifies this notion further. [6] [7]

In this work, to differentiate the Mahalanobis distance from the polynomial Mahalanobis distance, the former will sometimes be called the linear Mahalanobis distance.

3.4 Polynomial Mahalanobis Distance

The n order polynomial Mahalanobis distance $d^n_{PM}(A,B)$ between the statistical populations A and B is defined as the Mahalanobis distance between the statistical populations A' and B', where A' and B' consist of all the points in A and B, respectively, mapped into all polynomial terms of order n or less. For instance, a vector (x,y,z) would be mapped into $(x,y,z,x^2,y^2,z^2,xy,xz,yz)$ in the second order polynomial Mahalanobis distance.

The distribution of the polynomial Mahalanobis distance around a reference point varies according to its polynomial order, but, using a collection of points to build the covariance matrix, even with an order as low as 2 the distribution can be shaped in irregular ways (as exemplified by subsection 3.6), which is greatly useful when segmenting images with highly irregular objects. The downside to obtaining this result is that the distance can take a prohibitively large time to be calculated when directly using the linear Mahalanobis distance with mapped points. [8]

3.5 Bhattacharyya Distance

Let p and q be discrete probability distributions of the same size, the Bhattacharyya distance $d_B(p,q)$ between p and q is defined as: [9]

$$d_B(p,q) = -ln(BC(p,q)$$

where $BC(p,q)$ is called the Bhattacharyya coefficient between p and q, as is defined as: [10]

$$BC(p,q) = \sum_{i=1}^{k} \sqrt{p_i q_i}$$

Although it is not a true metric, as it does not satisfy the triangle inequality property, the Bhattacharyya distance is still useful for measuring similarities. It can also be calculated between continuous probability distributions, but this is not relevant for this work, as images are always composed of discrete elements.

It is related to the Hellinger distance (which is a true metric) $d_H(p,q)$ between p and q, as both can be calculated from the Bhattacharyya coefficient: [11]

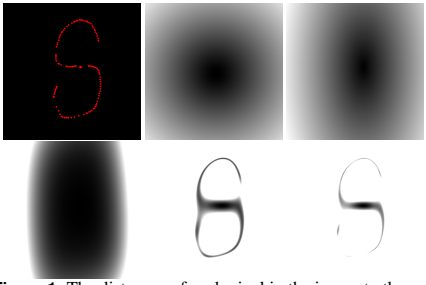

Figure 1. The distances of each pixel in the image to the reference point in the center of the 'S' shaped collection of points in the first image. The second image represents the Euclidean distance. The third image represents the Mahalanobis distance. The images in the second row represent the polynomial Mahalanobis distances of orders 2, 4 and 6, respectively.

$$d_H(p,q) = \sqrt{1 - BC(p,q)}$$

3.6 Examples of Distribution

With the exception of the Bhattacharyya distance, which in this work is used for measuring similarity between groups of pixels instead of individual ones, both the Mahalanobis distance and the polynomial Mahalanobis distance can be compared between themselves and the Euclidean distance. They have different behaviors when comparing the distances between multiple points and a single point of reference, as illustrated by the pictures of figure 1, so they have different usages depending on the application.

Each pixel in the image represent the spatial distance between itself and a reference point, indicated by the central red point in first image. The smaller red points in the first image represent the statistical population used to build the Mahalanobis distance metrics (linear and polynomial) showed in figure 1.

The Mahalanobis distance follows the shape of the dispersion of the population, growing stricter with the polynomial order. The scaling of the shades of grey were altered for better visualization, so the steep growth of the values of the borders of the images representing the polynomial Mahalanobis distances are not well represented, but the general behavior of the distances can still be visualized.

4. Implementation

The library presented by this work, named libmahala, was developed entirely in C++, and is heavily integrated with the OpenCV library, for its already existing support for CV applications in both C++ and Python. It has four classes, one for each distance plus one for a graphical interface which

serves to help the user of the application to collect pixels as points for the distances. The library is hosted at ⟨https://codigos.ufsc.br/lapix/libmahala⟩.

The Python version of the library was made using simple conventions of the pybind11 library, with template functions being separated by the possible types that can be used by the OpenCV library. The conversion between the `cv::Mat` type used by the C++ version of OpenCV and the `numpy.array` type used by the Python version was done using the open library from the repository hosted at ⟨https://github.com/edmBernard/pybind11_opencv_numpy⟩.

4.1 Mahalanobis Distance

The Mahalanobis distance class was implemented following the formulation presented by [8], which circumvents the problem of the occasional impossibility of calculating the inverse of the covariance matrix. It stores the inverse of the covariance matrix as well as the matrices and values needed for calculating it. These auxiliary values are necessary because these values can be changed during the lifetime of the object. It provides different methods for calculating the distance, each of which can be the most appropriate for a different situation.

4.2 Polynomial Mahalanobis Distance

The polynomial Mahalanobis distance class was adapted from it implementation of [13], adjusting its use to make it in line with the pattern used by the library. Its structure is almost identical to the Mahalanobis distance class, but slightly simpler, due to the immutability of some of its attributes.

4.3 Bhattacharyya Distance

The Bhattacharyya distance class was built upon the Hellinger distance implementation of OpenCV, using the latter to calculate the former. The class itself holds parameters for converting a collection of points or images into a histogram using another of OpenCV's functionalities, which is then used by the function OpenCV uses to calculate the Hellinger distance.

5. Experiments and Results

As a demonstration of usage of this library, two experiments were conducted using the library's functionalities. The first one was an application of simple thresholding on a few images provided by the Laboratory of Image Processing and Computer Graphics (LAPIX) from the Federal University of Santa Catarina (UFSC), first applying a distance function on each pixel and a reference point. The second one sought to identify desired segments in images of cellular nuclei, provided by [14]. The experiments are detailed further in their respective subsections.

5.1 Mahalanobis and Polynomial Mahalanobis Distance

This experiment used 9 images of common objects with different levels of light on different parts of the objects, using the graphical interface of the library to collect reference points for instantiating the metrics, as well as a center point for calculating the distances. For each image, the distance between each pixel and the chosen reference pixel was calculated, and then thresholded, using the same threshold value for the entire image and for all images for each distance used. The distances used were the Euclidean distance, linear Mahalanobis distance, second order polynomial Mahalanobis distance, fourth order polynomial Mahalanobis distance, and sixth order polynomial Mahalanobis distance. Figure 2 shows the original image, the chosen points for building the metrics, and the thresholded image for each distance. The images show the original image, the collected pixels highlighted in white, and the thresholded image using the Euclidean distance, the Mahalanobis distance, the second order polynomial Mahalanobis distance, the fourth order polynomial Mahalanobis distance, and the sixth order polynomial Mahalanobis distance, respectively. The threshold values used for each distance were 150, 3, 0.25, 1, and 1, respectively.

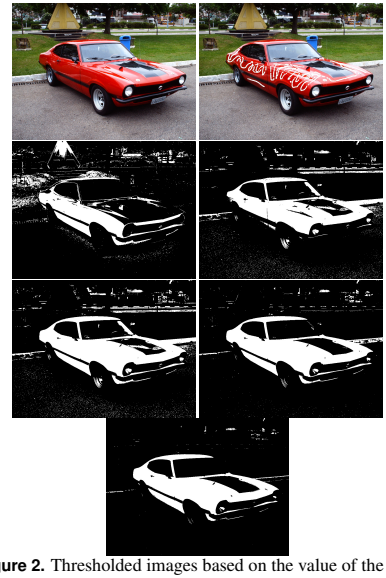

Figure 2. Thresholded images based on the value of the distance between each pixel of the first image and a reference pixel, marked in the second image on the right side of the car.

5.2 Bhattacharyya Distance

This experiment used 11 images of cellular nuclei and partitioned them into 30 segments each using the Mumford-Shah algorithm. One image was chosen as a reference, preselecting which segments contained a cellular nucleus. Then, each of the segments of the other images had the Bhattacharyya distance between them and each of the segments of the reference

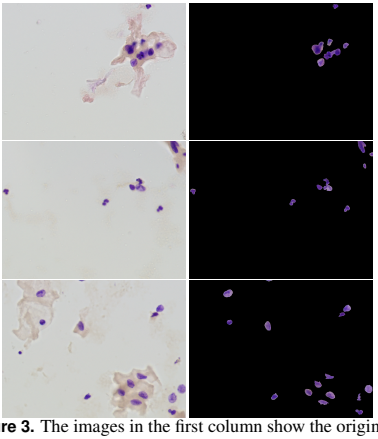

Figure 3. The images in the first column show the original images, and the ones in the second columns show the selected segments. The images in the first row were used as reference, and the segments were chosen manually.

image calculated. If the closest segment was that of a nucleus, it was marked as nucleus. The results are shown in figure 3, which compares the original images with the ones where only the selected segments are shown.

6. Conclusions

This work presents the usefulness of certain mathematical functions in CV applications, evaluates the lack of tools for helping developers and scientists when it comes to the integration of these functions into their works, and presents a library that can fill the void found by the systematic literature review.

The experiments performed showed that the tools presented by this work can be powerful even in simple applications, and future works using the library's functionalities in more complex algorithms are expected. The library is also expected to be further worked upon, as it is open-sourced and will be maintained by the LAPIX.

References

1 George Stockman and Linda G. Shapiro. *Computer Vision*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 2001.

2 Pei-Gee Peter Ho. *Image Segmentation*. INTECH Open Access Publisher, Janeza Trdine 9, 51000 Rijeka, Croatia, 2011.

3 Michel Deza and Elena Deza. *Encyclopedia of Distances* Berlin Heidelberg: Springer-Verlag, 2009.

4 John Robilliard Giles. *Introduction to the Analysis of Metric Spaces*. Cambridge University Press, The Pitt Building, Trumpington Street, Cambridge CB2 1RP, 1987.

5 Howard Anton. *Elementary Linear Algebra*. John Wiley & Sons, New York, 1987.

6 Prasanta Chandra Mahalanobis. *On the generalised distance in statistics*. Proceedings of the National Institute of Sciences of India. 2 (1): 49–55. Retrieved 2016-09-27.

7 Geoffrey John McLachlan. *Mahalanobis Distance*. Resonance. 1999. 4. 20-26. 10.1007/BF02834632.

8 Greg Grudic and Jane Mulligan. *Outdoor Path Labeling Using Polynomial Mahalanobis Distance*. Robotics: Science and Systems 2006, Philadelphia, PA, USA, August 16-19, 2006.

9 Bidyut Kr. Patraa, Raimo Launonena, Ville Ollikainena and Sukumar Nandi. *A new similarity measure using Bhattacharyya coefficient for collaborative filtering in sparse data*. KNOSYS 3090. 2015. 82. 163-177. doi:10.1016/j.knosys.2015.03.001.

10 Ondřej Straka and Miroslav Šimandl. *Using the Bhattacharyya Distance in Functional Sampling Density of Particle Filter*. Department of Cybernetics and Research Centre: Data - Algorithms - Decision, University of West Bohemia in Pilsen. 2005.

11 Roma Goussakov. *Hellinger Distance-basedSimilarity Measures forRecommender Systems*. Umeå University, Faculty of Social Sciences, Umeå School of Business and Economics (USBE), Statistics. 2020.

12 Barbara Kitchenham. *Procedures for Performing Systematic Reviews*. Keele, UK, Keele Univ.. 33. 2004.

13 L. E. Carvalho, S. L. Mantelli Neto, A. C. Sobieranski, E. Comunello and A. von Wangenheim. *Improving Graph-Based Image Segmentation Using Nonlinear Color Similarity Metrics*. International Journal of Image and Graphics Vol. 15, No. 4 (2015) 1550018. DOI:10.1142/S0219467815500187.

14 Antonio Buschetto Macarini, L., Von Wangenheim, A., Perozzo Daltoé, F., Sherlley Casimiro Onofre, A., Botelho de Miranda Onofre, F., & Ricardo Stemmer, M. (2020). *Towards a Complete Pipeline for Segmenting Nuclei in Feulgen-Stained Images.* Anais do Computer on the Beach, 11(1), 169-175. doi:https://doi.org/10.14210/cotb.v11n1.p169-175

APÊNDICE D – CÓDIGO FONTE

```
#include "BhattacharyyaDistance.hpp"

BhattaDist::BhattaDist(vector<int> channels, vector<int> histSize, vector<float> ranges)
    : _channels(channels), _histSize(histSize), _ranges(ranges) {
        assert(channels.size() == histSize.size());
        assert(2*channels.size() == ranges.size());
}

BhattaDist::BhattaDist() {}

BhattaDist::~BhattaDist() {}

vector<int> BhattaDist::channels() {
    return _channels;
}

vector<int> BhattaDist::histSize() {
    return _histSize;
}

vector<float> BhattaDist::ranges() {
    return _ranges;
}

double BhattaDist::calcBetweenPoints(Mat& points1, Mat& points2) {
    Mat img1 = delinearizeImage<double>(points1, points1.rows, 1);
    Mat img2 = delinearizeImage<double>(points2, points2.rows, 1);
    return calcBetweenImg(img1, img2);
}

double BhattaDist::calcBetweenImg(const Mat& image1, const Mat& image2,
                                  const Mat& mask1, const Mat& mask2) {
    vector<Mat> imageVec1, imageVec2;
    imageVec1.push_back(image1);
    imageVec2.push_back(image2);
    Mat hist1, hist2;
    cv::calcHist(imageVec1, _channels, mask1, hist1, _histSize, _ranges);
    cv::calcHist(imageVec2, _channels, mask2, hist2, _histSize, _ranges);
    return calcBetweenHist(hist1, hist2);
}

double BhattaDist::calcBetweenHist(const Mat &hist1, const Mat &hist2) {
    double hellinger = compareHist(hist1, hist2, HISTCMP_BHATTACHARYYA);
    double bhatCoeff = -(hellinger * hellinger) + 1;
    return -log(bhatCoeff);
}

// *! \brief Transforma uma matriz de C canais, N linhas e M colunas em uma matriz
//             de (N*M) linhas e C colunas
//      \param image A matriz a ser transformada
//      \return A matriz linearizada
// *//
template <typename T> Mat BhattaDist::linearizeImage(Mat& image) {
    int numberOfChannels = image.channels();
```

```
    Mat linearized = Mat(numberOfChannels, image.rows * image.cols, image.type() % 8);
    vector<Mat> bgrArray;
    split(image, bgrArray);

    for (int c = 0; c < numberOfChannels; c++) {
        Mat a = bgrArray[c];
        for (int i = 0; i < image.rows; i++) {
            for (int j = 0; j < image.cols; j++) {
                int currentIndex = i*image.cols + j;
                linearized.at<T>(c, currentIndex) = (double)a.at<T>(i,j);
            }
        }
    }

    linearized = linearized.t();

    return linearized;
}

// /*! \brief Transforma uma matriz de 1 canal, (N*M) linhas e C colunas em uma matriz
//             de C canais, N linhas e M colunas
//     \param image A matriz a ser transformada
//     \param rows Número de linhas da matriz resultante
//     \param rows Número de colunas da matriz resultante
//     \return A matriz delinearizada
// */
template <typename T> Mat BhattaDist::delinearizeImage(Mat& linearized, int rows, int cols) {
    assert(linearized.rows == rows*cols);
    int numberOfChannels = linearized.cols;

    Mat result;
    vector<Mat> channels;
    linearized = linearized.t();

    Mat a = Mat(rows, cols, linearized.type());
    for (int c = 0; c < numberOfChannels; c++) {
        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < cols; j++) {
                int currentIndex = i*cols + j;
                a.at<T>(i, j) = linearized.at<T>(c, currentIndex);
            }
        }
        channels.push_back(a.clone());
    }

    linearized = linearized.t();

    merge(channels, result);

    return result;
}
```

//end_file

```
#pragma once
```

```
#include "opencv2/highgui/highgui.hpp"
#include "opencv2/imgproc/imgproc.hpp"

#include <iostream>
#include <vector>
#include <math.h>

using namespace cv;
using namespace std;

/*! \brief Classe que contém os parâmetros necessários para transformar matrizes de vizinhanças
           de pontos em histogramas para permitir
           o cálculo da distância de Bhattacharyya entre elas
    \param channels Dimensões dos pontos a serem consideradas (somente leitura)
    \param histSize Vetor de tamanho do histograma para cada dimensão (somente leitura)
    \param ranges Vetor com os limites inferiores e superiores dos valores para cada dimensão
           (somente leitura)
*/
class BhattaDist {
public:
    /*! \brief Construtor
        \param channels Dimensões dos pontos a serem consideradas
        \param histSize Vetor de tamanho do histograma para cada dimensão
        \param ranges Vetor com os limites inferiores e superiores dos valores para cada dimensão
    */
    BhattaDist(vector<int> channels, vector<int> histSize, vector<float> ranges);
    BhattaDist();
    virtual ~BhattaDist();

    // getters (need to be updated)
    vector<int> channels();
    vector<int> histSize();
    vector<float> ranges();

    /*! \brief Calcula a distância de Bhattacharyya entre dois conjuntos de pontos
        \param points1 Uma matriz que contém conjunto de pontos, onde cada linha é um ponto e
                       cada coluna é uma dimensão
        \param points2 Uma matriz que contém conjunto de pontos, onde cada linha é um ponto e
                       cada coluna é uma dimensão
        \return O valor da distância de Bhattacharyya entre os dois conjuntos de pontos
    */
    double calcBetweenPoints(Mat& points1, Mat& points2);
    /*! \brief Calcula a distância de Bhattacharyya entre duas imagens
        \param image1 Uma imagem
        \param image2 Uma imagem
        \param mask1 Máscara a ser aplicada na primeira imagem a fim de filtrar quais pontos
                     serão escolhidos para construir o histograma
        \param mask2 Máscara a ser aplicada na segunda imagem a fim de filtrar quais pontos
                     serão escolhidos para construir o histograma
        \return O valor da distância de Bhattacharyya entre as duas imagens
    */
    double calcBetweenImg(const Mat& image1, const Mat& image2,
                          const Mat& mask1 = Mat(), const Mat& mask2 = Mat());

    /*! \brief Calcula a distância de Bhattacharyya entre dois histogramas
        \param hist1 Um histograma
        \param hist2 Um histograma
```

```
        \return O valor da distância de Bhattacharyya entre os dois histogramas
    */
    static double calcBetweenHist(const Mat &hist1, const Mat &hist2);

private:
    vector<int> _channels;
    vector<int> _histSize;
    vector<float> _ranges;

    template <typename T> Mat linearizeImage(Mat& image);
    template <typename T> Mat delinearizeImage(Mat& linearized, int rows, int cols);
};
```

//end_file

```
#include <Python.h>
#include "pybind11/pybind11.h"
#include "pybind11/stl.h"
#include "MahalanobisDistance.hpp"
#include "PolynomialMahalanobisDistance.hpp"
#include "BhattacharyyaDistance.hpp"
#include "PointCollector.hpp"

#include "ndarray_converter.h"

namespace py = pybind11;

PYBIND11_MODULE(libmahalapy, m) {
    NDArrayConverter::init_numpy();

    cv::Mat emptyMat = cv::Mat();

    py::class_<MahalaDist>(m, "MahalaDist")
        .def(py::init<const cv::Mat&, double, cv::Mat>(),
            py::arg("input"), py::arg("smin") = 4e-6, py::arg("reference") = emptyMat)
        .def_property_readonly("reference", &MahalaDist::reference)
        .def_property_readonly("dimension", &MahalaDist::dimension)
        .def_property_readonly("dirty", &MahalaDist::dirty)
        .def_property_readonly("u", &MahalaDist::u)
        .def_property_readonly("w", &MahalaDist::w)
        .def_property_readonly("c", &MahalaDist::c)
        .def_property_readonly("sigma2", &MahalaDist::sigma2)
        .def_property("smin",
            static_cast<double (MahalaDist::*)()> (&MahalaDist::smin),
            static_cast<void (MahalaDist::*)(double)> (&MahalaDist::smin))
        .def("build", &MahalaDist::build)
        .def("pointTo", &MahalaDist::pointTo,
            py::arg("point1"), py::arg("point2"))
        .def("pointToReference", &MahalaDist::pointToReference,
            py::arg("point"))
        .def("pointsTo", &MahalaDist::pointsTo,
            py::arg("points"), py::arg("ref"))
        .def("pointsToReference", &MahalaDist::pointsToReference,
            py::arg("points"))
        .def("imageTo_uchar", &MahalaDist::imageTo<uchar>,
            py::arg("image"), py::arg("refVector"))
        .def("imageTo_shcar", &MahalaDist::imageTo<schar>,
```

```
            py::arg("image"), py::arg("refVector"))
        .def("imageTo_ushort", &MahalaDist::imageTo<ushort>,
            py::arg("image"), py::arg("refVector"))
        .def("imageTo_short", &MahalaDist::imageTo<short>,
            py::arg("image"), py::arg("refVector"))
        .def("imageTo_int", &MahalaDist::imageTo<int>,
            py::arg("image"), py::arg("refVector"))
        .def("imageTo_float", &MahalaDist::imageTo<float>,
            py::arg("image"), py::arg("refVector"))
        .def("imageTo_double", &MahalaDist::imageTo<double>,
            py::arg("image"), py::arg("refVector"))
        .def("imageToReference_uchar", &MahalaDist::imageToReference<uchar>,
            py::arg("image"))
        .def("imageToReference_schar", &MahalaDist::imageToReference<schar>,
            py::arg("image"))
        .def("imageToReference_ushort", &MahalaDist::imageToReference<ushort>,
            py::arg("image"))
        .def("imageToReference_short", &MahalaDist::imageToReference<short>,
            py::arg("image"))
        .def("imageToReference_int", &MahalaDist::imageToReference<int>,
            py::arg("image"))
        .def("imageToReference_float", &MahalaDist::imageToReference<float>,
            py::arg("image"))
        .def("imageToReference_double", &MahalaDist::imageToReference<double>,
            py::arg("image"))
        ;

    py::class_<PointCollector>(m, "PointCollector")
        .def(py::init<cv::Mat&>(),
            py::arg("input"))
        .def(py::init<const char*, int>(),
            py::arg("path"), py::arg("flags"))
        .def_property_readonly("collectedPixels", &PointCollector::collectedPixels)
        .def_property_readonly("collectedCoordinates", &PointCollector::collectedCoordinates)
        .def_property_readonly("referencePixel", &PointCollector::referencePixel)
        .def_property_readonly("referenceCoordinate", &PointCollector::referenceCoordinate)
        ;

    py::class_<PolyMahalaDist>(m, "PolyMahalaDist")
        .def(py::init<const cv::Mat&, int, double, cv::Mat>(),
            py::arg("input"), py::arg("order"), py::arg("sig_max") = 4e-6,
            py::arg("reference") = emptyMat)
        .def_property_readonly("reference", &PolyMahalaDist::reference)
        .def_property_readonly("dimension", &PolyMahalaDist::dimension)
        .def_property_readonly("eps_svd", &PolyMahalaDist::eps_svd)
        .def_property_readonly("order", &PolyMahalaDist::order)
        .def("pointTo", &PolyMahalaDist::pointTo,
            py::arg("im_data"), py::arg("refVector"))
        .def("pointToReference", &PolyMahalaDist::pointToReference,
            py::arg("im_data"))
        .def("pointsTo", &PolyMahalaDist::pointsTo,
            py::arg("im_data"), py::arg("refVector"))
        .def("pointsToReference", &PolyMahalaDist::pointsToReference,
            py::arg("im_data"))
        .def("imageTo_uchar", &PolyMahalaDist::imageTo<uchar>,
            py::arg("image"), py::arg("refVector"))
        .def("imageTo_shcar", &PolyMahalaDist::imageTo<schar>,
```

```
            py::arg("image"), py::arg("refVector"))
        .def("imageTo_ushort", &PolyMahalaDist::imageTo<ushort>,
            py::arg("image"), py::arg("refVector"))
        .def("imageTo_short", &PolyMahalaDist::imageTo<short>,
            py::arg("image"), py::arg("refVector"))
        .def("imageTo_int", &PolyMahalaDist::imageTo<int>,
            py::arg("image"), py::arg("refVector"))
        .def("imageTo_float", &PolyMahalaDist::imageTo<float>,
            py::arg("image"), py::arg("refVector"))
        .def("imageTo_double", &PolyMahalaDist::imageTo<double>,
            py::arg("image"), py::arg("refVector"))
        .def("imageToReference_uchar", &PolyMahalaDist::imageToReference<uchar>,
            py::arg("image"))
        .def("imageToReference_schar", &PolyMahalaDist::imageToReference<schar>,
            py::arg("image"))
        .def("imageToReference_ushort", &PolyMahalaDist::imageToReference<ushort>,
            py::arg("image"))
        .def("imageToReference_short", &PolyMahalaDist::imageToReference<short>,
            py::arg("image"))
        .def("imageToReference_int", &PolyMahalaDist::imageToReference<int>,
            py::arg("image"))
        .def("imageToReference_float", &PolyMahalaDist::imageToReference<float>,
            py::arg("image"))
        .def("imageToReference_double", &PolyMahalaDist::imageToReference<double>,
            py::arg("image"))
        ;

    py::class_<BhattaDist>(m, "BhattaDist")
        .def(py::init<std::vector<int>, std::vector<int>, std::vector<float>>(),
            py::arg("channels"), py::arg("histSize"), py::arg("ranges"))
        .def_property_readonly("channels", &BhattaDist::channels)
        .def_property_readonly("histSize", &BhattaDist::histSize)
        .def_property_readonly("ranges", &BhattaDist::ranges)
        .def_static("calcBetweenHist", &BhattaDist::calcBetweenHist,
            py::arg("hist1"), py::arg("hist2"))
        .def("calcBetweenImg", &BhattaDist::calcBetweenImg,
            py::arg("image1"), py::arg("image2"),
            py::arg("mask1") = emptyMat, py::arg("mask2") = emptyMat)
        .def("calcBetweenPoints", &BhattaDist::calcBetweenPoints,
            py::arg("points1"), py::arg("points2"))
        ;
}
```

//end_file

```
#include "MahalanobisDistance.hpp"

MahalaDist::MahalaDist(const Mat& input, double smin, Mat reference)
    : _smin(smin), _reference(reference)
{
    assert(input.data);
    assert(input.rows > 1);
    assert(input.type() == CV_64FC1);

    _u = 0;
    _w = 0;
    _sigma2 = 0;
```

```
    _dimension = input.cols;
    _numberOfPoints = input.rows;


    if(!_reference.data){
        _reference = Mat(_dimension, 1, CV_64FC1);

        for (int i = 0; i < _dimension; i++) {
            _reference.at<double>(i) = (mean(input.col(i)))[0];
        }
    }

    Mat a = Mat(input.size(), input.type());

    Mat refT = _reference.t();
    for (int i = 0; i < _numberOfPoints; i++) {
        a.row(i) = input.row(i) - refT;
    }

    /*
     * The following if section was supposed to make calculations more
     * efficient by reducing the size of the _c matrix, but it doesn't
     * work with the used formula
     */

    // if (_dimension < _numberOfPoints) {
        Mat aTa = (a.t() * a);
    // } else {
    //     _c = (_a * _a.t());
    // }
    Mat discard;
    SVD::compute(aTa, _w, discard, _u);
    // discard.release();

    // _c is not directly used but it's still in the code for completeness' sake
    _c = aTa / (_numberOfPoints - 1);

    // _cInv = _c.inv();

    _dirty = 1;
    assert(input.data);
    assert(input.type() == CV_64FC1);
}

MahalaDist::MahalaDist() {}

MahalaDist::~MahalaDist() {}

/*-------------------------------------*/

Mat MahalaDist::reference() {
    return _reference;
}

double MahalaDist::smin() {
    return _smin;
```

```
}

int MahalaDist::dimension() {
    return _dimension;
}

bool MahalaDist::dirty() {
    return _dirty;
}

/*-------------------------------------*/

const Mat MahalaDist::u() const {
    return _u;
}

Mat MahalaDist::w() {
    return _w;
}

Mat MahalaDist::c() {
    return _c;
}

Mat MahalaDist::cSigma2Inv() {
    return _cSigma2Inv;
}

double MahalaDist::sigma2() {
    assert(!_dirty);

    return _sigma2;
}

/*-------------------------------------*/

void MahalaDist::smin(double smin) {
    _smin = smin;
    _dirty = 1;
}

void MahalaDist::build() {
    if (!_dirty) return;

    double w0 = _w.at<double>(0);
    _sigma2 = _smin * w0;


    /*
     * Old way of doing Mat wSigma2 = _w + _sigma2;
     */
    // for(int k = 0; k < _w.rows; k++) {
    //      if (_w.at<double>(k) >= _sigma2) {
    //          // _w.at<double>(k) += _sigma2;
    //          _k++;
    //      }
    //      else{
```

```
    //          // _w.at<double>(k) = 0;
    //      }
    // }

    /*
     * The following if sections were supposed to make calculations more
     * efficient by reducing the size of the _uK matrix, but they don't
     * work with the used formula
     */
    // if (_dimension < _numberOfPoints) {
        // _uK = Mat(_u, Rect(0,0, _k, _u.rows)).clone();
    // } else {
    //      Mat b = (_u.t() * _a).t();
    //      for (int k = 0; k < _k; k++) {
    //          b.col(k) /= cv::norm(b.col(k));
    //      }
    //      _uK = Mat(b, Rect(0,0, _k, b.rows)).clone();
    //      // _u = b.t();
    // }

    Mat wSigma2 = Mat::diag(_w + _sigma2);

    _cSigma2Inv = (_u.t() * wSigma2.inv() * _u) * (_numberOfPoints-1);

    // if (_dimension < _numberOfPoints) {
        // assert(_k <= _dimension);
        // assert(_uK.cols == _k);
        // assert(_w.rows == _dimension && _w.cols == 1);
    // } else {
        // assert(_k <= _numberOfPoints);
        // assert(_uK.cols == _k);
        // assert(_w.rows == _numberOfPoints && _w.cols == 1);
    // }
    _dirty = 0;
}

/*-------------------------------------*/

double MahalaDist::pointTo(Mat& point1, Mat& point2) {
    assert(!_dirty);
    assert(point1.rows == _dimension && point2.rows == _dimension);
    assert(point1.cols == 1 && point2.cols == 1);
    assert(point1.type() == CV_64FC1 && point2.type() == CV_64FC1);
    Mat point1T = point1.t();
    return pointsTo(point1T, point2).at<double>(0);
}

double MahalaDist::pointToReference(Mat& point) {
    assert(point.rows == _dimension);
    assert(point.cols == 1);
    assert(point.type() == CV_64FC1);
    Mat pointT = point.t();
    return pointsTo(pointT, _reference).at<double>(0);
}

Mat MahalaDist::pointsTo(Mat& points, Mat& ref) {
    assert(!_dirty);
```

```
    assert(points.cols == ref.rows);
    assert(points.cols == _dimension);
    assert(points.type() == CV_64FC1);
    assert(ref.type() == CV_64FC1);

    Mat result = Mat::zeros(points.rows, 1, CV_64FC1);

    Mat diff = Mat(points.size(), CV_64FC1);
    Mat refT = ref.t();
    // Mat rowDiffTemp; //matriz temporaria pra guardas a diferença de cada linha.
    for(int i = 0; i < points.rows; i++){
        // rowDiffTemp = (points.row(i)-refT);
        // rowDiffTemp.copyTo(diff.row(i));
        diff.row(i) = (points.row(i)-refT);
    }

    for (int i = 0; i < points.rows; i++) {
        Mat diffrow = diff.row(i);
        result.row(i) = (diffrow * _cSigma2Inv * diffrow.t());
        // result.row(i) = (diffrow * _cInv * diffrow.t());
    }
    pow(result, 0.5, result);

    return result;
}

Mat MahalaDist::pointsToReference(Mat& points) {
    assert(points.cols == _dimension);
    assert(points.type() == CV_64FC1);
    return pointsTo(points, _reference);
}

template <typename T> Mat MahalaDist::imageTo(Mat& image, Mat& ref) {
    assert(!_dirty);

    Mat linearized = linearizeImage<T>(image);
    linearized.convertTo(linearized, CV_64FC1);

    Mat distMat = pointsTo(linearized, ref);

    Mat result = delinearizeImage<double>(distMat, image.rows, image.cols);

    return result;
}

template <typename T> Mat MahalaDist::imageToReference(Mat& image) {
    return imageTo<T>(image, _reference);
}

// *! \brief Transforma uma matriz de C canais, N linhas e M colunas em uma matriz
//             de (N*M) linhas e C colunas
//     \param image A matriz a ser transformada
//     \return A matriz linearizada
// *//
template <typename T> Mat MahalaDist::linearizeImage(Mat& image) {
    int numberOfChannels = image.channels();
```

```
    Mat linearized = Mat(numberOfChannels, image.rows * image.cols, image.type() % 8);
    vector<Mat> bgrArray;
    split(image, bgrArray);

    for (int c = 0; c < numberOfChannels; c++) {
        Mat a = bgrArray[c];
        for (int i = 0; i < image.rows; i++) {
            for (int j = 0; j < image.cols; j++) {
                int currentIndex = i*image.cols + j;
                linearized.at<T>(c, currentIndex) = (double)a.at<T>(i,j);
            }
        }
    }

    linearized = linearized.t();

    return linearized;
}

// /*! \brief Transforma uma matriz de 1 canal, (N*M) linhas e C colunas em uma matriz
//             de C canais, N linhas e M colunas
//     \param image A matriz a ser transformada
//     \param rows Número de linhas da matriz resultante
//     \param rows Número de colunas da matriz resultante
//     \return A matriz delinearizada
// */
template <typename T> Mat MahalaDist::delinearizeImage(Mat& linearized, int rows, int cols) {
    assert(linearized.rows == rows*cols);
    int numberOfChannels = linearized.cols;

    Mat result;
    vector<Mat> channels;
    linearized = linearized.t();

    Mat a = Mat(rows, cols, linearized.type());
    for (int c = 0; c < numberOfChannels; c++) {
        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < cols; j++) {
                int currentIndex = i*cols + j;
                a.at<T>(i, j) = linearized.at<T>(c, currentIndex);
            }
        }
        channels.push_back(a.clone());
    }

    linearized = linearized.t();

    merge(channels, result);

    return result;
}

template Mat MahalaDist::imageTo<uchar>(Mat& image, Mat& ref);
template Mat MahalaDist::imageTo<schar>(Mat& image, Mat& ref);
template Mat MahalaDist::imageTo<ushort>(Mat& image, Mat& ref);
template Mat MahalaDist::imageTo<short>(Mat& image, Mat& ref);
template Mat MahalaDist::imageTo<int>(Mat& image, Mat& ref);
```

```
template Mat MahalaDist::imageTo<float>(Mat& image, Mat& ref);
template Mat MahalaDist::imageTo<double>(Mat& image, Mat& ref);
template Mat MahalaDist::imageToReference<uchar>(Mat& image);
template Mat MahalaDist::imageToReference<schar>(Mat& image);
template Mat MahalaDist::imageToReference<ushort>(Mat& image);
template Mat MahalaDist::imageToReference<short>(Mat& image);
template Mat MahalaDist::imageToReference<int>(Mat& image);
template Mat MahalaDist::imageToReference<float>(Mat& image);
template Mat MahalaDist::imageToReference<double>(Mat& image);
```

//end_file

```
#pragma once

#include "opencv2/highgui/highgui.hpp"
#include "opencv2/imgproc/imgproc.hpp"

#include <iostream>
#include <vector>
#include <math.h>

// #include <libmahala>

using namespace cv;
using namespace std;

/*! \brief Classe que contém os parâmetros necessários para calcular a distância de
           Mahalanobis entre pontos e realiza o cálculo
    \param c A matriz de covariância da vizinhança de pontos (somente leitura)
    \param u Matriz lateral resultante de uma operação de SVD em C * (N-1), em que
             C é a matriz de covariância associada à classe e N é a cardinalidade
             da vizinhança de pontos (somente leitura)
    \param w Matriz central resultante de uma operação de SVD em C * (N-1), em que
             C é a matriz de covariância associada à classe e N é a cardinalidade
             da vizinhança de pontos (somente leitura)
    \param reference Centro da métrica (média da vizinhança de pontos, por padrão) (somente leitura)
    \param cSigma2Inv Matriz usada para realizar o cálculo da distância no lugar da
                      inversa da matriz de covariância da vizinhança de pontos C, calculada
                      a partir de C e smin (somente leitura)
    \param smin Parâmetro do usuário, usado para garantir a inversão da matriz de
                covariância da vizinhança de pontos C, alterando o resultado
                proporcionalmente ao seu valor (leitura e escrita)
    \param sigma2 Valor derivado de smin, usado diretamente
                  para o cálculo de cSigma2Inv (somente leitura)
    \param dimension Número de dimensões dos pontos usados para construir a métrica (somente leitura)
    \param dirty Indica se o método build() precisa ser chamado antes de o objeto poder
                 realizar o cálculo da distância (somente leitura)
*/
class MahalaDist {
public:
    /*! \brief Construtor
        \param input Matriz da vizinhança de pontos, em que cada linha é um ponto
                     e cada coluna uma dimensão
        \param smin Parâmetro usado para garantir a inversão da matriz de
                    covariância da vizinhança de pontos C, alterando o resultado
                    proporcionalmente ao seu valor
        \param reference Centro da métrica (média da vizinhança de pontos, por padrão)
```

```
                    na forma de uma matriz coluna
*/
MahalaDist(const Mat& input, double smin = 4e-6, Mat reference = Mat());
MahalaDist();
virtual ~MahalaDist();

// getters
Mat reference();
double smin();
int dimension();
bool dirty();
const Mat u() const;
Mat w();
Mat c();
Mat cSigma2Inv();
double sigma2();

// setter
void smin(double smin);

/*! \brief Calcula os parâmetros necessários para o cálculo
           da distância que são dependentes do atributo smin
*/
void build();

/*! \brief Calcula a distância de Mahalanobis entre dois pontos
    \param point1 Um ponto na forma de uma matriz coluna
    \param point2 Um ponto na forma de uma matriz coluna
    \return O valor da distância de Mahalanobis entre os pontos
*/
double pointTo(Mat& point1, Mat& point2);
/*! \brief Calcula a distância de Mahalanobis entre um ponto e o centro da métrica
    \param point Um ponto na forma de uma matriz coluna
    \return O valor da distância de Mahalanobis entre os pontos
*/
double pointToReference(Mat& point);
/*! \brief Calcula a distância de Mahalanobis entre um conjunto de pontos
           e um único ponto
    \param points Uma matriz que contém conjunto de pontos, onde cada linha
                  é um ponto e cada coluna é uma dimensão
    \param ref Um único ponto na forma de uma matriz coluna
    \return Matriz de todos os valores da distância entre o ponto correspondente
            no conjunto de ponto passada como argumento e o ponto singular de referência
*/
Mat pointsTo(Mat& points, Mat& ref);
/*! \brief Calcula a distância de Mahalanobis entre um conjunto de pontos
           e o centro da métrica
    \param points Uma matriz que contém conjunto de pontos, onde cada linha
                  é um ponto e cada coluna é uma dimensão
    \return Matriz de todos os valores da distância entre o ponto correspondente
            no conjunto de ponto passada como argumento e o centro da métrica
*/
Mat pointsToReference(Mat& points);
/*! \brief Transforma uma imagem em um conjunto de pontos e calcula a distância
           de Mahalanobis entre ela e um ponto de referência
    \tparam T O mesmo tipo que seria usado no método at<T>() da classe
              cv::Mat para acessar um elemento de image
```

```
        \param image Uma imagem
        \param ref Um ponto na forma de uma matriz coluna
        \return Uma imagem em que o valor de cada pixel é a distância de Mahalanobis entre
                o pixel correspondente na imagem passada como argumento e o ponto de referência
    */
    template <typename T> Mat imageTo(Mat& image, Mat& ref);
    /*! \brief Transforma uma imagem em um conjunto de pontos e calcula a distância
               de Mahalanobis entre ela e o centro da métrica
        \tparam T O mesmo tipo que seria usado no método at<T>() da classe
                  cv::Mat para acessar um elemento de image
        \param image Uma imagem
        \return Uma imagem em que o valor de cada pixel é a distância de Mahalanobis entre
                o pixel correspondente na imagem passada como argumento e o centro da métrica
    */
    template <typename T> Mat imageToReference(Mat& image);
private:
    Mat _c;
    Mat _u;
    Mat _w;
    Mat _reference;
    Mat _cSigma2Inv;
    double _smin;
    double _sigma2;
    int _dimension;
    int _numberOfPoints;
    bool _dirty;

    template <typename T> Mat linearizeImage(Mat& image);
    template <typename T> Mat delinearizeImage(Mat& linearized, int rows, int cols);
};
```

//end_file

```
#include "PointCollector.hpp"

PointCollector::PointCollector(Mat& input) {
    Mat paintedImage = input.clone();
    bool showCollectedPoints = true;

    Scalar color = Scalar(255, 255, 255);

    Mat mask = Mat::zeros(input.size(), CV_8UC1);

    namedWindow("image");

    bool pressedLeft = false;
    bool pressedRight = false;
    CallbackParams cp = {input, paintedImage, mask, _collectedPixels,
                         _collectedCoordinates, pressedLeft, pressedRight,
                         _referencePixel, _referenceCoordinate, color};

    setMouseCallback("image", onMouse, (void*)&cp);

    while (true) {
        imshow("image", showCollectedPoints ? paintedImage : input);

        int c = waitKeyEx(1);
```

```
// cout << (short)c << endl;
// cout << c << endl;

if ((short)c == 27) break;

switch (c) {
    case 1048691: // s key
        showCollectedPoints = !showCollectedPoints;
        break;
    case 1048690: // r key
        _collectedPixels.release();
        _collectedCoordinates.release();
        _referencePixel.release();
        _referenceCoordinate.release();
        paintedImage = input.clone();
        mask = Mat::zeros(input.size(), CV_8UC1);
        break;
    case 1048625: // 1 key
        color[0] = max(0.0, color[0]-1);
        redraw(input, paintedImage, _collectedCoordinates,
            _referenceCoordinate, color);
        break;
    case 1048626: // 2 key
        color[0] = min(255.0, color[0]+1);
        redraw(input, paintedImage, _collectedCoordinates,
            _referenceCoordinate, color);
        break;
        break;
    case 1048627: // 3 key
        color[1] = max(0.0, color[1]-1);
        redraw(input, paintedImage, _collectedCoordinates,
            _referenceCoordinate, color);
        break;
        break;
    case 1048628: // 4 key
        color[1] = min(255.0, color[1]+1);
        redraw(input, paintedImage, _collectedCoordinates,
            _referenceCoordinate, color);
        break;
        break;
    case 1048629: // 5 key
        color[2] = max(0.0, color[2]-1);
        redraw(input, paintedImage, _collectedCoordinates,
            _referenceCoordinate, color);
        break;
        break;
    case 1048630: // 6 key
        color[2] = min(255.0, color[2]+1);
        redraw(input, paintedImage, _collectedCoordinates,
            _referenceCoordinate, color);
        break;
        break;
    case 1114145: // SHIFT+1 key
        color[0] = max(0.0, color[0]-16);
        redraw(input, paintedImage, _collectedCoordinates,
            _referenceCoordinate, color);
```

```
            break;
            break;
        case 1114176: // SHIFT+2 key
            color[0] = min(255.0, color[0]+16);
            redraw(input, paintedImage, _collectedCoordinates,
                _referenceCoordinate, color);
            break;
            break;
        case 1114147: // SHIFT+3 key
            color[1] = max(0.0, color[1]-16);
            redraw(input, paintedImage, _collectedCoordinates,
                _referenceCoordinate, color);
            break;
            break;
        case 1114148: // SHIFT+4 key
            color[1] = min(255.0, color[1]+16);
            redraw(input, paintedImage, _collectedCoordinates,
                _referenceCoordinate, color);
            break;
            break;
        case 1114149: // SHIFT+5 key
            color[2] = max(0.0, color[2]-16);
            redraw(input, paintedImage, _collectedCoordinates,
                _referenceCoordinate, color);
            break;
            break;
        case 1179223: // SHIFT+6 key
            color[2] = min(255.0, color[2]+16);
            redraw(input, paintedImage, _collectedCoordinates,
                _referenceCoordinate, color);
            break;
            break;
        case 1572913: // ALT+1 key
            color[0] = max(0.0, color[0]-64);
            redraw(input, paintedImage, _collectedCoordinates,
                _referenceCoordinate, color);
            break;
            break;
        case 1572914: // ALT+2 key
            color[0] = min(255.0, color[0]+64);
            redraw(input, paintedImage, _collectedCoordinates,
                _referenceCoordinate, color);
            break;
            break;
        case 1572915: // ALT+3 key
            color[1] = max(0.0, color[1]-64);
            redraw(input, paintedImage, _collectedCoordinates,
                _referenceCoordinate, color);
            break;
            break;
        case 1572916: // ALT+4 key
            color[1] = min(255.0, color[1]+64);
            redraw(input, paintedImage, _collectedCoordinates,
                _referenceCoordinate, color);
            break;
            break;
        case 1572917: // ALT+5 key
```

```
                color[2] = max(0.0, color[2]-64);
                redraw(input, paintedImage, _collectedCoordinates,
                    _referenceCoordinate, color);
                break;
                break;
            case 1572918: // ALT+6 key
                color[2] = min(255.0, color[2]+64);
                redraw(input, paintedImage, _collectedCoordinates,
                    _referenceCoordinate, color);
                break;
                break;
        }
    }

    cv::destroyWindow("image");
}

PointCollector::PointCollector(const char* path, int flags) {
    cv::ImreadModes cvFlags = (cv::ImreadModes) flags;
    Mat inputImage = imread(path, cvFlags);
    Mat paintedImage = inputImage.clone();
    bool showCollectedPoints = true;

    Mat mask = Mat::zeros(inputImage.size(), CV_8UC1);

    Scalar color = Scalar(255, 255, 255);

    namedWindow("image");

    bool pressedLeft = false;
    bool pressedRight = false;
    CallbackParams cp = {inputImage, paintedImage, mask, _collectedPixels,
                         _collectedCoordinates, pressedLeft, pressedRight,
                         _referencePixel, _referenceCoordinate, color};

    setMouseCallback("image", onMouse, (void*)&cp);

    while (true) {
        imshow("image", showCollectedPoints ? paintedImage : inputImage);

        int c = waitKeyEx(1);

        // cout << c << endl;

        if ((short)c == 27) break;

        switch (c) {
            case 1048691: // s key
                showCollectedPoints = !showCollectedPoints;
                break;
            case 1048690: // r key
                _collectedPixels.release();
                _collectedCoordinates.release();
                _referencePixel.release();
                _referenceCoordinate.release();
                paintedImage = inputImage.clone();
                mask = Mat::zeros(inputImage.size(), CV_8UC1);
```

```
        break;
    case 1048625: // 1 key
        color[0] = max(0.0, color[0]-1);
        redraw(inputImage, paintedImage, _collectedCoordinates,
            _referenceCoordinate, color);
        break;
        break;
    case 1048626: // 2 key
        color[0] = min(255.0, color[0]+1);
        redraw(inputImage, paintedImage, _collectedCoordinates,
            _referenceCoordinate, color);
        break;
        break;
    case 1048627: // 3 key
        color[1] = max(0.0, color[1]-1);
        redraw(inputImage, paintedImage, _collectedCoordinates,
            _referenceCoordinate, color);
        break;
        break;
    case 1048628: // 4 key
        color[1] = min(255.0, color[1]+1);
        redraw(inputImage, paintedImage, _collectedCoordinates,
            _referenceCoordinate, color);
        break;
        break;
    case 1048629: // 5 key
        color[2] = max(0.0, color[2]-1);
        redraw(inputImage, paintedImage, _collectedCoordinates,
            _referenceCoordinate, color);
        break;
        break;
    case 1048630: // 6 key
        color[2] = min(255.0, color[2]+1);
        redraw(inputImage, paintedImage, _collectedCoordinates,
            _referenceCoordinate, color);
        break;
        break;
    case 1114145: // SHIFT+1 key
        color[0] = max(0.0, color[0]-16);
        redraw(inputImage, paintedImage, _collectedCoordinates,
            _referenceCoordinate, color);
        break;
        break;
    case 1114176: // SHIFT+2 key
        color[0] = min(255.0, color[0]+16);
        redraw(inputImage, paintedImage, _collectedCoordinates,
            _referenceCoordinate, color);
        break;
        break;
    case 1114147: // SHIFT+3 key
        color[1] = max(0.0, color[1]-16);
        redraw(inputImage, paintedImage, _collectedCoordinates,
            _referenceCoordinate, color);
        break;
        break;
    case 1114148: // SHIFT+4 key
        color[1] = min(255.0, color[1]+16);
```

```
                redraw(inputImage, paintedImage, _collectedCoordinates,
                    _referenceCoordinate, color);
                break;
                break;
            case 1114149: // SHIFT+5 key
                color[2] = max(0.0, color[2]-16);
                redraw(inputImage, paintedImage, _collectedCoordinates,
                    _referenceCoordinate, color);
                break;
                break;
            case 1179223: // SHIFT+6 key
                color[2] = min(255.0, color[2]+16);
                redraw(inputImage, paintedImage, _collectedCoordinates,
                    _referenceCoordinate, color);
                break;
                break;
            case 1572913: // ALT+1 key
                color[0] = max(0.0, color[0]-64);
                redraw(inputImage, paintedImage, _collectedCoordinates,
                    _referenceCoordinate, color);
                break;
                break;
            case 1572914: // ALT+2 key
                color[0] = min(255.0, color[0]+64);
                redraw(inputImage, paintedImage, _collectedCoordinates,
                    _referenceCoordinate, color);
                break;
                break;
            case 1572915: // ALT+3 key
                color[1] = max(0.0, color[1]-64);
                redraw(inputImage, paintedImage, _collectedCoordinates,
                    _referenceCoordinate, color);
                break;
                break;
            case 1572916: // ALT+4 key
                color[1] = min(255.0, color[1]+64);
                redraw(inputImage, paintedImage, _collectedCoordinates,
                    _referenceCoordinate, color);
                break;
                break;
            case 1572917: // ALT+5 key
                color[2] = max(0.0, color[2]-64);
                redraw(inputImage, paintedImage, _collectedCoordinates,
                    _referenceCoordinate, color);
                break;
                break;
            case 1572918: // ALT+6 key
                color[2] = min(255.0, color[2]+64);
                redraw(inputImage, paintedImage, _collectedCoordinates,
                    _referenceCoordinate, color);
                break;
                break;
        }
    }

    cv::destroyWindow("image");
}
```

```
PointCollector::PointCollector() {}

PointCollector::~PointCollector() {}



Mat& PointCollector::collectedPixels() {
    return _collectedPixels;
}
Mat& PointCollector::collectedCoordinates() {
    return _collectedCoordinates;
}
Mat& PointCollector::referencePixel() {
    return _referencePixel;
}
Mat& PointCollector::referenceCoordinate() {
    return _referenceCoordinate;
}



void PointCollector::redraw(const Mat& originalImage, Mat& imageToPaint, const Mat& points,
                            const Mat& reference, const Scalar& color) {
    imageToPaint = originalImage.clone();
    if (reference.data) {
        circle(imageToPaint, Point(reference.at<double>(0), reference.at<double>(1)),
                                    5, color, -1);
    }
    for (int i = 0; i < points.rows; i++) {
        Point toDraw = Point(points.at<double>(i, 0), points.at<double>(i, 1));
        circle(imageToPaint, toDraw, 3, color, -1);
    }
}

void PointCollector::onMouse(int event, int x, int y, int flags, void* param) {
    CallbackParams* mp = (CallbackParams*) param;
    bool& pressedLeft = mp->pressedLeft;
    bool& pressedRight = mp->pressedRight;


    if (event == EVENT_LBUTTONDOWN) pressedLeft = true;
    if (event == EVENT_LBUTTONUP) pressedLeft = false;
    if (event == EVENT_RBUTTONDOWN) pressedRight = true;
    if (event == EVENT_RBUTTONUP) pressedRight = false;
    if (pressedLeft || pressedRight) {
        int chans = mp->img.channels();
        x = max(min(mp->img.cols-1, x), 0);
        y = max(min(mp->img.rows-1, y), 0);

        Mat bgr = Mat(1, chans, CV_64FC1);
        Mat splitmat[chans];
        split(Mat(mp->img, Rect(x,y,1,1)).clone(), splitmat);

        switch (mp->img.type() % 8) {
            case 0:
                for (int i = 0; i < chans; i++) {
```

```
                    bgr.at<double>(0,i) = (double)splitmat[i].at<uchar>(0,0);
                }
                break;
            case 1:
                for (int i = 0; i < chans; i++) {
                    bgr.at<double>(0,i) = (double)splitmat[i].at<schar>(0,0);
                }
                break;
            case 2:
                for (int i = 0; i < chans; i++) {
                    bgr.at<double>(0,i) = (double)splitmat[i].at<ushort>(0,0);
                }
                break;
            case 3:
                for (int i = 0; i < chans; i++) {
                    bgr.at<double>(0,i) = (double)splitmat[i].at<short>(0,0);
                }
                break;
            case 4:
                for (int i = 0; i < chans; i++) {
                    bgr.at<double>(0,i) = (double)splitmat[i].at<int>(0,0);
                }
                break;
            case 5:
                for (int i = 0; i < chans; i++) {
                    bgr.at<double>(0,i) = (double)splitmat[i].at<float>(0,0);
                }
                break;
            case 6:
                for (int i = 0; i < chans; i++) {
                    bgr.at<double>(0,i) = (double)splitmat[i].at<double>(0,0);
                }
                break;
            default:
                assert(mp->img.type() % 8 != 7 || mp->img.type() >= 0);
        }

        Mat xy = Mat(1, 2, CV_64FC1);
        xy.at<double>(0) = x;
        xy.at<double>(1) = y;

        if (pressedRight) {
            mp->referencePixel = bgr.t();
            mp->referenceCoordinate = xy.t();
            redraw(mp->img, mp->paintedImg, mp->coordinates, mp->referenceCoordinate, mp->color);
        } else {
            if (mp->mask.at<uchar>(y, x) == 0) {
                mp->pixels.push_back(bgr);
                mp->coordinates.push_back(xy);
                circle(mp->paintedImg, Point(x,y), 3, Scalar(255, 255, 255), -1);
                mp->mask.at<uchar>(y, x) = (uchar)255;
            }
        }
    }
}
```

//end_file

```
#pragma once

#include "opencv2/highgui/highgui.hpp"
#include "opencv2/imgproc/imgproc.hpp"

#include <iostream>
#include <vector>
#include <math.h>

using namespace cv;
using namespace std;

/*! \brief Cria uma janela interativa com uma imagem na qual o usuário pode clicar para
        extrair os pixels (coordenadas) e seus valores (apenas uma vez por coordenada)
    \param collectedPixels Valores dos pixels extraídos em que cada linha é um pixel
                        e cada coluna é um canal (somente leitura)
    \param collectedCoordinates Coordenadas dos pixels extraídos, em que cada linha
                                é um pixel (somente leitura)
    \param referencePixel Valores do pixel central (somente leitura)
    \param referenceCoordinate Coordenadas do pixel central (somente leitura)
*/
class PointCollector {
public:
    /*! \brief Construtor que gera e mantém uma janela interativa
        \param input Imagem que será exibida para usuário extrair pontos
    */
    PointCollector(Mat& input);
    /*! \brief Construtor que gera e mantém uma janela interativa
        \param path Caminho para um arquivo contendo uma imagem
                    que será exibida para usuário extrair pontos
        \param flags flags usadas pela função de leitura de arquivo do OpenCV
    */
    PointCollector(const char* path, int flags);
    PointCollector();
    virtual ~PointCollector();

    /*
     * GETTERS
     */
    Mat& collectedPixels();
    Mat& collectedCoordinates();
    Mat& referencePixel();
    Mat& referenceCoordinate();
private:
    Mat _collectedPixels;
    Mat _collectedCoordinates;
    Mat _referencePixel;
    Mat _referenceCoordinate;

    struct CallbackParams {
        Mat& img;
        Mat& paintedImg;
        Mat& mask;
        Mat& pixels;
        Mat& coordinates;
        bool& pressedLeft;
        bool& pressedRight;
```

```
        Mat& referencePixel;
        Mat& referenceCoordinate;
        Scalar& color;
    };

    static void redraw(const Mat& originalImage, Mat& imageToPaint, const Mat& points,
                       const Mat& reference, const Scalar& color);

    static void onMouse(int event, int x, int y, int flags, void* param);
};
```

//end_file

```
#include "PolynomialMahalanobisDistance.hpp"

PolyMahalaDist::PolyMahalaDist(const Mat& input, int order, double eps_svd, Mat reference) {
    // static const int numThread = 4;//omp_get_max_threads();

    assert(order > 0);

    _max_level = 0;
    _order = order;
    _numberOfPoints = input.rows;
    _dimension = input.cols;
    _eps_svd = eps_svd;

    if (order == 1) {
        _baseMaha = MahalaDist(input, _eps_svd, reference);
        _baseMaha.build();
        return;
    }

    if (!reference.data) {
        _reference = calc_mean(input);
    } else {
        _reference = reference;
    }

    Mat a = Mat(input.rows, input.cols, CV_64FC1);

    Mat refT = _reference.t();
    for (int i = 0; i < _numberOfPoints; i++) {
        a.row(i) = input.row(i) - refT;
    }

    Mat uCont = Mat();

    vector<double> my_lambda;
    vector<int> ind_null, ind_basis;
    int indn_length, indb_length;
    double s_max, s_min;


    if (_numberOfPoints > _dimension) {
        Mat aT = a.t();
        Mat aTa = aT * a;
```

```
        Mat uContTmp, s, v;
        // SVD::compute(aTa, s, uContTmp, v);
        SVD::compute(aTa, s, v, uContTmp);

        Mat uContTmpT = uContTmp.t();
        // uContTmp.release();
        uContTmp = uContTmpT;

        vector<double> s_val;

        for (uint i = 0; i < _dimension; i++) {
            s_val.push_back(s.at<double>(i));
        }

        s_max = getMaxValue(s_val.data(), s_val.size());
        s_min = _eps_svd * s_max;

        for (uint i = 0; i < _dimension; i++) {
            s_val[i] = (s_val[i] < s_min) ? 0 : s_val[i];
        }

        ind_null = find_eq(0, s_val.data(), s_val.size());
        indn_length = ind_null.size();
        ind_basis = find_eq(1, s_val.data(), s_val.size());
        indb_length = ind_basis.size();

        if (my_lambda.size()) my_lambda.clear();
        int my_lambdaSize = indb_length;
        for (uint i = 0; i < indb_length; i++) {
            my_lambda.push_back(s_val[ind_basis[i]]);
        }

        assert(my_lambdaSize > 0);

        // if (uCont.data) uCont.release();
        uCont = Mat(_dimension, my_lambdaSize, CV_64FC1);
        for (uint i = 0; i < _dimension; i++) {
            for (uint j = 0; j < my_lambdaSize; j++) {
                uCont.at<double>(i, j) = uContTmp.at<double>(i, j);
            }
        }
    } else {
        Mat aT = a.t();
        Mat aTa = a * aT;

        Mat uTmp, s, v;
        // SVD::compute(aTa, s, uTmp, v);
        SVD::compute(aTa, s, v, uTmp);

        Mat uT = uTmp.t();

        vector<double> s_val;

        for (uint i = 0; i < _numberOfPoints; i++) {
            s_val.push_back(s.at<double>(i));
        }
```

```
    s_max = getMaxValue(s_val.data(), s_val.size());
    s_min = _eps_svd * s_max;

    for (uint i = 0; i < _numberOfPoints; i++) {
        s_val[i] = (s_val[i] < s_min) ? 0 : s_val[i];
    }

    ind_null = find_eq(0, s_val.data(), s_val.size());
    indn_length = ind_null.size();
    ind_basis = find_eq(1, s_val.data(), s_val.size());
    indb_length = ind_basis.size();

    if (my_lambda.size()) my_lambda.clear();

    if (indb_length == 0) {
        std::cout << "Problema!" << endl;
        std::cout << "S = " << endl << s << endl;
        std::cout << "A = " << endl << a << endl;
        std::cout << "UTmp = " << endl << uTmp << endl;
    }
    int my_lambdaSize = indb_length;
    for (uint i = 0; i < indb_length; i++) {
        my_lambda.push_back(s_val[ind_basis[i]]);
    }

    assert(my_lambdaSize > 0);

    Mat u = Mat(_numberOfPoints, my_lambdaSize, CV_64FC1);
    for (uint i = 0; i < _numberOfPoints; i++) {
        for (uint j = 0; j < my_lambdaSize; j++) {
            u.at<double>(i, j) = uT.at<double>(i, j);
        }
    }

    // uT.release();
    uT = u.t();
    Mat uContTmp = uT * a;
    Mat uCont = uContTmp.t();

    // vector<double> uCont_dist;

    for (uint i = 0; i < uCont.cols; i++) {
        // uCont_dist.push_back(norm(uCont.col(i)));
        uCont.col(i) /= norm(uCont.col(i));//uCont_dist[i];
    }

}

Mat proj_A = a * uCont;
proj_A = proj_A.clone();
double max_aP = getMaxAbsValue(proj_A.ptr<double>(0), proj_A.rows * proj_A.cols);

if (max_aP > _eps_svd) {
    proj_A /= max_aP;
} else {
    max_aP = 1;
}
```

```
uint n_proj = proj_A.rows;
uint d_proj = proj_A.cols;

int num_svds = 1;

lev_basis newBasis;

newBasis.A_basis = uCont;
newBasis.max_aP = max_aP;
newBasis.ind_usesize = indb_length;
newBasis.ind_use = ind_basis;
newBasis.d_proj = d_proj;
newBasis.dmssize = indb_length;
for (uint i = 0; i < indb_length; i++) {
    newBasis.dms.push_back(-my_lambda[i] / (s_min * (my_lambda[i] + s_min)));
}
newBasis.sigma_inv = 1 / s_min;

// m_model->levBegin = newBasis; // TROCAR
_basisVec.push_back(newBasis);

bool cont = false;
Mat new_dim;

if (d_proj > 1) {
    uint sizeC;
    new_dim = polynomialProjection(proj_A);
    double var = calcVarianceScalar(new_dim, -1);

    if (var > _eps_svd)
        cont = true;
}


if (_order == 1) {
    cont = false;
}

lev_basis currBasis = _basisVec[0];
int nt, dt;
//***       THE BIG WHILE !!!!!!!
while (cont) {
    n_proj = new_dim.rows;
    d_proj = new_dim.cols;

    // if (a.data) a.release();

    vector<int> ind_use;
    if (d_proj == 1) {
        a = new_dim.clone();
    } else {
        vector<int> ind_useTmp;
        a = removeNullDimensions(new_dim, ind_useTmp);

        for (uint i = 0; i < ind_useTmp[0]; i++) {
            ind_use.push_back(ind_useTmp[i+1]);
```

```
    }
}

//%%%%%%%%%%%%%%%%%%%%%%%%%%%%
//%%%%%%%%%%%%%%%%%%%%%%%%%%%%
//%%%%%%%%%%%%%%%%%%%%%%%%%%%%
//%%% Find PCA basis
//%%%%%%%%%%%%%%%%%%%%%%%%%%%%
//%%%%%%%%%%%%%%%%%%%%%%%%%%%%
//%%%%%%%%%%%%%%%%%%%%%%%%%%%%

nt = a.rows;
dt = a.cols;

// if (aTa.data) aTa.release();

if (nt >= dt) {
    Mat aT = a.t();
    Mat aTa = aT * a;

    Mat uContTmp, s, v;
    // SVD::compute(aTa, s, uContTmp, v); // A = U S V^T
    SVD::compute(aTa, s, v, uContTmp); // A = U S V^T

    Mat uContTmpT = uContTmp.t();
    // uContTmp.release();
    uContTmp = uContTmpT;

    vector<double> s_val;
    for (uint i = 0; i < dt; i++) {
        s_val.push_back(s.at<double>(i));
    }

    s_max = getMaxValue(s_val.data(), s_val.size());
    s_min = _eps_svd * s_max;

    for (uint i = 0; i < dt; i++) {
        s_val[i] = (s_val[i] < s_min) ? 0 : s_val[i];
    }

    ind_null = find_eq(0, s_val.data(), s_val.size());
    indn_length = ind_null.size();
    ind_basis = find_eq(1, s_val.data(), s_val.size());
    indb_length = ind_basis.size();

    if (my_lambda.size()) my_lambda.clear();
    int my_lambdaSize = indb_length;
    for (uint i = 0; i < indb_length; i++) {
        my_lambda.push_back(s_val[ind_basis[i]]);
    }

    assert(my_lambdaSize > 0);

    uCont = Mat(dt, my_lambdaSize, CV_64FC1);
    for (uint i = 0; i < dt; i++) {
        for (uint j = 0; j < my_lambdaSize; j++) {
            uCont.at<double>(i, j) = uContTmp.at<double>(i, j);
```

```
            }
        }
    } else {
        Mat aT = a.t();
        Mat aTa = a * aT;

        Mat uTmp, s, v;
        // SVD::compute(aTa, s, uTmp, v);
        SVD::compute(aTa, s, v, uTmp);

        Mat uT = uTmp.t();

        vector<double> s_val;
        for (uint i = 0; i < nt; i++) {
            s_val.push_back(s.at<double>(i));
        }

        s_max = getMaxValue(s_val.data(), s_val.size());
        s_min = _eps_svd * s_max;

        for (uint i = 0; i < nt; i++) {
            s_val[i] = (s_val[i] < s_min) ? 0 : s_val[i];
        }

        ind_null = find_eq(0, s_val.data(), s_val.size());
        indn_length = ind_null.size();
        ind_basis = find_eq(1, s_val.data(), s_val.size());
        indb_length = ind_basis.size();

        if (indb_length == 0)
            std::cout << "Problema!" << endl;

        if (my_lambda.size()) my_lambda.clear();
        int my_lambdaSize = indb_length;
        for (uint i = 0; i < indb_length; i++) {
            my_lambda.push_back(s_val[ind_basis[i]]);
        }

        assert(my_lambdaSize > 0);

        Mat u = Mat(nt, my_lambdaSize, CV_64FC1);
        for (uint i = 0; i < nt; i++) {
            for (uint j = 0; j < my_lambdaSize; j++) {
                u.at<double>(i, j) = uT.at<double>(i, j);
            }
        }

        // uT.release();
        uT = u.t();
        Mat uContTmp = uT * a;
        uCont = uContTmp.t();

        // vector<double> uCont_dist;
        for (uint i = 0; i < uCont.cols; i++) {
            // uCont_dist.push_back(norm(uCont.col(i)));
            uCont.col(i) /= norm(uCont.col(i));//uCont_dist[i];
        }
```

```
        }

        // if (proj_A.data) proj_A.release();
        proj_A = a * uCont;
        proj_A = proj_A.clone();
        max_aP = getMaxAbsValue(proj_A.ptr<double>(0), proj_A.rows * proj_A.cols);

        if (max_aP > _eps_svd) {
            proj_A /= max_aP;
        } else {
            max_aP = 1;
        }

        n_proj = proj_A.rows;
        d_proj = proj_A.cols;

        num_svds++;

        lev_basis nextBasis;
        nextBasis.A_basis = uCont;
        nextBasis.max_aP = max_aP;
        nextBasis.ind_usesize = dt;
        nextBasis.ind_use = ind_use;
        nextBasis.d_proj = d_proj;
        nextBasis.dmssize = indb_length;
        for (uint i = 0; i < indb_length; i++) {
            nextBasis.dms.push_back(-my_lambda[i] / (s_min * (my_lambda[i] + s_min)));
        }
        nextBasis.sigma_inv = 1 / s_min;
        _basisVec.push_back(nextBasis);
        currBasis = nextBasis;

        if (d_proj > 1) {
            new_dim = polynomialProjection(proj_A);
            double var = calcVarianceScalar(new_dim, -1);

            if (var > _eps_svd)
                cont = true;
            else
                cont = false;
        } else {
            cont = false;
        }

        if (num_svds >= _order)
            cont = false;
    }
}

PolyMahalaDist::PolyMahalaDist() {}

PolyMahalaDist::~PolyMahalaDist() {}

//--------------------------------------------------------------------------------

Mat PolyMahalaDist::reference() {
    return _reference;
```

```
}
double PolyMahalaDist::eps_svd() {
    return _eps_svd;
}
int PolyMahalaDist::dimension() {
    return _dimension;
}
int PolyMahalaDist::order() {
    return _order;
}

//-------------------------------------------------------------------------

double PolyMahalaDist::pointTo(Mat& im_data, Mat& refVector) {
    assert(im_data.cols == 1);
    assert(im_data.rows == _dimension);
    Mat im_dataT = im_data.t();
    return pointsTo(im_dataT, refVector).at<double>(0,0);
}

double PolyMahalaDist::pointToReference(Mat& im_data) {
    return pointTo(im_data, _reference);
}

Mat PolyMahalaDist::pointsTo(Mat& im_data, Mat& refVector) {
    if (_order == 1) {
        return _baseMaha.pointsTo(im_data, refVector);
    }

    int size = im_data.rows;
    int dimensions = im_data.cols;

    _max_level = 0;

    Mat refT = refVector.t();
    Mat x = Mat(size, dimensions, CV_64FC1);
    for (uint i = 0; i < size; i++) {
        x.row(i) = im_data.row(i) - refT;
    }

    int projCount = 0;
    lev_basis currBasis = _basisVec[0];

    Mat proj_A_sq = x * currBasis.A_basis;

    proj_A_sq = proj_A_sq.mul(proj_A_sq);

    Mat a_sq = x.mul(x);

    Mat a_sqTmp = a_sq * currBasis.sigma_inv;

    Mat a_sqTmpT = a_sqTmp.t();

    Mat q1 = Mat::zeros(a_sqTmpT.cols, 1, CV_64FC1);
    for (int i = 0; i < a_sqTmpT.rows; i++) {
        for (int j = 0; j < a_sqTmpT.cols; j++) {
            q1.at<double>(j) += a_sqTmpT.at<double>(i, j);
```

```
    }
}

Mat q2 = Mat::zeros(proj_A_sq.rows, 1, CV_64FC1);
for (int i = 0; i < proj_A_sq.rows; i++) {
    for (int j = 0; j < proj_A_sq.cols; j++) {
        q2.at<double>(i) += proj_A_sq.at<double>(i, j) * currBasis.dms[j];
    }
}

Mat q_in = max(q1 + q2, 0);

Mat output_m_intensValues = q_in.clone();

_max_level++;

Mat new_dim = Mat();
if (_order > 1) {
    Mat proj_A = x * currBasis.A_basis;

    proj_A /= currBasis.max_aP;

    new_dim = polynomialProjection(proj_A);
}

for (projCount = 1; projCount < _basisVec.size(); projCount++) {
    currBasis = _basisVec[projCount];

    Mat new_dim_used = removeNullIndexes(new_dim, currBasis.ind_use);

    // if (proj_A_sq.data) proj_A_sq.release();
    proj_A_sq = new_dim_used * currBasis.A_basis;

    proj_A_sq = proj_A_sq.mul(proj_A_sq);

    // if (a_sq.data) a_sq.release();
    a_sq = new_dim_used.mul(new_dim_used);

    Mat a_sqTmp = a_sq * currBasis.sigma_inv;

    Mat a_sqTmpT = a_sqTmp.t();

    Mat q1 = Mat::zeros(a_sqTmpT.cols, 1, CV_64FC1);
    for (uint i = 0; i < a_sqTmpT.rows; i++) {
        for (uint j = 0; j < a_sqTmpT.cols; j++) {
            q1.at<double>(j) += a_sqTmpT.at<double>(i, j);
        }
    }

    Mat q2 = Mat::zeros(proj_A_sq.rows, 1, CV_64FC1);
    for (uint i = 0; i < proj_A_sq.rows; i++) {
        for (uint j = 0; j < proj_A_sq.cols; j++) {
            q1.at<double>(i) += proj_A_sq.at<double>(i, j) * currBasis.dms[j];
        }
    }

    Mat q_in = max(q1 + q2, 0);
```

```
        Mat proj = new_dim_used * currBasis.A_basis;
        proj /= currBasis.max_aP;

        new_dim = polynomialProjection(proj);

        output_m_intensValues += q_in;

        _max_level++;
    }

    return output_m_intensValues;
}

Mat PolyMahalaDist::pointsToReference(Mat& im_data) {
    return pointsTo(im_data, _reference);
}

template <typename T> Mat PolyMahalaDist::imageTo(Mat& image, Mat& refVector) {
    Mat linearized = linearizeImage<T>(image);
    linearized.convertTo(linearized, CV_64FC1);

    Mat distMat = pointsTo(linearized, refVector);

    Mat result = delinearizeImage<double>(distMat, image.rows, image.cols);

    return result;
}

template <typename T> Mat PolyMahalaDist::imageToReference(Mat& image) {
    return imageTo<T>(image, _reference);
}

// *! \brief Transforma uma matriz de C canais, N linhas e M colunas em uma matriz
//            de (N*M) linhas e C colunas
//     \param image A matriz a ser transformada
//     \return A matriz linearizada
// *//
template <typename T> Mat PolyMahalaDist::linearizeImage(Mat& image) {
    int numberOfChannels = image.channels();

    Mat linearized = Mat(numberOfChannels, image.rows * image.cols, image.type() % 8);
    vector<Mat> bgrArray;
    split(image, bgrArray);

    for (int c = 0; c < numberOfChannels; c++) {
        Mat a = bgrArray[c];
        for (int i = 0; i < image.rows; i++) {
            for (int j = 0; j < image.cols; j++) {
                int currentIndex = i*image.cols + j;
                linearized.at<T>(c, currentIndex) = (double)a.at<T>(i,j);
            }
        }
    }

    linearized = linearized.t();
```

```
    return linearized;
}

// /*! \brief Transforma uma matriz de 1 canal, (N*M) linhas e C colunas em uma matriz
//             de C canais, N linhas e M colunas
//     \param image A matriz a ser transformada
//     \param rows Número de linhas da matriz resultante
//     \param rows Número de colunas da matriz resultante
//     \return A matriz delinearizada
// */
template <typename T> Mat PolyMahalaDist::delinearizeImage(Mat& linearized, int rows, int cols) {
    assert(linearized.rows == rows*cols);
    int numberOfChannels = linearized.cols;

    Mat result;
    vector<Mat> channels;
    linearized = linearized.t();

    Mat a = Mat(rows, cols, linearized.type());
    for (int c = 0; c < numberOfChannels; c++) {
        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < cols; j++) {
                int currentIndex = i*cols + j;
                a.at<T>(i, j) = linearized.at<T>(c, currentIndex);
            }
        }
        channels.push_back(a.clone());
    }

    linearized = linearized.t();

    merge(channels, result);

    return result;
}

Mat PolyMahalaDist::calc_mean(Mat data) {
    int d = data.cols;

    Mat m = Mat(d, 1, CV_64FC1);

    for (uint i = 0; i < d; i++) {
        m.at<double>(i) = (mean(data.col(i)))[0];
    }

    return m;
}

double PolyMahalaDist::getMaxValue(double *in, uint size) {
    double max = 0;
    for (uint i = 0; i < size; i++) {
        if (in[i] > max) max = in[i];
    }
    return max;
}

double PolyMahalaDist::getMaxAbsValue(double *in, uint size) {
```

```
    double max = 0;
    for (uint i = 0; i < size; i++) {
        if (abs(in[i]) > max) max = abs(in[i]);
    }
    return max;
}

Mat PolyMahalaDist::polynomialProjection(Mat vec) {
    Mat gvec; //= Mat(vec.rows, (vec.cols + 2) * (vec.cols + 1) / 2 - 1, vec.type());

    Mat vec_sq = vec.mul(vec); // square each of vec's elements
    Mat vec_cross = Mat(vec.rows, vec.cols * (vec.cols - 1) / 2, vec.type());
    // cross products between vec's elements

    int k = 0;
    // calculates the cross products:
    for (int i = 0; i < vec.cols; i++) {
        for (int j = i+1; j < vec.cols; j++) {
            vec_cross.col(k) = vec.col(i).mul(vec.col(j));
            k++;
        }
    }

    Mat mats[3] = {vec, vec_sq, vec_cross};
    hconcat(mats, 3, gvec);

    return gvec;
}

vector<int> PolyMahalaDist::find_eq(int opt, double *in, uint size) {
    vector<int> indexes;
    if (opt < 0) {
        for (uint i = 0; i < size; i++) {
            if (in[i] < 0) indexes.push_back(i);
        }
    } else if (opt > 0) {
        for (uint i = 0; i < size; i++) {
            if (in[i] > 0) indexes.push_back(i);
        }
    } else {
        for (uint i = 0; i < size; i++) {
            if (in[i] == 0) indexes.push_back(i);
        }
    }

    return indexes;
}

double PolyMahalaDist::calcVarianceScalar(Mat A, int column) {
    assert(A.type() == CV_64FC1);
    double var = 0;
    uint nt = A.rows;
    uint dt = A.cols;

    if (column == -1) {
        double sum = 0;
        for (uint i = 0; i < nt; i++) {
```

```
            for (uint j = 0; j < dt; j++) {
                sum += A.at<double>(i, j);
            }
        }
        double mean = sum / (nt * dt);

        sum = 0;
        for (uint i = 0; i < nt; i++) {
            for (uint j = 0; j < dt; j++) {
                sum += (A.at<double>(i, j) - mean) * (A.at<double>(i, j) - mean);
            }
        }
        var = 1 / ((double) (nt * dt) - 1) * sum;
    } else {
        double sum = 0;
        for (unsigned int i = 0; i < nt; i++) {
            sum += A.at<double>(i, column);
        }
        double mean = sum / nt;

        sum = 0;
        for (uint i = 0; i < nt; i++) {
            sum += (A.at<double>(i, column) - mean) * (A.at<double>(i, column) - mean);
        }
        var = 1 / ((double) nt - 1) * sum;
    }
    return var;
}

Mat PolyMahalaDist::calcVarianceVector(Mat A) {
    Mat var_dim = Mat::zeros(A.cols, 1, A.type());

    uint nt = A.rows;
    uint dt = A.cols;

    Mat mean = Mat::zeros(A.cols, 1, A.type());
    for (uint k = 0; k < dt; k++) {
        for (uint i = 0; i < nt; i++) {
            mean.at<double>(k) += A.at<double>(i, k);
        }
    }
    mean /= nt;

    for (uint k = 0; k < dt; k++) {
        for (uint i = 0; i < nt; i++) {
            var_dim.at<double>(k) += (A.at<double>(i, k) - mean.at<double>(k))
                                     * (A.at<double>(i, k) - mean.at<double>(k));
        }
    }
    var_dim /= (nt - 1);

    return var_dim;
}

Mat PolyMahalaDist::removeNullIndexes(Mat A, vector<int> ind_use) {
    Mat new_dim_usedT, aT = A.t();
```

```
    for (uint i = 0; i < ind_use.size(); i++) {
        new_dim_usedT.push_back(aT.row(ind_use[i]));
    }

    Mat new_dim_used = new_dim_usedT.t();

    return new_dim_used;
}

Mat PolyMahalaDist::removeNullDimensions(Mat A, vector<int> &ind_use) {

    uint nt = A.rows;
    uint dt = A.cols;

    Mat var_new_dim = calcVarianceVector(A);

    uint N = 0;
    double maxVar = getMaxValue(var_new_dim.clone().ptr<double>(0), dt);
    for (uint i = 0; i < dt; i++) {
        if (var_new_dim.at<double>(i) > 1e-8 * maxVar) N++;
    }
    ind_use.push_back(N);

    Mat newMat = Mat(nt, N, CV_64FC1);
    int k = 0;
    for (uint i = 0; i < dt; i++) {
        if (var_new_dim.at<double>(i) > 1e-8 * maxVar) {
            for (uint j = 0; j < nt; j++) {
                newMat.at<double>(j, k) = A.at<double>(j, i);
            }
            ind_use.push_back(i);
            k++;
        }
    }

    return newMat;
}

template Mat PolyMahalaDist::imageTo<uchar>(Mat& image, Mat& refVector);
template Mat PolyMahalaDist::imageTo<schar>(Mat& image, Mat& refVector);
template Mat PolyMahalaDist::imageTo<ushort>(Mat& image, Mat& refVector);
template Mat PolyMahalaDist::imageTo<short>(Mat& image, Mat& refVector);
template Mat PolyMahalaDist::imageTo<int>(Mat& image, Mat& refVector);
template Mat PolyMahalaDist::imageTo<float>(Mat& image, Mat& refVector);
template Mat PolyMahalaDist::imageTo<double>(Mat& image, Mat& refVector);
template Mat PolyMahalaDist::imageToReference<uchar>(Mat& image);
template Mat PolyMahalaDist::imageToReference<schar>(Mat& image);
template Mat PolyMahalaDist::imageToReference<ushort>(Mat& image);
template Mat PolyMahalaDist::imageToReference<short>(Mat& image);
template Mat PolyMahalaDist::imageToReference<int>(Mat& image);
template Mat PolyMahalaDist::imageToReference<float>(Mat& image);
template Mat PolyMahalaDist::imageToReference<double>(Mat& image);
```

//end_file

```
#pragma once
```

```
#include "opencv2/highgui/highgui.hpp"
#include "opencv2/imgproc/imgproc.hpp"
#include "MahalanobisDistance.hpp"

#include <iostream>
#include <vector>
#include <math.h>
// #include <omp.h>

using namespace cv;
using namespace std;

/*! \brief Classe que contém os parâmetros necessários para calcular a distância de
          Mahalanobis polinomial entre pontos e realiza o cálculo
    \param reference Centro da métrica (média da vizinhança de pontos, por padrão)
                     (somente leitura)
    \param eps_svd Parâmetro do usuário, usado para garantir a inversão da matriz de
                   covariância da vizinhança de pontos C, alterando o resultado
                   proporcionalmente ao seu valor (leitura e escrita)
    \param order Ordem dos termos polinomiais (somente leitura)
    \param dimension Número inicial de dimensões da vizinhança de pontos (somente leitura)
*/
class PolyMahalaDist {
public:


    /*! \brief Construtor
        \param input Matriz da vizinhança de pontos, em que cada linha é um ponto
                     e cada coluna uma dimensão
        \param order Ordem dos termos polinomiais
        \param eps_svd Parâmetro do usuário, usado para garantir a inversão da matriz de
                       covariância da vizinhança de pontos C, alterando o resultado
                       proporcionalmente ao seu valor
        \param reference Centro da métrica (média da vizinhança de pontos, por padrão)
                         na forma de uma matriz coluna
    */
    PolyMahalaDist(const Mat& input, int order, double eps_svd = 4e-6, Mat reference = Mat());
    PolyMahalaDist();
    virtual ~PolyMahalaDist();

    // getters
    Mat reference();
    double eps_svd();
    int dimension();
    int order();

    // // setters
    // void eps_svd(double eps_svd);
    // void order(int order);

    // /*! \brief builds the parameters that are used on the calculation
    //             of the distances and are based on the _smin and _l class members
    // */
    // void build();

    /*! \brief Calcula a distância de Mahalanobis polinomial entre dois pontos
        \param point1 Um ponto na forma de uma matriz coluna
```

```
        \param point2 Um ponto na forma de uma matriz coluna
        \return O valor da distância de Mahalanobis polinomial entre os pontos
    */
    double pointTo(Mat& im_data, Mat& refVector);
    /*! \brief Calcula a distância de Mahalanobis polinomial entre um ponto e o centro da métrica
        \param point Um ponto na forma de uma matriz coluna
        \return O valor da distância de Mahalanobis polinomial entre os pontos
    */
    double pointToReference(Mat& im_data);
    /*! \brief Calcula a distância de Mahalanobis polinomial entre
               um conjunto de pontos e um único ponto
        \param points Uma matriz que contém conjunto de pontos,
                      onde cada linha é um ponto e cada coluna é uma dimensão
        \param ref Um único ponto na forma de uma matriz coluna
        \return Matriz de todos os valores da distância de Mahalanobis
                polinomial entre o ponto correspondente no conjunto de
                pontos passada como argumento e o ponto singular de referência
    */
    Mat pointsTo(Mat& im_data, Mat& refVector);
    /*! \brief Calcula a distância de Mahalanobis polinomial entre
               um conjunto de pontos e o centro da métrica
        \param points Uma matriz que contém conjunto de pontos,
                      onde cada linha é um ponto e cada coluna é uma dimensão
        \return Matriz de todos os valores da distância de Mahalanobis polinomial
                entre o ponto correspondente no conjunto de pontos
                passada como argumento e o centro da métrica
    */
    Mat pointsToReference(Mat& im_data);
    /*! \brief Transforma uma imagem em um conjunto de pontos e calcula a distância
               de Mahalanobis polinomial entre ela e um ponto de referência
        \tparam T O mesmo tipo que seria usado no método at<T>() da classe
                  cv::Mat para acessar um elemento de image
        \param image Uma imagem
        \param ref Um ponto na forma de uma matriz coluna
        \return Uma imagem em que o valor de cada pixel é a distância de Mahalanobis polinomial entre
                o pixel correspondente na imagem passada como argumento e o ponto de referência
    */
    template <typename T> Mat imageTo(Mat& image, Mat& refVector);
    /*! \brief Transforma uma imagem em um conjunto de pontos e calcula a distância
               de Mahalanobis polinomial entre ela e o centro da métrica
        \tparam T O mesmo tipo que seria usado no método at<T>() da classe
                  cv::Mat para acessar um elemento de image
        \param image Uma imagem
        \return Uma imagem em que o valor de cada pixel é a distância
                de Mahalanobis polinomial entre o pixel correspondente
                na imagem passada como argumento e o centro da métrica
    */
    template <typename T> Mat imageToReference(Mat& image);
private:
    Mat _reference;
    double _eps_svd;
    int _order;
    int _max_level;
    int _dimension;
    int _numberOfPoints;
    bool _dirty;
```

```
/*
 * struct usada para armazenar os parâmetros de cada nível da expansão polinomial
 */
struct lev_basis {
    Mat A_basis;
    double max_aP;
    int ind_usesize;
    vector<int> ind_use;

    int d_proj;
    int dmssize;
    vector<double> dms;

    double sigma_inv;
};

/*
 * Distância de Mahalanobis linear,
 * realiza o cálculo quando a ordem do polinômio é 1 (linear)
 */
MahalaDist _baseMaha;

vector<lev_basis> _basisVec;

template <typename T> Mat linearizeImage(Mat& image);
template <typename T> Mat delinearizeImage(Mat& linearized, int rows, int cols);

// /*! \brief Calcula a média de uma vizinhança de pontos
//     \param data Vizinhança de pontos
//     \return Vetor de médias
// */
Mat calc_mean(Mat data);

// /*! \brief Procura o maior valor em um dataset
//     \param in Ponteiro para o início de um dataset
//     \param size Tamanho do dataset
//     \return Valor máximo encontrado
// */
double getMaxValue(double *in, uint size);

// /*! \brief Procura o maior valor absoluto em um dataset
//     \param in Ponteiro para o início de um dataset
//     \param size Tamanho do dataset
//     \return Valor absoluto máximo encontrado
// */
double getMaxAbsValue(double *in, uint size);

// /*! \brief Realiza a expansão polinomial de um conjunto de pontos em segunda ordem
//     \param vec Conjunto de pontos a serem expandidos polinomialmente
//     \return Conjunto de pontos expandidos polinomialmente
// */
Mat polynomialProjection(Mat vec);

// /*! \brief Encontra em um dataset valores iguais, maiores ou menores que 0
//     \param opt Escolha de valores (0 para iguais a 0, negativo para menores
//                que 0 e positivo para maiores que 0)
//     \param in Ponteiro para o início de um dataset
```

```
    //      \param size Tamanho do dataset
    //      \return Vetor de valores encontrados
    // */
    vector<int> find_eq(int opt, double *in, uint size);

    // /*! \brief Calcula a variância de um conjunto de pontos
    //      \param A Matriz que guarda um conjunto de pontos, em que cada linha
    //             é um ponto e cada coluna é uma dimensão
    //      \param column Dimensão escolhida para calcular a variância
    //                  (valor negativo para calcular a variância de todos os elementos da matriz)
    //      \return Variância calculada
    // */
    double calcVarianceScalar(Mat A, int column);

    // /*! \brief Calcula a variância de um conjunto de pontos
    //      \param A Matriz que guarda um conjunto de pontos, em que cada linha
    //             é um ponto e cada coluna é uma dimensão
    //      \return Vetor das variâncias de cada dimensão dos pontos do conjunto
    // */
    Mat calcVarianceVector(Mat A);

    // /*! \brief Remove colunas de uma matriz, mantendo apenas as especificadas
    //      \param A Matriz a ter colunas removidas
    //      \param ind_use Vetor de colunas a serem mantidas
    //      \return Matriz com colunas removidas
    // */
    Mat removeNullIndexes(Mat A, vector<int> ind_use);

    // /*! \brief Remove colunas de uma matriz que possuem uma variância menor que
    //             1e-8 vezes a maior variância da matriz
    //      \param A Matriz a ter colunas removidas
    //      \param ind_use Vetor que ao término da função armazenará
    //             as colunas que foram mantidas
    //      \return Matriz com colunas removidas
    // */
    Mat removeNullDimensions(Mat A, vector<int> &ind_use);
};
```

//end_file

```
#include "libmahala/MahalanobisDistance.hpp"
#include "libmahala/BhattacharyyaDistance.hpp"
#include "libmahala/PolynomialMahalanobisDistance.hpp"
#include "libmahala/PointCollector.hpp"

#include <iostream>
#include <chrono>
#include <ctime>

void test(String name, Mat& input, Mat& resultEuc, Mat& resultMaha,
          Mat& resultMaha2, Mat& resultMaha4, Mat& resultMaha6);

int main() {

    cv::Mat input_flower, input_church, input_horses, input_place, input_starfish;
    cv::Mat thEuc_flower, thMaha_flower, thMaha2_flower, thMaha4_flower, thMaha6_flower;
    cv::Mat thEuc_church, thMaha_church, thMaha2_church, thMaha4_church, thMaha6_church;
```

```
cv::Mat thEuc_horses, thMaha_horses, thMaha2_horses, thMaha4_horses, thMaha6_horses;
cv::Mat thEuc_place, thMaha_place, thMaha2_place, thMaha4_place, thMaha6_place;
cv::Mat thEuc_starfish, thMaha_starfish, thMaha2_starfish, thMaha4_starfish,
        thMaha6_starfish;
std::vector<String> names = {"flower", "church", "horses", "place", "starfish"};

test(names[0], input_flower, thEuc_flower, thMaha_flower, thMaha2_flower,
        thMaha4_flower, thMaha6_flower);
test(names[1], input_church, thEuc_church, thMaha_church, thMaha2_church,
        thMaha4_church, thMaha6_church);
test(names[2], input_horses, thEuc_horses, thMaha_horses, thMaha2_horses,
        thMaha4_horses, thMaha6_horses);
test(names[3], input_place, thEuc_place, thMaha_place, thMaha2_place,
        thMaha4_place, thMaha6_place);
test(names[4], input_starfish, thEuc_starfish, thMaha_starfish, thMaha2_starfish,
        thMaha4_starfish, thMaha6_starfish);

auto startBhatt = std::chrono::system_clock::now();
std::vector<int> channels = {0, 1, 2};
std::vector<int> size = {256, 256, 256};
std::vector<float> ranges = {0, 256, 0, 256, 0, 256};
BhattaDist bd = BhattaDist(channels, size, ranges);

thMaha6_flower.convertTo(thMaha6_flower, CV_8UC1);
thMaha6_church.convertTo(thMaha6_church, CV_8UC1);
thMaha6_horses.convertTo(thMaha6_horses, CV_8UC1);
thMaha6_place.convertTo(thMaha6_place, CV_8UC1);
thMaha6_starfish.convertTo(thMaha6_starfish, CV_8UC1);

double dist_flower2church = bd.calcBetweenImg(input_flower, input_church,
                            thMaha6_flower, thMaha6_church);
double dist_flower2horses = bd.calcBetweenImg(input_flower, input_horses,
                            thMaha6_flower, thMaha6_horses);
double dist_flower2place = bd.calcBetweenImg(input_flower, input_place,
                            thMaha6_flower, thMaha6_place);
double dist_flower2starfish = bd.calcBetweenImg(input_flower, input_starfish,
                            thMaha6_flower, thMaha6_starfish);

double dist_horses2church = bd.calcBetweenImg(input_horses, input_church,
                            thMaha6_horses, thMaha6_church);
double dist_horses2place = bd.calcBetweenImg(input_horses, input_place,
                            thMaha6_horses, thMaha6_place);
double dist_horses2starfish = bd.calcBetweenImg(input_horses, input_starfish,
                            thMaha6_horses, thMaha6_starfish);

auto endBhatt = std::chrono::system_clock::now();
std::chrono::duration<double> elapsed_secondsBhatt = startBhatt-endBhatt;
cout << "elapsed_secondsBhatt:␣" << elapsed_secondsBhatt.count() << endl;

cout << "dist_flower2church:␣" << dist_flower2church << endl;
cout << "dist_flower2horses:␣" << dist_flower2horses << endl;
cout << "dist_flower2place:␣" << dist_flower2place << endl;
cout << "dist_flower2starfish:␣" << dist_flower2starfish << endl;

cout << "dist_horses2church:␣" << dist_horses2church << endl;
cout << "dist_horses2place:␣" << dist_horses2place << endl;
cout << "dist_horses2starfish:␣" << dist_horses2starfish << endl;
```

```
}

void test(String name, Mat& input, Mat& resultEuc, Mat& resultMaha,
          Mat& resultMaha2, Mat& resultMaha4, Mat& resultMaha6) {
    cout << name << "_begin" << endl;

    cv::Mat img = cv::imread("images/"+name+".jpg", cv::IMREAD_COLOR);
    input = img.clone();
    PointCollector pc = PointCollector(img);

    cv::Mat cp = pc.collectedPixels();
    cv::Mat rp = pc.referencePixel();

    cv::Mat cc = pc.collectedCoordinates();
    cv::Mat rc = pc.referenceCoordinate();
    cv::Mat painted_img = img.clone();

    // cout << "pontos extraídos" << endl;

    cv::circle(painted_img, Point(rc), 5, Scalar(255, 255, 255), -1);
    // cout << "círculo grande desenhado" << endl;
    for (int i = 0; i < cp.rows; i++) {
        cv::circle(painted_img, Point(cc.row(i)), 3, Scalar(255, 255, 255), -1);
    }
    // cout << "círculos pequenos desenhados" << endl;

    cv::imwrite("images/"+name+"_painted.jpg", painted_img);

    // cout << "imagem salva" << endl;

    auto startEuc = std::chrono::system_clock::now();
    cv::Mat distsEuc = cv::Mat(img.size(), CV_64F);
    // cout << "matriz criada" << endl;
    for (uint i = 0; i < img.rows; i++) {
        for (uint j = 0; j < img.cols; j++) {
            cv::Mat pointSplit = cv::Mat(3, 1, CV_64F);
            // cout << "matrizinha criada" << endl;
            pointSplit.at<double>(0) = double(img.at<Vec3b>(i, j)[0]);
            pointSplit.at<double>(1) = double(img.at<Vec3b>(i, j)[1]);
            pointSplit.at<double>(2) = double(img.at<Vec3b>(i, j)[2]);
            // cout << "valores setados" << endl;
            distsEuc.at<double>(i, j) = norm(pointSplit - rp);
            // cout << "distância calculada" << endl;
        }
    }
    auto endEuc = std::chrono::system_clock::now();
    std::chrono::duration<double> elapsed_secondsEuc = endEuc-startEuc;
    cout << "elapsed_secondsEuc: " << elapsed_secondsEuc.count() << endl;

    double minEuc, maxEuc;
    cv::minMaxLoc(distsEuc, &minEuc, &maxEuc);
    cv::threshold(distsEuc, resultEuc, maxEuc/2, 255, cv::THRESH_BINARY_INV);
    cv::imwrite("images/"+name+"_thresholdedEuc.jpg", resultEuc);

    auto startMahala = std::chrono::system_clock::now();
    MahalaDist md = MahalaDist(cp, 4e-6, rp);
    md.build();
```

```
    cv::Mat resultMahala = md.imageToReference<uchar>(img);
    auto endMahala = std::chrono::system_clock::now();
    std::chrono::duration<double> elapsed_secondsMahala = endMahala-startMahala;
    cout << "elapsed_secondsMahala:␣" << elapsed_secondsMahala.count() << endl;

    double minMahala, maxMahala;
    cv::minMaxLoc(resultMahala, &minMahala, &maxMahala);
    cv::threshold(resultMahala, resultMaha, maxMahala/2, 255, cv::THRESH_BINARY_INV);
    cv::imwrite("images/"+name+"_thresholdedMahala.jpg", resultMaha);

    auto startPolyMahala2 = std::chrono::system_clock::now();
    PolyMahalaDist pmd2 = PolyMahalaDist(cp, 2, 4e-6, rp);
    cv::Mat resultPolyMahala2 = pmd2.imageToReference<uchar>(img);
    auto endPolyMahala2 = std::chrono::system_clock::now();
    std::chrono::duration<double> elapsed_secondsPolyMahala2 = endPolyMahala2-startPolyMahala2;
    cout << "elapsed_secondsPolyMahala2:␣" << elapsed_secondsPolyMahala2.count() << endl;

    double minPolyMahala2, maxPolyMahala2;
    cv::minMaxLoc(resultMahala, &minPolyMahala2, &maxPolyMahala2);
    cv::threshold(resultPolyMahala2, resultMaha2, 1, 255, cv::THRESH_BINARY_INV);
    cv::imwrite("images/"+name+"_thresholdedPolyMahala2.jpg", resultMaha2);

    auto startPolyMahala4 = std::chrono::system_clock::now();
    PolyMahalaDist pmd4 = PolyMahalaDist(cp, 4, 4e-6, rp);
    cv::Mat resultPolyMahala4 = pmd4.imageToReference<uchar>(img);
    auto endPolyMahala4 = std::chrono::system_clock::now();
    std::chrono::duration<double> elapsed_secondsPolyMahala4 = endPolyMahala4-startPolyMahala4;
    cout << "elapsed_secondsPolyMahala4:␣" << elapsed_secondsPolyMahala4.count() << endl;

    double minPolyMahala4, maxPolyMahala4;
    cv::minMaxLoc(resultMahala, &minPolyMahala4, &maxPolyMahala4);
    cv::threshold(resultPolyMahala4, resultMaha4, 1, 255, cv::THRESH_BINARY_INV);
    cv::imwrite("images/"+name+"_thresholdedPolyMahala4.jpg", resultMaha4);

    auto startPolyMahala6 = std::chrono::system_clock::now();
    PolyMahalaDist pmd6 = PolyMahalaDist(cp, 6, 4e-6, rp);
    cv::Mat resultPolyMahala6 = pmd6.imageToReference<uchar>(img);
    auto endPolyMahala6 = std::chrono::system_clock::now();
    std::chrono::duration<double> elapsed_secondsPolyMahala6 = endPolyMahala6-startPolyMahala6;
    cout << "elapsed_secondsPolyMahala6:␣" << elapsed_secondsPolyMahala6.count() << endl;

    double minPolyMahala6, maxPolyMahala6;
    cv::minMaxLoc(resultMahala, &minPolyMahala6, &maxPolyMahala6);
    cv::threshold(resultPolyMahala6, resultMaha6, 1, 255, cv::THRESH_BINARY_INV);
    cv::imwrite("images/"+name+"_thresholdedPolyMahala6.jpg", resultMaha6);
    cout << name << "_end" << endl;
}
```

//end_file

ANEXO A – DOCUMENTAÇÃO DA BIBLIOTECA

libmahala

Generated by Doxygen 1.9.2

Chapter 1

Class Index

1.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 2

Class Documentation

2.1 BhattaDist Class Reference

Class that holds the necessary parameters to transform points neighborhood matrices into histograms to allow the calculation of the Bhattacharyya distance between them. For more information, check the documentation for the calcHist() function of the OpenCV library.

`#include <BhattacharyyaDistance.hpp>`

Public Member Functions

- BhattaDist (vector< int > channels, vector< int > histSize, vector< float > ranges)

  Constructor.
- vector< int > **channels** ()
- vector< int > **histSize** ()
- vector< float > **ranges** ()
- double calcBetweenPoints (Mat &points1, Mat &points2)

  Calculates the Bhattacharyya distance between two sets of points.
- double calcBetweenImg (const Mat &image1, const Mat &image2, const Mat &mask1=Mat(), const Mat &mask2=Mat())

  Calculates the Bhattacharyya distance between two images.

Static Public Member Functions

- static double calcBetweenHist (const Mat &hist1, const Mat &hist2)

  Calculates the Bhattacharyya distance between two histograms.

2.1.1 Detailed Description

Class that holds the necessary parameters to transform points neighborhood matrices into histograms to allow the calculation of the Bhattacharyya distance between them. For more information, check the documentation for the calcHist() function of the OpenCV library.

Parameters

channels	Dimensions of the points to be considered (read only)
histSize	Vector of the sizes of the histogram in each dimension (read only)
ranges	Vector with the lower and upper limits of the values for each dimension (read only)

2.1.2 Constructor & Destructor Documentation

2.1.2.1 BhattaDist()

```
BhattaDist::BhattaDist (
            vector< int > channels,
            vector< int > histSize,
            vector< float > ranges )
```

Constructor.

Parameters

channels	Dimensions of the points to be considered
histSize	Vector of the sizes of the histogram in each dimension
ranges	Vector with the lower and upper limits of the values for each dimension

2.1.3 Member Function Documentation

2.1.3.1 calcBetweenHist()

```
double BhattaDist::calcBetweenHist (
            const Mat & hist1,
            const Mat & hist2 )  [static]
```

Calculates the Bhattacharyya distance between two histograms.

Parameters

hist1	The first operand, a histogram
hist2	The second operand, a histogram

Returns

The value of the Bhattacharyya distance between the two histograms

2.1.3.2 calcBetweenImg()

```
double BhattaDist::calcBetweenImg (
            const Mat & image1,
            const Mat & image2,
            const Mat & mask1 = Mat(),
            const Mat & mask2 = Mat() )
```

Calculates the Bhattacharyya distance between two images.

Parameters

image1	The first operand, an image
image2	The second operand, an image
mask1	Mask to be applied to the first operand to filter which points will be considered when calculating the histogram
mask2	Mask to be applied to the second operand to filter which points will be considered when calculating the histogram

Returns

The value of the Bhattacharyya distance between the two images

2.1.3.3 calcBetweenPoints()

```
double BhattaDist::calcBetweenPoints (
            Mat & points1,
            Mat & points2 )
```

Calculates the Bhattacharyya distance between two sets of points.

Parameters

points1	The first operand, a matrix that contains a set of points, of which each row is a point and each column is a dimension
points2	The second operand, a matrix that contains a set of points, of which each row is a point and each column is a dimension

Returns

The value of the Bhattacharyya distance between the two sets of points

The documentation for this class was generated from the following files:

- BhattacharyyaDistance.hpp
- BhattacharyyaDistance.cpp

2.2 MahalaDist Class Reference

Class that holds the necessary parameters to calculate the Mahalanobis distance between points and performs the calculation.

`#include <MahalanobisDistance.hpp>`

Public Member Functions

- MahalaDist (const Mat &input, double smin=4e-6, Mat reference=Mat())

  Constructor.
- Mat **reference** ()
- double **smin** ()
- int **dimension** ()
- bool **dirty** ()
- const Mat **u** () const
- Mat **w** ()
- Mat **c** ()
- Mat **cSigma2Inv** ()
- double **sigma2** ()
- void **smin** (double smin)
- void build ()

  Calculates the parameters that are dependent on smin and that are necessary to calculate the distance.
- double pointTo (Mat &point1, Mat &point2)

  Calculates the Mahalanobis distance between two points.
- double pointToReference (Mat &point)

  Calculates the Mahalanobis distance between a point and the center of the metric.
- Mat pointsTo (Mat &points, Mat &ref)

  Calculates the Mahalanobis distance between a set of points and a single point.
- Mat pointsToReference (Mat &points)

  Calculates the Mahalanobis distance between a set of points and the center of the metric.
- template<typename T >
  Mat imageTo (Mat &image, Mat &ref)

  Transforms an image into a set of points and calculates the Mahalanobis distance between it and a single reference point.
- template<typename T >
  Mat imageToReference (Mat &image)

  Transforms an image into a set of points and calculates the Mahalanobis distance between it and the center of the metric.

2.2.1 Detailed Description

Class that holds the necessary parameters to calculate the Mahalanobis distance between points and performs the calculation.

Parameters

c	The covariance matrix of a statistical population of points (read only)
u	Lateral matrix resultant of an SVC operation in C $*$ (N-1), in which C is the covariance matrix of the object and N is the size of the population of points (read only)

Parameters

w	Central matrix resultant of an SVC operation in C ∗ (N-1), in which C is the covariance matrix of the object and N is the size of the population of points (read only)
reference	Center of the metric (mean of the population of points, by default) (read only)
cSigma2Inv	Matrix used to calculate the Mahalanobis distance instead of the inverse of the covariance matrix of the population of points C, calculated using C and smin (read only)
smin	User provided parameter, used to guarantee that the covariance matrix of the population of points will be inversible, and altering the result of the distance function proportionally to its value (read and write)
sigma2	Value derived of smin, used to calculate cSigma2Inv (read only)
dimension	Number of dimensions of the population of points used to build the metric (read only)
dirty	Indicates whether the build() method should be called before the distances can be calculated (read only)

2.2.2 Constructor & Destructor Documentation

2.2.2.1 MahalaDist()

```
MahalaDist::MahalaDist (
            const Mat & input,
            double smin = 4e-6,
            Mat reference = Mat() )
```

Constructor.

Parameters

input	Matrix that holds a statistical population of points, in which each row is a point and each column a dimension
smin	Parameter, used to guarantee that the covariance matrix of the population of points will be inversible, and altering the result of the distance function proportionally to its value
reference	Center of the metric (mean of the population of points, by default) in the form of a column matrix

2.2.3 Member Function Documentation

2.2.3.1 imageTo()

```
template<typename T >
Mat MahalaDist::imageTo (
            Mat & image,
            Mat & ref )
```

Transforms an image into a set of points and calculates the Mahalanobis distance between it and a single reference point.

Template Parameters

T	The same type that would be used in the at<T>() method of the cv::Mat class of the OpenCV library to access an element of said image

Parameters

image	The first operand, an image
ref	The second operand, a point in the form of a column matrix

Returns

A single channel image in which the value of each pixel is the value of the Mahalanobis distance between the correspondent pixel in the first operand and the reference point

2.2.3.2 imageToReference()

```
template<typename T >
Mat MahalaDist::imageToReference (
            Mat & image )
```

Transforms an image into a set of points and calculates the Mahalanobis distance between it and the center of the metric.

Template Parameters

T	The same type that would be used in the at<T>() method of the cv::Mat class of the OpenCV library to access an element of said image

Parameters

image	An image

Returns

A single channel image in which the value of each pixel is the value of the Mahalanobis distance between the correspondent pixel in the image parameter and the center of the metric

2.2.3.3 pointsTo()

```
Mat MahalaDist::pointsTo (
            Mat & points,
            Mat & ref )
```

Calculates the Mahalanobis distance between a set of points and a single point.

Parameters

points	The first operand, a matrix that represent a set of points, in which each row is a point and each column a dimension
ref	The second operand, a single point in the form of a column matrix

Returns

Matrix with all the values of the Mahalanobis distances between the correspondent point in the point set and the singular reference point

2.2.3.4 pointsToReference()

```
Mat MahalaDist::pointsToReference (
            Mat & points )
```

Calculates the Mahalanobis distance between a set of points and the center of the metric.

Parameters

points	A matrix that represent a set of points, in which each row is a point and each column a dimension

Returns

Matrix with all the values of the Mahalanobis distances between the correspondent point in the point set and the center of the metric

2.2.3.5 pointTo()

```
double MahalaDist::pointTo (
            Mat & point1,
            Mat & point2 )
```

Calculates the Mahalanobis distance between two points.

Parameters

point1	The first operand, a point in the form of a column matrix
point2	The second operand, a point in the form of a column matrix

Returns

The value of the Mahalanobis distance between the points

2.2.3.6 pointToReference()

```
double MahalaDist::pointToReference (
            Mat & point )
```

Calculates the Mahalanobis distance between a point and the center of the metric.

Parameters

point	A point in the form of a column matrix

Returns

The value of the Mahalanobis distance between the points

The documentation for this class was generated from the following files:

- MahalanobisDistance.hpp
- MahalanobisDistance.cpp

2.3 PointCollector Class Reference

Open an interactive window with an image on which the user can click on to extract the values and coordinates of pixels (only once per pixel)

```
#include <PointCollector.hpp>
```

Public Member Functions

- PointCollector (Mat &input)
  Constructor that opens, maintains, and then closes a window using an input image as a matrix.
- PointCollector (const char *path, int flags)
  Constructor that opens, maintains, and then closes a window using an input image read from an image file.
- Mat & **collectedPixels** ()
- Mat & **collectedCoordinates** ()
- Mat & **referencePixel** ()
- Mat & **referenceCoordinate** ()

2.3.1 Detailed Description

Open an interactive window with an image on which the user can click on to extract the values and coordinates of pixels (only once per pixel)

Parameters

collectedPixels	Values of the extracted pixels, in which each row is a pixel and and each column is a channel (read only)
collectedCoordinates	Coordinates of each pixel, in which each row is a pixel (read only)
referencePixel	Values of the reference pixel chosen with the right mouse button (read only)
referenceCoordinate	Coordinates of the reference pixel chosen with the right mouse button (read only)

2.3.2 Constructor & Destructor Documentation

2.3.2.1 PointCollector() [1/2]

```
PointCollector::PointCollector (
            Mat & input )
```

Constructor that opens, maintains, and then closes a window using an input image as a matrix.

Parameters

input	Image that will be shown in the window

2.3.2.2 PointCollector() [2/2]

```
PointCollector::PointCollector (
            const char * path,
            int flags )
```

Constructor that opens, maintains, and then closes a window using an input image read from an image file.

Parameters

path	Path to the file that contain the image that will be shown in the window
flags	Flags used by the imread() function of the OpenCV library to open the image file

The documentation for this class was generated from the following files:

- PointCollector.hpp
- PointCollector.cpp

2.4 PolyMahalaDist Class Reference

Class that holds the necessary parameters to calculate the polynomial Mahalanobis distance between points and performs the calculation.

```
#include <PolynomialMahalanobisDistance.hpp>
```

Public Member Functions

- PolyMahalaDist (const Mat &input, int order, double eps_svd=4e-6, Mat reference=Mat())

  Constructor.
- Mat **reference** ()
- double **eps_svd** ()
- int **dimension** ()
- int **order** ()
- double pointTo (Mat &im_data, Mat &refVector)

  builds the parameters that are used on the calculation of the distances and are based on the _smin and _I class members / void build();
- double pointToReference (Mat &im_data)

  Calculates the polynomial Mahalanobis distance between a point and the center of the metric.
- Mat pointsTo (Mat &im_data, Mat &refVector)

  Calculates the polynomial Mahalanobis distance between a set of points and a single point.
- Mat pointsToReference (Mat &im_data)

  Calculates the polynomial Mahalanobis distance between a set of points and the center of the metric.
- template<typename T >
  Mat imageTo (Mat &image, Mat &refVector)

  Transforms an image into a set of points and calculates the polynomial Mahalanobis distance between it and a single reference point.
- template<typename T >
  Mat imageToReference (Mat &image)

  Transforms an image into a set of points and calculates the polynomial Mahalanobis distance between it and the center of the metric.

2.4.1 Detailed Description

Class that holds the necessary parameters to calculate the polynomial Mahalanobis distance between points and performs the calculation.

Parameters

reference	Center of the metric (mean of the population of points, by default) (read only)
cSigma2Inv	Matrix used to calculate the Mahalanobis distance instead of the inverse of the covariance matrix of the population of points C, calculated using C and smin (read only)
eps_svd	User provided parameter, used to guarantee that the covariance matrix of the population of points will be inversible, and altering the result of the distance function proportionally to its value (read and write)
order	Highest polynomial order of the expanded terms (read only)
dimension	Number of dimensions of the population of points used to build the metric (read only)

2.4.2 Constructor & Destructor Documentation

2.4.2.1 PolyMahalaDist()

```
PolyMahalaDist::PolyMahalaDist (
            const Mat & input,
```

```
            int order,
            double eps_svd = 4e-6,
            Mat reference = Mat() )
```

Constructor.

Parameters

input	Matrix that holds a statistical population of points, in which each row is a point and each column a dimension
order	Highest polynomial order of the expanded terms
eps_svd	User provided parameter, used to guarantee that the covariance matrix of the population of points will be inversible, and altering the result of the distance function proportionally to its value
reference	Center of the metric (mean of the population of points, by default) in the form of a column matrix

2.4.3 Member Function Documentation

2.4.3.1 imageTo()

```
template<typename T >
Mat PolyMahalaDist::imageTo (
            Mat & image,
            Mat & refVector )
```

Transforms an image into a set of points and calculates the polynomial Mahalanobis distance between it and a single reference point.

Template Parameters

T	The same type that would be used in the at<T>() method of the cv::Mat class of the OpenCV library to access an element of said image

Parameters

image	The first operand, an image
ref	The second operand, a point in the form of a column matrix

Returns

A single channel image in which the value of each pixel is the value of the polynomial Mahalanobis distance between the correspondent pixel in the first operand and the reference point

2.4.3.2 imageToReference()

```
template<typename T >
Mat PolyMahalaDist::imageToReference (
            Mat & image )
```

Transforms an image into a set of points and calculates the polynomial Mahalanobis distance between it and the center of the metric.

Template Parameters

T	The same type that would be used in the at<T>() method of the cv::Mat class of the OpenCV library to access an element of said image

Parameters

image	An image

Returns

A single channel image in which the value of each pixel is the value of the polynomial Mahalanobis distance between the correspondent pixel in the image parameter and the center of the metric

2.4.3.3 pointsTo()

```
Mat PolyMahalaDist::pointsTo (
            Mat & im_data,
            Mat & refVector )
```

Calculates the polynomial Mahalanobis distance between a set of points and a single point.

Parameters

points	The first operand, a matrix that represent a set of points, in which each row is a point and each column a dimension
ref	The second operand, a single point in the form of a column matrix

Returns

Matrix with all the values of the polynomial Mahalanobis distances between the correspondent point in the point set and the singular reference point

2.4.3.4 pointsToReference()

```
Mat PolyMahalaDist::pointsToReference (
            Mat & im_data )
```

Calculates the polynomial Mahalanobis distance between a set of points and the center of the metric.

Parameters

points	A matrix that represent a set of points, in which each row is a point and each column a dimension

Returns

Matrix with all the values of the polynomial Mahalanobis distances between the correspondent point in the point set and the center of the metric

2.4.3.5 pointTo()

```
double PolyMahalaDist::pointTo (
            Mat & im_data,
            Mat & refVector )
```

builds the parameters that are used on the calculation of the distances and are based on the _smin and _l class members / void build();

```
/*! \brief Calculates the polynomial Mahalanobis distance between two points
    \param point1 The first operand, a point in the form of a column matrix
    \param point2 The second operand, a point in the form of a column matrix
    \return The value of the polynomial Mahalanobis distance between the points
```

2.4.3.6 pointToReference()

```
double PolyMahalaDist::pointToReference (
            Mat & im_data )
```

Calculates the polynomial Mahalanobis distance between a point and the center of the metric.

Parameters

point	A point in the form of a column matrix

Returns

The value of the polynomial Mahalanobis distance between the points

The documentation for this class was generated from the following files:

- PolynomialMahalanobisDistance.hpp
- PolynomialMahalanobisDistance.cpp

Index