LabRoboticsProject

Generated by Doxygen 1.8.14

Contents

Chapter 1

Class Index

1.1 Class List

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

Chapter 2

File Index

2.1 File List

Here is a list of all files with brief descriptions:

Chapter 3

Class Documentation

3.1 Angle Class Reference

This class allows to save and handle angles. It supports DEG and RAD, operations such as addition and subtraction with operators overloading, conversion from RAD to DEG and viceversa and normalization of the angle.

`#include <maths.hh>`

Public Types

- enum ANGLE_TYPE { DEG, RAD, INVALID }

Public Member Functions

- Angle ()

  A void constructor to create an angle.
- Angle (double _th, ANGLE_TYPE _type)

  This constructor takes the angle value and the type of angle and stores them. It also normalize the angle in case is above 2pi (360°) or below 0.
- double get () const

  Returns the dimension of the angle.
- ANGLE_TYPE getType () const

  Returns the type of the angle.
- string getTypeName () const
- template<class T >
  void set (const T _th)

  Set the value of the angle.
- void setType (ANGLE_TYPE _type)

  Set the type of the angle.
- double degToRad ()

  Convert and store the angle from DEG to RAD.
- double radToDeg ()

  Converts and stores the angle from RAD to DEG.
- double toRad () const

  Converts but does not store the value of the angle from DEG to RAD.

- double toDeg () const

  Converts but does not store the value of the angle from RAD to DEG.
- void normalize ()

  Normalize the angle, that is to set it in $[0, 2\pi]$ *or* $[0, 360]$.
- Angle add (const Angle phi)

  Sums and angle to this one. In the process a new angle is created so `normalize()` *is also called.*
- Angle sub (const Angle phi)

  Subtracts and angle to this one. In the process a new angle is created so `normalize()` *is also called.*
- Angle copy (const Angle phi)

  Copies an angle to this one. In the process a new angle is created so `normalize()` *is also called.*
- Angle operator+ (const Angle phi)
- Angle operator- (const Angle phi)
- Angle operator= (const Angle phi)
- Angle & operator+= (const Angle phi)
- Angle & operator-= (const Angle phi)

Friends

- ostream & operator<< (ostream &out, const Angle &data)

3.1.1 Detailed Description

This class allows to save and handle angles. It supports DEG and RAD, operations such as addition and subtraction with operators overloading, conversion from RAD to DEG and viceversa and normalization of the angle.

3.1.2 Member Enumeration Documentation

3.1.2.1 ANGLE_TYPE

`enum Angle::ANGLE_TYPE`

Enumerator

DEG	
RAD	
INVALID	

3.1.3 Constructor & Destructor Documentation

3.1.3.1 Angle() [1/2]

```
Angle::Angle ( )  [inline]
```

A void constructor to create an angle.

3.1.3.2 Angle() [2/2]

```
Angle::Angle (
            double _th,
            ANGLE_TYPE _type )  [inline]
```

This constructor takes the angle value and the type of angle and stores them. It also normalize the angle in case is above 2pi (360°) or below 0.

Parameters

in	*_th*	The dimension of the angle.
in	*_type*	The type of the angle.

3.1.4 Member Function Documentation

3.1.4.1 add()

```
Angle Angle::add (
            const Angle phi )  [inline]
```

Sums and angle to this one. In the process a new angle is created so `normalize()` is also called.

Parameters

in	*phi*	The angle to be summed.

Returns

The angle summed.

3.1.4.2 copy()

```
Angle Angle::copy (
            const Angle phi )  [inline]
```

Copies an angle to this one. In the process a new angle is created so `normalize()` is also called.

Parameters

in	*phi*	The angle to be copied.

Returns

The new angle.

3.1.4.3 degToRad()

```
double Angle::degToRad ( )  [inline]
```

Convert and store the angle from DEG to RAD.

Returns

The value of the angle.

3.1.4.4 get()

```
double Angle::get ( ) const  [inline]
```

Returns the dimension of the angle.

3.1.4.5 getType()

```
ANGLE_TYPE Angle::getType ( ) const  [inline]
```

Returns the type of the angle.

3.1.4.6 getTypeName()

```
string Angle::getTypeName ( ) const  [inline]
```

<Returns a string that tells the type of angle.

3.1.4.7 normalize()

```
void Angle::normalize ( )  [inline]
```

Normalize the angle, that is to set it in $[0, 2\pi]$ or $[0, 360]$.

3.1.4.8 operator+()

```
Angle Angle::operator+ (
            const Angle phi )  [inline]
```

This function overload the operator +. It simply calls the `add()` function.

Parameters

in	*phi*	The angle to be summed.

Returns

The angle summed.

3.1.4.9 operator+=()

```
Angle& Angle::operator+= (
            const Angle phi )  [inline]
```

This function overload the operator +. It simply calls the `add()` function and then assign the result to this.

Parameters

in	*phi*	The angle to be summed.

Returns

`this`.

3.1.4.10 operator-()

```
Angle Angle::operator- (
            const Angle phi )  [inline]
```

This function overload the operator -. It simply calls the `sub()` function.

Parameters

in	*phi*	The angle to be subtracted.

Returns

The angle subtracted.

3.1.4.11 operator-=()

```
Angle& Angle::operator-= (
            const Angle phi )  [inline]
```

This function overload the operator -. It simply calls the `sub()` function and then assign the result to this.

Parameters

in	*phi*	The angle to be subtracted.

Returns

this.

3.1.4.12 operator=()

```
Angle Angle::operator= (
            const Angle phi )  [inline]
```

This function overload the operator -. It simply calls the `sub()` function.

Parameters

in	*phi*	The angle to be copied.

Returns

The new angle.

3.1.4.13 radToDeg()

```
double Angle::radToDeg ( )  [inline]
```

Converts and stores the angle from RAD to DEG.

Returns

The value of the angle.

3.1.4.14 set()

```
template<class T >
void Angle::set (
            const T _th )  [inline]
```

Set the value of the angle.

Template Parameters

T	The programming type for the value to be stored. It's then cast to `double`.

Parameters

in	↩ _↩ *th*	The dimension of the angle to be stored.

3.1.4.15 setType()

```
void Angle::setType (
            ANGLE_TYPE _type )  [inline]
```

Set the type of the angle.

Parameters

in	↩ _↩ *th*	The type of the angle to be stored.

3.1.4.16 sub()

```
Angle Angle::sub (
            const Angle phi )  [inline]
```

Subtracts and angle to this one. In the process a new angle is created so `normalize()` is also called.

Parameters

in	*phi*	The angle to be subtracted.

Returns

The angle subtracted.

3.1.4.17 toDeg()

```
double Angle::toDeg ( ) const  [inline]
```

Converts but does not store the value of the angle from RAD to DEG.

Returns

The value of the angle

3.1.4.18 toRad()

```
double Angle::toRad ( ) const  [inline]
```

Converts but does not store the value of the angle from DEG to RAD.

Returns

The value of the angle

3.1.5 Friends And Related Function Documentation

3.1.5.1 operator<<

```
ostream& operator<< (
            ostream & out,
            const Angle & data )  [friend]
```

This function overload the << operator so to print with `std::cout` the most essential info, that is the dimension and the type of angle.

Parameters

in	*out*	The out stream.
in	*data*	The angle to print.

Returns

An output stream to be printed.

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

- src/maths.hh

3.2 Configuration2< T1 > Class Template Reference

This class stores a configuration, that is a point and an angle.

```
#include <maths.hh>
```

Public Member Functions

- Configuration2 ()

  Default constructor that use as point (0,0) and as angle 0 RAD.
- Configuration2 (const T1 _x, const T1 _y, const Angle _th)

  Default constructor that takes the coordinates, the angle, and stores them.
- Configuration2 (const Point2< T1 > P, const Angle _th)

  Default constructor that takes the point, the angle, and stores them.
- T1 x () const
- T1 y () const
- Angle angle () const
- int x (const T1 _x)

  This function stores a new value for the abscissa.
- int y (const T1 _y)

  This function stores a new value for the ordinate.
- void angle (const Angle _th)

  This function stores a new value for the angle.
- template<class T2 >
  int offset (const T2 _offset, const Angle phi, const Angle _th)

  This function compute the offset of the point given a vector, that is the lenght of the vector and its angle. The angle must be an `Angle` variable. It takes also another `Angle` to change the `Angle` in the configuration.
- int offset (Configuration2< T1 > p)

  This function compute the offset of the point given another `Configuration2`.
- int offset (Point2< T1 > p, const Angle _th=Angle())

  This function compute the offset of the point given a `Point2` containing the offsets for the abscissa and the ordiante and an `Angle` to change the `Angle` in the configuration.
- int offset_x (const T1 _offset)

  Function to add an offset to the abscissa.
- int offset_y (const Angle _offset)

  Function to add an offset to the ordinate.
- void offset_angle (const Angle _th)

  Function to add an offset to the angle.
- template<class T2 >
  Tuple< double > distance (Configuration2< T2 > B, DISTANCE_TYPE dist_type=EUCLIDEAN)

  Wrapper to compute different distances. T2 The type of the elements in the second `Configuration2`.
- template<class T2 >
  Tuple< double > EuDistance (Configuration2< T2 > B)

  Function that compute the Euclidean Distance between two configurations. T2 The type of the elements in the second `Configuration2`.
- template<class T2 >
  Tuple< double > MaDistance (Configuration2< T2 > B)

  Function that compute the Manhattan Distance between two configurations. T2 The type of the elements in the second `Configuration2`.

Friends

- ostream & operator<< (ostream &out, const Configuration2 &data)

  Overload of operator << to output the content of a `Configuration2`.

3.2.1 Detailed Description

template<class T1>
class Configuration2< T1 >

This class stores a configuration, that is a point and an angle.

Template Parameters

T1	The type of the coordinates.

3.2.2 Constructor & Destructor Documentation

3.2.2.1 Configuration2() [1/3]

```
template<class T1>
Configuration2< T1 >::Configuration2 ( )  [inline]
```

Default constructor that use as point (0,0) and as angle 0 RAD.

3.2.2.2 Configuration2() [2/3]

```
template<class T1>
Configuration2< T1 >::Configuration2 (
            const T1 _x,
            const T1 _y,
            const Angle _th )  [inline]
```

Default constructor that takes the coordinates, the angle, and stores them.

Parameters

in	$\hookleftarrow$ _$\hookleftarrow$ *x*	The abscissa coordinate.
in	$\hookleftarrow$ _$\hookleftarrow$ *y*	The ordinate coordinate.
in	$\hookleftarrow$ _$\hookleftarrow$ *th*	The angle.

3.2.2.3 Configuration2() [3/3]

```
template<class T1>
Configuration2< T1 >::Configuration2 (
            const Point2< T1 > P,
            const Angle _th )  [inline]
```

Default constructor that takes the point, the angle, and stores them.

Parameters

in	*P*	The coordinates.
in	↩ _↩ *th*	The angle.

3.2.3 Member Function Documentation

3.2.3.1 angle() [1/2]

```
template<class T1>
Angle Configuration2< T1 >::angle ( ) const  [inline]
```

Returns

The angle.

3.2.3.2 angle() [2/2]

```
template<class T1>
void Configuration2< T1 >::angle (
            const Angle _th )  [inline]
```

This function stores a new value for the angle.

Parameters

in	↩ _↩ *th*	The value to be stored.

Returns

1 if everything went ok, 0 otherwise.

3.2.3.3 distance()

```
template<class T1>
template<class T2 >
```

```
Tuple<double> Configuration2< T1 >::distance (
            Configuration2< T2 > B,
            DISTANCE_TYPE dist_type = EUCLIDEAN )  [inline]
```

Wrapper to compute different distances. T2 The type of the elements in the second Configuration2.

Parameters

in	*B*	The second Configuration2 to use for computing the distance.
in	*dist*	The type of distance to be computed.

Returns

The distance between the two configurations.

3.2.3.4 EuDistance()

```
template<class T1>
template<class T2 >
Tuple<double> Configuration2< T1 >::EuDistance (
            Configuration2< T2 > B )  [inline]
```

Function that compute the Euclidean Distance between two configurations. T2 The type of the elements in the second Configuration2.

Parameters

in	*B*	the second Configuration2 to use for computing the distance.

Returns

The Euclidean distance between the two configurations.

3.2.3.5 MaDistance()

```
template<class T1>
template<class T2 >
Tuple<double> Configuration2< T1 >::MaDistance (
            Configuration2< T2 > B )  [inline]
```

Function that compute the Manhattan Distance between two configurations. T2 The type of the elements in the second Configuration2.

Parameters

in	*B*	the second Configuration2 to use for computing the distance.

Returns

The Manhattan distance between the two configurations.

3.2.3.6 offset() [1/3]

```
template<class T1>
template<class T2 >
int Configuration2< T1 >::offset (
            const T2 _offset,
            const Angle phi,
            const Angle _th )  [inline]
```

This function compute the offset of the point given a vector, that is the lenght of the vector and its angle. The angle must be an `Angle` variable. It takes also another `Angle` to change the `Angle` in the configuration.

Template Parameters

3.2.3.7 offset() [2/3]

```
template<class T1>
int Configuration2< T1 >::offset (
            Configuration2< T1 > p )  [inline]
```

This function compute the offset of the point given another `Configuration2`.

Parameters

in	*p*	The configuration containing the offsets.

Returns

1 if everything went fine, 0 otherwise.

3.2.3.8 offset() [3/3]

```
template<class T1>
int Configuration2< T1 >::offset (
            Point2< T1 > p,
            const Angle _th = Angle() )  [inline]
```

This function compute the offset of the point given a `Point2` containing the offsets for the abscissa and the ordiante and an `Angle` to change the `Angle` in the configuration.

Parameters

in	*p*	The point containing the offsets.
in	*_th*	The offset for the Angle in the configuration. It's set to 0 as default so to easily change just the coordinates.

Returns

1 if everything went fine, 0 otherwise.

3.2.3.9 offset_angle()

```
template<class T1>
void Configuration2< T1 >::offset_angle (
            const Angle _th )  [inline]
```

Function to add an offset to the angle.

Parameters

in	*_offset*	The offset.

Returns

1 if everything went fine, 0 otherwise.

3.2.3.10 offset_x()

```
template<class T1>
int Configuration2< T1 >::offset_x (
            const T1 _offset )  [inline]
```

Function to add an offset to the abscissa.

Parameters

in	*_offset*	The offset.

Returns

1 if everything went fine, 0 otherwise.

3.2.3.11 offset_y()

```
template<class T1>
int Configuration2< T1 >::offset_y (
            const Angle _offset )  [inline]
```

Function to add an offset to the ordinate.

Parameters

in	*_offset*	The offset.

Returns

1 if everything went fine, 0 otherwise.

3.2.3.12 x() [1/2]

```
template<class T1>
T1 Configuration2< T1 >::x ( ) const  [inline]
```

Returns

The abscissa coordinate.

3.2.3.13 x() [2/2]

```
template<class T1>
int Configuration2< T1 >::x (
            const T1 _x )  [inline]
```

This function stores a new value for the abscissa.

Parameters

in	↩ *_x*	The value to be stored.

Returns

1 if everything went ok, 0 otherwise.

3.2.3.14 y() [1/2]

```
template<class T1>
T1 Configuration2< T1 >::y ( ) const  [inline]
```

Returns

The ordinate coordinate.

3.2.3.15 y() [2/2]

```
template<class T1>
int Configuration2< T1 >::y (
            const T1 _y )  [inline]
```

This function stores a new value for the ordinate.

Parameters

in	↩ _↩ *y*	The value to be stored.

Returns

1 if everything went ok, 0 otherwise.

3.2.4 Friends And Related Function Documentation

3.2.4.1 operator<<

```
template<class T1>
ostream& operator<< (
            ostream & out,
            const Configuration2< T1 > & data )  [friend]
```

Overload of operator << to output the content of a Configuration2.

Parameters

in	*out*	The output stream.
in	*data*	The Configuration2 to print.

Returns

An output stream to be printed.

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

- src/maths.hh

3.3 dubins Class Reference

```
#include <dubins.hh>
```

Public Member Functions

- dubins (double x0, double y0, double th0, double x1, double y1, double th1, double max=1.0)
- dubins (Point2 _P0, Point2 _P1, Angle _th1, Angle _th2, Kmax=1.0)
- dubins (Configuration2 _P0, Configuration2 _P1, Kmax=1.0)
- ~dubins ()

3.3.1 Constructor & Destructor Documentation

3.3.1.1 dubins() [1/3]

```
dubins::dubins (
            double x0,
            double y0,
            double th0,
            double x1,
            double y1,
            double th1,
            double max = 1.0 )
```

3.3.1.2 dubins() [2/3]

```
dubins::dubins (
            Point2 _P0,
            Point2 _P1,
            Angle _th1,
            Angle _th2,
            Kmax  = 1.0 )
```

3.3.1.3 dubins() [3/3]

```
dubins::dubins (
            Configuration2 _P0,
            Configuration2 _P1,
            Kmax  = 1.0 )
```

3.3.1.4 ~dubins()

```
dubins::~dubins ( )
```

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

- src/dubins.hh

3.4 Point2< T > Class Template Reference

Class that stores two value to construct a point in 2D. The value is saved in a Tuple.

```
#include <maths.hh>
```

Public Member Functions

- Point2 ()

  Default constructor to build an empty Tuple.
- Point2 (const T _x, const T _y)

  Constructor that taked to elements and builds a point.
- T x () const
- T y () const
- int x (const T _x)

  Set the abscissa value.
- int y (const T _y)

  Set the ordinate value.
- template<class T1 >
  int offset (const T1 _offset, const Angle th)

  This function compute the offset of the point given a vector, that is the lenght of the vector and its angle. The angle must be an `Angle` variable.
- int offset (const Point2< T > p)

  This function compute an offset given another point made of the abscissa offset and the ordinate offset.
- int offset (const Tuple< T > p)

  This function compute an offset given a `Tuple` made of the abscissa offset and the ordinate offset.
- int offset_x (const T _offset)

  This function compute an offset for the abscissa.
- int offset_y (const T _offset)

  This function compute an offset for the ordinate.
- template<class T1 >
  double distance (Point2< T1 > B, DISTANCE_TYPE dist=EUCLIDEAN)

Wrapper to compute different distances. T1 The type of the elements in the second Point2.

- template<class T1 >
  double MaDistance (Point2< T1 > B)

  Function that compute the Manhattan Distance between two points. T1 The type of the elements in the second Point2.

- template<class T1 >
  double EuDistance (Point2< T1 > B)

  Function that compute the Euclidean Distance between two points. T1 The type of the elements in the second Point2.

Friends

- ostream & operator<< (ostream &out, const Point2< T > &data)

  Overload of operator << to output the content of a Point2.

3.4.1 Detailed Description

template<class T>
class Point2< T >

Class that stores two value to construct a point in 2D. The value is saved in a Tuple.

Template Parameters

T	The type of the coordinates to be stored.

3.4.2 Constructor & Destructor Documentation

3.4.2.1 Point2() [1/2]

```
template<class T>
Point2< T >::Point2 ( )  [inline]
```

Default constructor to build an empty Tuple.

3.4.2.2 Point2() [2/2]

```
template<class T>
Point2< T >::Point2 (
            const T _x,
            const T _y )  [inline]
```

Constructor that taked to elements and builds a point.

Parameters

in	↩ _↩ *x*	The abscissa coordinate.
in	↩ _↩ *y*	The ordinate coordinate.

3.4.3 Member Function Documentation

3.4.3.1 distance()

```
template<class T>
template<class T1 >
double Point2< T >::distance (
            Point2< T1 > B,
            DISTANCE_TYPE dist = EUCLIDEAN )  [inline]
```

Wrapper to compute different distances. T1 The type of the elements in the second Point2.

Parameters

in	*B*	The second Point2 to use for computing the distance.
in	*dist*	The type of distance to be computed.

Returns

The distance between the two points.

3.4.3.2 EuDistance()

```
template<class T>
template<class T1 >
double Point2< T >::EuDistance (
            Point2< T1 > B )  [inline]
```

Function that compute the Euclidean Distance between two points. T1 The type of the elements in the second Point2.

Parameters

in	*B*	the second Point2 to use for computing the distance.

Returns

The Euclidean distance between the two points.

3.4.3.3 MaDistance()

```
template<class T>
template<class T1 >
double Point2< T >::MaDistance (
            Point2< T1 > B )  [inline]
```

Function that compute the Manhattan Distance between two points. T1 The type of the elements in the second Point2.

Parameters

in	*B*	the second Point2 to use for computing the distance.

Returns

The Manhattan distance between the two points.

3.4.3.4 offset() [1/3]

```
template<class T>
template<class T1 >
int Point2< T >::offset (
            const T1 _offset,
            const Angle th )  [inline]
```

This function compute the offset of the point given a vector, that is the lenght of the vector and its angle. The angle must be an Angle variable.

Template Parameters

3.4.3.5 offset() [2/3]

```
template<class T>
int Point2< T >::offset (
            const Point2< T > p )  [inline]
```

This function compute an offset given another point made of the abscissa offset and the ordinate offset.

Parameters

in	*p*	The point with the offsets.

Returns

1 if everything went fine, 0 otherwise.

3.4.3.6 offset() [3/3]

```
template<class T>
int Point2< T >::offset (
            const Tuple< T > p )  [inline]
```

This function compute an offset given a `Tuple` made of the abscissa offset and the ordinate offset.

Parameters

in	*p*	The `Tuple` with the offsets. Its dimension must be 2.

Returns

1 if everything went fine, 0 otherwise.

3.4.3.7 offset_x()

```
template<class T>
int Point2< T >::offset_x (
            const T _offset )  [inline]
```

This function compute an offset for the abscissa.

Parameters

in	*_offset*	The offset.

Returns

1 if everything went fine, 0 otherwise.

3.4.3.8 offset_y()

```
template<class T>
int Point2< T >::offset_y (
            const T _offset )  [inline]
```

This function compute an offset for the ordinate.

Parameters

in	*_offset*	The offset.

Returns

1 if everything went fine, 0 otherwise.

3.4.3.9 x() [1/2]

```
template<class T>
T Point2< T >::x ( ) const  [inline]
```

Returns

The abscissa coordinate

3.4.3.10 x() [2/2]

```
template<class T>
int Point2< T >::x (
            const T _x )  [inline]
```

Set the abscissa value.

Parameters

in	↩ *_x*	The new abscissa value

Returns

1 if it was successful, 0 otherwise.

3.4.3.11 y() [1/2]

```
template<class T>
T Point2< T >::y ( ) const  [inline]
```

Returns

The ordinate coordinate

3.4.3.12 y() [2/2]

```
template<class T>
int Point2< T >::y (
            const T _y )  [inline]
```

Set the ordinate value.

Parameters

in	↩ _↩ *x*	The new ordinate value

Returns

1 if it was successful, 0 otherwise.

3.4.4 Friends And Related Function Documentation

3.4.4.1 operator<<

```
template<class T>
ostream& operator<< (
            ostream & out,
            const Point2< T > & data )  [friend]
```

Overload of operator << to output the content of a Point2.

Parameters

in	*out*	The output stream.
in	*data*	The Point2 to print.

Returns

An output stream to be printed.

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

- src/maths.hh

3.5 Settings Class Reference

`#include <calibration.hh>`

Public Types

- enum Pattern { NOT_EXISTING =0, CHESSBOARD =1 }
- enum InputType { INVALID =0, IMAGE_LIST =3 }

Public Member Functions

- Settings ()

  Constructor that sets `goodInput` to false.
- void write (FileStorage &fs) const

  Write serialization.
- void read (const FileNode &node)

  Read serialization.
- void validate ()

  This function validate the content of the file.
- Mat nextImage ()

  Get next image from list.

Static Public Member Functions

- static bool readStringList (const string &filename, vector< string > &l)

  Read from file a list of images.
- static bool isListOfImages (const string &filename)

  Check if the file from which is trying to retrive a list is a valid format (xml or yaml).

Public Attributes

- Size boardSize

  The size of the board -> Number of items by width and height.
- Pattern calibrationPattern = CHESSBOARD

  One of the Chessboard, circles, or asymmetric circle pattern.
- float squareSize

  The size of a square in your defined unit (point, millimeter,etc).
- int nrFrames

  The number of frames to use from the input for calibration.
- float aspectRatio

  The aspect ratio.
- int delay

  In case of a video input.
- bool writePoints

  Write detected feature points.
- bool writeExtrinsics

  Write extrinsic parameters.
- bool calibZeroTangentDist

  Assume zero tangential distortion.
- bool calibFixPrincipalPoint

  Fix the principal point at the center.
- bool flipVertical

  Flip the captured images around the horizontal axis.
- string outputFileName

  The name of the file where to write.
- bool showUndistorsed

  Show undistorted images after calibration.
- string input

  The input.
- bool useFisheye = false

  use fisheye camera model for calibration
- bool fixK1

  fix K1 distortion coefficient
- bool fixK2

  fix K2 distortion coefficient
- bool fixK3

  fix K3 distortion coefficient
- bool fixK4

  fix K4 distortion coefficient
- bool fixK5

  fix K5 distortion coefficient
- int cameraID
- vector< string > imageList
- size_t atImageList
- VideoCapture inputCapture
- InputType inputType = IMAGE_LIST
- bool goodInput
- int flag

3.5.1 Member Enumeration Documentation

3.5.1.1 InputType

```
enum Settings::InputType
```

Enumerator

INVALID	
IMAGE_LIST	

3.5.1.2 Pattern

```
enum Settings::Pattern
```

Enumerator

NOT_EXISTING	
CHESSBOARD	

3.5.2 Constructor & Destructor Documentation

3.5.2.1 Settings()

```
Settings::Settings ( )  [inline]
```

Constructor that sets `goodInput` to false.

3.5.3 Member Function Documentation

3.5.3.1 isListOfImages()

```
bool Settings::isListOfImages (
            const string & filename )  [static]
```

Check if the file from which is trying to retrive a list is a valid format (xml or yaml).

Parameters

in	*filename*	The name of the file to check for validity.

Returns

`false` is the file is not xml or yaml
`true` otherwise.

3.5.3.2 nextImage()

```
Mat Settings::nextImage ( )
```

Get next image from list.

Returns

A matrix containing the next image to consider.

3.5.3.3 read()

```
void Settings::read (
            const FileNode & node )
```

Read serialization.

This function read data from a file and stores each node in their corresponding variables.

Parameters

in	*node*	The node of the file to consider.

3.5.3.4 readStringList()

```
bool Settings::readStringList (
            const string & filename,
            vector< string > & l )  [static]
```

Read from file a list of images.

Parameters

`in`	*filename*	The name of the file from which to read.
`out`	*l*	A vector which will contain the names of the file from the list.

Returns

`false` if the file could not be opened or if the file doesn't contain a list
`true` otherwise.

3.5.3.5 validate()

```
void Settings::validate ( )
```

This function validate the content of the file.

Even though this function doesn't return anything nor has any parameters for output, it sets a variable of the `Settings` class, that is `googInput`, to `false` if some infos were wrong. `true` otherwise. The options it takes in consideration are the following:

- Size must be positive.
- Cells must be greater than 10^{-6}.
- The number of frames considered, that is images, must be greater than 0.
- Check for valid input, that is a valid list of images.
- Else a list of image is being used.
- Check the field pattern: if it doesn't correspond to a known one than it's invalid.

3.5.3.6 write()

```
void Settings::write (
            FileStorage & fs ) const
```

Write serialization.

This function write data to a file.

Parameters

in	*fs*	The filename where to write.

3.5.4 Member Data Documentation

3.5.4.1 aspectRatio

```
float Settings::aspectRatio
```

The aspect ratio.

3.5.4.2 atImageList

```
size_t Settings::atImageList
```

3.5.4.3 boardSize

```
Size Settings::boardSize
```

The size of the board -> Number of items by width and height.

3.5.4.4 calibFixPrincipalPoint

```
bool Settings::calibFixPrincipalPoint
```

Fix the principal point at the center.

3.5.4.5 calibrationPattern

```
Pattern Settings::calibrationPattern = CHESSBOARD
```

One of the Chessboard, circles, or asymmetric circle pattern.

3.5.4.6 calibZeroTangentDist

```
bool Settings::calibZeroTangentDist
```

Assume zero tangential distortion.

3.5.4.7 cameraID

```
int Settings::cameraID
```

3.5.4.8 delay

```
int Settings::delay
```

In case of a video input.

3.5.4.9 fixK1

```
bool Settings::fixK1
```

fix K1 distortion coefficient

3.5.4.10 fixK2

```
bool Settings::fixK2
```

fix K2 distortion coefficient

3.5.4.11 fixK3

```
bool Settings::fixK3
```

fix K3 distortion coefficient

3.5.4.12 fixK4

```
bool Settings::fixK4
```

fix K4 distortion coefficient

3.5.4.13 fixK5

```
bool Settings::fixK5
```

fix K5 distortion coefficient

3.5.4.14 flag

```
int Settings::flag
```

3.5.4.15 flipVertical

```
bool Settings::flipVertical
```

Flip the captured images around the horizontal axis.

3.5.4.16 goodInput

```
bool Settings::goodInput
```

3.5.4.17 imageList

```
vector<string> Settings::imageList
```

3.5.4.18 input

```
string Settings::input
```

The input.

3.5.4.19 inputCapture

```
VideoCapture Settings::inputCapture
```

3.5.4.20 inputType

```
InputType Settings::inputType = IMAGE_LIST
```

3.5.4.21 nrFrames

```
int Settings::nrFrames
```

The number of frames to use from the input for calibration.

3.5.4.22 outputFileName

```
string Settings::outputFileName
```

The name of the file where to write.

3.5.4.23 showUndistorsed

```
bool Settings::showUndistorsed
```

Show undistorted images after calibration.

3.5.4.24 squareSize

```
float Settings::squareSize
```

The size of a square in your defined unit (point, millimeter,etc).

3.5.4.25 useFisheye

```
bool Settings::useFisheye = false
```

use fisheye camera model for calibration

3.5.4.26 writeExtrinsics

```
bool Settings::writeExtrinsics
```

Write extrinsic parameters.

3.5.4.27 writePoints

```
bool Settings::writePoints
```

Write detected feature points.

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

- src/calibration.hh
- src/calibration.cc

3.6 Tuple< T > Class Template Reference

```
#include <maths.hh>
```

Public Member Functions

- Tuple ()

  Defualt constructor.
- Tuple (int _n,...)

  Constructors that takes the number of objectes to be stored, the objects and then stores them.
- int size () const
- T get (const int _n) const

  Gets the n-th element.
- void add (const T _new)

  Adds a value at the end of the list.
- int remove (const T pos)

  Removes a value from the list.
- int set (const int pos, const T _new)

  Set a value in a certain position, or adds the element if the position equals the number of elements.
- template<class T1 >
  double EuDistance (const Tuple< T1 > B)

  Function that compute the Euclidean Distance between two tuples. They must have the same number of elements. T1 The type of the elements in the second Tuple.
- template<class T1 >
  double MaDistance (const Tuple< T1 > B)

  Function that compute the Manhattan Distance between two tuples. They must have the same number of elements. T1 The type of the elements in the second Tuple.
- template<class T1 >
  double distance (const Tuple< T1 > B, const DISTANCE_TYPE dist=EUCLIDEAN)

  Wrapper to compute different distances. They must have the same number of elements. T1 The type of the elements in the second Tuple.

Friends

- ostream & operator<< (ostream &out, const Tuple &data)

  Overload of operator << to output the content of the tuple.

3.6.1 Detailed Description

template<class T>
class Tuple< T >

This class allows the definition and storage of tuples of different dimensions. Functions to compute distance between tuples are also available.

Template Parameters

T	The type of elements to be stored.

3.6.2 Constructor & Destructor Documentation

3.6.2.1 Tuple() [1/2]

```
template<class T>
Tuple< T >::Tuple ( )  [inline]
```

Defualt constructor.

3.6.2.2 Tuple() [2/2]

```
template<class T>
Tuple< T >::Tuple (
            int _n,
             ...  )  [inline]
```

Constructors that takes the number of objectes to be stored, the objects and then stores them.

Parameters

in	↩ _↩ *n*	Number of obejctes to store.
in	...	Objects to store.

3.6.3 Member Function Documentation

3.6.3.1 add()

```
template<class T>
void Tuple< T >::add (
            const T _new )  [inline]
```

Adds a value at the end of the list.

Parameters

in	*_new*	The new value to be added.

3.6.3.2 distance()

```
template<class T>
template<class T1 >
double Tuple< T >::distance (
            const Tuple< T1 > B,
            const DISTANCE_TYPE dist = EUCLIDEAN )  [inline]
```

Wrapper to compute different distances. They must have the same number of elements. T1 The type of the elements in the second Tuple.

Parameters

in	*B*	The second Tuple to use for computing the distance.
in	*dist*	The type of distance to be computed.

Returns

The distance between the two Tuple.

3.6.3.3 EuDistance()

```
template<class T>
template<class T1 >
double Tuple< T >::EuDistance (
            const Tuple< T1 > B )  [inline]
```

Function that compute the Euclidean Distance between two tuples. They must have the same number of elements. T1 The type of the elements in the second Tuple.

Parameters

in	*B*	the second Tuple to use for computing the distance.

Returns

The Euclidean distance between the two Tuple.

3.6.3.4 get()

```
template<class T>
T Tuple< T >::get (
            const int _n ) const  [inline]
```

Gets the n-th element.

Parameters

in	↩ _↩ *n*	The position of the element to retrieve.

Returns

The element in the n-th position.

3.6.3.5 MaDistance()

```
template<class T>
template<class T1 >
double Tuple< T >::MaDistance (
            const Tuple< T1 > B )  [inline]
```

Function that compute the Manhattan Distance between two tuples. They must have the same number of elements. T1 The type of the elements in the second Tuple.

Parameters

in	*B*	the second Tuple to use for computing the distance.

Returns

The Manhattan distance between the two Tuple.

3.6.3.6 remove()

```
template<class T>
int Tuple< T >::remove (
            const T pos )  [inline]
```

Removes a value from the list.

Parameters

in	*pos*	The position of the value to be removed.

3.6.3.7 set()

```
template<class T>
int Tuple< T >::set (
            const int pos,
            const T _new )  [inline]
```

Set a value in a certain position, or adds the element if the position equals the number of elements.

Parameters

in	*pos*	Must be in $[0, n-1]$. If pos $= n$ then the element is added at the end of the vector.
in	*_new*	The new element to be set.

Returns

1 if everything went right, 0 if the position was greater than n or less the 0.

3.6.3.8 size()

```
template<class T>
int Tuple< T >::size ( ) const  [inline]
```

Returns

The number of stored elements.

3.6.4 Friends And Related Function Documentation

3.6.4.1 operator<<

```
template<class T>
ostream& operator<< (
            ostream & out,
            const Tuple< T > & data )  [friend]
```

Overload of operator << to output the content of the tuple.

Parameters

in	*out*	The output stream.
in	*data*	The Tuple to print.

Returns

An output stream to be printed.

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

- src/maths.hh

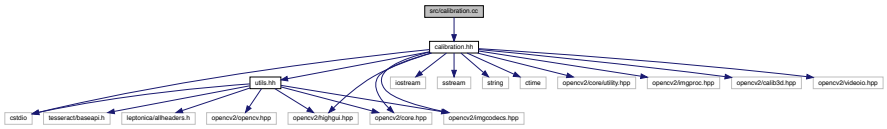
Chapter 4

File Documentation

4.1 src/calibration.cc File Reference

```
#include "calibration.hh"
```

Include dependency graph for calibration.cc:



Functions

- int calibration (const string inputFile)

  Function to run the complete calibration.
- static void read (const FileNode &node, Settings &x, const Settings &default_value)

  Reads settings from file. If there is none then initiate a new `Settings`.
- static double computeReprojectionErrors (const vector< vector< Point3f > > &objectPoints, const vector< vector< Point2f > > &imagePoints, const vector< Mat > &rvecs, const vector< Mat > &tvecs, const Mat &cameraMatrix, const Mat &distCoeffs, vector< float > &perViewErrors, bool fisheye)

  Compute the errors of the projection.
- void calcBoardCornerPositions (Size boardSize, float squareSize, vector< Point3f > &corners)

  This function compute the position of the upper corners of every cell.
- static bool runCalibration (Settings &s, Size &imageSize, Mat &cameraMatrix, Mat &distCoeffs, vector< vector< Point2f > > imagePoints, vector< Mat > &rvecs, vector< Mat > &tvecs, vector< float > &reproj↩ Errs, double &totalAvgErr)

  This function run the calibration creating the matrixed for the camera and the distorsion coefficients.
- static void saveCameraParams (const Settings &s, const Size &imageSize, const Mat &cameraMatrix, const Mat &distCoeffs, const vector< Mat > &rvecs, const vector< Mat > &tvecs, const vector< float > &reproj↩ Errs, const vector< vector< Point2f > > &imagePoints, const double totalAvgErr)

  Function to save the computed parameters to a file.
- bool runCalibrationAndSave (Settings &s, Size imageSize, Mat &cameraMatrix, Mat &distCoeffs, vector< vector< Point2f > > imagePoints)

  Reads settings from file. If there is none then initiate a new `Settings`.

4.1.1 Function Documentation

4.1.1.1 calcBoardCornerPositions()

```
void calcBoardCornerPositions (
            Size boardSize,
            float squareSize,
            vector< Point3f > & corners )
```

This function compute the position of the upper corners of every cell.

Parameters

in	*boardSiz*	The dimension of the chess board.
in	*squareSize*	The dimension of the edge of a cell.
out	*corners*	A vector of Point3fs which equals to the corners of the cells.

4.1.1.2 calibration()

```
int calibration (
            const string inputFile )
```

Function to run the complete calibration.

Parameters

in	*inputFile*	Name of the setting.xml file. It's set to default to default.xml

Returns

-2 if the settings file could be load but the input was not well-formed
-1 if the settings file could not be opened.
0 if everything went fine.

4.1.1.3 computeReprojectionErrors()

```
static double computeReprojectionErrors (
            const vector< vector< Point3f > > & objectPoints,
            const vector< vector< Point2f > > & imagePoints,
            const vector< Mat > & rvecs,
            const vector< Mat > & tvecs,
            const Mat & cameraMatrix,
```

```
            const Mat & distCoeffs,
            vector< float > & perViewErrors,
            bool fisheye )  [static]
```

Compute the errors of the projection.

Parameters

in	*objectPoints*	The real image points which will be projected
in	*rvecs*	Input vector of rotation vectors estimated for each pattern view.
in	*tvecs*	Input vector of translation vectors estimated for each pattern view.
in	*cameraMatrix*	The matrix containing the parameters for the camera
in	*distCoeffs*	The matrix containing the distortion coefficients.
in	*fisheye*	A variable which says if a fish eye correction should be applied or no.
out	*perViewErrors*	A vector containing the error for each image.
out	*imagePoints*	The projected points for each image.

Returns

The total error.

4.1.1.4 read()

```
static void read (
            const FileNode & node,
            Settings & x,
            const Settings & default_value )  [inline], [static]
```

Reads settings from file. If there is none then initiate a new `Settings`.

Parameters

in	*node*	node to consider for getting settings;
in	*x*	`Settings` to configure;
in	*default_value*	`Settings` default value. Setted to `Settings()`.

4.1.1.5 runCalibration()

```
static bool runCalibration (
            Settings & s,
            Size & imageSize,
            Mat & cameraMatrix,
            Mat & distCoeffs,
            vector< vector< Point2f > > imagePoints,
            vector< Mat > & rvecs,
```

```
            vector< Mat > & tvecs,
            vector< float > & reprojErrs,
            double & totalAvgErr )  [static]
```

This function run the calibration creating the matrixed for the camera and the distorsion coefficients.

Parameters

in	*s*	The Settings read from the file and memorized.
in	*imageSize*	The size of the image used in calibrateCamera() to initialize the camera matrix.
in	*imagePoints*	The projected points for each image.
in	*reprojErrs*	The re-projection error, that is a geometric error corresponding to the image distance between a projected point and a measured one.
out	*cameraMatrix*	The matrix of the camera parameters
out	*distCoeffs*	The matrix of the distorsion coefficients.
out	*rvecs*	Output vector of rotation vectors estimated for each pattern view.
out	*tvecs*	Output vector of translation vectors estimated for each pattern view.
out	*totalAvgErr*	The total avarage error given from distorsion.

Returns

false if one or more elements in the cameraMatrix and distCoeffs are invalid.
true if all the elements are valid.

4.1.1.6 runCalibrationAndSave()

```
bool runCalibrationAndSave (
            Settings & s,
            Size imageSize,
            Mat & cameraMatrix,
            Mat & distCoeffs,
            vector< vector< Point2f > > imagePoints )
```

Reads settings from file. If there is none then initiate a new Settings.

Parameters

in	*s*	The Settings being used during the execution.
in	*imageSize*	The dimensions of the images.
in	*imagePoints*	The projected points for each image.
out	*cameraMatrix*	The matrix which is used to store the values for the camera parameters.
out	*distCoeffs*	The matrix which is used to store the distortion coefficients.

Returns

true if the calibration succeded.
false otherwise.

4.1.1.7 saveCameraParams()

```
static void saveCameraParams (
            const Settings & s,
            const Size & imageSize,
            const Mat & cameraMatrix,
            const Mat & distCoeffs,
            const vector< Mat > & rvecs,
            const vector< Mat > & tvecs,
            const vector< float > & reprojErrs,
            const vector< vector< Point2f > > & imagePoints,
            const double totalAvgErr )  [static]
```

Function to save the computed parameters to a file.

Parameters

in	*s*	Use the Settings got at the beginning for information as the output file name, image and board size.
in	*imageSize*	The size of the imgage.
in	*cameraMatrix*	The camera matrix.
in	*distCoeffs*	The distorsion coefficient matrix.
	[int]	rvecs Vector of rotation vectors estimated for each pattern view.
in	*tvecs*	Vector of translation vectors estimated for each pattern view.
in	*reprojErrs*	The re-projection error, that is a geometric error corresponding to the image distance between a projected point and a measured one.
in	*imagePoints*	The projected points for each image.
in	*totalAvgErr*	The total avarage error given from distorsion.

Open file for writing

Stores time of calibration

Store infos about the images

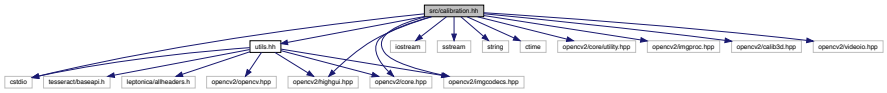
4.2 src/calibration.hh File Reference

Library for calibration.

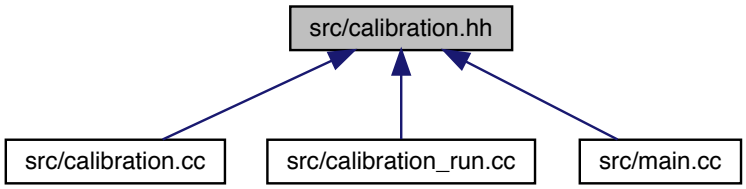
```
#include "utils.hh"
#include <iostream>
#include <sstream>
#include <string>
#include <ctime>
#include <cstdio>
#include <opencv2/core.hpp>
#include <opencv2/core/utility.hpp>
#include <opencv2/imgproc.hpp>
#include <opencv2/calib3d.hpp>
#include <opencv2/imgcodecs.hpp>
#include <opencv2/videoio.hpp>
```

```
#include <opencv2/highgui.hpp>
```

Include dependency graph for calibration.hh:



This graph shows which files directly or indirectly include this file:



Classes

- class Settings

Enumerations

- enum { DETECTION = 0, CAPTURING = 1, CALIBRATED = 2 }

Functions

- int calibration (const string inputFile="data/calib_config.xml")

  Function to run the complete calibration.
- bool runCalibrationAndSave (Settings &s, Size imageSize, Mat &cameraMatrix, Mat &distCoeffs, vector< vector< Point2f > > imagePoints)

  Reads settings from file. If there is none then initiate a new `Settings`.

4.2.1 Detailed Description

Library for calibration.

4.2.2 Enumeration Type Documentation

4.2.2.1 anonymous enum

```
anonymous enum
```

Enumerator

DETECTION	
CAPTURING	
CALIBRATED	

4.2.3 Function Documentation

4.2.3.1 calibration()

```
int calibration (
            const string inputFile )
```

Function to run the complete calibration.

Parameters

in	*inputFile*	Name of the setting.xml file. It's set to default to default.xml

Returns

-2 if the settings file could be load but the input was not well-formed
-1 if the settings file could not be opened.
0 if everything went fine.

4.2.3.2 runCalibrationAndSave()

```
bool runCalibrationAndSave (
            Settings & s,
            Size imageSize,
            Mat & cameraMatrix,
            Mat & distCoeffs,
            vector< vector< Point2f > > imagePoints )
```

Reads settings from file. If there is none then initiate a new Settings.

Parameters

in	*s*	The Settings being used during the execution.
in	*imageSize*	The dimensions of the images.
in	*imagePoints*	The projected points for each image.
out	*cameraMatrix*	The matrix which is used to store the values for the camera parameters.
out	*distCoeffs*	The matrix which is used to store the distortion coefficients.

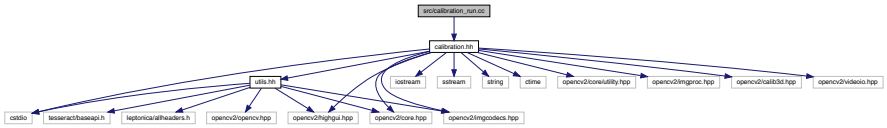
Returns

`true` if the calibration succeded.
`false` otherwise.

4.3 src/calibration_run.cc File Reference

```
#include "calibration.hh"
```

Include dependency graph for calibration_run.cc:



Functions

- int main ()

4.3.1 Function Documentation

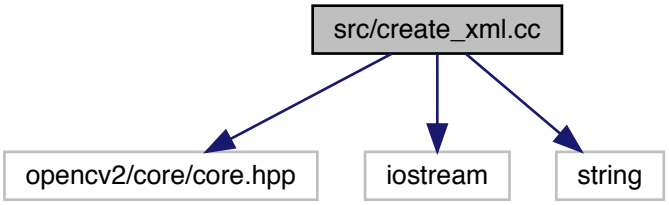
4.3.1.1 main()

```
int main ( )
```

4.4 src/create_xml.cc File Reference

```
#include <opencv2/core/core.hpp>
#include <iostream>
#include <string>
```

Include dependency graph for create_xml.cc:



Functions

- int main ()

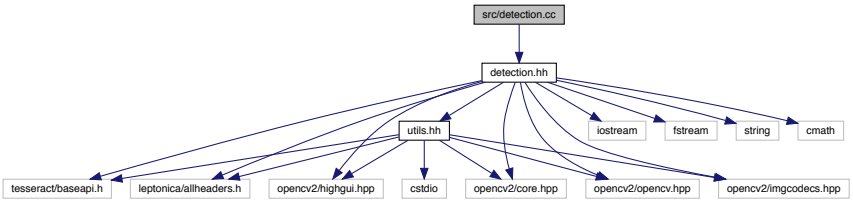
4.4.1 Function Documentation

4.4.1.1 main()

```
int main ( )
```

4.5 src/detection.cc File Reference

```
#include "detection.hh"
```
Include dependency graph for detection.cc:



Macros

- #define WAIT

Functions

- int detection ()

  Loads some images and detects shapes according to different colors.
- void load_number_template ()

  Load some templates and save them in the global variable 'templates'.
- void shape_detection (const Mat &img, const int color)

  Detect shapes inside the image according to the variable 'color'.
- void erode_dilation (Mat &img, const int color)

  It apply some filtering function for isolate the subject and remove the noise.
- void find_contours (const Mat &img, Mat original, const int color)

  Given an image, in black/white format, identify all the borders that delimit the shapes.
- void save_convex_hull (const vector< vector< Point >> &contours, const int color, const vector< int > &victims)

  Given some vector save it in a xml file.
- int number_recognition (Rect blob, const Mat &base)

  Detect a number on an image inside a region of interest.
- void crop_number_section (Mat &ROI)

  Given an image identify the region of interest(ROI) and crop it out.

Variables

- const string xml_settings = "data/settings.xml"
- FileStorage fs_xml
- vector< Mat > templates

4.5.1 Macro Definition Documentation

4.5.1.1 WAIT

```
#define WAIT
```

4.5.2 Function Documentation

4.5.2.1 crop_number_section()

```
void crop_number_section (
            Mat & ROI )
```

Given an image identify the region of interest(ROI) and crop it out.

Parameters

in,out	*ROI*	Is the image that the function will going to elaborate.

4.5.2.2 detection()

```
int detection ( )
```

Loads some images and detects shapes according to different colors.

Returns

Return 0 if the function reach the end.

4.5.2.3 erode_dilation()

```
void erode_dilation (
            Mat & img,
            const int color )
```

It apply some filtering function for isolate the subject and remove the noise.

An example of the sub functions called are: GaussianBlur, Erosion, Dilation and Threshold.

Parameters

in,out	*img*	Is the image on which the function apply the filtering.
in	*color*	Can has 4 value: 0 -> Red 1 -> Green 2 -> Blue 3 -> Black According to the color the filtering functions apply can change in the type and in the order.

4.5.2.4 find_contours()

```
void find_contours (
            const Mat & img,
            Mat original,
            const int color )
```

Given an image, in black/white format, identify all the borders that delimit the shapes.

Parameters

in	*img*	Is an image in HSV format at the base of the elaboration process.
out	*original*	Is the original source of 'img', it is used for showing the detected contours.
in	*color*	Can has 3 value: 0 -> Red 1 -> Green 2 -> Blue Is used for decid which procedure apply to the image.

4.5.2.5 load_number_template()

```
void load_number_template ( )
```

Load some templates and save them in the global variable 'templates'.

4.5.2.6 number_recognition()

```
int number_recognition (
            Rect blob,
            const Mat & base )
```

Detect a number on an image inside a region of interest.

Parameters

in	*blob*	Identify the region of interest inside the image 'base'.
in	*base*	Is the image where the function will going to search the number.

Returns

The number recognise, '-1' otherwise.

4.5.2.7 save_convex_hull()

```
void save_convex_hull (
            const vector< vector< Point >> & contours,
            const int color,
            const vector< int > & victims )
```

Given some vector save it in a xml file.

Parameters

in	*contours*	Is a vector that is saved in a xml file.
in	*color*	Is the parameter according to which the function decide if saved ('color==1') or not ('otherwise') the vector 'victims'.
in	*victims*	Is a vector that is saved in a xml file.

4.5.2.8 shape_detection()

```
void shape_detection (
            const Mat & img,
            const int color )
```

Detect shapes inside the image according to the variable 'color'.

Parameters

in	*img*	Image on which the research will done.
in	*color*	Can has 3 value: 0 -> Red 1 -> Green
		2 -> Blue These color identify the possible spectrum that the function search on the image.

4.5.3 Variable Documentation

4.5.3.1 fs_xml

```
FileStorage fs_xml
```

4.5.3.2 templates

```
vector<Mat> templates
```

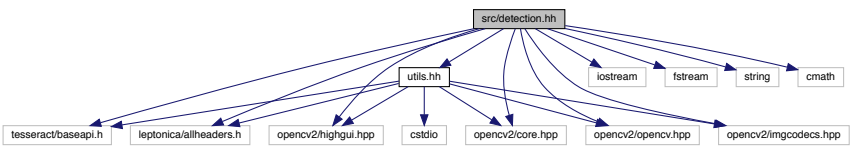
4.5.3.3 xml_settings

```
const string xml_settings = "data/settings.xml"
```

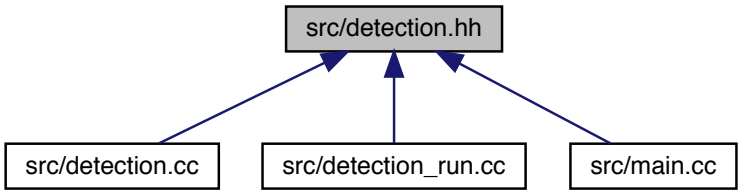
4.6 src/detection.hh File Reference

```
#include <tesseract/baseapi.h>
#include <leptonica/allheaders.h>
#include "utils.hh"
#include <iostream>
#include <fstream>
#include <string>
#include <cmath>
#include <opencv2/highgui.hpp>
#include <opencv2/core.hpp>
#include <opencv2/opencv.hpp>
#include <opencv2/imgcodecs.hpp>
```

Include dependency graph for detection.hh:



This graph shows which files directly or indirectly include this file:



Functions

- int detection ()

  Loads some images and detects shapes according to different colors.
- void shape_detection (const Mat &img, const int color)

  Detect shapes inside the image according to the variable 'color'.
- void erode_dilation (Mat &img, const int color)

  It apply some filtering function for isolate the subject and remove the noise.
- void find_contours (const Mat &img, Mat original, const int color)

  Given an image, in black/white format, identify all the borders that delimit the shapes.
- int number_recognition (Rect blob, const Mat &base)

  Detect a number on an image inside a region of interest.
- void save_convex_hull (const vector< vector< Point >> &contours, const int color, const vector< int > &victims)

  Given some vector save it in a xml file.
- void load_number_template ()

  Load some templates and save them in the global variable 'templates'.
- void crop_number_section (Mat &processROI)

  Given an image identify the region of interest(ROI) and crop it out.

4.6.1 Function Documentation

4.6.1.1 crop_number_section()

```
void crop_number_section (
            Mat & ROI )
```

Given an image identify the region of interest(ROI) and crop it out.

Parameters

in,out	*ROI*	Is the image that the function will going to elaborate.

4.6.1.2 detection()

```
int detection ( )
```

Loads some images and detects shapes according to different colors.

Returns

Return 0 if the function reach the end.

4.6.1.3 erode_dilation()

```
void erode_dilation (
            Mat & img,
            const int color )
```

It apply some filtering function for isolate the subject and remove the noise.

An example of the sub functions called are: GaussianBlur, Erosion, Dilation and Threshold.

Parameters

in,out	*img*	Is the image on which the function apply the filtering.
in	*color*	Can has 4 value: 0 -> Red 1 -> Green 2 -> Blue 3 -> Black According to the color the filtering functions apply can change in the type and in the order.

4.6.1.4 find_contours()

```
void find_contours (
            const Mat & img,
            Mat original,
            const int color )
```

Given an image, in black/white format, identify all the borders that delimit the shapes.

Parameters

in	*img*	Is an image in HSV format at the base of the elaboration process.
out	*original*	Is the original source of 'img', it is used for showing the detected contours.

Parameters

in	*color*	Can has 3 value: 0 -> Red 1 -> Green 2 -> Blue Is used for decid which procedure apply to the image.

4.6.1.5 load_number_template()

```
void load_number_template ( )
```

Load some templates and save them in the global variable 'templates'.

4.6.1.6 number_recognition()

```
int number_recognition (
            Rect blob,
            const Mat & base )
```

Detect a number on an image inside a region of interest.

Parameters

in	*blob*	Identify the region of interest inside the image 'base'.
in	*base*	Is the image where the function will going to search the number.

Returns

The number recognise, '-1' otherwise.

4.6.1.7 save_convex_hull()

```
void save_convex_hull (
            const vector< vector< Point >> & contours,
            const int color,
            const vector< int > & victims )
```

Given some vector save it in a xml file.

Parameters

in	*contours*	Is a vector that is saved in a xml file.
in	*color*	Is the parameter according to which the function decide if saved ('color==1') or not ('otherwise') the vector 'victims'.
in	*victims*	Is a vector that is saved in a xml file.

4.6.1.8 shape_detection()

```
void shape_detection (
            const Mat & img,
            const int color )
```

Detect shapes inside the image according to the variable 'color'.

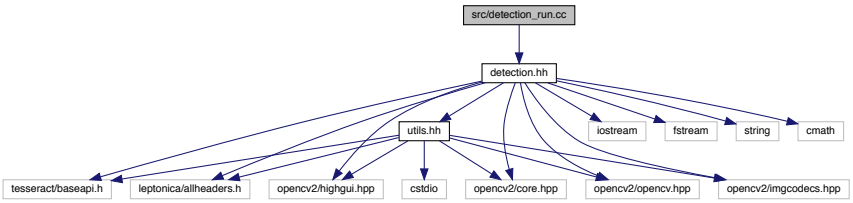
Parameters

in	*img*	Image on which the research will done.
in	*color*	Can has 3 value: 0 -> Red 1 -> Green 2 -> Blue These color identify the possible spectrum that the function search on the image.

4.7 src/detection_run.cc File Reference

```
#include "detection.hh"
```

Include dependency graph for detection_run.cc:



Functions

- int main ()

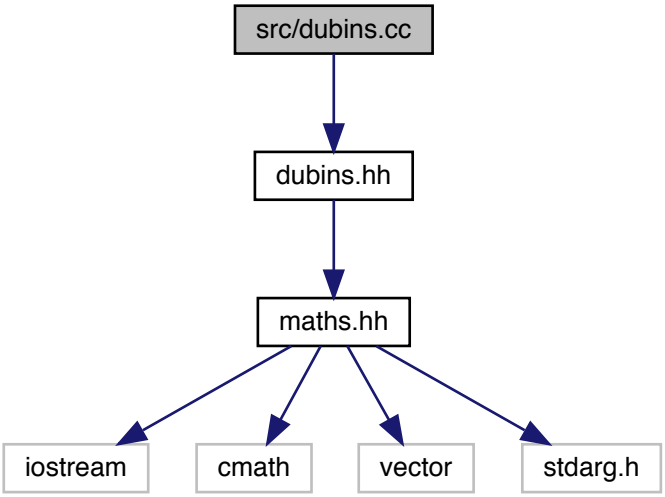
4.7.1 Function Documentation

4.7.1.1 main()

```
int main ( )
```

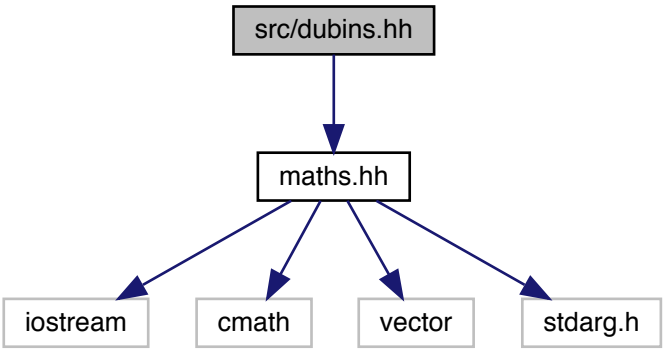
4.8 src/dubins.cc File Reference

`#include "dubins.hh"`
Include dependency graph for dubins.cc:

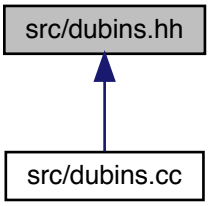


4.9 src/dubins.hh File Reference

`#include "maths.hh"`
Include dependency graph for dubins.hh:



This graph shows which files directly or indirectly include this file:



Classes

- class dubins

4.10 src/main.cc File Reference

```
#include <tesseract/baseapi.h>
#include <leptonica/allheaders.h>
#include "utils.hh"
#include "detection.hh"
#include "unwrapping.hh"
#include "calibration.hh"
```

Include dependency graph for main.cc:

Functions

- int main ()

4.10.1 Function Documentation

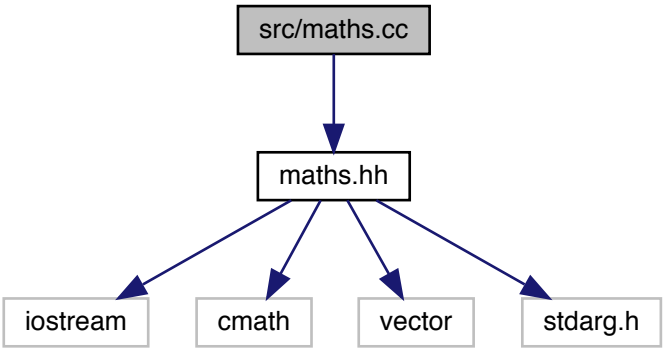
4.10.1.1 main()

```
int main ( )
```

4.11 src/maths.cc File Reference

```
#include "maths.hh"
```

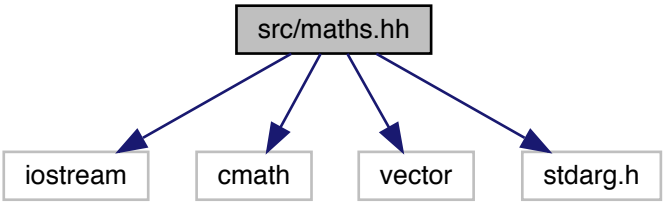
Include dependency graph for maths.cc:



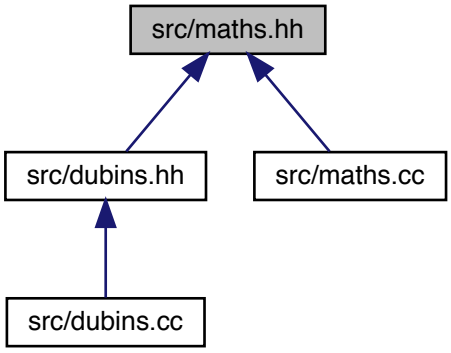
4.12 src/maths.hh File Reference

```
#include <iostream>
#include <cmath>
#include <vector>
#include <stdarg.h>
```

Include dependency graph for maths.hh:



This graph shows which files directly or indirectly include this file:



Classes

- class Angle

  This class allows to save and handle angles. It supports DEG and RAD, operations such as addition and subtraction with operators overloading, conversion from RAD to DEG and viceversa and normalization of the angle.
- class Tuple< T >
- class Point2< T >

  Class that stores two value to construct a point in 2D. The value is saved in a Tuple.
- class Configuration2< T1 >

  This class stores a configuration, that is a point and an angle.

Macros

- #define DEGTORAD M_PI/180
- #define RADTODEG 180/M_PI

Enumerations

- enum DISTANCE_TYPE { EUCLIDEAN, MANHATTAN }

Functions

- template<class T >
  T pow2 (const T x)

4.12.1 Macro Definition Documentation

4.12.1.1 DEGTORAD

```
#define DEGTORAD M_PI/180
```

4.12.1.2 RADTODEG

```
#define RADTODEG 180/M_PI
```

4.12.2 Enumeration Type Documentation

4.12.2.1 DISTANCE_TYPE

```
enum DISTANCE_TYPE
```

Enumerator

EUCLIDEAN	
MANHATTAN	

4.12.3 Function Documentation

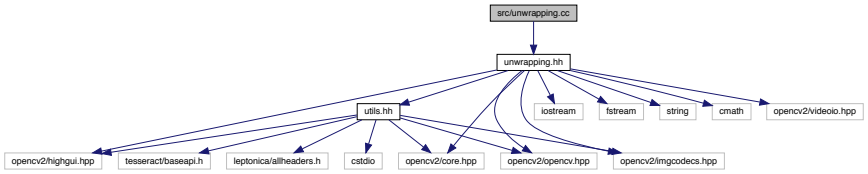
4.12.3.1 pow2()

```
template<class T >
T pow2 (
            const T x )  [inline]
```

4.13 src/unwrapping.cc File Reference

```
#include "unwrapping.hh"
```

Include dependency graph for unwrapping.cc:



Macros

- #define WAIT
- #define area_ratio 0.7

Functions

- static float distance (Point c1, Point c2)

  Compute the euclidean distance.
- static void swap (int &a, int &b)

  Swap the two integers passed.
- int unwrapping ()

  Take some images according to a xml and unwrap the black rectangle inside the image after appling undistortion trasformation.
- void loadCoefficients (const string filename, Mat &camera_matrix, Mat &dist_coeffs)

  Load coefficients from a file.

Variables

- const string xml_settings = "data/settings.xml"

4.13.1 Macro Definition Documentation

4.13.1.1 area_ratio

```
#define area_ratio 0.7
```

4.13.1.2 WAIT

```
#define WAIT
```

4.13.2 Function Documentation

4.13.2.1 distance()

```
static float distance (
            Point c1,
            Point c2 )  [static]
```

Compute the euclidean distance.

Parameters

in,out	*c1*	The first point.
in,out	*c2*	The second point.

Returns

The euclidean distance.

4.13.2.2 loadCoefficients()

```
void loadCoefficients (
            const string filename,
            Mat & camera_matrix,
            Mat & dist_coeffs )
```

Load coefficients from a file.

Load two matrix 'camera_matrix' and 'distortion_coefficients' from the xml file passed.

Parameters

in	*filename*	The string that identify the location of the xml file.
out	*camera_matrix*	Where the 'camera_matrix' matrix is saved.
out	*dist_coeffs*	Where the 'distortion_coefficients' matrix is saved.

4.13.2.3 swap()

```
static void swap (
            int & a,
            int & b )  [static]
```

Swap the two integers passed.

Parameters

in,out	*a*	First parameter.
in,out	*b*	Second parameter.

4.13.2.4 unwrapping()

```
int unwrapping ( )
```

Take some images according to a xml and unwrap the black rectangle inside the image after appling undistortion trasformation.

Load from the xml file 'data/settings.xml' the name of some images, load the images from the file,
apply the calibration (undistortion trasformation) thanks to the matrices load with the 'loadCoefficients' function.
Then, with the use of a filter for the black the region of interest (a rectangle) is identified and all the perspective is rotated for reach a top view of the rectangle.
Finally, the images are saved on some files.

Returns

A 0 is return if the function reach the end.

4.13.3 Variable Documentation

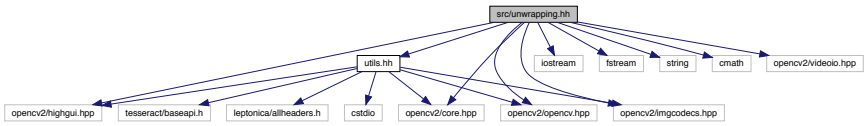
4.13.3.1 xml_settings

```
const string xml_settings = "data/settings.xml"
```

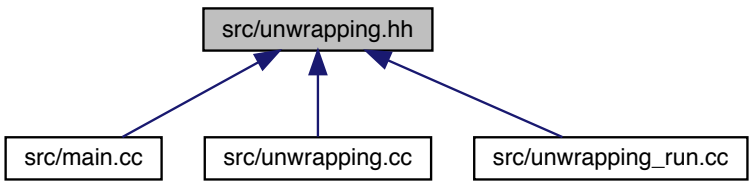
4.14 src/unwrapping.hh File Reference

```
#include "utils.hh"
#include <iostream>
#include <fstream>
#include <string>
#include <cmath>
#include <opencv2/videoio.hpp>
#include <opencv2/highgui.hpp>
#include <opencv2/core.hpp>
#include <opencv2/opencv.hpp>
#include <opencv2/imgcodecs.hpp>
```

Include dependency graph for unwrapping.hh:



This graph shows which files directly or indirectly include this file:



Functions

- int unwrapping ()

  Take some images according to a xml and unwrap the black rectangle inside the image after appling undistortion trasformation.
- void loadCoefficients (const string filename, Mat &camera_matrix, Mat &dist_coeffs)

  Load coefficients from a file.

4.14.1 Function Documentation

4.14.1.1 loadCoefficients()

```
void loadCoefficients (
            const string filename,
            Mat & camera_matrix,
            Mat & dist_coeffs )
```

Load coefficients from a file.

Load two matrix 'camera_matrix' and 'distortion_coefficients' from the xml file passed.

Parameters

in	*filename*	The string that identify the location of the xml file.
out	*camera_matrix*	Where the 'camera_matrix' matrix is saved.
out	*dist_coeffs*	Where the 'distortion_coefficients' matrix is saved.

4.14.1.2 unwrapping()

```
int unwrapping ( )
```

Take some images according to a xml and unwrap the black rectangle inside the image after appling undistortion trasformation.

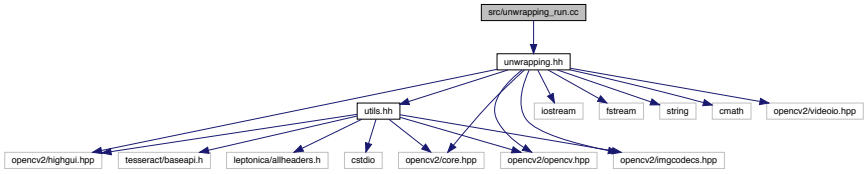
Load from the xml file 'data/settings.xml' the name of some images, load the images from the file,
apply the calibration (undistortion trasformation) thanks to the matrices load with the 'loadCoefficients' function.
Then, with the use of a filter for the black the region of interest (a rectangle) is identified and all the perspective is rotated for reach a top view of the rectangle.
Finally, the images are saved on some files.

Returns

A 0 is return if the function reach the end.

4.15 src/unwrapping_run.cc File Reference

```
#include "unwrapping.hh"
```
Include dependency graph for unwrapping_run.cc:



Functions

- int main ()

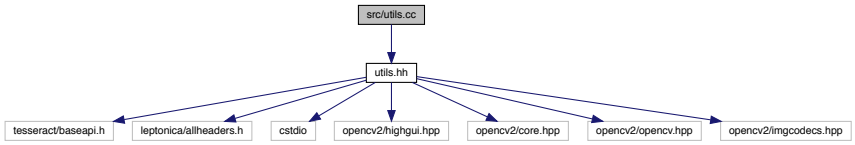
4.15.1 Function Documentation

4.15.1.1 main()

```
int main ( )
```

4.16 src/utils.cc File Reference

```
#include "utils.hh"
```
Include dependency graph for utils.cc:



Functions

- void my_imshow (const char *win_name, Mat img, bool reset)

4.16.1 Function Documentation

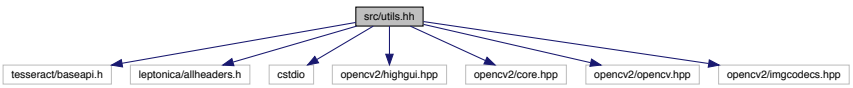
4.16.1.1 my_imshow()

```
void my_imshow (
            const char * win_name,
            Mat img,
            bool reset )
```

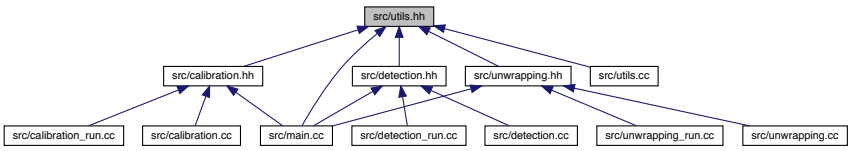
4.17 src/utils.hh File Reference

```
#include <tesseract/baseapi.h>
#include <leptonica/allheaders.h>
#include <cstdio>
#include <opencv2/highgui.hpp>
#include <opencv2/core.hpp>
#include <opencv2/opencv.hpp>
#include <opencv2/imgcodecs.hpp>
```

Include dependency graph for utils.hh:



This graph shows which files directly or indirectly include this file:



Macros

- #define INFO(msg)

Functions

- void my_imshow (const char *win_name, Mat img, bool reset=false)

4.17.1 Macro Definition Documentation

4.17.1.1 INFO

```
#define INFO(
              msg )
```

4.17.2 Function Documentation

4.17.2.1 my_imshow()

```
void my_imshow (
             const char * win_name,
             Mat img,
             bool reset = false )
```

Index