Threading Splines Through 2D Channels

1.0

Generated by Doxygen 1.8.10

Fri Mar 25 2016 22:52:59

Contents

1 The BC2 Library Documentation **1**

2 Introduction **3**

3 Installing and compiling the library **7**

4 The BuildCurve2d class API **11**

5 Using the library API **15**

6 Examples and file formats **19**

7 License **23**

8 Acknowledgements **25**

9 Module Index **27**

9.1 Modules 27

10 Namespace Index **29**

10.1 Namespace List 29

11 Hierarchical Index **31**

11.1 Class Hierarchy 31

12 Class Index **33**

12.1 Class List 33

13 File Index **35**

13.1 File List 35

14 Module Documentation **37**

14.1 Namespace bc2. 37

14.1.1 Detailed Description 37

15 Namespace Documentation **39**

15.1 bc2 Namespace Reference . . . 39

15.1.1 Detailed Description . . . 39

16 Class Documentation **41**

16.1 bc2::a3 Class Reference . . . 41

16.1.1 Detailed Description . . . 43

16.1.2 Member Function Documentation . . . 43

16.1.2.1 a(unsigned i, double u) const . . . 43

16.1.2.2 a1(double u) const . . . 44

16.1.2.3 a1lower(double u) const . . . 44

16.1.2.4 a1upper(double u) const . . . 45

16.1.2.5 alower(unsigned i, double u) const . . . 46

16.1.2.6 aupper(unsigned i, double u) const . . . 47

16.1.2.7 degree() const . . . 48

16.1.2.8 h(double u) const . . . 48

16.2 bc2::Bound Class Reference . . . 49

16.2.1 Detailed Description . . . 50

16.2.2 Constructor & Destructor Documentation . . . 50

16.2.2.1 Bound(CONSTRAINTYPE type, double value, unsigned row) . . . 50

16.2.2.2 Bound(const Bound &b) . . . 50

16.2.3 Member Function Documentation . . . 51

16.2.3.1 get_row() const . . . 51

16.2.3.2 get_type() const . . . 51

16.2.3.3 get_value() const . . . 51

16.3 bc2::BuildCurve2D Class Reference . . . 52

16.3.1 Detailed Description . . . 55

16.3.2 Constructor & Destructor Documentation . . . 55

16.3.2.1 BuildCurve2D(unsigned np, unsigned nc, bool closed, double *lx, double *ly, double *ux, double *uy, TabulatedFunction *tf) . . . 55

16.3.2.2 BuildCurve2D(const BuildCurve2D &b) . . . 56

16.3.3 Member Function Documentation . . . 56

16.3.3.1 build(int &error) . . . 56

16.3.3.2 compute_c0continuity_constraints(unsigned &eqline) . . . 57

16.3.3.3 compute_c1continuity_constraints(unsigned &eqline) . . . 58

16.3.3.4 compute_channel_corners_outside_sleeve_constraints(unsigned &eqline) . . . 60

16.3.3.5 compute_control_value_column_index(unsigned p, unsigned i, unsigned v) const . . . 65

16.3.3.6 compute_correspondence_constraints(unsigned &eqline) . . . 66

16.3.3.7 compute_min_max_constraints(unsigned &eqline) . . . 67
16.3.3.8 compute_normal_to_lower_envelope(sizetype s, double &nx, double &ny) const . . . 68
16.3.3.9 compute_normal_to_upper_envelope(sizetype s, double &nx, double &ny) const . . . 69
16.3.3.10 compute_second_difference_column_index(unsigned p, unsigned i, unsigned l, unsigned v) const . . . 69
16.3.3.11 compute_sleeve_corners_in_channel_constraints(unsigned &eqline) . . . 70
16.3.3.12 get_bound_of_ith_constraint(unsigned i) const . . . 74
16.3.3.13 get_coefficient_identifier(unsigned i, unsigned j) const . . . 74
16.3.3.14 get_coefficient_value(unsigned i, unsigned j) const . . . 75
16.3.3.15 get_control_value(unsigned p, unsigned i, unsigned v) const . . . 75
16.3.3.16 get_lower_bound_on_second_difference_value(unsigned p, unsigned i, unsigned v) const . . . 76
16.3.3.17 get_lp_solver_result_information(glp_prob *lp) . . . 77
16.3.3.18 get_number_of_coefficients_in_the_ith_constraint(unsigned i) const . . . 78
16.3.3.19 get_number_of_constraints() const . . . 78
16.3.3.20 get_number_of_curve_pieces() const . . . 79
16.3.3.21 get_solver_error_message(int error) . . . 79
16.3.3.22 get_spline_degree() const . . . 80
16.3.3.23 get_upper_bound_on_second_difference_value(unsigned p, unsigned i, unsigned v) const . . . 80
16.3.3.24 h(double u) const . . . 81
16.3.3.25 is_equality(unsigned i) const . . . 82
16.3.3.26 is_greater_than_or_equal_to(unsigned i) const . . . 82
16.3.3.27 is_less_than_or_equal_to(unsigned i) const . . . 83
16.3.3.28 lf(double u, double b0, double bd) const . . . 84
16.3.3.29 minimum_value() const . . . 84
16.3.3.30 set_up_lp_constraints(glp_prob *lp) const . . . 84
16.3.3.31 set_up_objective_function(glp_prob *lp) const . . . 85
16.3.3.32 set_up_structural_variables(glp_prob *lp) const . . . 86
16.3.3.33 solve_lp(sizetype rows, sizetype cols) . . . 87
16.4 bc2::Coefficient Class Reference . . . 88
16.4.1 Detailed Description . . . 89
16.4.2 Constructor & Destructor Documentation . . . 89
16.4.2.1 Coefficient(unsigned row, unsigned col, double value) . . . 89
16.4.2.2 Coefficient(const Coefficient &c) . . . 89
16.4.3 Member Function Documentation . . . 90
16.4.3.1 get_col() const . . . 90
16.4.3.2 get_row() const . . . 90

16.4.3.3 get_value() const . . . 90
16.5 bc2::ExceptionObject Class Reference . . . 91
16.5.1 Detailed Description . . . 93
16.5.2 Constructor & Destructor Documentation . . . 93
16.5.2.1 ExceptionObject(const char *file, unsigned ln) . . . 93
16.5.2.2 ExceptionObject(const char *file, unsigned int ln, const char *desc) . . . 93
16.5.2.3 ExceptionObject(const char *file, unsigned ln, const char *desc, const char *loc) . . . 94
16.5.2.4 ExceptionObject(const ExceptionObject &xpt) . . . 94
16.5.3 Member Function Documentation . . . 94
16.5.3.1 get_description() const . . . 95
16.5.3.2 get_file() const . . . 95
16.5.3.3 get_line() const . . . 95
16.5.3.4 get_location() const . . . 95
16.5.3.5 get_name_of_class() const . . . 96
16.5.3.6 set_description(const std::string &s) . . . 96
16.5.3.7 set_description(const char *s) . . . 96
16.5.3.8 set_location(const std::string &s) . . . 96
16.5.3.9 set_location(const char *s) . . . 97
16.5.3.10 what() const . . . 98
16.6 bc2::TabulatedFunction Class Reference . . . 98
16.6.1 Detailed Description . . . 99
16.6.2 Member Function Documentation . . . 99
16.6.2.1 a(unsigned i, double u) const =0 . . . 99
16.6.2.2 alower(unsigned i, double u) const =0 . . . 100
16.6.2.3 aupper(unsigned i, double u) const =0 . . . 101
16.6.2.4 degree() const =0 . . . 101

17 File Documentation **103**

17.1 a3.hpp File Reference . . . 103
17.1.1 Detailed Description . . . 104
17.2 bound.hpp File Reference . . . 105
17.2.1 Detailed Description . . . 105
17.3 buildcurve2d.cpp File Reference . . . 106
17.3.1 Detailed Description . . . 107
17.4 buildcurve2d.hpp File Reference . . . 107
17.4.1 Detailed Description . . . 108
17.5 coefficient.hpp File Reference . . . 109

17.5.1 Detailed Description . . . 109
17.6 exceptionobject.hpp File Reference . . . 110
17.6.1 Detailed Description . . . 111
17.6.2 Macro Definition Documentation . . . 112
17.6.2.1 treat_exception . . . 112
17.7 main.cpp File Reference . . . 112
17.7.1 Detailed Description . . . 113
17.7.2 Function Documentation . . . 114
17.7.2.1 main(int argc, char ∗argv[]) . . . 114
17.7.2.2 read_input(const string &fn, unsigned &np, unsigned &nc, bool &closed, unsigned &dg, double ∗&lx, double ∗&ly, double ∗&ux, double ∗&uy) . . . 117
17.7.2.3 write_lp(const string &fn, const BuildCurve2D &b) . . . 118
17.7.2.4 write_solution(const string &fn, const BuildCurve2D &b) . . . 121
17.8 tabulatedfunction.hpp File Reference . . . 122
17.8.1 Detailed Description . . . 123

Index **125**

Chapter 1

The BC2 Library Documentation

- Introduction
- Installing and compiling the library
- The BuildCurve2d class API
- Using the library API
- Examples and file formats
- License
- Acknowledgements

Chapter 2

Introduction

The BC2 Library consists of a set of C++ classes for solving two-dimensional instances of the **channel problem**. A detailed description of the channel problem and its solution (as I implemented in the BC2 library) can be found in the following paper:

- 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 the paper (at least its Section 3) before you try to use the BC2 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,

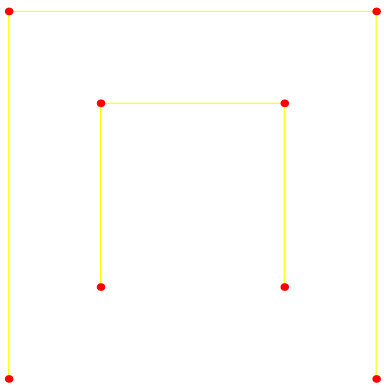


Figure 2.1: Example of a channel

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, any two corresponding edges do not have to be parallel.

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

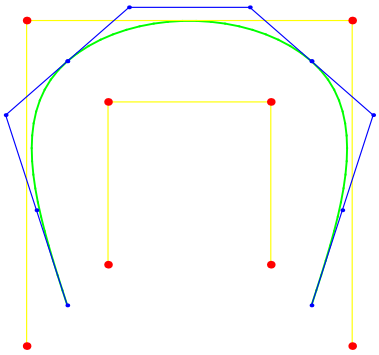


Figure 2.2: Example of a solution for the channel problem

The spline curve in shown in green and its control polygon is shown in blue. Myles and Peters modeled the solution of 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 `BC2` library, we adopt the same objective function given in Myles and Peters' paper, which aims at minimizing the total variation of curvature. 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.

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

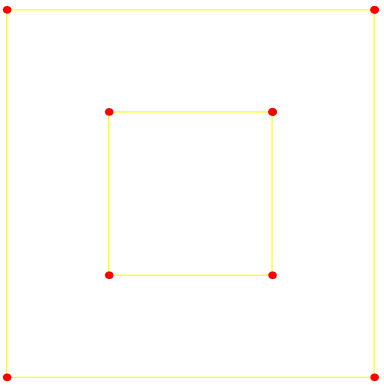


Figure 2.3: Example of a channel

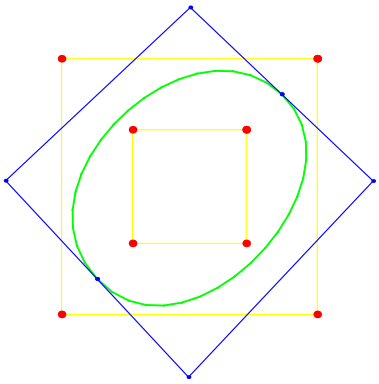


Figure 2.4: Example of a solution for the channel problem

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 thre values of three parameters: np, nc, and d. Parameter np specifies the number of Bézier curves that make up the final spline curve. Parameter nc specifies the number of *c-sections* of the channel that delimit each of the np Bézier curves. A *c-section* is a subset of consecutive and corresponding edges of the lower and upper envelopes. If the channel is *open*, then we must have that $n = np \times nc$, where n is the number of edges of either envelope of the channel. Note that the number of vertices in each envelope is $n + 1$ in this case. If the channel is *closed*, then we must have that $n + 1 = np \times nc$.

For the first example of the channel problem I showed above, we have $np = 3$ and $nc = 1$:

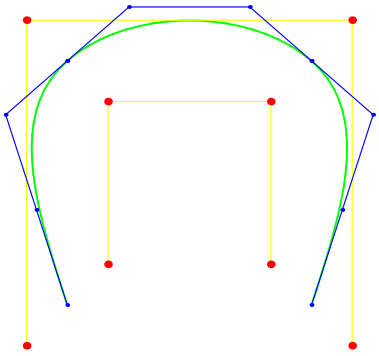


Figure 2.5: Example of a channel

That is, the spline consists of exactly $np = 3$ Bézier curves, each of which is bounded (above and below) by $nc = 1$ pairs of corresponding edges of the channel: an edge of the lower envelope and an edge of the upper envelope. Observe that each envelope has exactly $n = 3$ edges.

For the second example of the channel problem I showed above, we have $np = 2$ and $nc = 2$:

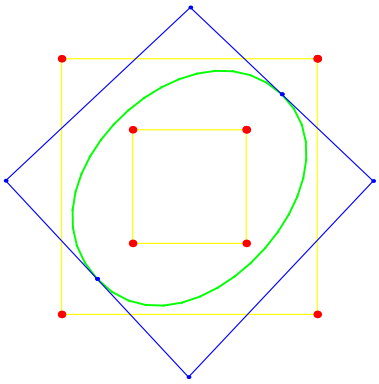


Figure 2.6: Example of a solution for the channel problem

That is, the spline consists of exactly $np = 2$ Bézier curves, each of which is bounded (above and below) by $nc = 2$ pairs of corresponding edges of the channel: an edge of the lower envelope and an edge of the upper envelope. Observe that each envelope has exactly $(n + 1) = 4$ edges.

Finally, we must also specify the parameter d, which determines the degree of each Bézier curve defining the spline. In both examples I showed above, I set $d = 3$. So, the spline consists of a set of cubic polynomials in Bézier form. Any two consecutives Bézier curves join each other with C^1 continuity. This continuity constraint may sometimes be too restrictive for the problem to have a feasible solution. This is often the case when the channel is very narrow, its c-sections meet at sharp angles, and the degree d of the spline is low. If an instance of the channel problem has no solution, the main function of the `BC2` 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.

Chapter 3

Installing and compiling the library

The BC2 library can be easily installed by downloading and unzipping the file bc2d.zip. After doing that, one should see a directory named bc2d with subdirectories bin, data, doc, lib, scripts, src, and tst inside. Subdirectory scripts contains an installation script, install.sh, that compiles the *BC2* 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 BC2 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 = buildcurve2d.o

LIB = libBC2D.a

buildcurve2d.o: $(INC1)/a3.hpp $(INC1)/tabulatedfunction.hpp \
                $(INC1)/bound.hpp $(INC1)/coefficient.hpp \
                $(INC1)/exceptionobject.hpp $(INC2)/glpk.h \
                $(INC1)/buildcurve2d.hpp $(INC1)/buildcurve2d.cpp
        $(CC) $(CFLAGS) $(INC1)/buildcurve2d.cpp $(INCS)

all: $(OBJ)

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

```
        $(INC1)/bound.hpp $(INC1)/coefficient.hpp \
        $(INC1)/exceptionobject.hpp $(INC2)/glpk.h \
        $(INC1)/buildcurve2d.hpp $(INC1)/buildcurve2d.cpp
        $(AR) rc $(LIB) $(OBJ)
        ranlib $(LIB)
        mv $(LIB) ../lib

clean:
        rm -f *.o *~

realclean:
        rm -f *.o *~ ../lib/libBC2D.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 -lBC2D -lglpk

OBJS = main.o

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

main.o: $(INC2)/a3.hpp $(INC2)/tabulatedfunction.hpp \
        $(INC2)/exceptionobject.hpp $(INC3)/glpk.h \
        $(INC2)/buildcurve2d.hpp $(INC1)/main.cpp
        $(CC) $(CFLAGS) $(INC1)/main.cpp $(INCS)

clean:
        rm -fr *.o *~

realclean:
        rm -fr *.o *~ ../bin/bc2d
```

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 BC2 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 libBC2D.a inside subdirectory lib, and the executable test-bc2d 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 BuildCurve2d class API.

I also left an XCode project file inside the bc2d 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 BC2 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 BuildCurve2d class API

The main class of the `BC2` library is `BuildCurve2D`. To solve the channel problem, we first instantiate an object of this class using the class constructor:

```
BuildCurve2D b(
               np ,
               nc ,
               closed ,
               &lx[ 0 ] ,
               &ly[ 0 ] ,
               &ux[ 0 ] ,
               &uy[ 0 ] ,
               tf
              ) ;
```

Variables `np` and `nc` hold the values of the parameters np 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.

The last parameter of the above constructor is a pointer, i.e., `tf`, to an object derived from a class called `Tabulated↩ Function`. It is the responsability of the library user to implement a class that inherits from `TabulatedFunction`. The derived class must implement the pure virtual methods of the class `TabulatedFunction`. Those methods correspond to the tabulated functions $\overline{a}_i^d$ and $\underline{a}_i^d$ described in the paper by Myles and Peters (see Introduction). In turn, both functions depend on the degree d the user chooses for the Bézier curves of the spline to be threaded into the channel. For every choice of d, there are $(d-2)$ pairs of functions $\overline{a}_i^d$ and $\underline{a}_i^d$, with $i = 1, \ldots, (d-2)$. To make my code as generic as possible, I decided to let the user choose d and provide functions $\overline{a}_i^d$ and $\underline{a}_i^d$ as methods of a concrete class. To make this task easier, I coded a class named `a3`, which inherits from `TabulatedFunction` and implements the pure virtual methods of `TabulatedFunction` corresponding to $d = 3$. More specifically, class `a3` implements the methods

```
virtual double alower( unsigned i , double u ) const throw( ExceptionObject ) = 0 ;
```

and

```
virtual double aupper( unsigned i , double u ) const throw( ExceptionObject ) = 0 ;
```

of class `TabulatedFunction`. The former returns the value $\underline{a}_i^d(u)$, while the latter returns the value $\overline{a}_i^d(u)$. If you read Myles and Peters' paper, you can see that functions $\overline{a}_i^d$ and $\underline{a}_i^d$ correspond to lower and upper piecewise-linear envelopes of the special polynomials a_i^d of degree d. I also decided to force the user to code a function for computing $a_i^d(u)$. Although the `BC2` library need not compute $a_i^d(u)$, I might find some use for it in future versions of the library. So, `TabulatedFunction` has another pure virtual method,

```
virtual double a( unsigned i , double u ) const throw( ExceptionObject ) = 0 ;
```

which is intended to compute $a_i^d(u)$. Finally, `TabulatedFunction` has a pure virtual method to return the value of the degree d of a_i^d:

```
virtual unsigned degree() const = 0 ;
```

The above method is called by the constructor of `BuildCurve2D` via pointer `tf`, so that the information regarding the degree of the spline can be obtained. So, if we use class `a3`, we would write

```
TabulatedFunction* tf = new a3() ;
BuildCurve2D b(
               np ,
               nc ,
               closed ,
               &lx[ 0 ] ,
               &ly[ 0 ] ,
               &ux[ 0 ] ,
               &uy[ 0 ] ,
               tf
              ) ;
```

to create an instance of the channel problem, which asks for a C^1 spline of degree 3. If you want to solve the channel problem using a C^1 spline of degree d, with $d \neq 3$, you must provide a class similar to `a3` and replace `a3()` in the above piece of code with the constructor of your class. The construction of such a class requires knowledge on how to create piecewise-linear enclosures (SLEFEs) of Bézier functions of arbitrary degree d. You can find a nice overview on SLE↩FEs in the paper presented by Jörg Peters at the SIAM Conference on Geometric Design and Computing (GD), Seattle, Washington, US, Nov. 10 - 13, 2003: `Mid-Structures Linking Curved and Linear Geometry`. A former PhD student of Dr. Peters, Xiaobin Wu, made available a code that allows us to compute $\overline{a}_i^d$ and $\underline{a}_i^d$, for an arbitrary degree d. You can find his code `here` by looking for the project *Subdividable Linear Maximum-norm Enclosure* (SubLiME). Using Wu's code, you can obtain the breakpoints of the lower and upper piecewise-linear enclosures of functions $\overline{a}_i^d$ and $\underline{a}_i^d$. Using these breakpoints, you can write the code corresponding to $\overline{a}_i^d$ and $\underline{a}_i^d$.

Once an instance of the channel problem is created, the next step is to find a solution for it. Class `BuildCurve2D` 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 `BuildCurve2D`. 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 `BuildCurve2D`:

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

The above function has three input parameters: `p`, `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 p-th Bézier curve of the spline, i.e., $b_{i,v}^{p}$. Parameter `p` holds a value in the interval $[0, np-1]$. Parameter `i` holds a value in the interval $[0, d]$. Parameter `v` holds the value 0 or 1, where 0 corresponds to the x coordinate and 1 corresponds to the y coordinate of $b_{i,v}^{p}$. The following piece of code prints out the coordinates of all control points of the spline found by the GNU GLPK solver:

```
for ( unsigned p = 0 ; p < np ; p++ ) {
  for ( unsigned i = 0 ; i <= dg ; i++ ) {
    double x = b.get_control_value( p , i , 0 ) ;
    double y = b.get_control_value( p , i , 1 ) ;
    cout << x
         << '\t'
         << y
         << endl ;
  }
}
```

The set of public methods of class `BuildCurve2D` 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 BC2 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 <string>                   // std::string
#include <cstdlib>                  // exit, EXIT_SUCCESS, EXIT_FAILURE
#include <iomanip>                  // std::setprecision
#include <cassert>                  // assert
#include <ctime>                    // time, clock, CLOCKS_PER_SEC, clock_t
#include <cstddef>                  // size_t
#include <cmath>                    // fabs

#include "exceptionobject.hpp"      // bc2::ExceptionObject
#include "tabulatedfunction.hpp"    // bc2::TabulatedFunction
#include "a3.hpp"                   // bc2::a3
#include "buildcurve2d.hpp"         // bc2::BuildCurve2D
```

File buildcurve2d.hpp contains the definition of class BuildCurve2D; file a3.hpp contains the definition of class a3; file tabulatedfunction.hpp contains the definition of (abstract) class TabulatedFunction; and file exceptionobject.hpp contains the definition of a class, ExceptionObject, that I use to create 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:

```
int main( int argc , char* argv[] ) {
  if ( ( argc != 3 ) && ( argc != 4 ) ) {
    cerr << "Usage: "
         << endl
         << "\t\t test-bc2d 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 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 ] ) ;

  unsigned np ;
  unsigned nc ;
  bool closed ;
```

```
  unsigned dg ;
  double* lx ;
  double* ly ;
  double* ux ;
  double* uy ;

  try {
    read_input( fn1 , np , nc , closed , dg , 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: `np`, `nc`, `closed`, `dg`, `lx` , `ly`, `ux`, and `uy`. I already talked about all these parameters, except for `dg`. Actually, `dg` is the degree of the resulting spline. At this point, this value is included just to make sure that there is a consistency between the degree informed in the input file and the degree to be informed by the class that inherits from `TabulatedFunction`. In other words, the degree must be the same. Observe that the memory occupied by the arrays `lx` , `ly`, `ux`, and `uy` is allocated inside function `read_input()`.

The next lines1 invoke the constructor of `BuildCurved2D` to create the given instance of the channel problem:

```
assert( dg == 3 ) ;
TabulatedFunction* tf = new a3() ;
BuildCurve2D* builder = 0 ;
try {
  builder = new BuildCurve2D(
                             np ,
                             nc ,
                             closed ,
                             &lx[ 0 ] ,
                             &ly[ 0 ] ,
                             &ux[ 0 ] ,
                             &uy[ 0 ] ,
                             tf
                            ) ;
}
catch ( const ExceptionObject& xpt ) {
  treat_exception( xpt ) ;
  exit( EXIT_FAILURE ) ;
}
```

Observe that there is an assertion just before the constructor of `class` a3 is invoked. This assertion ensures that the degree `dg` of the spline, given by the description of the instance of the channel problem, is equal to 3. This is because we are using functions $\underline{a}_i^3$ and $\overline{a}_i^3$ to solve the problem. In general, if you want to solve the problem using a spline of degree d, for some fixed $d \geq 2$, you must replace 3 with d in the assertion, and then change the next line to something like

```
TabulatedFunction* tf = new ad() ;
```

where `ad` is the class derived from `TabulatedFunction` you must implement, i.e., the class representing functions $\underline{a}_i^d$ and $\overline{a}_i^d$. This is all you need to re-use this program to solve the channel problem with a spline of degree d. Once the

instance of the channel problem has been created, which is equivalent to saying that an object of `class` BuildCurve2D has been instantiated, we can ask the class to solve the problem, which is done by invoking function `build()` (see section The BuildCurve2d 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 `BuildCurve2D` 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 `BuildCurve2D` as explained in section The BuildCurve2d class API. Below is the body of `write_solution()`:

```
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 number of curve pieces and the degree of the spline.
  //

  unsigned np = b.get_number_of_curve_pieces() ;
  unsigned dg = b.get_spline_degree() ;

  ou << np
     << '\t'
     << dg
     << endl ;

  for ( unsigned p = 0 ; p < np ; p++ ) {
    for ( unsigned i = 0 ; i <= dg ; i++ ) {
      double x ;
      double y ;
      try {
        x = b.get_control_value( p , i , 0 ) ;
        y = b.get_control_value( p , i , 1 ) ;
      }
      catch ( const ExceptionObject& xpt ) {
        treat_exception( xpt ) ;
        ou.close() ;
        exit( EXIT_FAILURE ) ;
      }
      ou << x
         << '\t'
         << y
         << endl ;
    }
  }

  //
  // Close file
  //

  ou.close() ;
}
```

Observe that before asking for each coordinate of every control point, function `write_solution()` obtains the number np of Bézier curves making up the spline and the degree d of each curve, which is 3 in our program. Using these two parameters, we can obtain the two coordinates of the i-th control point of the p-th curve: b_i^p, which is done by the two lines below:

```
x = b.get_control_value( p , i , 0 ) ;
y = b.get_control_value( p , i , 1 ) ;
```

Chapter 6

Examples and file formats

If a spline of degree 3 is enough for your needs, then you can readily use the program I explained in section Using the library API to solve your instances of the channel problem. Luckily, the C^1-continuity constraint won't be too restrictive for the channels you have. So, let us assume that the program I made available is good enough for you. To solve the channel problem using my program, you must give the program 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

np nc closed nn dg

in this order, where *np* is the number of Bézier curves that make up the resulting spline, *nc* is the number of c-segments of the channel that delimit each Bézier curve, *closed* is a flag to indicate whether the channel is open or closed, *nn* is the number of vertices of either envelope of the channel, and *dg* is the degree of each Bézier curve. See section Introduction for a more detailed description of the above parameters. After the first line, there are *nn* lines, each of which 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:

```
3 1 0 4 3
 2.00000000     -2.00000000
 2.00000000      2.00000000
-2.00000000      2.00000000
-2.00000000     -2.00000000
 4.00000000     -4.00000000
 4.00000000      4.00000000
```

```
-4.00000000      4.00000000
-4.00000000     -4.00000000
```

The above file describes the *open* channel

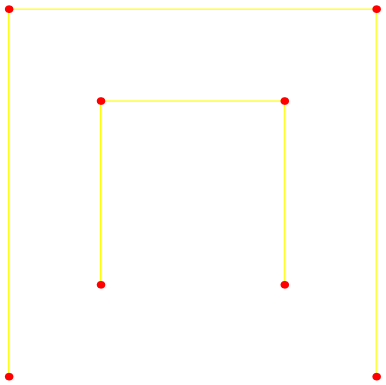


Figure 6.1: Example of a channel

and asks for a spline of degree $dg = 3$ consisting of $np = 3$ Bézier curves, each of which is delimited by $nc = 1$ 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 $nn + 1 = np + nc$, as required (see section Introduction). Function `read_input()` (see section Using the library API) reads in the input .chn file and obtains the values of *np*, *nc*, *dg*, *nn*, *lx*, *ly*, *ux*, and *uy*. Once the problem is solved, my program generates an output file with extension .spl. This file contains the Cartesian coordinates of the control points of the `np` Bézier curves defining the spline. The first line of a .spl file specifies the total number of control points and the degree of the spline, i.e.,

ncp dg

The value of *ncp* must be equal to the product of *np* and *dg+1*. 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:

$b^0_{0,x}, b^0_{0,y}$
$b^0_{1,x}, b^0_{1,y}$
...
$b^0_{d,x}, b^0_{d,y}$
$b^1_{0,x}, b^1_{0,y}$
$b^1_{1,x}, b^1_{1,y}$
...
$b^1_{d,x}, b^1_{d,y}$
$\vdots$
$b^{np-1}_{0,x}, b^{np-1}_{0,y}$
$b^{np-1}_{1,x}, b^{np-1}_{1,y}$
...
$b^{np-1}_{d,x}, b^{np-1}_{d,y}$

where $b^p_{i,x}$ and $b^p_{i,x}$ 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:

```
3               3
3.000000       -3.000000
3.757762       -0.662038
```

```
 4.515524        1.675924
 3.000000        3.000000
 3.000000        3.000000
 1.484476        4.324076
-1.484476        4.324076
-3.000000        3.000000
-3.000000        3.000000
-4.515524        1.675924
-3.757762       -0.662038
-3.000000       -3.000000
```

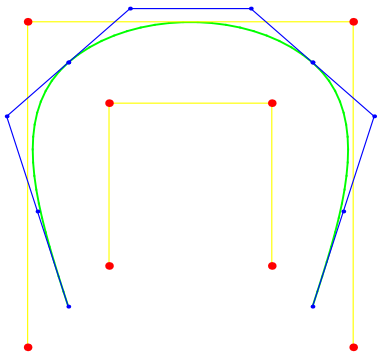


Figure 6.2: Example of a solution for the channel problem

You can find more examples of .chn files in the subdirectory `data/channels`. I wrote a script, `run.sh`, that executes my program 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. But, you can easily write a script in `Matlab` (or an equivalent tool) to do that.

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 `test-bc2d` should be located:

```
test-bc2d < 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

```
test-bc2d < your input CHN file > < your output SPL file > < your output LP file >
```

When the execution ends, the thord 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 `BC2` library code.

Chapter 9

Module Index

9.1 Modules

Here is a list of all modules:

Namespace bc2. 37

Chapter 10

Namespace Index

10.1 Namespace List

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

bc2

The namespace bc2 contains the definition and implementation of a set of classes for computing a C1 spline cubic curve that passes through a given planar channel delimited by two polygonal chains 39

Chapter 11

Hierarchical Index

11.1 Class Hierarchy

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

bc2::Bound 49
bc2::BuildCurve2D 52
bc2::Coefficient 88
exception
  bc2::ExceptionObject 91
bc2::TabulatedFunction 98
  bc2::a3 41

Chapter 12

Class Index

12.1 Class List

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

bc2::a3
: This class represents two-sided, piecewise linear enclosures for two polynomial functions of degree 3 in Bézier form . . . 41

bc2::Bound
: This class represents the type of a constraint (i.e., equality or inequality) and the value of its right-hand side: a real . . . 49

bc2::BuildCurve2D
: This class provides methods for threading a C1 spline curve of degree d through a planar channel delimited by a pair of polygonal chain . . . 52

bc2::Coefficient
: This class represents a nonzero coefficient of a variable of a linear constraint (inequality or equality) of a linear program . . . 88

bc2::ExceptionObject
: This class extends class *exception* of STL and provides us with a customized way of handling exceptions and showing error messages . . . 91

bc2::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 . . . 98

Chapter 13

File Index

13.1 File List

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

a3.hpp
Definition of a class for representing piecewise linear enclosures of certain cubic polynomial functions in Bézier form . . . 103

bound.hpp
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 . . . 105

buildcurve2d.cpp
Implementation of a class for threading a C1 spline curve of degree d through a planar channel defined by a pair of polygonal chains . . . 106

buildcurve2d.hpp
Definition of a class for threading a C1 spline curve of degree d through a planar channel defined by a pair of polygonal chains . . . 107

coefficient.hpp
Definition of a class for representing a nonzero coefficient of a variable of a linear constraint (inequality or equality) of an LP . . . 109

exceptionobject.hpp
Definition of a class for handling exceptions . . . 110

main.cpp
A simple program for testing the bc2d library . . . 112

tabulatedfunction.hpp
Definition of an abstract class for representing piecewise linear enclosures of certain polynomial functions of arbitrary degree . . . 122

Chapter 14

Module Documentation

14.1 Namespace bc2.

Namespaces

- bc2

  The namespace bc2 contains the definition and implementation of a set of classes for computing a C1 spline cubic curve that passes through a given planar channel delimited by two polygonal chains.

Classes

- class bc2::a3

  This class represents two-sided, piecewise linear enclosures for two polynomial functions of degree 3 in Bézier form.
- class bc2::Bound

  This class represents the type of a constraint (i.e., equality or inequality) and the value of its right-hand side: a real.
- class bc2::BuildCurve2D

  This class provides methods for threading a C1 spline curve of degree d through a planar channel delimited by a pair of polygonal chain.
- class bc2::Coefficient

  This class represents a nonzero coefficient of a variable of a linear constraint (inequality or equality) of a linear program.
- class bc2::ExceptionObject

  This class extends class exception of STL and provides us with a customized way of handling exceptions and showing error messages.
- class bc2::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 bc2 Namespace Reference

The namespace bc2 contains the definition and implementation of a set of classes for computing a C1 spline cubic curve that passes through 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.
- class BuildCurve2D

  This class provides methods for threading a C1 spline curve of degree d through a planar channel delimited by a pair of polygonal chain.
- class Coefficient

  This class represents a nonzero coefficient of a variable of a linear constraint (inequality or equality) of a linear program.
- 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 bc2 contains the definition and implementation of a set of classes for computing a C1 spline cubic curve that passes through a given planar channel delimited by two polygonal chains.

The namespace bc2 contains the definition and implementation of a set of classes for computing a C1 spline curve of degree d passing through a given planar channel delimited by two polygonal chains.

Chapter 16

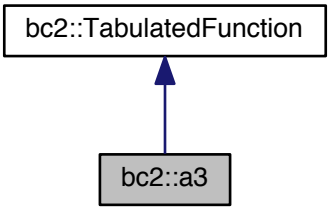
Class Documentation

16.1 bc2::a3 Class Reference

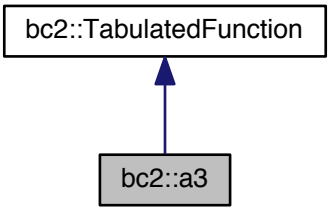
This class represents two-sided, piecewise linear enclosures for two polynomial functions of degree 3 in Bézier form.

`#include <a3.hpp>`

Inheritance diagram for bc2::a3:



Collaboration diagram for bc2::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 (unsigned i, 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 (unsigned i, 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 (unsigned i, 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 (double u) const

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

- double a1upper (double u) const

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

- double a1 (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 (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 in 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 71 of file a3.hpp.

16.1.2 Member Function Documentation

16.1.2.1 double bc2::a3::a (unsigned *i*, 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	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.

Implements bc2::TabulatedFunction.

Definition at line 221 of file a3.hpp.

References a1().

```
226     {
227       if ( ( i != 1 ) && ( i != 2 ) ) {
228         std::stringstream ss( std::stringstream::in | std::stringstream::out ) ;
229         ss << "Index of the polynomial function is out of range" ;
230         throw ExceptionObject( __FILE__ , __LINE__ , ss.str().c_str() ) ;
231       }
232
233       if ( ( u < 0 ) || ( u > 1 ) ) {
234         std::stringstream ss( std::stringstream::in | std::stringstream::out ) ;
235         ss << "Parameter value must belong to the interval [0,1]" ;
```

```
236         throw ExceptionObject( __FILE__ , __LINE__ , ss.str().c_str() ) ;
237       }
238
239       return ( i == 1 ) ? a1( u ) : a1( 1 - u ) ;
240     }
```

16.1.2.2 double bc2::a3::a1 (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().

```
330     {
331 #ifdef DEBUGMODE
332       assert( u >= 0 ) ;
333       assert( u <= 1 ) ;
334 #endif
335
336       return -u * ( 2 - u * ( 3 - u ) ) ;
337     }
```

16.1.2.3 double bc2::a3::a1lower (double *u*) const [inline], [protected]

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

Definition at line 278 of file a3.hpp.

References h().

Referenced by alower().

```
279     {
280       const double onethird = double( 1 ) / 3 ;
281
282       double res = _l0 * h( u )
283                        + _l1 * h( u -     onethird )
284                        + _l2 * h( u - 2 * onethird )
285                        + _l3 * h( u - 1 ) ;
286
287       return res ;
288     }
```

16.1.2.4 double bc2::a3::a1upper (double *u*) const [inline], [protected]

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

Definition at line 304 of file a3.hpp.

References h().

Referenced by aupper().

```
305     {
306       const double onethird = double( 1 ) / 3 ;
307
308       double res = -10 * h( u - onethird ) - 8 * h( u - 2 * onethird ) ;
309
310       res *= ( double(1) / 27 ) ;
311
312       return res ;
313     }
```

16.1.2.5 double bc2::a3::alower (unsigned *i*, 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 bc2::TabulatedFunction.

Definition at line 145 of file a3.hpp.

References a1lower().

```
150     {
151       if ( ( i != 1 ) && ( i != 2 ) ) {
152         std::stringstream ss( std::stringstream::in | std::stringstream::out ) ;
153         ss << "Index of the polynomial function is out of range" ;
154         throw ExceptionObject( __FILE__ , __LINE__ , ss.str().c_str() ) ;
155       }
156
157       if ( ( u < 0 ) || ( u > 1 ) ) {
158         std::stringstream ss( std::stringstream::in | std::stringstream::out ) ;
159         ss << "Parameter value must belong to the interval [0,1]" ;
160         throw ExceptionObject( __FILE__ , __LINE__ , ss.str().c_str() ) ;
161       }
162
163       return ( i == 1 ) ? allower( u ) : allower( 1 - u ) ;
164     }
```

16.1.2.6 double bc2::a3::aupper (unsigned *i*, 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 bc2::TabulatedFunction.

Definition at line 183 of file a3.hpp.

References a1upper().

```
188     {
189       if ( ( i != 1 ) && ( i != 2 ) ) {
190         std::stringstream ss( std::stringstream::in | std::stringstream::out ) ;
191         ss << "Index of the polynomial function is out of range" ;
192         throw ExceptionObject( __FILE__ , __LINE__ , ss.str().c_str() ) ;
193       }
194
195       if ( ( u < 0 ) || ( u > 1 ) ) {
196         std::stringstream ss( std::stringstream::in | std::stringstream::out ) ;
197         ss << "Parameter value must belong to the interval [0,1]" ;
198         throw ExceptionObject( __FILE__ , __LINE__ , ss.str().c_str() ) ;
199       }
200
201       return ( i == 1 ) ? a1upper( u ) : a1upper( 1 - u ) ;
202     }
```

16.1.2.7 unsigned bc2::a3::degree ( ) const [inline], [virtual]

Returns the degree of tabulated functions.

Returns

The degree of the tabulated functions.

Implements bc2::TabulatedFunction.

Definition at line 251 of file a3.hpp.

```
252     {
253       return 3 ;
254     }
```

16.1.2.8 double bc2::a3::h (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().

```
353     {
354       const double onethird = 1.0 / 3.0 ;
355
356       if ( u <= -onethird ) {
357         return 0 ;
358       }
359       else if ( u <= 0 ) {
360         return 3 * u + 1 ;
361       }
362       else if ( u <= onethird ) {
363         return 1 - 3 * u ;
364       }
365
366       return 0 ;
367     }
```

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

- a3.hpp

16.2 bc2::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.

`#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 (CONSTRAINTYPE type, double value, unsigned 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.
- unsigned get_row () const

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

Protected Attributes

- CONSTRAINTYPE _ctype

  The type of the constraint associated with this bound.

- double _value

  The bound value.

- unsigned _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.

Definition at line 56 of file bound.hpp.

16.2.2 Constructor & Destructor Documentation

16.2.2.1 bc2::Bound::Bound (CONSTRAINTYPE *type*, double *value*, unsigned *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 122 of file bound.hpp.

```
123     :
124       _ctype( type ) ,
125       _value( value ) ,
126       _row( row )
127     {
128     }
```

16.2.2.2 bc2::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 140 of file bound.hpp.

```
141     :
142       _ctype( b._ctype ) ,
143       _value( b._value ) ,
144       _row( b._row )
145     {
146     }
```

16.2.3 Member Function Documentation

16.2.3.1 unsigned bc2::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 constraint coefficient matrix of a linear program.

Returns

The identifier of the constraint associated with this bound.

Definition at line 201 of file bound.hpp.

References _row.

```
202     {
203       return _row ;
204     }
```

16.2.3.2 CONSTRAINTYPE bc2::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 169 of file bound.hpp.

References _ctype.

```
170     {
171       return _ctype ;
172     }
```

16.2.3.3 double bc2::Bound::get_value ( ) const [inline]

Returns the value of this bound.

Returns

The value of this bound.

Definition at line 183 of file bound.hpp.

References _value.

```
184     {
185       return _value ;
186     }
```

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

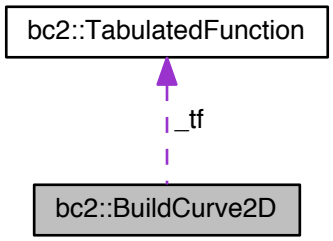
- bound.hpp

16.3 bc2::BuildCurve2D Class Reference

This class provides methods for threading a C1 spline curve of degree d through a planar channel delimited by a pair of polygonal chain.

`#include <buildcurve2d.hpp>`

Collaboration diagram for bc2::BuildCurve2D:



Public Types

- typedef std::vector< double >::size_type sizetype

  Defines a type for the size of the array of the Cartesian coordinates of the lower and upper envelopes of the channel.
- typedef std::vector< std::vector< Coefficient > >::size_type constsizetype

  Defines a type for the size of the array of constraints.
- typedef std::vector< Coefficient >::size_type coeffsizetype

  Defines a type for the size of the array of coefficients.

Public Member Functions

- BuildCurve2D (unsigned np, unsigned nc, bool closed, double *lx, double *ly, double *ux, double *uy, TabulatedFunction *tf) throw (ExceptionObject)

  Creates an instance of this class.
- BuildCurve2D (const BuildCurve2D &b)

  Clones an instance of this class.
- ~BuildCurve2D ()

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

  Solves the channel problem by solving a linear program.
- unsigned get_number_of_curve_pieces () const

  Returns the number of curves pieces of the spline curve.
- unsigned get_spline_degree () const

  Returns the degree of the spline curve.
- unsigned 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 (unsigned p, unsigned i, unsigned v) const throw (ExceptionObject)

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

- unsigned get_number_of_coefficients_in_the_ith_constraint (unsigned i) const throw (ExceptionObject)

  Returns the number of coefficients of the i-th constraint of the instance of the linear program corresponding to the channel problem solved by this class.

- unsigned get_coefficient_identifier (unsigned i, unsigned j) const throw (ExceptionObject)

  Returns the number of the column that corresponds to the j-th coefficient of the i-th constraint in the constraint matrix of the instance of the linear program corresponding to the channel problem.

- double get_coefficient_value (unsigned i, unsigned j) const throw (ExceptionObject)

  Returns the (i , j) entry of the matrix of constraints of the instance of the linear program corresponding to the channel problem.

- double get_bound_of_ith_constraint (unsigned i) const throw (ExceptionObject)

  Returns the real value that bounds the i-th constraint.

- bool is_equality (unsigned 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 (unsigned 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 (unsigned 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 (unsigned p, unsigned i, unsigned v) const throw (ExceptionObject)

  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 piece of the spline curve threaded into the channel.

- double get_upper_bound_on_second_difference_value (unsigned p, unsigned i, unsigned 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 piece of the 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

- double lf (double u, double b0, double bd) const

  Computes the value of the affine function ℓ at a given point of the interval $[0, 1]$ of the real line.

- double h (double u) const

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

- void compute_normal_to_lower_envelope (sizetype 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 (sizetype s, double &nx, double &ny) const

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

- unsigned compute_control_value_column_index (unsigned p, unsigned i, unsigned 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 piece of the spline curve to be threaded into the channel.

- unsigned compute_second_difference_column_index (unsigned p, unsigned i, unsigned l, unsigned 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 piece of the spline curve to be threaded into the channel.
- void compute_min_max_constraints (unsigned &eqline)

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

  Computes the equations defining the correspondence constraints.
- void compute_c0continuity_constraints (unsigned &eqline)

  Computes the equations defining the C0-continuity constraints.
- void compute_c1continuity_constraints (unsigned &eqline)

  Computes the equations defining the C1-continuity constraints.
- void compute_sleeve_corners_in_channel_constraints (unsigned &eqline)

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

  Computes the equations defining the constraints that ensure that the corners of the channel are located outside the sleeve.
- int solve_lp (sizetype rows, sizetype 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

- unsigned _np

  The number of polynomial pieces of the spline curve.
- unsigned _nc

  The number of c-segments in each c-piece of the channel.
- unsigned _dg

  The degree of the curve to be threaded into 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 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.

16.3.1 Detailed Description

This class provides methods for threading a C1 spline curve of degree d through a planar channel delimited by a pair of polygonal chain.

Attention

This class is based on a particular case (i.e., the planar case) 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 79 of file buildcurve2d.hpp.

16.3.2 Constructor & Destructor Documentation

16.3.2.1 bc2::BuildCurve2D::BuildCurve2D (unsigned *np*, unsigned *nc*, bool *closed*, double * *lx*, double * *ly*, double * *ux*, double * *uy*, TabulatedFunction * *tf*) throw ExceptionObject)

Creates an instance of this class.

Parameters

np	The number of polynomial pieces of the spline curve.
nc	The number of c-segments in each c-piece 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 polygonal chain of the channel.
ly	A pointer to an array with the y-coordinates of the lower polygonal chain of the channel.
ux	A pointer to an array with the x-coordinates of the upper polygonal chain of the channel.
uy	A pointer to an array with the y-coordinates of the upper polygonal chain of the channel.
tf	A pointer to the lower and upper a functions.

Definition at line 81 of file buildcurve2d.cpp.

```
92   :
93     _np( np ) ,
94     _nc( nc )
95   {
96     if ( tf == 0 ) {
97       std::stringstream ss( std::stringstream::in | std::stringstream::out ) ;
98       ss << "Pointer to function A() is null" ;
99       throw ExceptionObject( __FILE__ , __LINE__ , ss.str().c_str() ) ;
100     }
```

```
101
102     if ( tf->degree() < 2 ) {
103       std::stringstream ss( std::stringstream::in | std::stringstream::out ) ;
104       ss << "The degree of the spline must be at least 2" ;
105       throw ExceptionObject( __FILE__ , __LINE__ , ss.str().c_str() ) ;
106     }
107
108     _closed = closed ;
109     _dg = tf->degree() ;
110
111     unsigned nn = ( _closed ) ? ( np * nc ) : ( ( np * nc ) + 1 ) ;
112
113     _lxcoords.resize( nn ) ;
114     _lycoords.resize( nn ) ;
115     _uxcoords.resize( nn ) ;
116     _uycoords.resize( nn ) ;
117
118     for ( unsigned i = 0 ; i < nn ; i++ ) {
119       _lxcoords[ i ] = lx[ i ] ;
120       _lycoords[ i ] = ly[ i ] ;
121       _uxcoords[ i ] = ux[ i ] ;
122       _uycoords[ i ] = uy[ i ] ;
123     }
124
125     _tf = tf ;
126
127     _ofvalue = DBL_MAX ;
128
129     return ;
130   }
```

16.3.2.2 bc2::BuildCurve2D::BuildCurve2D (const BuildCurve2D & *b*)

Clones an instance of this class.

Parameters

b	A reference to another instance of this class.

Definition at line 141 of file buildcurve2d.cpp.

```
142   :
143     _np( b._np ) ,
144     _nc( b._nc ) ,
145     _dg( b._dg ) ,
146     _closed( b._closed ) ,
147     _lxcoords( b._lxcoords ) ,
148     _lycoords( b._lycoords ) ,
149     _uxcoords( b._uxcoords ) ,
150     _uycoords( b._uycoords ) ,
151     _tf( b._tf ) ,
152     _coefficients( b._coefficients ) ,
153     _bounds( b._bounds ) ,
154     _ctrlpts( b._ctrlpts ) ,
155     _secdiff( b._secdiff ) ,
156     _ofvalue( b._ofvalue )
157   {
158   }
```

16.3.3 Member Function Documentation

16.3.3.1 bool bc2::BuildCurve2D::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 187 of file buildcurve2d.cpp.

References _bounds, _closed, _coefficients, _dg, _nc, _np, compute_c0continuity_constraints(), compute_c1continuity_constraints(), compute_channel_corners_outside_sleeve_constraints(), compute_correspondence_constraints(), compute_min_max_constraints(), compute_sleeve_corners_in_channel_constraints(), and solve_lp().

Referenced by main().

```
188   {
189     sizetype rows = ( sizetype )
190                 (
191                     ( _np * 6 * ( _dg - 1 ) )                                    // min-max
192                   + ( ( 2 * _np ) + ( ( !_closed ) ? 2 : 0 ) )                   //
      correspondence
193                   + ( 2 * ( _np - 1 ) + ( ( _closed ) ? 2 : 0 ) )                // c0
194                   + ( 2 * ( _np - 1 ) + ( ( _closed ) ? 2 : 0 ) )                // c1
195                   + ( 4 * ( _np ) * ( _dg + 1 ) )                                // sleeve corners
196                   + ( _np * ( 8 * ( _nc - 1 ) + 16 ) - ( ( _closed ) ? 0 : 16 ) ) // channel
      corners
197                 ) ;
198
199     sizetype cols = sizetype( _np * ( 6 * _dg - 2 ) ) ;
200
201     std::vector< std::vector< Coefficient > >::size_type crows =
202       std::vector< std::vector< Coefficient > >::size_type( rows ) ;
203     std::vector< Bound >::size_type brows =
204       std::vector< Bound >::size_type( rows ) ;
205
206     _coefficients.resize( crows ) ;
207     _bounds.resize( brows ) ;
208
209     unsigned eqline = 0 ;
210
211     compute_min_max_constraints( eqline ) ;
212     compute_correspondence_constraints( eqline ) ;
213     compute_c0continuity_constraints( eqline ) ;
214     compute_c1continuity_constraints( eqline ) ;
215     compute_sleeve_corners_in_channel_constraints( eqline ) ;
216     compute_channel_corners_outside_sleeve_constraints(
      eqline ) ;
217
218     error = solve_lp( rows , cols ) ;
219
220     return ( error == 0 ) ;
221   }
```

16.3.3.2 void bc2::BuildCurve2D::compute_c0continuity_constraints (unsigned & *eqline*) [private]

Computes the equations defining the C0-continuity constraints.

Parameters

eqline	A reference to the counter of equations.

Definition at line 652 of file buildcurve2d.cpp.

References _bounds, _closed, _coefficients, _dg, _np, and compute_control_value_column_index().

Referenced by build().

```
653   {
654     /*
655      * For each curve p, with p less  than the number of pieces of the
656      * spline curve, make the last control point of the p-th piece the
657      * same as  the first control point  of the (p+1)-th piece  of the
658      * curve.
659      */
660     for ( unsigned p = 0 ; p < ( _np - 1 ) ; p++ ) {
661       for ( unsigned v = 0 ; v < 2 ; v++ ) {
662         /*
663          * Get the  column index  of the v-th  coordinate of  the last
664          * control point of  the p-th piece and the index  of the v-th
665          * coordinate  of  the first  control  point  of the  (p+1)-th
666          * piece.
667          */
668         unsigned jd = compute_control_value_column_index(
669                                                          p ,
670                                                          _dg ,
671                                                          v
672                                                         ) ;
673
674         unsigned j0 = compute_control_value_column_index(
675                                                          p + 1 ,
676                                                          0 ,
677                                                          v
678                                                         ) ;
679
680         _coefficients[ eqline ].push_back( Coefficient( eqline , jd ,  1 ) ) ;
681         _coefficients[ eqline ].push_back( Coefficient( eqline , j0 , -1 ) ) ;
682         _bounds[ eqline ] = Bound( Bound::EQT , 0 , eqline ) ;
683
684         ++eqline ;  // increment the equation counter ;
685       }
686     }
687
688     if ( _closed ) {
689       for ( unsigned v = 0 ; v < 2 ; v++ ) {
690         /*
691          * Get the  column index  of the v-th  coordinate of  the last
692          * control point of  the p-th piece and the index  of the v-th
693          * coordinate of the first control point of the first piece.
694          */
695         unsigned jd = compute_control_value_column_index(
696                                                          _np - 1 ,
697                                                          _dg ,
698                                                          v
699                                                         ) ;
700
701         unsigned j0 = compute_control_value_column_index(
702                                                          0 ,
703                                                          0 ,
704                                                          v
705                                                         ) ;
706
707         _coefficients[ eqline ].push_back( Coefficient( eqline , jd ,  1 ) ) ;
708         _coefficients[ eqline ].push_back( Coefficient( eqline , j0 , -1 ) ) ;
709         _bounds[ eqline ] = Bound( Bound::EQT , 0 , eqline ) ;
710
711         ++eqline ;  // increment the equation counter ;
712       }
713     }
714
715     return ;
716   }
```

16.3.3.3 void bc2::BuildCurve2D::compute_c1continuity_constraints (unsigned & *eqline*) [private]

Computes the equations defining the C1-continuity constraints.

Parameters

eqline	A reference to the counter of equations.

Definition at line 729 of file buildcurve2d.cpp.

References _bounds, _closed, _coefficients, _dg, _np, and compute_control_value_column_index().

Referenced by build().

```
730   {
731     /*
732      * For each curve p, with p less  than the number of pieces of the
733      * spline curve,  make the  last control point  of the  p-th piece
734      * equal to the midpoint of the line segment defined by the before
735      * the last  control point of p-th  piece and the second  point of
736      * the (p+1)-th piece of the curve.
737      */
738     for ( unsigned p = 0 ; p < ( _np - 1 ) ; p++ ) {
739       for ( unsigned v = 0 ; v < 2 ; v++ ) {
740         /*
741          * Get the column indices of  the v-th coordinates of the last
742          * control  point  of the  p-th  piece,  the before  the  last
743          * control point  of the  p-th piece,  and the  second control
744          * point of the (p+1)-th piece of the curve.
745          */
746         unsigned jd = compute_control_value_column_index(
747                                                          p ,
748                                                          _dg ,
749                                                          v
750                                                          ) ;
751
752         unsigned jp = compute_control_value_column_index(
753                                                          p ,
754                                                          _dg - 1 ,
755                                                          v
756                                                          ) ;
757
758         unsigned j2 = compute_control_value_column_index(
759                                                          p + 1 ,
760                                                          1 ,
761                                                          v
762                                                          ) ;
763
764         _coefficients[ eqline ].push_back( Coefficient( eqline , jd ,  2 ) ) ;
765         _coefficients[ eqline ].push_back( Coefficient( eqline , jp , -1 ) ) ;
766         _coefficients[ eqline ].push_back( Coefficient( eqline , j2 , -1 ) ) ;
767         _bounds[ eqline ] = Bound( Bound::EQT , 0 , eqline ) ;
768
769         ++eqline ;  // increment the equation counter ;
770       }
771     }
772
773     if ( _closed ) {
774       for ( unsigned v = 0 ; v < 2 ; v++ ) {
775         /*
776          * Get the column indices of  the v-th coordinates of the last
777          * control  point  of the  last  piece,  the before  the  last
778          * control point  of the  last piece,  and the  second control
779          * point of the first piece of the spline.
780          */
781         unsigned jd = compute_control_value_column_index(
782                                                          _np - 1 ,
783                                                          _dg ,
784                                                          v
785                                                         ) ;
786
787         unsigned jp = compute_control_value_column_index(
788                                                          _np - 1 ,
789                                                          _dg - 1 ,
790                                                          v
791                                                         ) ;
792
793         unsigned j2 = compute_control_value_column_index(
794                                                          0 ,
795                                                          1 ,
796                                                          v
797                                                         ) ;
798
799         _coefficients[ eqline ].push_back( Coefficient( eqline , jd ,  2 ) ) ;
800         _coefficients[ eqline ].push_back( Coefficient( eqline , jp , -1 ) ) ;
```

```
801         _coefficients[ eqline ].push_back( Coefficient( eqline , j2 , -1 ) ) ;
802         _bounds[ eqline ] = Bound( Bound::EQT , 0 , eqline ) ;
803
804         ++eqline ;  // increment the equation counter ;
805       }
806     }
807
808     return ;
809   }
```

16.3.3.4 void bc2::BuildCurve2D::compute_channel_corners_outside_sleeve_constraints (unsigned & *eqline*) [private]

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

Parameters

eqline	A reference to the counter of equations.

Definition at line 1070 of file buildcurve2d.cpp.

References _bounds, _closed, _coefficients, _dg, _lxcoords, _lycoords, _nc, _np, _tf, _uxcoords, _uycoords, bc2↩ ::TabulatedFunction::alower(), bc2::TabulatedFunction::aupper(), compute_control_value_column_index(), compute↩ _normal_to_lower_envelope(), compute_normal_to_upper_envelope(), compute_second_difference_column_index(), and treat_exception.

Referenced by build().

```
1071   {
1072     /*
1073      * Compute outward normals to the line segments of the c-segments.
1074      */
1075     std::vector< std::vector< double > > nl( _np * _nc , std::vector< double >( 2 , 0 ) ) ;
1076     std::vector< std::vector< double > > nu( _np * _nc , std::vector< double >( 2 , 0 ) ) ;
1077     if ( ( _np > 1 ) || ( _nc > 1 ) ) {
1078       for ( sizetype c = 0 ; c < sizetype( _np * _nc ) ; c++ ) {
1079         compute_normal_to_lower_envelope( c , nl[ c ][ 0 ] , nl[ c ][ 1 ] )
     ;
1080         compute_normal_to_upper_envelope( c , nu[ c ][ 0 ] , nu[ c ][ 1 ] )
     ;
1081       }
1082     }
1083
1084     /*
1085      * For each  curve p, for  each channel  corner that belongs  to a
1086      * c-segment delimiting the p-th curve piece, compute a constraint
1087      * that ensures that  the channel corner is outside  the sleeve of
1088      * the p-th piece.
1089      */
1090     for ( unsigned p = 0 ; p < _np ; p++ ) {
1091       for ( unsigned c = 0 ; c <= _nc ; c++ ) {
1092         /*
1093          * Check whether  the current constraint is  equivalent to the
1094          * sleeve inside channel constraint. If so, there is no reason
1095          * to create another equation, which would otherwise slow down
1096          * computations.
1097          */
1098         if ( ( c == 0 ) && ( p == 0 ) && ( !_closed ) ) {
1099           continue ;
1100         }
1101
1102         if ( ( c == _nc ) && ( p == ( _np - 1 ) ) && ( !_closed ) ) {
1103           continue ;
1104         }
1105
1106         /*
1107          * Find the  index \e s  of the  nearest sleeve corner  at the
1108          * c-section containing the  channel corner or to  the left of
1109          * it.
1110          */
1111         sizetype s = sizetype( ceil( c * ( double( _dg ) / _nc ) ) ) ;
1112
```

```
1113         /*
1114          * Find the  index \e t  of the  nearest sleeve corner  at the
1115          * c-section containing the channel corner  or to the right of
1116          * it.
1117          */
1118         sizetype t = ( s == 0 ) ? s : s - 1 ;
1119 
1120         /*
1121          * Compute the  parameter value  that gives the  e-piece point
1122          * that matches the channel corner in a parametrization of the
1123          * c-piece.
1124          */
1125         double u = s - c * ( double( _dg ) / _nc ) ;
1126 
1127         /*
1128          * Copmpute the parameter values of a point in the e-piece.
1129          */
1130         double sp = double( s ) / _dg ;
1131         double tp = double( t ) / _dg ;
1132 
1133         /*
1134          * Select  the  normals to  the  line  segments of  the  lower
1135          * envelope incident  to the channel  corner \e c of  the p-th
1136          * piece.
1137          */
1138         sizetype cl = ( ( p * _nc ) + c ) % ( _np * _nc ) ;
1139         sizetype cr = ( cl == 0 ) ? ( _np * _nc ) - 1 : cl - 1 ;
1140 
1141         /*
1142          * Compute  the column  indices of  the linear  program matrix
1143          * corresponding  to the  second differences  and the  control
1144          * points involved in the constraints associated with the c-th
1145          * corner.
1146          */
1147         std::vector< std::vector< unsigned > > cp( 2 , std::vector< unsigned >( 2 , 0 ) ) ;
1148 
1149         cp[ 0 ][ 0 ] = compute_control_value_column_index(
1150                                                           p ,
1151                                                           0 ,
1152                                                           0
1153                                                          ) ;
1154         cp[ 0 ][ 1 ] = compute_control_value_column_index(
1155                                                           p ,
1156                                                           0 ,
1157                                                           1
1158                                                          ) ;
1159         cp[ 1 ][ 0 ] = compute_control_value_column_index(
1160                                                           p ,
1161                                                           _dg ,
1162                                                           0
1163                                                           ) ;
1164         cp[ 1 ][ 1 ] = compute_control_value_column_index(
1165                                                           p ,
1166                                                           _dg ,
1167                                                           1
1168                                                           ) ;
1169 
1170         std::vector< std::vector< std::vector< unsigned > > > sd(
1171                                                                  _dg - 1 ,
1172                                                                  std::vector< std::vector< unsigned > >(
1173                                                                    2 ,
1174                                                                    std::vector< unsigned >( 2 , 0 )
1175                                                                  )
1176                                                                ) ;
1177 
1178         for ( unsigned j = 1 ; j < _dg ; j++ ) {
1179           for ( unsigned l = 0 ; l < 2 ; l++ ) {
1180             for ( unsigned v = 0 ; v < 2 ; v++ ) {
1181               /*
1182                * Compute  the column  indices  of  the linear  program
1183                * matrix  corresponding to  the  values  of the  second
1184                * differences in the constraints.
1185                */
1186               sd[ j - 1 ][ l ][ v ] = compute_second_difference_column_index
     (
1187                                                                                 p ,
1188                                                                                 j ,
1189                                                                                 l ,
1190                                                                                 v
1191                                                                                ) ;
1192             }
```

```
1193           }
1194         }
1195
1196
1197         /*
1198          * Constraints related to the sleeve lower envelope.
1199          */
1200
1201         for ( unsigned j = 1 ; j < _dg ; j++ ) {
1202           double du ;
1203           double dl ;
1204           try {
1205             du = ( u * _tf->alower( j , tp ) ) + ( ( 1 - u ) * _tf->
     alower( j , sp ) ) ;
1206             dl = ( u * _tf->aupper( j , tp ) ) + ( ( 1 - u ) * _tf->
     aupper( j , sp ) ) ;
1207           }
1208           catch ( const ExceptionObject& xpt ) {
1209             treat_exception( xpt ) ;
1210             exit( EXIT_FAILURE ) ;
1211           }
1212
1213           for ( sizetype v = 0 ; v < 2 ; v++ ) {
1214             double temp ;
1215             temp = dl * nl[ cr ][ sizetype( v ) ] ;
1216             if ( fabs( temp ) > 0 ) {
1217               _coefficients[ eqline     ].push_back( Coefficient( eqline     , sd[
     sizetype( j ) - 1 ][ 0 ][ sizetype( v ) ] , temp  ) ) ;
1218             }
1219
1220             temp = du * nl[ cr ][ sizetype( v ) ] ;
1221             if ( fabs( temp ) > 0 ) {
1222               _coefficients[ eqline     ].push_back( Coefficient( eqline     , sd[
     sizetype( j ) - 1 ][ 1 ][ sizetype( v ) ] , temp  ) ) ;
1223             }
1224
1225             temp = dl * nl[ cl ][ sizetype( v ) ] ;
1226             if ( fabs( temp ) > 0 ) {
1227               _coefficients[ eqline + 1 ].push_back( Coefficient( eqline + 1 , sd[
     sizetype( j ) - 1 ][ 0 ][ sizetype( v ) ] , temp  ) ) ;
1228             }
1229
1230             temp = du * nl[ cl ][ sizetype( v ) ] ;
1231             if ( fabs( temp ) > 0 ) {
1232               _coefficients[ eqline + 1 ].push_back( Coefficient( eqline + 1 , sd[
     sizetype( j ) - 1 ][ 1 ][ sizetype( v ) ] , temp  ) ) ;
1233             }
1234
1235             temp = dl * nu[ cr ][ sizetype( v ) ] ;
1236             if ( fabs( temp ) > 0 ) {
1237               _coefficients[ eqline + 2 ].push_back( Coefficient( eqline + 2 , sd[
     sizetype( j ) - 1 ][ 0 ][ sizetype( v ) ] , temp  ) ) ;
1238             }
1239
1240             temp = du * nu[ cr ][ sizetype( v ) ] ;
1241             if ( fabs( temp ) > 0 ) {
1242               _coefficients[ eqline + 2 ].push_back( Coefficient( eqline + 2 , sd[
     sizetype( j ) - 1 ][ 1 ][ sizetype( v ) ] , temp  ) ) ;
1243             }
1244
1245             temp = dl * nu[ cl ][ sizetype( v ) ] ;
1246             if ( fabs( temp ) > 0 ) {
1247               _coefficients[ eqline + 3 ].push_back( Coefficient( eqline + 3 , sd[
     sizetype( j ) - 1 ][ 0 ][ sizetype( v ) ] , temp ) ) ;
1248             }
1249
1250             temp = du * nu[ cl ][ sizetype( v ) ] ;
1251             if ( fabs( temp ) > 0 ) {
1252               _coefficients[ eqline + 3 ].push_back( Coefficient( eqline + 3 , sd[
     sizetype( j ) - 1 ][ 1 ][ sizetype( v ) ] , temp ) ) ;
1253             }
1254           }
1255         }
1256
1257         double p0 = ( ( 1 - u ) * ( 1 - sp ) ) + ( u * ( 1 - tp ) )  ;
1258         double pd = ( ( 1 - u ) * (     sp ) ) + ( u * (     tp ) )  ;
1259
1260         for ( sizetype v = 0 ; v < 2 ; v++ ) {
1261           double temp ;
1262           temp = p0 * nl[ cr ][ sizetype( v ) ] ;
1263           if ( fabs( temp ) > 0 ) {
```

```
1264            _coefficients[ eqline     ].push_back( Coefficient( eqline     , cp[ 0 ][
      sizetype( v ) ] , temp ) ) ;
1265          }
1266
1267          temp = pd * nl[ cr ][ sizetype( v ) ] ;
1268          if ( fabs( temp ) > 0 ) {
1269            _coefficients[ eqline     ].push_back( Coefficient( eqline     , cp[ 1 ][
      sizetype( v ) ] , temp ) ) ;
1270          }
1271
1272          temp = p0 * nl[ cl ][ sizetype( v ) ] ;
1273          if ( fabs( temp ) > 0 ) {
1274            _coefficients[ eqline + 1 ].push_back( Coefficient( eqline + 1 , cp[ 0 ][
      sizetype( v ) ] , temp ) ) ;
1275          }
1276
1277          temp = pd * nl[ cl ][ sizetype( v ) ] ;
1278          if ( fabs( temp ) > 0 ) {
1279            _coefficients[ eqline + 1 ].push_back( Coefficient( eqline + 1 , cp[ 1 ][
      sizetype( v ) ] , temp ) ) ;
1280          }
1281
1282          temp = p0 * nu[ cr ][ sizetype( v ) ] ;
1283          if ( fabs( temp ) > 0 ) {
1284            _coefficients[ eqline + 2 ].push_back( Coefficient( eqline + 2 , cp[ 0 ][
      sizetype( v ) ] , temp ) ) ;
1285          }
1286
1287          temp = pd * nu[ cr ][ sizetype( v ) ] ;
1288          if ( fabs( temp ) > 0 ) {
1289            _coefficients[ eqline + 2 ].push_back( Coefficient( eqline + 2 , cp[ 1 ][
      sizetype( v ) ] , temp ) ) ;
1290          }
1291
1292          temp = p0 * nu[ cl ][ sizetype( v ) ] ;
1293          if ( fabs( temp ) > 0 ) {
1294            _coefficients[ eqline + 3 ].push_back( Coefficient( eqline + 3 , cp[ 0 ][
      sizetype( v ) ] , temp ) ) ;
1295          }
1296
1297          temp = pd * nu[ cl ][ sizetype( v ) ] ;
1298          if ( fabs( temp ) > 0 ) {
1299            _coefficients[ eqline + 3 ].push_back( Coefficient( eqline + 3 , cp[ 1 ][
      sizetype( v ) ] , temp ) ) ;
1300          }
1301        }
1302
1303        _bounds[ eqline     ] = Bound( Bound::LTE , _lxcoords[ cl ] * nl[ cr ][ 0 ] +
      _lycoords[ cl ] * nl[ cr ][ 1 ] , eqline     ) ;
1304        _bounds[ eqline + 1 ] = Bound( Bound::LTE , _lxcoords[ cl ] * nl[ cl ][ 0 ] +
      _lycoords[ cl ] * nl[ cl ][ 1 ] , eqline + 1 ) ;
1305
1306        _bounds[ eqline + 2 ] = Bound( Bound::LTE , _uxcoords[ cl ] * nu[ cr ][ 0 ] +
      _uycoords[ cl ] * nu[ cr ][ 1 ] , eqline + 2 ) ;
1307        _bounds[ eqline + 3 ] = Bound( Bound::LTE , _uxcoords[ cl ] * nu[ cl ][ 0 ] +
      _uycoords[ cl ] * nu[ cl ][ 1 ] , eqline + 3 ) ;
1308
1309        /*
1310         * Increment equation counter
1311         */
1312        eqline += 4 ;
1313
1314        /*
1315         * Constraints related to the sleeve upper envelope.
1316         */
1317
1318        for ( unsigned j = 1 ; j < _dg ; j++ ) {
1319          double du ;
1320          double dl ;
1321          try {
1322            du = ( u * _tf->aupper( j , tp ) ) + ( ( 1 - u ) * _tf->
      aupper( j , sp ) ) ;
1323            dl = ( u * _tf->alower( j , tp ) ) + ( ( 1 - u ) * _tf->
      alower( j , sp ) ) ;
1324          }
1325          catch ( const ExceptionObject& xpt ) {
1326            treat_exception( xpt ) ;
1327            exit( EXIT_FAILURE ) ;
1328          }
1329
1330          for ( sizetype v = 0 ; v < 2 ; v++ ) {
```

```
1331            double temp ;
1332            temp = dl * nl[ cr ][ sizetype( v ) ] ;
1333            if ( fabs( temp ) > 0 ) {
1334              _coefficients[ eqline     ].push_back( Coefficient( eqline     , sd[
    sizetype( j ) - 1 ][ 0 ][ sizetype( v ) ] , temp ) ) ;
1335            }
1336
1337            temp = du * nl[ cr ][ sizetype( v ) ] ;
1338            if ( fabs( temp ) > 0 ) {
1339              _coefficients[ eqline     ].push_back( Coefficient( eqline     , sd[
    sizetype( j ) - 1 ][ 1 ][ sizetype( v ) ] , temp ) ) ;
1340            }
1341
1342            temp = dl * nl[ cl ][ sizetype( v ) ] ;
1343            if ( fabs( temp ) > 0 ) {
1344              _coefficients[ eqline + 1 ].push_back( Coefficient( eqline + 1 , sd[
    sizetype( j ) - 1 ][ 0 ][ sizetype( v ) ] , temp ) ) ;
1345            }
1346
1347            temp = du * nl[ cl ][ sizetype( v ) ] ;
1348            if ( fabs( temp ) > 0 ) {
1349              _coefficients[ eqline + 1 ].push_back( Coefficient( eqline + 1 , sd[
    sizetype( j ) - 1 ][ 1 ][ sizetype( v ) ] , temp ) ) ;
1350            }
1351
1352            temp = dl * nu[ cr ][ sizetype( v ) ] ;
1353            if ( fabs( temp ) > 0 ) {
1354              _coefficients[ eqline + 2 ].push_back( Coefficient( eqline + 2 , sd[
    sizetype( j ) - 1 ][ 0 ][ sizetype( v ) ] , temp ) ) ;
1355            }
1356
1357            temp = du * nu[ cr ][ sizetype( v ) ] ;
1358            if ( fabs( temp ) > 0 ) {
1359              _coefficients[ eqline + 2 ].push_back( Coefficient( eqline + 2 , sd[
    sizetype( j ) - 1 ][ 1 ][ sizetype( v ) ] , temp ) ) ;
1360            }
1361
1362            temp = dl * nu[ cl ][ sizetype( v ) ] ;
1363            if ( fabs( temp ) > 0 ) {
1364              _coefficients[ eqline + 3 ].push_back( Coefficient( eqline + 3 , sd[
    sizetype( j ) - 1 ][ 0 ][ sizetype( v ) ] , temp ) ) ;
1365            }
1366
1367            temp = du * nu[ cl ][ sizetype( v ) ] ;
1368            if ( fabs( temp ) > 0 ) {
1369              _coefficients[ eqline + 3 ].push_back( Coefficient( eqline + 3 , sd[
    sizetype( j ) - 1 ][ 1 ][ sizetype( v ) ] , temp ) ) ;
1370            }
1371          }
1372        }
1373
1374        for ( sizetype v = 0 ; v < 2 ; v++ ) {
1375          double temp ;
1376          temp = p0 * nl[ cr ][ sizetype( v ) ] ;
1377          if ( fabs( temp ) > 0 ) {
1378            _coefficients[ eqline     ].push_back( Coefficient( eqline     , cp[ 0 ][
    sizetype( v ) ] , temp ) ) ;
1379          }
1380
1381          temp = pd * nl[ cr ][ sizetype( v ) ] ;
1382          if ( fabs( temp ) > 0 ) {
1383            _coefficients[ eqline     ].push_back( Coefficient( eqline     , cp[ 1 ][
    sizetype( v ) ] , temp ) ) ;
1384          }
1385
1386          temp = p0 * nl[ cl ][ sizetype( v ) ] ;
1387          if ( fabs( temp ) > 0 ) {
1388            _coefficients[ eqline + 1 ].push_back( Coefficient( eqline + 1 , cp[ 0 ][
    sizetype( v ) ] , temp ) ) ;
1389          }
1390
1391          temp = pd * nl[ cl ][ sizetype( v ) ] ;
1392          if ( fabs( temp ) > 0 ) {
1393            _coefficients[ eqline + 1 ].push_back( Coefficient( eqline + 1 , cp[ 1 ][
    sizetype( v ) ] , temp ) ) ;
1394          }
1395
1396          temp = p0 * nu[ cr ][ sizetype( v ) ] ;
1397          if ( fabs( temp ) > 0 ) {
1398            _coefficients[ eqline + 2 ].push_back( Coefficient( eqline + 2 , cp[ 0 ][
    sizetype( v ) ] , temp ) ) ;
```

```
1399           }
1400
1401           temp = pd * nu[ cr ][ sizetype( v ) ] ;
1402           if ( fabs( temp ) > 0 ) {
1403             _coefficients[ eqline + 2 ].push_back( Coefficient( eqline + 2 , cp[ 1 ][
       sizetype( v ) ] , temp ) ) ;
1404           }
1405
1406           temp = p0 * nu[ cl ][ sizetype( v ) ] ;
1407           if ( fabs( temp ) > 0 ) {
1408             _coefficients[ eqline + 3 ].push_back( Coefficient( eqline + 3 , cp[ 0 ][
       sizetype( v ) ] , temp ) ) ;
1409           }
1410
1411           temp = pd * nu[ cl ][ sizetype( v ) ] ;
1412           if ( fabs( temp ) > 0 ) {
1413             _coefficients[ eqline + 3 ].push_back( Coefficient( eqline + 3 , cp[ 1 ][
       sizetype( v ) ] , temp ) ) ;
1414           }
1415         }
1416
1417         _bounds[ eqline     ] = Bound( Bound::LTE , _lxcoords[ cl ] * nl[ cr ][ 0 ] +
       _lycoords[ cl ] * nl[ cr ][ 1 ] , eqline     ) ;
1418         _bounds[ eqline + 1 ] = Bound( Bound::LTE , _lxcoords[ cl ] * nl[ cl ][ 0 ] +
       _lycoords[ cl ] * nl[ cl ][ 1 ] , eqline + 1 ) ;
1419
1420         _bounds[ eqline + 2 ] = Bound( Bound::LTE , _uxcoords[ cl ] * nu[ cr ][ 0 ] +
       _uycoords[ cl ] * nu[ cr ][ 1 ] , eqline + 2 ) ;
1421         _bounds[ eqline + 3 ] = Bound( Bound::LTE , _uxcoords[ cl ] * nu[ cl ][ 0 ] +
       _uycoords[ cl ] * nu[ cl ][ 1 ] , eqline + 3 ) ;
1422
1423         /*
1424          * Increment equation counter
1425          */
1426         eqline += 4 ;
1427       }
1428     }
1429
1430     return ;
1431   }
```

16.3.3.5 unsigned bc2::BuildCurve2D::compute_control_value_column_index (unsigned *p*, unsigned *i*, unsigned *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 piece of the spline curve to be threaded into the channel.

Parameters

p	Index of the spline curve piece.
i	Index of a control point of the p-th spline curve piece.
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 piece of the spline curve to be threaded into the channel.

Definition at line 382 of file buildcurve2d.cpp.

References _dg, and _np.

Referenced by compute_c0continuity_constraints(), compute_c1continuity_constraints(), compute_channel_corners↩_outside_sleeve_constraints(), compute_correspondence_constraints(), compute_min_max_constraints(), compute_↩sleeve_corners_in_channel_constraints(), get_lp_solver_result_information(), and set_up_structural_variables().

```
388   {
389 #ifdef DEBUGMODE
```

```
390     assert( p < _np ) ;
391     assert( i <= _dg ) ;
392     assert( v <= 1 ) ;
393 #endif
394
395     unsigned offset = ( ( 6 * _dg ) - 2 ) * p ;
396
397     return offset + ( 2 * i ) + v ;
398   }
```

16.3.3.6 void bc2::BuildCurve2D::compute_correspondence_constraints (unsigned & *eqline*) [private]

Computes the equations defining the correspondence constraints.

Parameters

eqline	A reference to the counter of equations.

Definition at line 566 of file buildcurve2d.cpp.

References _bounds, _closed, _coefficients, _dg, _lxcoords, _lycoords, _nc, _np, _uxcoords, _uycoords, and compute_control_value_column_index().

Referenced by build().

```
567   {
568     /*
569      * For  each curve  p, let  the coordinates  of the  first control
570      * point of  the p-th  curve be  equal to  the coordinates  of the
571      * middle point of the endpoints of the first c-section delimiting
572      * the p-th curve.
573      */
574     for ( unsigned p = 0 ; p < _np ; p++ ) {
575       /*
576        * Get the column index of the coordinates of the first point of
577        * the p-th spline curve.
578        */
579       unsigned jx = compute_control_value_column_index( p , 0 , 0 ) ;
580       unsigned jy = compute_control_value_column_index( p , 0 , 1 ) ;
581
582       /*
583        * Find  the index  of  the first  c-segment  covering the  p-th
584        * curve.
585        */
586       sizetype s = sizetype( _nc * p ) ;
587
588       double x = _lxcoords[ s ] + _uxcoords[ s ] ;
589       double y = _lycoords[ s ] + _uycoords[ s ] ;
590
591       _coefficients[ eqline ].push_back( Coefficient( eqline , jx ,  2 ) ) ;
592       _bounds[ eqline ] = Bound( Bound::EQT , x , eqline ) ;
593
594       ++eqline ;  // increment the equation counter ;
595
596       _coefficients[ eqline ].push_back( Coefficient( eqline , jy ,  2 ) ) ;
597       _bounds[ eqline ] = Bound( Bound::EQT , y , eqline ) ;
598
599       ++eqline ;  // increment the equation counter ;
600     }
601
602     if ( !_closed ) {
603       /*
604        * Get the column  index of the coordinates of  the last control
605        * point of the last spline curve.
606        */
607       unsigned jx = compute_control_value_column_index(
608                                                        _np - 1 ,
609                                                        _dg ,
610                                                        0
611                                                       ) ;
612       unsigned jy = compute_control_value_column_index(
613                                                        _np - 1 ,
614                                                        _dg ,
615                                                        1
```

```
616                                                 ) ;
617
618       /*
619        * Find the  index of  the last  c-segments delimiting  the last
620        * curve.
621        */
622       sizetype s = sizetype( _nc * _np ) ;
623
624       double x = _lxcoords[ s ] + _uxcoords[ s ] ;
625       double y = _lycoords[ s ] + _uycoords[ s ] ;
626
627       _coefficients[ eqline ].push_back( Coefficient( eqline , jx ,  2 ) ) ;
628       _bounds[ eqline ] = Bound( Bound::EQT , x , eqline ) ;
629
630       ++eqline ;  // increment the equation counter ;
631
632       _coefficients[ eqline ].push_back( Coefficient( eqline , jy ,  2 ) ) ;
633       _bounds[ eqline ] = Bound( Bound::EQT , y , eqline ) ;
634
635       ++eqline ;  // increment the equation counter ;
636     }
637
638     return ;
639   }
```

16.3.3.7 void bc2::BuildCurve2D::compute_min_max_constraints (unsigned & *eqline*) [private]

Computes the equations defining the min-max constraints.

Parameters

eqline	A reference to the counter of equations.

Definition at line 456 of file buildcurve2d.cpp.

References _bounds, _coefficients, _dg, _np, compute_control_value_column_index(), and compute_second_↩ difference_column_index().

Referenced by build().

```
457   {
458     /*
459      * For each  curve p, for each  second difference i, and  for each
460      * coordinate v of  the i-th second difference of  the p-th curve,
461      * computes  the  three  equations corresponding  to  the  min-max
462      * constraints.
463      */
464     for ( unsigned p = 0 ; p < _np ; p++ ) {
465       for ( unsigned i = 1 ; i < _dg ; i++ ) {
466         for ( unsigned v = 0 ; v < 2 ; v++ ) {
467           /*
468            * Get  the column  indices of  the lower  bound and  of the
469            * upper bound  of the  v-th coordinate  of the  i-th second
470            * difference.
471            *
472            */
473           unsigned jl = compute_second_difference_column_index(
474                                                                p ,
475                                                                i ,
476                                                                0 ,
477                                                                v
478                                                               ) ;
479
480           unsigned ju = compute_second_difference_column_index(
481                                                                p ,
482                                                                i ,
483                                                                1 ,
484                                                                v
485                                                               ) ;
486           /*
487            * Get  the  column indices  of  the  v-th coordinates  that
488            * define  the i-th  second  difference of  the p-th  spline
489            * curve.
490            */
```

```
491           unsigned c1 = compute_control_value_column_index(
492                                                            p ,
493                                                            i - 1 ,
494                                                            v
495                                                           ) ;
496           unsigned c2 = compute_control_value_column_index(
497                                                            p ,
498                                                            i ,
499                                                            v
500                                                           ) ;
501           unsigned c3 = compute_control_value_column_index(
502                                                            p ,
503                                                            i + 1 ,
504                                                            v
505                                                           ) ;
506
507           /*
508            * Set the nonzero coefficients of the next three equations.
509            */
510
511           /*
512            * The (unknown) upper  bound of the v-th  coordinate of the
513            * i-th second difference  must be greater than  or equal to
514            * the value of the corresponding v-th coordinate itself.
515            */
516           _coefficients[ eqline ].push_back( Coefficient( eqline , ju ,  1 ) ) ;
517           _coefficients[ eqline ].push_back( Coefficient( eqline , c1 , -1 ) ) ;
518           _coefficients[ eqline ].push_back( Coefficient( eqline , c2 ,  2 ) ) ;
519           _coefficients[ eqline ].push_back( Coefficient( eqline , c3 , -1 ) ) ;
520           _bounds[ eqline ] = Bound( Bound::GTE , 0 , eqline ) ;
521
522           ++eqline ;  // increment the equation counter ;
523
524           /*
525            * The (unknown) upper  bound of the v-th  coordinate of the
526            * i-th second difference  must be greater than  or equal to
527            * zero.
528            */
529           _coefficients[ eqline ].push_back( Coefficient( eqline , ju ,  1 ) ) ;
530           _bounds[ eqline ] = Bound( Bound::GTE , 0 , eqline ) ;
531
532           ++eqline ;  // increment the equation counter ;
533
534           /*
535            * The  sum  of the  upper  and  lower  bounds of  the  v-th
536            * coordinate of the i-th second difference must be equal to
537            * the  value of  the  v-th coordinate  of  the i-th  second
538            * difference.
539            */
540           _coefficients[ eqline ].push_back( Coefficient( eqline , ju ,  1 ) ) ;
541           _coefficients[ eqline ].push_back( Coefficient( eqline , jl ,  1 ) ) ;
542           _coefficients[ eqline ].push_back( Coefficient( eqline , c1 , -1 ) ) ;
543           _coefficients[ eqline ].push_back( Coefficient( eqline , c2 ,  2 ) ) ;
544           _coefficients[ eqline ].push_back( Coefficient( eqline , c3 , -1 ) ) ;
545           _bounds[ eqline ] = Bound( Bound::EQT , 0 , eqline ) ;
546
547           ++eqline ;  // increment the equation counter ;
548         }
549       }
550     }
551
552     return ;
553   }
```

16.3.3.8 void bc2::BuildCurve2D::compute_normal_to_lower_envelope (sizetype *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 311 of file buildcurve2d.cpp.

References _lxcoords, _lycoords, _nc, and _np.

Referenced by compute_channel_corners_outside_sleeve_constraints(), and compute_sleeve_corners_in_channel_↩ constraints().

```
317   {
318 #ifdef DEBUGMODE
319     assert( s < ( _np * _nc ) ) ;
320 #endif
321     sizetype t = s + 1 ;
322 
323     nx = _lycoords[ s ] - _lycoords[ t ] ;
324     ny = _lxcoords[ t ] - _lxcoords[ s ] ;
325 
326     return ;
327   }
```

16.3.3.9 void bc2::BuildCurve2D::compute_normal_to_upper_envelope (sizetype *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 344 of file buildcurve2d.cpp.

References _nc, _np, _uxcoords, and _uycoords.

Referenced by compute_channel_corners_outside_sleeve_constraints(), and compute_sleeve_corners_in_channel_↩ constraints().

```
350   {
351 #ifdef DEBUGMODE
352     assert( s < ( _np * _nc ) ) ;
353 #endif
354     sizetype t = s + 1 ;
355 
356     nx = _uycoords[ t ] - _uycoords[ s ] ;
357     ny = _uxcoords[ s ] - _uxcoords[ t ] ;
358 
359     return ;
360   }
```

16.3.3.10 unsigned bc2::BuildCurve2D::compute_second_difference_column_index (unsigned *p*, unsigned *i*, unsigned *l*, unsigned *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 piece of the spline curve to be threaded into the channel.

Parameters

p	Index of the spline curve piece.
i	Index of the second difference of the p-th spline curve piece.
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 piece of the spline curve to be threaded into the channel.

Definition at line 424 of file buildcurve2d.cpp.

References _dg, and _np.

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

```
431   {
432 #ifdef DEBUGMODE
433     assert( p < _np ) ;
434     assert( i >= 1 ) ;
435     assert( i < _dg ) ;
436     assert( l <= 1 ) ;
437     assert( v <= 1 ) ;
438 #endif
439
440     unsigned offset = ( ( 6 * _dg ) - 2 ) * p + ( 2 * _dg + 2 ) ;
441
442
443     return offset + ( 4 * ( i - 1 ) ) + ( 2 * l ) + v ;
444   }
```

16.3.3.11 void bc2::BuildCurve2D::compute_sleeve_corners_in_channel_constraints (unsigned & *eqline*) [private]

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

Parameters

eqline	A reference to the counter of equations.

Definition at line 822 of file buildcurve2d.cpp.

References _bounds, _coefficients, _dg, _lxcoords, _lycoords, _nc, _np, _tf, _uxcoords, _uycoords, bc2::TabulatedFunction::alower(), bc2::TabulatedFunction::aupper(), compute_control_value_column_index(), compute_normal_to_lower_envelope(), compute_normal_to_upper_envelope(), compute_second_difference_column_index(), and treat_exception.

Referenced by build().

```
823   {
824     /*
825      * Compute outward normals to the line segments of the c-segments.
826      */
827     std::vector< std::vector< double > > nl( _np * _nc , std::vector< double >( 2 , 0 ) ) ;
828     std::vector< std::vector< double > > nu( _np * _nc , std::vector< double >( 2 , 0 ) ) ;
829     for ( sizetype c = 0 ; c < sizetype( _np * _nc ) ; c++ ) {
830       compute_normal_to_lower_envelope( c , nl[ c ][ 0 ] , nl[ c ][ 1 ] ) ;
831       compute_normal_to_upper_envelope( c , nu[ c ][ 0 ] , nu[ c ][ 1 ] ) ;
832     }
833
834     /*
```

```
835      * For each  curve p, each  sleeve corner  of the p-th  curve, and
836      * each line segment  bounding the sleeve corner  (i.e., the lower
837      * and upper  line segment of  a c-segment), compute  a constraint
838      * that ensures that  the sleeve corner is on the  correct side of
839      * the segment.
840      */
841     for ( unsigned p = 0 ; p < _np ; p++ ) {
842       for ( unsigned s = 0 ; s <= _dg ; s++ ) {
843         /*
844          * Find the  index \e c of  the upper and lower  line segments
845          * that bound the s-th corner of the sleeve of the p-th curve.
846          */
847         sizetype c = sizetype( floor( s * ( double( _nc ) / _dg ) ) ) ;
848         if ( s == _dg ) {
849           --c ;
850         }
851         c += sizetype( ( p * _nc ) ) ;
852
853         double t = double( s ) / _dg ;
854
855         /*
856          * Compute  the column  indices of  the linear  program matrix
857          * corresponding  to the  second differences  and the  control
858          * points involved in the constraints associated with the c-th
859          * corner.
860          */
861         std::vector< std::vector< unsigned > > cp( 2 , std::vector< unsigned >( 2 , 0 ) ) ;
862
863         cp[ 0 ][ 0 ] = compute_control_value_column_index(
864                                                           p ,
865                                                           0 ,
866                                                           0
867                                                          ) ;
868         cp[ 0 ][ 1 ] = compute_control_value_column_index(
869                                                           p ,
870                                                           0 ,
871                                                           1
872                                                          ) ;
873         cp[ 1 ][ 0 ] = compute_control_value_column_index(
874                                                           p ,
875                                                           _dg ,
876                                                           0
877                                                           ) ;
878         cp[ 1 ][ 1 ] = compute_control_value_column_index(
879                                                           p ,
880                                                           _dg ,
881                                                           1
882                                                           ) ;
883
884         std::vector< std::vector< std::vector< unsigned > > > sd(
885                                                                  _dg - 1 ,
886                                                                  std::vector< std::vector< unsigned > >
887                                                                  (
888                                                                    2 ,
889                                                                    std::vector< unsigned >( 2 , 0 )
890                                                                  )
891                                                                 ) ;
892
893         for ( unsigned j = 1 ; j < _dg ; j++ ) {
894           for ( unsigned l = 0 ; l < 2 ; l++ ) {
895             for ( unsigned v = 0 ; v < 2 ; v++ ) {
896               /*
897                * Compute  the column  indices  of  the linear  program
898                * matrix  corresponding to  the  values  of the  second
899                * differences in the constraints.
900                */
901               sd[ j - 1 ][ l ][ v ] = compute_second_difference_column_index
    (
902                                                                                 p ,
903                                                                                 j ,
904                                                                                 l ,
905                                                                                 v
906                                                                                ) ;
907             }
908           }
909         }
910
911         /*
912          * Compute the constraints corresponding to the lower envelope of the sleeve.
913          */
914
```

```
915         for ( unsigned j = 1 ; j < _dg ; j++ ) {
916           double du ;
917           double dl ;
918           try {
919             du = _tf->alower( j , t ) ;
920             dl = _tf->aupper( j , t ) ;
921           }
922           catch ( const ExceptionObject& xpt ) {
923             treat_exception( xpt ) ;
924             exit( EXIT_FAILURE ) ;
925           }
926
927           for ( sizetype v = 0 ; v < 2 ; v++ ) {
928             double temp ;
929             temp = dl * nl[ c ][ sizetype( v ) ] ;
930             if ( fabs( temp ) > 0 ) {
931               _coefficients[ eqline     ].push_back( Coefficient( eqline     , sd[
    sizetype( j ) - 1 ][ 0 ][ sizetype( v ) ] ,  temp ) ) ;
932             }
933
934             temp = du * nl[ c ][ sizetype( v ) ] ;
935             if ( fabs( temp ) > 0 ) {
936               _coefficients[ eqline     ].push_back( Coefficient( eqline     , sd[
    sizetype( j ) - 1 ][ 1 ][ sizetype( v ) ] ,  temp ) ) ;
937             }
938
939             temp = dl * nu[ c ][ sizetype( v ) ] ;
940             if ( fabs( temp ) > 0 ) {
941               _coefficients[ eqline + 1 ].push_back( Coefficient( eqline + 1 , sd[
    sizetype( j ) - 1 ][ 0 ][ sizetype( v ) ] ,  temp ) ) ;
942             }
943
944             temp = du * nu[ c ][ sizetype( v ) ] ;
945             if ( fabs( temp ) > 0 ) {
946               _coefficients[ eqline + 1 ].push_back( Coefficient( eqline + 1 , sd[
    sizetype( j ) - 1 ][ 1 ][ sizetype( v ) ] ,  temp ) ) ;
947             }
948           }
949         }
950
951         for ( sizetype v = 0 ; v < 2 ; v++ ) {
952           double temp ;
953           temp = ( 1 - t ) * nl[ c ][ sizetype( v ) ] ;
954           if ( fabs( temp ) > 0 ) {
955             _coefficients[ eqline     ].push_back( Coefficient( eqline     , cp[ 0 ][
    sizetype( v ) ] , temp ) ) ;
956           }
957
958           temp = (     t ) * nl[ c ][ sizetype( v ) ] ;
959           if ( fabs( temp ) > 0 ) {
960             _coefficients[ eqline     ].push_back( Coefficient( eqline     , cp[ 1 ][
    sizetype( v ) ] , temp ) ) ;
961           }
962
963           temp = ( 1 - t ) * nu[ c ][ sizetype( v ) ] ;
964           if ( fabs( temp ) > 0 ) {
965             _coefficients[ eqline + 1 ].push_back( Coefficient( eqline + 1 , cp[ 0 ][
    sizetype( v ) ] , temp ) ) ;
966           }
967
968           temp = (     t ) * nu[ c ][ sizetype( v ) ] ;
969           if ( fabs( temp ) > 0 ) {
970             _coefficients[ eqline + 1 ].push_back( Coefficient( eqline + 1 , cp[ 1 ][
    sizetype( v ) ] , temp ) ) ;
971           }
972         }
973
974         _bounds[ eqline     ] = Bound( Bound::LTE , _lxcoords[ c ] * nl[ c ][ 0 ] +
    _lycoords[ c ] * nl[ c ][ 1 ] , eqline     ) ;
975         _bounds[ eqline + 1 ] = Bound( Bound::LTE , _uxcoords[ c ] * nu[ c ][ 0 ] +
    _uycoords[ c ] * nu[ c ][ 1 ] , eqline + 1 ) ;
976
977         /*
978          * Increment equation counter
979          */
980         eqline += 2 ;
981
982         /*
983          * Compute the constraints corresponding to the upper envelope of the sleeve.
984          */
985
```

```
986         for ( unsigned j = 1 ; j < _dg ; j++ ) {
987           double du ;
988           double dl ;
989           try {
990             du = _tf->aupper( j , t ) ;
991             dl = _tf->alower( j , t ) ;
992           }
993           catch ( const ExceptionObject& xpt ) {
994             treat_exception( xpt ) ;
995             exit( EXIT_FAILURE ) ;
996           }
997
998           for ( sizetype v = 0 ; v < 2 ; v++ ) {
999             double temp ;
1000             temp = dl * nl[ c ][ sizetype( v ) ] ;
1001             if ( fabs( temp ) > 0 ) {
1002               _coefficients[ eqline     ].push_back( Coefficient( eqline     , sd[
     sizetype( j ) - 1 ][ 0 ][ sizetype( v ) ] , temp ) ) ;
1003             }
1004
1005             temp = du * nl[ c ][ sizetype( v ) ] ;
1006             if ( fabs( temp ) > 0 ) {
1007               _coefficients[ eqline     ].push_back( Coefficient( eqline     , sd[
     sizetype( j ) - 1 ][ 1 ][ sizetype( v ) ] , temp ) ) ;
1008             }
1009
1010             temp = dl * nu[ c ][ sizetype( v ) ] ;
1011             if ( fabs( temp ) > 0 ) {
1012               _coefficients[ eqline + 1 ].push_back( Coefficient( eqline + 1 , sd[
     sizetype( j ) - 1 ][ 0 ][ sizetype( v ) ] , temp ) ) ;
1013             }
1014
1015             temp = du * nu[ c ][ sizetype( v ) ] ;
1016             if ( fabs( temp ) > 0 ) {
1017               _coefficients[ eqline + 1 ].push_back( Coefficient( eqline + 1 , sd[
     sizetype( j ) - 1 ][ 1 ][ sizetype( v ) ] , temp ) ) ;
1018             }
1019           }
1020         }
1021
1022         for ( sizetype v = 0 ; v < 2 ; v++ ) {
1023           double temp ;
1024           temp = ( 1 - t ) * nl[ c ][ sizetype( v ) ] ;
1025           if ( fabs( temp ) > 0 ) {
1026             _coefficients[ eqline     ].push_back( Coefficient( eqline     , cp[ 0 ][
     sizetype( v ) ] , temp ) ) ;
1027           }
1028
1029           temp = (     t ) * nl[ c ][ sizetype( v ) ] ;
1030           if ( fabs( temp ) > 0 ) {
1031             _coefficients[ eqline     ].push_back( Coefficient( eqline     , cp[ 1 ][
     sizetype( v ) ] , temp ) ) ;
1032           }
1033
1034           temp = ( 1 - t ) * nu[ c ][ sizetype( v ) ] ;
1035           if ( fabs( temp ) > 0 ) {
1036             _coefficients[ eqline + 1 ].push_back( Coefficient( eqline + 1 , cp[ 0 ][
     sizetype( v ) ] , temp ) ) ;
1037           }
1038
1039           temp = (     t ) * nu[ c ][ sizetype( v ) ] ;
1040           if ( fabs( temp ) > 0 ) {
1041             _coefficients[ eqline + 1 ].push_back( Coefficient( eqline + 1 , cp[ 1 ][
     sizetype( v ) ] , temp ) ) ;
1042           }
1043         }
1044
1045         _bounds[ eqline     ] = Bound( Bound::LTE , _lxcoords[ c ] * nl[ c ][ 0 ] +
     _lycoords[ c ] * nl[ c ][ 1 ] , eqline     ) ;
1046         _bounds[ eqline + 1 ] = Bound( Bound::LTE , _uxcoords[ c ] * nu[ c ][ 0 ] +
     _uycoords[ c ] * nu[ c ][ 1 ] , eqline + 1 ) ;
1047
1048         /*
1049          * Increment equation counter
1050          */
1051         eqline += 2 ;
1052       }
1053     }
1054
1055     return ;
1056   }
```

16.3.3.12 double bc2::BuildCurve2D::get_bound_of_ith_constraint (unsigned *i*) const throw ExceptionObject) [inline]

Returns the real value that bounds the i-th constraint.

Parameters

i	The index of a constraint.

Returns

The real value that bounds the i-th constraint.

Definition at line 446 of file buildcurve2d.hpp.

Referenced by write_lp().

```
447     {
448       if ( _coefficients.empty() ) {
449         std::stringstream ss( std::stringstream::in | std::stringstream::out ) ;
450         ss << "No constraint has been created so far" ;
451         throw ExceptionObject( __FILE__ , __LINE__ , ss.str().c_str() ) ;
452       }
453
454       if ( constsizetype( i ) >= _coefficients.size() ) {
455         std::stringstream ss( std::stringstream::in | std::stringstream::out ) ;
456         ss << "Constraint index is out of range" ;
457         throw ExceptionObject( __FILE__ , __LINE__ , ss.str().c_str() ) ;
458       }
459
460 #ifdef DEBUGMODE
461       assert( _bounds.size() == _coefficients.size() ) ;
462       assert( _bounds.size() > std::vector< std::vector< Bound > >::size_type( i ) ) ;
463 #endif
464
465       return _bounds[ std::vector< std::vector< Bound > >::size_type( i ) ].get_value() ;
466     }
```

16.3.3.13 unsigned bc2::BuildCurve2D::get_coefficient_identifier (unsigned *i*, unsigned *j*) const throw ExceptionObject) [inline]

Returns the number of the column that corresponds to the j-th coefficient of the i-th constraint in the constraint matrix of the instance of the linear program corresponding to the channel problem.

Parameters

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

Returns

The number of the column that corresponds to the j-th coefficient of the i-th constraint in the constraint matrix of the instance of the linear program corresponding to the channel problem.

Definition at line 373 of file buildcurve2d.hpp.

Referenced by write_lp().

```
374     {
375       if ( _coefficients.empty() ) {
376         std::stringstream ss( std::stringstream::in | std::stringstream::out ) ;
377         ss << "No constraint has been created so far" ;
378         throw ExceptionObject( __FILE__ , __LINE__ , ss.str().c_str() ) ;
379       }
380
```

```
381       if ( constsizetype( i ) >= _coefficients.size() ) {
382         std::stringstream ss( std::stringstream::in | std::stringstream::out ) ;
383         ss << "Constraint index is out of range" ;
384         throw ExceptionObject( __FILE__ , __LINE__ , ss.str().c_str() ) ;
385       }
386
387       if ( coeffsizetype( j ) >= _coefficients[
    constsizetype( i ) ].size() ) {
388         std::stringstream ss( std::stringstream::in | std::stringstream::out ) ;
389         ss << "Coefficient index is out of range" ;
390         throw ExceptionObject( __FILE__ , __LINE__ , ss.str().c_str() ) ;
391       }
392
393       return _coefficients[ constsizetype( i ) ][
    coeffsizetype( j ) ].get_col() ;
394     }
```

16.3.3.14 unsigned bc2::BuildCurve2D::get_coefficient_value (unsigned *i*, unsigned *j*) const throw ExceptionObject) [inline]

Returns the (i , j) entry of the matrix of constraints of the instance of the linear program corresponding to the channel problem.

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 of constraints of the instance of the linear program corresponding to the channel problem.

Definition at line 412 of file buildcurve2d.hpp.

Referenced by write_lp().

```
413     {
414       if ( _coefficients.empty() ) {
415         std::stringstream ss( std::stringstream::in | std::stringstream::out ) ;
416         ss << "No constraint has been created so far" ;
417         throw ExceptionObject( __FILE__ , __LINE__ , ss.str().c_str() ) ;
418       }
419
420       if ( constsizetype( i ) >= _coefficients.size() ) {
421         std::stringstream ss( std::stringstream::in | std::stringstream::out ) ;
422         ss << "Constraint index is out of range" ;
423         throw ExceptionObject( __FILE__ , __LINE__ , ss.str().c_str() ) ;
424       }
425
426       if ( coeffsizetype( j ) >= _coefficients[
    constsizetype( i ) ].size() ) {
427         std::stringstream ss( std::stringstream::in | std::stringstream::out ) ;
428         ss << "Coefficient index is out of range" ;
429         throw ExceptionObject( __FILE__ , __LINE__ , ss.str().c_str() ) ;
430       }
431
432       return _coefficients[ constsizetype( i ) ][
    coeffsizetype( j ) ].get_value() ;
433     }
```

16.3.3.15 double bc2::BuildCurve2D::get_control_value (unsigned *p*, unsigned *i*, unsigned *v*) const throw ExceptionObject) [inline]

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

Parameters

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

Returns

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

Definition at line 287 of file buildcurve2d.hpp.

Referenced by write_solution().

```
293     {
294       if ( p >= _np ) {
295         std::stringstream ss( std::stringstream::in | std::stringstream::out ) ;
296         ss << "Index of the curve is out of range" ;
297         throw ExceptionObject( __FILE__ , __LINE__ , ss.str().c_str() ) ;
298       }
299
300       if ( i > _dg ) {
301         std::stringstream ss( std::stringstream::in | std::stringstream::out ) ;
302         ss << "Index of the control point is out of range" ;
303         throw ExceptionObject( __FILE__ , __LINE__ , ss.str().c_str() ) ;
304       }
305
306       if ( v >= 2 ) {
307         std::stringstream ss( std::stringstream::in | std::stringstream::out ) ;
308         ss << "Index of the Cartesian coordinate is out of range" ;
309         throw ExceptionObject( __FILE__ , __LINE__ , ss.str().c_str() ) ;
310       }
311
312       if ( _ctrlpts.empty() ) {
313         std::stringstream ss( std::stringstream::in | std::stringstream::out ) ;
314         ss << "Control points have not been computed" ;
315         throw ExceptionObject( __FILE__ , __LINE__ , ss.str().c_str() ) ;
316       }
317
318       sizetype index = ( 2 * ( _dg + 1 ) * p ) + ( 2 * i ) + v ;
319
320       return _ctrlpts[ index ] ;
321     }
```

16.3.3.16 double bc2::BuildCurve2D::get_lower_bound_on_second_difference_value (unsigned *p*, unsigned *i*, unsigned *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 vector of the p-th curve piece of the spline curve threaded into the channel.

Parameters

p	The index of the p-th curve piece of the spline.
i	The index of the i-th second difference of the p-th curve of the spline.
v	The v-th Cartesian coordinate of the i-th control point of the p-th curve of the 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 piece of the spline curve threaded into the channel.

Definition at line 598 of file buildcurve2d.hpp.

References _dg.

```
604     {
605       if ( p >= _np ) {
606         std::stringstream ss( std::stringstream::in | std::stringstream::out ) ;
607         ss << "Index of the curve is out of range" ;
608         throw ExceptionObject( __FILE__ , __LINE__ , ss.str().c_str() ) ;
609       }
610
611       if ( ( i < 1 ) || ( i > _dg ) ) {
612         std::stringstream ss( std::stringstream::in | std::stringstream::out ) ;
613         ss << "Index of the second difference vector is out of range" ;
614         throw ExceptionObject( __FILE__ , __LINE__ , ss.str().c_str() ) ;
615       }
616
617       if ( v >= 2 ) {
618         std::stringstream ss( std::stringstream::in | std::stringstream::out ) ;
619         ss << "Index of the Cartesian coordinate is out of range" ;
620         throw ExceptionObject( __FILE__ , __LINE__ , ss.str().c_str() ) ;
621       }
622
623       if ( _secdiff.empty() ) {
624         std::stringstream ss( std::stringstream::in | std::stringstream::out ) ;
625         ss << "Second differences have not been computed" ;
626         throw ExceptionObject( __FILE__ , __LINE__ , ss.str().c_str() ) ;
627       }
628
629       sizetype index = ( 4 * ( _dg - 1 ) * p ) + ( 4 * ( i - 1 ) ) + v ;
630
631       return _secdiff[ index ] ;
632     }
```

16.3.3.17 void bc2::BuildCurve2D::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 1715 of file buildcurve2d.cpp.

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

Referenced by solve_lp().

```
1716   {
1717     /*
1718      * Obtain the control points of the spline curve.
1719      */
1720     for ( unsigned p = 0 ; p < _np ; p++ ) {
1721       for ( unsigned k = 0 ; k <= _dg ; k++ ) {
1722         for ( unsigned v = 0 ; v < 2 ; v++ ) {
1723           unsigned c = compute_control_value_column_index(
1724                                                           p ,
1725                                                           k ,
1726                                                           v
1727                                                          ) ;
1728           _ctrlpts.push_back(
1729                              glp_get_col_prim(
1730                                               lp ,
1731                                               int( c ) + 1
1732                                              )
1733                             ) ;
1734         }
1735       }
1736     }
1737
1738     /*
1739      * Obtain the lower and upper bounds of the second differences.
1740      */
1741     for ( unsigned p = 0 ; p < _np ; p++ ) {
1742       for ( unsigned k = 1 ; k < _dg ; k++ ) {
```

```
1743          for ( unsigned l = 0 ; l < 2 ; l++ ) {
1744            for ( unsigned v = 0 ; v < 2 ; v++ ) {
1745              unsigned c = compute_second_difference_column_index(
1746                                                                  p ,
1747                                                                  k ,
1748                                                                  l ,
1749                                                                  v
1750                                                                ) ;
1751
1752              _secdiff.push_back(
1753                                  glp_get_col_prim(
1754                                                   lp ,
1755                                                   int( c ) + 1
1756                                                  )
1757                                 ) ;
1758            }
1759          }
1760        }
1761      }
1762
1763      /*
1764       * Obtain the minimum value of the objective function.
1765       */
1766      _ofvalue = glp_get_obj_val( lp ) ;
1767
1768      return ;
1769    }
```

16.3.3.18 unsigned bc2::BuildCurve2D::get_number_of_coefficients_in_the_ith_constraint (unsigned *i*) const throw ExceptionObject) [inline]

Returns the number of coefficients of the i-th constraint of the instance of the linear program corresponding to the channel problem solved by this class.

Parameters

i	The index of a constraint.

Returns

The number of coefficients of the i-th constraint of the instance of the linear program corresponding to the channel problem solved by this class.

Definition at line 338 of file buildcurve2d.hpp.

Referenced by write_lp().

```
339      {
340        if ( _coefficients.empty() ) {
341          std::stringstream ss( std::stringstream::in | std::stringstream::out ) ;
342          ss << "No constraint has been created so far" ;
343          throw ExceptionObject( __FILE__ , __LINE__ , ss.str().c_str() ) ;
344        }
345
346        if ( constsizetype( i ) >= _coefficients.size() ) {
347          std::stringstream ss( std::stringstream::in | std::stringstream::out ) ;
348          ss << "Constraint index is out of range" ;
349          throw ExceptionObject( __FILE__ , __LINE__ , ss.str().c_str() ) ;
350        }
351
352        return unsigned( _coefficients[ constsizetype( i ) ].size() ) ;
353      }
```

16.3.3.19 unsigned bc2::BuildCurve2D::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 258 of file buildcurve2d.hpp.

Referenced by write_lp().

```
259     {
260       if ( _coefficients.empty() ) {
261         std::stringstream ss( std::stringstream::in | std::stringstream::out ) ;
262         ss << "No constraint has been created so far" ;
263         throw ExceptionObject( __FILE__ , __LINE__ , ss.str().c_str() ) ;
264       }
265
266       return unsigned( _coefficients.size() ) ;
267     }
```

16.3.3.20 unsigned bc2::BuildCurve2D::get_number_of_curve_pieces ( ) const [inline]

Returns the number of curves pieces of the spline curve.

Returns

The number of curves pieces of the spline curve.

Definition at line 226 of file buildcurve2d.hpp.

References _np.

Referenced by write_lp(), and write_solution().

```
227     {
228       return _np ;
229     }
```

16.3.3.21 std::string bc2::BuildCurve2D::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 720 of file buildcurve2d.hpp.

Referenced by main().

```
721     {
722       std::string message ;
723       switch ( error ) {
724         case GLP_EBADB :
725           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." ;
726           break ;
```

```
727         case GLP_ESING :
728           message = "Unable to start the search because the basis matrix corresponding to the initial
      basis is singular within the working precision." ;
729           break ;
730         case GLP_ECOND :
731           message = "Unable to start the search because the basis matrix corresponding to the initial
      basis is ill-conditioned." ;
732           break ;
733         case GLP_EBOUND :
734           message = "Unable to start the search because some double-bounded variables have incorrect
      bounds." ;
735           break ;
736         case GLP_EFAIL :
737           message = "The search was prematurely terminated due to the solver failure." ;
738           break ;
739         case GLP_EOBJLL :
740           message = "The search was prematurely terminated because the objective function being maximized
      has reached its lower limit and continues decreasing." ;
741           break ;
742         case GLP_EOBJUL :
743           message = "The search was prematurely terminated because the objective function being minimized
      has reached its upper limit and continues increasing." ;
744           break ;
745         case GLP_EITLIM :
746           message = "The search was prematurely terminated because the simplex iteration limit has been
      exceeded." ;
747           break ;
748         case GLP_ETMLIM :
749           message = "The search was prematurely terminated because the time limit has been exceeded." ;
750           break ;
751         case GLP_ENOPFS :
752           message = "The LP problem instance has no primal feasible solution." ;
753           break ;
754         case GLP_ENODFS :
755           message = "The LP problem instance has no dual feasible solution." ;
756           break ;
757         default :
758           message = "Unknown reason." ;
759           break ;
760       }
761
762       return message ;
763     }
```

16.3.3.22 unsigned bc2::BuildCurve2D::get_spline_degree ( ) const [inline]

Returns the degree of the spline curve.

Returns

The degree of the spline curve.

Definition at line 240 of file buildcurve2d.hpp.

References _dg.

Referenced by write_lp(), and write_solution().

```
241     {
242       return _dg ;
243     }
```

16.3.3.23 double bc2::BuildCurve2D::get_upper_bound_on_second_difference_value (unsigned *p*, unsigned *i*, unsigned *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 piece of the spline curve threaded into the channel.

Parameters

p	The index of the p-th curve piece of the spline.
i	The index of the i-th second difference of the p-th curve of the spline.
v	The v-th Cartesian coordinate of the i-th control point of the p-th curve of the 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 piece of the spline curve threaded into the channel.

Definition at line 655 of file buildcurve2d.hpp.

References _dg.

```
661     {
662       if ( p >= _np ) {
663         std::stringstream ss( std::stringstream::in | std::stringstream::out ) ;
664         ss << "Index of the curve is out of range" ;
665         throw ExceptionObject( __FILE__ , __LINE__ , ss.str().c_str() ) ;
666       }
667
668       if ( ( i < 1 ) || ( i > _dg ) ) {
669         std::stringstream ss( std::stringstream::in | std::stringstream::out ) ;
670         ss << "Index of the second difference vector is out of range" ;
671         throw ExceptionObject( __FILE__ , __LINE__ , ss.str().c_str() ) ;
672       }
673
674       if ( v >= 2 ) {
675         std::stringstream ss( std::stringstream::in | std::stringstream::out ) ;
676         ss << "Index of the Cartesian coordinate is out of range" ;
677         throw ExceptionObject( __FILE__ , __LINE__ , ss.str().c_str() ) ;
678       }
679
680       if ( _secdiff.empty() ) {
681         std::stringstream ss( std::stringstream::in | std::stringstream::out ) ;
682         ss << "Second differences have not been computed" ;
683         throw ExceptionObject( __FILE__ , __LINE__ , ss.str().c_str() ) ;
684       }
685
686       sizetype index = ( 4 * ( _dg - 1 ) * p ) + ( 4 * ( i - 1 ) ) + 2 + v ;
687
688       return _secdiff[ index ] ;
689     }
```

16.3.3.24 double bc2::BuildCurve2D::h (double *u*) const [private]

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

Parameters

u	A parameter point in the real line.

Returns

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

Definition at line 279 of file buildcurve2d.cpp.

References _dg.

Referenced by set_up_lp_constraints().

```
280   {
281     const double onedth = 1.0 / _dg ;
```

```
282
283     if ( u <= -onedth ) {
284       return 0 ;
285     }
286     else if ( u <= 0 ) {
287       return _dg * u + 1 ;
288     }
289     else if ( u <= onedth ) {
290       return 1 - _dg * u ;
291     }
292
293     return 0 ;
294   }
```

16.3.3.25 bool bc2::BuildCurve2D::is_equality (unsigned *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 482 of file buildcurve2d.hpp.

Referenced by write_lp().

```
483     {
484       if ( _coefficients.empty() ) {
485         std::stringstream ss( std::stringstream::in | std::stringstream::out ) ;
486         ss << "No constraint has been created so far" ;
487         throw ExceptionObject( __FILE__ , __LINE__ , ss.str().c_str() ) ;
488       }
489
490       if ( constsizetype( i ) >= _coefficients.size() ) {
491         std::stringstream ss( std::stringstream::in | std::stringstream::out ) ;
492         ss << "Constraint index is out of range" ;
493         throw ExceptionObject( __FILE__ , __LINE__ , ss.str().c_str() ) ;
494       }
495
496 #ifdef DEBUGMODE
497       assert( _bounds.size() == _coefficients.size() ) ;
498       assert( _bounds.size() > std::vector< std::vector< Bound > >::size_type( i ) ) ;
499 #endif
500
501       return _bounds[ std::vector< std::vector< Bound > >::size_type( i ) ].get_type() == Bound::EQT
      ;
502     }
```

16.3.3.26 bool bc2::BuildCurve2D::is_greater_than_or_equal_to (unsigned *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 518 of file buildcurve2d.hpp.

Referenced by write_lp().

```
519     {
520       if ( _coefficients.empty() ) {
521         std::stringstream ss( std::stringstream::in | std::stringstream::out ) ;
522         ss << "No constraint has been created so far" ;
523         throw ExceptionObject( __FILE__ , __LINE__ , ss.str().c_str() ) ;
524       }
525 
526       if ( constsizetype( i ) >= _coefficients.size() ) {
527         std::stringstream ss( std::stringstream::in | std::stringstream::out ) ;
528         ss << "Constraint index is out of range" ;
529         throw ExceptionObject( __FILE__ , __LINE__ , ss.str().c_str() ) ;
530       }
531 
532 #ifdef DEBUGMODE
533       assert( _bounds.size() == _coefficients.size() ) ;
534       assert( _bounds.size() > std::vector< std::vector< Bound > >::size_type( i ) ) ;
535 #endif
536 
537       return _bounds[ std::vector< std::vector< Bound > >::size_type( i ) ].get_type() == Bound::GTE
      ;
538     }
```

16.3.3.27 bool bc2::BuildCurve2D::is_less_than_or_equal_to (unsigned *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 555 of file buildcurve2d.hpp.

Referenced by write_lp().

```
556     {
557       if ( _coefficients.empty() ) {
558         std::stringstream ss( std::stringstream::in | std::stringstream::out ) ;
559         ss << "No constraint has been created so far" ;
560         throw ExceptionObject( __FILE__ , __LINE__ , ss.str().c_str() ) ;
561       }
562 
563       if ( constsizetype( i ) >= _coefficients.size() ) {
564         std::stringstream ss( std::stringstream::in | std::stringstream::out ) ;
565         ss << "Constraint index is out of range" ;
566         throw ExceptionObject( __FILE__ , __LINE__ , ss.str().c_str() ) ;
567       }
568 
569 #ifdef DEBUGMODE
570       assert( _bounds.size() == _coefficients.size() ) ;
571       assert( _bounds.size() > std::vector< std::vector< Bound > >::size_type( i ) ) ;
572 #endif
573 
574       return _bounds[ std::vector< std::vector< Bound > >::size_type( i ) ].get_type() == Bound::LTE
      ;
575     }
```

16.3.3.28 double bc2::BuildCurve2D::lf (double *u*, double *b0*, double *bd*) const [private]

Computes the value of the affine function ℓ at a given point of the interval $[0, 1]$ of the real line.

Parameters

u	A parameter point in the interval $[0, 1]$.
b0	The x- or y-coordinate of the first control point of a Bézier curve of degree d.
bd	The x- or y-coordinate of the last control point of a Bézier curve of degree d.

Returns

The value of the affine function ℓ at a given point of the interval $[0, 1]$ of the real line.

Definition at line 248 of file buildcurve2d.cpp.

```
254   {
255 #ifdef DEBUGMODE
256     assert( u >= 0 ) ;
257     assert( u <= 1 ) ;
258 #endif
259
260     double res = b0 + ( bd - b0 ) * u ;
261
262     return res ;
263   }
```

16.3.3.29 double bc2::BuildCurve2D::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 704 of file buildcurve2d.hpp.

References _ofvalue.

```
705     {
706       return _ofvalue ;
707     }
```

16.3.3.30 void bc2::BuildCurve2D::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 1525 of file buildcurve2d.cpp.

References _bounds, _coefficients, and h().

Referenced by solve_lp().

```
1526   {
1527     /*
1528      * Set up the bounds on the constraints of the problem.
1529      */
1530
1531     for ( std::vector< Bound >::size_type j = 0 ; j < _bounds.size() ; j++ ) {
1532 #ifdef DEBUGMODE
1533       assert( unsigned( j ) == _bounds[ j ].get_row() ) ;
1534 #endif
1535
1536       int i = int( _bounds[ j ].get_row() + 1 ) ;
1537
1538       std::stringstream ss ( std::stringstream::in | std::stringstream::out ) ;
1539       ss << "c" << i ;
1540       glp_set_row_name( lp , i , ss.str().c_str() ) ;
1541
1542       double val = _bounds[ j ].get_value() ;
1543       if ( _bounds[ j ].get_type() == Bound::LTE ) {
1544         glp_set_row_bnds( lp , i , GLP_UP ,   0 , val ) ;
1545       }
1546       else if ( _bounds[ j ].get_type() == Bound::GTE ) {
1547         glp_set_row_bnds( lp , i , GLP_LO , val ,   0 ) ;
1548       }
1549       else {
1550         glp_set_row_bnds( lp , i , GLP_FX , val , val ) ;
1551       }
1552     }
1553
1554
1555     /*
1556      * Obtain the coefficients of the constraints of the problem.
1557      */
1558
1559     std::vector< int    > ia ; ia.push_back( 0 ) ;  // GLPK starts indexing array \e ia at 1
1560     std::vector< int    > ja ; ja.push_back( 0 ) ;  // GLPK starts indexing array \e ja at 1
1561     std::vector< double > ar ; ar.push_back( 0 ) ;  // GLPK starts indexing array \e ar at 1
1562
1563     int h = 0 ;
1564     for ( std::vector< std::vector< Coefficient > >::size_type j = 0 ; j <
      _coefficients.size() ; j++ ) {
1565       for ( std::vector< Coefficient >::size_type k = 0 ; k < _coefficients[ j ].size() ; k++
     ) {
1566 #ifdef DEBUGMODE
1567         assert( _coefficients[ j ][ k ].get_row() == unsigned( j ) ) ;
1568 #endif
1569         ia.push_back( int( _coefficients[ j ][ k ].get_row() + 1 ) ) ;
1570         ja.push_back( int( _coefficients[ j ][ k ].get_col() + 1 ) ) ;
1571         ar.push_back( _coefficients[ j ][ k ].get_value() ) ;
1572         ++h ;
1573       }
1574     }
1575
1576     glp_load_matrix(
1577                     lp ,
1578                     h ,
1579                     &ia[ 0 ] ,
1580                     &ja[ 0 ] ,
1581                     &ar[ 0 ]
1582                     ) ;
1583
1584     return ;
1585   }
```

16.3.3.31 void bc2::BuildCurve2D::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 1676 of file buildcurve2d.cpp.

References _dg, _np, and compute_second_difference_column_index().

Referenced by solve_lp().

```
1677   {
1678     for ( unsigned p = 0 ; p < _np ; p++ ) {
1679       for ( unsigned k = 1 ; k < _dg ; k++ ) {
1680         for ( unsigned l = 0 ; l < 2 ; l++ ) {
1681           for ( unsigned v = 0 ; v < 2 ; v++ ) {
1682             unsigned c = compute_second_difference_column_index(
1683                                                                 p ,
1684                                                                 k ,
1685                                                                 l ,
1686                                                                 v
1687                                                                ) ;
1688
1689             if ( l == 0 ) {
1690               glp_set_obj_coef( lp , int( c ) + 1 , -1 ) ;
1691             }
1692             else {
1693               glp_set_obj_coef( lp , int( c ) + 1 ,  1 ) ;
1694             }
1695           }
1696         }
1697       }
1698     }
1699
1700     return ;
1701   }
```

16.3.3.32 void bc2::BuildCurve2D::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 1599 of file buildcurve2d.cpp.

References _dg, _np, compute_control_value_column_index(), and compute_second_difference_column_index().

Referenced by solve_lp().

```
1600   {
1601     for ( unsigned p = 0 ; p < _np ; p++ ) {
1602       for ( unsigned k = 1 ; k < _dg ; k++ ) {
1603         for ( unsigned l = 0 ; l < 2 ; l++ ) {
1604           for ( unsigned v = 0 ; v < 2 ; v++ ) {
1605             unsigned c = compute_second_difference_column_index(
1606                                                                 p ,
1607                                                                 k ,
1608                                                                 l ,
1609                                                                 v
1610                                                                ) ;
1611
1612             if ( l == 0 ) {
1613               std::stringstream ss ( std::stringstream::in | std::stringstream::out ) ;
1614               if ( v == 0 ) {
1615                 ss << "mx" << ( ( _dg - 1 ) * p ) + k ;
1616               }
1617               else {
1618                 ss << "my" << ( ( _dg - 1 ) * p ) + k ;
1619               }
1620               glp_set_col_name( lp , int( c ) + 1 , ss.str().c_str() ) ;
1621               glp_set_col_bnds( lp , int( c ) + 1 , GLP_UP , 0 , 0 ) ;
1622             }
1623             else {
```

```
1624              std::stringstream ss ( std::stringstream::in | std::stringstream::out ) ;
1625              if ( v == 0 ) {
1626                ss << "px" << ( ( _dg - 1 ) * p ) + k ;
1627              }
1628              else {
1629                ss << "py" << ( ( _dg - 1 ) * p ) + k ;
1630              }
1631              glp_set_col_name( lp , int( c ) + 1 , ss.str().c_str() ) ;
1632              glp_set_col_bnds( lp , int( c ) + 1 , GLP_LO , 0 , 0 ) ;
1633            }
1634          }
1635        }
1636      }
1637    }
1638
1639    for ( unsigned p = 0 ; p < _np ; p++ ) {
1640      for ( unsigned k = 0 ; k <= _dg ; k++ ) {
1641        for ( unsigned v = 0 ; v < 2 ; v++ ) {
1642          unsigned c = compute_control_value_column_index(
1643                                                          p ,
1644                                                          k ,
1645                                                          v
1646                                                         ) ;
1647
1648          std::stringstream ss ( std::stringstream::in | std::stringstream::out ) ;
1649          if ( v == 0 ) {
1650            ss << "x" << ( ( _dg + 1 ) * p ) + k + 1 ;
1651          }
1652          else {
1653            ss << "y" << ( ( _dg + 1 ) * p ) + k + 1 ;
1654          }
1655          glp_set_col_name( lp , int( c ) + 1 , ss.str().c_str() ) ;
1656          glp_set_col_bnds( lp , int( c ) + 1 , GLP_FR , 0 , 0 ) ;
1657        }
1658      }
1659    }
1660
1661    return ;
1662  }
```

16.3.3.33 int bc2::BuildCurve2D::solve_lp (sizetype *rows*, sizetype *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 1447 of file buildcurve2d.cpp.

References get_lp_solver_result_information(), set_up_lp_constraints(), set_up_objective_function(), and set_up_↩ structural_variables().

Referenced by build().

```
1451  {
1452    /*
1453     * Create the LP problem.
1454     */
1455    glp_prob* lp = glp_create_prob() ;
1456
1457    /*
1458     * Set up the number of constraints and structural variables.
1459     */
1460    glp_add_rows( lp , int( rows ) ) ;
```

```
1461     glp_add_cols( lp , int( cols ) ) ;
1462
1463     /*
1464      * Set the problem as a minimization one.
1465      */
1466     glp_set_obj_dir( lp , GLP_MIN ) ;
1467
1468     /*
1469      * Set up the constraints of the problem.
1470      */
1471     set_up_lp_constraints( lp ) ;
1472
1473     /*
1474      * Define bounds on the structural variables of the problem.
1475      */
1476     set_up_structural_variables( lp ) ;
1477
1478     /*
1479      * Define objective function.
1480      */
1481     set_up_objective_function( lp ) ;
1482
1483     /*
1484      * Set parameters of the solver.
1485      */
1486     glp_smcp param ;
1487     glp_init_smcp( &param ) ;
1488
1489     param.msg_lev  = GLP_MSG_OFF ;
1490     param.presolve = GLP_ON ;
1491
1492     /*
1493      * Call the solver.
1494      */
1495
1496     int res = glp_simplex( lp , &param ) ;
1497
1498     if ( res == 0 ) {
1499       /*
1500        * Get the solver result information.
1501        */
1502       get_lp_solver_result_information( lp ) ;
1503     }
1504
1505     /*
1506      * Release memory held by the solver.
1507      */
1508     glp_delete_prob( lp ) ;
1509
1510     return res ;
1511   }
```

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

- buildcurve2d.hpp
- buildcurve2d.cpp

16.4 bc2::Coefficient Class Reference

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

`#include <coefficient.hpp>`

Public Member Functions

- Coefficient ()

  Creates an instance of this class.
- Coefficient (unsigned row, unsigned 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.

- unsigned 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.

- unsigned get_col () const

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

- double get_value () const

  Returns the value of this coefficient.

Protected Attributes

- unsigned _row

  The identifier of the constraint this coefficient belongs to.

- unsigned _col

  The identifier of the unknown multiplied by this coefficient.

- double _value

  The coefficient value.

16.4.1 Detailed Description

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

Definition at line 56 of file coefficient.hpp.

16.4.2 Constructor & Destructor Documentation

16.4.2.1 bc2::Coefficient::Coefficient (unsigned *row,* unsigned *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 105 of file coefficient.hpp.

```
106     :
107       _row( row ) ,
108       _col( col ) ,
109       _value( value )
110     {
111     }
```

16.4.2.2 bc2::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 123 of file coefficient.hpp.

```
124     :
125       _row( c._row ) ,
126       _col( c._col ) ,
127       _value( c._value )
128     {
129     }
```

16.4.3 Member Function Documentation

16.4.3.1 unsigned bc2::Coefficient::get_col ( ) const [inline]

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

Returns

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

Definition at line 175 of file coefficient.hpp.

References _col.

```
176     {
177       return _col ;
178     }
```

16.4.3.2 unsigned bc2::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 the coefficient is associated with.

Definition at line 156 of file coefficient.hpp.

References _row.

```
157     {
158       return _row ;
159     }
```

16.4.3.3 double bc2::Coefficient::get_value ( ) const [inline]

Returns the value of this coefficient.

Returns

The value of this coefficient.

Definition at line 189 of file coefficient.hpp.

References _value.

```
190     {
191       return _value ;
192     }
```

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

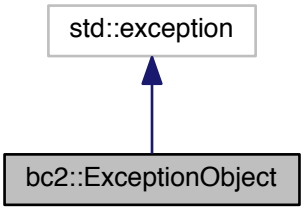
- coefficient.hpp

16.5 bc2::ExceptionObject Class Reference

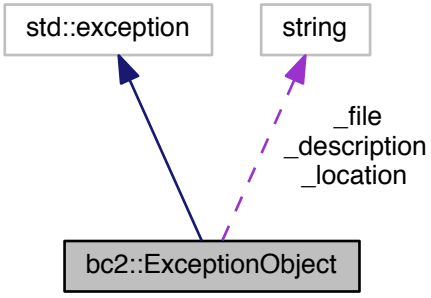
This class extends class *exception* of STL and provides us with a customized way of handling exceptions and showing error messages.

`#include <exceptionobject.hpp>`

Inheritance diagram for bc2::ExceptionObject:



Collaboration diagram for bc2::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 triggers this exception.

- virtual const char * get_file () const

  Returns the name of the file containing the line that triggers the exception.

- virtual unsigned get_line () const

  Returns the line that triggers this exception.

- virtual const char * what () const throw ()

  Returns a description of the error causing 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 bc2::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 is triggered.
ln	Number of the line containined the instruction that caused the exception.

Definition at line 126 of file exceptionobject.hpp.

```
127     :
128       _location( "Unknown" ) ,
129       _description( "Unknown" ) ,
130       _file( file ) ,
131       _line( ln )
132     {
133     }
```

16.5.2.2 bc2::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 is triggered.
ln	Number of the line containined 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.

```
150     :
151       _location( "Unknown" ) ,
152       _description( desc ) ,
153       _file( file ) ,
154       _line( ln )
155     {
156     }
```

16.5.2.3 bc2::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 is triggered.
ln	Number of the line containined 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 bc2::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 * bc2::ExceptionObject::get_description ( ) const [inline], [virtual]

Returns a description of the error that triggers this exception.

Returns

A description of the error that triggers this exception.

Definition at line 323 of file exceptionobject.hpp.

```
324     {
325       return _description.c_str() ;
326     }
```

16.5.3.2 const char * bc2::ExceptionObject::get_file ( ) const [inline], [virtual]

Returns the name of the file containing the line that triggers the exception.

Returns

The name of the file containing the line that triggers the exception.

Definition at line 339 of file exceptionobject.hpp.

```
340     {
341       return _file.c_str() ;
342     }
```

16.5.3.3 unsigned bc2::ExceptionObject::get_line ( ) const [inline], [virtual]

Returns the line that triggers this exception.

Returns

The line that triggers this exception.

Definition at line 353 of file exceptionobject.hpp.

References _line.

```
354     {
355       return _line ;
356     }
```

16.5.3.4 const char * bc2::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.

```
308     {
309       return _location.c_str() ;
310     }
```

16.5.3.5 const char * bc2::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.

```
238     {
239       return "ExceptionObject" ;
240     }
```

16.5.3.6 void bc2::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.

```
280     {
281       _description = s ;
282     }
```

16.5.3.7 void bc2::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.

```
294     {
295       _description = s ;
296     }
```

16.5.3.8 void bc2::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.

```
252     {
253       _location = s ;
254     }
```

16.5.3.9 void bc2::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.

```
266     {
267       _location = s ;
268     }
```

16.5.3.10 const char * bc2::ExceptionObject::what ( ) const throw) [inline], [virtual]

Returns a description of the error causing this exception.

Returns

A description of the error causing this exception.

Definition at line 368 of file exceptionobject.hpp.

```
369     {
370       return _description.c_str() ;
371     }
```

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

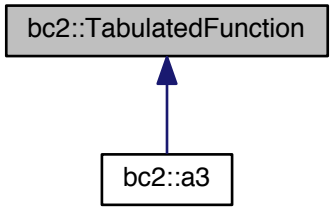
- exceptionobject.hpp

16.6 bc2::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 bc2::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 (unsigned i, 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 (unsigned i, 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 (unsigned i, 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 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 in 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 70 of file tabulatedfunction.hpp.

16.6.2 Member Function Documentation

16.6.2.1 double bc2::TabulatedFunction::a (unsigned *i*, 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 bc2::a3.

16.6.2.2 double bc2::TabulatedFunction::alower (unsigned *i,* 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 bc2::a3.

Referenced by bc2::BuildCurve2D::compute_channel_corners_outside_sleeve_constraints(), and bc2::BuildCurve2D↩ ::compute_sleeve_corners_in_channel_constraints().

16.6.2.3 double bc2::TabulatedFunction::aupper (unsigned *i*, 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 bc2::a3.

Referenced by bc2::BuildCurve2D::compute_channel_corners_outside_sleeve_constraints(), and bc2::BuildCurve2D↩ ::compute_sleeve_corners_in_channel_constraints().

16.6.2.4 unsigned bc2::TabulatedFunction::degree ( ) const [pure virtual]

Returns the degree of tabulated functions.

Returns

The degree of the tabulated functions.

Implemented in bc2::a3.

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

- tabulatedfunction.hpp

Chapter 17

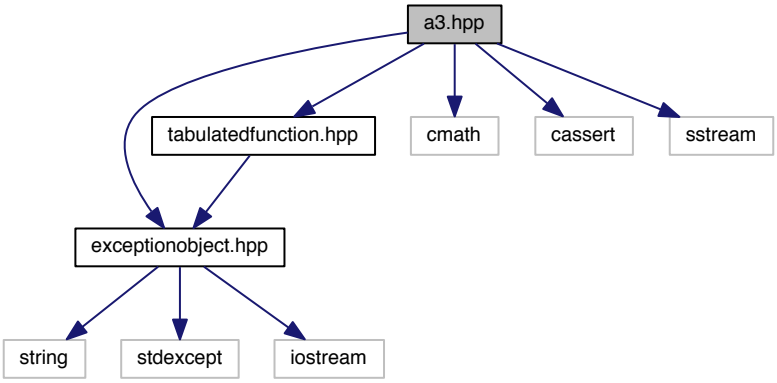
File Documentation

17.1 a3.hpp File Reference

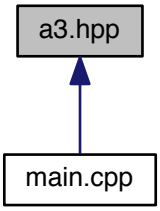
Definition of a class for representing piecewise linear enclosures of certain cubic polynomial functions in Bézier form.

```
#include "exceptionobject.hpp"
#include "tabulatedfunction.hpp"
#include <cmath>
#include <cassert>
#include <sstream>
```

Include dependency graph for a3.hpp:



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



Classes

- class bc2::a3

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

Namespaces

- bc2

  The namespace bc2 contains the definition and implementation of a set of classes for computing a C1 spline cubic curve that passes through 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

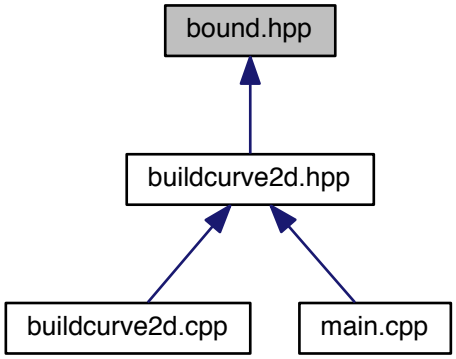
Attention

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.

17.2 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.

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



Classes

- class bc2::Bound

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

Namespaces

- bc2

  The namespace bc2 contains the definition and implementation of a set of classes for computing a C1 spline cubic curve that passes through 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

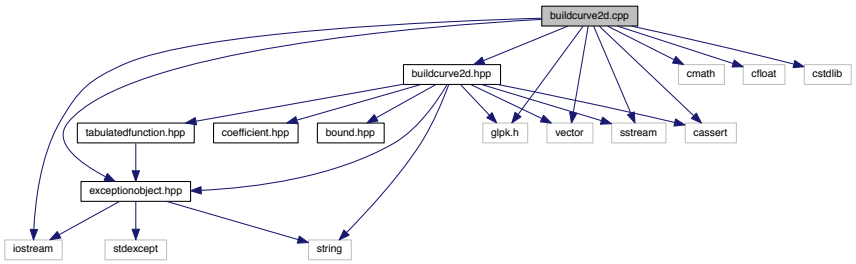
This program is distributed WITHOUT ANY WARRANTY, and it may be freely redistributed under the condition that the copyright notices are not removed, and no compensation is received. Private, research, and institutional use is free. Distribution of this code as part of a commercial system is permissible ONLY BY DIRECT ARRANGEMENT WITH THE AUTHOR.

17.3 buildcurve2d.cpp File Reference

Implementation of a class for threading a C1 spline curve of degree d through a planar channel defined by a pair of polygonal chains.

```
#include "buildcurve2d.hpp"
#include "exceptionobject.hpp"
#include "glpk.h"
#include <cmath>
#include <cassert>
#include <sstream>
#include <iostream>
#include <vector>
#include <cfloat>
#include <cstdlib>
```

Include dependency graph for buildcurve2d.cpp:



Namespaces

- bc2

  The namespace bc2 contains the definition and implementation of a set of classes for computing a C1 spline cubic curve that passes through a given planar channel delimited by two polygonal chains.

17.3.1 Detailed Description

Implementation of a class for threading a C1 spline curve of degree d 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

March 2016

Attention

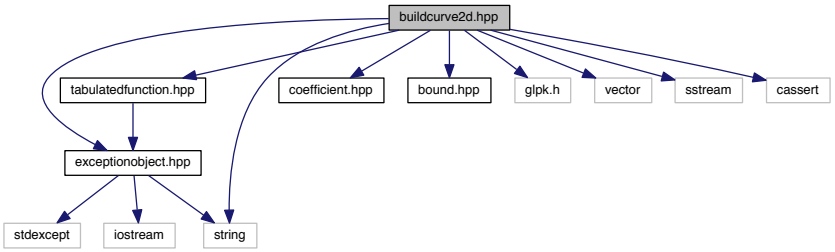
This program is distributed WITHOUT ANY WARRANTY, and it may be freely redistributed under the condition that the copyright notices are not removed, and no compensation is received. Private, research, and institutional use is free. Distribution of this code as part of a commercial system is permissible ONLY BY DIRECT ARRANGEMENT WITH THE AUTHOR.

17.4 buildcurve2d.hpp File Reference

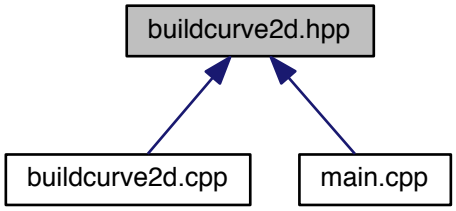
Definition of a class for threading a C1 spline curve of degree d through a planar channel defined by a pair of polygonal chains.

```
#include "exceptionobject.hpp"
#include "tabulatedfunction.hpp"
#include "coefficient.hpp"
#include "bound.hpp"
#include "glpk.h"
#include <vector>
#include <string>
#include <sstream>
#include <cassert>
```

Include dependency graph for buildcurve2d.hpp:



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



Classes

- class bc2::BuildCurve2D

  This class provides methods for threading a C1 spline curve of degree d through a planar channel delimited by a pair of polygonal chain.

Namespaces

- bc2

  The namespace bc2 contains the definition and implementation of a set of classes for computing a C1 spline cubic curve that passes through a given planar channel delimited by two polygonal chains.

17.4.1 Detailed Description

Definition of a class for threading a C1 spline curve of degree d 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

March 2016

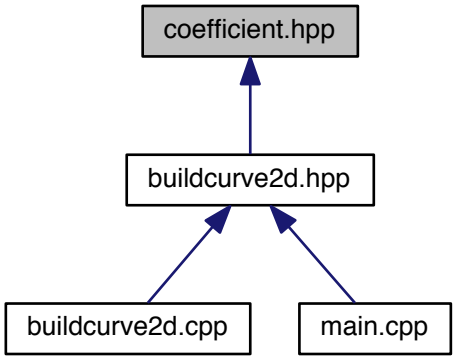
Attention

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.

17.5 coefficient.hpp File Reference

Definition of a class for representing a nonzero coefficient of a variable of a linear constraint (inequality or equality) of an LP.

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



Classes

- class bc2::Coefficient

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

Namespaces

- bc2

  The namespace bc2 contains the definition and implementation of a set of classes for computing a C1 spline cubic curve that passes through a given planar channel delimited by two polygonal chains.

17.5.1 Detailed Description

Definition of a class for representing a nonzero coefficient of a variable of a linear constraint (inequality or equality) of an LP.

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

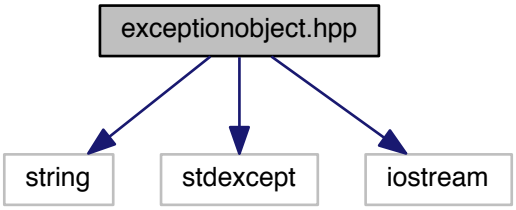
Attention

17.6 exceptionobject.hpp File Reference

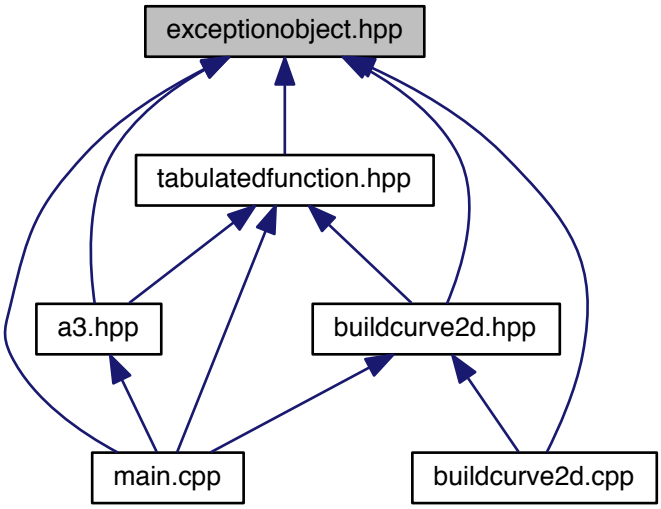
Definition of a class for handling exceptions.

```
#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 bc2::ExceptionObject

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

Namespaces

- bc2

  The namespace bc2 contains the definition and implementation of a set of classes for computing a C1 spline cubic curve that passes through 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.

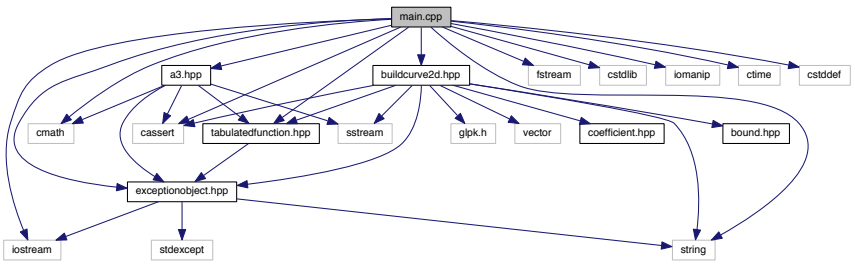
Referenced by bc2::BuildCurve2D::compute_channel_corners_outside_sleeve_constraints(), bc2::BuildCurve2D↩ ::compute_sleeve_corners_in_channel_constraints(), main(), write_lp(), and write_solution().

17.7 main.cpp File Reference

A simple program for testing the bc2d library.

```
#include <iostream>
#include <fstream>
#include <string>
#include <cstdlib>
#include <iomanip>
#include <cassert>
#include <ctime>
#include <cstddef>
#include <cmath>
#include "exceptionobject.hpp"
#include "tabulatedfunction.hpp"
#include "a3.hpp"
#include "buildcurve2d.hpp"
```

Include dependency graph for main.cpp:



Functions

- void read_input (const string &fn, unsigned &np, unsigned &nc, bool &closed, unsigned &dg, 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 BuildCurve2D &b)

  Write the control points of the spline curve threaded into a channel to an output file.
- void write_lp (const string &fn, const BuildCurve2D &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 bc2d 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 135 of file main.cpp.

References bc2::BuildCurve2D::build(), bc2::BuildCurve2D::get_solver_error_message(), read_input(), treat_exception, write_lp(), and write_solution().

```
135                                     {
136   //
137   // Check command-line arguments.
138   //
139
140   if ( ( argc != 3 ) && ( argc != 4 ) ) {
141     cerr << "Usage: "
142          << endl
143          << "\t\t test-bc2d arg1 arg2 [ arg3 ]"
144          << endl
145          << "\t\t arg1: name of the file describing the polygonal channel"
146          << endl
147          << "\t\t arg2: name of the output file describing the computed spline curve"
148          << endl
149          << "\t\t arg3: name of an output file to store a CPLEX format definition of the LP solved by this
      program (OPTIONAL)"
150          << endl
151          << endl ;
152     cerr.flush() ;
153
154     return EXIT_FAILURE ;
155   }
156
157   //
158   // Read in the input file.
159   //
160
161   clock_t start, end ;
162
```

```
163   cerr << endl
164        << "Reading file describing a polygonal channel..."
165        << endl ;
166   cerr.flush() ;
167
168   string fn1( argv[ 1 ] ) ;
169
170   unsigned np ;
171   unsigned nc ;
172   bool closed ;
173   unsigned dg ;
174   double* lx ;
175   double* ly ;
176   double* ux ;
177   double* uy ;
178
179   start = clock() ;
180   try {
181     read_input( fn1 , np , nc , closed , dg , lx , ly , ux , uy ) ;
182   }
183   catch ( const ExceptionObject& xpt ) {
184     treat_exception( xpt ) ;
185     exit( EXIT_FAILURE ) ;
186   }
187   end = clock() ;
188
189   cerr << ( (double) ( end - start ) ) / CLOCKS_PER_SEC
190        << " seconds."
191        << endl
192        << endl ;
193   cerr.flush() ;
194
195   //
196   // Compute a cubic spline curve that passes through the channel.
197   //
198
199   cerr << "Computing a cubic spline curve that passes through the channel... "
200        << endl ;
201   cerr.flush() ;
202
203   start = clock() ;
204   assert( dg == 3 ) ;
205   TabulatedFunction* tf = new a3() ;
206   BuildCurve2D* builder = 0 ;
207   try {
208     builder = new BuildCurve2D(
209                                np ,
210                                nc ,
211                                closed ,
212                                &lx[ 0 ] ,
213                                &ly[ 0 ] ,
214                                &ux[ 0 ] ,
215                                &uy[ 0 ] ,
216                                tf
217                               ) ;
218   }
219   catch ( const ExceptionObject& xpt ) {
220     treat_exception( xpt ) ;
221     exit( EXIT_FAILURE ) ;
222   }
223   int error ;
224   bool res = builder->build( error ) ;
225   end = clock() ;
226
227   cerr << ( (double) ( end - start ) ) / CLOCKS_PER_SEC
228        << " seconds."
229        << endl
230        << endl ;
231   cerr.flush() ;
232
233   if ( res ) {
234     //
235     // Write the control points of all spline pieces to a file.
236     //
237
238     cerr << "Writing out the control points of all spline pieces to a file..."
239               << endl ;
240     cerr.flush() ;
241
242     start = clock() ;
243     string fn2( argv[ 2 ] ) ;
```

```
244     write_solution(
245                    fn2 ,
246                    *builder
247                   ) ;
248     end = clock() ;
249
250     cerr << ( (double) ( end - start ) ) / CLOCKS_PER_SEC
251          << " seconds."
252          << endl
253          << endl ;
254     cerr.flush() ;
255   }
256   else {
257     //
258     // Print the error message returned by the LP solver.
259     //
260     cerr << endl
261          << "ATTENTION: "
262          << endl
263          << builder->get_solver_error_message( error )
264          << endl
265          << endl ;
266   }
267
268   //
269   // Generate a description of the linear program in CPLEX format.
270   //
271   if ( argc == 4 ) {
272     cerr << "Writing out a description of the linear program in CPLEX format..."
273               << endl ;
274     cerr.flush() ;
275
276     start = clock() ;
277     string fn3( argv[ 3 ] ) ;
278     write_lp(
279              fn3 ,
280              *builder
281             ) ;
282     end = clock() ;
283
284     cerr << ( (double) ( end - start ) ) / CLOCKS_PER_SEC
285          << " seconds."
286          << endl
287          << endl ;
288     cerr.flush() ;
289   }
290
291   //
292   // Release memory
293   //
294
295   cerr << "Releasing memory..."
296        << endl ;
297   cerr.flush() ;
298
299   start = clock() ;
300   if ( lx != 0 ) delete[ ] lx ;
301   if ( ly != 0 ) delete[ ] ly ;
302   if ( ux != 0 ) delete[ ] ux ;
303   if ( uy != 0 ) delete[ ] uy ;
304   if ( builder != 0 ) delete builder ;
305   end = clock() ;
306
307   cerr << ( (double) ( end - start ) ) / CLOCKS_PER_SEC
308   << " seconds."
309   << endl
310   << endl ;
311   cerr.flush() ;
312
313   //
314   // Done.
315   //
316
317   cerr << "Finished."
318        << endl
319        << endl
320        << endl ;
321   cerr.flush() ;
322
323   return EXIT_SUCCESS ;
324 }
```

17.7.2.2 void read_input (const string & *fn*, unsigned & *np*, unsigned & *nc*, bool & *closed*, unsigned & *dg*, 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 pieces of the spline to be threaded into the channel.
nc	A reference to the number of c-segments of each c-piece of the channel.
closed	A reference to a flag to indicate whether the channel is closed.
dg	Degree of the spline curve to be threaded into the channel.
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 351 of file main.cpp.

Referenced by main().

```
363 {
364   //
365   // Open the input file
366   //
367   std::ifstream in( fn.c_str() ) ;
368
369   if ( in.is_open() ) {
370     //
371     // Read in the number of polynomial pieces.
372     //
373     in >> np ;
374
375     //
376     // Read in the number of c-segments of each c-piece of the channel.
377     //
378     in >> nc ;
379
380     //
381     // Read in the flag indicating whether the channel is closed.
382     //
383     unsigned flag ;
384     in >> flag ;
385
386     if ( ( flag != 0 ) && ( flag != 1 ) ) {
387       std::stringstream ss( std::stringstream::in | std::stringstream::out ) ;
388       ss << "Flag value indicating whether the channel is closed or open is invalid" ;
389       in.close() ;
390       throw ExceptionObject( __FILE__ , __LINE__ , ss.str().c_str() ) ;
391     }
392
393     closed = ( flag == 1 ) ;
394
395     //
396     // Read in the number of points in each polygonal chain.
397     //
398     unsigned nn ;
399     in >> nn ;
400
401       if (
402           (  closed && ( nn != (   np * nc )       ) )
403           ||
404           ( !closed && ( nn != ( ( np * nc ) + 1 ) ) )
405          )
406     {
407       std::stringstream ss( std::stringstream::in | std::stringstream::out ) ;
408       ss << "Inconsistency among the number of channel breakpoints, curve pieces, and c-segments per
```

```
      c-piece" ;
409       in.close() ;
410       throw ExceptionObject( __FILE__ , __LINE__ , ss.str().c_str() ) ;
411     }
412
413     //
414     // Read in the degree of the spline to be threaded into the channel.
415     //
416     in >> dg ;
417
418     //
419     // Read in  the coordinates of  the points of the  lower polygonal
420     // chain.
421     //
422
423     lx = new double[ nn ] ;
424     ly = new double[ nn ] ;
425
426     for ( unsigned i = 0 ; i < nn ; i++ ) {
427       //
428       // Read in the X and Y coordinates of the i-th point.
429       //
430       in >> lx[ i ] ;
431       in >> ly[ i ] ;
432     }
433
434     //
435     // Read in  the coordinates of  the points of the  upper polygonal
436     // chain.
437     //
438
439     ux = new double[ nn ] ;
440     uy = new double[ nn ] ;
441
442     for ( unsigned i = 0 ; i < nn ; i++ ) {
443       //
444       // Read in the X and Y coordinates of the i-th point.
445       //
446       in >> ux[ i ] ;
447       in >> uy[ i ] ;
448     }
449
450     //
451     // Close file
452     //
453
454     in.close() ;
455   }
456
457   return ;
458 }
```

17.7.2.3 void write_lp (const string & *fn,* const BuildCurve2D & *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 542 of file main.cpp.

References bc2::BuildCurve2D::get_bound_of_ith_constraint(), bc2::BuildCurve2D::get_coefficient_identifier(), bc2::↩ BuildCurve2D::get_coefficient_value(), bc2::BuildCurve2D::get_number_of_coefficients_in_the_ith_constraint(), bc2↩ ::BuildCurve2D::get_number_of_constraints(), bc2::BuildCurve2D::get_number_of_curve_pieces(), bc2::BuildCurve2↩ D::get_spline_degree(), bc2::BuildCurve2D::is_equality(), bc2::BuildCurve2D::is_greater_than_or_equal_to(), bc2::↩ BuildCurve2D::is_less_than_or_equal_to(), and treat_exception.

Referenced by main().

```
546 {
547   //
548   // Create the output file
549   //
550
551   const unsigned np = b.get_number_of_curve_pieces() ;
552   const unsigned dg = b.get_spline_degree() ;
553
554   const unsigned NUMCPC = 2 * ( dg + 1 ) ;
555   const unsigned NUMSDC = 4 * ( dg - 1 ) ;
556   const unsigned NUMCRTLPTS = np * NUMCPC ;
557   const unsigned NUMSECDIFF = np * NUMSDC ;
558
559   std::ofstream ou( fn.c_str() ) ;
560
561   if ( ou.is_open() ) {
562     //
563     // Set the precision of the floating-point numbers.
564     //
565
566     ou << std::setprecision( 6 ) << std::fixed ;
567
568     //
569     // Write the objective function
570     //
571
572     ou << "Minimize"
573        << std::endl ;
574     ou << '\t'
575        << "obj: " ;
576
577     unsigned j = 1 ;
578     for ( unsigned i = 0 ; i < NUMSECDIFF ; i += 4 ) {
579       ou << "-mx" << j << " " ;
580       ou << "-my" << j << " " ;
581       ou << "+px" << j << " " ;
582       ou << "+py" << j << " " ;
583       ++j ;
584     }
585
586     ou << std::endl ;
587
588     //
589     // Write the constraints
590     //
591
592     ou << "Subject To" << std::endl ;
593
594     try {
595
596       for( unsigned i = 0 ; i < b.get_number_of_constraints() ; i++ ) {
597         /*
598          * Write out the number of the constraint.
599          */
600         ou << '\t' << "c" << i + 1 << ": " ;
601
602         /*
603          * Get the coefficients of the i-th constraint.
604          */
605         for( j = 0 ; j < b.get_number_of_coefficients_in_the_ith_constraint
    ( i ) ; ++j ) {
606           /*
607            * Get the column index of the coefficient.
608            */
609           unsigned col = b.get_coefficient_identifier( i , j ) ;
610
611           /*
612            * Get the value of the coefficient.
613            */
614           double value = b.get_coefficient_value( i , j ) ;
615
616           /*
617            * Compute the  index of the  curve piece associated  with the
618            * coefficient, and  find the type of  structural variable the
619            * coefficient is.
620            */
621           unsigned pth = col / ( NUMCPC + NUMSDC ) ;
622           unsigned var = col % ( NUMCPC + NUMSDC ) ;
623           if ( var < NUMCPC ) {
624             unsigned co = pth * ( dg + 1 ) + ( var / 2 ) + 1 ;
625             if ( ( var % 2 ) == 0 ) {
```

```
626               if ( value > 0 ) {
627                 ou << "+" << value << "x" << co << " " ;
628               }
629               else {
630                 ou << value << "x" << co << " " ;
631               }
632             }
633             else {
634               if ( value > 0 ) {
635                 ou << "+" << value << "y" << co << " " ;
636               }
637               else {
638                 ou << value << "y" << co << " " ;
639               }
640             }
641           }
642           else {
643             var -= NUMCPC ;
644             unsigned co = pth * ( dg - 1 ) + ( var / 4 ) + 1 ;
645             unsigned ro = var % 4 ;
646             if ( ro == 0 ) {
647               if ( value > 0 ) {
648                 ou << "+" << value << "mx" << co << " " ;
649               }
650               else {
651                 ou << value << "mx" << co << " " ;
652               }
653             }
654             else if ( ro == 1 ) {
655               if ( value > 0 ) {
656                 ou << "+" << value << "my" << co << " " ;
657               }
658               else {
659                 ou << value << "my" << co << " " ;
660               }
661             }
662             else if ( ro == 2 ) {
663               if ( value > 0 ) {
664                 ou << "+" << value << "px" << co << " " ;
665               }
666               else {
667                 ou << value << "px" << co << " " ;
668               }
669             }
670             else if ( ro == 3 ) {
671               if ( value > 0 ) {
672                 ou << "+" << value << "py" << co << " " ;
673               }
674               else {
675                 ou << value << "py" << co << " " ;
676               }
677             }
678           }
679         }
680
681         if ( b.is_equality( i ) ) {
682           ou << " = " ;
683         }
684         else if ( b.is_less_than_or_equal_to( i ) ) {
685           ou << " <= " ;
686         }
687         else {
688 #ifdef DEBUGMODE
689           assert( b.is_greater_than_or_equal_to( i ) ) ;
690 #endif
691           ou << " >= " ;
692         }
693
694         ou << b.get_bound_of_ith_constraint( i ) << std::endl ;
695       }
696
697     }
698     catch ( const ExceptionObject& xpt ) {
699       treat_exception( xpt ) ;
700       exit( EXIT_FAILURE ) ;
701     }
702
703     //
704     // Write the bounds
705     //
706
```

```
707     ou << "Bounds" << std::endl ;
708
709     for ( unsigned k = 0 ; k < NUMCRTLPTS ; k += 2 ) {
710       unsigned h = ( k >> 1 ) + 1 ;
711       ou << '\t' << "x" << h << " free" << std::endl ;
712       ou << '\t' << "y" << h << " free" << std::endl ;
713     }
714
715     for ( unsigned k = 0 ; k < NUMSECDIFF ; k += 4 ) {
716       unsigned h = ( k >> 2 ) + 1 ;
717       ou << '\t' << "-inf <= mx" << h << " <= 0" << std::endl ;
718       ou << '\t' << "-inf <= my" << h << " <= 0" << std::endl ;
719       ou << '\t' << "0 <= px" << h << " <= +inf" << std::endl ;
720       ou << '\t' << "0 <= py" << h << " <= +inf" << std::endl ;
721     }
722
723     ou << "End" << std::endl ;
724
725     //
726     // Close file
727     //
728
729     ou.close() ;
730   }
731
732   return ;
733 }
```

17.7.2.4 void write_solution (const string & *fn*, const BuildCurve2D & *b*)

Write the control points of the spline curve threaded into a channel to an output file.

Parameters

fn	The name of the output file.
b	An instance of the spline curve builder.

Definition at line 471 of file main.cpp.

References bc2::BuildCurve2D::get_control_value(), bc2::BuildCurve2D::get_number_of_curve_pieces(), bc2::BuildCurve2D::get_spline_degree(), and treat_exception.

Referenced by main().

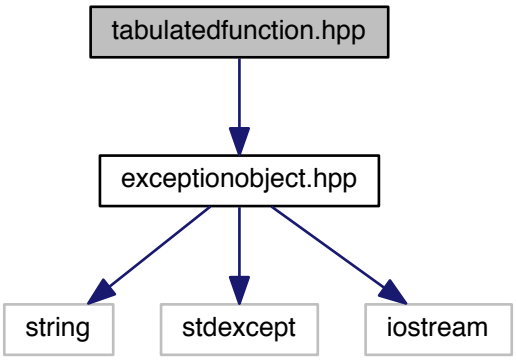
```
475 {
476   using std::endl ;
477
478   std::ofstream ou( fn.c_str() ) ;
479
480   if ( ou.is_open() ) {
481     //
482     // Set the precision of the floating-point numbers.
483     //
484
485     ou << std::setprecision( 6 ) << std::fixed ;
486
487     //
488     // Write the number of curve pieces and the degree of the spline.
489     //
490
491     unsigned np = b.get_number_of_curve_pieces() ;
492     unsigned dg = b.get_spline_degree() ;
493
494     ou << np
495        << '\t'
496        << dg
497        << endl ;
498
499     for ( unsigned p = 0 ; p < np ; p++ ) {
500       for ( unsigned i = 0 ; i <= dg ; i++ ) {
501         double x ;
502         double y ;
503         try {
```

```
504           x = b.get_control_value( p , i , 0 ) ;
505           y = b.get_control_value( p , i , 1 ) ;
506         }
507         catch ( const ExceptionObject& xpt ) {
508           treat_exception( xpt ) ;
509           ou.close() ;
510           exit( EXIT_FAILURE ) ;
511         }
512         ou << x
513            << '\t'
514            << y
515            << endl ;
516       }
517     }
518
519     //
520     // Close file
521     //
522
523     ou.close() ;
524   }
525
526   return ;
527 }
```

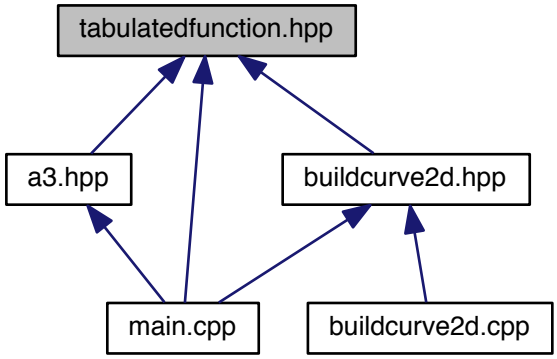
17.8 tabulatedfunction.hpp File Reference

Definition of an abstract class for representing piecewise linear enclosures of certain polynomial functions of arbitrary degree.

`#include "exceptionobject.hpp"`
Include dependency graph for tabulatedfunction.hpp:



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



Classes

- class bc2::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

- bc2

  The namespace bc2 contains the definition and implementation of a set of classes for computing a C1 spline cubic curve that passes through 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

a
    bc2::TabulatedFunction, 99
    bc2::a3, 43
a1
    bc2::a3, 44
a1lower
    bc2::a3, 44
a1upper
    bc2::a3, 44
a3.hpp, 103
alower
    bc2::TabulatedFunction, 99
    bc2::a3, 46
aupper
    bc2::TabulatedFunction, 101
    bc2::a3, 46

bc2, 39
bc2::Bound, 49
    Bound, 50
    get_row, 51
    get_type, 51
    get_value, 51
bc2::BuildCurve2D, 52
    build, 56
    BuildCurve2D, 55, 56
    compute_c0continuity_constraints, 57
    compute_c1continuity_constraints, 58
    compute_channel_corners_outside_sleeve_↩
        constraints, 60
    compute_control_value_column_index, 65
    compute_correspondence_constraints, 66
    compute_min_max_constraints, 67
    compute_normal_to_lower_envelope, 68
    compute_normal_to_upper_envelope, 69
    compute_second_difference_column_index, 69
    compute_sleeve_corners_in_channel_constraints,
        70
    get_bound_of_ith_constraint, 74
    get_coefficient_identifier, 74
    get_coefficient_value, 75
    get_control_value, 75
    get_lower_bound_on_second_difference_value, 76
    get_lp_solver_result_information, 77
    get_number_of_coefficients_in_the_ith_constraint,
        78
    get_number_of_constraints, 78
    get_number_of_curve_pieces, 79
    get_solver_error_message, 79
    get_spline_degree, 80
    get_upper_bound_on_second_difference_value, 80
    h, 81
    is_equality, 82
    is_greater_than_or_equal_to, 82
    is_less_than_or_equal_to, 83
    lf, 83
    minimum_value, 84
    set_up_lp_constraints, 84
    set_up_objective_function, 85
    set_up_structural_variables, 86
    solve_lp, 87
bc2::Coefficient, 88
    Coefficient, 89
    get_col, 90
    get_row, 90
    get_value, 90
bc2::ExceptionObject, 91
    ExceptionObject, 93, 94
    get_description, 94
    get_file, 95
    get_line, 95
    get_location, 95
    get_name_of_class, 95
    set_description, 96
    set_location, 96
    what, 98
bc2::TabulatedFunction, 98
    a, 99
    alower, 99
    aupper, 101
    degree, 101
bc2::a3, 41
    a, 43
    a1, 44
    a1lower, 44
    a1upper, 44
    alower, 46
    aupper, 46
    degree, 48
    h, 48
Bound

bc2::Bound, 50
bound.hpp, 105
build
bc2::BuildCurve2D, 56
BuildCurve2D
bc2::BuildCurve2D, 55, 56
buildcurve2d.cpp, 106
buildcurve2d.hpp, 107

Coefficient
bc2::Coefficient, 89
coefficient.hpp, 109
compute_c0continuity_constraints
bc2::BuildCurve2D, 57
compute_c1continuity_constraints
bc2::BuildCurve2D, 58
compute_channel_corners_outside_sleeve_constraints
bc2::BuildCurve2D, 60
compute_control_value_column_index
bc2::BuildCurve2D, 65
compute_correspondence_constraints
bc2::BuildCurve2D, 66
compute_min_max_constraints
bc2::BuildCurve2D, 67
compute_normal_to_lower_envelope
bc2::BuildCurve2D, 68
compute_normal_to_upper_envelope
bc2::BuildCurve2D, 69
compute_second_difference_column_index
bc2::BuildCurve2D, 69
compute_sleeve_corners_in_channel_constraints
bc2::BuildCurve2D, 70

degree
bc2::TabulatedFunction, 101
bc2::a3, 48

ExceptionObject
bc2::ExceptionObject, 93, 94
exceptionobject.hpp, 110
treat_exception, 112

get_bound_of_ith_constraint
bc2::BuildCurve2D, 74
get_coefficient_identifier
bc2::BuildCurve2D, 74
get_coefficient_value
bc2::BuildCurve2D, 75
get_col
bc2::Coefficient, 90
get_control_value
bc2::BuildCurve2D, 75
get_description
bc2::ExceptionObject, 94
get_file
bc2::ExceptionObject, 95
get_line
bc2::ExceptionObject, 95
get_location
bc2::ExceptionObject, 95
get_lower_bound_on_second_difference_value
bc2::BuildCurve2D, 76
get_lp_solver_result_information
bc2::BuildCurve2D, 77
get_name_of_class
bc2::ExceptionObject, 95
get_number_of_coefficients_in_the_ith_constraint
bc2::BuildCurve2D, 78
get_number_of_constraints
bc2::BuildCurve2D, 78
get_number_of_curve_pieces
bc2::BuildCurve2D, 79
get_row
bc2::Bound, 51
bc2::Coefficient, 90
get_solver_error_message
bc2::BuildCurve2D, 79
get_spline_degree
bc2::BuildCurve2D, 80
get_type
bc2::Bound, 51
get_upper_bound_on_second_difference_value
bc2::BuildCurve2D, 80
get_value
bc2::Bound, 51
bc2::Coefficient, 90

h
bc2::BuildCurve2D, 81
bc2::a3, 48

is_equality
bc2::BuildCurve2D, 82
is_greater_than_or_equal_to
bc2::BuildCurve2D, 82
is_less_than_or_equal_to
bc2::BuildCurve2D, 83

lf
bc2::BuildCurve2D, 83

main
main.cpp, 114
main.cpp, 112
main, 114
read_input, 117
write_lp, 118
write_solution, 121
minimum_value
bc2::BuildCurve2D, 84

Namespace bc2., 37

read_input
    main.cpp, 117

set_description
    bc2::ExceptionObject, 96
set_location
    bc2::ExceptionObject, 96
set_up_lp_constraints
    bc2::BuildCurve2D, 84
set_up_objective_function
    bc2::BuildCurve2D, 85
set_up_structural_variables
    bc2::BuildCurve2D, 86
solve_lp
    bc2::BuildCurve2D, 87

tabulatedfunction.hpp, 122
treat_exception
    exceptionobject.hpp, 112

what
    bc2::ExceptionObject, 98
write_lp
    main.cpp, 118
write_solution
    main.cpp, 121