Modeling activity: Digital analysis of fingerprints

Nicolas CAVREL - Yassine FAKIHANI - Ben SARFATI

Generated by Doxygen 1.8.13

Contents

Chapter 1

Namespace Index

1.1 Namespace List

Here is a list of all namespaces with brief descriptions:

Chapter 2

Class Index

2.1 Class List

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

Chapter 3

File Index

3.1 File List

Here is a list of all files with brief descriptions:

Chapter 4

Namespace Documentation

4.1 cv Namespace Reference

4.1.1 Detailed Description

The OpenCV (Open Source Computer Vision Library) library

4.2 Eigen Namespace Reference

4.2.1 Detailed Description

A C++ template library for linear algebra

4.3 std Namespace Reference

4.3.1 Detailed Description

The C++ Standard Library

Chapter 5

Class Documentation

5.1 Coordinates Class Reference

The class "Coordinates" contains one constructor and most of the methods we used to define many functions in the class “image”. It represents the position of one pixel in the attribute matrix accordingly to the basis we choose to do this project.

```
#include <coordinates.h>
```

Public Member Functions

- Coordinates (float, float)

  This constructor allows us to create an instance of this class by initializing the attributes x and y thanks to the given parameters.
- float x_get ()

  Here we define a getter to know the value of the coordinate along the X-axis.
- float y_get ()

  Here we define another getter to know the value of the coordinate along the Y-axis.
- Coordinates operator+ (Coordinates)

  We overloaded the operator “+” to be able to add two instances of type “coordinates”.
- Coordinates operator+ (float)

  We overloaded the operator “+” to add the actual instance with an element of type float.
- Coordinates operator- (Coordinates)

  We overloaded the operator “-” to be able to subtract two instances of type “coordinates”.
- Coordinates operator- (float)

  We overloaded again the operator “-” but this time, we will subtract the actual instance with an element of type float.
- float norm ()

  This method calculates the Euclidean norm of an instance of this type, I.e. its distance from the origin of the basis.
- void rotation (Coordinates &, float)

  This method performs the rotation of a point (I.e. an instance of type Coordinates) around a rotation point and by a given angle, using the formula described in the report.

Friends

- ostream & operator<< (ostream &, const Coordinates &)

  Declaration of a friend independent function in order to overload the operator “<<” for this class.

5.1.1 Detailed Description

The class "Coordinates" contains one constructor and most of the methods we used to define many functions in the class “image”. It represents the position of one pixel in the attribute matrix accordingly to the basis we choose to do this project.

Definition at line 25 of file coordinates.h.

5.1.2 Constructor & Destructor Documentation

5.1.2.1 Coordinates()

```
Coordinates::Coordinates (
            float x = 0,
            float y = 0 )
```

This constructor allows us to create an instance of this class by initializing the attributes x and y thanks to the given parameters.

Parameters

float←_x	float←_y

Returns

Nothing

Definition at line 3 of file coordinates.cpp.

5.1.3 Member Function Documentation

5.1.3.1 norm()

```
float Coordinates::norm ( )
```

This method calculates the Euclidean norm of an instance of this type, I.e. its distance from the origin of the basis.

Parameters

Nothing	

Returns

A float which represent the Euclidean norm of an instance.

Definition at line 46 of file coordinates.cpp.

5.1.3.2 operator+() [1/2]

```
Coordinates Coordinates::operator+ (
             Coordinates p )
```

We overloaded the operator "+" to be able to add two instances of type "coordinates".

Parameters

Instance_of_type_coordiantes	

Returns

The result is an instance of this type, its value is the sum of the two previous ones, coordinate by coordinate.

Definition at line 16 of file coordinates.cpp.

5.1.3.3 operator+() [2/2]

```
Coordinates Coordinates::operator+ (
             float n )
```

We overloaded the operator "+" to add the actual instance with an element of type float.

Parameters

Float_number	

Returns

The result is an instance of this type, its value is the sum of the previous one plus the float number.

Definition at line 22 of file coordinates.cpp.

5.1.3.4 operator-() [1/2]

```
Coordinates Coordinates::operator- (
             Coordinates p )
```

We overloaded the operator "-" to be able to subtract two instances of type "coordinates".

Parameters

Instance_of_type_coordiantes	

Returns

The result is an instance of this type, its value is the subtraction of the two previous ones, coordinate by coordinate.

Definition at line 28 of file coordinates.cpp.

5.1.3.5 operator-() [2/2]

```
Coordinates Coordinates::operator- (
            float n )
```

We overloaded again the operator “-” but this time, we will subtract the actual instance with an element of type float.

Parameters

Float_number	

Returns

The result is an instance of this type, its value is the subtraction of the previous instance minus the float number.

Definition at line 34 of file coordinates.cpp.

5.1.3.6 rotation()

```
void Coordinates::rotation (
            Coordinates & rotation_point,
            float angle )
```

This method performs the rotation of a point (I.e. an instance of type Coordinates) around a rotation point and by a given angle, using the formula described in the report.

Parameters

Rotation_point	Angle

Returns

Nothing, but it modifies the values of the attributes x and y.

Definition at line 50 of file coordinates.cpp.

5.1.3.7 x_get()

```
float Coordinates::x_get ( )
```

Here we define a getter to know the value of the coordinate along the X-axis.

Returns

The value of the attribute x.

Definition at line 8 of file coordinates.cpp.

5.1.3.8 y_get()

```
float Coordinates::y_get ( )
```

Here we define another getter to know the value of the coordinate along the Y-axis.

Returns

The value of the attribute y.

Definition at line 12 of file coordinates.cpp.

5.1.4 Friends And Related Function Documentation

5.1.4.1 operator<<

```
ostream& operator<< (
            ostream & o,
            const Coordinates & p )  [friend]
```

Declaration of a friend independent function in order to overload the operator “<<” for this class.

Definition at line 40 of file coordinates.cpp.

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

- include/coordinates.h
- src/coordinates.cpp

5.2 image Class Reference

The class image contains three constructors and most of the functions we defined to respond to many questions.

```
#include <image.h>
```

Public Member Functions

- image (unsigned int, unsigned int, unsigned)

  Constructor 1: Define an image from its dimensions and number of pixels.
- image (string)

  Constructor 2: Creating an instance of type "image" from its path in the OS.
- image (Mat)

  Constructor 3: We initialize all the attributes from an instance of type Mat.
- unsigned int width_get ()

  Here we define a getter to know the width of an image.
- unsigned int height_get ()

  Here we define another getter to know the height of an image.
- Mat matrix_get ()

  Another getter to know the values of attribute of type "Mat".
- unsigned int return_pixel_value (unsigned int, unsigned int)

  A getter to know the intensity value of the pixel at the coordinate (x, y). If im is an instance of this class. To know the pixel intensity at the position (50,50), we can do: im.return_pixel_value(50,50).
- void white_square (Coordinates, Coordinates)

  We create white squares at a given position. Each position is of type "Coordinates". It raises an error if there no logical values of both parameters. In order to draw a white square in the image im such that the beginning of the square is the point (30,30) and its end is (50,65), we can do: im.white_square(Coordinates(30,30),Coordinates(50,65)).
- void black_square (Coordinates, Coordinates)

  We create black squares at a given position. Each position is of type "Coordinates". In order to draw a black square in the image im such that the beginning of the square is the point (30,30) and its end is (50,65), we can do: im.←black_square(Coordinates(30,30),Coordinates(50,65)).
- unsigned int return_max_intensity ()

  This method calculates the maximum intensity value. It uses the method "return_pixel_value" to browse all the elements of attribute "matrix".
- unsigned int return_min_intensity ()

  This method calculates the minimum intensity of an image.
- void save_image (string)

  In order to save an image after some modification we define the following method. We save them with the ".png" extension.
- void return_symetry_y ()

  Performing the symmetry of an image along the Y-axis.
- void return_symetry_x ()

  To perform the symmetry of the image along the X-axis.
- void return_diagonal_symetry ()

  The diagonal symmetry which is a combination of the symmetry along the X axis and the one along the Y axis. So this function uses the methods "return_symetry_x" and "return_symetry_y". We can print the transformed image in the screen thanks to the operator "<<".
- void balance_intensity_exp (Coordinates, Coordinates, unsigned int)

  This method balances the intensity of the current instance of image throught the exponential function. The user has to provide two coordinates corresponding to the two opposite edge of the rectangle of application. The third parameter is reguling the strentgh of the balancing function : the bigger is param_intensity the stronger is the function.
- void balance_intensity_quadratic (Coordinates, Coordinates, unsigned int)

This method balances the intensity of the current instance of image throught the quadratic function. The user has to provide two coordinates corresponding to the two opposite edge of the rectangle of application. The third parameter is reguling the strentgh of the balancing function : the bigger is param_intensity the stronger is the function.

- void balance_intensity_normal_2D (Coordinates, Coordinates, unsigned int)

  This method balances the intensity of the current instance of image throught the exponential function. This time the method will be applied considering the elliptic shape of the fingerpint. The user has to provide two coordinates corresponding to the two opposite edge of the rectangle of application. The third parameter is reguling the strentgh of the balancing function : the bigger is param_intensity the stronger is the function.

- Coordinates barycenter ()

  We created this method in order to find the barycenter of an image, because we considered that the barycenter of an image is equals to the center of pressure.

- void print_barycenter ()

  We used the method “barycenter” to know the coordinates of the barycenter, then we applied the function "black_↩ square" to print it.

- vector< Coordinates > contour ()

  This method computes the coordinates of the contour of the finger inside the image. It uses both classes "vector" and "coordinates".

- void print_contour ()

  This method uses the functions “contour " and “black_square” for the purpose of printing the contour of the finger in the image. In order to print the contour of the finger in an image represented by the instance im, we can do as follow: im.print_contour(). It modifies the attribute matrix of the image. To see the modifications, all what we have to do is to use the overloaded operator “<<”.

- vector< float > ellipse_parameters ()

  This method estimates the two ellipse parameters a and b using the first naive and unoptimized method (taking the min and max distance from the center of the ellipse).

- vector< float > ellipse_parameters_gradient (float)

  This method estimates the two ellipse parameters a and b using the gradient descent method. We need here to provide a step size for the method in the epsilon parameter. The smaller is epsilon and the preciser is the algorithm, but it also makes it slower.

- void print_ellipse ()

  Computes and prints the best matching ellipse using the gradient descent algorithm. It uses a generic step for the gradient method. This function doesn't modify the current instance of image.

- void translation (float, float)

  The translation along the X-axis and the Y-axis using the bilinear interpolation. The algorithm we use here to interpolate is like the one in the method “rotation_interpolation_bilinear”. In order to translate an image -represented by the instance im- with 11 pixels following the X-axis and 30 pixels following the Y-axis, we can do as follow: im.↩ translation(11.,30.).

- void rotation_interpolation_bilinear (Coordinates, float)

  Thanks to this function we can effectuate a rotation of the image around a point and with a specific angle, both given as parameters. Here we interpolate using the bilinear interpolation (In the report we detailed the theoretical calculation) which is quite good, but not as good as the interpolation with the bicubic method. To rotate an image -represented by the instance im- around its barycenter and with an angle PI/4, we can do as follow: im.rotation_↩ interpolation_bilinear(im.barycenter(),PI/4)

- void rotation_interpolation_bicubis (Coordinates, float)

  To do a better rotation of an image around a point and with a specific angle, we implemented this method where we interpolate using the bicubic interpolation thanks to the independent function “interpolation_bicubique”. Also, inside this method we normalize all the values resulting from the execution of the independent function. To rotate an image -represented by the instance im- around its barycenter and with an angle PI/4, we can do as follow: im.rotation_↩ interpolation_bicubis(im.barycenter(),PI/4)

- void rotation_warping (Coordinates, float, float)

  Performs the rotation warping of an image, which is a rotation decreasing of strength the further away you are from the center. This center is defined by the rotation_center parameter. A negative rotation_strength is perfoming the rotation in the other way. This method uses a bilinear interpolation to do so.

- void translation_warping_x (Coordinates, Coordinates, Coordinates, float)

  Performs the translation warping according the Ox axis, which is a stretching of the image. You can choose either to compress or to zoom on the image by changing the positivity of the strength of the strength_x parameter. The parameter center defines the center of the stretching and the parameters beg and end the zone of application of the function.

- void translation_warping_y (Coordinates, Coordinates, Coordinates, float)

    Similar to the translation_warping_x funtion but applied to the Oy axis.
- void translation_warping (Coordinates, Coordinates, Coordinates, float, float)

    This function allows to apply the functions translation_warping_x and translation_warping_y one after this other.
- void bigger_image ()

    In a way to save information, and ovoid any loss of pixels intensity when making a rotation, we'll start computing rotations with bigger images, then we call the method "smaller_image" to come back to the normal dimensions of the image. If im is an instance of this class, we can do like this to make bigger this image: im.bigger_image()
- void smaller_image ()

    After calling the method "bigger_image" we need to return an image with the exact same dimension of the original one. That's why we must implement the inverse method which is "smaller_image".
- int detect_trans_along_x_with_first_loss_function (image &)

    This function deal with the case where the wrap function is a translation along the X-axis such that there is only a single translation parameter px to estimate. Here we use the first lost function. We didn't call the one defined independently named "first_loss_function". it can take time to be executed. If im1 and im2 are two instances representing the same image, in order to test this function, we can do like this: im2.translation(4.,0.) then im1.detect_trans_along←_x_with_first_loss_function(im2) and the result returned by this method will be 4! Here you can test also negative values!
- void detect_trans_along_x_and_y_with_first_loss_function (image &)

    This function deal with the case where the wrap function is a translation along the X-axis and Y-axis such that there are two translation parameters px and py to estimate. Here we use the first loss function. We didn't call the one defined independently named "first_loss_function". The execution of this function may take a lot of time because it's a greedy startegy. If im1 and im2 are two instances representing the same image. In order to test this function, we can do like this: im2.translation(10.,11.) then im1.detect_trans_along_x_and_y_with_first_loss_function(im2) and the result printed on the screen will be (10.,11.) ! Here you can test also negative values!
- void detect_trans_along_x_and_y_with_second_loss_function (image &)

    The execution of this function may take a lot of time because it's a greedy strategy. This function is like the one called "detect_trans_along_x_and_y_with_first_loss_function", because it deals with the case where the wrap function is a translation along the X-axis and Y-axis such that there are two translation parameters px and py to estimate. Its advantage is that it uses (implicitly) the second loss function. So, we didn't call the method named "second_loss_←function". Finally, it's worth noting that in this method we maximize the second loss function and we do not minimize it like we did in the method "detect_trans_along_x_and_y_with_first_loss_function". If im1 and im2 are two instances representing the same image, in order to test this function, we can do like this: im2.translation(10.,11.) then im1.←detect_trans_along_x_and_y_with_second_loss_function(im2) and the result printed on the screen will be (10.,11.) ! Here you can test also negative values!
- void rotation (Coordinates, float)

    Here we do a transformation of an image using the rotation around a point and with an angle without any interpolation, using the mathematical formula we gave in the report and two methods of the class coordiantes. To rotate an image -represented by the instance im- around its barycenter and with an angle PI/4, we can do as follow: im.rotation(im1.←barycenter(),PI/4).
- float detect_rot (image &, Coordinates, int)

    This function detects if the current instance of image is the rotation of the image contained in the image_rotation parameter. We also have to specify where we want the rotation to happen with the rotation_center parameter. As it is impossible in general to find the exact rotation of a picture and another (if the image is a rotation of exactly sqrt(2) then it would take an infinite time to compute the exact rotation value) we have to specify the number of step we want to do with the pameter nb_test : the more step and the more accurate is the result, but the slower is the algorithm...
- vector$<$ float $>$ detect_trans (image &, int, int)

    This function detect if the current instance of image is a translation of the one contained in the image_trans parameter. As for the detect_rot function, we have to define the number of operations we want to perform along the Ox and Oy axis. This can be done by inputing it in the parameters nb_testx and nb_testy.
- int first_loss_function (image &)

    We define here the first loss function which is the sum of squared errors between the pixels of two images. The sum is taken over all pixels of images.
- float second_loss_function (image &)

    We define here the second loss function where we calculate the mean of the pixel's intensity of both images. We browse all the pixels of both images.
- vector$<$ float $>$ detect_warp (image &, int, int, int)

This function detects if the current instance of image is a warp of the image contained in the parameter image←↩ _compare using the first naive method (trying out every rotation/translation and computing the error on the whole image). We have to profide the number of test we want to perform on the rotation, on the transalations along the Ox and Oy axises respectively in the nb_test_rot, nb_testx and nb_testy parameters. Once again, the bigger are these numbers and the slower is the algorithm but the more accurate is the result.

- vector< float > detect_warp_small_square (image &, int, int)

  This function detects if the current instance of image is a warp of the image contained in the parameter image_←↩ compare using optimized small square method (we here try to match only a small square of the image). In this method, we just have to give the number of test we want to perform on the rotation throught the nb_test_rot parameter. We no longer need the number of test for the translation since the efficiency of the algorithm allows us to test every integer values for the translation. We can also set the size of the test square with square_size parameter (in number of pixels).

- image create_small_image (Coordinates, int)

  Auxiliary function for the detect_warp_small_square method. It helps creating the small piece of image to compare with the test square. It will create a square image centered on the coordinate point an of size image_size.

Friends

- ostream & operator<< (ostream &, const image &)

  In order to overload the operator "<<" we defined this function, which must be declared as a friend function.

5.2.1 Detailed Description

The class image contains three constructors and most of the functions we defined to respond to many questions.

Definition at line 33 of file image.h.

5.2.2 Constructor & Destructor Documentation

5.2.2.1 image() [1/3]

```
image::image (
            unsigned int,
            unsigned int,
            unsigned  )
```

Constructor 1: Define an image from its dimensions and number of pixels.

Parameters

height	width number_of_pixel

Returns

Nothing

5.2.2.2 image() [2/3]

```
image::image (
            string image_path )
```

Constructor 2: Creating an instance of type “image” from its path in the OS.

We assume that all of the images we will process are grayscale and we use the OpenCV's methods to initialize all the attributes. For instance, an argument of this class may be: /home/Bureau/UGA/MSIAM-1/Project-Janvier/clean_↩ finger.png

Parameters

image_path	

Returns

Nothing

Definition at line 10 of file image.cpp.

5.2.2.3 image() [3/3]

```
image::image (
            Mat M )
```

Constructor 3: We initialize all the attributes from an instance of type Mat.

Parameters

Matrix_of_type_Mat	

Returns

Nothing

Definition at line 18 of file image.cpp.

5.2.3 Member Function Documentation

5.2.3.1 balance_intensity_exp()

```
void image::balance_intensity_exp (
            Coordinates beg,
            Coordinates end,
            unsigned int param_intensity )
```

This method balances the intensity of the current instance of image throught the exponential function. The user has to provide two coordinates corresponding to the two opposite edge of the rectangle of application. The third parameter is reguling the strentgh of the balancing function : the bigger is param_intensity the stronger is the function.

Parameters

beg	end param_intensity

Returns

Nothing but modifies the instance on which it is applied.

Definition at line 142 of file image.cpp.

5.2.3.2 balance_intensity_normal_2D()

```
void image::balance_intensity_normal_2D (
            Coordinates beg,
            Coordinates end,
            unsigned int param_intensity )
```

This method balances the intensity of the current instance of image throught the exponential function. This time the method will be applied considering the elliptic shape of the fingerpint. The user has to provide two coordinates corresponding to the two opposite edge of the rectangle of application. The third parameter is reguling the strentgh of the balancing function : the bigger is param_intensity the stronger is the function.

Parameters

beg	end param_intensity

Returns

Nothing but modifies the instance on which it is applied.

Definition at line 178 of file image.cpp.

5.2.3.3 balance_intensity_quadratic()

```
void image::balance_intensity_quadratic (
            Coordinates beg,
            Coordinates end,
            unsigned int param_intensity )
```

This method balances the intensity of the current instance of image throught the quadratic function. The user has to provide two coordinates corresponding to the two opposite edge of the rectangle of application. The third parameter is reguling the strentgh of the balancing function : the bigger is param_intensity the stronger is the function.

Parameters

beg	end param_intensity

Returns

Nothing but modifies the instance on which it is applied.

Definition at line 160 of file image.cpp.

5.2.3.4 barycenter()

```
Coordinates image::barycenter ( )
```

We created this method in order to find the barycenter of an image, because we considered that the barycenter of an image is equals to the center of pressure.

Returns

it returns an instance of type “Coordinates”, its attributes are the location of barycenter in the image.

Definition at line 202 of file image.cpp.

5.2.3.5 bigger_image()

```
void image::bigger_image ( )
```

In a way to save information, and ovoid any loss of pixels intensity when making a rotation, we'll start computing rotations with bigger images, then we call the method “smaller_image” to come back to the normal dimensions of the image. If im is an instance of this class, we can do like this to make bigger this image: im.bigger_image()

Parameters

Nothing	

Definition at line 509 of file image.cpp.

5.2.3.6 black_square()

```
void image::black_square (
            Coordinates beg,
            Coordinates end )
```

We create black squares at a given position. Each position is of type “Coordinates”. In order to draw a black square in the image im such that the beginning of the square is the point (30,30) and its end is (50,65), we can do: im.black_square(Coordinates(30,30),Coordinates(50,65)).

Parameters

Coordinates_beginning	Coordinates_end

Returns

It creates a black square on the image.

Definition at line 94 of file image.cpp.

5.2.3.7 contour()

```
vector< Coordinates > image::contour ( )
```

This method computes the coordinates of the contour of the finger inside the image. It uses both classes "vector" and "coordinates".

Parameters

Nothing	

Returns

The result send by this method is a vector containing the coordinates of each pixels representing the contour of the finger.

Definition at line 226 of file image.cpp.

5.2.3.8 create_small_image()

```
image image::create_small_image (
            Coordinates point,
            int image_size )
```

Auxiliary function for the detect_warp_small_square method. It helps creating the small piece of image to compare with the test square. It will create a square image centered on the coordinate point an of size image_size.

Parameters

point	image_size

Returns

Returns an image as described above.

Definition at line 1068 of file image.cpp.

5.2.3.9 detect_rot()

```
float image::detect_rot (
            image & image_rotation,
            Coordinates rotation_center,
            int nb_test )
```

This function detects if the current instance of image is the rotation of the image contained in the image_rotation parameter. We also have to specify where we want the rotation to happen with the rotation_center parameter. As it is impossible in general to find the exact rotation of a picture and another (if the image is a rotation of exactly sqrt(2) then it would take an infinite time to compute the exact rotation value) we have to specify the number of step we want to do with the pameter nb_test : the more step and the more accurate is the result, but the slower is the algorithm...

Parameters

image_rotation	rotation_center nb_test

Returns

Returns a float representing the rotation angle in radius.

Definition at line 863 of file image.cpp.

5.2.3.10 detect_trans()

```
vector< float > image::detect_trans (
            image & image_trans,
            int nb_testx,
            int nb_testy )
```

This function detect if the current instance of image is a translation of the one contained in the image_trans parameter. As for the detect_rot function, we have to define the number of operations we want to perform along the Ox and Oy axis. This can be done by inputing it in the parameters nb_testx and nb_testy.

Parameters

image_trans	nb_testx nb_testy

Returns

Returns a vector of 3 floats, the two first are respectively the translation answer along the Ox and Oy axis and the last one the error value.

Definition at line 885 of file image.cpp.

5.2.3.11 detect_trans_along_x_and_y_with_first_loss_function()

```
void image::detect_trans_along_x_and_y_with_first_loss_function (
            image & image_compare )
```

This function deal with the case where the wrap function is a translation along the X-axis and Y-axis such that there are two translation parameters px and py to estimate. Here we use the first loss function. We didn't call the one defined independently named “first_loss_function”. The execution of this function may take a lot of time because it's a greedy startegy. If im1 and im2 are two instances representing the same image. In order to test this function, we can do like this: im2.translation(10.,11.) then im1.detect_trans_along_x_and_y_with_first_loss_function(im2) and the result printed on the screen will be (10.,11.) ! Here you can test also negative values!

Parameters

image	

Returns

Nothing but it prints on the screen the estimated parameter px and py.

Definition at line 698 of file image.cpp.

5.2.3.12 detect_trans_along_x_and_y_with_second_loss_function()

```
void image::detect_trans_along_x_and_y_with_second_loss_function (
            image & image_compare )
```

The execution of this function may take a lot of time because it's a greedy strategy. This function is like the one called “detect_trans_along_x_and_y_with_first_loss_function”, because it deals with the case where the wrap function is a translation along the X-axis and Y-axis such that there are two translation parameters px and py to estimate. Its advantage is that it uses (implicitly) the second loss function. So, we didn't call the method named “second_↩ loss_function”. Finally, it's worth noting that in this method we maximize the second loss function and we do not minimize it like we did in the method “detect_trans_along_x_and_y_with_first_loss_function”. If im1 and im2 are two instances representing the same image, in order to test this function, we can do like this: im2.translation(10.,11.) then im1.detect_trans_along_x_and_y_with_second_loss_function(im2) and the result printed on the screen will be (10.,11.) ! Here you can test also negative values!

Parameters

image	

Returns

Nothing but it prints on the screen the estimated parameter px and py.

Definition at line 758 of file image.cpp.

5.2.3.13 detect_trans_along_x_with_first_loss_function()

```
int image::detect_trans_along_x_with_first_loss_function (
            image & image_compare )
```

This function deal with the case where the wrap function is a translation along the X-axis such that there is only a single translation parameter px to estimate. Here we use the first lost function. We didn't call the one defined independently named “first_loss_function”. it can take time to be executed. If im1 and im2 are two instances representing the same image, in order to test this function, we can do like this: im2.translation(4.,0.) then im1.↩ detect_trans_along_x_with_first_loss_function(im2) and the result returned by this method will be 4! Here you can test also negative values!

Parameters

image	

Returns

An integer which represent the estimated parameter px.

Definition at line 661 of file image.cpp.

5.2.3.14 detect_warp()

```
vector< float > image::detect_warp (
            image & image_compare,
            int nb_test_rot,
            int nb_testx,
            int nb_testy )
```

This function detects if the current instance of image is a warp of the image contained in the parameter image↩ _compare using the first naive method (trying out every rotation/translation and computing the error on the whole image). We have to profide the number of test we want to perform on the rotation, on the transalations along the Ox and Oy axises respectively in the nb_test_rot, nb_testx and nb_testy parameters. Once again, the bigger are these numbers and the slower is the algorithm but the more accurate is the result.

Parameters

image_compare	nb_test_rot nb_testx nb_testy

Returns

Returns a vector of float of size 3, the two first parameter are the two translation answers (along Ox and Oy) and the last is the rotation answer.

Definition at line 958 of file image.cpp.

5.2.3.15 detect_warp_small_square()

```
vector< float > image::detect_warp_small_square (
            image & image_compare,
            int nb_test_rot,
            int square_size )
```

This function detects if the current instance of image is a warp of the image contained in the parameter image←_compare using optimized small square method (we here try to match only a small square of the image). In this method, we just have to give the number of test we want to perform on the rotation throught the nb_test_rot parameter. We no longer need the number of test for the translation since the efficiency of the algorithm allows us to test every integer values for the translation. We can also set the size of the test square with square_size parameter (in number of pixels).

Parameters

image_compare	nb_test_rot square_size

Returns

Returns a vector of float of size 3, the two first parameter are the two translation answers (along Ox and Oy) and the last is the rotation answer.

Definition at line 981 of file image.cpp.

5.2.3.16 ellipse_parameters()

```
vector< float > image::ellipse_parameters ( )
```

This method estimates the two ellipse parameters a and b using the first naive and unoptimized method (taking the min and max distance from the center of the ellipse).

Parameters

None	

Returns

Returns a vector of float of size 2, the first element is the estimated value of a and the second the estimated value of b.

Definition at line 259 of file image.cpp.

5.2.3.17 ellipse_parameters_gradient()

```
vector< float > image::ellipse_parameters_gradient (
            float epsilon )
```

This method estimates the two ellipse parameters a and b using the gradient descent method. We need here to provide a step size for the method in the epsilon parameter. The smaller is epsilon and the preciser is the algorithm, but it also makes it slower.

Parameters

epsilon	

Returns

Returns a vector of float of size 2, the first element is the estimated value of a and the second the estimated value of b.

Definition at line 276 of file image.cpp.

5.2.3.18 first_loss_function()

```
int image::first_loss_function (
            image & im2 )
```

We define here the first loss function which is the sum of squared errors between the pixels of two images. The sum is taken over all pixels of images.

Parameters

image	

Returns

An integer which represent the sum taken over all pixels of both images.

Definition at line 1019 of file image.cpp.

5.2.3.19 height_get()

```
unsigned int image::height_get ( )
```

Here we define another getter to know the height of an image.

Returns

The height of the image

Definition at line 29 of file image.cpp.

5.2.3.20 matrix_get()

```
Mat image::matrix_get ( )
```

Another getter to know the values of attribute of type “Mat”.

Returns

The matrix containing the values of the pixels.

Definition at line 33 of file image.cpp.

5.2.3.21 print_barycenter()

```
void image::print_barycenter ( )
```

We used the method “barycenter” to know the coordinates of the barycenter, then we applied the function "black↩_square" to print it.

Returns

It modifies the attribute “matrix” of the image.

Definition at line 221 of file image.cpp.

5.2.3.22 print_contour()

```
void image::print_contour ( )
```

This method uses the functions “contour " and “black_square” for the purpose of printing the contour of the finger in the image. In order to print the contour of the finger in an image represented by the instance im, we can do as follow: im.print_contour(). It modifies the attribute matrix of the image. To see the modifications, all what we have to do is to use the overloaded operator “<<”.

Parameters

Nothing	

Definition at line 249 of file image.cpp.

5.2.3.23 print_ellipse()

```
void image::print_ellipse ( )
```

Computes and prints the best matching ellipse using the gradient descent algorithm. It uses a generic step for the gradient method. This function doesn't modify the current instance of image.

Parameters

None	

Returns

Nothing but prints the resulting image on screen.

Definition at line 328 of file image.cpp.

5.2.3.24 return_diagonal_symetry()

```
void image::return_diagonal_symetry ( )
```

The diagonal symmetry which is a combination of the symmetry along the X axis and the one along the Y axis. So this function uses the methods "return_symetry_x" and "return_symetry_y". We can print the transformed image in the screen thanks to the operator “<<”.

Parameters

Nothing	

Returns

It modifies the attributes "matrix" of the original image.

Definition at line 137 of file image.cpp.

5.2.3.25 return_max_intensity()

```
unsigned int image::return_max_intensity ( )
```

This method calculates the maximum intensity value. It uses the method "return_pixel_value" to browse all the elements of attribute "matrix".

Parameters

Nothing	

Returns

the maximum intensity of an image

Definition at line 46 of file image.cpp.

5.2.3.26 return_min_intensity()

```
unsigned int image::return_min_intensity ( )
```

This method calculates the minimum intensity of an image.

Parameters

Nothing	

Returns

the minimum intensity of an image

Definition at line 58 of file image.cpp.

5.2.3.27 return_pixel_value()

```
unsigned int image::return_pixel_value (
            unsigned int x,
            unsigned int y )
```

A getter to know the intensity value of the pixel at the coordinate (x, y). If im is an instance of this class. To know the pixel intensity at the position (50,50), we can do: im.return_pixel_value(50,50).

Parameters

unsigned_int	unsigned_int

Returns

The intensity value Which is of type unsigned int.

Definition at line 37 of file image.cpp.

5.2.3.28 return_symetry_x()

```
void image::return_symetry_x ( )
```

To perform the symmetry of the image along the X-axis.

Parameters

Nothing	

Returns

It modifies the attributes "matrix" of the original image.

Definition at line 126 of file image.cpp.

5.2.3.29 return_symetry_y()

```
void image::return_symetry_y ( )
```

Performing the symmetry of an image along the Y-axis.

Parameters

Nothing	

Returns

It modifies the attributes "matrix" of the original image.

Definition at line 115 of file image.cpp.

5.2.3.30 rotation()

```
void image::rotation (
            Coordinates rotation_point,
            float angle )
```

Here we do a transformation of an image using the rotation around a point and with an angle without any interpolation, using the mathematical formula we gave in the report and two methods of the class coordiantes. To rotate an image -represented by the instance im- around its barycenter and with an angle PI/4, we can do as follow: im.rotation(im1.barycenter(),PI/4).

Parameters

Rotation_point	Angle

Returns

Nothing, but it modifies the attributes “matrix”.

Definition at line 348 of file image.cpp.

5.2.3.31 rotation_interpolation_bicubis()

```
void image::rotation_interpolation_bicubis (
            Coordinates rotation_point,
            float angle )
```

To do a better rotation of an image around a point and with a specific angle, we implemented this method where we interpolate using the bicubic interpolation thanks to the independent function “interpolation_bicubique”. Also, inside this method we normalize all the values resulting from the execution of the independent function. To rotate an image -represented by the instance im- around its barycenter and with an angle PI/4, we can do as follow: im.rotation_interpolation_bicubis(im.barycenter(),PI/4)

Parameters

Rotation_point	Angle

Returns

Nothing, but it modifies the attributes “matrix”.

Definition at line 398 of file image.cpp.

5.2.3.32 rotation_interpolation_bilinear()

```
void image::rotation_interpolation_bilinear (
            Coordinates rotation_point,
            float angle )
```

Thanks to this function we can effectuate a rotation of the image around a point and with a specific angle, both given as parameters. Here we interpolate using the bilinear interpolation (In the report we detailed the theoretical calculation) which is quite good, but not as good as the interpolation with the bicubic method. To rotate an image -represented by the instance im- around its barycenter and with an angle PI/4, we can do as follow: im.rotation_↩
interpolation_bilinear(im.barycenter(),PI/4)

Parameters

Rotation_point	Angle

Returns

It modifies the attribute "matrix", we can use the operator "<<" to show the modified image.

Definition at line 372 of file image.cpp.

5.2.3.33 rotation_warping()

```
void image::rotation_warping (
            Coordinates rotation_center,
            float radius,
            float rotation_strength )
```

Performs the rotation warping of an image, which is a rotation decreasing of strength the further away you are from the center. This center is defined by the rotation_center parameter. A negative rotation_strength is perfoming the rotation in the other way. This method uses a bilinear interpolation to do so.

Parameters

rotation_center	radius rotation_strength

Returns

Nothing, but it modifies the attributes "matrix".

Definition at line 545 of file image.cpp.

5.2.3.34 save_image()

```
void image::save_image (
            string saving_name )
```

In order to save an image after some modification we define the following method. We save them with the ".png" extension.

Parameters

saving_name	

Returns

It saves the image in the directory “bin”.

Definition at line 41 of file image.cpp.

5.2.3.35 second_loss_function()

```
float image::second_loss_function (
            image & im2 )
```

We define here the second loss function where we calculate the mean of the pixel's intensity of both images. We browse all the pixels of both images.

Parameters

image	

Returns

An integer which represent the sum taken over all pixels of both images.

Definition at line 1039 of file image.cpp.

5.2.3.36 smaller_image()

```
void image::smaller_image ( )
```

After calling the method "bigger_image" we need to return an image with the exact same dimension of the original one. That’s why we must implement the inverse method which is “smaller_image”.

Parameters

Nothing	

Returns

Nothing, but it modifies the attributes “matrix”.

Definition at line 532 of file image.cpp.

5.2.3.37 translation()

```
void image::translation (
            float alpha,
            float beta )
```

The translation along the X-axis and the Y-axis using the bilinear interpolation. The algorithm we use here to interpolate is like the one in the method “rotation_interpolation_bilinear”. In order to translate an image -represented by the instance im- with 11 pixels following the X-axis and 30 pixels following the Y-axis, we can do as follow: im.↩ translation(11.,30.).

Parameters

Float_alpha	Float_beta

Returns

It modifies the attribute “matrix”.

Definition at line 486 of file image.cpp.

5.2.3.38 translation_warping()

```
void image::translation_warping (
            Coordinates center,
            Coordinates beg,
            Coordinates end,
            float strength_x,
            float strength_y )
```

This function allows to apply the functions translation_warping_x and translation_warping_y one after this other.

Parameters

center	beg end strength_x strength_y

Returns

Nothing, but it modifies the attributes “matrix”.

Definition at line 653 of file image.cpp.

5.2.3.39 translation_warping_x()

```
void image::translation_warping_x (
            Coordinates center,
```

```
Coordinates beg,
Coordinates end,
float strength_x )
```

Performs the translation warping according the Ox axis, which is a stretching of the image. You can choose either to compress or to zoom on the image by changing the positivity of the strength of the strength_x parameter. The parameter center defines the center of the stretching and the parameters beg and end the zone of application of the function.

Parameters

center	beg end strength_x

Returns

Nothing, but it modifies the attributes “matrix”.

Definition at line 579 of file image.cpp.

5.2.3.40 translation_warping_y()

```
void image::translation_warping_y (
            Coordinates center,
            Coordinates beg,
            Coordinates end,
            float strength_y )
```

Similar to the translation_warping_x funtion but applied to the Oy axis.

Parameters

center	beg end strength_y

Returns

Nothing, but it modifies the attributes “matrix”.

Definition at line 616 of file image.cpp.

5.2.3.41 white_square()

```
void image::white_square (
            Coordinates beg,
            Coordinates end )
```

We create white squares at a given position. Each position is of type “Coordinates”. It raises an error if there no logical values of both parameters. In order to draw a white square in the image im such that the beginning of the square is the point (30,30) and its end is (50,65), we can do: im.white_square(Coordinates(30,30),Coordinates(50,65)).

Parameters

Coordinates_beginning	Coordinates_end

Returns

It creates a white square on the image.

Definition at line 70 of file image.cpp.

5.2.3.42 width_get()

```
unsigned int image::width_get ( )
```

Here we define a getter to know the width of an image.

Returns

The width of the image

Definition at line 25 of file image.cpp.

5.2.4 Friends And Related Function Documentation

5.2.4.1 operator<<

```
ostream& operator<< (
             ostream & o,
             const image & im )   [friend]
```

In order to overload the operator "<<" we defined this function, which must be declared as a friend function.

Definition at line 856 of file image.cpp.

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

- include/image.h
- src/image.cpp

Chapter 6

File Documentation

6.1 include/coordinates.h File Reference

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

Classes

- class Coordinates

  The class "Coordinates" contains one constructor and most of the methods we used to define many functions in the class “image”. It represents the position of one pixel in the attribute matrix accordingly to the basis we choose to do this project.

Namespaces

- std

Macros

- #define PI 3.14159265

6.1.1 Macro Definition Documentation

6.1.1.1 PI

```
#define PI 3.14159265
```

Definition at line 15 of file coordinates.h.

6.2 include/image.h File Reference

This file contains the C++ functions declaration for the class "image" we used to do this project, and the declaration for one independent function.

```
#include <opencv2/core/core.hpp>
#include <opencv2/highgui/highgui.hpp>
#include <iostream>
#include <opencv2/opencv.hpp>
#include "coordinates.h"
#include <algorithm>
#include <Eigen/Dense>
```

Classes

- class image

  The class image contains three constructors and most of the functions we defined to respond to many questions.

Namespaces

- Eigen
- std
- cv

Functions

- int interpolation_bicubique (Matrix4d, Coordinates)

  In order to interpolate using the bicubic interpolation, we use this independent function. We use here some classes of the library “Eigen”. We detailed all the theoretical calculation in the report.

6.2.1 Detailed Description

This file contains the C++ functions declaration for the class "image" we used to do this project, and the declaration for one independent function.

Author

Nicolas Cavrel - Yassine Fakihani - Ben Sarfati

6.2.2 Function Documentation

6.2.2.1 interpolation_bicubique()

```
int interpolation_bicubique (
            Matrix4d ,
            Coordinates  )
```

In order to interpolate using the bicubic interpolation, we use this independent function. We use here some classes of the library “Eigen”. We detailed all the theoretical calculation in the report.

Parameters

Matrix4d←↩ _M	Coordinates←↩ _P

Returns

The interpolated point.

Definition at line 435 of file image.cpp.

6.3 src/coordinates.cpp File Reference

```
#include "coordinates.h"
```

Functions

- ostream & operator<< (ostream &o, const Coordinates &p)

6.3.1 Function Documentation

6.3.1.1 operator<<()

```
ostream& operator<< (
            ostream & o,
            const Coordinates & p )
```

Definition at line 40 of file coordinates.cpp.

6.4 src/image.cpp File Reference

```
#include "image.h"
#include <limits>
```

Functions

- int interpolation_bicubique (Matrix4d M, Coordinates P)

  In order to interpolate using the bicubic interpolation, we use this independent function. We use here some classes of the library "Eigen". We detailed all the theoretical calculation in the report.
- ostream & operator<< (ostream &o, const image &im)

6.4.1 Function Documentation

6.4.1.1 interpolation_bicubique()

```
int interpolation_bicubique (
            Matrix4d ,
            Coordinates  )
```

In order to interpolate using the bicubic interpolation, we use this independent function. We use here some classes of the library “Eigen”. We detailed all the theoretical calculation in the report.

Parameters

Matrix4d↩ _M	Coordinates↩ _P

Returns

The interpolated point.

Definition at line 435 of file image.cpp.

6.4.1.2 operator<<()

```
ostream& operator<< (
            ostream & o,
            const image & im )
```

Definition at line 856 of file image.cpp.

6.5 src/main.cpp File Reference

```
#include "image.h"
```

Functions

- int main (int argc, char **argv)

6.5.1 Function Documentation

6.5.1.1 main()

```
int main (
            int argc,
            char ** argv )
```

Definition at line 3 of file main.cpp.

6.6 test/tests.cpp File Reference

```
#include <gtest/gtest.h>
#include "coordinates.h"
```

Functions

- TEST (TestGet, XValue)
- int main (int argc, char **argv)

6.6.1 Function Documentation

6.6.1.1 main()

```
int main (
            int argc,
            char ** argv )
```

Definition at line 12 of file tests.cpp.

6.6.1.2 TEST()

```
TEST (
            TestGet ,
            XValue  )
```

Definition at line 6 of file tests.cpp.

Index