Threading B-Splines Through 2D Channels

1.0

Generated by Doxygen 1.8.10

Tue Jul 12 2016 17:17:38

Contents

Chapter 1

The Channel2D Library Documentation

Chapter 2

Introduction

The `Channel2D` Library consists of a set of `C++` classes for solving planar instances of the **channel problem**. A detailed description of the channel problem and its solution (slightly different from the one implemented in the `Channel2D` library) can be found in the following papers:

- David Lutterkort and Jörg Peters. Smooth paths in a polygonal channel. Proceedings of the 15th Annual ACM Symposium on Computational Geometry (SoCG), Miami Beach, FL, USA, June 13-16, 1999. (`PS`)

- Ashish Myles and Jörg Peters. Threading splines through 3d channels. *Computer-Aided Design* (CAD), v. 37, n. 2, pp. 139-148, 2005. (`PDF`)

I really encourage you to read both papers (at least Section 3 of the second paper from top to bottom) before you try to use the `Channel2D` library, as the input file format requires some idea about the input values and unknowns of the problem.

For the 2D version of the channel problem, we are given a channel, which is a planar region delimited by two polygonal chains: the *lower* and *upper envelopes* of the channel. For instance,

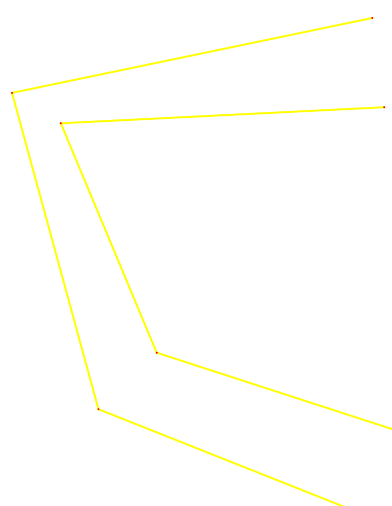


Figure 2.1: Example of an open channel

The two polygonal chains must have the same number of vertices (resp. edges). There is a one-to-one correspondence between the set of points (resp. edges) of the lower and upper envelopes. To be more precise, given the sequences of vertices (resp. edges) of the lower and upper envelopes, obtained by a *counterclockwise* traversal of the envelope, the i-th vertex (resp. edge) of the lower envelope is in correspondence with the i-th vertex (resp. edge) of the upper envelope. However, the two corresponding edges need not be parallel.

A solution for the problem is a C^k spline curve of a given degree d, with $k \geq 1$ and $d \geq 2$, which is entirely contained in the channel and whose endpoints belong to (distinct) extremities of the channel. For instance,

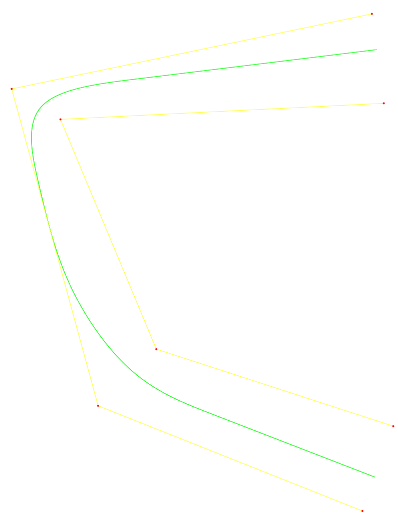


Figure 2.2: Example of a solution for the channel problem

The spline curve in shown in green and its control polygon is shown in blue. Myles and Peters devised a solution for the channel problem as a linear program whose constraints are responsible for keeping the spline inside the channel. In turn, the objective function can be tuned to influence on the geometry of the spline. In the `Channel2D` library, the same objective function given by Myles and Peters' paper was adopted, which aims at minimizing the total curvature variation. This is done indirectly by defining a linear function based on the second differences of the Bézier coefficients of the curves that make up the spline.

The main differences between the solution implemented in the `Channel2D` library and the one proposed by Myles and Peters are two-fold. First, the constraints of the linear program have been slightly modified, so that the resulting curve is C^2 rather than C^1. Second, the resulting curve is always a cubic uniform b-spline curve, i.e., the degree d is fixed and equal to 3. In Myles and Peters' paper, the degree d of the curve is chosen by the user.

A channel can either be open (as in the previous example) or closed (as shown below).

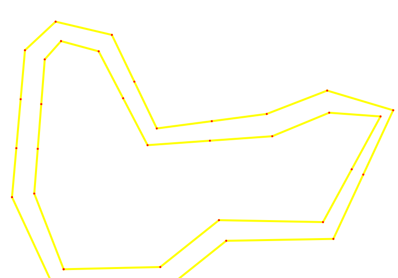


Figure 2.3: Example of a closed channel

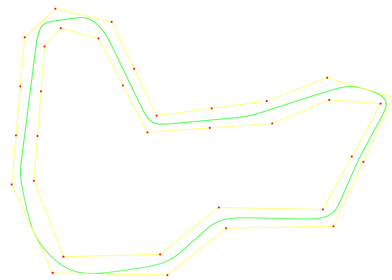


Figure 2.4: Example of a solution for the channel problem

To specify an instance of the channel problem, you must provide the Cartesian coordinates, $(lx_0, ly_0), \ldots, (lx_n, ly_n)$ and $(ux_0, uy_0), \ldots, (ux_n, uy_n)$, of the lower and upper envelopes, respectively, together with three parameter values: ns, nc, and $closed$. Parameter ns specifies the number of b-spline segments of the spline curve. Since the curve is a cubic b-spline, each b-spline curve segment is given by four consecutive control points. So, the number of control points of the curve is equal to $ns + 3$, and thus there is no need to specify the number of control points of the curve. Parameter nc specifies the number of *c-segments* of the channel. Each *c-segment* is given by a pair of corresponding edges of the lower and upper envelopes of the channel. The number of curve segments, ns, must be a multiple of the number of c-segment, nc. We have experimentally observed that choosing $ns = 3 \times nc$ is a good trade off between smoothness and number of control points of the curve. Note that the number of vertices in each envelope is $nc + 1$ if the channel is *open*, and equal to nc if the channel is closed.

For the first example of the channel problem I showed above, we have $ns = 9$ and $nc = 3$:

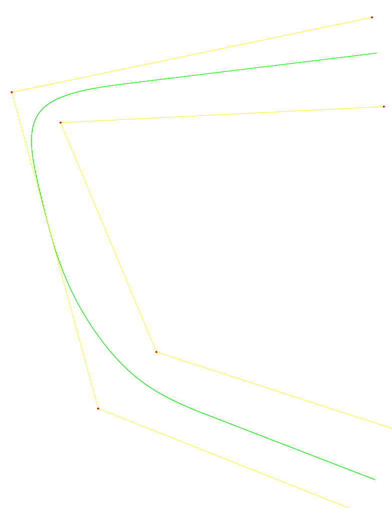


Figure 2.5: Example of a channel

That is, the spline consists of exactly $ns = 9$ curve segments. Starting from the first curve segment, each three consecutive curve segments are bounded (above and below) by the same pair of corresponding edges of the channel: an edge of the lower envelope and an edge of the upper envelope. Each envelope has exactly $nc = 3$ edges and $nc + 1 = 4$ vertices. The entire b-spline curve has $ns + 3 = 12$ control points.

For the second example of the channel problem I showed above, we have $ns = 51$ and $nc = 17$:

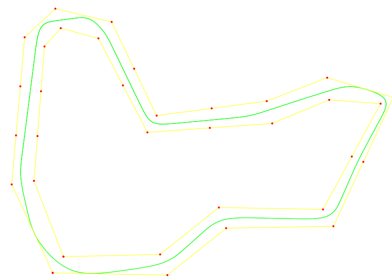


Figure 2.6: Example of a solution for the channel problem

That is, the spline consists of exactly $ns = 51$ curve segments. Starting from the first curve segment, each three consecutive curve segments are bounded (above and below) by the same pair of corresponding edges of the channel: an edge of the lower envelope and an edge of the upper envelope. Each envelope has exactly $nc = 17$ edges and $nc = 17$ vertices. The entire b-spline curve has $ns + 3 = 54$ control points.

It is worth mentioning that the channel problem may not have a solution if the value of ns is not large enough. In our experiments, letting $ns = 3 \times nc$ was sufficient to get a solution for all instances of the test dataset. But, if I had chosen $ns = 2 \times nc$, for instance, the code would not be able to build a few curves from the same dataset. If an instance of the channel problem has no solution, the main function of the `Channel2D` library will show a message to indicate the infeasibility of the problem. In principle, the method could apply a midpoint subdivision to the curve and try to solve the problem again, but such an approach has not been implemented in the current version of the library. I also noted that the infeasibility of the problem is sometimes dictated by the channel geometry. As a rule of thumb, the lengths of any two consecutive c-segments of the channel should not differ by a factor greater than 2. This is even more critical when two consecutive c-segments meet at a sharp angle. I actually wrote code to refine channels, so that the lengths of any two consecutive c-segments of the channel do not differ by a factor greater than 2. This code has not been packaged together with the Channel2D library code, but if you are interested in having a copy of it, please email me.

Chapter 3

Installing and compiling the library

The Channel2D library can be easily installed by downloading and unzipping the file channel-2d.zip. After doing that, one should see a directory named channel-2d with subdirectories bin, data, doc, lib, scripts, src, and tst inside. Subdirectory scripts contains an installation script, install.sh, that compiles the *Channel2D* library and an executable file that demonstrates how to use the library.

Before you execute the installation script install.sh, make sure you install the GNU GLPK in your computer. This toolkit contains the linear program solver used by the Channel2D library. If your computer runs Mac OSX, then you can install GLPK from macports. If your computer is based on a Unix-like system, such as Linux, then you can follow the installation instructions in the GLPK documentation pages. If your computer runs Windows, then you may install GLPK by following the instructions you find here. Once you have installed GNU GLPK in your computer, take note of the directories where the header file glpk.h and the library file libglpk.a are. In my computer, these files can be found in the following directories:

```
/opt/local/include
```

and

```
/opt/local/lib
```

Finally, you must edit two of my files named Makefile, so that you replace the directories above with the corresponding ones in your computer. The first Makefile is inside subdirectory src. Its content is:

```
CC = g++
AR = ar

#CFLAGS = -g -c -std=c++11 -Wall -pedantic
CFLAGS = -O2 -c -std=c++11 -Wall -pedantic

INC1 = .
INC2 = /opt/local/include

INCS = -I$(INC1) -I$(INC2)

OBJ = curvebuilder.o

LIB = libChannel2D.a

curvebuilder.o: $(INC1)/a3.hpp $(INC1)/tabulatedfunction.hpp \
                $(INC1)/bound.hpp $(INC1)/coefficient.hpp \
                $(INC1)/exceptionobject.hpp $(INC2)/glpk.h \
                $(INC1)/curvebuilder.hpp $(INC1)/curvebuilder.cpp
        $(CC) $(CFLAGS) $(INC1)/curvebuilder.cpp $(INCS)

all: $(OBJ)

lib:    $(INC1)/a3.hpp $(INC1)/tabulatedfunction.hpp \
```

```
        $(INC1)/bound.hpp $(INC1)/coefficient.hpp \
        $(INC1)/exceptionobject.hpp $(INC2)/glpk.h \
        $(INC1)/curvebuilder.hpp $(INC1)/curvebuilder.cpp
        $(AR) rc $(LIB) $(OBJ)
        ranlib $(LIB)
        mv $(LIB) ../lib

clean:
        rm -f *.o *~

realclean:
        rm -f *.o *~ ../lib/libChannel2D.a
```

Replace the line

```
INC2 = /opt/local/include
```

with

```
INC2 = path to the include directory of your computer where glpk.h is
```

Repeat the above step for the `Makefile` inside subdirectory `tst`. Its content is

```
CC = g++

#CFLAGS = -g -c -Wall -pedantic -std=c++11 -DDEBUGMODE
CFLAGS = -O2 -c -Wall -pedantic -std=c++11

#LFLAGS = -g
LFLAGS = -O2

INC1 = .
INC2 = ../src
INC3 = /opt/local/include

INCS = -I$(INC1) -I$(INC2) -I$(INC3)

LIB1 = ../lib
LIB2 = /opt/local/lib

LIBS = -L$(LIB1) -L$(LIB2) -lm -lChannel2D -lglpk

OBJS = main.o

bc2d:   $(OBJS)
        $(CC) $(LFLAGS) $(OBJS) -o channel2d $(LIBS)
        mv channel2d ../bin/.

main.o: $(INC2)/a3.hpp $(INC2)/tabulatedfunction.hpp \
        $(INC2)/exceptionobject.hpp $(INC3)/glpk.h \
        $(INC2)/curvebuilder.hpp $(INC1)/main.cpp
        $(CC) $(CFLAGS) $(INC1)/main.cpp $(INCS)

clean:
        rm -fr *.o *~

realclean:
        rm -fr *.o *~ ../bin/channel2d
```

Replace the lines

```
INC3 = /opt/local/include
LIB2 = /opt/local/lib
```

with

```
INC3 = path to the include directory of your computer where glpk.h is
LIB2 = path to the lib directory of your computer where libglpk.a is
```

If you reach this point after executing the instructions above, then you are ready to compile the `Channel2D` library as well as a simple program to demonstrate how the library can be used. This is easy. The hard part is the installation of the GNU GLPK. If your computer runs Mac OSX or Linux, open a terminal, go to subdirectory `scripts`, and execute the script `install.sh`:

```
cd scripts
```

and

```
sh install.sh
```

If everything goes as expected in the compilation process, one should see the library `libChannel2D.a` inside subdirectory `lib`, and the executable `channel2d` inside subdirectory `bin`. Using this executable, we can run some examples of the channel problem, which are located in subdirectory `data/channels`. You find the details in section The CurveBuilder class API.

I also left an `XCode` project file inside the `channel-2d` directory, but I currently have no `Windows` machine to create a .NET project file. So, if your computer runs Windows, you may have to create your own .NET project file by inspecting the two `Makefile` listed before. As you can see, they are both quite small. So, it should not be a problem to create your own .NET project file.

The current version of the library was successfully compiled and tested using the following operating system(s) / compiler(s).

- Mac OSX 10.10.5 / GNU gcc 4.2.1 and clang-602.0.53

The `Channel2D` library code is based on plain features of the C++ language. Apart from the GLPK functions, there is nothing that should prevent the code from being successfully compiled by any wide used and up-to-date C++ compiler. However, if you face any problems, please feel free to contact me. Use the email address given inside the sources files of the library.

Chapter 4

The CurveBuilder class API

The main class of the `Channel2D` library is `CurveBuilder`. To solve the channel problem, we first instantiate an object of this class using the class constructor:

```
CurveBuilder(
  size_t ns ,
  size_t nc ,
  bool closed ,
  double* lx ,
  double* ly ,
  double* ux ,
  double* uy
)
throw( ExceptionObject ) ;
```

as in

```
CurveBuilder* builder = 0 ;
try {
  builder = new CurveBuilder(
                            np ,
                            nc ,
                            closed ,
                            &lx[ 0 ] ,
                            &ly[ 0 ] ,
                            &ux[ 0 ] ,
                            &uy[ 0 ]
                           ) ;
}
catch ( const ExceptionObject& xpt ) {
  treat_exception( xpt ) ;
  exit( EXIT_FAILURE ) ;
}
```

Variables `ns` and `nc` hold the values of the parameters ns and nc, respectively, that we discussed in section Introduction. Variable `closed` is boolean. If its value is `true`, then the channel is assumed to be closed. If its value is `false`, then the channel is assumed to be open. Variables `lx` and `ly` are two arrays of elements of type `double` that hold the x and y coordinates of the lower envelope of the channel. Likewise, variables `ux` and `uy` are two arrays hold of elements of type `double` that hold the x and y coordinates of the upper envelope of the channel. It is assumed that the vertices with coordinates (lx[i],ly[i]) and (ux[i],uy[i]) are corresponding vertices of the lower and upper envelopes, respectively. **IT IS VERY IMPORTANT** that the vertices are listed in the same order they are visited in a *counterclockwise traversal* of the envelopes (starting at one extreme of the channel). This is equivalent to walking along the edges of the envelopes from the "outside" of the channel in a counterclockwise direction. The reason for such a restriction is that my code must compute outward normals to the edges of the envelopes, and the direction of these normals matters! If the vertices are not given as they are found in a counterclockwise traversal of the envelope edges, the direction of the normals will be opposite to the correct one. As a result, the inequalities of the linear program will be incorrectly defined, which will prevent the solver from finding the correct optimal solution for the channel problem.

Once an instance of the channel problem is created, the next step is to find a solution for it. Class CurveBuilder offers the following method for solving the channel problem:

```
bool build( int& error ) ;
```

This method calls the GNU GLPK linear program (LP) solver to solve the instance of the channel problem defined by the constructor of the class CurveBuilder. If the solver finds a solution, build returns the logic value true. Otherwise, it returns the logic value false. In addition, the error code returned by the GLPK solver is stored in error. Using this error code, we can find out why the solver could not solve the problem. If the problem has been specified correctly (and if my code has no bug!), the fact that the solver cannot find a solution is mostly due to the infeasibility of the problem.

A typical call for build() is shown below:

```
int error ;
bool res = b.build( error ) ;
```

If the value of res is true, then we can recover the control points of the splines by invoking another function of class CurveBuilder:

```
double get_control_value( unsigned i , unsigned v ) const throw( ExceptionObject )
```

The above function has two input parameters: i and v. These parameters tells function get_control_value that we want the v-th coordinate of the i-th control point of the b-spline curve, i.e., $b_{i,v}$. Parameter i holds a value in the interval $[0, ns+2]$. Parameter v holds the value 0 or 1, where 0 corresponds to the x coordinate and 1 corresponds to the y coordinate of $b_{i,v}$. The following piece of code prints out the coordinates of all control points of the spline found by the GNU GLPK solver:

```
for ( size_t i = 0 ; i < NumberOfControlPoints ; i++ ) {
  double x ;
  double y ;
  try {
    x = b.get_control_value( i , 0 ) ;
    y = b.get_control_value( i , 1 ) ;
  }
  catch ( const ExceptionObject& xpt ) {
    treat_exception( xpt ) ;
    ou.close() ;
    exit( EXIT_FAILURE ) ;
  }
}
```

The set of public methods of class CurveBuilder consists of many more functions. But, the ones presented here are enough to prescribe, solve, and obtain the solution of an instance of the channel problem. Section Using the library API describes a simple C++ program to read a file with the description of an instance of the channel problem, solve the problem using the functions I explained before, and then save the solution of the problem to an output file.

Chapter 5

Using the library API

I wrote a simple C++ program to show how to use the Channel2D library to solve an instance of the channel problem. Here, I will examine and explain each line of the main() function of the program. You can find the program in the subdirectory tst. The program has only one file: main.cpp. Below are the header files included in main.cpp:

```
#include <iostream>                 // std::cout, std::endl, std::cerr
#include <fstream>                  // std::ifstream, std::ofstream
#include <sstream>                  // std::sstream
#include <string>                   // std::string
#include <cstdlib>                  // exit, EXIT_SUCCESS, EXIT_FAILURE, size_t
#include <iomanip>                  // std::setprecision
#include <cassert>                  // assert
#include <ctime>                    // time, clock, CLOCKS_PER_SEC, clock_t
#include <cmath>                    // fabs

#include "exceptionobject.hpp"      // channel::ExceptionObject
#include "curvebuilder.hpp"         // channel::CurveBuilder

using std::cin  ;
using std::cout ;
using std::cerr ;
using std::endl ;
using std::string ;
using channel::CurveBuilder ;
using channel::ExceptionObject ;
```

File curvebuilder.hpp contains the definition of class CurveBuilder and file exceptionobject.hpp contains the definition of a class, ExceptionObject, that I use to throw and treat exceptions in a more friendly way. The next lines check the command-line arguments and read an input file with the input values of an instance of the channel problem:

```
if ( ( argc != 3 ) && ( argc != 4 ) ) {
  cerr << "Usage: "
       << endl
       << "\t\t channel2d arg1 arg2 [ arg3 ]"
       << endl
       << "\t\t arg1: name of the file describing the polygonal channel"
       << endl
       << "\t\t arg2: name of the output file describing the computed cubic b-spline curve"
       << endl
       << "\t\t arg3: name of an output file to store a CPLEX format definition of the LP solved by this
       program (OPTIONAL)"
       << endl
       << endl ;
  cerr.flush() ;

  return EXIT_FAILURE ;
}

string fn1( argv[ 1 ] ) ;

size_t ns ;
```

```
size_t nc ;
bool closed ;
double* lx ;
double* ly ;
double* ux ;
double* uy ;

start = clock() ;
try {
  read_input( fn1 , ns , nc , closed , lx , ly , ux , uy ) ;
}
catch ( const ExceptionObject& xpt ) {
  treat_exception( xpt ) ;
  exit( EXIT_FAILURE ) ;
}
```

As we can see, the program requires two or three file names as command-line arguments. The first name refers to the file containing the input values of an instance of the channel problem. The second name refers to the file in which we want the program to write out the control points of the resulting spline curve, i.e., the solution of the channel problem. The third name is *optional* and refers to a file in which the program will store a description of the linear program corresponding to the instance of the channel problem given as input. I initially created this option as a way of debugging my code as needed. The description of the LP is given in CPLEX format, which is quite easy to read and look for mistakes. We can also give this description to any LP solver that takes in files in CPLEX format. The GNU GLPK itself is such a solver. We can use its `glpsol` function to solve an instance of a linear program written in CPLEX format. When I was done with the first version of the code, I though it would be useful to leave the option of generating this file in the distributed version of the code.

After checking the number of input command-line arguments, the code reads in the input file using function `read_input()`. This function recovers the input values of the instance of the problem: `ns`, `nc`, `closed`, `lx` , `ly`, `ux`, and `uy`. I already talked about all these parameters. Observe that the memory occupied by the arrays `lx` , `ly`, `ux`, and `uy` is allocated inside function `read_input()`.

The next lines invoke the constructor of `CurvedBuilder` to create the given instance of the channel problem:

```
CurveBuilder* builder = 0 ;
try {
  builder = new CurveBuilder(
                             ns ,
                             nc ,
                             closed ,
                             &lx[ 0 ] ,
                             &ly[ 0 ] ,
                             &ux[ 0 ] ,
                             &uy[ 0 ]
                            ) ;
}
catch ( const ExceptionObject& xpt ) {
  treat_exception( xpt ) ;
  exit( EXIT_FAILURE ) ;
}
```

Once the instance of the channel problem has been created, which is equivalent to saying that an object of `class` CurveBuilder has been instantiated, we can ask the object to solve the problem, which is done by invoking function `build()` (see section The CurveBuilder class API).

```
int error ;
bool res = builder->build( error ) ;
```

If this function returns `true`, the solver has found an optimal solution for the problem, and thus the code can recover the control points of the resulting spline. Otherwise, the code prints out a message to explain why the solver could not find a solution for the problem. This is done by examining the value of the variable `error` passed to function `build()`. See below:

```
if ( res ) {
  string fn2( argv[ 2 ] ) ;
```

```
  write_solution( fn2 , *builder ) ;
}
else {
  cerr << endl
       << "ATTENTION: "
       << endl
       << builder->get_solver_error_message( error )
       << endl
       << endl ;
}
```

Function `get_solver_error_message()` from the API of class `CurveBuilder` is invoked when the solver cannot find a solution for the given instance of the channel problem. The GNU GLPK solver returns an error code that allows us to know why the solver failed. When given this code, function `get_solver_error_message()` simply compares it with all error codes provided by the GLPK, and then returns a message explaining the meaning of the error code.

If a third file name is provided among the command-line arguments, then a description of the linear program corresponding to the given instance of the channel problem is written out to a file using the CPLEX format. As I mentioned before, such an output is only necessary if we want to verify whether my code was able to assemble the correct linear program. Another possible use for it is when the GNU GLPK solver is not able to find a solution. We can then give the linear program to another solver or to the `glpsol` function of the GNU GLPK to obtain more information on why the problem could not be solved. It might be the case that additional information can actually tell us the exact point of the channel that caused infeasibility of the problem.

```
if ( argc == 4 ) {
  string fn3( argv[ 3 ] ) ;
  write_lp( fn3 , *builder ) ;
}
```

The remaining of the `main()` function just releases memory:

```
if ( lx != 0 ) delete[ ] lx ;
if ( ly != 0 ) delete[ ] ly ;
if ( ux != 0 ) delete[ ] ux ;
if ( uy != 0 ) delete[ ] uy ;
if ( builder != 0 ) delete builder ;

return EXIT_SUCCESS ;
```

The auxiliary functions of the program are `read_input()`, `write_solution()`, and `write_lp()`. I will only comment on the code of the second function.

Function `write_solution()` must obtain the control points of the resulting spline in order to write them out to a file. This is done by invoking function `get_control_point()` of class `CurveBuilder` as explained in section The CurveBuilder class API. Below is the body of `write_solution()`:

```
using std::endl ;
std::ofstream ou( fn.c_str() ) ;
if ( ou.is_open() ) {
  ou << std::setprecision( 6 ) << std::fixed ;
  const size_t NumberOfControlPoints = b.get_number_of_control_points() ;
  ou << NumberOfControlPoints
     << '\t'
     << 3
     << endl ;

  for ( size_t i = 0 ; i < NumberOfControlPoints ; i++ ) {
    double x ;
    double y ;
    try {
      x = b.get_control_value( i , 0 ) ;
      y = b.get_control_value( i , 1 ) ;
    }
    catch ( const ExceptionObject& xpt ) {
      treat_exception( xpt ) ;
      ou.close() ;
```

```
      exit( EXIT_FAILURE ) ;
    }
    ou << x << '\t' << y << endl ;
  }
  ou.close() ;
}
```

Chapter 6

Examples and file formats

To solve the channel problem using the `main()` function I described in Section Using the library API, you must give the function a .chn file. This file must contain the complete information about one particular instance of the channel problem. The *first line* of the file contains the values of the input parameters

ns nc closed

in this order, where *ns* is the number of curve segments of the entire b-spline curve, *nc* is the number of c-segments of the channel, and *closed* is a flag to indicate whether the channel is open or closed. See section Introduction for a detailed description of the above parameters. After the first line, there are $2 \times nn$ lines, where $nn = nc + 1$ if the curve is open, and $nn = nc$ if the curve is closed. Each line contains the first and second Cartesian coordinates of a vertex of the lower envelope of the channel:

lx$[0]$ *ly*$[0]$
lx$[1]$ *ly*$[1]$
...
lx$[nn-1]$ *ly*$[nn-1]$

Recall that the coordinates must be given in the same order their corresponding vertices appear in a counterclockwise traversal of the "outside" of the lower envelope, from one extreme of the channel to the other. Right after the coordinates of the vertices of the lower envelope, the coordinates of the vertices of the upper envelope are listed using the same rules:

ux$[0]$ *uy*$[0]$
ux$[1]$ *uy*$[1]$
...
ux$[nn-1]$ *uy*$[nn-1]$

Recall also that $(lx[i], ly[i])$ and $(ux[i], uy[i])$ must be coordinates of the corresponding vertices of the lower and upper envelopes, respectively.

Here is an example of a typical .chn file:

```
9       3       0
639.130835      36.518734
632.034992      36.165892
634.138728      31.121699
639.338308      29.430348
638.869165      38.481266
630.965008      36.834108
632.861272      29.878301
638.661692      27.569652
```

The above file describes the *open* channel

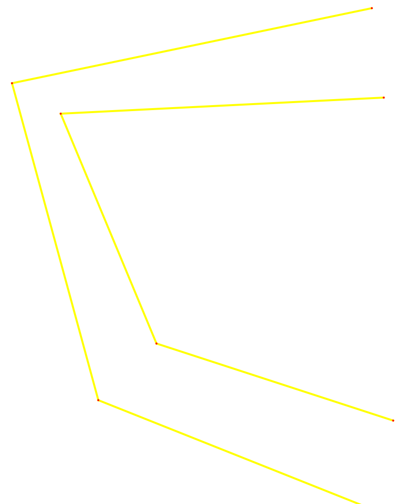


Figure 6.1: Example of a channel

and asks for a b-spline of degree 3 consisting of $ns = 9$ curve segments. Starting from the first segment, each three consecutive segments are delimited by one c-segment of the channel (i.e., by only one pair of edges). Each envelope of the channel has $nn = 4$ vertices, and the channel is open. Observe that $nn = nc+1$. Function `read_input()` (see section Using the library API) reads in the input .chn file and obtains the values of *ns*, *nc*, *nn*, *lx*, *ly*, *ux*, and *uy*. Once the problem is solved, the program generates an output file with extension .spl. This file contains the Cartesian coordinates of the control points of the entire b-spline curve. The first line of a .spl file specifies the total number of control points and the degree of the spline (which is always 3 equal to), i.e.,

ncp dg

After the first line, there are ncp lines. Each line specifies the pair of Cartesian coordinates of a control point. These coordinates are listed as follows:

$b_{0,x}, b_{0,y}$
$b_{1,x}, b_{1,y}$
$\vdots$
$b_{ncp-1,x}, b_{ncp-1,y}$

where $b_{ncp-1,x}$ and $b_{ncp-1,y}$ are the first and second Cartesian coordinates of the i-th control point of the p-th Bézier curve of resulting spline. Below, you find the .spl file corresponding to the solution of the instance of the channel problem described by the .chn file given above, as well as a plot of the spline and its control points:

```
12      3
641.603639      38.017630
638.973833      37.696253
636.344027      37.374876
633.714221      37.053499
631.084414      36.732122
631.590112      34.617083
632.095810      32.502043
633.577118      30.387004
635.362191      29.695979
637.147265      29.004955
638.932338      28.313930
640.717412      27.622906
```

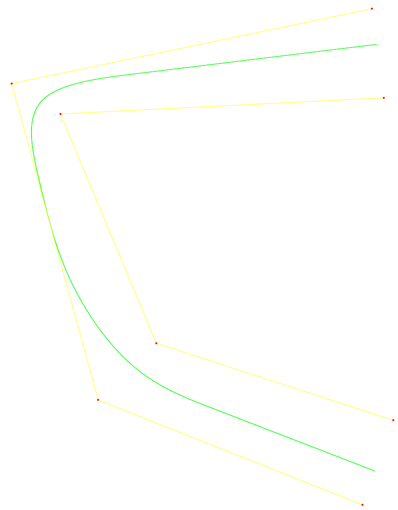


Figure 6.2: Example of a solution for the channel problem

You can find more examples of .chn files in the subdirectory `data/channels`. I wrote a script, `run.sh`, that executes `channel2d` on every input file in subdirectory `data/channels`, and then save the resulting .spl files in subdirectory `data/spcurves`. If your computer runs Mac OSX or a Unix-like system, then you can execute `run.sh`

```
sh run.sh
```

inside subdirectory `scripts`. I didn't provide any GUI to visualize the curves specified by the .spl files. If you decide to write your own .chn file to be tested by my program, execute the line below inside subdirectory `bin`, where the program `channel2d` should be located:

```
channel2d < your input CHN file > < your output SPL file >
```

If you want to see the instance of the linear program assembled by my program and solved by the GLPK solver, execute the line

```
channel2d < your input CHN file > < your output SPL file > < your output LP file >
```

When the execution ends, the third file stores a description of the instance of the linear program using the CPLEX language. Usually, we save such a file with the extension .lp. You can use the function `glpsol` of the GNU GLPK to solve the linear program written in CPLEX language. To that end, execute:

```
glpsol --lp < your LP file >
```

I am assuming that you installed GLPK in your computer and that the path to function `glpsol` is known. By executing `glpsol`, you can compare the solution given by this function with the solution produced by my code. They should be the same! If that is not the case, then I made a mistake when writing the code for generating the CPLEX description of the instance of the linear program that solves the channel problem.

Chapter 7

License

Chapter 8

Acknowledgements

I would like to acknowledge Dr. Jörg Peters for hosting me at CISE-UFL during my sabbatical year (2015-2016), and for patiently helped me understand the papers that underlie the `Channel2D` library code.

Chapter 9

Module Index

9.1 Modules

Here is a list of all modules:

Chapter 10

Namespace Index

10.1 Namespace List

Here is a list of all documented namespaces with brief descriptions:

Chapter 11

Hierarchical Index

11.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

Chapter 12

Class Index

12.1 Class List

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

Chapter 13

File Index

13.1 File List

Here is a list of all documented files with brief descriptions:

Chapter 14

Module Documentation

14.1 Namespace channel.

Namespaces

- channel

  The namespace channel contains the definition and implementation of a set of classes for threading a cubic b-spline curve into a given planar channel delimited by two polygonal chains.

Classes

- class channel::a3

  This class represents two-sided, piecewise linear enclosures for two polynomial functions of degree 3 in Bézier form.
- class channel::Bound

  This class represents the type of a constraint (i.e., equality or inequality) and the value of its right-hand side: a real number.
- class channel::CurveBuilder

  This class provides methods for threading a cubic b-spline curve through a planar channel delimited by a pair of polygonal chains.
- class channel::Coefficient

  This class represents a nonzero coefficient of an unknown of a constraint (inequality or equality) of a linear program instance.
- class channel::ExceptionObject

  This class extends class exception of STL and provides us with a customized way of handling exceptions and showing error messages.
- class channel::TabulatedFunction

  This class represents two-sided, piecewise linear enclosures of a set of $(d-1)$ polynomial functions of degree d in Bézier form. The enclosures must be made available by implementating a pure virtual method in derived classes.

14.1.1 Detailed Description

Chapter 15

Namespace Documentation

15.1 channel Namespace Reference

The namespace channel contains the definition and implementation of a set of classes for threading a cubic b-spline curve into a given planar channel delimited by two polygonal chains.

Classes

- class a3

  This class represents two-sided, piecewise linear enclosures for two polynomial functions of degree 3 in Bézier form.
- class Bound

  This class represents the type of a constraint (i.e., equality or inequality) and the value of its right-hand side: a real number.
- class Coefficient

  This class represents a nonzero coefficient of an unknown of a constraint (inequality or equality) of a linear program instance.
- class CurveBuilder

  This class provides methods for threading a cubic b-spline curve through a planar channel delimited by a pair of polygonal chains.
- class ExceptionObject

  This class extends class exception of STL and provides us with a customized way of handling exceptions and showing error messages.
- class TabulatedFunction

  This class represents two-sided, piecewise linear enclosures of a set of $(d-1)$ polynomial functions of degree d in Bézier form. The enclosures must be made available by implementating a pure virtual method in derived classes.

15.1.1 Detailed Description

The namespace channel contains the definition and implementation of a set of classes for threading a cubic b-spline curve into a given planar channel delimited by two polygonal chains.

Chapter 16

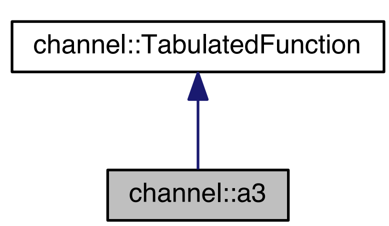
Class Documentation

16.1 channel::a3 Class Reference

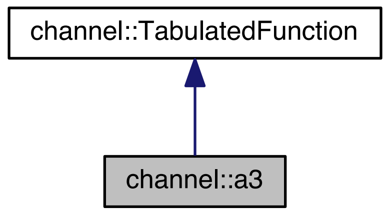
This class represents two-sided, piecewise linear enclosures for two polynomial functions of degree 3 in Bézier form.

`#include <a3.hpp>`

Inheritance diagram for channel::a3:



Collaboration diagram for channel::a3:



Public Member Functions

- a3 ()

  Creates an instance of this class.
- virtual ∼a3 ()

  Releases the memory held by an instance of this class.
- virtual double alower (const size_t i, const double u) const throw (ExceptionObject)

  Evaluates the piecewise linear function corresponding to the lower enclosure of the i-th tabulated function at a point in $[0, 1]$.
- virtual double aupper (const size_t i, const double u) const throw (ExceptionObject)

  Evaluates the piecewise linear function corresponding to the upper enclosure of the i-th tabulated function at a point in $[0, 1]$.
- virtual double a (const size_t i, const double u) const throw (ExceptionObject)

  Computes the value of the i-th polynomial function a at a given point of the interval $[0, 1]$ of the real line.
- virtual unsigned degree () const

  Returns the degree of tabulated functions.

Protected Member Functions

- double a1lower (const double u) const

  Compute the image of a given point of the interval $[0, 1]$ under the lower enclosure of function a_1.
- double a1upper (const double u) const

  Compute the image of a given point of the interval $[0, 1]$ under the upper enclosure of function a_1.
- double a1 (const double u) const

  Computes the value of the cubic polynomial function a_1 at a given point of the interval $[0, 1]$ of the real line.
- double h (const double u) const

  Computes the value of a piecewise linear hat function at a given point of the real line.

Protected Attributes

- double _l0

  1st control value of the lower enclosure of the polynomial a_1.
- double _l1

  2nd control value of the lower enclosure of the polynomial a_1.
- double _l2

  3rd control value of the lower enclosure of the polynomial a_1.
- double _l3

  4th control value of the lower enclosure of the polynomial a_1.

16.1.1 Detailed Description

This class represents two-sided, piecewise linear enclosures for two polynomial functions of degree 3 in Bézier form.

Attention

This class is based on the work described in

```
J. Peters and X. Wu.
On  the  optimality   of  piecewise  linear  max-norm
enclosures  based on  slefes. In  Proceedings of  the
2002 St Malo conference on Curves and Surfaces, 2003.
```

Definition at line 73 of file a3.hpp.

16.1.2 Member Function Documentation

16.1.2.1 double channel::a3::a (const size_t *i*, const double *u*) const throw **ExceptionObject**) [inline], [virtual]

Computes the value of the i-th polynomial function a at a given point of the interval $[0,1]$ of the real line.

Parameters

i	Index of the i-th polynomial function.
u	A parameter point in the interval $[0,1]$.

Returns

The value of the i-th polynomial function a at a given point u of the interval $[0,1]$ of the real line.

Implements channel::TabulatedFunction.

Definition at line 223 of file a3.hpp.

References a1().

```
  {
    if ( ( i != 1 ) && ( i != 2 ) ) {
      std::stringstream ss( std::stringstream::in | std::stringstream::out ) ;
      ss << "Index of the polynomial function is out of range" ;
      throw ExceptionObject( __FILE__ , __LINE__ , ss.str().c_str() ) ;
    }

    if ( ( u < 0 ) || ( u > 1 ) ) {
      std::stringstream ss( std::stringstream::in | std::stringstream::out ) ;
      ss << "Parameter value must belong to the interval [0,1]" ;
```

```
        throw ExceptionObject( __FILE__ , __LINE__ , ss.str().c_str() ) ;
      }

      return ( i == 1 ) ? a1( u ) : a1( 1 - u ) ;
    }
```

16.1.2.2 double channel::a3::a1 (const double *u*) const [inline], [protected]

Computes the value of the cubic polynomial function a_1 at a given point of the interval $[0, 1]$ of the real line.

Parameters

u	A parameter point in the interval $[0, 1]$.

Returns

The value of the cubic polynomial function a_1 at a given point of the interval $[0, 1]$ of the real line.

Definition at line 329 of file a3.hpp.

Referenced by a().

```
    {
#ifdef DEBUGMODE
      assert( u >= 0 ) ;
      assert( u <= 1 ) ;
#endif

      return -u * ( 2 - u * ( 3 - u ) ) ;
    }
```

16.1.2.3 double channel::a3::a1lower (const double *u*) const [inline], [protected]

Compute the image of a given point of the interval $[0, 1]$ under the lower enclosure of function a_1.

Parameters

u	A point in the interval $[0, 1]$.

Returns

The image of a given point of the interval $[0, 1]$ under the lower enclosure of function a_1.

Definition at line 279 of file a3.hpp.

References h().

Referenced by alower().

```
    {
      const double onethird = double( 1 ) / 3 ;

      double res = _l0 * h( u )
                 + _l1 * h( u -     onethird )
                 + _l2 * h( u - 2 * onethird )
                 + _l3 * h( u - 1 ) ;

      return res ;
    }
```

16.1.2.4 double channel::a3::a1upper (const double *u*) const [inline], [protected]

Compute the image of a given point of the interval $[0, 1]$ under the upper enclosure of function a_1.

Parameters

u	A point in the interval $[0,1]$.

Returns

The image of a given point of the interval $[0,1]$ under the upper enclosure of function a_1.

Definition at line 304 of file a3.hpp.

References h().

Referenced by aupper().

```
{
  const double onethird = double( 1 ) / 3 ;

  double res = -10 * h( u - onethird ) - 8 * h( u - 2 * onethird ) ;

  res *= ( double(1) / 27 ) ;

  return res ;
}
```

16.1.2.5 double channel::a3::alower (const size_t *i*, const double *u*) const throw ExceptionObject) [inline], [virtual]

Evaluates the piecewise linear function corresponding to the lower enclosure of the i-th tabulated function at a point in $[0,1]$.

Parameters

i	The index of the i-th polynomial function.
u	A value in the interval $[0,1]$.

Returns

The value of the piecewise linear function corresponding to the lower enclosure of the i-th tabulated function at a point in $[0,1]$.

Implements channel::TabulatedFunction.

Definition at line 147 of file a3.hpp.

References a1lower().

```
{
  if ( ( i != 1 ) && ( i != 2 ) ) {
    std::stringstream ss( std::stringstream::in | std::stringstream::out ) ;
    ss << "Index of the polynomial function is out of range" ;
    throw ExceptionObject( __FILE__ , __LINE__ , ss.str().c_str() ) ;
  }

  if ( ( u < 0 ) || ( u > 1 ) ) {
    std::stringstream ss( std::stringstream::in | std::stringstream::out ) ;
    ss << "Parameter value must belong to the interval [0,1]" ;
    throw ExceptionObject( __FILE__ , __LINE__ , ss.str().c_str() ) ;
  }

  return ( i == 1 ) ? allower( u ) : allower( 1 - u ) ;
}
```

16.1.2.6 double channel::a3::aupper (const size_t *i*, const double *u*) const throw **ExceptionObject**) [inline], [virtual]

Evaluates the piecewise linear function corresponding to the upper enclosure of the i-th tabulated function at a point in $[0, 1]$.

Parameters

i	The index of the i-th polynomial function.
u	A value in the interval $[0, 1]$.

Returns

The value of the piecewise linear function corresponding to the upper enclosure of the i-th tabulated function at a point in $[0, 1]$.

Implements channel::TabulatedFunction.

Definition at line 185 of file a3.hpp.

References a1upper().

```
{
  if ( ( i != 1 ) && ( i != 2 ) ) {
    std::stringstream ss( std::stringstream::in | std::stringstream::out ) ;
    ss << "Index of the polynomial function is out of range" ;
    throw ExceptionObject( __FILE__ , __LINE__ , ss.str().c_str() ) ;
  }

  if ( ( u < 0 ) || ( u > 1 ) ) {
    std::stringstream ss( std::stringstream::in | std::stringstream::out ) ;
    ss << "Parameter value must belong to the interval [0,1]" ;
    throw ExceptionObject( __FILE__ , __LINE__ , ss.str().c_str() ) ;
  }

  return ( i == 1 ) ? a1upper( u ) : a1upper( 1 - u ) ;
}
```

16.1.2.7 unsigned channel::a3::degree ( ) const [inline], [virtual]

Returns the degree of tabulated functions.

Returns

The degree of the tabulated functions.

Implements channel::TabulatedFunction.

Definition at line 253 of file a3.hpp.

```
{
  return 3 ;
}
```

16.1.2.8 double channel::a3::h (const double *u*) const [inline], [protected]

Computes the value of a piecewise linear hat function at a given point of the real line.

Parameters

u	A parameter point of the real line.

Returns

The value of a piecewise linear hat function at a given point of the real line.

Definition at line 352 of file a3.hpp.

Referenced by a1lower(), and a1upper().

```
{
  const double onethird = 1.0 / 3.0 ;

  if ( u <= -onethird ) {
    return 0 ;
  }
  else if ( u <= 0 ) {
    return 3 * u + 1 ;
  }
  else if ( u <= onethird ) {
    return 1 - 3 * u ;
  }

  return 0 ;
}
```

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

- a3.hpp

16.2 channel::Bound Class Reference

This class represents the type of a constraint (i.e., equality or inequality) and the value of its right-hand side: a real number.

`#include <bound.hpp>`

Public Types

- enum CONSTRAINTYPE { **EQT**, **LTE**, **GTE** }

  Defines a type for the type of a constraint.

Public Member Functions

- Bound ()

  Creates an instance of this class.
- Bound (const CONSTRAINTYPE type, const double value, const size_t row)

  Creates an instance of this class.
- Bound (const Bound &b)

  Creates an instance of this class from another instance of this class.
- ∼Bound ()

  Releases the memory held by an instance of this class.
- CONSTRAINTYPE get_type () const

  Returns the type of the constraint associated with this bound.
- double get_value () const

  Returns the value of this bound.
- size_t get_row () const

  Returns the identifier of the constraint associated with this bound. This identifier corresponds to the number of a row in the coefficient matrix associated with of a linear program instance.

Protected Attributes

- CONSTRAINTYPE _ctype

    The type of the constraint associated with this bound.
- double _value

    The bound value.
- size_t _row

    The identifier of the constraint associated with this bound.

16.2.1 Detailed Description

This class represents the type of a constraint (i.e., equality or inequality) and the value of its right-hand side: a real number.

Definition at line 58 of file bound.hpp.

16.2.2 Constructor & Destructor Documentation

16.2.2.1 channel::Bound::Bound (const CONSTRAINTYPE *type*, const double *value*, const size_t *row*) [inline]

Creates an instance of this class.

Parameters

type	The type of the constraint associated with this bound.
value	The value of the bound.
row	The identifier of the constraint associated with this bound.

Definition at line 124 of file bound.hpp.

```
    :
      _ctype( type ) ,
      _value( value ) ,
      _row( row )
    {
    }
```

16.2.2.2 channel::Bound::Bound (const Bound & *b*) [inline]

Creates an instance of this class from another instance of this class.

Parameters

b	An instance of this class.

Definition at line 142 of file bound.hpp.

```
    :
      _ctype( b._ctype ) ,
      _value( b._value ) ,
      _row( b._row )
    {
    }
```

16.2.3 Member Function Documentation

16.2.3.1 size_t channel::Bound::get_row ( ) const [inline]

Returns the identifier of the constraint associated with this bound. This identifier corresponds to the number of a row in the coefficient matrix associated with of a linear program instance.

Returns

The identifier of the constraint associated with this bound.

Definition at line 204 of file bound.hpp.

References _row.

```
{
  return _row ;
}
```

16.2.3.2 CONSTRAINTYPE channel::Bound::get_type ( ) const [inline]

Returns the type of the constraint associated with this bound.

Returns

The type of the constraint associated with this bound.

Definition at line 171 of file bound.hpp.

References _ctype.

```
{
  return _ctype ;
}
```

16.2.3.3 double channel::Bound::get_value ( ) const [inline]

Returns the value of this bound.

Returns

The value of this bound.

Definition at line 185 of file bound.hpp.

References _value.

```
{
  return _value ;
}
```

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

- bound.hpp

16.3 channel::Coefficient Class Reference

This class represents a nonzero coefficient of an unknown of a constraint (inequality or equality) of a linear program instance.

`#include <coefficient.hpp>`

Public Member Functions

- Coefficient ()

  Creates an instance of this class.
- Coefficient (const size_t row, const size_t col, double value)

  Creates an instance of this class.
- Coefficient (const Coefficient &c)

  Creates an instance of this class from another instance of this class.
- ∼Coefficient ()

  Releases the memory held by an instance of this class.
- size_t get_row () const

  Returns the identifier of the constraint the coefficient is associated with. This identifier corresponds to the number of a row in the constraint coefficient matrix of a linear program.
- size_t get_col () const

  Returns the identifier of the unknown multiplied by this coefficient in a constraint of a linear program instance. This identifier corresponds to the number of a column in the coefficient matrix of the linear program instance.
- double get_value () const

  Returns the value of this coefficient.

Protected Attributes

- size_t _row

  The identifier of the constraint this coefficient belongs to.
- size_t _col

  The identifier of the unknown multiplied by this coefficient.
- double _value

  The coefficient value.

16.3.1 Detailed Description

This class represents a nonzero coefficient of an unknown of a constraint (inequality or equality) of a linear program instance.

Definition at line 59 of file coefficient.hpp.

16.3.2 Constructor & Destructor Documentation

16.3.2.1 channel::Coefficient::Coefficient (const size_t *row,* const size_t *col,* double *value*) `[inline]`

Creates an instance of this class.

Parameters

row	The identifier of the constraint this coefficient belongs to.
col	The identifier of the unknown multiplied by this coefficient.
value	The value of the coefficient.

Definition at line 108 of file coefficient.hpp.

```
    :
      _row( row ) ,
      _col( col ) ,
      _value( value )
    {
    }
```

16.3.2.2 channel::Coefficient::Coefficient (const Coefficient & *c*) [inline]

Creates an instance of this class from another instance of this class.

Parameters

c	An instance of this class.

Definition at line 126 of file coefficient.hpp.

```
    :
      _row( c._row ) ,
      _col( c._col ) ,
      _value( c._value )
    {
    }
```

16.3.3 Member Function Documentation

16.3.3.1 size_t channel::Coefficient::get_col ( ) const [inline]

Returns the identifier of the unknown multiplied by this coefficient in a constraint of a linear program instance. This identifier corresponds to the number of a column in the coefficient matrix of the linear program instance.

Returns

The identifier of the unknown multiplied by this coefficient in a constraint of a linear program instance.

Definition at line 178 of file coefficient.hpp.

References _col.

```
    {
      return _col ;
    }
```

16.3.3.2 size_t channel::Coefficient::get_row ( ) const [inline]

Returns the identifier of the constraint the coefficient is associated with. This identifier corresponds to the number of a row in the constraint coefficient matrix of a linear program.

Returns

The identifier of the constraint this coefficient is associated with.

Definition at line 159 of file coefficient.hpp.

References _row.

```
{
  return _row ;
}
```

16.3.3.3 double channel::Coefficient::get_value ( ) const [inline]

Returns the value of this coefficient.

Returns

The value of this coefficient.

Definition at line 192 of file coefficient.hpp.

References _value.

```
{
  return _value ;
}
```

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

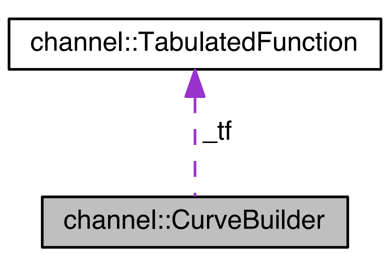
- coefficient.hpp

16.4 channel::CurveBuilder Class Reference

This class provides methods for threading a cubic b-spline curve through a planar channel delimited by a pair of polygonal chains.

`#include <curvebuilder.hpp>`

Collaboration diagram for channel::CurveBuilder:



Public Member Functions

- CurveBuilder (size_t np, size_t nc, bool closed, double *lx, double *ly, double *ux, double *uy) throw (Exception↩ Object)

  Creates an instance of this class.
- CurveBuilder (const CurveBuilder &b)

  Clones an instance of this class.
- ∼CurveBuilder ()

  Releases the memory held by an instance of this class.
- bool build (int &error)

  Solves the channel problem by solving a linear program.
- size_t get_degree () const

  Returns the degree of the bspline curve.
- size_t get_number_of_segments () const

  Returns the number of b-spline segments.
- size_t get_number_of_csegments () const

  Returns the number of c-segments of the channel.
- bool is_curve_closed () const

  Returns the logic value true if the b-spline curve is closed, and the logic value false otherwise.
- size_t get_number_of_control_points () const

  Returns the number of control points of the b-spline.
- size_t get_number_of_constraints () const throw (ExceptionObject)

  Returns the number of constraints of the instance of the linear program corresponding to the channel problem solved by this class.
- double get_control_value (const size_t i, const size_t v) const throw (ExceptionObject)

  Returns the v-th coordinate of the i-th control point of the b-spline curve threaded into the channel.
- size_t get_number_of_coefficients_in_the_ith_constraint (const size_t i) const throw (ExceptionObject)

  Returns the number of coefficients of the i-th constraint of the instance of the linear program.
- size_t get_coefficient_identifier (const size_t i, const size_t j) const throw (ExceptionObject)

  Returns the index of the column that corresponds to the j-th coefficient of the i-th constraint in the matrix associated with the linear program (LP) instance.
- double get_coefficient_value (const size_t i, const size_t j) const throw (ExceptionObject)

  Returns the (i, j) entry of the matrix associated with the instance of the linear program.
- double get_bound_of_ith_constraint (const size_t i) const throw (ExceptionObject)

  Returns the real value on the right-hand side of the equality or inequality corresponding to the i-th constraint.
- bool is_equality (const size_t i) const throw (ExceptionObject)

  Returns the logic value true if the type of the i-th constraint is equality; otherwise, returns the logic value false.
- bool is_greater_than_or_equal_to (const size_t i) const throw (ExceptionObject)

  Returns the logic value true if the i-th constraint is an inequality of the type greater than or equal to; otherwise, returns the logic value false.
- bool is_less_than_or_equal_to (const size_t i) const throw (ExceptionObject)

  Returns the logic value true if the i-th constraint is an inequality of the type less than or equal to; otherwise, returns the logic value false.
- double get_lower_bound_on_second_difference_value (const size_t p, const size_t i, const size_t v) const throw (ExceptionObject)

  Returns the lower bound (found by the LP solver) on the v-th coordinate of the i-th second difference of the i-th curve segment of the b-spline curve threaded into the channel.
- double get_upper_bound_on_second_difference_value (const size_t p, const size_t i, const size_t v) const throw (ExceptionObject)

Returns the upper bound (found by the LP solver) on the v-th coordinate of the i-th second difference vector of the p-th curve segment of the b-spline curve threaded into the channel.

- double minimum_value () const

  Returns the optimal (minimum) value of the objective function of the instance of the channel problem as found by the LP solver.
- std::string get_solver_error_message (int error)

  Returns the error message of the GLPK solver associated with a given error code.

Private Member Functions

- void compute_min_max_constraints (size_t &eqline)

  Computes the equations defining the min-max constraints.
- void compute_correspondence_constraints (size_t &eqline)

  Computes the equations defining the constraints on the location of the endpoints of the b-spline curve threaded into the channel.
- void compute_sleeve_corners_in_channel_constraints (size_t &eqline)

  Computes the equations defining the constraints that ensure that the breakpoints of the sleeves are inside the channel.
- void compute_channel_corners_outside_sleeve_constraints (size_t &eqline)

  Computes the equations defining the constraints that ensure that the corners of the channel are located on the boundary or outside the sleeve.
- void compute_sleeve_inside_csegment_constraints (size_t &eqline)

  Computes the equations defining the constraints that ensure the bspline segments associated with a c-segment remain inside it.
- void compute_normal_to_lower_envelope (const size_t s, double &nx, double &ny) const

  Computes an outward normal to the s-th line segment of the lower envelope of the channel.
- void compute_normal_to_upper_envelope (const size_t s, double &nx, double &ny) const

  Computes an outward normal to the s-th line segment of the upper envelope of the channel.
- void compute_normal_to_csection (const size_t s, double &nx, double &ny) const

  Computes a normal to the s-th c-section of the channel.
- size_t compute_control_value_column_index (const size_t p, const size_t i, const size_t v) const

  Computes the index of the linear program matrix column corresponding to the x- or y-coordinate of the i-th control point of the p-th segment of the b-spline to be threaded into the channel.
- void insert_coefficient (const size_t eqline, const size_t index, const double value)

  Assigns a value to the coefficient of an unknown of a given constraint of the linear program (LP). The unknown is identified by its corresponding column index in the associated matrix of the LP.
- void insert_bound (const size_t eqline, const Bound::CONSTRAINTYPE type, const double value)

  Assigns a real value to the right-hand side of a constraint (equality or inequality) of an instance of the linear program associated with the channel problem.
- size_t compute_second_difference_column_index (const size_t p, const size_t i, const size_t l, const size_t v) const

  Computes the index of the linear program matrix column corresponding to the x- or y-coordinate of the l-th bound of the i-th second difference of the p-th segment of the b-spline to be threaded into the channel.
- size_t compute_index_of_endpoint_barycentric_coordinate (const size_t i) const

  Computes the index of the linear program matrix column corresponding to the barycentric coordinate defining the i-th endpoint of the b-spline.
- size_t compute_index_of_corner_barycentric_coordinate (const size_t i) const

  Computes the index of the linear program matrix column corresponding to the barycentric coordinate associated with a channel corner.
- void insert_min_max_constraints (const size_t eqline, const size_t lo, const size_t up, const size_t b0, const size_t b1, const size_t b2)

Inserts the coefficients of the equations defining the three min-max constraints into the matrix associated with the linear program (LP), and sets the right-hand side of the constraints as well.

- void insert_extreme_point_correspondence_constraint (const size_t eqline, const std::vector< size_t > &col, const std::vector< double > &val, const double rhs)

  Inserts into the linear program (LP) matrix the coefficients of the unknowns and the right-hand side value of a constraint corresponding to the location of the starting or final point of the b-spline curve.
- void insert_periodic_correspondence_constraints (const size_t eqline, const std::vector< size_t > &strx, const std::vector< size_t > &stry, const std::vector< size_t > &endx, const std::vector< size_t > &endy)

  Inserts into the linear program (LP) matrix the coefficients of the unknowns and the right-hand side values of the constraints that ensure that the first three control points are the same as the last three control points (in this order).
- void insert_nonlinear_terms_of_epiece_point_lower_bound (const size_t eqline, const double s, const size_t c, const std::vector< std::vector< std::vector< size_t > > > &sd, const std::vector< std::vector< double > > &nl, const std::vector< std::vector< double > > &nu)

  Inserts the coefficients of the second difference terms of the equation defining lower bounds for the e-piece points into the matrix associated with an instance of the Linear Program (LP). The terms belong to the constraint that forces the e-piece points to be inside a certain c-section of the channel.
- void insert_nonlinear_terms_of_epiece_point_lower_bound (const size_t eqline, const double s, const size_t c, const std::vector< std::vector< std::vector< size_t > > > &sd, const std::vector< std::vector< double > > &ncsec)

  Inserts the coefficients of the second difference terms of the equation defining lower bounds for the e-piece points into the matrix associated with an instance of the Linear Program (LP). The terms belong to the constraint that forces one e-piece point to be on the right or left side of a channel c-section.
- void insert_nonlinear_terms_of_epiece_point_upper_bound (const size_t eqline, const double s, const size_t c, const std::vector< std::vector< std::vector< size_t > > > &sd, const std::vector< std::vector< double > > &nl, const std::vector< std::vector< double > > &nu)

  Inserts into the matrix associated with the Linear Program (LP) the coefficients of the lower and upper bounds of the second difference terms of the equation defining upper bounds for the e-piece points. These terms occur in the constraint that keep the sleeve inside a certain c-section of the channel.
- void insert_nonlinear_terms_of_epiece_point_upper_bound (const size_t eqline, const double s, const size_t c, const std::vector< std::vector< std::vector< size_t > > > &sd, const std::vector< std::vector< double > > &ncsec)

  Inserts the coefficients of the second difference terms of the equation defining upper bounds for the e-piece points into the matrix associated with an instance of the Linear Program (LP). The terms belong to the constraint that forces one e-piece point to be on the right or left side of a channel c-section.
- void insert_linear_terms_of_epiece_point_bounds (const size_t eqline, const double s, const double t, const size_t p, const size_t c, const std::vector< std::vector< size_t > > &cp, const std::vector< std::vector< double > > &nl, const std::vector< std::vector< double > > &nu)

  Inserts the coefficients of the linear terms of the equation defining lower and upper bounds for the e-piece points into the matrix associated with the Linear Program (LP). These terms occur in the constraint that enforces an e-piece point to stay inside channel.
- void insert_linear_terms_of_epiece_point_bounds (const size_t eqline, const double s, const double t, const size_t p, const size_t c, const std::vector< std::vector< size_t > > &cp, const std::vector< std::vector< double > > &ncsec)

  Inserts the coefficients of the linear terms of the equation defining lower and upper bounds for the e-piece points into the matrix associated with the Linear Program (LP). These terms occur in the constraint that enforces an e-piece point to stay either on the right or on left side of a channel c-section.
- void insert_rhs_of_sleeve_corners_in_channel_constraints (const size_t eqline, const size_t c, const std::vector< std::vector< double > > &nl, const std::vector< std::vector< double > > &nu)

  Inserts into the matrix associated with the Linear Program (LP) the right-hand side values of the constraints that enforce a sleeve point to stay inside a c-segment of the channel. The type of each constraint (equality or inequality: ==, >= or <=) is also set here.
- void insert_rhs_of_sleeve_inside_csegment_constraints (const size_t eqline, const size_t c, const std::vector< std::vector< double > > &ncsec)

Inserts into the matrix associated with the Linear Program (LP) the right-hand side values of the constraints that enforce one e-piece breakpoint to stay on the right side of a c-section of the channel, and another e-piece breakpoint to stay on the left side of the same c-section.

- void evaluate_bounding_polynomial (const size_t j, const double t, double &lower, double &upper)

  Obtains a lower bound and an upper bound for the value of a precomputed, bounding polynomial at a given parameter value.
- void insert_csegment_constraint (const size_t eqline, const double lower, const double upper, const size_t sdlo, const size_t sdup, const double normal)

  Inserts the coefficients of the lower and upper bounds of a constraint second difference term into the matrix associated with an instance of the linear program (LP). The term belongs to the equation defining the upper (or lower) bound of a point of an e-piece. The constraint ensures that the point lies on a specific side of the oriented suppporting line of one of the four line segments delimiting a c-segment of the channel.
- void insert_csegment_constraint (const size_t eqline, const double c0, const double c1, const double c2, const double c3, const size_t b0, const size_t b1, const size_t b2, const size_t b3, const double normal)

  Inserts the coefficients of the linear terms of the upper and lower bounds of an e-piece point equation into the matrix associated with an instance of the linear program (LP). The constraint ensures that the point of the e-piece lies inside a c-segment of the channel.
- void insert_csegment_constraint (const size_t eqline, const double c0, const double c1, const double c2, const double c3, const double c4, const size_t b0, const size_t b1, const size_t b2, const size_t b3, const size_t b4, const double normal)

  Inserts the coefficients of the linear terms of the upper and lower bounds of an e-piece point equation into the matrix associated with an instance of the linear program (LP). This point belongs to an e-piece segment whose endpoints bound b-spline curve points in two distinct, but consecutive curve segments. The constraint ensures that the e-piece point lies inside a c-segment of the channel.
- int solve_lp (const size_t rows, const size_t cols)

  Solves the linear program corresponding to the channel problem.
- void set_up_lp_constraints (glp_prob ∗lp) const

  Assemble the matrix of constraints of the linear program, and define the type (equality or inequality) and bounds on the constraints.
- void set_up_structural_variables (glp_prob ∗lp) const

  Define lower and/or upper bounds on the structural variables of the linear program corresponding to the channel problem.
- void set_up_objective_function (glp_prob ∗lp) const

  Define the objective function of the linear program corresponding to the channel problem, which is a minimization problem.
- void get_lp_solver_result_information (glp_prob ∗lp)

  Obtain the optimal values found by the LP solver for the structural values of the linear programming corresponding to the channel problem.

Private Attributes

- size_t _np

  The number of b-spline segments per channel segment.
- size_t _nc

  The number of segments of the channel.
- bool _closed

  A flag to indicate whether the channel is closed.
- std::vector< double > _lxcoords

  X coordinates of the lower polygonal chain of the channel.
- std::vector< double > _lycoords

  Y coordinates of the lower polygonal chain of the channel.
- std::vector< double > _uxcoords

X coordinates of the upper polygonal chain of the channel.

- std::vector$<$ double $>$ _uycoords

  Y coordinates of the upper polygonal chain of the channel.

- TabulatedFunction $*$ _tf

  A pointer to the lower and upper a functions.

- std::vector$<$ std::vector$<$ Coefficient $>$ $>$ _coefficients

  Coefficients of the constraints of the linear program.

- std::vector$<$ Bound $>$ _bounds

  Type of the constraints and their bounds.

- std::vector$<$ double $>$ _ctrlpts

  X and Y coordinates of the control points of the resulting b-spline.

- std::vector$<$ double $>$ _secdiff

  Lower and upper bounds on the second difference values.

- double _ofvalue

  Optimal value (i.e., minimum) of the objective function.

16.4.1 Detailed Description

This class provides methods for threading a cubic b-spline curve through a planar channel delimited by a pair of polygonal chains.

Attention

This class is based on a particular case (i.e., planar and cubic curves) of the method described by Myles & Peters in

A. Myles and J. Peters, Threading splines through 3d channels Computer-Aided Design, 37(2), 139-148, 2005.

Definition at line 80 of file curvebuilder.hpp.

16.4.2 Constructor & Destructor Documentation

16.4.2.1 channel::CurveBuilder::CurveBuilder (size_t *np*, size_t *nc*, bool *closed*, double $*$ *lx*, double $*$ *ly*, double $*$ *ux*, double $*$ *uy*) throw **ExceptionObject**)

Creates an instance of this class.

Parameters

np	The number of b-spline segments.
nc	The number of c-segments of the channel.
closed	A flag to indicate whether the channel is closed.
lx	A pointer to an array with the x-coordinates of the lower envelope of the channel.
ly	A pointer to an array with the y-coordinates of the lower envelope of the channel.
ux	A pointer to an array with the x-coordinates of the upper envelope of the channel.
uy	A pointer to an array with the y-coordinates of the upper envelope of the channel.

Definition at line 80 of file curvebuilder.cpp.

```
  :
    _np( np ) ,
    _nc( nc ) ,
    _closed ( closed )
```

```
{
  if ( _closed ) {
    if ( _np < 4 ) {
      std::stringstream ss( std::stringstream::in | std::stringstream::out ) ;
      ss << "The number of curve segments must be at least 4 for a closed curve" ;
      throw ExceptionObject( __FILE__ , __LINE__ , ss.str().c_str() ) ;
    }
    if ( _nc < 3 ) {
      std::stringstream ss( std::stringstream::in | std::stringstream::out ) ;
      ss << "The number of segments of a closed channel must be at least 3" ;
      throw ExceptionObject( __FILE__ , __LINE__ , ss.str().c_str() ) ;
    }
  }
  else {
    if ( _np < 1 ) {
      std::stringstream ss( std::stringstream::in | std::stringstream::out ) ;
      ss << "The number of curve segments must be at least 1" ;
      throw ExceptionObject( __FILE__ , __LINE__ , ss.str().c_str() ) ;
    }
    if ( _nc < 1 ) {
      std::stringstream ss( std::stringstream::in | std::stringstream::out ) ;
      ss << "The number of segments of an open channel must be at least 1" ;
      throw ExceptionObject( __FILE__ , __LINE__ , ss.str().c_str() ) ;
    }
  }

  size_t nn = ( _closed ) ? _nc : ( _nc + 1 ) ;

  _lxcoords.resize( nn ) ;
  _lycoords.resize( nn ) ;
  _uxcoords.resize( nn ) ;
  _uycoords.resize( nn ) ;

  for ( unsigned i = 0 ; i < nn ; i++ ) {
    _lxcoords[ i ] = lx[ i ] ;
    _lycoords[ i ] = ly[ i ] ;
    _uxcoords[ i ] = ux[ i ] ;
    _uycoords[ i ] = uy[ i ] ;
  }

  _tf = new a3() ;

  _ofvalue = DBL_MAX ;

  return ;
}
```

16.4.2.2 channel::CurveBuilder::CurveBuilder (const **CurveBuilder** & *b*)

Clones an instance of this class.

Parameters

b	A reference to another instance of this class.

Definition at line 150 of file curvebuilder.cpp.

```
:
  _np( b._np ) ,
  _nc( b._nc ) ,
  _closed( b._closed ) ,
  _lxcoords( b._lxcoords ) ,
  _lycoords( b._lycoords ) ,
  _uxcoords( b._uxcoords ) ,
  _uycoords( b._uycoords ) ,
  _tf( b._tf ) ,
  _coefficients( b._coefficients ) ,
  _bounds( b._bounds ) ,
  _ctrlpts( b._ctrlpts ) ,
  _secdiff( b._secdiff ) ,
  _ofvalue( b._ofvalue )
{
}
```

16.4.3 Member Function Documentation

16.4.3.1 bool channel::CurveBuilder::build (int & *error*)

Solves the channel problem by solving a linear program.

Parameters

error	Code returned by the LP solver whenever a solution could not be found. If a solution is found, this parameter is ignored.

Returns

The logic value true if the LP solver is able to find an optimal solution for the channel problem; otherwise, the logic value false is returned.

Definition at line 195 of file curvebuilder.cpp.

References _bounds, _closed, _coefficients, _nc, _np, compute_channel_corners_outside_sleeve_constraints(), compute_correspondence_constraints(), compute_min_max_constraints(), compute_sleeve_corners_in_channel_constraints(), compute_sleeve_inside_csegment_constraints(), and solve_lp().

Referenced by main().

```
{
  // Compute the number  of linear constraints (i.e.,  the number of
  // rows of the matrix) of the linear program whose solution yields
  // the curve.
  size_t rows = (
                      ( 6 * ( _np + 1 ) )          // min-max
                    + ( ( _closed ) ? 8 : 4 )      // correspondence
                    + ( 2 * ( _nc - 1 ) )          // channel corners
                    + ( ( 3 * 4 * _np ) - 4 )      // sleeve corners
                                + ( 4 * ( _nc - 1 ) )          // sleeve in csegments
                   ) ;

  // Compute  the  unknowns (i.e.,  the  number  of columns  of  the
  // matrix)  of  the  linear  program  whose  solution  yields  the
  // b-spline curve.
  size_t cols = ( 6 * _np ) + 10 + ( ( _closed ) ? 1 : 2 ) + ( _nc - 1 ) ;

  //
  // Allocate memory for the array of coefficients and bounds.
  //
  _coefficients.resize( rows ) ;
  _bounds.resize( rows ) ;

  //
  // Initialize the equation counter.
  //
  size_t eqline = 0 ;

  //
  // Compute the min-max constraints.
  //
  compute_min_max_constraints( eqline ) ;

  //
  // Compute the correspondence constraints.
  //
  compute_correspondence_constraints( eqline ) ;

  //
  // Compute channel corners outside sleeve constraints.
  //
  compute_channel_corners_outside_sleeve_constraints(
eqline ) ;

  //
  // Compute the sleeve corners in channel constraints.
  //
  compute_sleeve_corners_in_channel_constraints( eqline ) ;
```

```
    //
    // Compute the sleeve inside csegment constraints.
    //
    compute_sleeve_inside_csegment_constraints( eqline ) ;

    //
    // Solve the LP and get the solution.
    //
    error = solve_lp( rows , cols ) ;

    return ( error == 0 ) ;
  }
```

16.4.3.2 void channel::CurveBuilder::compute_channel_corners_outside_sleeve_constraints (size_t & *eqline*) [private]

Computes the equations defining the constraints that ensure that the corners of the channel are located on the boundary or outside the sleeve.

Parameters

eqline	A reference to the counter of equations.

Definition at line 730 of file curvebuilder.cpp.

References _lxcoords, _lycoords, _nc, _np, _uxcoords, _uycoords, compute_control_value_column_index(), compute↩ _index_of_corner_barycentric_coordinate(), insert_bound(), and insert_coefficient().

Referenced by build().

```
  {
    //
    // This restriction applies to channels with at least 3 c-sections
    // only.
    //
    if ( _nc < 2 ) {
      return ;
    }

    // For  each  inner  corner  of   the  given  channel,  compute  a
    // constraint that ensures that the  channel corner is outside the
    // sleeve.
    const double NpOverNc = _np / double( _nc ) ;
    const double onesixth = 1 / double( 6 ) ;

    for ( size_t c = 1 ; c < _nc ; c++ ) {
      //
      // Find the parameter \e t corresponding to the \e c corner.
      //
      double t = ( c * NpOverNc ) + 3 ;

      //
      // Find the curve segment \e  p containing point at parameter \e
      // t.
      size_t p = ( size_t ) floor( t - 3 ) ;

      // Compute  the  column indices  of  the  linear program  matrix
      // corresponding to  the four  control points defining  the p-th
      // segment of the b-spline curve.

      std::vector< std::vector< size_t > > cp( 4 , std::vector< size_t >( 2 , 0 ) ) ;

      for ( size_t i = 0 ; i < 4 ; i++ ) {
        for  ( size_t j = 0 ; j < 2 ; j++ ) {
          cp[ i ][ j ] = compute_control_value_column_index( p , i , j )
  ;
        }
      }

      //
      // Compute the coefficients of the control points.
      //
      double s = t - floor( t ) ;
```

```
    std::vector< double > coeffs( 4 ) ;

    coeffs[ 0 ] = onesixth * ( 1 + s * ( -3 + s * ( 3 - s ) ) ) ;
    coeffs[ 1 ] = onesixth * ( 4 + s * s * ( - 6 + 3 * s ) ) ;
    coeffs[ 2 ] = onesixth * ( 1 + s * ( 3 + s * ( 3 - 3 * s ) ) ) ;
    coeffs[ 3 ] = onesixth * ( s * s * s ) ;

    // Get  the   LP  matrix  column  index   corresponding  to  the
    // barycentric coordinate associated with the c-th corner of the
    // channel.
    size_t k = compute_index_of_corner_barycentric_coordinate
( c ) ;

    //
    // Insert the constraints into the LP program.
    //
    for ( size_t i = 0 ; i < 4 ; i++ ) {
      insert_coefficient( eqline     , cp[ i ][ 0 ] , coeffs[ i ] ) ;
      insert_coefficient( eqline + 1 , cp[ i ][ 1 ] , coeffs[ i ] ) ;
    }

    insert_coefficient( eqline     , k , ( _lxcoords[ c ] -
_uxcoords[ c ] ) ) ;
    insert_coefficient( eqline + 1 , k , ( _lycoords[ c ] -
_uycoords[ c ] ) ) ;

    insert_bound( eqline     , Bound::EQT , _lxcoords[ c ] ) ;
    insert_bound( eqline + 1 , Bound::EQT , _lycoords[ c ] ) ;

    //
    // Increment equation counter.
    //
    eqline += 2 ;
  }

  return ;
}
```

16.4.3.3 size_t channel::CurveBuilder::compute_control_value_column_index (const size_t *p*, const size_t *i*, const size_t *v*) const [private]

Computes the index of the linear program matrix column corresponding to the x- or y-coordinate of the i-th control point of the p-th segment of the b-spline to be threaded into the channel.

Parameters

p	Index of the b-spline segment.
i	Index of a control point of the p-th b-spline segment.
v	Index of the x- or y-coordinate of the control point.

Returns

The index of the linear program matrix column corresponding to the x- or y-coordinate of the i-th control point of the p-th segment of the b-spline to be threaded into the channel.

Definition at line 1215 of file curvebuilder.cpp.

References _np.

Referenced by compute_channel_corners_outside_sleeve_constraints(), compute_correspondence_constraints(), compute_min_max_constraints(), compute_sleeve_corners_in_channel_constraints(), compute_sleeve_inside_csegment_constraints(), get_lp_solver_result_information(), and set_up_structural_variables().

```
  {
#ifdef DEBUGMODE
    assert( p < _np ) ;
    assert( i <=  3 ) ;
    assert( v <=  1 ) ;
```

```
#endif

    return 2 * ( p + i ) + v ;
  }
```

16.4.3.4 void channel::CurveBuilder::compute_correspondence_constraints (size_t & *eqline*) [private]

Computes the equations defining the constraints on the location of the endpoints of the b-spline curve threaded into the channel.

Parameters

eqline	A reference to the counter of equations.

Definition at line 406 of file curvebuilder.cpp.

References _closed, _lxcoords, _lycoords, _nc, _np, _uxcoords, _uycoords, compute_control_value_column_index(), compute_index_of_endpoint_barycentric_coordinate(), insert_extreme_point_correspondence_constraint(), and insert_periodic_correspondence_constraints().

Referenced by build().

```
  {
    // Get the column  index of the x- and y-coordinates  of the first
    // three control  points of the  b-spline to be threaded  into the
    // channel.

    std::vector< size_t > strx( 4 ) ;

    strx[ 0 ] = compute_control_value_column_index( 0 , 0 , 0 ) ;
    strx[ 1 ] = compute_control_value_column_index( 0 , 1 , 0 ) ;
    strx[ 2 ] = compute_control_value_column_index( 0 , 2 , 0 ) ;

    // Get  the column  index of  the barycentric  coordinate defining
    // the first  endpoint of the  b-spline with respect to  the first
    // channel points.

    strx[ 3 ] = compute_index_of_endpoint_barycentric_coordinate
  ( 0 ) ;

    //
    // Get the coefficients of the unknowns of the constraint.
    //
    std::vector< double > vals( 4 ) ;

    vals[ 0 ] = double( 1 ) / double( 6 ) ;
    vals[ 1 ] = double( 2 ) / double( 3 ) ;
    vals[ 2 ] = vals[ 0 ] ;
    vals[ 3 ] = _lxcoords[ 0 ] - _uxcoords[ 0 ]   ;

    //
    // Constraint corresponding to first Cartesian coordinate.
    //
    insert_extreme_point_correspondence_constraint(
                                                 eqline ,
                                                 strx ,
                                                 vals ,
                                                 _lxcoords[ 0 ]
                                                ) ;

    ++eqline ;  // increment the equation counter.

    //
    // Constraint corresponding to second Cartesian coordinate.
    //

    std::vector< size_t > stry( 4 ) ;

    stry[ 0 ] = compute_control_value_column_index( 0 , 0 , 1 ) ;
    stry[ 1 ] = compute_control_value_column_index( 0 , 1 , 1 ) ;
    stry[ 2 ] = compute_control_value_column_index( 0 , 2 , 1 ) ;
    stry[ 3 ] = strx[ 3 ] ;

```

```
    vals[ 3 ] = _lycoords[ 0 ] - _uycoords[ 0 ]  ;

    insert_extreme_point_correspondence_constraint(
                                                   eqline ,
                                                   stry ,
                                                   vals ,
                                                   _lycoords[ 0 ]
                                                  ) ;

    ++eqline ;  // increment the equation counter.

    // -------------------------------------------------------------
    //
    // If the curve is closed, then the last three control points must
    // match the  first three control  points. Otherwise, we  must fix
    // the position of the final of  the curve, which differs from the
    // starting one.
    //
    // -------------------------------------------------------------

    // Get the  column index of the  x- and y-coordinates of  the last
    // three control  points of the  b-spline to be threaded  into the
    // channel.

    std::vector< size_t > endx( 4 ) ;

    endx[ 0 ] = compute_control_value_column_index(
  _np - 1 , 1 , 0 ) ;
    endx[ 1 ] = compute_control_value_column_index(
  _np - 1 , 2 , 0 ) ;
    endx[ 2 ] = compute_control_value_column_index(
  _np - 1 , 3 , 0 ) ;

    std::vector< size_t > endy( 4 ) ;

    endy[ 0 ] = compute_control_value_column_index(
  _np - 1 , 1 , 1 ) ;
    endy[ 1 ] = compute_control_value_column_index(
  _np - 1 , 2 , 1 ) ;
    endy[ 2 ] = compute_control_value_column_index(
  _np - 1 , 3 , 1 ) ;

    if ( _closed ) {
      //
      // Compute the  equations that  match the  first three  and last
      // three control points of the b-spline: last is equal to third,
      // ...
      //

      insert_periodic_correspondence_constraints(
                                                 eqline ,
                                                 strx ,
                                                 stry ,
                                                 endx ,
                                                 endy
                                                ) ;

      eqline += 6 ;   // increment the equation counter.
    }
    else {
      //
      // Compute the equations determining the b-spline final point.
      //

      // Get the  column index  of the  barycentric coordinate  of the
      // final point of the b-spline  with respect to the final points
      // of the channel.

      endx[ 3 ] = compute_index_of_endpoint_barycentric_coordinate
  ( 1 ) ;
      vals[ 3 ] = _lxcoords[ _nc ] - _uxcoords[ _nc ] ;

      //
      // Constraint corresponding to first Cartesian coordinate.
      //
      insert_extreme_point_correspondence_constraint(
                                                     eqline ,
                                                     endx ,
                                                     vals ,
                                                     _lxcoords[ _nc ]
                                                    ) ;
```

```
      endy[ 3 ] = endx[ 3 ] ;
      vals[ 3 ] = _lycoords[ _nc ] - _uycoords[ _nc ] ;

      ++eqline ;  // increment the equation counter.

      insert_extreme_point_correspondence_constraint(
                                                     eqline ,
                                                     endy ,
                                                     vals ,
                                                     _lycoords[ _nc ]
                                                    ) ;

      ++eqline ;  // increment the equation counter.
    }

    return ;
  }
```

16.4.3.5 size_t channel::CurveBuilder::compute_index_of_corner_barycentric_coordinate (const size_t *i*) const [private]

Computes the index of the linear program matrix column corresponding to the barycentric coordinate associated with a channel corner.

Parameters

i	Index of a channel corner.

Returns

The index of the linear program matrix column corresponding to the barycentric coordinate associated with a channel corner.

Definition at line 1325 of file curvebuilder.cpp.

References _closed, _nc, and _np.

Referenced by compute_channel_corners_outside_sleeve_constraints(), and set_up_structural_variables().

```
  {
#ifdef DEBUGMODE
    assert( i >   0 ) ;
    assert( i < _nc ) ;
#endif

    size_t offset =  ( 6 * _np ) + 10 + ( ( _closed ) ? 0 : 1 ) ;

    return offset + i ;

  }
```

16.4.3.6 size_t channel::CurveBuilder::compute_index_of_endpoint_barycentric_coordinate (const size_t *i*) const [private]

Computes the index of the linear program matrix column corresponding to the barycentric coordinate defining the i-th endpoint of the b-spline.

Parameters

i	Index of the i-th barycentric coordinate.

Returns

The index of the linear program matrix column corresponding to the barycentric coordinate defining the i-th end-point of the b-spline.

Definition at line 1293 of file curvebuilder.cpp.

References _closed, and _np.

Referenced by compute_correspondence_constraints(), and set_up_structural_variables().

```
  {
    #ifdef DEBUGMODE
    if ( _closed ) {
      assert( i == 0 ) ;
    }
    else {
      assert( i <= 1 ) ;
    }
#endif

    size_t offset =  ( 6 * _np ) + 10 ;

    return offset + i ;
  }
```

16.4.3.7 void channel::CurveBuilder::compute_min_max_constraints (size_t & *eqline*) [private]

Computes the equations defining the min-max constraints.

Parameters

eqline	A reference to the counter of equations.

Definition at line 274 of file curvebuilder.cpp.

References _np, compute_control_value_column_index(), compute_second_difference_column_index(), and insert_↩ min_max_constraints().

Referenced by build().

```
 {
   //
   // Obtain the min-max constraints for each second difference.
   //

   for ( size_t j = 1 ; j < 3 ; j++ ) {
     for ( size_t v = 0 ; v < 2 ; v++ ) {
       // Get  the column  indices of  the lower  bound and  of the
       // upper bound  of the  v-th coordinate  of the  j-th second
       // difference.
       size_t jl = compute_second_difference_column_index(
                                                          0 ,
                                                          j ,
                                                          0 ,
                                                          v
                                                         ) ;

       size_t ju = compute_second_difference_column_index(
                                                          0 ,
                                                          j ,
                                                          1 ,
                                                          v
                                                         ) ;

       // Get the column indices of  the v-th coordinates that define
```

```
        // the j-th second difference of the p-th curve segment.
        size_t c1 = compute_control_value_column_index(
                                                       0 ,
                                                       j - 1 ,
                                                       v
                                                      ) ;
        size_t c2 = compute_control_value_column_index(
                                                       0,
                                                       j ,
                                                       v
                                                      ) ;
        size_t c3 = compute_control_value_column_index(
                                                       0 ,
                                                       j + 1 ,
                                                       v
                                                      ) ;

        //
        // Set the nonzero coefficients of the next three equations.
        //
        insert_min_max_constraints(
                                   eqline ,
                                   jl ,
                                   ju ,
                                   c1 ,
                                   c2 ,
                                   c3
                                  ) ;

        //
        // Increment equation counter
        //
        eqline += 3 ;
      }
    }

    for ( size_t p = 1 ; p < _np ; p++ ) {
      for ( size_t v = 0 ; v < 2 ; v++ ) {
        // Get the column indices of the  lower bound and of the upper
        // bound of the v-th coordinate of the 2nd second difference.
        size_t jl = compute_second_difference_column_index(
                                                           p ,
                                                           2 ,
                                                           0 ,
                                                           v
                                                          ) ;

        size_t ju = compute_second_difference_column_index(
                                                           p ,
                                                           2 ,
                                                           1 ,
                                                           v
                                                          ) ;

        // Get the column indices of  the v-th coordinates that define
        // the i-th second difference of the p-th curve segment.
        size_t c1 = compute_control_value_column_index(
                                                       p ,
                                                       1 ,
                                                       v
                                                      ) ;
        size_t c2 = compute_control_value_column_index(
                                                       p ,
                                                       2 ,
                                                       v
                                                      ) ;
        size_t c3 = compute_control_value_column_index(
                                                       p ,
                                                       3 ,
                                                       v
                                                      ) ;

        //
        // Set the nonzero coefficients of the next three equations.
        //
        insert_min_max_constraints(
                                   eqline ,
                                   jl ,
                                   ju ,
                                   c1 ,
                                   c2 ,
```

```
                                    c3
                                  ) ;

        //
        // Increment equation counter
        //
        eqline += 3 ;
      }
    }

    return ;
  }
```

16.4.3.8 void channel::CurveBuilder::compute_normal_to_csection (const size_t *s*, double & *nx*, double & *ny*) const [private]

Computes a normal to the s-th c-section of the channel.

Parameters

s	Index of a c-section of the channel.
nx	A reference to the first Cartesian coordinate of the normal.
ny	A reference to the Second Cartesian coordinate of the normal.

Definition at line 1176 of file curvebuilder.cpp.

References _closed, _lxcoords, _lycoords, _nc, _uxcoords, and _uycoords.

Referenced by compute_sleeve_inside_csegment_constraints().

```
  {
#ifdef DEBUGMODE
    assert( s <= _nc ) ;
#endif

    size_t t = ( _closed ) ? ( s % _nc ) : s ;

    nx = _lycoords[ t ] - _uycoords[ t ] ;
    ny = _uxcoords[ t ] - _lxcoords[ t ] ;

    return ;
  }
```

16.4.3.9 void channel::CurveBuilder::compute_normal_to_lower_envelope (const size_t *s*, double & *nx*, double & *ny*) const [private]

Computes an outward normal to the s-th line segment of the lower envelope of the channel.

Parameters

s	Index of a line segment of the lower channel envelope.
nx	A reference to the first Cartesian coordinate of the normal.
ny	A reference to the second Cartesian coordinate of the normal.

Definition at line 1100 of file curvebuilder.cpp.

References _closed, _lxcoords, _lycoords, and _nc.

Referenced by compute_sleeve_corners_in_channel_constraints().

```
  {
#ifdef DEBUGMODE
    assert( s < _nc ) ;
#endif
```

```
1110
1111     size_t t = s + 1 ;
1112
1113     if ( _closed ) {
1114        t %= _nc ;
1115     }
1116
1117     nx = _lycoords[ s ] - _lycoords[ t ] ;
1118     ny = _lxcoords[ t ] - _lxcoords[ s ] ;
1119
1120     return ;
1121   }
```

16.4.3.10 void channel::CurveBuilder::compute_normal_to_upper_envelope (const size_t *s*, double & *nx*, double & *ny*) const [private]

Computes an outward normal to the s-th line segment of the upper envelope of the channel.

Parameters

s	Index of a line segment of the upper channel envelope.
nx	A reference to the first Cartesian coordinate of the normal.
ny	A reference to the second Cartesian coordinate of the normal.

Definition at line 1138 of file curvebuilder.cpp.

References _closed, _nc, _uxcoords, and _uycoords.

Referenced by compute_sleeve_corners_in_channel_constraints().

```
1144   {
1145 #ifdef DEBUGMODE
1146     assert( s < _nc ) ;
1147 #endif
1148
1149     size_t t = s + 1 ;
1150
1151     if ( _closed ) {
1152       t %= _nc ;
1153     }
1154
1155     nx = _uycoords[ t ] - _uycoords[ s ] ;
1156     ny = _uxcoords[ s ] - _uxcoords[ t ] ;
1157
1158     return ;
1159   }
```

16.4.3.11 size_t channel::CurveBuilder::compute_second_difference_column_index (const size_t *p*, const size_t *i*, const size_t *l*, const size_t *v*) const [private]

Computes the index of the linear program matrix column corresponding to the x- or y-coordinate of the l-th bound of the i-th second difference of the p-th segment of the b-spline to be threaded into the channel.

Computes the index of the linear program matrix column corresponding to the x- or y-coordinate of the l-th bound of the i-th second difference of the p-th segment of the b-spline to be threaded into the channel.

Parameters

p	Index of the b-spline segment.

i	Index of the second difference of the p-th b-spline segment.
l	Index of the l-th bound of the second difference (0 - lower bound; 1 - upper bound).
v	Index of the x- or y-coordinate of the second difference bound.

Returns

The index of the linear program matrix column corresponding to the x- or y-coordinate of the l-th bound of the i-th second difference of the p-th segment of the b-spline to be threaded into the channel.

Parameters

p	Index of the b-spline segment.
i	Index of the second difference of the p-th b-spline segment.
l	Index of the l-th bound of the second difference (0 - lower bound; 1 - upper bound).
v	Index of the x- or y-coordinate of the second difference bound.

Returns

The index of the linear program matrix column corresponding to the x- or y-coordinate of the l-th bound of the i-th second difference of the p-th segment of the b-spline to be threaded into the channel.

Definition at line 1256 of file curvebuilder.cpp.

References _np.

Referenced by compute_min_max_constraints(), compute_sleeve_corners_in_channel_constraints(), compute_sleeve_inside_csegment_constraints(), get_lp_solver_result_information(), set_up_objective_function(), and set_up_structural_variables().

```
  {
#ifdef DEBUGMODE
    assert( p < _np ) ;
    assert( i >= 1 ) ;
    assert( i <= 2 ) ;
    assert( l <= 1 ) ;
    assert( v <= 1 ) ;
#endif

    size_t offset = ( 2 * _np ) + 6 ;

    return offset + ( 4 * ( p + i - 1 ) ) + ( 2 * l ) + v ;
  }
```

16.4.3.12 void channel::CurveBuilder::compute_sleeve_corners_in_channel_constraints (size_t & *eqline*) [private]

Computes the equations defining the constraints that ensure that the breakpoints of the sleeves are inside the channel.

Parameters

eqline	A reference to the counter of equations.

Definition at line 562 of file curvebuilder.cpp.

References _nc, _np, compute_control_value_column_index(), compute_normal_to_lower_envelope(), compute_normal_to_upper_envelope(), compute_second_difference_column_index(), insert_linear_terms_of_epiece_point_bounds(), insert_nonlinear_terms_of_epiece_point_lower_bound(), insert_nonlinear_terms_of_epiece_point_upper_bound(), and insert_rhs_of_sleeve_corners_in_channel_constraints().

Referenced by build().

```
{
  //
  // Compute outward normals to the line segments of the channel.
  //
  std::vector< std::vector< double > > nl( _nc , std::vector< double >( 2 , 0 ) ) ;
  std::vector< std::vector< double > > nu( _nc , std::vector< double >( 2 , 0 ) ) ;

  for ( size_t c = 0 ; c < _nc ; c++ ) {
    compute_normal_to_lower_envelope( c , nl[ c ][ 0 ] , nl[ c ][ 1 ] ) ;
    compute_normal_to_upper_envelope( c , nu[ c ][ 0 ] , nu[ c ][ 1 ] ) ;
  }

  // Each segment of the b-spline must  be enclosed by a sleeve with
  // four breakpoints, two of which are shared with the previous and
  // next segment  (if any).  Each  breakpoint is constrained  to be
  // bounded by a pair of parallel segments (lower and upper) of the
  // channel.

  const size_t lo = ( 3 *   3 ) + 1 ;
  const size_t up = ( 3 * _np ) + 8 ;
  const double NcOverNp = _nc / double( _np ) ;

  for ( size_t u = lo ; u <= up ; u++ ) {
    //
    // Find the index of the channel segment corresponding to \e u.
    //
    double t = u / double( 3 ) ;
    double s = t - floor( t ) ;
    size_t p = ( size_t ) floor( t - 3 ) ;
    size_t c = ( size_t ) ( ( t - 3 ) * NcOverNp ) ;

    // Compute  the  column indices  of  the  linear program  matrix
    // corresponding to  the four  control points defining  the p-th
    // segment of the b-spline curve.

    std::vector< std::vector< size_t > > cp( 4 , std::vector< size_t >( 2 , 0 ) ) ;

    for ( size_t i = 0 ; i < 4 ; i++ ) {
      for  ( size_t j = 0 ; j < 2 ; j++ ) {
        cp[ i ][ j ] = compute_control_value_column_index( p , i , j )
;
      }
    }

    // Compute  the  column indices  of  the  linear program  matrix
    // of the values of the second difference bounds associated with
    // the p-th segment of the b-spline curve.

    std::vector< std::vector< std::vector< size_t > > > sd(
                                                          2 ,
                                                          std::vector< std::vector< size_t > >
                                                          (
                                                            2 ,
                                                            std::vector< size_t >( 2 , 0 )
                                                          )
                                                         ) ;

    for ( size_t j = 1 ; j < 3 ; j++ ) {
      for ( size_t l = 0 ; l < 2 ; l++ ) {
        for ( size_t v = 0 ; v < 2 ; v++ ) {
          sd[ j - 1 ][ l ][ v ] = compute_second_difference_column_index
( p , j , l , v ) ;
        }
      }
    }

    // ------------------------------------------------------------
    //
    // Process nonlinear terms of Constraint (3a).
    //
    // ------------------------------------------------------------

    //
    // Nonlinear terms of \f$\stackrel{e}{\sim}^p\f$
    //

    insert_nonlinear_terms_of_epiece_point_lower_bound(
                                                      eqline ,
                                                      s ,
                                                      c ,
                                                      sd ,
```

```
                                                             nl ,
                                                             nu
                                                            ) ;

          //
          // Nonlinear terms of \f$\stackrel{\sim}{e}^p\f$.
          //

          insert_nonlinear_terms_of_epiece_point_upper_bound(
                                                             eqline + 2 ,
                                                             s ,
                                                             c ,
                                                             sd ,
                                                             nl ,
                                                             nu
                                                            ) ;


          // ------------------------------------------------------------
          //
          // Process linear terms of Constraint (3a).
          //
          // ------------------------------------------------------------

          //
          // Linear terms of \f$\stackrel{e}{\sim}^p\f$
          //

          insert_linear_terms_of_epiece_point_bounds(
                                                     eqline ,
                                                     s ,
                                                     t ,
                                                     p ,
                                                     c ,
                                                     cp ,
                                                     nl ,
                                                     nu
                                                    ) ;

          //
          // Linear terms of \f$\stackrel{\sim}{e}^p\f$.
          //

          insert_linear_terms_of_epiece_point_bounds(
                                                     eqline + 2 ,
                                                     s ,
                                                     t ,
                                                     p ,
                                                     c ,
                                                     cp ,
                                                     nl ,
                                                     nu
                                                    ) ;

          // -------------------------------------------------------------
          //
          // Compute right-hand side of the constraints.
          //
          // -------------------------------------------------------------

          insert_rhs_of_sleeve_corners_in_channel_constraints
    (
                                                              eqline ,
                                                              c ,
                                                              nl ,
                                                              nu
                                                             ) ;

          //
          // Increment equation counter.
          //
          eqline += 4 ;
        }

        return ;
      }
```

16.4.3.13 void channel::CurveBuilder::compute_sleeve_inside_csegment_constraints (size_t & *eqline*) [private]

Computes the equations defining the constraints that ensure the bspline segments associated with a c-segment remain inside it.

Parameters

eqline	A reference to the counter of equations.

Definition at line 820 of file curvebuilder.cpp.

References _closed, _nc, _np, compute_control_value_column_index(), compute_normal_to_csection(), compute↩ _second_difference_column_index(), insert_linear_terms_of_epiece_point_bounds(), insert_nonlinear_terms_of_↩ epiece_point_lower_bound(), insert_nonlinear_terms_of_epiece_point_upper_bound(), and insert_rhs_of_sleeve_↩ inside_csegment_constraints().

Referenced by build().

```
{
  //
  // This restriction applies to channels with at least 3 c-sections
  // only.
  //
  if ( _nc < 2 ) {
    return ;
  }

  //
  // Compute normals to the c-sections of the channel.
  //

  const size_t NumberOfCSections = ( _closed ) ? _nc : _nc + 1 ;

  std::vector< std::vector< double > > ncsec(
                                           NumberOfCSections ,
                                           std::vector< double >( 2 , 0 )
                                          ) ;

  for ( size_t c = 0 ; c < NumberOfCSections ; c++ ) {
    compute_normal_to_csection(
                               c ,
                               ncsec[ c ][ 0 ] ,
                               ncsec[ c ][ 1 ]
                              ) ;
  }

  // For  each  inner  corner  of the  given  channel,  compute  two
  // constraints   which  ensure   that   the  e-piece   breakpoints
  // immediately  on the  right  (resp. left)  of the  corresponding
  // c-section remain  in the right  (resp.  left) c-segment  of the
  // channel.

  const double NpOverNc = _np / double( _nc ) ;
  const double onethird = 1 / double( 3 ) ;
  const double twothird = 2 * onethird ;

  for ( size_t c = 1 ; c < _nc ; c++ ) {
    //
    // Find the parameter \e t corresponding to the \e c corner.
    //
    double t = ( c * NpOverNc ) + 3 ;

    //
    // Find the curve segment \e  p containing point at parameter \e
    // t.
    size_t p = ( size_t ) floor( t - 3 ) ;

    // Compute the  indices of  the curve segments  corresponding to
    // the e-piece breakpoints immediately to  the right and left of
    // point \c p(t).

    double s = t - floor( t ) ;

    size_t p1 , p2 ;
    double s1 , s2 ;

    if ( s == 0 ) {
      p1 = p - 1 ;
      p2 = p ;
      s1 = twothird ;
      s2 = onethird ;
    }
    else if ( s < onethird ) {
```

```
        p1 = p ;
        p2 = p ;
        s1 = 0 ;
        s2 = onethird ;
      }
      else if ( s < twothird ) {
        p1 = p ;
        p2 = p ;
        s1 = onethird ;
        s2 = twothird ;
      }
      else {
        p1 = p ;
        p2 = p ;
        s1 = twothird ;
        s2 = 1 ;
      }

      double t1 = p1 + 3 + s1 ;
      double t2 = p2 + 3 + s2 ;

      // Compute the column indices of  the LP matrix corresponding to
      // the  second  difference  bounds  associated  with  the  p1-th
      // segment.

      std::vector< std::vector< std::vector< size_t > > > sd1(
                                                         2 ,
                                                         std::vector< std::vector< size_t > >
                                                         (
                                                          2 ,
                                                          std::vector< size_t >( 2 , 0 )
                                                         )
                                                        ) ;

      for ( size_t j = 1 ; j < 3 ; j++ ) {
        for ( size_t l = 0 ; l < 2 ; l++ ) {
          for ( size_t v = 0 ; v < 2 ; v++ ) {
            sd1[ j - 1 ][ l ][ v ] = compute_second_difference_column_index
(
                                                                        p1 ,
                                                                        j ,
                                                                        l ,
                                                                        v
                                                                       ) ;
          }
        }
      }

      // Compute the column indices of  the LP matrix corresponding to
      // the  second  difference  bounds  associated  with  the  p2-th
      // segment.

      std::vector< std::vector< std::vector< size_t > > > sd2(
                                                         2 ,
                                                         std::vector< std::vector< size_t > >
                                                         (
                                                          2 ,
                                                          std::vector< size_t >( 2 , 0 )
                                                         )
                                                        ) ;

      for ( size_t j = 1 ; j < 3 ; j++ ) {
        for ( size_t l = 0 ; l < 2 ; l++ ) {
          for ( size_t v = 0 ; v < 2 ; v++ ) {
            sd2[ j - 1 ][ l ][ v ] = compute_second_difference_column_index
(
                                                                        p2 ,
                                                                        j ,
                                                                        l ,
                                                                        v
                                                                       ) ;
          }
        }
      }

      // Compute the column indices of  the LP matrix corresponding to
      // the four  control points  defining the  p1-th segment  of the
      // curve.

      std::vector< std::vector< size_t > > cp1( 4 , std::vector< size_t >( 2 , 0 ) ) ;

```

```
    for ( size_t i = 0 ; i < 4 ; i++ ) {
      for  ( size_t j = 0 ; j < 2 ; j++ ) {
        cp1[ i ][ j ] = compute_control_value_column_index( p1 , i , j ) ;
      }
    }

    // Compute the column indices of  the LP matrix corresponding to
    // the four  control points  defining the  p2-th segment  of the
    // curve.

    std::vector< std::vector< size_t > > cp2( 4 , std::vector< size_t >( 2 , 0 ) ) ;

    for ( size_t i = 0 ; i < 4 ; i++ ) {
      for  ( size_t j = 0 ; j < 2 ; j++ ) {
        cp2[ i ][ j ] = compute_control_value_column_index( p2 , i , j ) ;
      }
    }

    // ------------------------------------------------------------
    //
    // Process nonlinear terms of Constraint (3c).
    //
    // ------------------------------------------------------------

    //
    // Nonlinear terms of \f$\stackrel{e}{\sim}^p( s_1 )\f$
    //
    insert_nonlinear_terms_of_epiece_point_lower_bound(
                                                    eqline ,
                                                    s1 ,
                                                    c ,
                                                    sd1 ,
                                                    ncsec
                                                   ) ;

     //
     // Nonlinear terms of \f$\stackrel{\sim}{e}^p( s_1 )\f$.
     //
     insert_nonlinear_terms_of_epiece_point_upper_bound(
                                                    eqline + 1 ,
                                                    s1 ,
                                                    c ,
                                                    sd1 ,
                                                    ncsec
                                                   ) ;

     //
     // Nonlinear terms of \f$\stackrel{e}{\sim}^p( s_2 )\f$
     //
     insert_nonlinear_terms_of_epiece_point_lower_bound(
                                                    eqline + 2 ,
                                                    s2 ,
                                                    c ,
                                                    sd2 ,
                                                    ncsec
                                                   ) ;

     //
     // Nonlinear terms of \f$\stackrel{\sim}{e}^p( s_2 )\f$.
     //
     insert_nonlinear_terms_of_epiece_point_upper_bound(
                                                    eqline + 3 ,
                                                    s2 ,
                                                    c ,
                                                    sd2 ,
                                                    ncsec
                                                   ) ;

     // ------------------------------------------------------------
     //
     // Process linear terms of Constraint (3c).
     //
     // ------------------------------------------------------------

     //
     // Linear terms of \f$\stackrel{e}{\sim}^p( s_1 )\f$
     //
     insert_linear_terms_of_epiece_point_bounds(
                                             eqline ,
```

```
                                                      s1 ,
                                                      t1 ,
                                                      p1 ,
                                                      c ,
                                                      cp1 ,
                                                      ncsec
                                                     ) ;

          //
          // Linear terms of \f$\stackrel{\sim}{e}^p( s_2 )\f$.
          //
          insert_linear_terms_of_epiece_point_bounds(
                                                      eqline + 2 ,
                                                      s2 ,
                                                      t2 ,
                                                      p2 ,
                                                      c ,
                                                      cp2 ,
                                                      ncsec
                                                     ) ;

          // -------------------------------------------------------------
          //
          // Compute right-hand side of the constraints.
          //
          // -------------------------------------------------------------

          insert_rhs_of_sleeve_inside_csegment_constraints(
                                                            eqline ,
                                                            c ,
                                                            ncsec
                                                           ) ;
          //
          // Increment equation counter.
          //
          eqline += 4 ;
        }

        return ;
      }
```

16.4.3.14 void channel::CurveBuilder::evaluate_bounding_polynomial (const size_t *j*, const double *t*, double & *lower*, double & *upper*) [private]

Obtains a lower bound and an upper bound for the value of a precomputed, bounding polynomial at a given parameter value.

Parameters

j	An index for the precomputed, bounding polynomial.
t	A parameter value.
lower	A reference to the lower bound.
upper	A reference to the upper bound.

Definition at line 2087 of file curvebuilder.cpp.

References _tf, channel::TabulatedFunction::alower(), channel::TabulatedFunction::aupper(), and treat_exception.

Referenced by insert_nonlinear_terms_of_epiece_point_lower_bound(), and insert_nonlinear_terms_of_epiece_point_upper_bound().

```
{
  try {
    lower = _tf->alower( j , t ) ;
    upper = _tf->aupper( j , t ) ;
  }
  catch ( const ExceptionObject& xpt ) {
    treat_exception( xpt ) ;
    exit( EXIT_FAILURE ) ;
  }

```

```
    return ;
  }
```

16.4.3.15 double channel::CurveBuilder::get_bound_of_ith_constraint (const size_t *i*) const throw ExceptionObject) [inline]

Returns the real value on the right-hand side of the equality or inequality corresponding to the i-th constraint.

Parameters

i	The index of a constraint.

Returns

The real value on the right-hand side of the equality or inequality corresponding to the i-th constraint.

Definition at line 439 of file curvebuilder.hpp.

Referenced by write_lp().

```
  {
    if ( _coefficients.empty() ) {
      std::stringstream ss( std::stringstream::in | std::stringstream::out ) ;
      ss << "No constraint has been created so far" ;
      throw ExceptionObject( __FILE__ , __LINE__ , ss.str().c_str() ) ;
    }

    if ( i >= _coefficients.size() ) {
      std::stringstream ss( std::stringstream::in | std::stringstream::out ) ;
      ss << "Constraint index is out of range" ;
      throw ExceptionObject( __FILE__ , __LINE__ , ss.str().c_str() ) ;
    }

#ifdef DEBUGMODE
    assert( _bounds.size() == _coefficients.size() ) ;
    assert( _bounds.size() > std::vector< std::vector< Bound > >::size_type( i ) ) ;
#endif

    return _bounds[ i ].get_value() ;
  }
```

16.4.3.16 size_t channel::CurveBuilder::get_coefficient_identifier (const size_t *i*, const size_t *j*) const throw ExceptionObject) [inline]

Returns the index of the column that corresponds to the j-th coefficient of the i-th constraint in the matrix associated with the linear program (LP) instance.

Parameters

i	The index of a constraint.
j	The j-th (nonzero) coefficient of the i-th constraint.

Returns

The index of the column that corresponds to the j-th coefficient of the i-th constraint in the matrix associated with the linear program (LP) instance.

Definition at line 364 of file curvebuilder.hpp.

Referenced by write_lp().

```
{
  if ( _coefficients.empty() ) {
    std::stringstream ss( std::stringstream::in | std::stringstream::out ) ;
    ss << "No constraint has been created so far" ;
    throw ExceptionObject( __FILE__ , __LINE__ , ss.str().c_str() ) ;
  }

  if ( i >= _coefficients.size() ) {
    std::stringstream ss( std::stringstream::in | std::stringstream::out ) ;
    ss << "Constraint index is out of range" ;
    throw ExceptionObject( __FILE__ , __LINE__ , ss.str().c_str() ) ;
  }

  if ( j >= _coefficients[ i ].size() ) {
    std::stringstream ss( std::stringstream::in | std::stringstream::out ) ;
    ss << "Coefficient index is out of range" ;
    throw ExceptionObject( __FILE__ , __LINE__ , ss.str().c_str() ) ;
  }

  return _coefficients[ i ][ j ].get_col() ;
}
```

16.4.3.17 size_t channel::CurveBuilder::get_coefficient_value (const size_t *i*, const size_t *j*) const throw **ExceptionObject**) [inline]

Returns the (i, j) entry of the matrix associated with the instance of the linear program.

Parameters

i	The index of a constraint.
j	The j-th (nonzero) coefficient of the i-th constraint.

Returns

The (i, j) entry of the matrix associated with the instance of the linear program.

Definition at line 402 of file curvebuilder.hpp.

Referenced by write_lp().

```
{
  if ( _coefficients.empty() ) {
    std::stringstream ss( std::stringstream::in | std::stringstream::out ) ;
    ss << "No constraint has been created so far" ;
    throw ExceptionObject( __FILE__ , __LINE__ , ss.str().c_str() ) ;
  }

  if ( i >= _coefficients.size() ) {
    std::stringstream ss( std::stringstream::in | std::stringstream::out ) ;
    ss << "Constraint index is out of range" ;
    throw ExceptionObject( __FILE__ , __LINE__ , ss.str().c_str() ) ;
  }

  if ( j >= _coefficients[ i ].size() ) {
    std::stringstream ss( std::stringstream::in | std::stringstream::out ) ;
    ss << "Coefficient index is out of range" ;
    throw ExceptionObject( __FILE__ , __LINE__ , ss.str().c_str() ) ;
  }

  return _coefficients[ i ][ j ].get_value() ;
}
```

16.4.3.18 double channel::CurveBuilder::get_control_value (const size_t *i*, const size_t *v*) const throw **ExceptionObject**) [inline]

Returns the v-th coordinate of the i-th control point of the b-spline curve threaded into the channel.

Parameters

i	The index of the i-th control point of the b-spline curve.
v	The v-th Cartesian coordinate of the i-th control point of the b-spline curve.

Returns

The v-th coordinate of the i-th control point of the b-spline curve threaded into the channel.

Definition at line 288 of file curvebuilder.hpp.

Referenced by write_solution().

```
{
  if ( i >= ( _np + 3 ) ) {
    std::stringstream ss( std::stringstream::in | std::stringstream::out ) ;
    ss << "Index of the control point is out of range" ;
    throw ExceptionObject( __FILE__ , __LINE__ , ss.str().c_str() ) ;
  }

  if ( v >= 2 ) {
    std::stringstream ss( std::stringstream::in | std::stringstream::out ) ;
    ss << "Index of the Cartesian coordinate is out of range" ;
    throw ExceptionObject( __FILE__ , __LINE__ , ss.str().c_str() ) ;
  }

  if ( _ctrlpts.empty() ) {
    std::stringstream ss( std::stringstream::in | std::stringstream::out ) ;
    ss << "Control points have not been computed" ;
    throw ExceptionObject( __FILE__ , __LINE__ , ss.str().c_str() ) ;
  }

  size_t index = ( 2 * i ) + v ;

  return _ctrlpts[ index ] ;
}
```

16.4.3.19 size_t channel::CurveBuilder::get_degree ( ) const [inline]

Returns the degree of the bspline curve.

Returns

The degree of the bspline curve.

Definition at line 184 of file curvebuilder.hpp.

Referenced by get_number_of_control_points(), and write_lp().

```
{
  return 3 ;
}
```

16.4.3.20 double channel::CurveBuilder::get_lower_bound_on_second_difference_value (const size_t *p*, const size_t *i*, const size_t *v*) const throw ExceptionObject) [inline]

Returns the lower bound (found by the LP solver) on the v-th coordinate of the i-th second difference of the i-th curve segment of the b-spline curve threaded into the channel.

Parameters

p	The index of a curve segment of the b-spline.
i	The index of the i-th second difference of the p-th curve segment of the b-spline.
v	The v-th Cartesian coordinate of the i-th control point of the p-th curve segment of the b-spline.

Returns

The lower bound (found by the LP solver) on the v-th coordinate of the i-th second difference vector of the p-th curve segment of the b-spline curve threaded into the channel.

Definition at line 591 of file curvebuilder.hpp.

```
  {
    if ( p >= _np ) {
      std::stringstream ss( std::stringstream::in | std::stringstream::out ) ;
      ss << "Index of the curve segment is out of range" ;
      throw ExceptionObject( __FILE__ , __LINE__ , ss.str().c_str() ) ;
    }

    if ( ( i < 1 ) || ( i > 3 ) ) {
      std::stringstream ss( std::stringstream::in | std::stringstream::out ) ;
      ss << "Index of the second difference vector is out of range" ;
      throw ExceptionObject( __FILE__ , __LINE__ , ss.str().c_str() ) ;
    }

    if ( v >= 2 ) {
      std::stringstream ss( std::stringstream::in | std::stringstream::out ) ;
      ss << "Index of the Cartesian coordinate is out of range" ;
      throw ExceptionObject( __FILE__ , __LINE__ , ss.str().c_str() ) ;
    }

    if ( _secdiff.empty() ) {
      std::stringstream ss( std::stringstream::in | std::stringstream::out ) ;
      ss << "Second differences have not been computed" ;
      throw ExceptionObject( __FILE__ , __LINE__ , ss.str().c_str() ) ;
    }

    size_t index = ( 4 * 2 * p ) + ( 4 * ( i - 1 ) ) + v ;

    return _secdiff[ index ] ;
  }
```

16.4.3.21 void channel::CurveBuilder::get_lp_solver_result_information (glp_prob ∗ *lp*) [private]

Obtain the optimal values found by the LP solver for the structural values of the linear programming corresponding to the channel problem.

Parameters

lp	A pointer to the instance of the LP program.

Definition at line 2706 of file curvebuilder.cpp.

References _ctrlpts, _np, _ofvalue, _secdiff, compute_control_value_column_index(), and compute_second_↩ difference_column_index().

Referenced by solve_lp().

```
{
  //
  // Obtain the control points of the spline curve.
  //
  for ( size_t i = 0 ; i < 4 ; i++ ) {
    for ( size_t v = 0 ; v < 2 ; v++ ) {
      size_t c = compute_control_value_column_index(
                                                         0 ,
```

```
                                                     i ,
                                                     v
                                                    ) ;
        _ctrlpts.push_back(
                           glp_get_col_prim(
                                            lp ,
                                            int( c ) + 1
                                           )
                          ) ;
      }
    }

    for ( size_t p = 1 ; p < _np ; p++ ) {
      for ( size_t v = 0 ; v < 2 ; v++ ) {
        size_t c = compute_control_value_column_index(
                                                     p ,
                                                     3 ,
                                                     v
                                                    ) ;
        _ctrlpts.push_back(
                           glp_get_col_prim(
                                            lp ,
                                            int( c ) + 1
                                           )
                          ) ;
      }
    }

    //
    // Obtain the lower and upper bounds of the second differences.
    //
    for ( size_t i = 1 ; i < 3 ; i++ ) {
      for ( size_t l = 0 ; l < 2 ; l++ ) {
        for ( size_t v = 0 ; v < 2 ; v++ ) {
          size_t c = compute_second_difference_column_index(
                                                           0 ,
                                                           i ,
                                                           l ,
                                                           v
                                                          ) ;

          _secdiff.push_back(
                             glp_get_col_prim(
                                              lp ,
                                              int( c ) + 1
                                             )
                            ) ;
        }
      }
    }

    for ( size_t p = 1 ; p < _np ; p++ ) {
      for ( size_t l = 0 ; l < 2 ; l++ ) {
        for ( size_t v = 0 ; v < 2 ; v++ ) {
          size_t c = compute_second_difference_column_index(
                                                           p ,
                                                           2 ,
                                                           l ,
                                                           v
                                                          ) ;

          _secdiff.push_back(
                             glp_get_col_prim(
                                              lp ,
                                              int( c ) + 1
                                             )
                            ) ;
        }
      }
    }

    //
    // Obtain the minimum value of the objective function.
    //
    _ofvalue = glp_get_obj_val( lp ) ;

    return ;
  }
```

16.4.3.22 size_t channel::CurveBuilder::get_number_of_coefficients_in_the_ith_constraint (const size_t *i*) const throw ExceptionObject) [inline]

Returns the number of coefficients of the i-th constraint of the instance of the linear program.

Parameters

i	The index of a constraint.

Returns

The number of coefficients of the i-th constraint of the instance of the linear program.

Definition at line 330 of file curvebuilder.hpp.

Referenced by write_lp().

```
{
  if ( _coefficients.empty() ) {
    std::stringstream ss( std::stringstream::in | std::stringstream::out ) ;
    ss << "No constraint has been created so far" ;
    throw ExceptionObject( __FILE__ , __LINE__ , ss.str().c_str() ) ;
  }

  if ( i >= _coefficients.size() ) {
    std::stringstream ss( std::stringstream::in | std::stringstream::out ) ;
    ss << "Constraint index is out of range" ;
    throw ExceptionObject( __FILE__ , __LINE__ , ss.str().c_str() ) ;
  }

  return _coefficients[ i ].size() ;
}
```

16.4.3.23 size_t channel::CurveBuilder::get_number_of_constraints ( ) const throw ExceptionObject) [inline]

Returns the number of constraints of the instance of the linear program corresponding to the channel problem solved by this class.

Returns

The number of constraints of the instance of the linear program corresponding to the channel problem solved by this class.

Definition at line 260 of file curvebuilder.hpp.

Referenced by write_lp().

```
{
  if ( _coefficients.empty() ) {
    std::stringstream ss( std::stringstream::in | std::stringstream::out ) ;
    ss << "No constraint has been created so far" ;
    throw ExceptionObject( __FILE__ , __LINE__ , ss.str().c_str() ) ;
  }

  return _coefficients.size() ;
}
```

16.4.3.24 size_t channel::CurveBuilder::get_number_of_control_points ( ) const [inline]

Returns the number of control points of the b-spline.

Returns

The number of control points of the b-spline.

Definition at line 242 of file curvebuilder.hpp.

References get_degree().

Referenced by write_solution().

```
{
  return _np + get_degree() ;
}
```

16.4.3.25 size_t channel::CurveBuilder::get_number_of_csegments ( ) const [inline]

Returns the number of c-segments of the channel.

Returns

The number of c-segments of the channel.

Definition at line 212 of file curvebuilder.hpp.

References _nc.

Referenced by write_lp().

```
{
  return _nc ;
}
```

16.4.3.26 size_t channel::CurveBuilder::get_number_of_segments ( ) const [inline]

Returns the number of b-spline segments.

Returns

The number of b-spline segments.

Definition at line 198 of file curvebuilder.hpp.

References _np.

Referenced by write_lp().

```
{
  return _np ;
}
```

16.4.3.27 std::string channel::CurveBuilder::get_solver_error_message (int *error*) [inline]

Returns the error message of the GLPK solver associated with a given error code.

Parameters

error	Error code returned by the LP solver.

Returns

The error message of the GLPK solver associated with a given error code.

Definition at line 715 of file curvebuilder.hpp.

Referenced by main().

```
716     {
717       std::string message ;
718       switch ( error ) {
719         case GLP_EBADB :
720           message = "Unable to start the search because the number of basic variables is not the same as the number of rows in the problem object." ;
721           break ;
722         case GLP_ESING :
723           message = "Unable to start the search because the basis matrix corresponding to the initial basis is singular within the working precision." ;
724           break ;
725         case GLP_ECOND :
726           message = "Unable to start the search because the basis matrix corresponding to the initial basis is ill-conditioned." ;
727           break ;
728         case GLP_EBOUND :
729           message = "Unable to start the search because some double-bounded variables have incorrect bounds." ;
730           break ;
731         case GLP_EFAIL :
732           message = "The search was prematurely terminated due to the solver failure." ;
733           break ;
734         case GLP_EOBJLL :
735           message = "The search was prematurely terminated because the objective function being maximized has reached its lower limit and continues decreasing." ;
736           break ;
737         case GLP_EOBJUL :
738           message = "The search was prematurely terminated because the objective function being minimized has reached its upper limit and continues increasing." ;
739           break ;
740         case GLP_EITLIM :
741           message = "The search was prematurely terminated because the simplex iteration limit has been exceeded." ;
742           break ;
743         case GLP_ETMLIM :
744           message = "The search was prematurely terminated because the time limit has been exceeded." ;
745           break ;
746         case GLP_ENOPFS :
747           message = "The LP problem instance has no primal feasible solution." ;
748           break ;
749         case GLP_ENODFS :
750           message = "The LP problem instance has no dual feasible solution." ;
751           break ;
752         default :
753           message = "Unknown reason." ;
754           break ;
755       }
756
757       return message ;
758     }
```

16.4.3.28 double channel::CurveBuilder::get_upper_bound_on_second_difference_value (const size_t *p*, const size_t *i*, const size_t *v*) const throw **ExceptionObject**) [inline]

Returns the upper bound (found by the LP solver) on the v-th coordinate of the i-th second difference vector of the p-th curve segment of the b-spline curve threaded into the channel.

Parameters

p	The index of the curve segment of the b-spline.
i	The index of the i-th second difference of the p-th segment of the b-spline.
v	The v-th Cartesian coordinate of the i-th control point of the p-th curve segment of the b-spline.

Returns

The upper bound (found by the LP solver) on the v-th coordinate of the i-th second difference vector of the p-th curve segment of the b-spline curve threaded into the channel.

Definition at line 648 of file curvebuilder.hpp.

```
{
  if ( p >= _np ) {
    std::stringstream ss( std::stringstream::in | std::stringstream::out ) ;
    ss << "Index of the curve is out of range" ;
    throw ExceptionObject( __FILE__ , __LINE__ , ss.str().c_str() ) ;
  }

  if ( ( i < 1 ) || ( i > 3 ) ) {
    std::stringstream ss( std::stringstream::in | std::stringstream::out ) ;
    ss << "Index of the second difference vector is out of range" ;
    throw ExceptionObject( __FILE__ , __LINE__ , ss.str().c_str() ) ;
  }

  if ( v >= 2 ) {
    std::stringstream ss( std::stringstream::in | std::stringstream::out ) ;
    ss << "Index of the Cartesian coordinate is out of range" ;
    throw ExceptionObject( __FILE__ , __LINE__ , ss.str().c_str() ) ;
  }

  if ( _secdiff.empty() ) {
    std::stringstream ss( std::stringstream::in | std::stringstream::out ) ;
    ss << "Second differences have not been computed" ;
    throw ExceptionObject( __FILE__ , __LINE__ , ss.str().c_str() ) ;
  }

  size_t index = ( 4 * 2 * p ) + ( 4 * ( i - 1 ) ) + 2 + v ;

  return _secdiff[ index ] ;
}
```

16.4.3.29 void channel::CurveBuilder::insert_bound (const size_t *eqline*, const Bound::CONSTRAINTYPE *type*, const double *value*) [inline], [private]

Assigns a real value to the right-hand side of a constraint (equality or inequality) of an instance of the linear program associated with the channel problem.

Parameters

eqline	Matrix row corresponding to the constraint.
type	Type of the bound (==, >= or <=).
value	Bound value (right-hand side of the constraint)

Definition at line 964 of file curvebuilder.hpp.

Referenced by compute_channel_corners_outside_sleeve_constraints(), insert_extreme_point_correspondence_constraint(), insert_min_max_constraints(), insert_periodic_correspondence_constraints(), insert_rhs_of_sleeve_corners_in_channel_constraints(), and insert_rhs_of_sleeve_inside_csegment_constraints().

```
{
  _bounds[ eqline ] = Bound(
                            type ,
                            value ,
```

```
                                eqline
                              ) ;

     return ;
   }
```

16.4.3.30 void channel::CurveBuilder::insert_coefficient (const size_t *eqline*, const size_t *index*, const double *value*) [inline], [private]

Assigns a value to the coefficient of an unknown of a given constraint of the linear program (LP). The unknown is identified by its corresponding column index in the associated matrix of the LP.

Parameters

eqline	Matrix row corresponding to the constraint.
index	Matrix column index corresponding to the unknown.
value	Value to be assigned to the unknown coefficient.

Definition at line 934 of file curvebuilder.hpp.

Referenced by compute_channel_corners_outside_sleeve_constraints(), insert_csegment_constraint(), insert_extreme_point_correspondence_constraint(), insert_min_max_constraints(), and insert_periodic_correspondence_constraints().

```
   {
     _coefficients[ eqline ].push_back(
                                      Coefficient(
                                                  eqline ,
                                                  index ,
                                                  value
                                                 )
                                     ) ;

     return ;
   }
```

16.4.3.31 void channel::CurveBuilder::insert_csegment_constraint (const size_t *eqline*, const double *lower*, const double *upper*, const size_t *sdlo*, const size_t *sdup*, const double *normal*) [private]

Inserts the coefficients of the lower and upper bounds of a constraint second difference term into the matrix associated with an instance of the linear program (LP). The term belongs to the equation defining the upper (or lower) bound of a point of an e-piece. The constraint ensures that the point lies on a specific side of the oriented suppporting line of one of the four line segments delimiting a c-segment of the channel.

Parameters

eqline	A counter for the number of constraints.
lower	Coefficient of the second difference lower bound term.
upper	Coefficient of the second difference upper bound term.
sdlo	The index of the LP matrix column corresponding to the second difference lower bound term.
sdup	The index of the LP matrix column corresponding to the second difference upper bound term.
normal	A normal to a supporting, oriented line of one of the four line segments delimiting a specific c-segment of the channel.

Definition at line 2133 of file curvebuilder.cpp.

References insert_coefficient().

Referenced by insert_csegment_constraint(), insert_linear_terms_of_epiece_point_bounds(), insert_nonlinear_terms_of_epiece_point_lower_bound(), and insert_nonlinear_terms_of_epiece_point_upper_bound().

```
{
  double temp ;

  temp = lower * normal ;
  if ( temp != 0 ) {
    insert_coefficient( eqline , sdlo , temp ) ;
  }

  temp = upper * normal ;
  if ( temp != 0 ) {
    insert_coefficient( eqline , sdup , temp ) ;
  }

  return ;
}
```

16.4.3.32 void channel::CurveBuilder::insert_csegment_constraint (const size_t *eqline*, const double *c0*, const double *c1*, const double *c2*, const double *c3*, const size_t *b0*, const size_t *b1*, const size_t *b2*, const size_t *b3*, const double *normal*) [private]

Inserts the coefficients of the linear terms of the upper and lower bounds of an e-piece point equation into the matrix associated with an instance of the linear program (LP). The constraint ensures that the point of the e-piece lies inside a c-segment of the channel.

Parameters

eqline	A counter for the number of constraints.
c0	Coefficient of the first control point of the b-spline segment containing the curve point associated to the e-piece point.
c1	Coefficient of the second control point of the b-spline segment containing the curve point associated to the e-piece point.
c2	Coefficient of the third control point of the b-spline segment containing the curve point associated to the e-piece point.
c3	Coefficient of the fourth control point of the b-spline segment containing the curve point associated to the e-piece point.
b0	Index of the LP matrix column corresponding to the first control point of the b-spline segment containing the curve point associated to the e-piece point.
b1	Index of the LP matrix column corresponding to the second control point of the b-spline segment containing the curve point associated to the e-piece point.
b2	Index of the LP matrix column corresponding to the third control point of the b-spline segment containing the curve point associated to the e-piece point.
b3	Index of the LP matrix column corresponding to the fourth control point of the b-spline segment containing the curve point associated to the e-piece point.
normal	A normal to a supporting, oriented line of one of the four line segments delimiting a specific c-segment of the channel.

Definition at line 2198 of file curvebuilder.cpp.

References insert_coefficient().

```
{
  double temp = c0 * normal ;
  if ( temp != 0 ) {
    insert_coefficient( eqline , b0 , temp ) ;
  }

  temp = c1 * normal ;
  if ( temp != 0 ) {
    insert_coefficient( eqline , b1 , temp ) ;
  }

  temp = c2 * normal ;
  if ( temp != 0 ) {
```

```
        insert_coefficient( eqline , b2 , temp ) ;
      }

      temp = c3 * normal ;
      if ( temp != 0 ) {
        insert_coefficient( eqline , b3 , temp ) ;
      }

      return ;
    }
```

16.4.3.33 void channel::CurveBuilder::insert_csegment_constraint (const size_t *eqline*, const double *c0*, const double *c1*, const double *c2*, const double *c3*, const double *c4*, const size_t *b0*, const size_t *b1*, const size_t *b2*, const size_t *b3*, const size_t *b4*, const double *normal*) [private]

Inserts the coefficients of the linear terms of the upper and lower bounds of an e-piece point equation into the matrix associated with an instance of the linear program (LP). This point belongs to an e-piece segment whose endpoints bound b-spline curve points in two distinct, but consecutive curve segments. The constraint ensures that the e-piece point lies inside a c-segment of the channel.

Parameters

eqline	A counter for the number of constraints.
c0	Coefficient of the first control point of the b-spline segment containing the curve point associated with the right endpoint of the e-piece segment.
c1	Coefficient of the second control point of the b-spline segment containing the curve point associated with the right endpoint of the e-piece segment.
c2	Coefficient of the third control point of the b-spline segment containing the curve point associated with the right endpoint of the e-piece segment.
c3	Coefficient of the fourth control point of the b-spline segment containing the curve point associated with the right endpoint of the e-piece segment.
c4	Coefficient of the fourth control point of the b-spline segment containing the curve point associated with the left endpoint of the e-piece segment.
b0	Index of the LP matrix column corresponding to the first control point of the b-spline segment containing the curve point associated with the right endpoint of the e-piece segment.
b1	Index of the LP matrix column corresponding to the second control point of the b-spline segment containing the curve point associated with the right endpoint of the e-piece segment.
b2	Index of the LP matrix column corresponding to the third control point of the b-spline segment containing the curve point associated with the right endpoint of the e-piece segment.
b3	Index of the LP matrix column corresponding to the fourth control point of the b-spline segment containing the curve point associated with the right endpoint of the e-piece segment.
b4	Index of the LP matrix column corresponding to the fourth control point of the b-spline segment containing the curve point associated with the left endpoint of the e-piece segment.
normal	A normal to a supporting, oriented line of one of the four line segments delimiting a specific c-segment of the channel.

Definition at line 2284 of file curvebuilder.cpp.

References insert_coefficient(), and insert_csegment_constraint().

```
    {
      insert_csegment_constraint(
                                eqline ,
                                c0 ,
                                c1 ,
                                c2 ,
                                c3 ,
                                b0 ,
                                b1 ,
                                b2 ,
```

```
                              b3 ,
                              normal
                            ) ;

  double temp = c4 * normal ;
  if ( temp != 0 ) {
    insert_coefficient( eqline , b4 , temp ) ;
  }

  return ;
}
```

16.4.3.34 void channel::CurveBuilder::insert_extreme_point_correspondence_constraint (const size_t *eqline*, const std::vector< size_t > & *col*, const std::vector< double > & *val*, const double *rhs*) [private]

Inserts into the linear program (LP) matrix the coefficients of the unknowns and the right-hand side value of a constraint corresponding to the location of the starting or final point of the b-spline curve.

Parameters

eqline	A reference to the counter of equations.
col	An array with the LP matrix column indices corresponding to the unknowns of the correspondence constraint.
val	An array with the values corresponding to the unknowns of the correspondence constraint.
rhs	The right-hand side value of the constraint.

Definition at line 1424 of file curvebuilder.cpp.

References insert_bound(), and insert_coefficient().

Referenced by compute_correspondence_constraints().

```
{
  for ( size_t i = 0 ; i < 4 ; i++ ) {
    insert_coefficient( eqline , col[ i ] , val[ i ] ) ;
  }

  insert_bound( eqline , Bound::EQT , rhs ) ;

  return ;
}
```

16.4.3.35 void channel::CurveBuilder::insert_linear_terms_of_epiece_point_bounds (const size_t *eqline*, const double *s*, const double *t*, const size_t *p*, const size_t *c*, const std::vector< std::vector< size_t > > & *cp*, const std::vector< std::vector< double > > & *nl*, const std::vector< std::vector< double > > & *nu*) [private]

Inserts the coefficients of the linear terms of the equation defining lower and upper bounds for the e-piece points into the matrix associated with the Linear Program (LP). These terms occur in the constraint that enforces an e-piece point to stay inside channel.

Parameters

eqline	A counter for the number of constraints.
s	A parameter value identifying a point on the e-piece.
t	A parameter value identifying the b-spline point that corresponds to the point on the e-piece at parameter *s*.

p	Index of the b-spline segment containing the b-spline point at parameter *t*.
c	An index identifying the c-segment the e-piece point belongs to.
cp	Array with the LP matrix column indices corresponding to the control points of the b-spline defining the p-th piece of the curve.
nl	Array of Cartesian coordinates of outward normals to the supporting lines of the lower envelope segments of the channel.
nu	Array of Cartesian coordinates of outward normals to the supporting lines of the upper envelope segments of the channel.

Definition at line 1850 of file curvebuilder.cpp.

References insert_csegment_constraint().

Referenced by compute_sleeve_corners_in_channel_constraints(), and compute_sleeve_inside_csegment_constraints().

```
{
  //
  // The coefficients are the same for each Cartesian coordinate.
  //
  const double onesixth = double( 1 ) / double( 6 ) ;

  // The upper and  lower bounds on the e-piece points  must be on
  // or  above the  lower envelope  of the  c-th c-segment  of the
  // channel.

  const double c0 = onesixth * ( 1 - s ) ;
  const double c1 = ( ( -2 + 3 * s ) * onesixth ) + ( p + 4 - t ) ;
  const double c2 = ( (  1 - 3 * s ) * onesixth ) + ( t - p - 3 ) ;
  const double c3 = onesixth * s ;

  for ( size_t v = 0 ; v < 2 ; v++ ) {
    //
    // Compute constraints for the v-th Cartesian coordinate.
    //
    insert_csegment_constraint(
                               eqline ,
                               c0 ,
                               c1 ,
                               c2 ,
                               c3 ,
                               cp[ 0 ][ v ] ,
                               cp[ 1 ][ v ] ,
                               cp[ 2 ][ v ] ,
                               cp[ 3 ][ v ] ,
                               nl[ c ][ v ]
                              ) ;

    // The upper and  lower bounds on the e-piece points  must be on
    // or  below the  upper envelope  of the  c-th c-segment  of the
    // channel.
    insert_csegment_constraint(
                               eqline + 1 ,
                               c0 ,
                               c1 ,
                               c2 ,
                               c3 ,
                               cp[ 0 ][ v ] ,
                               cp[ 1 ][ v ] ,
                               cp[ 2 ][ v ] ,
                               cp[ 3 ][ v ] ,
                               nu[ c ][ v ]
                              ) ;
  }

  return ;
}
```

16.4.3.36 void channel::CurveBuilder::insert_linear_terms_of_epiece_point_bounds (const size_t *eqline*, const double *s*, const double *t*, const size_t *p*, const size_t *c*, const std::vector< std::vector< size_t > > & *cp*, const std::vector< std::vector< double > > & *ncsec*) [private]

Inserts the coefficients of the linear terms of the equation defining lower and upper bounds for the e-piece points into the matrix associated with the Linear Program (LP). These terms occur in the constraint that enforces an e-piece point to stay either on the right or on left side of a channel c-section.

Parameters

eqline	A counter for the number of constraints.
s	A parameter value identifying a point on the e-piece.
t	A parameter value identifying the b-spline point that corresponds to the point on the e-piece at parameter *s*.
p	Index of the b-spline segment containing the b-spline point at parameter *t*.
c	An index identifying the c-segment the e-piece point belongs to.
cp	Array with the LP matrix column indices corresponding to the control points of the b-spline defining the p-th piece of the curve.
ncsec	Array of Cartesian coordinates of normals (pointing to the left) to the supporting lines of the c-sections of the channel.

Definition at line 1939 of file curvebuilder.cpp.

References insert_csegment_constraint().

```
{
  //
  // The coefficients are the same for each Cartesian coordinate.
  //
  const double onesixth = double( 1 ) / double( 6 ) ;

  // The upper and lower bounds on  the e-piece point must be either
  // on the left or on the right side of a c-section of the channel.

  const double c0 = onesixth * ( 1 - s ) ;
  const double c1 = ( ( -2 + 3 * s ) * onesixth ) + ( p + 4 - t ) ;
  const double c2 = ( (  1 - 3 * s ) * onesixth ) + ( t - p - 3 ) ;
  const double c3 = onesixth * s ;

  for ( size_t v = 0 ; v < 2 ; v++ ) {
    //
    // Compute constraints for the v-th Cartesian coordinate.
    //

    //
    // Lower bound --> Equation eqline
    //
    insert_csegment_constraint(
                               eqline ,
                               c0 ,
                               c1 ,
                               c2 ,
                               c3 ,
                               cp[ 0 ][ v ] ,
                               cp[ 1 ][ v ] ,
                               cp[ 2 ][ v ] ,
                               cp[ 3 ][ v ] ,
                               ncsec[ c ][ v ]
                              ) ;

    //
    // Upper bound --> Equation eqline + 1
    //
    insert_csegment_constraint(
                               eqline + 1 ,
                               c0 ,
                               c1 ,
                               c2 ,
                               c3 ,
                               cp[ 0 ][ v ] ,
                               cp[ 1 ][ v ] ,
```

```
                                    cp[ 2 ][ v ] ,
                                    cp[ 3 ][ v ] ,
                                    ncsec[ c ][ v ]
                                   ) ;
        }

        return ;
    }
```

16.4.3.37 void channel::CurveBuilder::insert_min_max_constraints (const size_t *eqline*, const size_t *lo*, const size_t *up*, const size_t *b0*, const size_t *b1*, const size_t *b2*) [private]

Inserts the coefficients of the equations defining the three min-max constraints into the matrix associated with the linear program (LP), and sets the right-hand side of the constraints as well.

Parameters

eqline	A reference to the counter of equations.
lo	Column index of the lower bound for a second difference.
up	Column index of the upper bound for a second difference.
b0	Column index of the first control value defining the second difference.
b1	Column index of the second control value defining the second difference.
b2	Column index of the third control value defining the second difference.

Definition at line 1361 of file curvebuilder.cpp.

References insert_bound(), and insert_coefficient().

Referenced by compute_min_max_constraints().

```
    {
      // First  min-max  constraint:  the  upper  bound  of  the  second
      // difference must  be greater than or  equal to the value  of the
      // second difference:

      const double onesixth = double( 1 ) / double( 6 ) ;

      insert_coefficient( eqline , up ,             1 ) ;
      insert_coefficient( eqline , b0 , -1 * onesixth ) ;
      insert_coefficient( eqline , b1 ,  2 * onesixth ) ;
      insert_coefficient( eqline , b2 , -1 * onesixth ) ;

      insert_bound( eqline , Bound::GTE , 0 ) ;

      // Second  min-max  constraint:  the  upper bound  on  the  second
      // difference must be greater than or equal to zero (i.e., must be
      // non-negative).

      insert_coefficient( eqline + 1 , up , 1 ) ;

      insert_bound( eqline + 1 , Bound::GTE , 0 ) ;

      // Third min-max constraint: the sum of the upper and lower bounds
      // of the second difference must be  equal to the value the second
      // difference.

      insert_coefficient( eqline + 2 , up ,             1 ) ;
      insert_coefficient( eqline + 2 , lo ,             1 ) ;
      insert_coefficient( eqline + 2 , b0 , -1 * onesixth ) ;
      insert_coefficient( eqline + 2 , b1 ,  2 * onesixth ) ;
      insert_coefficient( eqline + 2 , b2 , -1 * onesixth ) ;

      insert_bound( eqline + 2 , Bound::EQT , 0 ) ;

      return ;
    }
```

16.4.3.38 void channel::CurveBuilder::insert_nonlinear_terms_of_epiece_point_lower_bound (const size_t *eqline*, const double *s*, const size_t *c*, const std::vector< std::vector< std::vector< size_t > > > & *sd*, const std::vector< std::vector< double > > & *nl*, const std::vector< std::vector< double > > & *nu*) [private]

Inserts the coefficients of the second difference terms of the equation defining lower bounds for the e-piece points into the matrix associated with an instance of the Linear Program (LP). The terms belong to the constraint that forces the e-piece points to be inside a certain c-section of the channel.

Parameters

eqline	A counter for the number of constraints.
s	A parameter value identifying a point on the e-piece.
c	An index identifying a c-segment of the channel.
sd	Array with the LP matrix column indices corresponding to the lower and upper bounds on second differences occurring in the equation defining the e-piece points belonging to the c-segment.
nl	Array of Cartesian coordinates of outward normals to the supporting lines of the lower envelope segments of the channel.
nu	Array of Cartesian coordinates of outward normals to the supporting lines of the upper envelope segments of the channel.

Definition at line 1514 of file curvebuilder.cpp.

References evaluate_bounding_polynomial(), and insert_csegment_constraint().

Referenced by compute_sleeve_corners_in_channel_constraints(), and compute_sleeve_inside_csegment_↩ constraints().

```
{
  // Insert into the matrix associated  with the linear program (LP)
  // the  coefficients  of the  second  differences  of the  e-piece
  // breakpoint lower bound \f$\stackrel{e}{\sim}^p\f$ in constraint
  // (3a).

  //
  // The computation is performed for each second difference j.
  //

  for ( size_t j = 1 ; j < 3 ; j++ ) {
    //
    // Get lower and upper bounds for the special polynomial.
    //
    double dl ;
    double du ;
    evaluate_bounding_polynomial(
                                 j  ,
                                 s  ,
                                 du ,   // switch lower and upper bounds.
                                 dl     // switch lower and upper bounds.
                                ) ;

    //
    // The coefficients are the same for each Cartesian coordinate.
    //

    for ( size_t v = 0 ; v < 2 ; v++ ) {
      // Point \f$\stackrel{e}{\sim}^p( s )  \f$ of the e-piece must
      // be above  the lower envelope  of the c-th c-segment  of the
      // channel.
      insert_csegment_constraint(
                                 eqline ,
                                 dl ,
                                 du ,
                                 sd[ j - 1 ][ 0 ][ v ] ,
                                 sd[ j - 1 ][ 1 ][ v ] ,
                                 nl[ c ][ v ]
                                ) ;

      // Point \f$\stackrel{e}{\sim}^p( s )  \f$ of the e-piece must
      // be below  the upper envelope  of the c-th c-segment  of the
      // channel.
      insert_csegment_constraint(
```

```
                              eqline + 1 ,
                              dl ,
                              du ,
                              sd[ j - 1 ][ 0 ][ v ] ,
                              sd[ j - 1 ][ 1 ][ v ] ,
                              nu[ c ][ v ]
                             ) ;
      }
    }

    return ;
  }
```

16.4.3.39 void channel::CurveBuilder::insert_nonlinear_terms_of_epiece_point_lower_bound (const size_t *eqline*, const double *s*, const size_t *c*, const std::vector< std::vector< std::vector< size_t > > > & *sd*, const std::vector< std::vector< double > > & *ncsec*) [private]

Inserts the coefficients of the second difference terms of the equation defining lower bounds for the e-piece points into the matrix associated with an instance of the Linear Program (LP). The terms belong to the constraint that forces one e-piece point to be on the right or left side of a channel c-section.

Parameters

eqline	A counter for the number of constraints.
s	A parameter value identifying a point on the e-piece.
c	An index identifying a c-segment of the channel.
sd	Array with the LP matrix column indices corresponding to the lower and upper bounds on second differences occurring in the equation defining the e-piece points belonging to the c-segment.
ncsec	Array of Cartesian coordinates of normals (pointing to the left) to the supporting lines of the c-sections of the channel.

Definition at line 1602 of file curvebuilder.cpp.

References evaluate_bounding_polynomial(), and insert_csegment_constraint().

```
  {
    // Insert into the matrix associated  with the linear program (LP)
    // the  coefficients  of the  second  differences  of the  e-piece
    // breakpoint lower bound \f$\stackrel{e}{\sim}^p\f$ in constraint
    // (3c).

    //
    // The computation is performed for each second difference j.
    //

    for ( size_t j = 1 ; j < 3 ; j++ ) {
      //
      // Get lower and upper bounds for the special polynomial.
      //
      double dl ;
      double du ;
      evaluate_bounding_polynomial(
                              j  ,
                              s  ,
                              du ,   // switch lower and upper bounds.
                              dl     // switch lower and upper bounds.
                             ) ;

      //
      // The coefficients are the same for each Cartesian coordinate.
      //

      for ( size_t v = 0 ; v < 2 ; v++ ) {
        // Point \f$\stackrel{e}{\sim}^p( s )  \f$ of the e-piece must
        // be either  on the  right side  or on the  left side  of the
        // channel c-section.
        insert_csegment_constraint(
                              eqline ,
                              dl ,
```

```
                                     du ,
                                     sd[ j - 1 ][ 0 ][ v ] ,
                                     sd[ j - 1 ][ 1 ][ v ] ,
                                     ncsec[ c ][ v ]
                                    ) ;
        }
      }

      return ;
    }
```

16.4.3.40 void channel::CurveBuilder::insert_nonlinear_terms_of_epiece_point_upper_bound (const size_t *eqline*, const double *s*, const size_t *c*, const std::vector< std::vector< std::vector< size_t > > > & *sd*, const std::vector< std::vector< double > > & *nl*, const std::vector< std::vector< double > > & *nu*) [private]

Inserts into the matrix associated with the Linear Program (LP) the coefficients of the lower and upper bounds of the second difference terms of the equation defining upper bounds for the e-piece points. These terms occur in the constraint that keep the sleeve inside a certain c-section of the channel.

Inserts the coefficients of the second difference terms of the equation defining lower bounds for the e-piece points into the matrix associated with an instance of the Linear Program (LP). The terms belong to the constraint that forces the e-piece points to be inside a certain c-section of the channel.

Parameters

eqline	A counter for the number of constraints.
s	A parameter value identifying a point on the e-piece.
c	An index identifying a c-segment of the channel.
sd	Array with the LP matrix column indices corresponding to the lower and upper bounds on second differences occurring in the equation defining the points on the e-piece matched with the c-segment.
nl	Array of Cartesian coordinates of outward normals to the supporting lines of the lower envelope segments of the channel.
nu	Array of Cartesian coordinates of outward normals to the supporting lines of the upper envelope segments of the channel.
eqline	A counter for the number of constraints.
s	A parameter value identifying a point on the e-piece.
c	An index identifying a c-segment of the channel.
sd	Array with the LP matrix column indices corresponding to the lower and upper bounds on second differences occurring in the equation defining the e-piece points belonging to the c-segment.
nl	Array of Cartesian coordinates of outward normals to the supporting lines of the lower envelope segments of the channel.
nu	Array of Cartesian coordinates of outward normals to the supporting lines of the upper envelope segments of the channel.

Definition at line 1680 of file curvebuilder.cpp.

References evaluate_bounding_polynomial(), and insert_csegment_constraint().

Referenced by compute_sleeve_corners_in_channel_constraints(), and compute_sleeve_inside_csegment_constraints().

```
{
  // Insert into the matrix associated  with the linear program (LP)
  // the  coefficients  of the  second  differences  of the  e-piece
  // breakpoint lower bound \f$\stackrel{\sim}{e}^p\f$ in constraint
  // (3a).

  //
  // The computation is performed for each second difference j.
  //
```

```

    for ( size_t j = 1 ; j < 3 ; j++ ) {
      //
      // Get lower and upper bounds for the special polynomial.
      //
      double dl ;
      double du ;
      evaluate_bounding_polynomial(
                                   j  ,
                                   s  ,
                                   dl ,    // DON't switch lower and upper bounds.
                                   du      // DON't switch lower and upper bounds.
                                  ) ;

      //
      // The coefficients are the same for each Cartesian coordinate.
      //

      for ( size_t v = 0 ; v < 2 ; v++ ) {
        // Point \f$\stackrel{e}{\sim}^p( s )  \f$ of the e-piece must
        // be above  the lower envelope  of the c-th c-segment  of the
        // channel.
        insert_csegment_constraint(
                                   eqline ,
                                   dl ,
                                   du ,
                                   sd[ j - 1 ][ 0 ][ v ] ,
                                   sd[ j - 1 ][ 1 ][ v ] ,
                                   nl[ c ][ v ]
                                  ) ;

        // Point \f$\stackrel{e}{\sim}^p( s )  \f$ of the e-piece must
        // be below  the upper envelope  of the c-th c-segment  of the
        // channel.
        insert_csegment_constraint(
                                   eqline + 1 ,
                                   dl ,
                                   du ,
                                   sd[ j - 1 ][ 0 ][ v ] ,
                                   sd[ j - 1 ][ 1 ][ v ] ,
                                   nu[ c ][ v ]
                                  ) ;
      }
    }

    return ;
  }
```

16.4.3.41 void channel::CurveBuilder::insert_nonlinear_terms_of_epiece_point_upper_bound (const size_t *eqline*, const double *s*, const size_t *c*, const std::vector< std::vector< std::vector< size_t > > > & *sd*, const std::vector< std::vector< double > > & *ncsec*) [private]

Inserts the coefficients of the second difference terms of the equation defining upper bounds for the e-piece points into the matrix associated with an instance of the Linear Program (LP). The terms belong to the constraint that forces one e-piece point to be on the right or left side of a channel c-section.

Parameters

eqline	A counter for the number of constraints.
s	A parameter value identifying a point on the e-piece.
c	An index identifying a c-segment of the channel.
sd	Array with the LP matrix column indices corresponding to the lower and upper bounds on second differences occurring in the equation defining the e-piece points belonging to the c-segment.

ncsec	Array of Cartesian coordinates of normals (pointing to the left) to the supporting lines of the c-sections of the channel.

Definition at line 1768 of file curvebuilder.cpp.

References evaluate_bounding_polynomial(), and insert_csegment_constraint().

```
{
  // Insert into the matrix associated  with the linear program (LP)
  // the  coefficients  of the  second  differences  of the  e-piece
  // breakpoint lower bound \f$\stackrel{\sim}{e}^p\f$ in constraint
  // (3c).

  //
  // The computation is performed for each second difference j.
  //

  for ( size_t j = 1 ; j < 3 ; j++ ) {
    //
    // Get lower and upper bounds for the special polynomial.
    //
    double dl ;
    double du ;
    evaluate_bounding_polynomial(
                                 j  ,
                                 s  ,
                                 dl ,   // DON'T switch lower and upper bounds.
                                 du     // DON'T switch lower and upper bounds.
                                ) ;

    //
    // The coefficients are the same for each Cartesian coordinate.
    //

    for ( size_t v = 0 ; v < 2 ; v++ ) {
      // Point \f$\stackrel{\sim}{e}^p( s )  \f$ of the e-piece must
      // be either  on the  right side   or on the  left side   of the
      // channel c-section.
      insert_csegment_constraint(
                                 eqline ,
                                 dl ,
                                 du ,
                                 sd[ j - 1 ][ 0 ][ v ] ,
                                 sd[ j - 1 ][ 1 ][ v ] ,
                                 ncsec[ c ][ v ]
                                ) ;
    }
  }

  return ;
}
```

16.4.3.42 void channel::CurveBuilder::insert_periodic_correspondence_constraints (const size_t *eqline*, const std::vector< size_t > & *strx*, const std::vector< size_t > & *stry*, const std::vector< size_t > & *endx*, const std::vector< size_t > & *endy*) [private]

Inserts into the linear program (LP) matrix the coefficients of the unknowns and the right-hand side values of the constraints that ensure that the first three control points are the same as the last three control points (in this order).

Parameters

eqline	A reference to the counter of equations.
strx	An array with the column indices of the LP matrix corresponding to the first Cartesian coordinates of the first three control points.

stry	An array with the column indices of the LP matrix corresponding to the second Cartesian coordinates of the first three control points.
endx	An array with the column indices of the LP matrix corresponding to the first Cartesian coordinates of the last three control points.
endy	An array with the column indices of the LP matrix corresponding to the second Cartesian coordinates of the last three control points.

Definition at line 1465 of file curvebuilder.cpp.

References insert_bound(), and insert_coefficient().

Referenced by compute_correspondence_constraints().

```
{
  for ( size_t j = 0 ; j < 3 ; j++ ) {
    insert_coefficient( eqline + 2 * j , strx[ j ] ,  1 ) ;
    insert_coefficient( eqline + 2 * j , endx[ j ] , -1 ) ;

    insert_bound( eqline + 2 * j , Bound::EQT , 0 ) ;

    insert_coefficient( eqline + 2 * j + 1 , stry[ j ] ,  1 ) ;
    insert_coefficient( eqline + 2 * j + 1 , endy[ j ] , -1 ) ;

    insert_bound( eqline + 2 * j + 1 , Bound::EQT , 0 ) ;
  }

  return ;
}
```

16.4.3.43 void channel::CurveBuilder::insert_rhs_of_sleeve_corners_in_channel_constraints (const size_t *eqline,* const size_t *c,* const std::vector< std::vector< double > > & *nl,* const std::vector< std::vector< double > > & *nu*) [private]

Inserts into the matrix associated with the Linear Program (LP) the right-hand side values of the constraints that enforce a sleeve point to stay inside a c-segment of the channel. The type of each constraint (equality or inequality: ==, >= or <=) is also set here.

Parameters

eqline	A counter for the number of constraints.
c	An index identifying the c-segment the e-piece point belongs to.
nl	Array of Cartesian coordinates of outward normals to the supporting lines of the lower envelope segments of the channel.
nu	Array of Cartesian coordinates of outward normals to the supporting lines of the upper envelope segments of the channel.

Definition at line 2025 of file curvebuilder.cpp.

References _lxcoords, _lycoords, _uxcoords, _uycoords, and insert_bound().

Referenced by compute_sleeve_corners_in_channel_constraints().

```
{
  insert_bound( eqline     , Bound::LTE , _lxcoords[ c ] * nl[ c ][ 0 ] +
 _lycoords[ c ] * nl[ c ][ 1 ] ) ;
  insert_bound( eqline + 1 , Bound::LTE , _uxcoords[ c ] * nu[ c ][ 0 ] +
 _uycoords[ c ] * nu[ c ][ 1 ] ) ;

  insert_bound( eqline + 2 , Bound::LTE , _lxcoords[ c ] * nl[ c ][ 0 ] +
 _lycoords[ c ] * nl[ c ][ 1 ] ) ;
  insert_bound( eqline + 3 , Bound::LTE , _uxcoords[ c ] * nu[ c ][ 0 ] +
 _uycoords[ c ] * nu[ c ][ 1 ] ) ;

  return ;
}
```

16.4.3.44 void channel::CurveBuilder::insert_rhs_of_sleeve_inside_csegment_constraints (const size_t *eqline*, const size_t *c*, const std::vector< std::vector< double > > & *ncsec*) [private]

Inserts into the matrix associated with the Linear Program (LP) the right-hand side values of the constraints that enforce one e-piece breakpoint to stay on the right side of a c-section of the channel, and another e-piece breakpoint to stay on the left side of the same c-section.

Parameters

eqline	A counter for the number of constraints.
c	An index identifying the c-segment the e-piece points belongs to.
ncsec	Array of Cartesian coordinates of normals (pointing to the left) to the supporting lines of the c-sections of the channel.

Definition at line 2060 of file curvebuilder.cpp.

References _lxcoords, _lycoords, _uxcoords, _uycoords, and insert_bound().

Referenced by compute_sleeve_inside_csegment_constraints().

```
{
  insert_bound( eqline     , Bound::LTE , _lxcoords[ c ] * ncsec[ c ][ 0 ] + _lycoords[ c ] * ncsec[ c ][ 1 ] ) ;
  insert_bound( eqline + 1 , Bound::LTE , _uxcoords[ c ] * ncsec[ c ][ 0 ] + _uycoords[ c ] * ncsec[ c ][ 1 ] ) ;

  insert_bound( eqline + 2 , Bound::GTE , _lxcoords[ c ] * ncsec[ c ][ 0 ] + _lycoords[ c ] * ncsec[ c ][ 1 ] ) ;
  insert_bound( eqline + 3 , Bound::GTE , _uxcoords[ c ] * ncsec[ c ][ 0 ] + _uycoords[ c ] * ncsec[ c ][ 1 ] ) ;
}
```

16.4.3.45 bool channel::CurveBuilder::is_curve_closed ( ) const [inline]

Returns the logic value true if the b-spline curve is closed, and the logic value false otherwise.

Returns

The logic value true if the b-spline curve is closed, and the logic value false otherwise.

Definition at line 228 of file curvebuilder.hpp.

References _closed.

Referenced by write_lp().

```
{
  return _closed ;
}
```

16.4.3.46 bool channel::CurveBuilder::is_equality (const size_t *i*) const throw ExceptionObject) [inline]

Returns the logic value true if the type of the i-th constraint is equality; otherwise, returns the logic value false.

Parameters

i	The index of a constraint.

Returns

The logic value true if the type of the i-th constraint is equality; otherwise, the logic value false is returned.

Definition at line 475 of file curvebuilder.hpp.

Referenced by write_lp().

```
    {
      if ( _coefficients.empty() ) {
        std::stringstream ss( std::stringstream::in | std::stringstream::out ) ;
        ss << "No constraint has been created so far" ;
        throw ExceptionObject( __FILE__ , __LINE__ , ss.str().c_str() ) ;
      }

      if ( i >= _coefficients.size() ) {
        std::stringstream ss( std::stringstream::in | std::stringstream::out ) ;
        ss << "Constraint index is out of range" ;
        throw ExceptionObject( __FILE__ , __LINE__ , ss.str().c_str() ) ;
      }

#ifdef DEBUGMODE
      assert( _bounds.size() == _coefficients.size() ) ;
      assert( _bounds.size() > i ) ;
#endif

      return _bounds[ i ].get_type() == Bound::EQT ;
    }
```

16.4.3.47 bool channel::CurveBuilder::is_greater_than_or_equal_to (const size_t *i*) const throw **ExceptionObject**) [inline]

Returns the logic value true if the i-th constraint is an inequality of the type greater than or equal to; otherwise, returns the logic value false.

Parameters

i	The index of a constraint.

Returns

The logic value true if the i-th constraint is an inequality of the type greater than or equal to; otherwise, the logic value false is returned.

Definition at line 511 of file curvebuilder.hpp.

Referenced by write_lp().

```
    {
      if ( _coefficients.empty() ) {
        std::stringstream ss( std::stringstream::in | std::stringstream::out ) ;
        ss << "No constraint has been created so far" ;
        throw ExceptionObject( __FILE__ , __LINE__ , ss.str().c_str() ) ;
      }

      if ( i >= _coefficients.size() ) {
        std::stringstream ss( std::stringstream::in | std::stringstream::out ) ;
        ss << "Constraint index is out of range" ;
        throw ExceptionObject( __FILE__ , __LINE__ , ss.str().c_str() ) ;
      }

#ifdef DEBUGMODE
      assert( _bounds.size() == _coefficients.size() ) ;
```

```
    assert( _bounds.size() > i ) ;
#endif

    return _bounds[ i ].get_type() == Bound::GTE ;
  }
```

16.4.3.48 bool channel::CurveBuilder::is_less_than_or_equal_to (const size_t *i*) const throw **ExceptionObject**) [inline]

Returns the logic value true if the i-th constraint is an inequality of the type less than or equal to; otherwise, returns the logic value false.

Parameters

i	The index of a constraint.

Returns

The logic value true if the i-th constraint is an inequality of the type less than or equal to; otherwise, the logic value false is returned.

Definition at line 548 of file curvebuilder.hpp.

Referenced by write_lp().

```
  {
    if ( _coefficients.empty() ) {
      std::stringstream ss( std::stringstream::in | std::stringstream::out ) ;
      ss << "No constraint has been created so far" ;
      throw ExceptionObject( __FILE__ , __LINE__ , ss.str().c_str() ) ;
    }

    if ( i >= _coefficients.size() ) {
      std::stringstream ss( std::stringstream::in | std::stringstream::out ) ;
      ss << "Constraint index is out of range" ;
      throw ExceptionObject( __FILE__ , __LINE__ , ss.str().c_str() ) ;
    }

#ifdef DEBUGMODE
    assert( _bounds.size() == _coefficients.size() ) ;
    assert( _bounds.size() > i ) ;
#endif

    return _bounds[ i ].get_type() == Bound::LTE ;
  }
```

16.4.3.49 double channel::CurveBuilder::minimum_value ( ) const [inline]

Returns the optimal (minimum) value of the objective function of the instance of the channel problem as found by the LP solver.

Returns

The optimal (minimum) value of the objective function of the instance of the channel problem as found by the LP solver.

Definition at line 697 of file curvebuilder.hpp.

References _ofvalue.

```
  {
    return _ofvalue ;
  }
```

16.4.3.50 void channel::CurveBuilder::set_up_lp_constraints (glp_prob * *lp*) const [private]

Assemble the matrix of constraints of the linear program, and define the type (equality or inequality) and bounds on the constraints.

Parameters

lp	A pointer to the instance of the LP program.

Definition at line 2412 of file curvebuilder.cpp.

References _bounds, and _coefficients.

Referenced by solve_lp().

```
{
  /*
   * Set up the bounds on the constraints of the problem.
   */

  for ( size_t j = 0 ; j < _bounds.size() ; j++ ) {
#ifdef DEBUGMODE
    assert( j == _bounds[ j ].get_row() ) ;
#endif

    int i = int( _bounds[ j ].get_row() + 1 ) ;

    std::stringstream ss ( std::stringstream::in | std::stringstream::out ) ;
    ss << "c" << i ;
    glp_set_row_name( lp , i , ss.str().c_str() ) ;

    double val = _bounds[ j ].get_value() ;
    if ( _bounds[ j ].get_type() == Bound::LTE ) {
      glp_set_row_bnds( lp , i , GLP_UP ,   0 , val ) ;
    }
    else if ( _bounds[ j ].get_type() == Bound::GTE ) {
      glp_set_row_bnds( lp , i , GLP_LO , val ,   0 ) ;
    }
    else {
      glp_set_row_bnds( lp , i , GLP_FX , val , val ) ;
    }
  }


  /*
   * Obtain the coefficients of the constraints of the problem.
   */

  std::vector< int    > ia ; ia.push_back( 0 ) ;  // GLPK starts indexing array \e ia at 1
  std::vector< int    > ja ; ja.push_back( 0 ) ;  // GLPK starts indexing array \e ja at 1
  std::vector< double > ar ; ar.push_back( 0 ) ;  // GLPK starts indexing array \e ar at 1

  int h = 0 ;
  for ( size_t j = 0 ; j < _coefficients.size() ; j++ ) {
    for ( size_t k = 0 ; k < _coefficients[ j ].size() ; k++ ) {
#ifdef DEBUGMODE
      assert( _coefficients[ j ][ k ].get_row() == j ) ;
#endif
      ia.push_back( int( _coefficients[ j ][ k ].get_row() + 1 ) ) ;
      ja.push_back( int( _coefficients[ j ][ k ].get_col() + 1 ) ) ;
      ar.push_back( _coefficients[ j ][ k ].get_value() ) ;
      ++h ;
    }
  }

  glp_load_matrix(
                  lp ,
                  h ,
                  &ia[ 0 ] ,
                  &ja[ 0 ] ,
                  &ar[ 0 ]
                 ) ;

  return ;
}
```

16.4.3.51 void channel::CurveBuilder::set_up_objective_function (glp_prob ∗ *lp*) const [private]

Define the objective function of the linear program corresponding to the channel problem, which is a minimization problem.

Parameters

lp	A pointer to the instance of the LP program.

Definition at line 2643 of file curvebuilder.cpp.

References _np, and compute_second_difference_column_index().

Referenced by solve_lp().

```
{
  //
  // Add the first two second difference bounds to the function.
  //
  for ( size_t i = 1 ; i < 3 ; i++ ) {
    for ( size_t l = 0 ; l < 2 ; l++ ) {
      for ( size_t v = 0 ; v < 2 ; v++ ) {
        size_t c = compute_second_difference_column_index(
                                                          0 ,
                                                          i ,
                                                          l ,
                                                          v
                                                         ) ;

        if ( l == 0 ) {
          glp_set_obj_coef( lp , int( c ) + 1 , -1 ) ;
        }
        else {
          glp_set_obj_coef( lp , int( c ) + 1 ,  1 ) ;
        }
      }
    }
  }

  //
  // Add the remaining second difference bounds to the function.
  //
  for ( size_t p = 1 ; p < _np ; p++ ) {
    for ( size_t l = 0 ; l < 2 ; l++ ) {
      for ( size_t v = 0 ; v < 2 ; v++ ) {
        size_t c = compute_second_difference_column_index(
                                                          p ,
                                                          2 ,
                                                          l ,
                                                          v
                                                         ) ;

        if ( l == 0 ) {
          glp_set_obj_coef( lp , int( c ) + 1 , -1 ) ;
        }
        else {
          glp_set_obj_coef( lp , int( c ) + 1 ,  1 ) ;
        }
      }
    }
  }

  return ;
}
```

16.4.3.52 void channel::CurveBuilder::set_up_structural_variables (glp_prob ∗ *lp*) const [private]

Define lower and/or upper bounds on the structural variables of the linear program corresponding to the channel problem.

Parameters

lp	A pointer to the instance of the LP program.

Definition at line 2486 of file curvebuilder.cpp.

References _closed, _nc, _np, compute_control_value_column_index(), compute_index_of_corner_barycentric_↩ coordinate(), compute_index_of_endpoint_barycentric_coordinate(), and compute_second_difference_column_index().

Referenced by solve_lp().

```
{
  //
  // Set up bounds for the first two second differences.
  //
  for ( size_t i = 1 ; i <= 2 ; i++ ) {
    for ( size_t l = 0 ; l < 2 ; l++ ) {
      for ( size_t v = 0 ; v < 2 ; v++ ) {
        size_t c = compute_second_difference_column_index(
                                                           0 ,
                                                           i ,
                                                           l ,
                                                           v
                                                          ) ;
        if ( l == 0 ) {
          std::stringstream ss ( std::stringstream::in | std::stringstream::out ) ;
          if ( v == 0 ) {
            ss << "mx" << i ;
          }
          else {
            ss << "my" << i ;
          }
          glp_set_col_name( lp , int( c ) + 1 , ss.str().c_str() ) ;
          glp_set_col_bnds( lp , int( c ) + 1 , GLP_UP , 0 , 0 ) ;
        }
        else {
          std::stringstream ss ( std::stringstream::in | std::stringstream::out ) ;
          if ( v == 0 ) {
            ss << "px" << i ;
          }
          else {
            ss << "py" << i ;
          }
          glp_set_col_name( lp , int( c ) + 1 , ss.str().c_str() ) ;
          glp_set_col_bnds( lp , int( c ) + 1 , GLP_LO , 0 , 0 ) ;
        }
      }
    }
  }

  //
  // Set up bounds for the remaining second differences.
  //
  for ( size_t p = 1 ; p < _np ; p++ ) {
    for ( size_t l = 0 ; l < 2 ; l++ ) {
      for ( size_t v = 0 ; v < 2 ; v++ ) {
        size_t c = compute_second_difference_column_index(
                                                           p ,
                                                           2 ,
                                                           l ,
                                                           v
                                                          ) ;
        if ( l == 0 ) {
          std::stringstream ss ( std::stringstream::in | std::stringstream::out ) ;
          if ( v == 0 ) {
            ss << "mx" << p + 2 ;
          }
          else {
            ss << "my" << p + 2 ;
          }
          glp_set_col_name( lp , int( c ) + 1 , ss.str().c_str() ) ;
          glp_set_col_bnds( lp , int( c ) + 1 , GLP_UP , 0 , 0 ) ;
        }
        else {
          std::stringstream ss ( std::stringstream::in | std::stringstream::out ) ;
          if ( v == 0 ) {
            ss << "px" << p + 2 ;
          }
          else {
            ss << "py" << p + 2 ;
          }
          glp_set_col_name( lp , int( c ) + 1 , ss.str().c_str() ) ;
          glp_set_col_bnds( lp , int( c ) + 1 , GLP_LO , 0 , 0 ) ;
        }
      }
    }
  }

  //
  // Set up bounds for the first four control points.
```

```
    //
    for ( size_t i = 0 ; i < 4 ; i++ ) {
      for ( size_t v = 0 ; v < 2 ; v++ ) {
        size_t c = compute_control_value_column_index(
                                                       0 ,
                                                       i ,
                                                       v
                                                      ) ;

        std::stringstream ss ( std::stringstream::in | std::stringstream::out ) ;
        if ( v == 0 ) {
          ss << "x" << i + 1 ;
        }
        else {
          ss << "y" << i + 1 ;
        }
        glp_set_col_name( lp , int( c ) + 1 , ss.str().c_str() ) ;
        glp_set_col_bnds( lp , int( c ) + 1 , GLP_FR , 0 , 0 ) ;
      }
    }

    for ( size_t p = 1 ; p < _np ; p++ ) {
      for ( size_t v = 0 ; v < 2 ; v++ ) {
        size_t c = compute_control_value_column_index(
                                                       p ,
                                                       3 ,
                                                       v
                                                      ) ;

        std::stringstream ss ( std::stringstream::in | std::stringstream::out ) ;
        if ( v == 0 ) {
          ss << "x" << p + 4 ;
        }
        else {
          ss << "y" << p + 4 ;
        }
        glp_set_col_name( lp , int( c ) + 1 , ss.str().c_str() ) ;
        glp_set_col_bnds( lp , int( c ) + 1 , GLP_FR , 0 , 0 ) ;
      }
    }

    size_t s = compute_index_of_endpoint_barycentric_coordinate ( 0 ) ;
    std::stringstream ss ( std::stringstream::in | std::stringstream::out ) ;
    ss << "st" ;
    glp_set_col_name( lp , int( s ) + 1 , ss.str().c_str() ) ;
    glp_set_col_bnds( lp , int( s ) + 1 , GLP_DB , 0.40 , 0.60 ) ;

    if ( !_closed ) {
      size_t e = compute_index_of_endpoint_barycentric_coordinate ( 1 ) ;
      std::stringstream ss2 ( std::stringstream::in | std::stringstream::out ) ;
      ss2 << "en" ;
      glp_set_col_name( lp , int( e ) + 1 , ss2.str().c_str() ) ;
      glp_set_col_bnds( lp , int( e ) + 1 , GLP_DB , 0.40 , 0.60 ) ;
    }

    for ( size_t i = 1 ; i < _nc ; i++ ) {
      size_t corner_coord = compute_index_of_corner_barycentric_coordinate ( i ) ;
      std::stringstream ss ( std::stringstream::in | std::stringstream::out ) ;
      ss << "co" << i ;
      glp_set_col_name( lp , int( corner_coord ) + 1 , ss.str().c_str() ) ;
      glp_set_col_bnds( lp , int( corner_coord ) + 1 , GLP_DB , 0.40 , 0.60 ) ;
    }

    return ;
  }
```

16.4.3.53 int channel::CurveBuilder::solve_lp (const size_t *rows*, const size_t *cols*) [private]

Solves the linear program corresponding to the channel problem.

Parameters

rows	The number of constraints of the linear program.
cols	The number of unknowns of the linear program.

Returns

The code returned by the LP solver to indicate the status of the computation of the solution of the linear program.

Definition at line 2334 of file curvebuilder.cpp.

References get_lp_solver_result_information(), set_up_lp_constraints(), set_up_objective_function(), and set_up_structural_variables().

Referenced by build().

```
{
  /*
   * Create the LP problem.
   */
  glp_prob* lp = glp_create_prob() ;

  /*
   * Set up the number of constraints and structural variables.
   */
  glp_add_rows( lp , int( rows ) ) ;
  glp_add_cols( lp , int( cols ) ) ;

  /*
   * Set the problem as a minimization one.
   */
  glp_set_obj_dir( lp , GLP_MIN ) ;

  /*
   * Set up the constraints of the problem.
   */
  set_up_lp_constraints( lp ) ;

  /*
   * Define bounds on the structural variables of the problem.
   */
  set_up_structural_variables( lp ) ;

  /*
   * Define objective function.
   */
  set_up_objective_function( lp ) ;

  /*
   * Set parameters of the solver.
   */
  glp_smcp param ;
  glp_init_smcp( &param ) ;

  param.msg_lev  = GLP_MSG_OFF ;
  param.presolve = GLP_ON ;

  /*
   * Call the solver.
   */

  int res = glp_simplex( lp , &param ) ;

  if ( res == 0 ) {
    /*
     * Get the solver result information.
     */
    get_lp_solver_result_information( lp ) ;
  }

  /*
   * Release memory held by the solver.
   */
  glp_delete_prob( lp ) ;

```

```
    return res ;
}
```

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

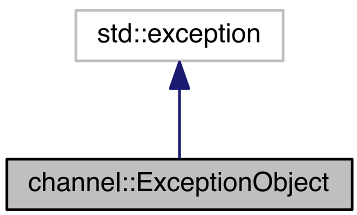
- curvebuilder.hpp
- curvebuilder.cpp

16.5 channel::ExceptionObject Class Reference

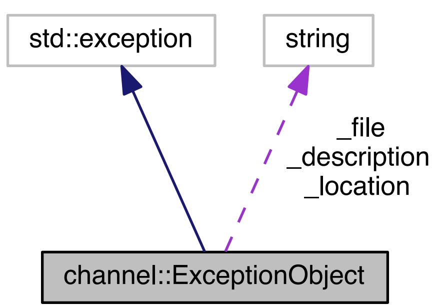
This class extends class *exception* of STL and provides us with a customized way of handling exceptions and showing error messages.

`#include <exceptionobject.hpp>`

Inheritance diagram for channel::ExceptionObject:



Collaboration diagram for channel::ExceptionObject:



Public Member Functions

- ExceptionObject ()

    Creates an instance of this class.
- ExceptionObject (const char ∗file, unsigned ln)

    Creates an instance of this class.
- ExceptionObject (const char ∗file, unsigned int ln, const char ∗desc)

    Creates an instance of this class.
- ExceptionObject (const char ∗file, unsigned ln, const char ∗desc, const char ∗loc)

    Creates an instance of this class.
- ExceptionObject (const ExceptionObject &xpt)

    Clones an instance of this class.
- virtual ∼ExceptionObject () throw ()

    Releases the memory held by an instance of this class.
- ExceptionObject & operator= (const ExceptionObject &xpt)

    Overloads the assignment operator.
- virtual const char ∗ get_name_of_class () const

    Returns the name of this class.
- virtual void set_location (const std::string &s)

    Assigns a location to this exception.
- virtual void set_location (const char ∗s)

    Assigns a location to this exception.
- virtual void set_description (const std::string &s)

    Assigns a description to this exception.
- virtual void set_description (const char ∗s)

    Assigns a description to this exception.
- virtual const char ∗ get_location () const

    Returns the location where this exception occurs.
- virtual const char ∗ get_description () const

    Returns a description of the error that caused this exception.
- virtual const char ∗ get_file () const

    Returns the name of the file containing the line that caused the exception.
- virtual unsigned get_line () const

    Returns the line that caused this exception.
- virtual const char ∗ what () const throw ()

    Returns a description of the error that caused this exception.

Protected Attributes

- std::string _location

    Location of the error in the line that caused the exception.
- std::string _description

    Description of the error.
- std::string _file

    File where the error occured.
- unsigned _line

    Line of the file where the error occurred.

16.5.1 Detailed Description

This class extends class *exception* of STL and provides us with a customized way of handling exceptions and showing error messages.

Definition at line 76 of file exceptionobject.hpp.

16.5.2 Constructor & Destructor Documentation

16.5.2.1 channel::ExceptionObject::ExceptionObject (const char ∗ *file*, unsigned *ln*) [inline]

Creates an instance of this class.

Parameters

file	A pointer to the name of the file where the exception occurred.
ln	Number of the line containing the instruction that caused the exception.

Definition at line 126 of file exceptionobject.hpp.

```
    :
      _location( "Unknown" ) ,
      _description( "Unknown" ) ,
      _file( file ) ,
      _line( ln )
    {
    }
```

16.5.2.2 channel::ExceptionObject::ExceptionObject (const char ∗ *file*, unsigned int *ln*, const char ∗ *desc*) [inline]

Creates an instance of this class.

Parameters

file	A pointer to the name of the file where the exception occurred.
ln	Number of the line containing the instruction that caused the exception.
desc	A pointer to a description of the error that caused the exception.

Definition at line 149 of file exceptionobject.hpp.

```
    :
      _location( "Unknown" ) ,
      _description( desc ) ,
      _file( file ) ,
      _line( ln )
    {
    }
```

16.5.2.3 channel::ExceptionObject::ExceptionObject (const char ∗ *file*, unsigned *ln*, const char ∗ *desc*, const char ∗ *loc*) [inline]

Creates an instance of this class.

Parameters

file	A pointer to the name of the file where the exception occurred.
ln	Number of the line containing the instruction that caused the exception.
desc	A pointer to a description of the error that caused the exception.
loc	A pointer to the location of the exception inside the line where it occurred.

Definition at line 174 of file exceptionobject.hpp.

```
175     :
176       _location( loc ) ,
177       _description( desc ) ,
178       _file( file ) ,
179       _line( ln )
180     {
181     }
```

16.5.2.4 channel::ExceptionObject::ExceptionObject (const ExceptionObject & *xpt*) [inline]

Clones an instance of this class.

Parameters

xpt	A reference to another instance of this class.

Definition at line 192 of file exceptionobject.hpp.

References _description, _file, _line, and _location.

```
192                                                   : exception()
193     {
194       _location = xpt._location ;
195       _description = xpt._description ;
196       _file = xpt._file ;
197       _line = xpt._line ;
198     }
```

16.5.3 Member Function Documentation

16.5.3.1 const char ∗ channel::ExceptionObject::get_description ( ) const [inline], [virtual]

Returns a description of the error that caused this exception.

Returns

A description of the error that caused this exception.

Definition at line 322 of file exceptionobject.hpp.

```
323     {
324       return _description.c_str() ;
325     }
```

16.5.3.2 const char ∗ channel::ExceptionObject::get_file ( ) const [inline], [virtual]

Returns the name of the file containing the line that caused the exception.

Returns

The name of the file containing the line that caused the exception.

Definition at line 338 of file exceptionobject.hpp.

```
{
  return _file.c_str() ;
}
```

16.5.3.3 unsigned channel::ExceptionObject::get_line ( ) const [inline], [virtual]

Returns the line that caused this exception.

Returns

The line that caused this exception.

Definition at line 352 of file exceptionobject.hpp.

References _line.

```
{
  return _line ;
}
```

16.5.3.4 const char * channel::ExceptionObject::get_location ( ) const [inline], [virtual]

Returns the location where this exception occurs.

Returns

The location where this exception occurs.

Definition at line 307 of file exceptionobject.hpp.

```
{
  return _location.c_str() ;
}
```

16.5.3.5 const char * channel::ExceptionObject::get_name_of_class ( ) const [inline], [virtual]

Returns the name of this class.

Returns

The name of this class.

Definition at line 237 of file exceptionobject.hpp.

```
{
  return "ExceptionObject" ;
}
```

16.5.3.6 void channel::ExceptionObject::set_description (const std::string & *s*) [inline], [virtual]

Assigns a description to this exception.

Parameters

s	A string containing the description.

Definition at line 279 of file exceptionobject.hpp.

```
{
  _description = s ;
}
```

16.5.3.7 void channel::ExceptionObject::set_description (const char ∗ *s*) [inline], [virtual]

Assigns a description to this exception.

Parameters

s	A pointer to a string containing the description.

Definition at line 293 of file exceptionobject.hpp.

```
{
  _description = s ;
}
```

16.5.3.8 void channel::ExceptionObject::set_location (const std::string & *s*) [inline], [virtual]

Assigns a location to this exception.

Parameters

s	A string containing the location.

Definition at line 251 of file exceptionobject.hpp.

```
{
  _location = s ;
}
```

16.5.3.9 void channel::ExceptionObject::set_location (const char ∗ *s*) [inline], [virtual]

Assigns a location to this exception.

Parameters

s	A pointer to a string containing the location.

Definition at line 265 of file exceptionobject.hpp.

```
{
  _location = s ;
}
```

16.5.3.10 const char ∗ channel::ExceptionObject::what ( ) const throw) [inline], [virtual]

Returns a description of the error that caused this exception.

Returns

A description of the error that caused this exception.

Definition at line 367 of file exceptionobject.hpp.

```
368     {
369       return _description.c_str() ;
370     }
```

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

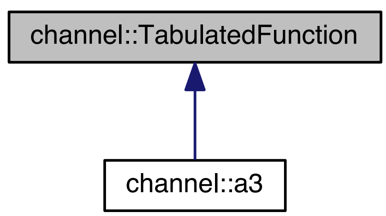
- exceptionobject.hpp

16.6 channel::TabulatedFunction Class Reference

This class represents two-sided, piecewise linear enclosures of a set of $(d-1)$ polynomial functions of degree d in Bézier form. The enclosures must be made available by implementating a pure virtual method in derived classes.

`#include <tabulatedfunction.hpp>`

Inheritance diagram for channel::TabulatedFunction:



Public Member Functions

- TabulatedFunction ()
  Creates an instance of this class.
- virtual ∼TabulatedFunction ()
  Releases the memory held by an instance of this class.
- virtual double alower (const size_t i, const double u) const =0 throw (ExceptionObject)
  Evaluates the piecewise linear function corresponding to the lower enclosure of the i-th tabulated function at a point in $[0,1]$.
- virtual double aupper (const size_t i, const double u) const =0 throw (ExceptionObject)
  Evaluates the piecewise linear function corresponding to the upper enclosure of the i-th tabulated function at a point in $[0,1]$.
- virtual double a (const size_t i, const double u) const =0 throw (ExceptionObject)
  Computes the value of the i-th polynomial function a at a given point of the interval $[0,1]$ of the real line.
- virtual unsigned degree () const =0
  Returns the degree of the tabulated functions.

16.6.1 Detailed Description

This class represents two-sided, piecewise linear enclosures of a set of $(d-1)$ polynomial functions of degree d in Bézier form. The enclosures must be made available by implementating a pure virtual method in derived classes.

Attention

This class is based on several papers surveyed in

```
J.  Peters.
Efficient one-sided linearization of spline geometry.
Proceeding  of the  10th International  Conference on
Mathematics of Surfaces,  Leeds, UK, September 15-17,
2003,  p.   297-319.   (Lecture  Notes   in  Computer
Science,   volume  2768,   Eds.   M.J.    Wilson  and
R.R. Martin).
```

Definition at line 73 of file tabulatedfunction.hpp.

16.6.2 Member Function Documentation

16.6.2.1 double channel::TabulatedFunction::a (const size_t *i*, const double *u*) const throw **ExceptionObject**) [pure virtual]

Computes the value of the i-th polynomial function a at a given point of the interval $[0,1]$ of the real line.

Parameters

i	The index of the i-th polynomial function.
u	A parameter point in the interval $[0,1]$.

Returns

The value of the i-th polynomial function a at a given point u of the interval $[0,1]$ of the real line.

Implemented in channel::a3.

16.6.2.2 double channel::TabulatedFunction::alower (const size_t *i*, const double *u*) const throw **ExceptionObject**) [pure virtual]

Evaluates the piecewise linear function corresponding to the lower enclosure of the i-th tabulated function at a point in $[0,1]$.

Parameters

i	The index of the i-th polynomial function.
u	A value in the interval $[0,1]$.

Returns

The value of the piecewise linear function corresponding to the lower enclosure of the i-th tabulated function at a point in $[0,1]$.

Implemented in channel::a3.

Referenced by channel::CurveBuilder::evaluate_bounding_polynomial().

16.6.2.3 double channel::TabulatedFunction::aupper (const size_t *i*, const double *u*) const throw ExceptionObject) [pure virtual]

Evaluates the piecewise linear function corresponding to the upper enclosure of the i-th tabulated function at a point in $[0, 1]$.

Parameters

i	The index of the i-th polynomial function.
u	A value in the interval $[0, 1]$.

Returns

The value of the piecewise linear function corresponding to the upper enclosure of the i-th tabulated function at a point in $[0, 1]$.

Implemented in channel::a3.

Referenced by channel::CurveBuilder::evaluate_bounding_polynomial().

16.6.2.4 unsigned channel::TabulatedFunction::degree ( ) const [pure virtual]

Returns the degree of the tabulated functions.

Returns

The degree of the tabulated functions.

Implemented in channel::a3.

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

- tabulatedfunction.hpp

Chapter 17

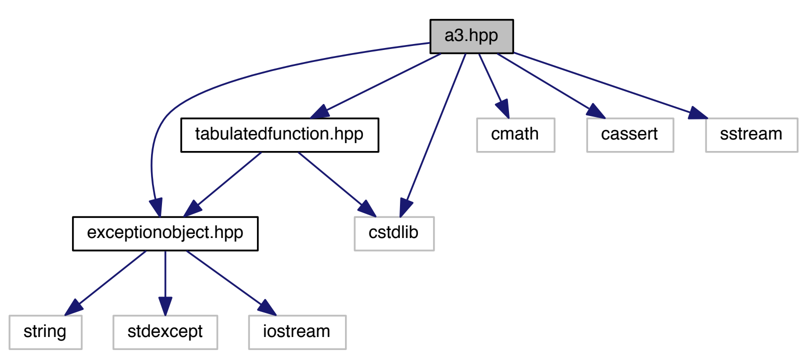
File Documentation

17.1 a3.hpp File Reference

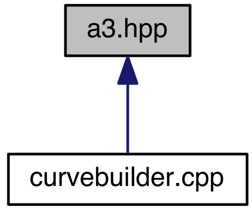
Definition of a class for representing piecewise linear enclosures of certain cubic polynomial functions in Bézier form.

```
#include "exceptionobject.hpp"
#include "tabulatedfunction.hpp"
#include <cmath>
#include <cassert>
#include <sstream>
#include <cstdlib>
```

Include dependency graph for a3.hpp:



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



Classes

- class channel::a3

    This class represents two-sided, piecewise linear enclosures for two polynomial functions of degree 3 in Bézier form.

Namespaces

- channel

    The namespace channel contains the definition and implementation of a set of classes for threading a cubic b-spline curve into a given planar channel delimited by two polygonal chains.

17.1.1 Detailed Description

Definition of a class for representing piecewise linear enclosures of certain cubic polynomial functions in Bézier form.

Author

Marcelo Ferreira Siqueira
Universidade Federal do Rio Grande do Norte,
Departamento de Matemática,
mfsiqueira at mat (dot) ufrn (dot) br

Version

1.0

Date

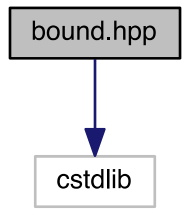
March 2016

Attention

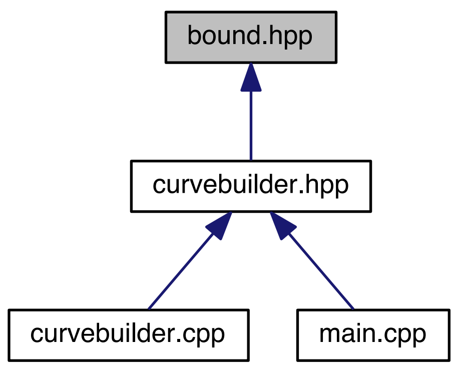
17.2 bound.hpp File Reference

Definition of a class for representing the type of a linear constraint (i.e., equality or inequality) and its right-hand side: a real number.

`#include <cstdlib>`
Include dependency graph for bound.hpp:



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



Classes

- class channel::Bound

  This class represents the type of a constraint (i.e., equality or inequality) and the value of its right-hand side: a real number.

Namespaces

- channel

The namespace channel contains the definition and implementation of a set of classes for threading a cubic b-spline curve into a given planar channel delimited by two polygonal chains.

17.2.1 Detailed Description

Definition of a class for representing the type of a linear constraint (i.e., equality or inequality) and its right-hand side: a real number.

Author

Marcelo Ferreira Siqueira
Universidade Federal do Rio Grande do Norte,
Departamento de Matemática,
mfsiqueira at mat (dot) ufrn (dot) br

Version

1.0

Date

March 2016

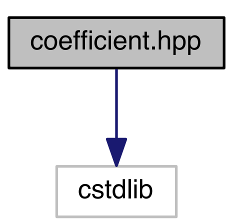
Attention

17.3 coefficient.hpp File Reference

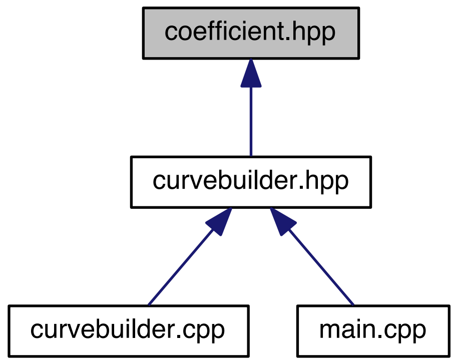
Definition of a class for representing a nonzero coefficient of an unknown of a constraint (inequality or equality) of a linear program instance.

```
#include <cstdlib>
```

Include dependency graph for coefficient.hpp:



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



Classes

- class channel::Coefficient

  This class represents a nonzero coefficient of an unknown of a constraint (inequality or equality) of a linear program instance.

Namespaces

- channel

  The namespace channel contains the definition and implementation of a set of classes for threading a cubic b-spline curve into a given planar channel delimited by two polygonal chains.

17.3.1 Detailed Description

Definition of a class for representing a nonzero coefficient of an unknown of a constraint (inequality or equality) of a linear program instance.

Author

Marcelo Ferreira Siqueira
Universidade Federal do Rio Grande do Norte,
Departamento de Matemática,
mfsiqueira at mat (dot) ufrn (dot) br

Version

1.0

Date

March 2016

Attention

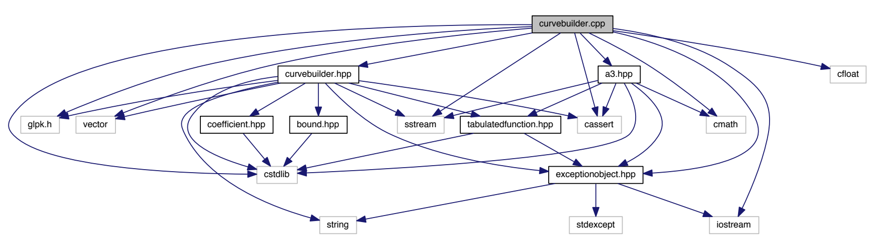
This program is distributed WITHOUT ANY WARRANTY, and it may be freely redistributed under the condition that the copyright notices are not removed, and no compensation is received. Private, research, and institutional use is free. Distribution of this code as part of a commercial system is permissible ONLY BY DIRECT ARRANGEMENT WITH THE AUTHOR.

17.4 curvebuilder.cpp File Reference

Implementation of a class for threading a b-spline curve of degree 3 through a planar channel defined by a pair of polygonal chains.

```
#include "curvebuilder.hpp"
#include "exceptionobject.hpp"
#include "a3.hpp"
#include "glpk.h"
#include <cmath>
#include <cassert>
#include <sstream>
#include <iostream>
#include <vector>
#include <cfloat>
#include <cstdlib>
```

Include dependency graph for curvebuilder.cpp:



Namespaces

- channel

  The namespace channel contains the definition and implementation of a set of classes for threading a cubic b-spline curve into a given planar channel delimited by two polygonal chains.

17.4.1 Detailed Description

Implementation of a class for threading a b-spline curve of degree 3 through a planar channel defined by a pair of polygonal chains.

Author

Marcelo Ferreira Siqueira
Universidade Federal do Rio Grande do Norte,
Departamento de Matemática,
mfsiqueira at mat (dot) ufrn (dot) br

Version

1.0

Date

May 2016

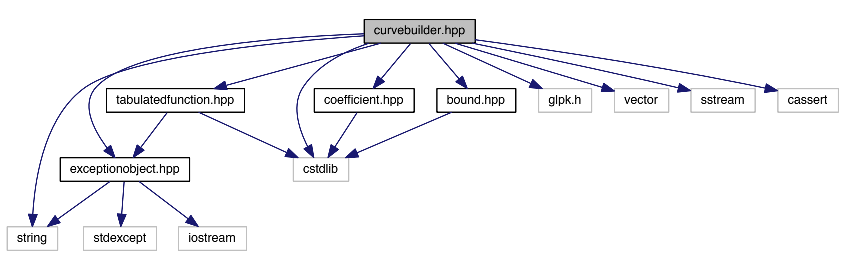
Attention

17.5 curvebuilder.hpp File Reference

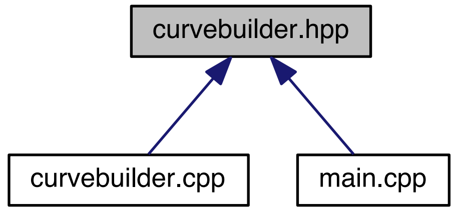
Definition of a class for threading a b-spline curve of degree 3 through a planar channel defined by a pair of polygonal chains.

```
#include "exceptionobject.hpp"
#include "tabulatedfunction.hpp"
#include "coefficient.hpp"
#include "bound.hpp"
#include "glpk.h"
#include <vector>
#include <string>
#include <sstream>
#include <cassert>
#include <cstdlib>
```

Include dependency graph for curvebuilder.hpp:



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



Classes

- class channel::CurveBuilder

    This class provides methods for threading a cubic b-spline curve through a planar channel delimited by a pair of polygonal chains.

Namespaces

- channel

    The namespace channel contains the definition and implementation of a set of classes for threading a cubic b-spline curve into a given planar channel delimited by two polygonal chains.

17.5.1 Detailed Description

Definition of a class for threading a b-spline curve of degree 3 through a planar channel defined by a pair of polygonal chains.

Author

Marcelo Ferreira Siqueira
Universidade Federal do Rio Grande do Norte,
Departamento de Matemática,
mfsiqueira at mat (dot) ufrn (dot) br

Version

1.0

Date

May 2016

Attention

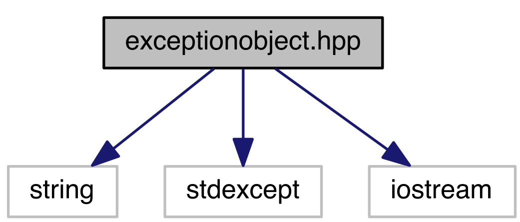
This program is distributed WITHOUT ANY WARRANTY, and it may be freely redistributed under the condition that the copyright notices are not removed, and no compensation is received. Private, research, and institutional use is free. Distribution of this code as part of a commercial system is permissible ONLY BY DIRECT ARRANGEMENT WITH THE AUTHOR.

17.6 exceptionobject.hpp File Reference

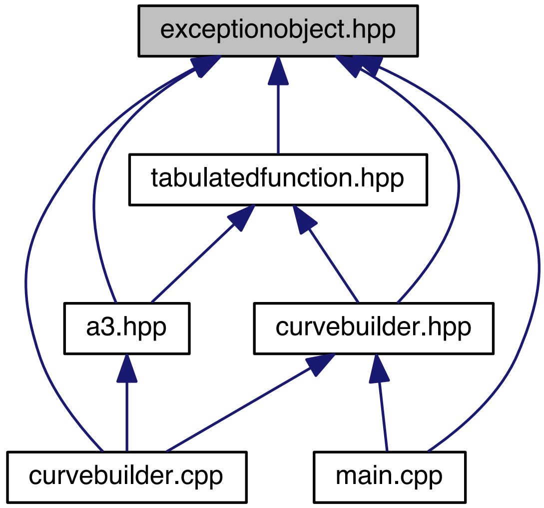
Definition of a class for handling exceptions.

```
#include <string>
#include <stdexcept>
#include <iostream>
```

Include dependency graph for exceptionobject.hpp:



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



Classes

- class channel::ExceptionObject

  This class extends class exception of STL and provides us with a customized way of handling exceptions and showing error messages.

Namespaces

- channel

  The namespace channel contains the definition and implementation of a set of classes for threading a cubic b-spline curve into a given planar channel delimited by two polygonal chains.

Macros

- #define treat_exception(e)

  Prints out the description of the error that caused an exception as well as the file containing the instruction that threw the exception and the line of the instruction in the file.

17.6.1 Detailed Description

Definition of a class for handling exceptions.

Author

Marcelo Ferreira Siqueira
Universidade Federal do Rio Grande do Norte,
Departamento de Matemática,
mfsiqueira at mat (dot) ufrn (dot) br

Version

1.0

Date

March 2016

Attention

17.6.2 Macro Definition Documentation

17.6.2.1 #define treat_exception(*e*)

Value:

```
std::cerr << std::endl \
          << "Exception: " << e.get_description() << std::endl \
          << "File: "      << e.get_file()        << std::endl \
          << "Line: "      << e.get_line()        << std::endl \
          << std::endl ;
```

Prints out the description of the error that caused an exception as well as the file containing the instruction that threw the exception and the line of the instruction in the file.

Parameters

e	An exception.

Definition at line 42 of file exceptionobject.hpp.

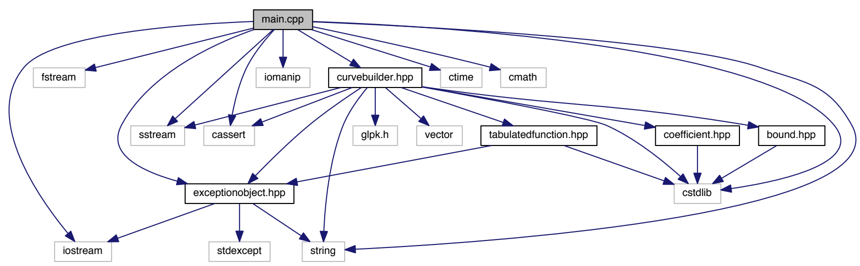
Referenced by channel::CurveBuilder::evaluate_bounding_polynomial(), main(), write_lp(), and write_solution().

17.7 main.cpp File Reference

A simple program for testing the channel-2d library.

```
#include <iostream>
#include <fstream>
#include <sstream>
#include <string>
#include <cstdlib>
#include <iomanip>
#include <cassert>
#include <ctime>
#include <cmath>
#include "exceptionobject.hpp"
#include "curvebuilder.hpp"
```

Include dependency graph for main.cpp:



Functions

- void read_input (const string &fn, size_t &np, size_t &nc, bool &closed, double ∗&lx, double ∗&ly, double ∗&ux, double ∗&uy) throw (ExceptionObject)

  Read in a file describing a polygonal channel.
- void write_solution (const string &fn, const CurveBuilder &b)

  Write the control points of the b-spline curve to an output file.
- void write_lp (const string &fn, const CurveBuilder &b)

  Write the instance of the linear program problem solved by this program in CPLEX format. The output file can be given to the gpsolve function of the GNU GLPK or to debug the assembly of the constraints.
- int main (int argc, char ∗argv[ ])

  A simple program for testing the bc2d library.

17.7.1 Detailed Description

A simple program for testing the channel-2d library.

Author

Marcelo Ferreira Siqueira
Universidade Federal do Rio Grande do Norte,
Departamento de Matemática,
mfsiqueira at mat (dot) ufrn (dot) br

Version

1.0

Date

March 2016

Attention

This program is distributed WITHOUT ANY WARRANTY, and it may be freely redistributed under the condition that the copyright notices are not removed, and no compensation is received. Private, research, and institutional use is free. Distribution of this code as part of a commercial system is permissible ONLY BY DIRECT ARRANGEMENT WITH THE AUTHOR.

17.7.2 Function Documentation

17.7.2.1 int main (int *argc*, char * *argv[ ]*)

A simple program for testing the bc2d library.

Parameters

argc	The number of command-line arguments.
argv	An array with the command-line arguments.

Returns

An integer number.

Definition at line 125 of file main.cpp.

References channel::CurveBuilder::build(), channel::CurveBuilder::get_solver_error_message(), read_input(), treat_exception, write_lp(), and write_solution().

```
125                                     {
126   //
127   // Check command-line arguments.
128   //
129
130   if ( ( argc != 3 ) && ( argc != 4 ) ) {
131     cerr << "Usage: "
132          << endl
133          << "\t\t channel2d arg1 arg2 [ arg3 ]"
134          << endl
135          << "\t\t arg1: name of the file describing the polygonal channel"
136          << endl
137          << "\t\t arg2: name of the output file describing the computed cubic b-spline curve"
138          << endl
139          << "\t\t arg3: name of an output file to store a CPLEX format definition of the LP solved by this
      program (OPTIONAL)"
140          << endl
141          << endl ;
142     cerr.flush() ;
143
144     return EXIT_FAILURE ;
145   }
146
147   //
148   // Read in the input file.
149   //
150
151   clock_t start, end ;
152
```

```
cerr << endl
     << "Reading file describing a polygonal channel..."
     << endl ;
cerr.flush() ;

string fn1( argv[ 1 ] ) ;

size_t np ;
size_t nc ;
bool closed ;
double* lx ;
double* ly ;
double* ux ;
double* uy ;

start = clock() ;
try {
  read_input( fn1 , np , nc , closed , lx , ly , ux , uy ) ;
}
catch ( const ExceptionObject& xpt ) {
  treat_exception( xpt ) ;
  exit( EXIT_FAILURE ) ;
}
end = clock() ;

cerr << ( (double) ( end - start ) ) / CLOCKS_PER_SEC
     << " seconds."
     << endl
     << endl ;
cerr.flush() ;

//
// Compute a cubic b-spline curve that passes through the channel.
//

cerr << "Computing a cubic b-spline curve that passes through the channel... "
     << endl ;
cerr.flush() ;

start = clock() ;
CurveBuilder* builder = 0 ;
try {
  builder = new CurveBuilder(
                             np ,
                             nc ,
                             closed ,
                             &lx[ 0 ] ,
                             &ly[ 0 ] ,
                             &ux[ 0 ] ,
                             &uy[ 0 ]
                            ) ;
}
catch ( const ExceptionObject& xpt ) {
  treat_exception( xpt ) ;
  exit( EXIT_FAILURE ) ;
}

int error ;
bool res = builder->build( error ) ;
end = clock() ;

cerr << ( (double) ( end - start ) ) / CLOCKS_PER_SEC
     << " seconds."
     << endl
     << endl ;
cerr.flush() ;

if ( res ) {
  //
  // Write the control points of the b-spline to a file.
  //
  cerr << "Writing out the control points of the b-spline to a file..."
             << endl ;
  cerr.flush() ;

  start = clock() ;
  string fn2( argv[ 2 ] ) ;
  write_solution(
                 fn2 ,
                 *builder
                ) ;
```

```
    end = clock() ;

    cerr << ( (double) ( end - start ) ) / CLOCKS_PER_SEC
         << " seconds."
         << endl
         << endl ;
    cerr.flush() ;
  }
  else {
    //
    // Print the error message returned by the LP solver.
    //
    cerr << endl
         << "ATTENTION: "
         << endl
         << builder->get_solver_error_message( error )
         << endl
         << endl ;
  }

  //
  // Generate a description of the linear program in CPLEX format.
  //
  if ( argc == 4 ) {
    cerr << "Writing out a description of the linear program in CPLEX format..."
               << endl ;
    cerr.flush() ;

    start = clock() ;
    string fn3( argv[ 3 ] ) ;
    write_lp(
             fn3 ,
             *builder
            ) ;
    end = clock() ;

    cerr << ( (double) ( end - start ) ) / CLOCKS_PER_SEC
         << " seconds."
         << endl
         << endl ;
    cerr.flush() ;
  }

  //
  // Release memory
  //

  cerr << "Releasing memory..."
       << endl ;
  cerr.flush() ;

  start = clock() ;
  if ( lx != 0 ) delete[ ] lx ;
  if ( ly != 0 ) delete[ ] ly ;
  if ( ux != 0 ) delete[ ] ux ;
  if ( uy != 0 ) delete[ ] uy ;
  if ( builder != 0 ) delete builder ;
  end = clock() ;

  cerr << ( (double) ( end - start ) ) / CLOCKS_PER_SEC
  << " seconds."
  << endl
  << endl ;
  cerr.flush() ;

  //
  // Done.
  //

  cerr << "Finished."
       << endl
       << endl
       << endl ;
  cerr.flush() ;

  return EXIT_SUCCESS ;
}
```

17.7.2.2 void read_input (const string & *fn,* size_t & *np,* size_t & *nc,* bool & *closed,* double ∗& *lx,* double ∗& *ly,* double ∗& *ux,* double ∗& *uy*) throw **ExceptionObject**)

Read in a file describing a polygonal channel.

Parameters

fn	The name of a file describing a polygonal channel.
np	A reference to the number of b-spline segments.
nc	A reference to the number of c-segments of the channel.
closed	A reference to a flag to indicate whether the channel is closed.
lx	A reference to a pointer to an array with the x-coordinates of the lower polygonal chain of the channel.
ly	A reference to a pointer to an array with the y-coordinates of the lower polygonal chain of the channel.
ux	A reference to a pointer to an array with the x-coordinates of the upper polygonal chain of the channel.
uy	A reference to a pointer to an array with the y-coordinates of the upper polygonal chain of the channel.

Definition at line 333 of file main.cpp.

Referenced by main().

```
{
  //
  // Open the input file
  //
  std::ifstream in( fn.c_str() ) ;

  if ( in.is_open() ) {
    //
    // Read in the number of segments of the b-spline.
    //
    in >> np ;

    //
    // Read in the number of c-segments of the channel.
    //
    in >> nc ;

    //
    // Read in the flag indicating whether the channel is closed.
    //
    unsigned flag ;
    in >> flag ;

    if ( ( flag != 0 ) && ( flag != 1 ) ) {
      std::stringstream ss( std::stringstream::in | std::stringstream::out ) ;
      ss << "Flag value indicating whether the channel is closed or open is invalid" ;
      in.close() ;
      throw ExceptionObject( __FILE__ , __LINE__ , ss.str().c_str() ) ;
    }

    closed = ( flag == 1 ) ;

    if ( closed ) {
      if ( np < 4 ) {
        std::stringstream ss( std::stringstream::in | std::stringstream::out ) ;
        ss << "The number of curve segments must be at least 4 for a closed curve" ;
        in.close() ;
        throw ExceptionObject( __FILE__ , __LINE__ , ss.str().c_str() ) ;
      }
      if ( nc < 3 ) {
        std::stringstream ss( std::stringstream::in | std::stringstream::out ) ;
        ss << "The number of segments of a closed channel must be at least 3" ;
        in.close() ;
        throw ExceptionObject( __FILE__ , __LINE__ , ss.str().c_str() ) ;
      }
    }
    else {
      if ( np < 1 ) {
        std::stringstream ss( std::stringstream::in | std::stringstream::out ) ;
        ss << "The number of curve segments must be at least 1" ;
        in.close() ;
        throw ExceptionObject( __FILE__ , __LINE__ , ss.str().c_str() ) ;
      }
      if ( nc < 1 ) {
        std::stringstream ss( std::stringstream::in | std::stringstream::out ) ;
```

```
        ss << "The number of segments of an open channel must be at least 1" ;
        in.close() ;
        throw ExceptionObject( __FILE__ , __LINE__ , ss.str().c_str() ) ;
      }
    }

    //
    // Read in the channel vertex coordinates.
    //
    const size_t nn = ( closed ) ? nc : ( nc + 1 ) ;

    lx = new double[ nn ] ;
    ly = new double[ nn ] ;

    for ( size_t i = 0 ; i < nn ; i++ ) {
      //
      // Read in the X and Y coordinates of the i-th vertex.
      //
      in >> lx[ i ] ;
      in >> ly[ i ] ;
    }

    ux = new double[ nn ] ;
    uy = new double[ nn ] ;

    for ( size_t i = 0 ; i < nn ; i++ ) {
      //
      // Read in the X and Y coordinates of the i-th vertex.
      //
      in >> ux[ i ] ;
      in >> uy[ i ] ;
    }

    //
    // Close file
    //

    in.close() ;
  }

  return ;
}
```

17.7.2.3 void write_lp (const string & *fn*, const **CurveBuilder** & *b*)

Write the instance of the linear program problem solved by this program in CPLEX format. The output file can be given to the *gpsolve* function of the GNU GLPK or to debug the assembly of the constraints.

Parameters

fn	The name of the output file.
b	An instance of the spline curve builder.

Definition at line 518 of file main.cpp.

References channel::CurveBuilder::get_bound_of_ith_constraint(), channel::CurveBuilder::get_coefficient_identifier(), channel::CurveBuilder::get_coefficient_value(), channel::CurveBuilder::get_degree(), channel::CurveBuilder::get_number_of_coefficients_in_the_ith_constraint(), channel::CurveBuilder::get_number_of_constraints(), channel::CurveBuilder::get_number_of_csegments(), channel::CurveBuilder::get_number_of_segments(), channel::CurveBuilder::is_curve_closed(), channel::CurveBuilder::is_equality(), channel::CurveBuilder::is_greater_than_or_equal_to(), channel::CurveBuilder::is_less_than_or_equal_to(), and treat_exception.

Referenced by main().

```
{
  //
  // Create the output file
  //

  const size_t np = b.get_number_of_segments() ;
  const size_t dg = b.get_degree() ;
```

```
const size_t NumberOfControlPoints = np + dg ;
const size_t NumberOfSecondDifferences = ( np - 1 ) * ( dg - 2 ) + ( dg - 1 ) ;

std::ofstream ou( fn.c_str() ) ;

if ( ou.is_open() ) {
  //
  // Set the precision of the floating-point numbers.
  //

  ou << std::setprecision( 6 ) << std::fixed ;

  //
  // Write the objective function
  //

  ou << "Minimize"
     << std::endl ;
  ou << '\t'
     << "obj: " ;

  size_t j = 1 ;
  for ( size_t i = 0 ; i < NumberOfSecondDifferences ; i++ ) {
    ou << "-mx" << j << " " ;
    ou << "-my" << j << " " ;
    ou << "+px" << j << " " ;
    ou << "+py" << j << " " ;
    ++j ;
  }

  ou << std::endl ;

  //
  // Write the constraints
  //

  ou << "Subject To" << std::endl ;

  try {

    for( size_t i = 0 ; i < b.get_number_of_constraints() ; i++ ) {
      //
      // Write out the number of the constraint.
      //
      ou << '\t' << "c" << i + 1 << ": " ;

      //
      // Get the coefficients of the i-th constraint.
      //
      for( j = 0 ; j < b.get_number_of_coefficients_in_the_ith_constraint
( i ) ; ++j ) {
        //
        // Get the column index of the coefficient.
        //
        size_t col = b.get_coefficient_identifier( i , j ) ;

        //
        // Get the value of the coefficient.
        //
        double value = b.get_coefficient_value( i , j ) ;

        //
        // Compute the  index of the  curve piece associated  with the
        // coefficient, and  find the type of  structural variable the
        // coefficient is.
        //
        if ( col < ( 2 * NumberOfControlPoints ) ) {
          if ( ( col % 2 ) == 0 ) {
            if ( value >= 0 ) {
              ou << "+" << value << "x" << ( col >> 1 ) + 1 << " " ;
            }
            else {
              ou << value << "x" << ( col >> 1 ) + 1 << " " ;
            }
          }
          else {
            if ( value >= 0 ) {
              ou << "+" << value << "y" << ( col >> 1 ) + 1 << " " ;
            }
            else {
```

```
                ou << value << "y" << ( col >> 1 ) + 1 << " " ;
              }
            }
          }
          else if ( col < ( ( 2 * NumberOfControlPoints ) + ( 4 * NumberOfSecondDifferences ) ) ) {
            col -= ( 2 * NumberOfControlPoints ) ;
            size_t ro = col % 4 ;
            if ( ro == 0 ) {
              if ( value >= 0 ) {
                ou << "+" << value << "mx" << ( col / 4 ) + 1 << " " ;
              }
              else {
                ou << value << "mx" << ( col / 4 ) + 1 << " " ;
              }
            }
            else if ( ro == 1 ) {
              if ( value >= 0 ) {
                ou << "+" << value << "my" << ( col / 4 ) + 1 << " " ;
              }
              else {
                ou << value << "my" << ( col / 4 ) + 1 << " " ;
              }
            }
            else if ( ro == 2 ) {
              if ( value >= 0 ) {
                ou << "+" << value << "px" << ( col / 4 ) + 1 << " " ;
              }
              else {
                ou << value << "px" << ( col / 4 ) + 1 << " " ;
              }
            }
            else if ( ro == 3 ) {
              if ( value >= 0 ) {
                ou << "+" << value << "py" << ( col / 4 ) + 1 << " " ;
              }
              else {
                ou << value << "py" << ( col / 4 ) + 1 << " " ;
              }
            }
          }
          else {
            col -= ( ( 2 * NumberOfControlPoints ) + ( 4 * NumberOfSecondDifferences ) ) ;

            if ( col == 0 ) {
              if ( value >= 0 ) {
                ou << "+" << value << "as" << " " ;
              }
              else if ( value < 0 ) {
                ou << value << "as" << " " ;
              }
            }
            else if ( ( col == 1 ) && !b.is_curve_closed() ) {
              if ( value >= 0 ) {
                ou << "+" << value << "ae" << " " ;
              }
              else if ( value < 0 ) {
                ou << value << "ae" << " " ;
              }
            }
            else if ( !b.is_curve_closed() ) {
              if ( value >= 0 ) {
                ou << "+" << value << "co" << ( col - 1 ) << " " ;
              }
              else if ( value < 0 ) {
                ou << value << "co" << ( col - 1 ) << " " ;
              }
            }
            else {
              if ( value >= 0 ) {
                ou << "+" << value << "co" << col << " " ;
              }
              else if ( value < 0 ) {
                ou << value << "co" << col << " " ;
              }
            }
          }
        }

        if ( b.is_equality( i ) ) {
          ou << " = " ;
        }
```

```
        else if ( b.is_less_than_or_equal_to( i ) ) {
          ou << " <= " ;
        }
        else {
#ifdef DEBUGMODE
          assert( b.is_greater_than_or_equal_to( i ) ) ;
#endif
          ou << " >= " ;
        }

        ou << b.get_bound_of_ith_constraint( i ) << std::endl ;
      }

    }
    catch ( const ExceptionObject& xpt ) {
      treat_exception( xpt ) ;
      exit( EXIT_FAILURE ) ;
    }

    //
    // Write the bounds
    //

    ou << "Bounds" << std::endl ;

    for ( unsigned k = 0 ; k < NumberOfControlPoints ; k++ ) {
      ou << '\t' << "x" << k + 1 << " free" << std::endl ;
      ou << '\t' << "y" << k + 1 << " free" << std::endl ;
    }

    for ( unsigned k = 0 ; k < NumberOfSecondDifferences ; k++ ) {
      ou << '\t' << "-inf <= mx" << k + 1 <<    " <= 0" << std::endl ;
      ou << '\t' << "-inf <= my" << k + 1 <<    " <= 0" << std::endl ;
      ou << '\t' <<     "0 <= px" << k + 1 << " <= +inf" << std::endl ;
      ou << '\t' <<     "0 <= py" << k + 1 << " <= +inf" << std::endl ;
    }

    ou << '\t' << 0.40 << " <= as <= " << 0.60 << std::endl ;

    if ( !b.is_curve_closed() ) {
      ou << '\t' << 0.40 << " <= ae <= " << 0.60 << std::endl ;
    }

    const size_t NumberOfCSegments = b.get_number_of_csegments() ;

    for ( unsigned k = 1 ; k < NumberOfCSegments ; k++ ) {
      ou << '\t'
      << 0.40
      << " <= co"
      << k
      << " <= "
      << 0.60
      << std::endl ;
    }

    ou << "End" << std::endl ;

    //
    // Close file
    //

    ou.close() ;
  }

  return ;
}
```

17.7.2.4 void write_solution (const string & *fn*, const CurveBuilder & *b*)

Write the control points of the b-spline curve to an output file.

Parameters

fn	The name of the output file.
b	An instance of the b-spline curve builder.

Definition at line 453 of file main.cpp.

References channel::CurveBuilder::get_control_value(), channel::CurveBuilder::get_number_of_control_points(), and treat_exception.

Referenced by main().

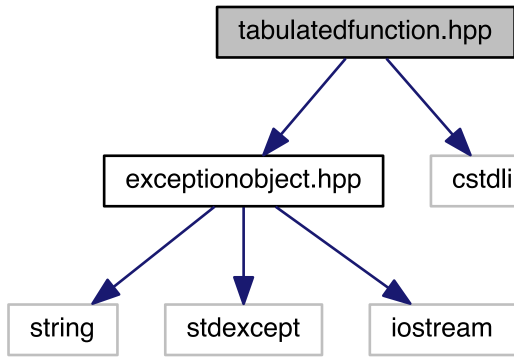
```
{
  using std::endl ;

  std::ofstream ou( fn.c_str() ) ;

  if ( ou.is_open() ) {
    //
    // Set the precision of the floating-point numbers.
    //

    ou << std::setprecision( 6 ) << std::fixed ;

    //
    // Write out the number of control points and the degree of the b-spline.
    //

    size_t ncp = b.get_number_of_control_points() ;

    ou << ncp
       << '\t'
       << 3
       << endl ;

    for ( size_t i = 0 ; i < ncp ; i++ ) {
      double x ;
      double y ;
      try {
        x = b.get_control_value( i , 0 ) ;
        y = b.get_control_value( i , 1 ) ;
      }
      catch ( const ExceptionObject& xpt ) {
        treat_exception( xpt ) ;
        ou.close() ;
        exit( EXIT_FAILURE ) ;
      }
      ou << x << '\t' << y << endl ;
    }

    //
    // Close file
    //

    ou.close() ;
  }

  return ;
}
```

17.8 tabulatedfunction.hpp File Reference

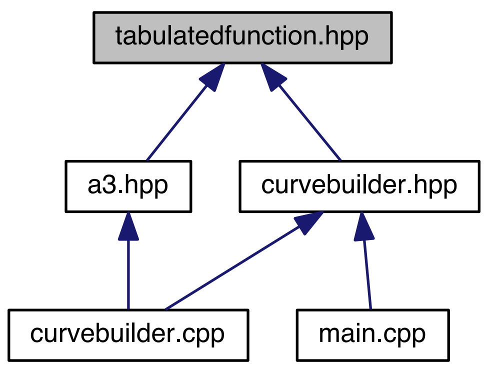
Definition of an abstract class for representing piecewise linear enclosures of certain polynomial functions of arbitrary degree.

```
#include "exceptionobject.hpp"
#include <cstdlib>
```

Include dependency graph for tabulatedfunction.hpp:



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



Classes

- class channel::TabulatedFunction

  This class represents two-sided, piecewise linear enclosures of a set of $(d-1)$ *polynomial functions of degree* d *in Bézier form. The enclosures must be made available by implementating a pure virtual method in derived classes.*

Namespaces

- channel

The namespace channel contains the definition and implementation of a set of classes for threading a cubic b-spline curve into a given planar channel delimited by two polygonal chains.

17.8.1 Detailed Description

Definition of an abstract class for representing piecewise linear enclosures of certain polynomial functions of arbitrary degree.

Author

Marcelo Ferreira Siqueira
Universidade Federal do Rio Grande do Norte,
Departamento de Matemática,
mfsiqueira at mat (dot) ufrn (dot) br

Version

1.0

Date

March 2016

Attention

Index