BSTools
procedure libraries

Przemysław Kiciak

Version 0.35, January, 26, 2015,
TEX-processed December 9, 2019.

Most of this file contains a quick and dirty translation
of the Polish documentation,
which is also quick and dirty.

Caution: This documentation is incomplete and it needs a thorough revision.

Contents

1. Overview

1.1 Introduction

I wrote the BSTools procedure package in order to make experiments being part of my scientific work, and for pleasure. The main part of the package consists of the procedures processing Bézier and B-spline curves and surfaces, hence the name. The mathematical properties of the curves and surfaces and the theoretical bases of the procedures processing these objects are described in my book

Podstawy modelowania krzywych i powierzchni
zastosowania w grafice komputerowej

published by Wydawnictwa Naukowo-Techniczne[1]. The BSTools package (version 0.12) is an appendix to the second edition of this book (of the year 2005). *As opposed to* the book (whose copying, even fragments, *must* be preceded by getting a permision of the publisher), this package *may be* freely copied and distributed, and it may be moodified and used in any programs, on terms of the FSF Lesser GNU Public License, to be read in the file `COPYING.LIB`.

My second book,

Konstrukcje powierzchni gładko wypełniających
wielokątne otwory

published by Oficyna Wydawnicza Politechniki Warszawskiej (prace naukowe, Elektronika, z. 159, 2007) contains a description of the constructions of surfaces of class G^1 and G^2, implemented in the library `libeghole` (the version 0.18 of the package, accompanying the book, contains two libraries, `libg1hole` and `libg2hole`, which have been merged and considerably extended).

The procedures of this package may be used for any (hopefully decent) purpose, for example to write a modelling system or a graphical application. To do this, they must be made robust, i.e. it is necessary to implement and test a full system of error detection and signalling. As it is known, the last person appropriate for making tests of any procedure is its author (but there is no justification for him if he does not do it). People interested in participation in this enterprise and interested in using the package in applications are welcome.

[1] Apart from my book there are also many other books, which describe the algorithms implemented in the procedures described here; to my knowledge, none of those books has been translated yet (before 2005) to Polish.

1.2 Short description of the libraries

The BSTools package currently consists of the following libraries:

`libpkvaria` — varieties, like scratch memory management, sorting etc.

`libpknum` — numerical procedures used in various constructions of B-spline curves, but suitable for general use.

`libpkgeom` — geometric procedures.

`libcamera` — perspective and parallel projections.

`libpsout` — PostScript(TM) picture generation.

`libmultibs` — processing of Bézier and B-spline curves and surfaces.

`libraybez` — ray tracing (computing ray/patch intersections).

`libeghole` — filling polygonal holes in piecewise bicubic spline surfaces with G^1, G^2 and G^1Q^2 continuity.

`libbsmesh` — procedures for processing meshes representing surfaces.

`libg1blending` — procedures of shape optimization of B-spline patches of degree $(2,2)$, of class G^1.

`libg2blending` — procedures of shape optimization of bicubic B-spline patches and mesh surfaces of class G^2.

`libmengerc` — shape optimization by minimization of integral Menger curvature of closed B-spline curves.

`libbsfile` — Reading/writing files with data describing curves and surfaces.

`libxgedit` — support for interaction (using windows and widgets) with an XWindow application (mainly for demonstration programs).

The procedures are written in C, with no hardware or system dependencies, with one exception: the sorting procedure in the `libpkvaria` library assumes the little-endian byte ordering. A migration to a big-endian processor (e.g. Motorola) requires the appropriate reimplementation of this procedure (this has been done, but it has not been tested yet).

1.3 Compilation

The `Makefiles` are written for the Linux system. To compile the package, the documentation and the demos, it is necessary to have

- the GNU make program,
- the gcc compiler and the ar program,
- XWindow and OpenGL libraries (for demonstration programs),
- the TEX system (for the documentation, which uses the LATEX2$_\varepsilon$ package and Concrete Roman and Euler font packages),
- plus Ghostscript and Ghostview programs for convenient browsing of the documentation and the pictures generated by test programs.

To compile the full package, run `make` from the main package directory. This may be preceded by `make clean`, in order to force the compilation of all sources.

The demonstration programs work in the XWindow system. There are no special requirements (like Motif etc.). Some demonstration programs use OpenGL, and the following libraries are needed: `libGL`, `libGLU` and `libGLX`.

1.4 Header files

The header files are stored in the `../include` directory. Each library may have more than one header file, to shorten the compilation time for programs not using all procedures.

`libpkvaria` — the file `pkvaria.h`.

`libpknum` — the files `pknumf.h` and `pknumd.h`, with procedure prototypes of IEEE-754 single precision (`float`) and double precision (`double`) floating-point arithmetic versions respectively. Including the `pknum.h` file causes including both above files, which helps to compile programs using procedures of both precisions.

`libpkgeom` — the files `pkgeomf.h`, `pkgeomd.h` and `pkgeom.h`, which make it possible to use the procedures of the single, double and both precisions.

The convex hull procedures have a separate file `convh.h`, with the prototypes for both precisions.

`libcamera` — the files `cameraf.h`, `camerad.h` and `camera.h` with the descriptions of the cameras, i.e. objects, which implement perspective and parallel projections, in single, double, and both precisions.

The files `stereof.h`, `stereod.h` and `stereo.h` describe pairs of such cameras, which may be used for the generation of stereo pairs of pictures.

`libpsout` — the file `psout.h` contains the prototypes of all procedures in this library.

`libmultibs` — the files `multibsf.h`, `multibsd.h` and `multibs.h` describe procedures of single, double and both precisions.

`libraybez` — the files `raybezf.h` (single precision), `raybezd.h` (double precision) and `raybez.h` (both versions).

`libeghole` — the files `eg1holef.h`, `eg2holef.h` (single precision), `eg1holed.h` and `eg2holed.h` (double precision). There are no header files for both precision versions together.

`libbsmesh` — the file `bsmesh.h`

`libg1blending` — the files `g1blengingf.h` and `g1blendingd.h` (single and double precision respectively).

`libg2blending` — the files `g2blendingf.h`, `g2blendingd.h` and `g2mblendingd.h`. Some procedures have the double precision version only, because of the insufficient range of the single precision numbers.

`libbsfile` — the file `bsfile.h`, the input/output procedures implemented at this point use double precision only.

`libxgedit` — the files `xgedit.h` and `xgledit.h`. Additional files, `xgergb.h` and `xglergb.h` are not supposed to be included directly by applications (they are included by `xgedit.h` and `xgledit.h`). These files contain some colour definitions, with English colour names.

The procedures are compiled as C programs and the header files contain the code, which causes the C++ compiler to see them in this way. Therefore the C++ programs should be linked with these libraries without problems.

1.5 Linking order

The procedures of some libraries refer to procedures of other libraries. To link the program it is necessary to list the libraries in the proper order (otherwise the compiler may fail to resolve some references). The proper order of the libraries is

```
xgedit raybez bsfile g2blending g1blending bsmesh camera
       eghole multibs psout pkgeom pknum pkvaria
```

and it should be preserved in the user-written `Makefiles`. Unused libraries may of course be omitted.

1.6 Principles of modification

The GNU license does not limit the modifications, which one might want to do (but the fact of modifying the software must be notified in the source code, to make clear that it was not the original author, who made the mess). Therefore any principles of making the modifications are only the authors wishes.

1. Except of PostScript file generation and the `libxgedit` library all procedures are totally independent of any environment, in which they might work (and this should be preserved).

2. Making a change in a procedure of the single or double precision, should be accompnied by a similar modification of the other version (if it exists; if not, it is desired to write that other version, using the same algorithm, though for some algorithms the single precision may be insufficient).

3. After each change the documentation should be updated, and a test program should be written. I would be grateful for notifying me about changes, so that I can incorporate them into the future versions of the package.

2. The libpkvaria library

The header file with the prototypes of procedures from the libpkvaria library is pkvaria.h.

2.1 Various small gadgets

```
#define false  0
#define true   1
#define EXP1   2.7182818284590452353
#define PI     3.1415926535897932384
#define SQRT2  1.4142135623730950488
#define SQRT3  1.7320508075688772935

typedef unsigned char boolean;
typedef unsigned char byte;
```

For boolean data it is better to use the name boolean than e.g. unsigned char for its type, and to write true and false instead of 0 and 1. Preserving this rule in the library procedures is however not quite perfect.

It is good to establish some conventions used in programs. It is known, that if one thing may be done in a number of ways, each person will do it in a different way. A team may thus commit a program with procedures, whose parameters are specified in inches and meters. For humans, degrees are more convenient as the angle measure unit. In the program code — radians. I am using the convention, that all angles are processed by the program in radians, and they are input and output in degrees, though e.g. PostScript and OpenGL use a different convention.

```
#define min(a,b) ((a)<(b) ? (a) : (b))
#define max(a,b) ((a)>(b) ? (a) : (b))
```

Two extremely useful macros.

```
double pkv_rpower ( double x, int e );
```

The procedure pkv_rpower computes x^e.

```
void pkv_HexByte ( byte b, char *s );
```

The procedure pkv_HexByte finds the hexadecimal representation of the value of the parameter b. The hexadecimal digits are stored in the array s, whose length must be at least 3.

2.2 Boxes

```
typedef struct Box2i {
    int x0, x1, y0, y1;
  } Box2i;

typedef struct Box2s {
    short x0, x1, y0, y1;
  } Box2s;
```

2.3 Scratch memory management

Many procedures of this package use the scratch memory for storing intermediate results of computations, and the memory blocks are deallocated in the order reverse to that of allocating. The memory management is implemented with use of a stack, which is a very fast and flexible method.

The scratch memory pool serviced by the procedures described below may also be used by other procedures — the only condition is creating a pool large enough at the beginning of the program execution and using this memory in a strictly "stack" manner.

```
char pkv_InitScratchMem ( int size );
```

The procedure pkv_InitScratchMem allocates (with malloc) a memory block of size size bytes and initializes the scratch memory management in this block. The procedure returns 0 if malloc failed and 1 if the memory pool has been successfully initialized.

This procedure must be called before calling any procedures using the scratch memory, i.e. which call pkv_GetScratchMem, pkv_GetScratchMemTop, pkv_FreeScratchMem or pkv_SetScratchMemTop).

Currently there is no extending of the scratch memory pool if it turns out to be too small during the program execution. The programmer therefore has to calculate the size of this pool large enough for doing the computations. Some help in doing this is experimenting and calling the procedure pkv_MaxScratchTaken.

```
void pkv_DestroyScratchMem ( void );
```

The procedure pkv_DestroyScratchMem deallocates (with free) the scratch memory pool created by a call to pkv_InitScratchMem.

```
void *pkv_GetScratchMem ( int size );
```

The procedure mbs_GetScratchMem allocates a memory block of size size bytes in the scratch memory pool and it returns the pointer to this block. If the allocation

is impossible (because the scratch memory pool is too small), the procedure returns the empty pointer (NULL).

```
void pkv_FreeScratchMem ( int size );
```

The procedure pkv_FreeScratchMem deallocates the last size bytes allocated earlier by calls to pkv_GetScratchMem.

The memory blocks deallocation is always done in the order reverse to that of their allocation. It is possible to deallocate a number of memory blocks with a single call to this procedure, by specifying the parameter, whose value is the sum of sizes of the blocks to be deallocated.

```
#define pkv_GetScratchMemi(size) \
  (int*)pkv_GetScratchMem ( (size)*sizeof(int) )
#define pkv_FreeScratchMemi(size) \
  pkv_FreeScratchMem ( (size)*sizeof(int) )
#define pkv_ScratchMemAvaili() \
  (pkv_ScratchMemAvail()/sizeof(int))

#define pkv_GetScratchMemf(size) \
  (float*)pkv_GetScratchMem ( (size)*sizeof(float) )
#define pkv_FreeScratchMemf(size) \
  pkv_FreeScratchMem ( (size)*sizeof(float) )
#define pkv_ScratchMemAvailf() \
  (pkv_ScratchMemAvail()/sizeof(float))

#define pkv_GetScratchMemd(size) \
  (double*)pkv_GetScratchMem ( (size)*sizeof(double) )
#define pkv_FreeScratchMemd(size) \
  pkv_FreeScratchMem ( (size)*sizeof(double) )
#define pkv_ScratchMemAvaild() \
  (pkv_ScratchMemAvail()/sizeof(double))
```

The macros above may be used to allocate memory blocks for arrays of floating point numbers. Using them makes the source code shorter and better readable.

```
void *pkv_GetScratchMemTop ( void );
void pkv_SetScratchMemTop ( void *p );
```

An alternative for remembering the number of allocated bytes (to be deallocated with pkv_FreeScratchMem) is to remember the pointer to the end of the allocated area in the pool. To do this, one may call pkv_GetScratchMemTop and store the value returned in a local variable. Then one or more memory blocks may be allocated with pkv_GetScratchMem). Their deallocation is done by calling pkv_SetScratchMemTop with the pointer given by pkv_GetScratchMemTop.


```
int pkv_ScratchMemAvail ( void );
```

The value of pkv_ScratchMemAvail is the current number of available bytes in the scratch memory pool. An attempt of allocating a greater block will fail — the procedure pkv_GetScratchMem will return NULL.

```
int pkv_MaxScratchTaken ( void );
```

The value of the procedure pkv_MaxScratchTaken is the greatest number of bytes allocated at a certain moment in the scratch memory pool, since the pool was created (with pkv_InitScratchMem), beforre calling this procedure.

2.4 Square angle measure

```
double pkv_SqAngle ( double x, double y );
```

The procedure pkv_SqAngle computes a measure of the angle between the vector $[x, y]$ and the Ox axis. This measure is computed with a couple of arithmetic operations, which is faster than using the cyclometric functions. The function values are in the interval $[0, 4)$.

The properties of this measure: if two vectors make the right angle, then the differences of the values of this measure is 1. Similarly, for the straight angle the difference is 2. The measure of the full angle is 4.

2.5 Data exchanging

```
void pkv_Exchange ( void *x, void *y, int size );
```

The procedure pkv_Exchange swaps the contents of two memory blocks of size size pointed by the parameters x and y. The blocks must be disjoint.

The procedure uses a buffer, whose length is not greater than 1KB, allocated with the pkv_GetScratchMem procedure, therefore to use this procedure it is necessary to create the large enough scratch memory pool (by calling pkv_InitScratchMem at the beginning of the program execution).

```
void pkv_Sort2f ( float *a, float *b );
void pkv_Sort2d ( double *a, double *b );
```

The procedures pkv_Sort2f and pkv_Sort2d swap the values of the variables *a and *b, if the first of them is greater than the second.

2.6 Sorting

2.6.1 CountSort

The procedures described below sort arrays of structures with numerical data (keys), integer or floating point. The sorting method depends on the length of the array. For a small number, InsertionSort is used. For longer arrays the CountSort algorithm is used.

The sorting method is stable, i.e. it does not swap the array elements whose keys have equal values, except that it puts the floating point $+0.0$s after the -0.0s.

```
#define ID_SHORT  0
#define ID_USHORT 1
#define ID_INT    2
#define ID_UINT   3
#define ID_FLOAT  4
#define ID_DOUBLE 5
```

The identifiers above denote the possible types of the keys. The types `short` and `unsigned short` are 16-bit integers. The types `int` and `unsigned int` are 32-bit integers. The types `float` and `double` are 32-bit and 64-bit IEEE-754 floating point numbers. The sorting procedures assume that the byte ordering is *little-endian* (Intel processors use this byte ordering).

```
#define SORT_OK        1
#define SORT_NO_MEMORY 0
#define SORT_BAD_DATA  2
```

The identifiers above denote possible values of the sorting procedures described below. If there is no error, the value returned is `SORT_OK`. The other possibilities indicate not enough scratch memory or invalid data.

```
char pkv_SortKernel ( void *ndata, int item_length, int num_offset,
                      int num_type, int num_data, int *permut );
```

The procedure `pkv_SortKernel` finds the proper sequence of the elements in the `ndata` array, i.e. the permutation, which puts the elements in the sorted (nondecreasing) order. The `ndata` array consists of structures of size `item_length` bytes. The key, i.e. the integer or floating point number with respect to which the data are to be sorted, is located in each structure `num_offset` bytes from the structure beginning. The number n of the structures (i.e. the length of the `ndata` array) is the value of the parametr `num_data`.

The array `permut` contains numbers from 0 to $n-1$. At the return (when there is no error), the array `permut` contains the same numbers, in the order corresponding to the proper permutation. The initial ordering of the numbers in this array is important if the data have to be sorted with respect to a number of keys. For

example, if the structures consist of two numbers, x and y, and the array has to be sorted so that the x-s form a nondecreasing sequence, and for x-s equal the y-s have to form a nondecreasing sequence, one should initialize the permut array by filling it with the numbers $0, \ldots, n-1$ (in an arbitrary order), then call `pkv_SortKernel` twice: first to sort the array with respect to y and then with respect to x. Then one can call `pkv_SortPermute` to set the data in the array in the right order.

```
void pkv_SortPermute ( void *ndata, int item_length, int num_data,
                       int *permut );
```

The procedure `pkv_SortPermute` permutes the structures in the array `ndata` according to the contents of the `permut` array, which has to contain the integer numbers from 0 to $n-1$. The number of structures n is the value of the parameter `num_data`, the length of the array element (in bytes) is specified by the parameter `item_length`.

```
char pkv_SortFast ( void *ndata, int item_length, int num_offset,
                    int num_type, int num_data );
```

The procedure `pkv_SortFast` sorts the array `ndata`, which contains `num_data` structures of size `item_length` bytes, which contain numeric keys of type specified with the parameter `num_type`, located `num_offset` bytes from the structure beginning.

2.6.2 QuickSort

The procedure described below sorts elements of a given sequence by comparisons. It is an implementation of the QuickSort algorithm, and two basic operations on the sequence—comparing and swapping two elements—are done by application-supplied procedures given by parameters. The QuickSort algorithm is not stable, ie. the order of two equal elements after sorting may be changed.

```
void pkv_QuickSort ( int n, boolean (*less)(int,int),
                     void (*swap)(int,int) );
```

Thew parameter `n` specifies the length of the sequence to be sorted; the elements are numbered from 0 to $n-1$. The parameter `less` points to a procedure, whose value `true` indicates that the i-th element of the sequence is less that the j-th element (where the numbers i and j are the values of parameters of the procedure). The parameter `swap` points to a procedure swapping the elements of the sequence indicated by its parameters.

2.6.3 Heap priority queue

The priority queue implemented with the procedures described in this section is an array of pointers to arbitrary objects; both inserting and removing an object is addind or deleting a pointer in the array.The priorities are defined by the application, which ought to supply a procedure, cmp, with two pointer parameters; the procedure must return true if the priority of the object pointed by the first parameter is higher.

```
int pkv_UpHeap ( void *a[], int l, boolean (*cmp)(void*,void*) );
int pkv_DownHeap ( void *a[], int l, int f,
                   boolean (*cmp)(void*,void*) );
int pkv_HeapInsert ( void *a[], int *l, void *newelem,
                     boolean (*cmp)(void*,void*) );
void pkv_HeapRemove ( void *a[], int *l, int el,
                      boolean (*cmp)(void*,void*) );
void pkv_HeapOrder ( void *a[], int n,
                     boolean (*cmp)(void*,void*) );
void pkv_HeapSort ( void *a[], int n,
                    boolean (*cmp)(void*,void*) );
```


2.7 Multidimensional array management

The procedures for processing curves and surfaces in the `libmultibs` library process the control points stored in one-dimensional arrays of floating point numbers. Such an array may be declared for example as n points in the three-dimensional space, but the memory area occupied by these data contains 3n numbers stored one-by-one.

Two-dimensional arrays of points have a similar contents and usually they are used for storing rectangular nets of control points. Such an array contains the coordinates of the points of the first column of the control net, then the second etc. The basic parameter, which makes it possible to access the right places in such an array is the *pitch*, which is the distance between the beginnings of two consecutive columns. Obviously, the pitch is irrelevant for one-dimensional arrays (or arrays with only one column).

From the point of view of the data processing arrays it is better to interpret them as two-dimensional arrays, without taking care of the actual number of points and the dimension of the space, whose elements are these points. The array cosists of *rows* of some fixed length (usually not greater than the pitch). After each row there may be an unused area, whose length is the difference of the pitch and the row length. The procedures and macros described below may be used to change the pitch, by moving closer or farther the rows, which changes the length of the unused areas, to copy the data between two arrays of various pitches, or to move the rows in an array without changing its pitch.

For special purposes it may be necessary to process arrays of bytes in this way (i.e. the smallest directly addressable memory cells). Therefore the C procedures are implemented for the arrays of `char`. The arrays of `float` or `double` are processed with macros, which multiply the row lengths and pitches by the size of `float` or `double`.

```
void pkv_Rearrangec ( int nrows, int rowlen,
                      int inpitch, int outpitch,
                      char *data );
```

The procedure `pkv_Rearrangec` moves the data in the array in order to change the pitch. The array consists of `nrows` rows. Each of them consists of `rowlen` bytes. The parameter `inpitch` specifies the initial pitch (the distance between the beginnings of consecutive rows). The parameter `outpitch` is the target pitch. Both pitches cannot be shorter than the row length.

```
void pkv_Selectc ( int nrows, int rowlen,
                   int inpitch, int outpitch,
                   const char *indata, char *outdata );
```

The procedure `pkv_Selectc` copies the data from the array `indata` to the array `outdata`. The data are stored in `nrows` rows of length `rowlen`. The pitch of the

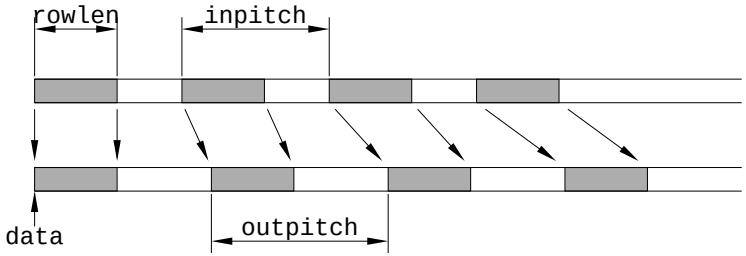


Figure 2.1. The effect of the procedures pkv_Rearrangec and pkv_Selectc

indata array is specified by the parameter inpitch, and the pitch of the outdata by outpitch. The arrays must occupy disjoint memory areas.

The effect of the procedure pkv_Selectc may be illustrated with the same picture (fig. 2.1) as the pkv_Rearrangec procedure, keeping in mind that the data are copied to a *different* array.

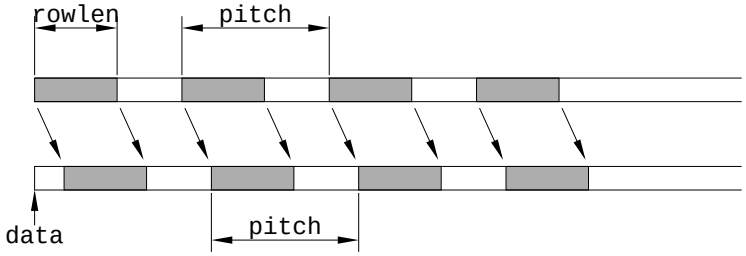


Figure 2.2. Moving data by the procedure pkv_Movec

```
void pkv_Movec ( int nrows, int rowlen,
                 int pitch, int shift, char *data );
```

The procedure pkv_Movec "moves" data in the array by shift bytes. The parameter nrows specifies the number of rows, rowlen is the row length, pitch is the array pitch (which remains unchanged). The parameter data is a pointer to the beginning of the firsy row before moving it. The value of the shift parameter may be positive or negative.

The contents of the array between the rows, if it is not overwritten by the row contents moved by pkv_Movec onto that place, are left unchanged. This makes it possible to "extend" all rows (with shortening the unused area), in order to make space in the rows for additional elements, or to remove some elements from all rows (which extends the unused areas).


```
void pkv_ZeroMatc ( int nrows, int rowlen, int pitch, char *data );
```

The procedure pkv_ZeroMatc initializes an array of bytes, by assigning the value 0 to all its elements. The contents of the unused areas is left unchanged.

```
void pkv_ReverseMatc ( int nrows, int rowlen,
                       int pitch, char *data );
```

The procedure pkv_ReverseMatc puts the rows of an array of bytes in the reverse order. The parameters nrows and rowlen specify the dimensions of this array. The parameter pitch is the pitch of the array data.

```
#define pkv_Rearrangef(nrows,rowlen,inpitch,outpitch,data) \
  pkv_Rearrangec(nrows,(rowlen)*sizeof(float), \
    (inpitch)*sizeof(float),(outpitch)*sizeof(float),(char*)data)
#define pkv_Selectf(nrows,rowlen,inpitch,outpitch,indata,outdata) \
  pkv_Selectc(nrows,(rowlen)*sizeof(float), \
    (inpitch)*sizeof(float),(outpitch)*sizeof(float), \
    (char*)indata,(char*)outdata)
#define pkv_Movef(nrows,rowlen,pitch,shift,data) \
  pkv_Movec(nrows,(rowlen)*sizeof(float),(pitch)*sizeof(float), \
    (shift)*sizeof(float),(char*)data)
#define pkv_ZeroMatf(nrows,rowlen,pitch,data) \
  pkv_ZeroMatc(nrows,(rowlen)*sizeof(float), \
    (pitch)*sizeof(float),(char*)data)
#define pkv_ReverseMatf(nrows,rowlen,pitch,data) \
  pkv_ReverseMatc ( nrows, (rowlen)*sizeof(float),
    (pitch)*sizeof(float), (char*)data )

#define pkv_Rearranged(nrows,rowlen,inpitch,outpitch,data) \
  pkv_Rearrangec(nrows,(rowlen)*sizeof(double), \
    (inpitch)*sizeof(double),(outpitch)*sizeof(double),(char*)data)
#define pkv_Selectd(nrows,rowlen,inpitch,outpitch,indata,outdata) \
  pkv_Selectc(nrows,(rowlen)*sizeof(double), \
    (inpitch)*sizeof(double),(outpitch)*sizeof(double), \
    (char*)indata,(char*)outdata)
#define pkv_ZeroMatd(nrows,rowlen,pitch,data) \
  pkv_ZeroMatc(nrows,(rowlen)*sizeof(double), \
    (pitch)*sizeof(double),(char*)data)
#define pkv_Moved(nrows,rowlen,pitch,shift,data) \
  pkv_Movec(nrows,(rowlen)*sizeof(double),(pitch)*sizeof(double), \
    (shift)*sizeof(double),(char*)data)
#define pkv_ReverseMatd(nrows,rowlen,pitch,data) \
  pkv_ReverseMatc ( nrows, (rowlen)*sizeof(double),
    (pitch)*sizeof(double), (char*)data )
```

The macros above may be used for processing arrays of `float` or `double` in the way described earlier.

The macros `pkv_Rearrangef` and `pkv_Rearranged` change the array pitch.

The macros `pkv_Selectf` and `pkv_Selectd` copy data between arrays of different pitches.

The macros `pkv_Movef` and `pkv_Moved` move the rows in the array.

The macros `pkv_ZeroMatf` and `pkv_ZeroMatd` initiaalize the contents of arrays by setting to 0.0 all the elements (floating point numbers — this is a trick based on the fact that all bits of a floating point 0 are 0).

The macros `pkv_ReverseMatf` and `pkv_ReverseMatd` reverse the order of rows of floating point arrays.

```
void pkv_Selectfd ( int nrows, int rowlen,
                    int inpitch, int outpitch,
                    const float *indata, double *outdata );
void pkv_Selectdf ( int nrows, int rowlen,
                    int inpitch, int outpitch,
                    const double *indata, float *outdata );
```

The above procedures copy data (floating point numbers) between arrays of elements of different precisions. This is similar to the effect of using `pkv_Selectf`, except that a conversion between `float` and `double` is done.

The array pitches (i.e. distances between the beginnings of consecutive rows) are expressed in the units being the lengths of `float` and `double` as appropriate.

The procedure `pkv_Selectfd` should work correctly for all possible data (representing numbers). The other procedure may cause the floating point overflow or underflow. Moreover, there may be rounding errors, which result from the fact that the set of floats is a subset of the set of doubles.

```
void pkv_TransposeMatrixc ( int nrows, int ncols, int elemsize,
                            int inpitch, const char *indata,
                            int outpitch, char *outdata );
```

The procedure `pkv_TransposeMatrixc` makes the transposition of a matrix $m \times n$. The parameters `nrows` and `ncols` specify the numbers m and n respectively. The size (in bytes) of the matrix element is the value of the `elemsize` parameter. The consecutive rows of the input matrix (whose elements are packed one-by-one) are given in the `indata` array, whose pitch (in bytes) is `inpitch`. The procedure writes the consecutive rows of the matrix transposition to the array `outdata`, whose pitch is `outpitch`.


```
#define pkv_TransposeMatrixf(nrows,ncols,inpitch,indata, \
    outpitch,outdata) \
  pkv_TransposeMatrixc ( nrows, ncols, sizeof(float), \
    (inpitch)*sizeof(float), (char*)indata, \
    (outpitch)*sizeof(float), (char*)outdata )
#define pkv_TransposeMatrixd(nrows,ncols,inpitch,indata, \
    outpitch,outdata) ...
```

The two macros above may be used to transpose conveniently numeric matrices, consisting of `float` or `double` floating point numbers. The parameters of those macros correspond to the parameters of the procedure `pkv_TransposeMatrixc` (except of `elemsize`). The pitch unit is the size of `float` or `double`.

2.8 Line segment rasterization

The procedure of line rasterization is placed in this library, because so far there is no better place. In future it is desirable to write a procedure of polygon rasterization, and if this is further developed, then making a separate library with raster graphics routines will be worth doing.

```
typedef struct {
  short x, y;
} xpoint;
```

The structure `xpoint` is intended to represent pixels; it is identical to the `Xpoint` structure defined in the file `Xlib.h`. Due to this, the pixels computed by the line rasterization procedures (and by the curve rasterization procedures of the `libmultibs` library) may be displayed by an XWindow application without further conversion. On the other hand, due to presence of this definition in the `pkvaria.h` file it is not necessary to include `Xlib.h` and it is possible to use this procedure in non-XWindow applications.

```
extern void   (*_pkv_OutputPixels)(const xpoint *buf, int n);
extern xpoint *_pkv_pixbuf;
extern int    _pkv_npix;
```

The variables above are used during the rasterization; these are: pointer to the pixel outputting procedure (which must be supplied by the application), pixel buffer pointer and pixel buffer counter. The application should not refer to these variables directly.

```
#define PKV_BUFSIZE 256
#define PKV_FLUSH ...
#define PKV_PIXEL(p,px) ...
#define PKV_SETPIXEL(xx,yy) ...
```

The macros above define the pixel buffer capacity (256 causes reserving 1KB for that buffer) and implement the buffer servicing. They are made available in the header file for the needs of the curve rasterization procedures from the `libmultibs` library.

```
void _pkv_InitPixelBuffer ( void );
void _pkv_DestroyPixelBuffer ( void );
```

Auxiliary procedures, of which the first allocates the pixel buffer, and the second deallocates it. The buffer is allocated in the scratch memory pool (by a call to `pkv_GetScratchMem`), therefore all the scratch memory allocated after its allocation must be deallocated up to the last byte before the pixel buffer deallocation.


```
void _pkv_DrawLine ( int x1, int y1, int x2, int y2 );
void pkv_DrawLine ( int x1, int y1, int x2, int y2,
                    void (*output)(const xpoint *buf, int n) );
```

The procedure `_pkv_DrawLine` implements the Bresenham algorithm of line segment rasterization. The procedure assumes that the pixel buffer has been allocated prior to the call to it (by `_pkv_InitPixelBuffer`), and the variable `_pkv_OutputPixels` points to the proper pixel output procedure (e.g. drawing the pixels on the screen).

The procedure intendeded to be called by applications is `pkv_DrawLine`, whose parameters: `x1`, `y1`, `x2`, `y2` specify the line segment end points, and the parameter `output` points to the proper pixel output routine. The procedure `pkv_DrawLine` allocates and initializes the pixel buffer and assigns the value of the parameter `output` to the variable `_pkv_OutputPixels`, then it calls `_pkv_DrawLine`, flushes the buffer and deallocates it.

The procedure pointed by `output` must have two parameters; the first is the pointer to the first pixel of the sequence to output, and the second indicates the number of those pixels. The `output` procedure may allocate scratch memory, but it must deallocate it before the return.

2.9 Exception handling

During the program execution there may appear exceptional situations, and the program must be able to deal with them. A typical problem is the memory shortage; if no large enough memory block is available, the program has to do at least one of the following actions:

- Halt (by a call to `exit`); not doing that would cause the program abortion by the operating system because of its improper behaviour, i.e. an attempt to access memory at a random address.
- Inform the user (before halting) about the appearance, place and nature of the exceptional situation. Without that the user will have no idea of the reason of getting a (guten) abend.
- Terminate the computation impossible of the error appearance without halting the program. In that case the user should also be informed, why the program refused to do something, but it is still at the user command with some other services.

In case of exceptional situations the procedures of the bstools package call the `pkv_SignalError` procedure. Its default action is writing (on `stderr`) a message and halting the program. Applications may (by calling the `pkv_SetErrorHandler` procedure) install their own exception handling procedures, which may display messages in a dialog box, and which may (with use of the procedures `setjmp` and `longjmp`, see their description in man pages) abort the unsuccessful computation by terminating a number of unfinished procedures and reset the program to some default state.

```
#define LIB_PKVARIA 0
#define LIB_PKNUM   1
#define LIB_GEOM    2
#define LIB_CAMERA  3
#define LIB_PSOUT   4
#define LIB_MULTIBS 5
#define LIB_RAYBEZ  6
```

The above symbolic names denote the library with the procedure signalling the exception. Applications may define its own identifiers, which should better be different from the above.

```
void pkv_SignalError (
        int module, int errno, const char *errstr );
```

The procedure `pkv_SignalError` by default prints a message to `stderr` and halts the program (with a call to `exit ( 1 );`). The message consists of the error

number (internal for the module, i.e. the library), being the value of the `errno` parameter, the module number (the parameter `module`) and the message text (pointed by the parameter `errstr`).

If an exception handling procedure is installed (with `pkv_SetErrorHandler`), then the procedure `pkv_SignalError` will call it, passing it its parameters.

```
void pkv_SetErrorHandler (
   void (*ehandler)( int module, int errno, const char *errstr ) );
```

The procedure `pkv_SetErrorHandler` installs an exception handling procedure, which henceforth will be called by `pkv_SignalError`. Setting the parameter `ehandler` the `NULL` value causes "uninstalling" any previously installed exception handler, i.e. restoring the default action of the `pkv_SignalError` procedure.

2.10 Wrappings of `malloc` and `free`

Some demonstration programs (`pomnij` and `pozwalaj`) launch a child process (using `fork` and `exec`), whose purpose is to perform the time-consuming numerical computations without locking the user interaction. In particular it is possible to terminate the computations before they are complete. If the child process gets the signal `SIGUSR1`, it must break the computations (this is done with `setjmp` and `longjmp`) and free all memory allocated dynamically in order to prepare for the next job.

The dynamic allocation/deallocation is a critical phase of the computation; `longjmp` is prohibited when `malloc` or `free` is working, and in particular after memory block allocation, but before assignment of its address to a variable. Also an application must register somehow all blocks currently allocated in order to clean up.

To make it possible, `malloc` and `free` may be called via the macros described below. They provide hooks for the application, i.e. pointers to procedures to be called when necessary.

Remark: so far not all procedures in the libraries use these macros.

```
extern boolean pkv_critical, pkv_signal;
extern void (*pkv_signal_handler)( void );
extern void (*pkv_register_memblock)( void *ptr, boolean alloc );
```

The variable `pkv_signal_handler` is `NULL` by default; an application may assign it the address of a signal handler, which will be called at once, except the signall arrived in the critical phase; then this procedure will be called after return from `malloc` or `free`.

A signal handler (registered with the `signal` procedure) should test the value of the variable `pkv_critical`. If it is `true`, then only the assignment `pkv_signal = true;` must be done. If the value of `pkv_critical` is `false`, then the procedure

pointed by pkv_signal_handler may be called, and this procedure is allowed to call longjmp. If after leaving the critical phase pkv_signal is true, the macro calls this procedure (so the signal is processed later, but it is done).

The variable pkv_register_memblock, if not NULL, must point to a procedure, which is called with the address of each block allocated or deallocated via the macro PKV_MALLOC or PKV_FREE (only if pkv_signal_handler is not NULL).

```
#define PKV_MALLOC(ptr,size) \
  { \
    if ( pkv_signal_handler ) { \
      pkv_signal = false; \
      pkv_critical = true; \
      (ptr) = malloc ( size ); \
      if ( pkv_register_memblock ) \
        pkv_register_memblock ( (void*)(ptr), true ); \
      pkv_critical = false; \
      if ( pkv_signal ) \
        pkv_signal_handler (); \
    } \
    else \
      (ptr) = malloc ( size ); \
  }

#define PKV_FREE(ptr) \
  { \
    if ( pkv_signal_handler ) { \
      pkv_signal = false; \
      pkv_critical = true; \
      free ( (void*)(ptr) ); \
      if ( pkv_register_memblock ) \
        pkv_register_memblock ( (void*)(ptr), false ); \
      (ptr) = NULL; \
      pkv_critical = false; \
      if ( pkv_signal ) \
        pkv_signal_handler (); \
    } \
    else { \
      free ( (void*)(ptr) ); \
      (ptr) = NULL; \
    } \
  }
```

The macro PKV_FREE, apart from deallocation of a memory block pointed by the macro parameter (using free), assings NULL to this parameter.

2.11 Debugging

```
void WriteArrayf ( const char *name, int lgt, const float *tab );
void WriteArrayd ( const char *name, int lgt, const double *tab );
```

The two above procedures may be used to produce control printouts during the program debugging. They print out (to stdout) a text name and lgt floating point numbers from the array tab.

```
void *DMalloc ( size_t size );
void DFree ( void *ptr );
```

These procedures may be called instead of malloc and free, if there is a suspicion, that the program writes outside of the allocated memory blocks. The procedure DMalloc allocates (with malloc) a memory block with additional 16 bytes, fills it with zeroes, stores (in the first four bytes) the size and returns the address of the eighth byte of the allocated block.

The procedure DFree verifies, whether the bytes $4, \ldots, 7$ and the last 8 bytes of the block to deallocate are 0 and it writes out a warning.

3. The libpknum library

This library contains procedures with various general numerical algorithms. Currently most of them are related with the linear algebra, but this may change if new needs have to be satisfied.

TODO: In the future it would be desirable to optimize these proceures or to reimplement them as the interface to the BLAS procedures of the LAPACK package. This should be accompanied with the appropriate changes of procedures in the libmultibs library, as many of them contain matrix operations instead of calls to specialized procedures.

3.1 Full matrix operations

The procedures described in this section process matrices represented as ordinary arrays with all coefficients. Sparse matrices, whose most coefficients are 0, may and often should be represented in a different way. In Section 3.2 there are descriptions of procedures for so called band matrices, being some kind of sparse matrices. In Section 3.6 there are procedures of processing sparse matrices, whose nonzero coefficients may be distributed in a completely irregular way.

The rows and columns of an $m \times n$ matrix are indexed from 0 to $m-1$ and from 0 to $n-1$ respectively.

3.1.1 Elementary operations

```
void pkn_AddMatrixf ( int nrows, int rowlen,
                      int inpitch1, const float *indata1,
                      int inpitch2, const float *indata2,
                      int outpitch, float *outdata );
void pkn_SubtractMatrixf ( int nrows, int rowlen,
                           int inpitch1, const float *indata1,
                           int inpitch2, const float *indata2,
                           int outpitch, float *outdata );
void pkn_AddMatrixMf ( int nrows, int rowlen,
                       int inpitch1, const float *indata1,
                       int inpitch2, const float *indata2,
                       double a,
                       int outpitch, float *outdata );
```



```
void pkn_MatrixMDifferencef ( int nrows, int rowlen,
                              int inpitch1, const float *indata1,
                              int inpitch2, const float *indata2,
                              double a,
                              int outpitch, float *outdata );
void pkn_MatrixLinCombf ( int nrows, int rowlen,
                          int inpitch1, const float *indata1,
                          double a,
                          int inpitch2, const float *indata2,
                          double b,
                          int outpitch, float *outdata );
```

The procedures above compute the matrices

$A + B$	pkn_AddMatrixf,
$A - B$	pkn_SubtractMatrixf,
$A + aB$	pkn_AddMatrixMf,
$a(A - B)$	pkn_MatrixMDifferencef,
$aA + bB$	pkn_MatrixLinCombf.

Both given matrices and the result have nrows rows and rowlen columns. The coefficients of A and B are given in the arrays indata1 and indata2. The result is stored in the array outdata. The pitches of the arrays are inpitch1, inpitch2 and outpitch respectively.

Remark: The important property of the array processing procedures is the fact that if there are unused areas between the rows, their contents are unchanged. This property is assumed by various other procedures, which may store some other data in these areas, with a guarantee of not destroying them. For instance, to initialize the zero matrix, only rows sholud be filled with zeros, not the whole array. In addition, it is legal to specify negative pitches, as long as it does not cause reading or writing outside of the area reserved for this purpose.

```
void pkn_MultMatrixNumf ( int nrows, int rowlen,
                          int inpitch, const float *indata,
                          double a,
                          int outpitch, float *outdata );
```

The procedure pkn_MultMatrixNumf computes the product of the matrix A of dimensions $m \times n$ ($m =$ nrows, $n =$ rowlen), and the number a.

The coefficients of the matrix are given in the array indata, with the pitch inpitch, and the result is stored in the array outdata, whose pitch is outpitch. The number a is the value of the parameter a.

If the pitch of the array outdata is greater than the row length, the contents of the unused areas between the rows is left unchanged.

```
void pkn_MultArrayf ( int nrows, int rowlen,
                      int pitch_a, const float *a,
                      int pitch_b, const float *b,
                      int pitch_c, float *c )
```

The procedure pkn_MultArrayf multiplies the coefficients of the matrices A and B, i.e. it computes the numbers $c_{ij} = a_{ij} b_{ij}$. These matrices and the matrix of the products C have the dimensions nrows×rowlen. The pitches of the arrays a, b and c with the coefficients of the matrices A, B and C are equal to pitch_a, pitch_b and pitch_c respectively.

```
void pkn_MultMatrixf ( int nrows_a, int rowlen_a,
                       int pitch_a, const float *a,
                       int rowlen_b, int pitch_b, const float *b,
                       int pitch_c, float *c );
```

The procedure pkn_MultMatrixf multiplies the rectangular matrices, i.e. it computes the product matrix $C = AB$, where $A \in \mathbb{R}^{m,n}$, $B \in \mathbb{R}^{n,l}$, and consequently $C \in \mathbb{R}^{m,l}$.

The parameters nrows_a, rowlen_a and rowlen_b have the values m, n and l respectively. The coefficients of A and B are given in the arrays a and b, with the pitches pitch_a and pitch_b. The parameter pitch_c specifies the pitch of the array c, in which the procedure stores the result.

```
void pkn_MultMatrixAddf ( int nrows_a, int rowlen_a,
                          int pitch_a, const float *a,
                          int rowlen_b, int pitch_b, const float *b,
                          int pitch_c, float *c );
void pkn_MultMatrixSubf ( int nrows_a, int rowlen_a,
                          int pitch_a, const float *a,
                          int rowlen_b, int pitch_b, const float *b,
                          int pitch_c, float *c );
```

The procedure pkn_MultMatrixAddf computes the sum of a matrix and the product of two matrices, i.e. the matrix $D = C + AB$, where $A \in \mathbb{R}^{m,n}$, $B \in \mathbb{R}^{n,l}$, and $C, D \in \mathbb{R}^{m,l}$.

The procedure pkn_MultMatrixSubf computes the matrix $D = C - AB$, for the matrices A, B, C having the dimensions as above.

The values of the parameters nrows_a, rowlen_a and rowlen_b are m, n and l respectively. The coefficients of A and B are given in the arrays a and b, whose pitches are pitch_a and pitch_b. The parameter pitch_c specifies the pitch of the array c, which initially contains the coefficients of C, and the coefficients of D on exit.


```
void pkn_MultTMatrixf ( int nrows_a, int rowlen_a,
                        int pitch_a, const float *a,
                        int rowlen_b, int pitch_b, const float *b,
                        int pitch_c, float *c );
```

The procedure pkn_MultTMatrixf multiplies the rectangular matrices, i.e. it computes the product matrix $C = A^T B$, where $A \in \mathbb{R}^{m,n}$, $B \in \mathbb{R}^{m,l}$, and consequently $C \in \mathbb{R}^{n,l}$.

The parameters nrows_a, rowlen_a and rowlen_b have the values m, n and l respectively. The coefficients of A and B are given in the arrays a and b, with the pitches pitch_a and pitch_b. The parameter pitch_c specifies the pitch of the array c, in which the procedure stores the result.

```
void pkn_MultTMatrixAddf ( int nrows_a, int rowlen_a, int pitch_a,
                           const float *a,
                           int rowlen_b, int pitch_b, const float *b,
                           int pitch_c, float *c );
void pkn_MultTMatrixSubf ( int nrows_a, int rowlen_a, int pitch_a,
                           const float *a,
                           int rowlen_b, int pitch_b, const float *b,
                           int pitch_c, float *c );
```

```
double pkn_ScalarProductf ( int spdimen,
                            const float *a, const float *b );
```

The value of the above procedure is the scalar product of two vectors, a and b in the space $\mathbb{R}^n$. The dimension n is the value of the parameter spdimen.

```
double pkn_SecondNormf ( int spdimen, const float *b );
```

The value of this procedure is the second norm (square root of the sum of squares of the coordinates) of the vector b, in the space $\mathbb{R}^n$ of dimension n =spdimen.

```
double pkn_detf ( int n, float *a );
```

The value of this procedure is the determinant of the matrix A, of dimensions $n \times n$. The parameter n specifies the dimensions of the matrix, whose coefficients are given in the array a (of length n^2; its pitch is n), containing the subsequent rows or columns. The contents of this array is destroyed.

The determinant is evaluated with the Gaussian elimination with full pivoting.

```
void pkn_MVectorSumf ( int m, int n, float *sum, ...  );
void pkn_MVectorLinCombf ( int m, int n, float *sum, ...  );
```

The procedures pkn_MVectorSumf and pkn_MVectorLinCombf compute respectively the sum and linear combination of m vectors in $\mathbb{R}^n$. The parameters m and n specify the numbers m and n, which must be positive. The parametr sum points to

the array in which the result is to be stored. At the call of pkn_MVectorSumf this parameter must be followed by m pointers to the arrays of floats, to be added.

At the call to pkn_MVectorLinCombf the parameter sum must be followed by m pairs of parameters; each pair consists of a pointer (of type float*) and the coefficient of the linear combination of type double.

3.1.2 Solving systems of linear equations

A system of linear equations $A\mathbf{x} = \mathbf{b}$ with a full nonsingular square matrix A may be solved with the Gaussian elimination method; the procedures described in this section implement this algorithm with full pivoting.

```
boolean pkn_GaussDecomposePLUQf ( int n, float *a,
                                  int *P, int *Q );
```

The procedure pkn_GaussDecomposePLUQf computes the factors of decomposition of a square matrix $A = P^{-1}LUQ$ with n rows and columns. These factors are: a permutation matrix P^{-1}, a lower triangular matrix L with diagonal coefficients equal to 1, an upper triangular matrix U and a permutation matrix Q.

The parameter n specifies the matrix dimensions. Its coefficients have to be stored in the array a of length n^2; the array contains consecutive rows. The procedure stores in this array the computed coefficients of the matrices L and U. The permutation matrices P and Q are represented by the numbers stored in the arrays P and Q of length $n - 1$.

The procedure returns true, if the computation is successful, or false, in case the matrix A turned out to be singular.

```
void pkn_multiSolvePLUQf ( int n, const float *lu,
                           const int *P, const int *Q,
                           int spdimen, int pitch, float *b );
```

The procedure pkn_multiSolvePLUQf solves the system of linear equations $AX = B$, where the matrix A with n rows and columns is nonsingular. The matrix B has d columns and n rows.

The matrix A is represented with its decomposition factors found by the procedure pkn_GaussDecomposePLUQf. The parameter n specifies its dmensions. The parameter spdimen specifies the number of columns d of the matrices B and X. The coefficients of B are stored in the array b, whole pitch is pitch. The computed solution is stored in this array.

```
boolean pkn_multiGaussSolveLinEqf ( int n, const float *a,
                                    int spdimen, int pitch, float *b );
```

The procedure pkn_multiGaussSolveLinEqf solves the system of equations $AX = B$ with a nonsingular matrix A with respect to the matrix X. To do this, the

procedure makes a copy of the array a (in order to leave its contents intact) and then it calls the procedures pkn_GaussDecomposePLUQf and pkn_multiSolvePLUQf. The value returned is true if the computation has been successful, or false otherwise. The cause of the failure may be a singular matrix A, or insufficient scratch memory.

As for $d < n$ the most time-consuming part of the algorithm is finding the decomposition of A, if it is necessary to solve a number of systems with the same matrix A and different matrices B, it is better not to use this procedure. Instead, the matrix A should be decomposed once, and then for each matrix B pkn_multiSolvePLUQf may be called.

```
boolean pkn_GaussInvertMatrixf ( int n, float *a );
```

The procedure pkn_GaussInvertMatrixf computes the inverse of a given matrix A of dimensions $n \times n$. It is best not to use it at all.

3.1.3 The QR decomposition and least-squares problems

The procedures described in this section compute the decomposition of a rectangular metrix A into the orthogonal factor Q and the upper triangular factor R, and use this decomposition to solve the linear least squares problem for a system of linear equations $A\mathbf{x} = \mathbf{b}$ with a full matrix A.

The orthogonal matrix Q represents the transformation, which is the composition of a sequence of symmetric reflections with respect to some hyperplanes. The correct method of representing such a matrix is to store the normal vectors of the hyperplanes.

The reflection with respect to a hyperplane is a mapping $\mathbb{R}^m \to \mathbb{R}^m$, whose matrix is given by the formula

$$H_i = I_m - \boldsymbol{w}_i \gamma_i \boldsymbol{w}_i^T, \quad \text{where} \quad \gamma_i = \frac{2}{\boldsymbol{w}_i^T \boldsymbol{w}_i}.$$

I_m is the identity matrix $m \times m$. The reflection hyperplane normal vector is $\boldsymbol{w}_i$. The reflections constructed in order to find the matrix decomposition are called the **Householder reflections**. They are chosen so as to obtain the images of consecutive columns being the columns of a triangular matrix. To spped up solving the least squares problems, apart from the vectors $\boldsymbol{w}_i$ also the numbers γ_i are stored; computing them based on $\boldsymbol{w}_i$ is possible, but it takes time.

```
boolean pkn_QRDecomposeMatrixf ( int nrows, int ncols,
                                 float *a, float *aa );
```

The procedure `pkn_QRDecomposeMatrixf` finds the decomposition of the matrix A, which has `nrows` rows and `ncols` columns, into the factors Q (orthogonal) and R (upper triangular). The coefficients of A have to be stored in the array `a`, of length `nrows`×`ncols`, which contains the subsequent rows of the matrix A.

Upon the return from the procedure, the array `a` contains the representations of these factors. The coefficients of the matrix R on the diagonal and above it are stored in the appropriate places of the array (the coefficient r_{ij} for $i \le j$ replaces the coefficient a_{ij}). The orthogonal matrix Q is represented as a sequence of the normal vectors of the Householder reflection hyperplanes, which transform the matrix A into R. The coordinates of these vectors are stored on the places of the array `a`, initially used to hold the coefficients a_{ij} for $i > j$. The remaining `ncols` coordinates, which do not fit there, and additional `ncols` numbers γ_i are stored in the array `aa`. The way of storing the coefficients of R and of the reflections representation is shown in the figure.

The procedure returns `true` in case of success. Failure, signalled by `false`, occurs when the columns of the matrix A are linearly dependent. The contents of the arrays `a` and `aa` are then undefinite.


```
a = {r00, r01, r02, r03,
     w10,r11, r12, r13,
     w20,w21,r22, r23,
     w30,w31,w32,r33,
     w40,w41,w42,w43,
     w50,w51,w52,w53};
aa = {w00,w11,w22,w33,
      γ0, γ1, γ2, γ3};
```

$$\boldsymbol{w}_0 = \begin{bmatrix} w_{00} \\ w_{10} \\ w_{20} \\ w_{30} \\ w_{40} \\ w_{50} \end{bmatrix}, \boldsymbol{w}_1 = \begin{bmatrix} 0 \\ w_{11} \\ w_{21} \\ w_{31} \\ w_{41} \\ w_{51} \end{bmatrix}, \ldots$$

Figure 3.1. Storing the representation of the matrices Q and R for a matrix 6×4

```
void pkn_multiReflectVectorf ( int nrows, int ncols,
                               const float *a, const float *aa,
                               int spdimen, int pitch, float *b );
```

The procedure `pkn_multiReflectVectorf` computes the product of the matrices Q^{-1} and B; the factor Q is an orthogonal matrix, whose representation computed by the procedure `pkn_QRDecomposeMatrixf` (in the form of a sequence of Householder reflections) is stored in the arrays `a` and `aa`. The matrix B having `ncols` rows and `spdimen` columns is stored in the array `b`. The pitch of the array `b`, i.e. the distance between the first coefficients of consecutive rows, is the value of the parameter `pitch`.

```
void pkn_multiInvReflectVectorf ( int nrows, int ncols,
                                  const float *a, const float *aa,
                                  int spdimen, int pitch, float *b );
```

The procedure `pkn_multiInvReflectVectorf` computes the product of the matrices Q and B; the factor Q is an orthogonal matrix, whose representation has been computed by the procedure `pkn_QRDecomposeMatrixf` (in the form of a sequence of Householder reflections), is stored in the arrays `a` and `aa`. The matrix B having `ncols` rows and `spdimen` columns is stored in the array `b`. The pitch of the array `b`, i.e. the distance between the first coefficients of two consecutive rows, is the value of the parameter `pitch`.

```
void pkn_multiMultUTVectorf ( int nrows, const float *a,
                              int spdimen, int bpitch, float *b,
                              int xpitch, float *x );
```

The procedure `pkn_multiMultUTVectorf` computes the product of the matrices R and B; the factor R is a triangular matrix, whose representation found e.g. by the procedure `pkn_QRDecomposeMatrixf` is stored in the array `a`. The matrix B having `nrows` rows and `spdimen` columns is stored in the array `b`. The pitch of the array `b`, i.e. the distance between the first coefficients of two consecutive rows, is the value of the parameter `bpitch`.

The result of the multiplication is stored in the array x, whose pitch is the vaalue of the parameter xpitch.

```
void pkn_multiMultInvUTVectorf ( int nrows, const float *a,
                                 int spdimen, int bpitch, float *b,
                                 int xpitch, float *x );
```

The procedure pkn_multiMultUTVectorf computes the product of the matrices R^{-1} and B; the matrix R is triangular, and its representation, perhaps computed by the procedure pkn_QRDecomposeMatrixf, is stored in the array a. The matrix B, having nrows rows and spdimen columns is stored in the array b. The pitch of the array b, i.e. the distance between the first coefficients of two consecutive rows, is the value of the parameter bpitch.

The result of the multiplication is stored in the array x, whose pitch is the vaalue of the parameter xpitch.

```
void pkn_multiMultTrUTVectorf ( int nrows, const float *a,
                                int spdimen, int bpitch, float *b,
                                int xpitch, float *x );
void pkn_multiMultInvTrUTVectorf ( int nrows, const float *a,
                                int spdimen, int bpitch, float *b,
                                int xpitch, float *x );
```

```
boolean pkn_multiSolveRLSQf ( int nrows, int ncols, float *a,
                              int spdimen, int bpitch, float *b,
                              int xpitch, float *x );
```

The procedure pkn_multiSolveRLSQf solves a linear least-squares problem for the system of equations $AX = B$, i.e. it decomposes the matrix A (whose columns must be linearly independent) into the factors Q and R, and then it computes the matrix $Y = Q^{-1}B$, and finally $X = R_1^{-1}Y$, where the square matrix R_1 is the upper block of the matrix R.

The numbers of rows and columns of the matrix A are specified by the parameters nrows and ncols. Its coefficients have to be stored in an array a (the consecutive rows must be stored without unused areas between them). The dimensions of the matrix B are nrows rows and spdimen columns. Its coefficients are to be stored in an array b, whose pitch is bpitch.

The result, i.e. the coefficients of the matrix X of dimensions ncols×spdimen are stored by the procedure in the array x, whose pitch is xpitch.

If the computation has been successful, the procedure returns true. Failure, indicated by false, occurs when the problem is nonregular, i.e. when the columns of the matrix A are linearly dependent.


```
void pkn_QRGetReflectionf ( int nrows, int ncols,
                            const float *a, const float *aa,
                            int nrefl, float *w, float *gamma );
```

The procedure pkn_QRGetReflectionf „extracts" the representation of one Householder reflection from the arrays a and aa, in which this representation has been stored by the procedure pkn_QRDecomposeMatrixf.

The parameters nrows and ncols describe the dimensions of the matrix A, whose QR decomposition factors are given in the arrays a and aa. The parametr nrefl, whose value i must be between 0 and ncols-1, specifies the number of the reflection. The coordinates of the reflection hyperplane normal vector $\boldsymbol{w}$ are stored in the array w of length l =nrows-i; they are the last l of nrows coordinates of this vector, and the first i coordinates are 0.

The variable *gamma obtains the value of the parameter γ_i. One may call the procedure with gamma=NULL, and then the parameter gamma is ignored.

3.2 Band matrix processing

3.2.1 The representation and basic procedeures

A band matrix $m \times n$ is a matrix which satisfies the following condition: there exists a number w and two nondecreasing sequences of numbers, $j_0 < \cdots < j_{m-1}$ and $k_0 < \cdots < k_{m-1}$, such that the coefficient a_{ij} (in the i-th row and j-th column) is 0, if $j < j_i$ or $j \geq k_i$, and for all i there is $k_i - j_i \leq w$. The number w is called the *band width* and if it is much smaller than the number of columns n, then representing such a matrix requires significantly less memory. Moreover, many algorithms of processing such matrices are much faster than the algorithms of processing full matrices.

```
typedef struct bandm_profile {
  int firstnz;
  int ind;
} bandm_profile;
```

The parameters, which describe a band matrix are: the numbers of columns and (not always necessary) rows, and two arrays. The first array, `prof`, of length $n + 1$ (greater by 1 than the number of columns) consists of structures of type `bandm_profile`, which describe consecutive columns of the matrix. The second array, `a`, is used for storing the array coefficients, according to the description in the first array.

The value of `prof[j].firstnz` (from 0 to $m-1$) is the index of the row, which contains the first nonzero coefficient of the j-th column. The value of `prof[j].ind` is the index of the array `a`, indicating the position of that coefficient. The consecutive cells of the array `a` hold the consecutive coefficients of this column. The number of consecutive coefficients from this column, which may be nonzero, is equal to `prof[j+1].ind-prof[j].ind`. An example of such a representation is shown in Figure 3.2.

To represent a sequence of reflections with respect to hyperplanes, whose normal vectors are $\boldsymbol{w}_0, \ldots, \boldsymbol{w}_{n-1}$, it is necessary to create the arrays `a` and `prof`, just like these for a band matrix. The array `a` is used to store the numbers γ_i, followed by the nonzero coordinates of the normal vectors $\boldsymbol{w}_i$ (just as if they were columns of a band matrix). The contents of the array `prof` makes it possible to find these coordinates. An example is in Figure 3.3.

The representation described above is intended to save storage space in case of reflections constructed in order to solve a linear least squares problem for a system of equations with a band matrix A. The composition of all reflections in the order of columns is the transformation described by the orthogonal matrix Q^T. By composing these reflections in the reverse order we obtain the matrix Q: the matrix R such that $A = QR$, is upper triangular.

$$\begin{bmatrix} a_0 & 0 & 0 & 0 & 0 \\ a_1 & a_4 & a_{10} & 0 & 0 \\ a_2 & a_5 & a_{11} & 0 & 0 \\ a_3 & a_6 & a_{12} & 0 & 0 \\ 0 & a_7 & a_{13} & 0 & 0 \\ 0 & a_8 & a_{14} & a_{17} & 0 \\ 0 & a_9 & a_{15} & a_{18} & 0 \\ 0 & 0 & a_{16} & a_{19} & 0 \\ 0 & 0 & 0 & a_{20} & a_{23} \\ 0 & 0 & 0 & a_{21} & a_{24} \\ 0 & 0 & 0 & a_{22} & a_{25} \\ 0 & 0 & 0 & 0 & a_{26} \end{bmatrix}$$

```
int ncols = 5;

bandm_profile prof[6] =
{{0,0},{1,4},{1,10},{5,17},{8,23},{*,27}};

float a[27] = { a0,...,a26 };
```

Figure 3.2. A band matrix and the arrays, which represent it

$$\begin{bmatrix} w_5 & 0 & 0 & 0 & 0 \\ w_6 & w_9 & 0 & 0 & 0 \\ w_7 & w_{10} & w_{15} & 0 & 0 \\ w_8 & w_{11} & w_{16} & w_{21} & 0 \\ 0 & w_{12} & w_{17} & w_{22} & w_{29} \\ 0 & w_{13} & w_{18} & w_{23} & w_{30} \\ 0 & w_{14} & w_{19} & w_{24} & w_{31} \\ 0 & 0 & w_{20} & w_{25} & w_{32} \\ 0 & 0 & 0 & w_{26} & w_{33} \\ 0 & 0 & 0 & w_{27} & w_{34} \\ 0 & 0 & 0 & w_{28} & w_{35} \\ 0 & 0 & 0 & 0 & w_{36} \end{bmatrix}$$

$\uparrow \quad \uparrow \quad \uparrow \quad \uparrow \quad \uparrow$

$\boldsymbol{w}_0 \quad \boldsymbol{w}_1 \quad \boldsymbol{w}_2 \quad \boldsymbol{w}_3 \quad \boldsymbol{w}_4$

```
int ncols = 5;

bandm_profile prof[6] =
  {{0,5},{1,9},{2,15},{3,21},
  {4,29},{*,37}};

float a[37] =
  { γ0,...,γ4,w5,...,w36 };
```

Figure 3.3. A representation of a sequence of reflections. The columns of the matrix on the left side are the normal vectors of the reflection hyperplanes

```
void pkn_BandmFindQRMSizes ( int ncols,
                             const bandm_profile *aprof,
                             int *qsize, int *rsize );
```

The procedure `pkn_BandmFindQRMSizes` computes the lengths of the arrays necessary to represent the coefficients of the matrices Q and R, being the factors of the orthogonal-triangular decomposition of a band matrix A. The matrix R will be represented as a band matrix in the „ordinary" way (i.e. the appropriate array will contain its nonzero coefficients, just like the matrix A to be decomposed), and

the matrix Q, which describes the composition of teh Householder reflections, will be represented by the normal vectors of the reflection hyperplanes. Both ways of representing these matrices are described above.

```
void pkn_BandmQRDecomposeMatrixf ( int nrows, int ncols,
                                   const bandm_profile *aprof,
                                   const float *a,
                                   bandm_profile *qprof, float *q,
                                   bandm_profile *rprof, float *r );
```

The procedure pkn_BandmQRDecomposeMatrixf finds the factors of decomposition of a band matrix A, i.e. the orthogonal matrix Q and the uupper triangular matrix R. The matrix A of dimensions nrows×ncols is represented with the array aprof, whose contents describes the positions of the nonzero coefficients in its columns, and the array a, where these coefficients are stored.

The computed upper triangular matrix R is also a band matrix. The procedure stores its representation in the arrays rprof and r. The former array must be of length at least ncols+1. The length of the latter array must be at least equal to that computed by the procedure pkn_BandmFindQRSizes, which should be called first.

The orthogonal matrix Q is the product of the matrices of the Householder reflections, which transform the matrix A to the triangular form. The number of reflections is ncols, therefore the array qprof must be of length at least ncols+1. The length of the array q for storing the coordinates of the reflection hyperplanes normal vectors, must not be less than the appropriate number computed by pkn_BandmFindQRSizes.

Remark: The number of columns, ncols, must be *less* than the number of rows, nrows; square matrices are decomposed with an error (to be fixed some time in the future).

```
void pkn_multiBandmReflectVectorf ( int ncols,
                                    const bandm_profile *qprof,
                                    const float *q,
                                    int spdimen, float *b );
```

The procedure pkn_multiBandmReflectVectorf performs ncols reflections of the columns of a matrix B, having spdimen columns. The consecutive rows of this matrix are stored in the array b, which upon return will hold the result. The order of the reflections is the same that the order of vectors in the array q (i.e. first with respect to $\boldsymbol{w}_0$, then $\boldsymbol{w}_1$ etc.).

The representation of the reflections is given in the arrays qprof and q, as described before.


```
void pkn_multiBandmInvReflectVectorf ( int ncols,
                                       const bandm_profile *qprof,
                                       const float *q,
                                       int spdimen, float *b );
```

The procedure pkn_multiBandmReflectVectorf performs ncols reflections of the columns of a matrix B, which has spdimen columns. The consecutive rows of this matrix are stored in the array b, which upon return will hold the result. The order of the reflections is reverse to the ordering of vectors in the array q (i.e. if $n = \texttt{ncols}$, then the reflection with respect to the hyperplane, whose normal vector is $\boldsymbol{w}_{n-1}$ is done first, then $\boldsymbol{w}_{n-2}$ etc.).

The representation of the reflections is given in the arrays qprof and q, as described before.

```
void pkn_multiBandmMultVectorf ( int nrows, int ncols,
                                 const bandm_profile *aprof,
                                 const float *a,
                                 int spdimen, const float *x,
                                 float *y );
```

The procedure pkn_multiBandmMultVectorf performs the multiplication of a band matrix A of dimensions nrows×ncols, represented with the arrays aprof and a and the matrix X of dimensions ncols×spdimen. The result — the matrix $Y = AX$ of dimensions nrows×spdimen is stored in the array y. The arrays x and y hold consecutive rows of the matrices X and Y.

```
void pkn_multiBandmMultInvUTMVectorf ( int nrows,
                                       const bandm_profile *rprof,
                                       const float *r,
                                       int spdimen, const float *x,
                                       float *y );
```

The procedure pkn_multiBandmMultInvUTMVectorf computes the matrix $Y = A^{-1}X$. The matrix A of dimensions nrows×nrows must be nonsingular upper triangular. The matrix X of dimensions nrows×spdimen is represented with the array x, containing the consecutive rows. The result is stored in the array y.

```
void pkn_multiBandmMultTrVectorf ( int ncols,
                                   const bandm_profile *aprof,
                                   const float *a,
                                   int spdimen, const float *x,
                                   float *y )
```

The procedure pkn_multiBandmMultTrVectorf multiplies the transposition of a band matrix A of dimensions $m \times n$, represented with the arrays aprof and a, and the matrix X of dimensions $n \times d$. The result — the matrix $Y = A^{T}X$ of

dimensions nrows×spdimen — is stored in the array y. The arrays x and y contain the consecutive rows of the matrices X and Y.

The number m is represented by the profile of the matrix A, n is the value of the parameter ncols and d is the value of spdimen.

```
void pkn_multiBandmMultInvTrUTMVectorf ( int nrows,
                                         const bandm_profile *rprof,
                                         const float *r,
                                         int spdimen, const float *x,
                                         float *y )
```

The procedure pkn_multiBandmMultInvTrUTMVectorf computes the matrix $Y = A^{-T}X$. The matrix A of dimensions nrows×nrows must be nonsingular upper triangular. The matrix X of dimensions nrows×spdimen is represented with the array x, containing the consecutive rows. The result is stored in the array y.

3.2.2 Solving linear least squares problems

An example of using the procedures described above to solve a regular linear least squares problem $A\mathbf{x} = \mathbf{b}$, with a column-regular band matrix A:

1. Create a representation of the matrix A.
2. Call pkn_BandmFindQRMSizes and then allocate arrays, whose lengths are computed by this procedure, for the representations of the arrays Q and R, being the decomposition factors of A.
3. Call pkn_BandmQRDecomposeMatrixf to find the decomposition of the matrix A.
4. Compute the vector $\mathbf{y} = Q^T\mathbf{b}$ by calling pkn_multiBandmReflectVectorf.
5. Compute $\mathbf{x} = R_1^{-1}\mathbf{y}_1$, where the matrix R_1 is the block $n \times n$, consisting of the initial rows of R and the vector $\mathbf{y}_1$ consists of the first n coordinates of $\mathbf{y}$. To do this, call pkn_multiBandmMultInvUTMVectorf.


```
void pkn_multiBandmSolveRLSQf ( int nrows, int ncols,
                                const bandm_profile *aprof,
                                const float *a,
                                int nrsides, int spdimen,
                                int bpitch, const float *b,
                                int xpitch, float *x );
```

The procedure pkn_multiBandmSolveRLSQf solves in the way described above z linear least squares problems, posed by the system of equations

$$A[\mathbf{x}_0, \ldots, \mathbf{x}_{z-1}] = [\mathbf{b}_0, \ldots, \mathbf{b}_{z-1}].$$

The band matrix A of dimensions $m \times n$ (given by the parameters nrows i ncols) is represented with use of the arrays aprof and a. The array b, whose length is bpitch×z, describes the right-hand sides of the systems of equations, i.e. z matrices $\mathbf{b}_0, \ldots, \mathbf{b}_{z-1}$, each of dimensions $m \times d$, whose consecutive rows are stored in the array without gaps; there are z =nrsides such matrices in the array, and the positions of the first coefficients of two consecutive matrices differ by bpitch. Each of the d columns of each matrix is one right-hand side vector of the system (thus in fact the procedure solves zd = nrsides×spdimen least squares problems with the same matrix A and a number of right-hand side vectors).

The solutions are the columns of the matrix $\mathbf{x}$, whose coefficients (in consecutive rows) are stored in the array x. This array must have length at least xpitch×z. The parameter xpitch specifies the distance between the first coefficients of consecutive matrices $\mathbf{x}_i$ in the array x.

3.2.3 Solving regular problems with constraints

A regular linear least squares problem with constraints is to find the vector $\mathbf{x}$, which satisfies the system of equations

$$C\mathbf{x} = \mathbf{d},$$

called the constraints equations, such that the vector $\mathbf{r} = A\mathbf{x} - \mathbf{b}$ has the smallest second norm, assuming that the matrix $A \in \mathbb{R}^{m,n}$ is columnwise-regular and the matrix $C \in \mathbb{R}^{w,n}$ is rowwise regular.

The linear independence of the rows of C implies the consistency of the constraints equations and such a problem has a unique solution. It may be found by solving the following system:

$$\begin{bmatrix} A & C^T \\ C & 0 \end{bmatrix} \begin{bmatrix} \mathbf{x} \\ \mathbf{y} \end{bmatrix} = \begin{bmatrix} \mathbf{b} \\ \mathbf{d} \end{bmatrix}.$$

A numerical method appropriate for doing this follows:

1. Decompose the matrix A to the factors Q and R, such that Q is an orthogonal matrix and R is upper triangular.

2. Solve the system of equations $R^T E = C^T$.
3. Decompose the matrix E to the factors U and F, such that U is an orthogonal matrix and F is upper triangular. By F_1 denote the $w \times w$ matrix, which is the block of F consisting of its w initial rows.
4. Using the factors QR of A compute the solution $\mathbf{x}_0$ of the regular least squares problem, by solving the system $R_1\mathbf{x}_0 = \mathbf{y}_1$ (see Section 3.2.2).
5. Solve the systems of equations $F_1^T\mathbf{e} = \mathbf{d} - C\mathbf{x}_0$ and $F_1\mathbf{f} = \mathbf{e}$.
6. Solve the systems $R_1^T\mathbf{g} = C^T\mathbf{f}$ and $R_1\mathbf{h} = \mathbf{g}$.
7. Compute $\mathbf{x} = \mathbf{x}_0 + \mathbf{h}$.

```
void pkn_multiBandmSolveCRLSQf ( int nrows, int ncols,
                      const bandm_profile *aprof, const float *a,
                      int nconstr, int cpitch, const float *c,
                      int nrsides, int spdimen,
                      int bpitch, const float *b,
                      int dpitch, const float *d,
                      int xpitch, float *x );
```

The procedure `pkn_multiBandmSolveCRLSQf` solves z regular least squares problems with constraints, for the system of linear equations

$$A[\mathbf{x}_0, \ldots, \mathbf{x}_{z-1}] = [\mathbf{b}_0, \ldots, \mathbf{b}_{z-1}],$$

with a band matrix A where the constraints are described by the system

$$C[\mathbf{x}_0, \ldots, \mathbf{x}_{z-1}] = [\mathbf{d}_0, \ldots, \mathbf{d}_{z-1}],$$

with a full matrix C.

The parameters: `nrows`, `ncols` — numbers of rows m and columns n of the matrix A, `aprof`, `a` — profile (i.e. the representation of the positions of nonzero coefficients) and the array with the nonzero coefficients of A, `nconstr` — number w of constraints (must be less than n), `cpitch` — pitch (i.e. distance of the beginnings of consecutive rows) of the array `c` with the coefficients of C, `nrsides` — number z, `spdimen` — length d of rows of the matrices $\mathbf{b}_i$ (and also $\mathbf{x}_i$ and $\mathbf{d}_i$), `bpitch` — pitch of the array `b` with the coefficients of the matrices $\mathbf{b}_0, \ldots, \mathbf{b}_{z-1}$ (the distance between the beginnings of the consecutive matrices; the rows of each matrix are stored without gaps), `dpitch` — pitch of the array `d` with the coefficients of the matrices $\mathbf{d}_0, \ldots, \mathbf{d}_{z-1}$, `xpitch` — pitch of the array `x` to store the result.

The parameter `d` may be `NULL` — then the constraint equations are homogeneous.

3.2.4 Solving dual linear least squares problems

A dual linear least-squares problem is finding the solution $\mathbf{x}$ of a system of equations $A\mathbf{x} = \mathbf{b}$ with a matrix $A \in \mathbb{R}^{m,n}$ row-regular, such that for a given vector $\mathbf{x}_0 \in \mathbb{R}^n$ the number $\|\mathbf{x} - \mathbf{x}_0\|_2$ is minimal. The procedures described earlier may be used to solve such a problem, if the program creates a band representation of the matrix A^T.

1. Create a band representation of the matrix A^T.
2. Call `pkn_BandmFindQRSizes` and allocate arrays of appropriate lengths for storing the representations of the factors Q and R of the decomposition of A^T.
3. Call `pkn_BandmQRDecomposeMatrixf` to find the decomposition of A^T (which is equivalent to decomposing A into the factors R^T and Q^T).
4. Compute the vector $\mathbf{z}_0 = Q^T\mathbf{x}_0$, by calling `pkn_multiBandmReflectVectorf`. If $\mathbf{x}_0 = \mathbf{0}$, then it is possible instead to set $\mathbf{z}_0 = \mathbf{0}$ (without any computation).
5. Call `pkn_multiBandmMultInvTrUTMVectorf` in order to solve the system $R_1^T\mathbf{z}_1 = \mathbf{b}$. If $\mathbf{b} = \mathbf{0}$, then it is possible instead to set $\mathbf{z}_1 = \mathbf{0}$ (without any computation). Compute the vector $\mathbf{z}$, whose first m coordinates are the corresponding coordinates of $\mathbf{z}_1$, and the other coordinates are the coordinates of $\mathbf{z}_0$.
6. Call `pkn_multiBandmInvReflectVectorf` to compute the solution, i.e. the vector $\mathbf{x} = Q\mathbf{z}$.

```
void pkn_multiBandmSolveDLSQf ( int nrows, int ncols,
                                const bandm_profile *atprof,
                                const float *at,
                                int nrsides, int spdimen,
                                int bpitch, const float *b,
                                int x0pitch, const float *x0,
                                int xpitch, float *x );
```

The procedure `pkn_multiBandmSolveDLSQf` solves dual linear least squares problems in the way described above. The parameters `nrows` (the number of rows, n), `ncols` (the number of columns, m), `atprof` (the profile) and `at` (the array of coefficients) describe the matrix A^T.

There are z`d` =`nrsides`$\times$`d` right-hand sides of the system, i.e. the matrices $\mathbf{b}$, with m rows and `d` =`spdimen` columns; each column is the right-hand side of one problem (thus the procedure solves `d` problems with the matrix A). The consecutive rows of $\mathbf{b}$ must be stored in the array `b`. The positions of the first coefficients of two consecutive matrices $\mathbf{b}$ differ by `bpitch`. The parameter `b` may also be `NULL`, which means that the right-hand sides of the systems are the zero vector.

The approximations of the solutions are the columns of z matrices $\mathbf{x}_0$ of dimensions $n \times d$. The consecutive rows of these matrices must be given in the array x0, the positions of the first coefficients of two consecutive matrices $\mathbf{x}_0$ differ by x0pitch. If the parameter x0 is NULL, then the matrices $\mathbf{x}_0$ are assumed to be zero.

The solutions are the columns of the matrices $\mathbf{x}$. The consecutive rows of these matrices are stored in the array x, whose length must be at least z×xpitch.

3.2.5 Debugging

The procedures described in this section print out into stdout matrices in the text form. These procedures are sometimes quite helpful in detecting bugs.

```
void pkn_PrintMatf ( int nrows, int ncols, const float *a );
```

The procedure pkn_PrintMatf prints the coefficients of a full matrix A, represented explicitly in an array.

The parameters nrows and ncols specify the numbers of rows and columns respectively. The array a contains the coefficients of A, row by row.

```
void pkn_PrintBandmf ( int ncols, const bandm_profile *aprof,
                       const float *a );
```

The procedure pkn_PrintBandmf prints a band matrix represented by the arrays aprof and a.

```
void pkn_PrintBandmRowSumf ( int ncols, const bandm_profile *aprof,
                       const float *a );
```

The procedure pkn_PrintBandmRowSumf prints a band matrix represented by the arrays aprof and a. The sum of coefficients for each row is written at the end of the row.

```
void pkn_PrintProfile ( int ncols, const bandm_profile *prof );
```

The procedure pkn_PrintProfile prints out the contents of the array prof, i.e. the profile of a band matrix.

3.3 Processing "packed" symmetric and triangular matrices

A square symmetric matrix $n \times n$ may be represented with $\frac{1}{2}(n+1)n$ numbers, i.e. almost twice less than a general matrix of the same dimensions. Also, triangular matrices (lower and upper) may be represented without storing the zero coefficients above or below the diagonal. The procedures described in this section process matrices represented in such a space-saving way.

$$A = \begin{bmatrix} a_{00} & a_{10} & a_{20} & a_{30} \\ a_{10} & a_{11} & a_{21} & a_{31} \\ a_{20} & a_{21} & a_{22} & a_{32} \\ a_{30} & a_{31} & a_{32} & a_{33} \end{bmatrix}$$

```
int n = 4;
float a[] = {a00,a10,a11,a20,a21,a22,
             a30,a31,a32,a33};
```

Figure 3.4. The space saving representation of a symmetric matrix

$$L = \begin{bmatrix} l_{00} & 0 & 0 & 0 \\ l_{10} & l_{11} & 0 & 0 \\ l_{20} & l_{21} & l_{22} & 0 \\ l_{30} & l_{31} & l_{32} & l_{33} \end{bmatrix} \qquad L^T = \begin{bmatrix} l_{00} & l_{10} & l_{20} & l_{30} \\ 0 & l_{11} & l_{21} & l_{31} \\ 0 & 0 & l_{22} & l_{32} \\ 0 & 0 & 0 & l_{33} \end{bmatrix}$$

```
int n = 4;
float l[] = {l00,l10,l11,l20,l21,l22,l30,l31,l32,l33};
```

Figure 3.5. The space saving representations of triangular matrices

```
#define pkn_SymMatIndex(i,j) \
  ( (i) >= (j) ? (i)*((i)+1)/2+(j) : (j)*((j)+1)/2+(i) )
```

The macro `pkn_SymMatIndex` computes the index of the coefficient a_{ij} of a symmetric matrix A in the array used to store the coefficients. This is also the index of the coefficient l_{ij} of a lower triangular matrix L, provided that $i \geq j$ (otherwise $l_{ij} = 0$).

```
boolean pkn_CholeskyDecompf ( int n, float *a );
```

The procedure `pkn_CholeskyDecompf` decomposes a symmetric positive-definite matrix A (i.e. such that $A^T = A$ and $\forall_{x \neq 0} x^T A x > 0$) into triangular factors: $A = LL^T$. The coefficients of the lower triangular matrix L are stored in the array `a`, initially occupied by the coefficients of the matrix A.

The parameter `n` specifies the dimensions of the matrices. The procedure returns `true`, if the decomposition has been computed successfully, and `false`, if during the computations the matrix A turned out not to be positive-definite. In that case the contents of the array `a` is indefinite.


```
void pkn_SymMatrixMultf ( int n, const float *a, int spdimen,
                          int bpitch, const float *b,
                          int xpitch, float *x );
void pkn_LowerTrMatrixMultf ( int n, const float *l, int spdimen,
                              int bpitch, const float *b,
                              int xpitch, float *x );
void pkn_UpperTrMatrixMultf ( int n, const float *l, int spdimen,
                              int bpitch, const float *b,
                              int xpitch, float *x );
```

The procedures `pkn_SymMatrixMultf`, `pkn_LowerTrMatrixMultf` and `pkn_UpperTrMatrixMultf` compute respectively the product X of a symmetric, lower triangular and upper triangular matrix A of dimensions $n \times n$ and the matrix B of dimensions $n \times d$, whose representation is „full".

The parameters `n` and `spdimen` specify the dimensions n and d of the matrices. The parameter `a` or `l` points to an array with the coefficients of the matrix A, stored in the space saving way. The parameter `b` is a pointer to an array with the coefficients of B, whose pitch is `bpitch`. The parameters `x` and `xpitch` are the pointer to an array for the result and the pitch of this array respectively.

Remark: There is no specific procedure of multiplication of symmetric matrices, because in general the product of symmetric matrices does not have to be symmetric. Also, there are no procedures of multiplication of two lower or upper triangular matrices, because so far I did not need them. If necessary, one can convert the matrices to the full representation (with the procedures described later) and use the procedure of multiplication of general matrices.

```
void pkn_LowerTrMatrixSolvef ( int n, const float *l, int spdimen,
                               int bpitch, const float *b,
                               int xpitch, float *x );
void pkn_UpperTrMatrixSolvef ( int n, const float *l, int spdimen,
                               int bpitch, const float *b,
                               int xpitch, float *x );
```

The procedures `pkn_LowerTrMatrixSolvef` and `pkn_UpperTrMatrixSolvef` solve systems of linear equations with a lower triangular matrix L and an upper triangular matrix L^T, represented in the space saving way. This is equivalent to multiplying the right-hand side matrix B by the matrix L^{-1} or L^{-T}.

The parameters `n` and `spdimen` specify the dimensions of the matrices L: $n \times n$ and B and X: $n \times d$. The arrays `l` and `b` contain the coefficients of L and B. The procedures store the results in the array `x`. The parameters `bpitch` and `xpitch` specify the pitches of the arrays `b` and `x`.

One can pass the same array as both parameters: `b` and `x`; in this case the result will replace the right-hand side matrix, but then both parameters, `bpitch`

and xpitch must have the same value. If the arrays are different, then the contents of b remains unchanged.

To solve a system of linear equations $A\mathbf{x} = \mathbf{b}$ with a symmetric positive-definite matrix A, one can call the procedure pkn_CholeskyDecompf, which computes the matrix L such that $A = LL^T$, and then solve the system $L\mathbf{y} = \mathbf{b}$ with pkn_LowerTrMatrixSolvef and $L^T\mathbf{x} = \mathbf{y}$ with pkn_UpperTrMatrixSolvef. This is a faster method than using an algorithm appropriate for general matrices (like the Gaussian elimination or Householder reflections).

```
void pkn_SymToFullMatrixf ( int n, const float *syma,
                            int pitch, float *fulla );
void pkn_FullToSymMatrixf ( int n, int pitch, const float *fulla,
                            float *syma );
#define pkn_FullToLTrMatrixf(n,pitch,fulla,ltra) \
  pkn_FullToSymMatrixf(n,pitch,fulla,ltra)
void pkn_LTrToFullMatrixf ( int n, const float *ltra,
                            int pitch, float *fulla );
void pkn_UTrToFullMatrixf ( int n, const float *utra,
                            int pitch, float *fulla );
void pkn_FullToUTrMatrixf ( int n, int pitch, const float *fulla,
                            float *utra );
```

The procedures and the macro above make the conversion between the space saving and full representations of symmetric and triangular matrices.

```
void pkn_ComputeQSQTf ( int m, const float *s,
                        int n, const float *a, const float *aa,
                        float *b );
void pkn_ComputeQTSQf ( int m, const float *s,
                        int n, const float *a, const float *aa,
                        float *b );
```

The procedures pkn_ComputeQSQTf and pkn_ComputeQTSQf compute respectively the products of the matrices

$$QSQ^T \quad \text{and} \quad Q^TSQ,$$

where S is a symmetric matrix $m \times m$, represented in the packed form, and the matrix Q is orthogonal $m \times m$. The matrix Q represents the composition of n Householder reflections, obtained by orthogonal-triangular decomposition of a matrix A, having dimensions $m \times n$, as described in Section 3.1.3.

The parameters m and n describe the dimensions of the matrices S and A. The array s contains the coefficients of the matrix S. The arrays a and aa contain the representations of the reflections (i.e. of the matrix Q), as described in Section 3.1.3. The coefficients of the product, which is a symmetric matrix, are stored in in the

array b. One may call the procedure with b=s; then the coefficients of S will be replaced in the array by the coefficients of the product.

The procedures implement the Ortega-Householder algorithm; for each subsequent reflection, represented by the matrix $H_i = I_m - \mathbf{v}_i\beta_i\mathbf{v}_i^T$, where $i = 0, \ldots, n-1$, the procedures compute

$$\begin{aligned}
&\text{the vector } \mathbf{u} = B_{i-1}\mathbf{v}_i\beta_i,\\
&\text{the wector } \mathbf{p} = \mathbf{u} - \mathbf{v}_i\mathbf{v}_i^T\mathbf{u}\beta_i/2,\\
&\text{the matrix } B_i = B_{i-1} - (\mathbf{v}_i\mathbf{p}^T + \mathbf{p}\mathbf{v}_i^T),
\end{aligned}$$

where $B_0 = S$ and $\mathbf{v}_i = \mathbf{w}_i$, $\beta_i = \gamma_i$ for the procedure pkn_ComputeQTSQf, and $\mathbf{v}_i = \mathbf{w}_{n-i-1}$, $\beta_i = \gamma_{n-i-1}$ for the procedure pkn_ComputeQSQTf. The final result is the matrix B_{n-1}.

```
void pkn_MatrixLowerTrMultf ( int m, int n, int bpitch,
          const float *b, const float *l, int xpitch, float *x );
void pkn_MatrixUpperTrMultf ( int m, int n, int bpitch,
          const float *b, const float *l, int xpitch, float *x );
void pkn_MatrixLowerTrSolvef ( int m, int n, int bpitch,
          const float *b, const float *l, int xpitch, float *x );
void pkn_MatrixUpperTrSolvef ( int m, int n, int bpitch,
          const float *b, const float *l, int xpitch, float *x );
void pkn_MatrixLowerTrMultAddf ( int m, int n, int bpitch,
          const float *b, const float *l, int xpitch, float *x );
void pkn_MatrixUpperTrMultAddf ( int m, int n, int bpitch,
          const float *b, const float *l, int xpitch, float *x );
boolean pkn_MatrixLowerTrSolveAddf ( int m, int n, int bpitch,
          const float *b, const float *l, int xpitch, float *x );
boolean pkn_MatrixUpperTrSolveAddf ( int m, int n, int bpitch,
          const float *b, const float *l, int xpitch, float *x );
void pkn_MatrixLowerTrMultSubf ( int m, int n, int bpitch,
          const float *b, const float *l, int xpitch, float *x );
void pkn_MatrixUpperTrMultSubf ( int m, int n, int bpitch,
          const float *b, const float *l, int xpitch, float *x );
boolean pkn_MatrixLowerTrSolveSubf ( int m, int n, int bpitch,
          const float *b, const float *l, int xpitch, float *x );
boolean pkn_MatrixUpperTrSolveSubf ( int m, int n, int bpitch,
          const float *b, const float *l, int xpitch, float *x );
```

```
void pkn_SymMatSubAATf ( int n, float *b, int m, int pitch_a,
                         const float *a );
```

3.4 Processing symmetric and triangular matrices with a nonregular band

Matrices $n \times n$, symmetric or triangular with a nonregular band are represented with two arrays: the profile and the coefficient array. The profile is an array of n integers; its i-th element is the index of the first nonzero coefficient of the i-th row (rows and columns are numbered from 0). Examples are shown in Figures 3.6 and 3.7.

$$\begin{bmatrix} a_{00} & a_{10} & & & & \\ a_{10} & a_{11} & a_{21} & & a_{41} & \\ & a_{21} & a_{22} & a_{32} & a_{42} & \\ & & a_{32} & a_{33} & a_{43} & a_{53} \\ & a_{41} & a_{42} & a_{43} & a_{44} & a_{54} \\ & & & a_{53} & a_{54} & a_{55} \end{bmatrix}$$

```
int n = 6;
float a[] = {a00,a10,a11,a21,a22,a32,
    a33,a41,a42,a43,a44,a53,a54,a55};
int prof[] = {0,0,1,2,1,3};
```

Figure 3.6. Representtion of a symmetric matrix with a nonregular band

$$\begin{bmatrix} l_{00} & & & & & \\ l_{10} & l_{11} & & & & \\ & l_{21} & l_{22} & & & \\ & & l_{32} & l_{33} & & \\ & l_{41} & l_{42} & l_{43} & l_{44} & \\ & & & l_{53} & l_{54} & l_{55} \end{bmatrix} \quad \begin{bmatrix} l_{00} & l_{10} & & & & \\ & l_{11} & l_{21} & & l_{41} & \\ & & l_{22} & l_{32} & l_{42} & \\ & & & l_{33} & l_{43} & l_{53} \\ & & & & l_{44} & l_{54} \\ & & & & & l_{55} \end{bmatrix}$$

```
int n = 6;
float l[] = {l00,l10,l11,l21,l22,l32,l33,l41,l42,l43,l44,l53,l54,l55};
int prof[] = {0,0,1,2,1,3};
```

Figure 3.7. Representation of triangular matrices with nonregular bands

```
int pkn_NRBArraySize ( int n, const int *prof );
```

The procedure `pkn_NRBArraySize` computes the length of the array for storing the matrix coefficients, based on the profile.

```
boolean pkn_NRBFindRowsf ( int n, const int *prof, const float *a,
                           float **row );
```

The procedure `pkn_NRBFindRowsf` stores in the array `row` of length n pointers to virtual rows; the access to the coefficient a_{ij} is given by the expression `row[i][j]`, assuming that the following condition is satisfied: `prof[i]` $\leq j \leq i$.

The procedures described below have the parameter `row`, which may be `NULL`; then they call `pkn_NRBFindRowsf`. One can also create the array `row` once, call

`pkn_NRBFindRowsf`, and then pass this array as the parameter to these procedures. This saves some time (and may be useful also during the computation of the coefficients of the matrix in the application).

```
boolean pkn_NRBSymCholeskyDecompf ( int n, const int *prof,
                                    float *a, float **row );
```

The procedure `pkn_NRBCholeskyDecompf` decomposes a symmetric, positive-definite matrix A into triangular factors L and L^T, using the Cholesky's method. The coefficients of the matrix L are stored in the array `a`, where they replace the coefficients of the given matrix A. The profile of both matrices, A and L are the same.

```
boolean pkn_NRBSymMultf ( int n, const int *prof,
                          const float *a, const float **row,
                          int spdimen, int xpitch, const float *x,
                          int ypitch, float *y );
boolean pkn_NRBLowerTrMultf ( int n, const int *prof,
                          const float *a, const float **row,
                          int spdimen, int xpitch, const float *x,
                          int ypitch, float *y );
boolean pkn_NRBUpperTrMultf ( int n, const int *prof,
                          const float *a, const float **row,
                          int spdimen, int xpitch, const float *x,
                          int ypitch, float *y );
```

The procedures `pkn_NRBSymMultf`, `pkn_NRBLowerTrMultf` and `pkn_NRBUpperTrMultf` compute respectively the product of a symmetric, lower triangulad or upper triangular matrix with a nonregular band, and of the full matrix X.

```
boolean pkn_NRBLowerTrSolvef ( int n, const int *prof,
                          const float *l, const float **row,
                          int spdimen, int bpitch, const float *b,
                          int xpitch, float *x );
boolean pkn_NRBUpperTrSolvef ( int n, const int *prof,
                          const float *l, const float **row,
                          int spdimen, int bpitch, const float *b,
                          int xpitch, float *x );
```

The procedures `pkn_NRBLowerTrSolvef` and `pkn_NRBUpperTrSolvef` solve respectively a system of linear equations with a lower or upper triangular matrix with a nonregular band.

```
boolean pkn_NRBSymFindEigenvalueIntervalf ( int n, const int *prof,
                                            float *a, float **row,
                                            float *amin, float *amax );
```

The procedure `pkn_NRBSymFindEigenvalueIntervalf` finds, based on the Gershgorin theorem, an interval containing all eigenvalues of a symmetric matrix with irregular band.

The parameters `n`, `prof`, `a` and `row` represent the matrix.

The parameters `amin`, `amax` point to the variables, to which the procedure has to assign the interval limits.

Return value `true` signals a success, `false` — a failure, which may be caused by insufficient scratch memory, when the parameter `row` is `NULL` and the procedure must itself construct the array of pointers to virtual rows based on the profile (an error may then be detected by the procedure `pkn_NRBFindRowsf`).

```
boolean pkn_NRBComputeQTSQf ( int n, int *prof, float *Amat,
                              float **Arows,
                              int w, float *Bmat, float *bb,
                              int *qaprof, float **QArows );
boolean pkn_NRBComputeQSQTf ( int n, int *prof, float *Amat,
                              float **Arows,
                              int w, float *Bmat, float *bb,
                              int *qaprof, float **QArows );
```

The input data for the above procedures are: a symmetric $n \times n$ matrix S with irregular band and an orthogonal matrix Q, represented by a sequence of w Householder reflections of $\mathbb{R}^n$ (where $w < n$). The matrix Q may be obtained by a QR decomposition of an $n \times w$ matrix B, using the procedure `pkn_QRDecomposeMatrixf`.

The procedure `pkn_NRBComputeQTSQf` has to compute the matrix $C = Q^T S Q$.

The procedure `pkn_NRBComputeQSQTf` has to compute the matrix $D = Q S Q^T$.

In both cases the result is represented as a symmetric matrix with irregular band.

Input parameters: `n`, `prof`, `Amat`, `Arows` — representation of the matrix S.
Caution: currently the parameter `Arows` must not be `NULL`, it must point to an array of `n` pointers to virtual rows of the matrix S.

The parameters `n`, `w`, `Bmat`, `bb` represent the matrix Q in the way described in Section 3.1.3. The number w is the number of reflections the columns of the matrix in the array `Bmat` contain the coordinates of the normal vectors of reflection hyperplanes $\boldsymbol{w}_i$ (except for initial zeros and the first nonzero coordinate), the array `bb` contains the first nonzero coordinate of each vector $\boldsymbol{w}_i$ and the numbers γ_i.

Output parameters: `qaprof` — pointer to an array of length `n`, in which the profile of the array C or D will be stored (this array has to be allocated by the caller), `QArows` — pointer to an array of length `n`, to hold the pointers of virtual rows of the computed matrix. Its coefficients are stored in an array allocated by

`malloc`; the address of the beginning of this array (to be passed to `free` when the time comes) is the address of the first virtual row.

The return value `true` signals a success and `false` signals a failure, which may be caused by insufficient memory in the scratch pool or in the heap processed by `malloc` and `free`.

Remark: in practical applications more useful may be the procedures `pkn_NRBComputeQTSQblf` and `pkn_NRBComputeQSQTblf` described below. They do the same task, but the result is conveniently divided into separate blocks.

```
boolean pkn_NRBComputeQTSQblf ( int n, int *prof, float *Amat,
                                float **Arows,
                                int w, float *Bmat, float *bb,
                                int *qa11prof, float **QA11rows,
                                int *qa22prof, float **QA22rows,
                                float **QA21 );
boolean pkn_NRBComputeQSQTblf ( int n, int *prof, float *Amat,
                                float **Arows,
                                int w, float *Bmat, float *bb,
                                int *qa11prof, float **QA11rows,
                                int *qa22prof, float **QA22rows,
                                float **QA21 );
```

The input data for the above procedures are: a symmetric $n \times n$ matrix S with irregular band and an orthogonal matrix Q, represented by a sequence of w Householder reflections of $\mathbb{R}^n$ (where $w < n$). The matrix Q may be obtained by a QR decomposition of an $n \times w$ matrix B, using the procedure `pkn_QRDecomposeMatrixf`.

The procedure `pkn_NRBComputeQTSQblf` has to compute blocks of the matrix $C = Q^T S Q$.

The procedure `pkn_NRBComputeQSQTblf` has to compute blocks of the matrix $D = Q S Q^T$.

The result, e.g. the matrix C, has the block structure

$$C = \begin{bmatrix} C_{11} & C_{21}^T \\ C_{21} & C_{22} \end{bmatrix}.$$

The blocks C_{11} and C_{22}, of dimensions $w \times w$ and $n - w \times n - w$ respectively, are symmetric matrices represented with irregular band. The block C_{21} of dimensions $n - w \times w$ is represented as a full matrix.

Input parameters are identical as these of the procedures described before: `n`, `prof`, `Amat`, `Arows` — representation of the matrix S.
Caution: currently the parameter `Arows` must not be `NULL`, it must point to an array of `n` pointers to virtual rows of the matrix S.

The parameters `n`, `w`, `Bmat`, `bb` represent the matrix Q in the way described in Section 3.1.3. The number w is the number of reflections the columns of the matrix

in the array `Bmat` contain the coordinates of the normal vectors of reflection hyperplanes $\boldsymbol{w}_i$ (except for initial zeros and the first nonzero coordinate), the array `bb` contains the first nonzero coordinate of each vector $\boldsymbol{w}_i$ and the numbers γ_i.

Output parameters: `qa11prof` and `qa22prof` — pointers to arrays of lengths w and $n-w$ respectively, in which the profiles of the matrices C_{11} and C_{22} will be stored (these arrays have to be allocated by the caller) `QA11rows` and `QA22rows`— pointers to the arrays of lengths w and $n-w$, in which pointers to virtual rows of the matrices C_{11} and C_{22} will be stored. The parameterr `QA21` points to the variable, to which the address of the first coefficient of the block C_{12} will be assigned (this is a full matrix, stored row by row, without gaps).

All coefficients of the result matrix are stored in a memory block allocated with `malloc`; the address of the beginning of this block (to be passed to `free` when necessary) is the address of the beginning of the first virtual row of C_{11}.

The return value `true` signals a success and `false` signals a failure, which may be caused by insufficient memory in the scratch pool or in the heap processed by `malloc` and `free`.

3.5 Processing block symmetric matrices

3.5.1 Matrices of the first type block structure

The procedures of filling polygonal holes in the library `libg2hole` need to solve systems of linear equations with symmetric positive-definite matrices having a block structure — with zero blocks apart from the diagonal and the last row and column. An example is shown in Figure 3.8

$$\begin{bmatrix} A_{00} & & & A_{30}^T \\ & A_{11} & & A_{31}^T \\ & & A_{22} & A_{32}^T \\ A_{30} & A_{31} & A_{32} & A_{33} \end{bmatrix} \qquad \begin{bmatrix} L_{00} & & & \\ & L_{11} & & \\ & & L_{22} & \\ L_{30} & L_{31} & L_{32} & L_{33} \end{bmatrix}$$

Figure 3.8. Structure of a block symmetric matrix and a lower triangular block matrix

The structure of such a matrix is described by three numbers. The first (k) is the number of diagonal blocks except for the last one, the second (r) specifies the dimensions of those blocks and the third number (s) specifies the dimensions of the last diagonal block. The matrix has therefore $kr+s$ rows and columns.

The coefficients of such a matrix are stored in an array `A`. The diagonal blocks are represented in the „packed" form, discussed in the previous section, the subdiagonal blocks are stored as full matrices. The length of the array `A` must be at least $kr(r+1)/2+s(s+1)/2+krs$.

To do most computations the procedures described below call the procedures of processing full matrices and packed symmetric matrices described in the preceding sections.

```
boolean pkn_Block1CholeskyDecompMf ( int k, int r, int s,
                                     float *A );
```

The procedure `pkn_Block1CholeskyDecompMf` finds the decomposition of the block matrix A into triangular factors L and L^T. The coefficients of the lower triangular matrix L are stored in the array `A`, where they replace the coefficients of the matrix A. This is possible, because the matrix L has zero blocks where the matrix A has the zero blocks.

The procedeure returns `true` after a successful computation, and `false` otherwise. Failure may be caused by a non-positive definite matrix A or by an ill-conditioned matrix, for which rounding errors may produce a nonpositive diagonal coefficient.

```
void pkn_Block1LowerTrMSolvef ( int k, int r, int s,
             const float *A,
             int spdimen, int xpitch, float *x );
void pkn_Block1UpperTrMSolvef ( int k, int r, int s,
             const float *A,
             int spdimen, int xpitch, float *x );
```

The procedures above solve the systems of equations $Lx = b$ and $L^T x = b$ respectively. The right-hand side and the solution are matrices $n \times d$ (where $n = kr+s$). The procedures replace the coefficients of the right-hand side in the array x, whose pitch is xpitch, by the coefficients of the solution.

To solve a system of linear equations with a symmetric positive definite block matrix A, one should decompose it into triangular factors (using the procedure pkn_Block1CholeskyDecompMf), and then call the above two procedures.

```
void pkn_Block1SymMatrixMultf ( int k, int r, int s,
                                float *A,
                                int spdimen, int xpitch, float *x,
                                int ypitch, float *y );
```

The procedure pkn_Block1SymMatrixMultf multiplies the matrices, i.e. it computes the product $y = Ax$, where A is a symmetric block matrix $n \times n$ (where $n = kr + s$), and the matrix x (and y) is full, of dimensions $n \times d$. The array x contains the coefficients of the matrix x. The array y is the place, where the result is stored. The pitches of the two arrays (i.e. distances between the first coefficients of consecutive rows) are equal to xpitch and ypitch respectively. The parameter spdimen specifies d.

3.5.2 Matrices of the second block type structure

The block structure of the third type processed by the libpknum library is shown in Figure 3.9. Such matrices are symmetric and they consist of $2k + 1 \times 2k + 1$ blocks, where $k \geq 3$.

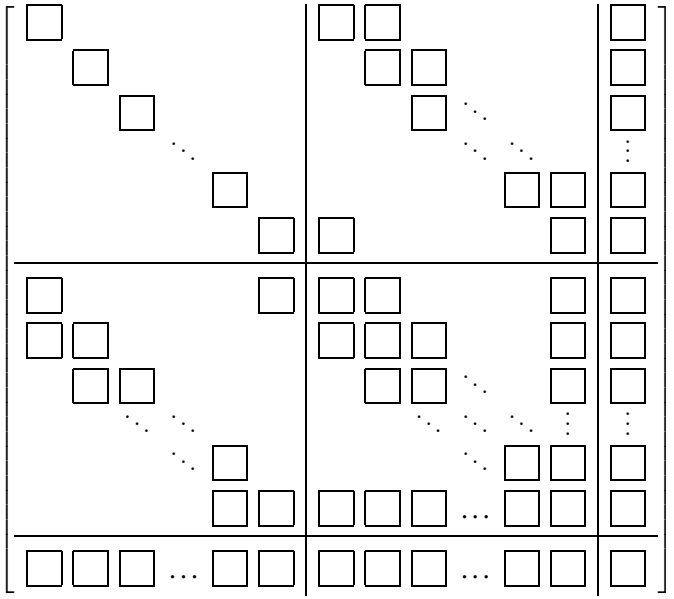


Figure 3.9. Second type block matrix structure for a symmetric matrix

Nonzero blocks are placed as shown, and the rows and columns are numbered from 0 to 2k:

- The blocks $A_{00}, \ldots, A_{k-1,k-1}$ are $r \times r$.
- The blocks $A_{kk}, \ldots, A_{2k-1,2k-1}$ are $s \times s$.
- The block $A_{2k,2k}$ is $t \times t$.

If the matrix A is positive-definite, then the lower triangular matrix L, such that $LL^T = A$, has zero blocks corresponding to the zero blocks of the matrix A.

The whole matrix has dimensions $k(r + s) + t \times k(r + s) + t$. To store the coefficients of its nonzero blocks one needs

- $k \cdot \frac{1}{2}(r + 1)r$ cells for the diagonal blocks $A_{00}, \ldots, A_{k-1,k-1}$,
- $k \cdot \frac{1}{2}(s + 1)s$ cells for the diagonal blocks $A_{kk}, \ldots, A_{2k-1,2k-1}$,

- $\frac{1}{2}(t+1)t$ cells for the diagonal block $A_{2k,2k}$,
- $2k \cdot rs$ cells for the blocks $A_{k,k-1}, A_{k,0}, A_{k+1,k}, A_{k+1,k+1}, \ldots, A_{2k-1,2k-2}$, $A_{2k-1,2k-1}$,
- $(2k-3) \cdot s^2$ cells for the blocks $A_{k+1,k}, \ldots, A_{2k-2,2k-3}$ and $A_{2k-1,k}, \ldots, A_{2k-1,2k-2}$.
- $k \cdot (r+s)t$ cells for the blocks $A_{2k,0}, \ldots, A_{2k,2k-1}$.

This fits in an array of length

$$k\Big(\frac{1}{2}(r+1)r + \frac{1}{2}(s+1)s + (r+s)(t+2s)\Big) + \frac{1}{2}(t+1)t - 3s^2.$$

Computing block positions in the array

Having two indices $i, j \in \{0, \ldots, 2k\}$, where $i \geq j$, one has to compute the position of the first element of the block A_{ij} in the array.

1. If $i = j < k$, then $p = i\frac{1}{2}(r+1)r$.
2. If $k \leq i = j < 2k$, then $p = k\frac{1}{2}(r+1)r + i\frac{1}{2}(s+1)s$.
3. If $i = j = 2k$, then $p = \frac{1}{2}k\big((r+1)r + (s+1)s\big)$.
4. Let $N_1 = \frac{1}{2}\big(k(r+1)r + k(s+1)s + (t+1)t\big)$.
   If $k \leq i < 2k$, $0 \leq j < k$ and $i - j \bmod k \in \{0, 1\}$, then
   $p = N_1 + \big(2(i-k) + 1 - (i-j) \bmod k\big)rs$.
5. Let $N_2 = N_1 + 2krs$.
   If $k < i < 2k-1$ and $j = i-1$, then $p = N_2 + (i-k-1)s^2$.
6. If $i = 2k-1$ and $k \leq j < 2k-2$, then $p = N2 + (j-2)s^2$.
7. Let $N_3 = N_2 + (2k-3)s^2$.
   If $i = 2k$ and $j < k$, then $p = N_3 + jrt$.
8. If $i = 2k$ and $k \leq j < 2k$, then $p = N_3 + krt + (j-k)st$.
9. Else the block A_{ij} is a zero block, whose coefficients are not stored.

For diagonal blocks only the lower triangle is stored in the packed form. The subdiagonal blocks are stored rowwise, like other full matrices.

Triangular decomposition of a symmetric matrix A

If the matrix L is lower-triangular and it consists of the blocks $L_{i,j}$ (i.e. for $i < j$ the block $L_{i,j}$ is zero), then the matrix $A = LL^T$ consists of the blocks

$$A_{i,j} = \sum_{l=0}^{j} L_{i,l} L_{l,j}^T.$$

The blocks of L may be found using the following algorithm:
Consecutively for $i = 0, \ldots, 2k$ compute (with the Cholesky's method) the lower triangular block $L_{i,i}$, such that $L_{i,i}L_{i,i}^T = A_{i,i} - \sum_{l=0}^{i-1} L_{i,l}L_{i,l}^T$, and then for $j = i+1, \ldots, 2k$ compute the block $L_{j,i} = (A_{j,i} - \sum_{l=0}^{i-1} L_{j,l}L_{i,l}^T)L_{i,i}^{-T}$.

For a matrix A having block structure discussed above, one may compute

1. For $i = 0, \ldots, k-1$ the matrix $L_{i,i}$ such that $L_{i,i}L_{i,i}^T = A_{i,i}$,
   and then $L_{j,i} = A_{j,i}L_{i,i}^{-T}$, where $j \in \{i+k, i+(k+1) \bmod k, 2k\}$.
2. The matrix $L_{k,k}$, such that $L_{k,k}L_{k,k}^T = A_{k,k} - L_{k,0}L_{k,0}^T - L_{k,k-1}L_{k,k-1}^T$,
   and then $L_{k+1,k}$, $L_{2k-1,k}$ i $L_{2k,k}$.
3. For $i = k+1, \ldots, 2k-3$ the matrix $L_{i,i}$ such that
   $L_{i,i}L_{i,i}^T = A_{i,i} - L_{i,i-k-1}L_{i,i-k-1}^T - L_{i,i-k}L_{i,i-k}^T - L_{i,i-1}L_{i,i-1}^T$,
   and then $L_{i+1,i}$, $L_{2k-1,i}$ i $L_{2k,i}$.
4. The matrix $L_{2k-2,2k-2}$, such that
   $L_{2k-2,2k-2}L_{2k-2,2k-2}^T = A_{2k-2,2k-2}$
   $-L_{2k-2,k-3}L_{2k-2,k-3}^T - L_{2k-2,k-2}L_{2k-2,k-2}^T - L_{2k-2,2k-3}L_{2k-2,2k-3}^T$,
   and then $L_{2k-1,2k-2} =$
   $(A_{2k-1,2k-2} - L_{2k-1,k-2}L_{2k-2,k-2}^T - L_{2k-1,2k-3}L_{2k-2,2k-3}^T)L_{2k-2,2k-2}^{-T}$
   and $L_{2k,2k-2} = (A_{2k,2k-2} - L_{2k,k-2}L_{2k-2,k-2}^T - L_{2k,2k-3}L_{2k-2,2k-3}^T)L_{2k-2,2k-2}^{-T}$.
5. The matrix $L_{2k-1,2k-1}$, such that
   $L_{2k-1,2k-1}L_{2k-1,2k-1}^T = A_{2k-1,2k-1} - \sum_{l=k-2}^{2k-2} L_{2k-1,l}L_{2k-1,l}^T$,
   and then $L_{2k,2k-1} = (A_{2k,2k-1} - \sum_{l=k-2}^{2k-2} L_{2k,l}L_{2k-1,l}^T)L_{2k-1,2k-1}^{-T}$.
6. The matrix $L_{2k,2k}$, such that $L_{2k,2k}L_{2k,2k}^T = A_{2k,2k} - \sum_{l=0}^{2k-1} L_{2k,l}L_{2k,l}^T$.

Procedures

```
int pkn_Block2ArraySize ( int k, int r, int s, int t );
int pkn_Block2FindBlockPos ( int k, int r, int s, int t,
                             int i, int j );
int pkn_Block2FindElemPos ( int k, int r, int s, int t,
                            int i, int j );
```

```
boolean pkn_Block2CholeskyDecompMf ( int k, int r, int s, int t,
                                     float *A );
```

```
void pkn_Block2LowerTrMSolvef ( int k, int r, int s, int t,
                                const float *L,
                                int spdimen, int xpitch, float *x );
void pkn_Block2UpperTrMSolvef ( int k, int r, int s, int t,
                                const float *L,
                                int spdimen, int xpitch, float *x );
```

```
void pkn_Block2SymMatrixMultf ( int k, int r, int s, int t,
                           float *A,
                           int spdimen, int xpitch, const float *x,
                           int ypitch, float *y );
```


3.5.3 Matrices of the third block type structure

The third block structure of matrices (Block3) differs from the first structure by the presence of nonzero co-diagonal blocks. A symmetric, positive-definite matrix having such a structure may be decomposed into triangular factors, $A = LL^T$, example is in Figure 3.10. The matrix is divided into $(k+1) \times (k+1)$ blocks, in

$$\begin{bmatrix} A_{00} & A_{10}^T & & & A_{40}^T \\ A_{10} & A_{11} & A_{21}^T & & A_{41}^T \\ & A_{21} & A_{22} & A_{32}^T & A_{42}^T \\ & & A_{32} & A_{33} & A_{43}^T \\ A_{40} & A_{41} & A_{42} & A_{43} & A_{44} \end{bmatrix} \qquad \begin{bmatrix} L_{00} & & & & \\ L_{10} & L_{11} & & & \\ & L_{21} & L_{22} & & \\ & & L_{32} & L_{33} & \\ L_{40} & L_{41} & L_{42} & L_{43} & L_{44} \end{bmatrix}$$

Figure 3.10. Block structure of the third type for a symmetric matrix and a lower triangular matrix

rows and columns numbered from 0 to k. The blocks in the columns $0, \ldots, k-1$ have r columns each, the blocks in the k-th column have s columns. The blocks in the rows $0, \ldots, k-1$ have r rows, the blocks in the k-th row have s rows. If $k = 2$, then such a matrix is full, but the storage of its coefficients in the array used to represent it is specific.

Symmetric diagonal blocks $A_{00}, \ldots, A_{kk}$ are represented in the packed form (lower triangle only, see Section 3.3). Other blocks, having dimensions $s \times r$ (in the bottom row) and $r \times r$ (under the diagonal, except for the bottom row) are represented like full matrices.

```
int pkn_Block3ArraySize ( int k, int r, int s );
int pkn_Block3FindBlockPos ( int k, int r, int s, int i, int j );
int pkn_Block3FindElemPos ( int k, int r, int s, int i, int j );
```

```
boolean pkn_Block3CholeskyDecompMf ( int k, int r, int s,
                                     float *A );
```

```
void pkn_Block3LowerTrMSolvef ( int k, int r, int s,
                                const float *L,
                                int spdimen, int xpitch, float *x );
void pkn_Block3UpperTrMSolvef ( int k, int r, int s,
                                const float *L,
                                int spdimen, int xpitch, float *x );
```

```
void pkn_Block3SymMatrixMultf ( int k, int r, int s,
                          const float *A,
                          int spdimen, int xpitch, const float *x,
                          int ypitch, float *y );
```


3.6 Irregular sparse matrices

A sparse matrix, whose nonzero coefficients may be a small fraction of all coefficients, may (and often have to) be represented using memory saving data structures. The representation of matrices used by the procedures described in this section is the following: two numbers, m and n, are the numbers of rows and columns respectively. Then n_{nz} is the number of nonzero coefficients (it must be between 0 and mn). The array `nzi`, whose entries are of type `index2`, contains positions of the nonzero coefficients a_{ij}, where $0 \le i < m$, $0 \le j < n$. The actual coefficients are stored *in a separate array* `a`, so that if $k \in \{0, \ldots, n_{nz} - 1\}$ and `nzi[k]` == {i,j}, then a_{ij} == `a[k]`. The ordering of entries in these arrays is irrelevant (in particular it is not assumed that the row or column indices are stored in ascending or any at all order).

Due to the separation of the distribution of the nonzero coefficients from the actual coefficients, the coefficients may be of various types: `float`, `double`, complex, vectors or matrices. Currently the procedures in the library implement multiplication algorithms for sparse matrices, whose coefficients are `float` and `double`.

There is another possibility—using submatrices. The `nzi` array in a submatrix representation has entries of type `index3`. The fields `i` and `j` are the indices of the row and column of the coefficient a_{ij}. The field `k` is the index to the array in which the coefficients are stored. In this way it is possible to define a number of submatrices of a sparse matrix without copying the coefficients.

```
typedef struct {
    int i, j;
  } index2;

typedef struct {
    int i, j, k;
  } index3;
```

Apart from the two arrays mentioned above, additional arrays may be used; permutation arrays, which hold orderings of the entries (sorted by rows and by columns in each row, or sorted by columns and by rows in each column), and additional arrays, with indices to the permutation arrays, pointing the first entry for each row or column. These arrays are used by procedures of matrix multiplication described later.

```
void pkn_SPMindex2to3 ( int nnz, index2 *ai, index3 *sai );
void pkn_SPMindex3to2 ( int nnz, index3 *sai, index2 *ai );
```

The procedures convert indices, which is sometimes useful. The parameter `nnz` is the length of the arrays being the next two parameters.

The procedure pkn_SPMindex2to3 for each entry of ai copies the fields i and j and assigns sai[k] = k;

The procedure pkn_SPMindex3to2 for each entry of sai copies the fields i and j and forgets k.

3.6.1 Multiplication of a matrix and a vector

```
boolean pkn_MultSPMVectorf ( int nrows, int ncols, int nnz,
                           const index2 *ai, const float *ac,
                           int spdimen, const float *x,
                           float *y );
boolean pkn_MultSPMTVectorf ( int nrows, int ncols, int nnz,
                            const index2 *ai, const float *ac,
                            int spdimen, const float *x,
                            float *y );

boolean pkn_MultSPsubMVectorf ( int nrows, int ncols, int nnz,
                              const index3 *ai, const float *ac,
                              int spdimen, const float *x,
                              float *y );
boolean pkn_MultSPsubMTVectorf ( int nrows, int ncols, int nnz,
                               const index3 *ai, const float *ac,
                               int spdimen, const float *x,
                               float *y );
```

The procedures above compute the product Ax or A^Tx, where A is a sparse matrix or submatrix, and x is a vector or a full matrix, whose rows have the length d =spdimen and are stored in an array x without gaps in between.

The parameters m =nrows and n =ncols specify the dimensions of the array A. The parameter nnz is the number of nonzero coefficients of A. The array ai contains positions of these coefficients and ac are the actual coefficients.

The array y is the place to store the result. Its length must be respectively md or nd if Ax or A^Tx is computed.

3.6.2 Multiplication of two sparse matrices

Multiplication of two sparse matrices may be done in a number of ways, which is the reason of providing that many procedures for that operation. First of all, as the product of two sparse matrices is usually also a sparse matrix, and the sparse representation is used for it, it is necessary to count the nonzero coefficients of the result before computing them—the application must allocate sufficient arrays and then the multiplication may be done.

There are two approaches to the multiplication of sparse matrices implemented so far. The first method is to do it directly, i.e. to obtain the result in the allocated arrays. Here the numerical computations are done together with finding the distribution of nonzero ceofficients of the product. The second approach separates these two computations. As each coefficient of the product is the sum of product of some coefficients of the factors, it is possible to find the list of the entries of the factors to multiply and add for each nonzero coefficients, and then store these lists in an additional array. The numerical computations, which take much less time, are done by a separate procedure. This approach is particularly effective, if there are a number of products of matrices, which have the same distributions of nonzero coefficients. A drawback is the amount of memory needed for storing the multiplication lists, which may be prohibitive for huge matrices.

Sorting by rows and columns

```
boolean pkn_SPMSortByRows ( int nrows, int ncols, int nnz,
                          index2 *ai, int *permut );
boolean pkn_SPMSortByCols ( int nrows, int ncols, int nnz,
                          index2 *ai, int *permut );
```

The parameters m =nrows and n =ncols specify the dimensions of a matrix A, the number n_{nz} =nnz is the number of nonzero coefficients, whose distribution is given in the array ai. The array permut of length n_{nz} on exit contains the permuted numbers from 0 to $n_{nz} - 1$.

The procedure pkn_SPMSortByRows finds the permutation such that if $k < l$ then ai[k].i < ai[l].i or (ai[k].i == ai[l].i and ai[k].j <= ai[l].j). This permutation establishes the ordering by rows.

The procedure pkn_SPMSortByCols finds the permutation such that if $k < l$ then ai[k].j < ai[l].j or (ai[k].j == ai[l].j and ai[k].i <= ai[l].i). This permutation establishes the ordering by columns.

The return value is true if the computation has been successful, or false if the sorting procedure failed because of insufficient scratch memory.

```
boolean pkn_SPMFindRows ( int nrows, int ncols, int nnz,
                        index2 *ai, int *permut, boolean ro,
                        int *rows );
boolean pkn_SPMFindCols ( int nrows, int ncols, int nnz,
                        index2 *ai, int *permut, boolean co,
                        int *cols );
```

The parameters m =nrows and n =ncols specify the dimensions of a matrix A, the number n_{nz} =nnz is the number of nonzero coefficients, whose distribution is given in the array ai.

If the next parameter, ro or co, is nonzero (e.g. true), the array permut on entry must contain the odering by rows or by columns, respectively. If the parameter ro or co is zero (false), then the sorting procedure pkn_SPMSortByRows or pkn_SPMSortByCols will be called.

The array rows must be of length $m + 1$, the array cols must be of length $n + 1$. On exit, this array contains indices of the first entries to the permut array, corresponding to the rows or columns (and the last entry is n_{nz}).

The return value is true if the computation has been successful, or false if an error has been detected or the sorting procedure failed.

```
boolean pkn_SPsubMSortByRows ( int nrows, int ncols, int nnz,
                               index3 *ai, int *permut );
boolean pkn_SPsubMSortByCols ( int nrows, int ncols, int nnz,
                               index3 *ai, int *permut );
boolean pkn_SPsubMFindRows ( int nrows, int ncols, int nnz,
                             index3 *ai, int *permut, boolean ro,
                             int *rows );
boolean pkn_SPsubMFindCols ( int nrows, int ncols, int nnz,
                             index3 *ai, int *permut, boolean co,
                             int *cols );
```

The four procedures above do the same things for sparse submatrices, that the previous four procedures for sparse matrices. The fields k of the index3 structures are ignored.

Counting the nonzero coefficients of the product

The algorithm of multiplying two sparse matrices has two variants, using respectively the row and the column ordering of the nonzero coefficients. The procedures implementing the first variant have names ending with the letter R, and these with the second variant have names with the suffix C (the suffix may be followed by the letter f or d, indicating the floating point precision; procedures with names without this letter do not make any floating point operations and they serve for both precisions). The complete set of multiplication procedures is only for the "C" version.

The number of procedures is increased by the fact that having two matrices, A and B, one may be interested in computing AB, AB^T or A^TB (there are no procedures for A^TB^T, but they might be added if necessary). Which procedure does what is indicated by the infix MM, MMT or MTM in the procedure identifier. Also there are variants for multiplying submatrices, which doubles the number of multiplication procedures.


```
boolean pkn_SPMCountMMnnzR ( int nra, int nca, int ncb,
                             int nnza, index2 *ai,
                             int *apermut, int *arows, boolean ra,
                             int nnzb, index2 *bi,
                             int *bpermut, int *brows, boolean rb,
                             int *nnzab, int *nmultab );
boolean pkn_SPMCountMMnnzC ( int nra, int nca, int ncb,
                             int nnza, index2 *ai,
                             int *apermut, int *acols, boolean ca,
                             int nnzb, index2 *bi,
                             int *bpermut, int *bcols, boolean cb,
                             int *nnzab, int *nmultab );
boolean pkn_SPMCountMMTnnzR ( int nra, int nca, int nrb,
                            int nnza, index2 *ai,
                            int *apermut, int *arows, boolean ra,
                            int nnzb, index2 *bi,
                            int *bpermut, int *bcols, boolean cb,
                            int *nnzab, int *nmultab );
boolean pkn_SPMCountMMTnnzC ( int nra, int nca, int nrb,
                            int nnza, index2 *ai,
                            int *apermut, int *acols, boolean ca,
                            int nnzb, index2 *bi,
                            int *bpermut, int *brows, boolean rb,
                            int *nnzab, int *nmultab );
boolean pkn_SPMCountMTMnnzR ( int nra, int nca, int ncb,
                            int nnza, index2 *ai,
                            int *apermut, int *acols, boolean ca,
                            int nnzb, index2 *bi,
                            int *bpermut, int *brows, boolean rb,
                            int *nnzab, int *nmultab );
boolean pkn_SPMCountMTMnnzC ( int nra, int nca, int ncb,
                            int nnza, index2 *ai,
                            int *apermut, int *arows, boolean ra,
                            int nnzb, index2 *bi,
                            int *bpermut, int *bcols, boolean cb,
                            int *nnzab, int *nmultab );
```

The procedures, whose headers are shown above, count the nonzero coefficients of the product of two sparse matrices and the total number of floating point multiplication of coefficients necessary to compute the product.

The parameters of these procedures are: nra, nca—numbers of rows and columns of the matrix A, nrb, ncb—numbers of rows and columns of the matrix B (always

one of those parameters is absent, as the dimensions of the matrices to multiply must match).

The parameters nnza and nnzb are numbers of nonzero coefficients, the arrays ai and bi contain distributions of the nonzero coefficients.

The arrays apermut and bpermut are used to store the permutations establishing the orderings of coefficients of the matrices. If the parameter ra is nonzero, then the permutation in apermut must represent the row ordering, and the array arows must contain indices to the apermut array, pointing to the first entries of subsequent rows (see the procedure pkn_SPsubMFindRows).

If the parameter ca is nonzero, then the permutation in apermut must represent the column ordering, and the array acols must contain indices to the apermut array, pointing to the first entries of subsequent columns (see the procedure pkn_SPsubMFindCols).

If the parameter ra or ca is zero (false), the proper ordering and indices to the rows or columns will be found, but the caller must provide arrays of sufficient capacity.

The same rules apply to the parameters bpermut, brows, bcols, rb and cb.

The number of nonzero coefficients of the product is assigned to the variable pointed by parameter nnzab, and the total number of floating point multiplications is assigned to the variable pointed by nmultab.

The procedures return true if the computation was successful, or false if an error has been detected or there was insufficient scratch memory.

```
boolean pkn_SPsubMCountMMnnzR ( int nra, int nca, int ncb,
                              int nnza, index3 *ai,
                              int *apermut, int *arows, boolean ra,
                              int nnzb, index3 *bi,
                              int *bpermut, int *brows, boolean rb,
                              int *nnzab, int *nmultab );
boolean pkn_SPsubMCountMMnnzC ( int nra, int nca, int ncb,
                              int nnza, index3 *ai,
                              int *apermut, int *acols, boolean ca,
                              int nnzb, index3 *bi,
                              int *bpermut, int *bcols, boolean cb,
                              int *nnzab, int *nmultab );
boolean pkn_SPsubMCountMMTnnzR ( int nra, int nca, int nrb,
                               int nnza, index3 *ai,
                               int *apermut, int *arows, boolean ra,
                               int nnzb, index3 *bi,
                               int *bpermut, int *bcols, boolean cb,
                               int *nnzab, int *nmultab );
boolean pkn_SPsubMCountMMTnnzC ( int nra, int nca, int nrb,
                               int nnza, index3 *ai,
```



```
                               int *apermut, int *acols, boolean ca,
                               int nnzb, index3 *bi,
                               int *bpermut, int *brows, boolean rb,
                               int *nnzab, int *nmultab );
boolean pkn_SPsubMCountMTMnnzR ( int nra, int nca, int ncb,
                               int nnza, index3 *ai,
                               int *apermut, int *acols, boolean ca,
                               int nnzb, index3 *bi,
                               int *bpermut, int *brows, boolean rb,
                               int *nnzab, int *nmultab );
boolean pkn_SPsubMCountMTMnnzC ( int nra, int nca, int ncb,
                               int nnza, index3 *ai,
                               int *apermut, int *arows, boolean ra,
                               int nnzb, index3 *bi,
                               int *bpermut, int *bcols, boolean cb,
                               int *nnzab, int *nmultab );
```

These procedures count the number of nonzero coefficients of the product and the total number of floating point multiplications for sparse submatrices. See the description of the procedures described in this section, whose names do not have the infix sub.

Finding the distribution of nonzero coefficients of the product

```
boolean pkn_SPMmultMMCempty ( int nra, int nca, int ncb,
                            int nnza, index2 *ai,
                            int *apermut, int *acols, boolean ca,
                            int nnzb, index2 *bi,
                            int *bpermut, int *bcols, boolean cb,
                            index2 *abi );
boolean pkn_SPMmultMMTCempty ( int nra, int nca, int nrb,
                             int nnza, index2 *ai,
                             int *apermut, int *acols, boolean ca,
                             int nnzb, index2 *bi,
                             int *bpermut, int *brows, boolean rb,
                             index2 *abi );
boolean pkn_SPMmultMTMCempty ( int nra, int nca, int ncb,
                             int nnza, index2 *ai,
                             int *apermut, int *arows, boolean ra,
                             int nnzb, index2 *bi,
                             int *bpermut, int *bcols, boolean ba,
                             index2 *abi );
```

The procedures above find the distribution of nonzero coefficients of the product of two matrices, AB, AB^T or A^TB respectively, based on the distribution of the nonzero coefficients of the matrices A and B.

The parameters of these procedures are: nra, nca—numbers of rows and columns of the matrix A, nrb, ncb—numbers of rows and columns of the matrix B (always one of those parameters is absent, as the dimensions of the matrices to multiply must match).

The parameters nnza and nnzb are numbers of nonzero coefficients, the arrays ai and bi contain distributions of the nonzero coefficients.

The arrays apermut and bpermut are used to store the permutations establishing the orderings of coefficients of the matrices. If the parameter ra is nonzero, then the permutation in apermut must represent the row ordering, and the array arows must contain indices to the apermut array, pointing to the first entries of subsequent rows (see the procedure pkn_SPsubMFindRows).

If the parameter ca is nonzero, then the permutation in apermut must represent the column ordering, and the array acols must contain indices to the apermut array, pointing to the first entries of subsequent columns (see the procedure pkn_SPsubMFindCols).

If the parameter ra or ca is zero (false), the proper ordering and indices to the rows or columns will be found, but the caller must provide arrays of sufficient capacity.

The same rules apply to the parameters bpermut, brows, bcols, rb and cb.

The array abi of length determined by procedure pkn_SPMCountMMnnzC (for AB), pkn_SPMCountMMTnnzC (for AB^T), or pkn_SPMCountMTMnnzC (for A^TB—this length is assigned to the variable pointed by the parameter nnzab of these procedures) must be allocated by the caller. On exit it contains the distribution of the nonzero coefficients of the product. No actual multiplication of the coefficient is done.

The fast matrix multiplication

```
boolean pkn_SPMmultMMCf ( int nra, int nca, int ncb,
                          int nnza, index2 *ai, float *ac,
                          int *apermut, int *acols, boolean ca,
                          int nnzb, index2 *bi, float *bc,
                          int *bpermut, int *bcols, boolean cb,
                          index2 *abi, float *abc );
boolean pkn_SPMmultMMTCf ( int nra, int nca, int nrb,
                           int nnza, index2 *ai, float *ac,
                           int *apermut, int *acols, boolean ca,
                           int nnzb, index2 *bi, float *bc,
                           int *bpermut, int *brows, boolean rb,
                           index2 *abi, float *abc );
boolean pkn_SPMmultMTMCf ( int nra, int nca, int ncb,
                           int nnza, index2 *ai, float *ac,
                           int *apermut, int *arows, boolean ra,
                           int nnzb, index2 *bi, float *bc,
                           int *bpermut, int *bcols, boolean cb,
                           index2 *abi, float *abc );
```


The procedures above compute the product of two sparse matrices, AB, AB^T or A^TB respectively, which involves finding the distribution of nonzero coefficients of the product.

The parameters of these procedures are: nra, nca—numbers of rows and columns of the matrix A, nrb, ncb—numbers of rows and columns of the matrix B (always one of those parameters is absent, as the dimensions of the matrices to multiply must match).

The parameters nnza and nnzb are numbers of nonzero coefficients, the arrays ai and bi contain distributions of the nonzero coefficients.

The arrays apermut and bpermut are used to store the permutations establishing the orderings of coefficients of the matrices. If the parameter ra is nonzero, then the permutation in apermut must represent the row ordering, and the array arows must contain indices to the apermut array, pointing to the first entries of subsequent rows (see the procedure pkn_SPsubMFindRows).

If the parameter ca is nonzero, then the permutation in apermut must represent the column ordering, and the array acols must contain indices to the apermut array, pointing to the first entries of subsequent columns (see the procedure pkn_SPsubMFindCols).

If the parameter ra or ca is zero (false), the proper ordering and indices to the rows or columns will be found, but the caller must provide arrays of sufficient capacity.

The same rules apply to the parameters bpermut, brows, bcols, rb and cb.

The arrays abi and abc, whose length ought to be determined by the procedure pkn_SPMCountMMnnzC (for AB), pkn_SPMCountMMTnnzC (for AB^T), or pkn_SPMCountMTMnnzC (for A^TB—this length is assigned to the variable pointed by the parameter nnzab of these procedures) must be allocated by the caller. On exit the array abi contains the distribution of the nonzero coefficients of the product, and the coefficients are stored in the array abc.

```
boolean pkn_SPsubMmultMMCf ( int nra, int nca, int ncb,
                             int nnza, index3 *ai, float *ac,
                             int *apermut, int *acols, boolean ca,
                             int nnzb, index3 *bi, float *bc,
                             int *bpermut, int *bcols, boolean cb,
                             index2 *abi, float *abc );
```

```
boolean pkn_SPsubMmultMMTCf ( int nra, int nca, int nrb,
                          int nnza, index3 *ai, float *ac,
                          int *apermut, int *acols, boolean ca,
                          int nnzb, index3 *bi, float *bc,
                          int *bpermut, int *brows, boolean rb,
                          index2 *abi, float *abc );
boolean pkn_SPsubMmultMTMCf ( int nra, int nca, int ncb,
                          int nnza, index3 *ai, float *ac,
                          int *apermut, int *arows, boolean ra,
                          int nnzb, index3 *bi, float *bc,
                          int *bpermut, int *bcols, boolean cb,
                          index2 *abi, float *abc );
```

The procedures above multiply two sparse submatrices, in the way analoguous to that used by the three procedures without the infix "sub" in their names, which are described above. Note that the result of the multiplication is a sparse *matrix*, not a submatrix, i.e. the distribution of the product is represented by the structures of type index2, not index3. For description of the parameters see the procedures for matrix multiplication.

The very fast matrix multiplication

A very fast matrix multiplication is possible if the distribution of the nonzero coefficients of the product is known and for each nonzero coefficient of the product a list of coefficients of the factors, to be multiplied and added, is known. In some applications this information may be found once, and then the floating point computations are instant.

The preparation involves counting the nonzero coefficients and multiplications, by calling one of the procedures pkn_SPMCountMMnnzR, pkn_SPMCountMMTnnzR, pkn_SPMCountMTMnnzR, pkn_SPMCountMMnnzC, pkn_SPMCountMMTnnzC, pkn_SPMCountMTMnnzC, or an analoguous procedure with the "sub" infix. These procedures find the numbers n_{nz} and n_{mult}, which are the number of nonzero coefficients of the product and the total number of multiplications to compute them.

Then it is necessary to allocate the arrays abi of length n_{nz} (to store the distribution of nonzero coefficients of the product), abpos of length $n_{nz}+1$ (to store the indices to the next array) and aikbkj of length n_{mult} (to store the lists of coefficients to multiply and add). The last stage of the preparation is calling one of the procedures, whose headers are shown below, to find the distribution of the nonzero product coefficients and the lists of coefficients of the factors to multiply.

After the preparation, the procedure pkn_SPMFastMultMMf may be called. This procedure makes only the floating point operations, and it is appropriate for all cases—AB, AB^T, A^TB, using the row and column ordering. Time savings are

considerable if the preparation is done once and then a number of matrices with the same distributions of nonzero coefficients are to be multiplied, on the other hand the lists of coefficients to multiply need much memory, which may be not available for huge matrices.

```
boolean pkn_SPMFindMMnnzR ( int nra, int nca, int ncb,
                    int nnza, index2 *ai, int *apermut, int *arows,
                    int nnzb, index2 *bi, int *bpermut, int *brows,
                    index2 *abi, int *abpos, index2 *aikbkj );
boolean pkn_SPMFindMMnnzC ( int nra, int nca, int ncb,
                    int nnza, index2 *ai, int *apermut, int *acols,
                    int nnzb, index2 *bi, int *bpermut, int *bcols,
                    index2 *abi, int *abpos, index2 *aikbkj );
boolean pkn_SPMFindMMTnnzR ( int nra, int nca, int nrb,
                     int nnza, index2 *ai, int *apermut, int *arows,
                     int nnzb, index2 *bi, int *bpermut, int *bcols,
                     index2 *abi, int *abpos, index2 *aikbkj );
boolean pkn_SPMFindMMTnnzC ( int nra, int nca, int nrb,
                     int nnza, index2 *ai, int *apermut, int *acols,
                     int nnzb, index2 *bi, int *bpermut, int *brows,
                     index2 *abi, int *abpos, index2 *aikbkj );
boolean pkn_SPMFindMTMnnzR ( int nra, int nca, int ncb,
                     int nnza, index2 *ai, int *apermut, int *acols,
                     int nnzb, index2 *bi, int *bpermut, int *brows,
                     index2 *abi, int *abpos, index2 *aikbkj );
boolean pkn_SPMFindMTMnnzC ( int nra, int nca, int ncb,
                     int nnza, index2 *ai, int *apermut, int *arows,
                     int nnzb, index2 *bi, int *bpermut, int *bcols,
                     index2 *abi, int *abpos, index2 *aikbkj );
```

The procedures above find the distributions of nonzero coefficients of the product of two sparse matrices, and the lists of factor coefficients to multiply, as a part of preparation for the fastest procedure of multiplying the sparse matrices in this package.

The parameters of these procedures are: nra, nca—numbers of rows and columns of the matrix A, nrb, ncb—numbers of rows and columns of the matrix B (always one of those parameters is absent, as the dimensions of the matrices to multiply must match).

The parameters nnza and nnzb are numbers of nonzero coefficients, the arrays ai and bi contain distributions of the nonzero coefficients.

The arrays apermut and bpermut are used to store the permutations establishing the orderings of coefficients of the matrices. If the parameter ra is nonzero, then the permutation in apermut must represent the row ordering, and the array arows

must contain indices to the apermut array, pointing to the first entries of subsequent rows (see the procedure pkn_SPsubMFindRows).

The permutation in apermut must represent the column ordering, and the array acols must contain indices to the apermut array, pointing to the first entries of subsequent columns (see the procedure pkn_SPsubMFindCols).

The same rules apply to the arrays bpermut and brows or bcols.

Results are stored in the arrays abi, abpos and aikbkj, which must be provided by the caller. The lengths of these arrays may be found in the way described above the headers.

The procedures return true in case of success or false after a failure.

```
boolean pkn_SPsubMFindMMnnzR ( int nra, int nca, int ncb,
                    int nnza, index3 *ai, int *apermut, int *arows,
                    int nnzb, index3 *bi, int *bpermut, int *brows,
                    index2 *abi, int *abpos, index2 *aikbkj );
boolean pkn_SPsubMFindMMnnzC ( int nra, int nca, int ncb,
                    int nnza, index3 *ai, int *apermut, int *acols,
                    int nnzb, index3 *bi, int *bpermut, int *bcols,
                    index2 *abi, int *abpos, index2 *aikbkj );
boolean pkn_SPsubMFindMMTnnzR ( int nra, int nca, int nrb,
                     int nnza, index3 *ai, int *apermut, int *arows,
                     int nnzb, index3 *bi, int *bpermut, int *bcols,
                     index2 *abi, int *abpos, index2 *aikbkj );
boolean pkn_SPsubMFindMMTnnzC ( int nra, int nca, int nrb,
                     int nnza, index3 *ai, int *apermut, int *acols,
                     int nnzb, index3 *bi, int *bpermut, int *brows,
                     index2 *abi, int *abpos, index2 *aikbkj );
boolean pkn_SPsubMFindMTMnnzR ( int nra, int nca, int ncb,
                     int nnza, index3 *ai, int *apermut, int *acols,
                     int nnzb, index3 *bi, int *bpermut, int *brows,
                     index2 *abi, int *abpos, index2 *aikbkj );
boolean pkn_SPsubMFindMTMnnzC ( int nra, int nca, int ncb,
                     int nnza, index3 *ai, int *apermut, int *arows,
                     int nnzb, index3 *bi, int *bpermut, int *bcols,
                     index2 *abi, int *abpos, index2 *aikbkj );
```

The procedures with headers shown above do the same thing as their corresponding procedures without the infix "sub", in order to prepare the fast multiplication of sparse submatrices. Note that the product is represented as a sparse matrix (not a submatrix), whose distribution of nonzero coefficients is represented by structures of type index2.

To see a description of the parameters, see the procedures above.


```
void pkn_SPMFastMultMMf ( float *ac, float *bc,
                          int nnzab, int *abpos, index2 *aikbkj,
                          float *abc );
```

The procedure pkn_SPMFastMultMMf performs multiplication and summing of floating point numbers in order to compute the coefficients of the product of two sparse matrices. Before calling it it is necessary to prepare the computation, using the procedures described above.

3.7 Conjugate gradient method for linear equations

Systems of linear equations $A\mathbf{x} = \mathbf{b}$ with a large matrix A should be solved using iterative methods; sometimes it is the only approach having a chance to work. If the matrix A is symmetric and positive-definite, then an algorithm worth using is the conjugate gradient method. The numerical operations in this method are: computation of the products $A\boldsymbol{v}$ for some vectors $\boldsymbol{v}$, and computing scalar products and linear combination of vectors. It is also possible to introduce a **preconditioner**, i.e. a matrix Q, which has three properties: it is symmetric and positive definite, it is an approximation of A, and it is easy to compute the product of $Q^{-1}\boldsymbol{w}$ for any vector $\boldsymbol{w}$ (i.e. to solve the system of equations $Q\mathbf{u} = \boldsymbol{w}$, which must be possible and much easier than solving $A\mathbf{x} = \mathbf{b}$).

```
boolean pkn_PCGf ( int n, void *usrdata, float *b, float *x,
          boolean (*multAx)( int n, void *usrdata,
                             const float *x, float *Ax ),
          boolean (*multQIx)( int n, void *usrdata,
                              const float *x, float *Qix ),
          int maxit, float eps, float delta, int *itm );
```

The procedure `pkn_PCGf` implements the conjugate gradient method of solving systems of linear equations $A\mathbf{x} = \mathbf{b}$ with symmetric positive-definite matrices A, using a preconditioner Q.

Parameters: `n`—number of equations and unknown variables, `usrdata`—pointer to any data structure, which is passed to the subprograms pointed by the parameters `multAx` and `multQIx`, `b`—array of coordinates of the vector $\mathbf{b}$.

The array `x` on entry contains an initial approximation of the solution (it must be initialised—set all entries to 0 if there is no better idea), on exit it contains the solution, or rather an approximation of the solution obtained after the last iteration.

The subprogram pointed by `multAx` must multiply the array A by vector $\mathbf{x}$, whose coordinates are in the array `x`, and store the result in the array `Ax`. The representation of the matrix A and the implementation of the multiplication is completely irrelevant. The data representing A may be accessible via the pointer `usrdata`, which is passed to this subprogram whenever `pkn_PCGf` calls it.

Similarly the subprogram pointed by `multQIx` must compute the product $Q^{-1}\boldsymbol{x}$ and store it in the array `Qix`. The way of representing the preconditioner Q or its inverse is irrelevant. If the parameter `multQIx` is `NULL`, then it is assumed that the preconditioner is the $n \times n$ identity matrix.

The values returned by the subprograms pointed by `multAx` and `multQIx` should be `true` in case of success and `false` in case of any failure. Returning `false` will result in termination of the conjugate gradient method iterations.

The parameter `maxit` is the limit of the number of iterations (it must be positive and not greater than n). The parameters `eps` (ε) and `delta` (δ) specify the

stop criterions—iterations are terminated if $\|\boldsymbol{v}\|_2 < \varepsilon$ or $\|\mathbf{r}\|_2 < \delta$ or the limit of iterations has been reached. Here $\boldsymbol{v}$ denotes the vector constructed by the conjugate gradient method, which determines the direction of the next line, along which the quadratic polynomial $\frac{1}{2}\mathbf{x}^T A\mathbf{x} - \mathbf{x}^T\mathbf{b}$ is minimised, and $\mathbf{r} = \mathbf{b} - A\mathbf{x}$ is the residuum vector.

The parameter `itm` points to a variable, to which `pkn_PCGf` assigns the number of iterations made.

The returned value of `pkn_PCGf` is `true` in case of success, or `false` if one of the subprograms passed as parameters failed (i.e. returned `false`) or there was insufficient scratch memory.

3.8 Triangular bit matrices

Matrices, whose elements are bits, may be used to represent the distribution of nonzero coefficients of sparse matrices, whose elements are numbers. This is useful e.g. to renumber the equations and variables of a system of equations in order to obtain a band matrix (with a narrow band, containing relatively few zeros), such that the system may be solved using a direct method. This approach was used in the procedures of shape optimization of the surfaces represented by meshes (in the `libg2blending` library—see Section 12.3). As the matrices used there are symmetric, the procedures described below process a packed representation of triangular matrices (such a matrix may be interpreted as the lower triangle of a symmetric matrix, and it takes a half of the storage space necessary for a square matrix).

```
int pkn_TMBSize ( int n );
```

The procedure `pkn_TMBsize` computes the number of bytes necessary to represent an $n \times n$ bit matrix. An application is supposed to call it prior to the memory allocation for the bit matrix.

```
boolean pkn_TMBElem ( byte *bittm, int i, int j );
```

The procedure `pkn_TMBElem` returns `true` if $b_{ij} = 1$ of `false` if $b_{ij} = 0$. The bits of the matrix are stored in the array `bittm`, and `i` and `j` are indices of the row and column, which are numbered from 0 to $n - 1$.

```
void pkn_TMBElemSet ( byte *bittm, int i, int j );
void pkn_TMBElemClear ( byte *bittm, int i, int j );
```

The two procedures above assign 1 or 0 respectively to the bit b_{ij} of the bit matrix. The bits are stored in the array `bittm`, and `i` and `j` are indices of the row and column, which are numbered from 0 to $n - 1$.

```
boolean pkn_TMBTestAndSet ( byte *bittm, int i, int j );
boolean pkn_TMBTestAndClear ( byte *bittm, int i, int j );
```

The two procedures above assign 1 or 0 respectively to the bit b_{ij} of the bit matrix, and their return value is the previous value of that bit. The bits are stored in the array `bittm`, and `i` and `j` are indices of the row and column, which are numbered from 0 to $n - 1$.

These operations are not uninterruptible, as is often necessary in concurrent programming. Joining the two operations is motivated by saving time of computing the proper byte and mask, which would have to be done twice if the two operations (extracting the bit and assigning the new value) were separated.

3.9 Solving nonlinear equations

```
boolean pkn_SolveSqEqf ( float p, float q, float *x1, float *x2 );
```

The procedure `pkn_SolveSqEqf` computes the zeros of the polynomial $x^2 + 2px + q$ with real coefficients. The parameters `p` and `q` specify the coefficients p and q. The parameters `x1` and `x2` are used to return the result.

If the zeros of the polynomial are *real*, then the procedure returns `true`. Then the value of `*x1` is the smaller zero and the value of `*x2` is the greater zero.

If the zeros are *complex*, then the value of the procedure is `false`. In this case the value of `*x1` is the real part and the value of `*x2` is the absolute value of the imaginary part of the zeros.

```
float pkn_Illinoisf ( float (*f) (float), float a, float b,
                      float eps, boolean *error );
```

The procedure `pkn_Illinoisf` computes with the accuracy up to ε a zero of a real function `f` in the interval $[a, b]$. The function must be continuous in this interval and its values at a and b must have different signs. If the function `f` has more than one zero in $[a, b]$, then the procedure will compute one of them. The numerical algorithm for smooth functions with zeros of multiplicity 1 is usually faster than the bisection.

The parameter `f` is a procedure computing the value of `f` for a given argument. The parameters `a` and `b` specify the interval $[a, b]$, in which the zero is searched. The parameter `eps` specifies the required accuracy ε of the solution (it must be a positive number, and it should not be less than the maximal limit accuracy, depending on the rounding errors of evaluation of the function `f`). The parameter `error` on return is `false` if there was no error detected, and `true` if the values of `f` at both ends of the interval $[a, b]$ have the same sign.

The zero of the function is returned as the value of the procedure.

3.10 Optimization

```
float pkn_GoldenRatf ( float (*f) (float), float a, float b,
                       float eps, boolean *error );
```

The procedure `pkn_GoldenRatf` uses the golden ratio method to find a minimum of a real function `f` of one variable in the interval $[a, b]$. The parameters `a`, `b` specify the ends of this interval the parameter `eps` specifies the required accuracy (its value must be positive), the procedure `*f` has to compute the value of the function `f` at the given point.

The parameter `*error` is assigned `true`, if the procedure does not detect any error (currently it does not detect any errors, but after tests this parameter may find its uses).

The value of the procedure is the computed minimal point of `f`; it is an approximation of some local minimum in the interval $[a, b]$.

3.11 Computing derivatives of composite functions

The procedures described in this section compute partial derivatives of order $1, \ldots, 4$ of a function $\mathbf{h}$, being the composition of a function $\mathbf{f}\colon \mathbb{R}^2 \to \mathbb{R}^2$ and $\mathbf{g}\colon \mathbb{R}^2 \to \mathbb{R}^d$, based on the partial derivatives of these functions. The procedures deal with functions of two variables, though the approach used here may be used for functions of other number of variables (but so far I did not need that).

Let $\mathbf{f}(u, v) = [x(u, v), y(u, v)]^T$. The formulae expressing the derivatives of the composition of consecutive orders may be derived recursively, using the formulae for the derivatives of the first order:

$$\begin{aligned} \mathbf{h}_u &= x_u \mathbf{g}_x + y_u \mathbf{g}_y, \\ \mathbf{h}_v &= x_v \mathbf{g}_x + y_v \mathbf{g}_y, \end{aligned}$$

and the formulae for the derivative of a product of functions. The formulae

$$\begin{aligned} \mathbf{h}_{uu} &= x_u^2 \mathbf{g}_{xx} + 2x_u y_u \mathbf{g}_{xy} + y_u^2 \mathbf{g}_{yy} + x_{uu} \mathbf{g}_x + y_{uu} \mathbf{g}_y, \\ \mathbf{h}_{uv} &= x_u x_v \mathbf{g}_{xx} + (x_u y_v + x_v y_u) \mathbf{g}_{xy} + y_u y_v \mathbf{g}_{yy} + x_{uv} \mathbf{g}_x + y_{uv} \mathbf{g}_y, \\ \mathbf{h}_{vv} &= x_v^2 \mathbf{g}_{xx} + 2x_v y_v \mathbf{g}_{xy} + y_v^2 \mathbf{g}_{yy} + x_{vv} \mathbf{g}_x + y_{vv} \mathbf{g}_y, \end{aligned}$$

and the formulae for the derivatives of higher orders, which are significantly longer, may be rewritten in matrix form, e.g.

$$\begin{bmatrix} \mathbf{h}_u \\ \mathbf{h}_v \end{bmatrix} = A_{11} \begin{bmatrix} \mathbf{g}_x \\ \mathbf{g}_y \end{bmatrix}, \tag{3.1}$$

$$\begin{bmatrix} \mathbf{h}_{uu} \\ \mathbf{h}_{uv} \\ \mathbf{h}_{vv} \end{bmatrix} = A_{21} \begin{bmatrix} \mathbf{g}_x \\ \mathbf{g}_y \end{bmatrix} + A_{22} \begin{bmatrix} \mathbf{g}_{xx} \\ \mathbf{g}_{xy} \\ \mathbf{g}_{yy} \end{bmatrix}, \tag{3.2}$$

$$\begin{bmatrix} \mathbf{h}_{uuu} \\ \mathbf{h}_{uuv} \\ \mathbf{h}_{uvv} \\ \mathbf{h}_{vvv} \end{bmatrix} = A_{31} \begin{bmatrix} \mathbf{g}_x \\ \mathbf{g}_y \end{bmatrix} + A_{32} \begin{bmatrix} \mathbf{g}_{xx} \\ \mathbf{g}_{xy} \\ \mathbf{g}_{yy} \end{bmatrix} + A_{33} \begin{bmatrix} \mathbf{g}_{xxx} \\ \mathbf{g}_{xxy} \\ \mathbf{g}_{xyy} \\ \mathbf{g}_{yyy} \end{bmatrix}, \tag{3.3}$$

$$\begin{bmatrix} \mathbf{h}_{uuuu} \\ \mathbf{h}_{uuuv} \\ \mathbf{h}_{uuvv} \\ \mathbf{h}_{uvvv} \\ \mathbf{h}_{vvvv} \end{bmatrix} = A_{41} \begin{bmatrix} \mathbf{g}_x \\ \mathbf{g}_y \end{bmatrix} + A_{42} \begin{bmatrix} \mathbf{g}_{xx} \\ \mathbf{g}_{xy} \\ \mathbf{g}_{yy} \end{bmatrix} + A_{43} \begin{bmatrix} \mathbf{g}_{xxx} \\ \mathbf{g}_{xxy} \\ \mathbf{g}_{xyy} \\ \mathbf{g}_{yyy} \end{bmatrix} + A_{44} \begin{bmatrix} \mathbf{g}_{xxxx} \\ \mathbf{g}_{xxxy} \\ \mathbf{g}_{xxyy} \\ \mathbf{g}_{xyyy} \\ \mathbf{g}_{yyyy} \end{bmatrix}. \tag{3.4}$$

The coefficients of the matrices $A_{11}, \ldots, A_{44}$ are expressions of the partial derivatives of the functions x and y.

3.11.1 Computing derivative transformation matrices

```
void pkn_Setup2DerA11Matrixf (
        float xu, float yu, float xv, float yv, float *A11 );
void pkn_Setup2DerA21Matrixf ( float xuu, float yuu, float xuv,
        float yuv, float xvv, float yvv, float *A21 );
void pkn_Setup2DerA22Matrixf (
        float xu, float yu, float xv, float yv, float *A22 );
void pkn_Setup2DerA31Matrixf (
        float xuuu, float yuuu, float xuuv, float yuuv,
        float xuvv, float yuvv, float xvvv, float yvvv,
        float *A31 );
void pkn_Setup2DerA32Matrixf (
        float xu, float yu, float xv, float yv,
        float xuu, float yuu, float xuv,
        float yuv, float xvv, float yvv, float *A32 );
void pkn_Setup2DerA33Matrixf (
        float xu, float yu, float xv, float yv, float *A33 );
void pkn_Setup2DerA41Matrixf (
        float xuuuu, float yuuuu, float xuuuv, float yuuuv,
        float xuuvv, float yuuvv, float xuvvv, float yuvvv,
        float xvvvv, float yvvvv, float *A41 );
void pkn_Setup2DerA42Matrixf (
        float xu, float yu, float xv, float yv,
        float xuu, float yuu, float xuv,
        float yuv, float xvv, float yvv,
        float xuuu, float yuuu, float xuuv, float yuuv,
        float xuvv, float yuvv, float xvvv, float yvvv,
        float *A42 );
void pkn_Setup2DerA43Matrixf (
        float xu, float yu, float xv, float yv,
        float xuu, float yuu, float xuv,
        float yuv, float xvv, float yvv, float *A43 );
void pkn_Setup2DerA44Matrixf (
        float xu, float yu, float xv, float yv, float *A44 );
```

The above procedures compute the matrices, which arrear in Formulae (3.1)–(3.4). The parameters of type `float` specify the derivatives of the functions x and y, e.g. the value of the parameter `xu` is x_u, i.e. $\frac{\partial x}{\partial u}$, the value of `yuuv` is $y_{uuv} = \frac{\partial^3 y}{\partial^2 u \partial v}$ etc. The coefficients of the matrices are stored in the arrays pointed by the parameters `A11`...`A44`.

3.11.2 Computing derivatives of composite functions

```
void pkn_Comp2Derivatives1f (
        float xu, float yu, float xv, float yv,
        int spdimen, const float *gx, const float *gy,
        float *hu, float *hv );
void pkn_Comp2Derivatives2f (
        float xu, float yu, float xv, float yv,
        float xuu, float yuu, float xuv,
        float yuv, float xvv, float yvv,
        int spdimen, const float *gx, const float *gy,
        const float *gxx, const float *gxy, const float *gyy,
        float *huu, float *huv, float *hvv );
void pkn_Comp2Derivatives3f (
        float xu, float yu, float xv, float yv,
        float xuu, float yuu, float xuv,
        float yuv, float xvv, float yvv,
        float xuuu, float yuuu, float xuuv, float yuuv,
        float xuvv, float yuvv, float xvvv, float yvvv,
        int spdimen, const float *gx, const float *gy,
        const float *gxx, const float *gxy, const float *gyy,
        const float *gxxx, const float *gxxy,
        const float *gxyy, const float *gyyy,
        float *huuu, float *huuv, float *huvv, float *hvvv );
void pkn_Comp2Derivatives4f (
        float xu, float yu, float xv, float yv,
        float xuu, float yuu, float xuv,
        float yuv, float xvv, float yvv,
        float xuuu, float yuuu, float xuuv, float yuuv,
        float xuvv, float yuvv, float xvvv, float yvvv,
        float xuuuu, float yuuuu, float xuuuv, float yuuuv,
        float xuuvv, float yuuvv, float xuvvv, float yuvvv,
        float xvvvv, float yvvvv,
        int spdimen, const float *gx, const float *gy,
        const float *gxx, const float *gxy, const float *gyy,
        const float *gxxx, const float *gxxy,
        const float *gxyy, const float *gyyy,
        const float *gxxxx, const float *gxxxy, const float *gxxyy,
        const float *gxyyy, const float *gyyyy,
        float *huuuu, float *huuuv, float *huuvv,
        float *huvvv, float *hvvvv );
```

The above procedures compute the partial derivatives of order $1, \ldots, 4$ of the function $\mathbf{h} = \mathbf{f} \circ \mathbf{g}$ based on the partial derivatives of the function $\mathbf{f}\colon \mathbb{R}^2 \to \mathbb{R}^2$,

described by two scalar functions, $x(u,v)$ and $y(u,v)$, and of the function $\mathbf{g}\colon \mathbb{R}^2 \to \mathbb{R}^d$.

The dimension d of the space, whose elements are the values of $\mathbf{g}$, is (for all these procedures) specified by the parameter spdimen.

The names of the other parameters denote their meaning. For example the value of xu is equal to the derivative of x with respect to u; similarly, the parameter yuuvv specifies the value $y_{uuvv} = \frac{\partial^4 y}{\partial^2 u \partial^2 v}$ etc.

Similarly, the parameter gx points to the array of d coordinates of the vector $\mathbf{g}_x = \frac{\partial \mathbf{g}}{\partial x}$, and the parameter hu points to the array, in which the procedure will store the d coordinates of the vector $\frac{\partial \mathbf{h}}{\partial u}$ etc.

As the computation of the derivatives of $\mathbf{h}$ of order n requires only the matrices $A_{n1}, \ldots, A_{nn}$ (see Formulae (3.1)–(3.4)), each procedure computes computes only the derivatives of one order — 1, 2, 3 or 4 respectively. These derivatives are computed based on the derivatives of $\mathbf{f}$ and $\mathbf{g}$ of orders $1, \ldots, n$.

Remark: If the function $\mathbf{f}$ is an affine mapping, then its derivatives of order higher than 1 are 0. In that case the matrices A_{ij} for $j < i$ are zero matrices and it is better (namely, a bit faster) to compute the derivatives of the n-th order of $\mathbf{h}$ by computing the matrix A_{nn} (with the appropriate procedure described in Section 3.11.1), and by multiplying it by the matrix, whose rows are the appropriate derivatives of the n-th order of the function $\mathbf{g}$.

3.11.3 Computing derivatives of compositions with inverse functions

```
void pkn_Comp2iDerivatives1f (
        float xu, float yu, float xv, float yv,
        int spdimen, const float *hu, const float *hv,
        float *gx, float *gy );
void pkn_Comp2iDerivatives2f (
        float xu, float yu, float xv, float yv,
        float xuu, float yuu, float xuv,
        float yuv, float xvv, float yvv,
        int spdimen, const float *hu, const float *hv,
        const float *huu, const float *huv, const float *hvv,
        float *gx, float *gy, float *gxx, float *gxy, float *gyy );
```



```
void pkn_Comp2iDerivatives3f (
        float xu, float yu, float xv, float yv,
        float xuu, float yuu, float xuv,
        float yuv, float xvv, float yvv,
        float xuuu, float yuuu, float xuuv, float yuuv,
        float xuvv, float yuvv, float xvvv, float yvvv,
        int spdimen, const float *hu, const float *hv,
        const float *huu, const float *huv, const float *hvv,
        const float *huuu, const float *huuv,
        const float *huvv, const float *hvvv,
        float *gx, float *gy, float *gxx, float *gxy, float *gyy,
        float *gxxx, float *gxxy, float *gxyy, float *gyyy );
void pkn_Comp2iDerivatives4f (
        float xu, float yu, float xv, float yv,
        float xuu, float yuu, float xuv,
        float yuv, float xvv, float yvv,
        float xuuu, float yuuu, float xuuv, float yuuv,
        float xuvv, float yuvv, float xvvv, float yvvv,
        float xuuuu, float yuuuu, float xuuuv,
        float yuuuv, float xuuvv, float yuuvv,
        float xuvvv, float yuvvv, float xvvvv, float yvvvv,
        int spdimen, const float *hu, const float *hv,
        const float *huu, const float *huv, const float *hvv,
        const float *huuu, const float *huuv, const float *huvv,
        const float *hvvv, const float *huuuu, const float *huuuv,
        const float *huuvv, const float *huvvv, const float *hvvvv,
        float *gx, float *gy, float *gxx, float *gxy, float *gyy,
        float *gxxx, float *gxxy, float *gxyy, float *gyyy,
        float *gxxxx, float *gxxxy, float *gxxyy,
        float *gxyyy, float *gyyyy );
```

The above procedures compute the partial derivatives of the function $\mathbf{g} = \mathbf{f}^{-1} \circ \mathbf{h}$, which is the composition of a function $\mathbf{f}^{-1}\colon \mathbb{R}^2 \to \mathbb{R}^2$ with $\mathbf{h}\colon \mathbb{R}^2 \to \mathbb{R}^d$. The function $\mathbf{f}$, given by two scalar functions, $x(u,v)$ and $y(u,v)$, must be regular (i.e. its partial derivative vectors of the first order must be linearly independent). Moreover, the functions $\mathbf{f}$ and $\mathbf{h}$ must be smooth enough.

The parameter spdimen of all the procedures specifies the dimension d of the space, whose elements are the values of $\mathbf{g}$ and $\mathbf{h}$. The other parameters have names, which explain their meanings. For example the parameter yu specifies the value of $y_u = \frac{\partial y}{\partial u}$ etc. Similarly, the parameter huv is the pointer to the array with the d coordinates of the vector $\mathbf{h}_{uv} = \frac{\partial^2 \mathbf{h}}{\partial u \partial v}$, and the parameter gyyyy points to the array, in which the d coordinates of the vector $\mathbf{g}_{yyyy} = \frac{\partial^4 \mathbf{g}}{\partial y^4}$ are to be stored by the procedure.

The algorithm is based on the interpretation of Formulae (3.1)–(3.4)) as systems of linear equations with unknown derivatives of the function $\mathbf{g}$. These equations are solved with the procedure `pkn_multiGaussSolveLinEqf`, which is an implementation of the Gaussian elimination method with full pivoting. Because the computation of the derivatives of order n of the function $\mathbf{g}$ must be preceded by computing the derivatives of order lower than n, the procedures have the parameters — pointers to the arrays intended to store all these derivatives (this is different than with the procedures described in the previous section).

3.11.4 Computing derivatives of inverse functions

```
void pkn_f2iDerivatives1f (
        float xu, float yu, float xv, float yv,
        float *gx, float *gy );
void pkn_f2iDerivatives2f (
        float xu, float yu, float xv, float yv,
        float xuu, float yuu, float xuv,
        float yuv, float xvv, float yvv,
        float *gx, float *gy, float *gxx, float *gxy, float *gyy );
void pkn_f2iDerivatives3f (
        float xu, float yu, float xv, float yv,
        float xuu, float yuu, float xuv,
        float yuv, float xvv, float yvv,
        float xuuu, float yuuu, float xuuv, float yuuv,
        float xuvv, float yuvv, float xvvv, float yvvv,
        float *gx, float *gy, float *gxx, float *gxy, float *gyy,
        float *gxxx, float *gxxy, float *gxyy, float *gyyy );
void pkn_f2iDerivatives4f (
        float xu, float yu, float xv, float yv,
        float xuu, float yuu, float xuv,
        float yuv, float xvv, float yvv,
        float xuuu, float yuuu, float xuuv, float yuuv,
        float xuvv, float yuvv, float xvvv, float yvvv,
        float xuuuu, float yuuuu, float xuuuv,
        float yuuuv, float xuuvv, float yuuvv,
        float xuvvv, float yuvvv, float xvvvv, float yvvvv,
        float *gx, float *gy, float *gxx, float *gxy, float *gyy,
        float *gxxx, float *gxxy, float *gxyy, float *gyyy,
        float *gxxxx, float *gxxxy, float *gxxyy,
        float *gxyyy, float *gyyyy );
```

The above procedures compute the partial derivatives of the function $\mathbf{g} = \mathbf{f}^{-1}$, where $\mathbf{f}\colon \mathbb{R}^2 \to \mathbb{R}^2$ is a regular and sufficiently smooth function, described by

two scalar functions, $x(u,v)$ and $y(u,v)$. The actual computation is done by the procedures described in the previous section, which compute the derivatives of the composition of $\mathbf{f}^{-1}$ with the function $\mathbf{h}$, being the identity mapping of $\mathbb{R}^2$.

3.12 Quadratures

```
boolean pkn_QuadRectanglesf ( float a, float b, int n,
                          float *qknots, float *qcoeff );
```

```
boolean pkn_QuadSimpsonf ( float a, float b, int n,
                        float *qknots, float *qcoeff );
```

```
boolean pkn_QuadGaussLegendre4f ( float a, float b, int n,
                              float *qknots, float *qcoeff );
boolean pkn_QuadGaussLegendre6f ( float a, float b, int n,
                              float *qknots, float *qcoeff );
boolean pkn_QuadGaussLegendre8f ( float a, float b, int n,
                              float *qknots, float *qcoeff );
boolean pkn_QuadGaussLegendre10f ( float a, float b, int n,
                               float *qknots, float *qcoeff );
boolean pkn_QuadGaussLegendre12f ( float a, float b, int n,
                               float *qknots, float *qcoeff );
boolean pkn_QuadGaussLegendre14f ( float a, float b, int n,
                               float *qknots, float *qcoeff );
boolean pkn_QuadGaussLegendre16f ( float a, float b, int n,
                               float *qknots, float *qcoeff );
boolean pkn_QuadGaussLegendre18f ( float a, float b, int n,
                               float *qknots, float *qcoeff );
boolean pkn_QuadGaussLegendre20f ( float a, float b, int n,
                               float *qknots, float *qcoeff );
```

4. The libpkgeom library

The libpkgeom library consists of procedures, which implement basic operations on points and vectors in two-, three- and four-dimensional spaces. The operations are: addition, subtraction, multiplication, interpolation and affine transformations. In addition, there is a procedure of computing the convex hull of a set of points in the plane. Other procedures of computational geometry will also be placed in this library.

All names of data types and procedures end with the letter f or d, which indicates the representation of coordinates — of single (float) or double (double) precision.

4.1 Point and vector operations

```
typedef struct point2f {
  float x, y;
} point2f vector2f;

typedef struct point3f {
  float x, y, z;
} point3f vector3f;

typedef struct point4f {
  float X, Y, Z, W;
} point4f vector4f;
```

Points and vectors are represented with pairs, triples and quadruples of numbers. An essential property of these representations is the absence of any additional data. Due to that, for example an array of n points in a plane may be passed to a procedure, which processes an array of $2n$ numbers. Therefore these structures should not be converted to C++ classes, and in particular no classes with additional attributes may be defined.

A structure of type point3f may represent a point in the 3D space or a point of a plane. In the latter case the fields x, y, z describe homogeneous coordinates of this point — its cartesian coordinates are equal to x/z and y/z. Analoguously, a structure of type point4f consists of fields, whose values are homogeneous coordinates of a point in the three-dimensional space.


```
typedef struct ray3f {
  point3f p;
  vector3f v;
} ray3f;
```

Structures of type ray3f represent rays, i.e. halflines in $\mathbb{R}^3$, with the origin at the point p and with the direction described by the vector v.

```
typedef union trans2f {
  struct {
    float a11, a12, a13;
    float a21, a22, a23;
  } U0;
  struct {
    float a[2][3];
    short detsgn;
  } U1;
} trans2f;

typedef union trans3f {
  struct {
    float a11, a12, a13, a14;
    float a21, a22, a23, a24;
    float a31, a32, a33, a34;
  } U0;
  struct {
    float a[3][4];
    short detsgn;
  } U1;
} trans3f;
```

Structures of type trans2f and trans3f represent affine transformations of the two- and three-dimensional spaces. The representation consists of a matrix 3×3 or 4×4, whose last row is either $[0, 0, 1]$ or $[0, 0, 0, 1]$. Therefore this row is not stored. The field detsgn describes the sign of the determinant of this matrix.

```
void SetPoint2f ( point2f *p, float x, float y );
#define SetVector2f(v,x,y) SetPoint2f ( v, x, y )
void SetPoint3f ( point3f *p, float x, float y, float z );
#define SetVector3f(v,x,y,z) SetPoint3f ( v, x, y, z )
void SetPoint4f ( point4f *p, float X, float Y, float Z, float W );
#define SetVector4f(v,X,Y,Z,W) SetPoint4f ( v, X, Y, Z, W )
```

The above procedures and macros initialize point and vector representations.

```
void TransPoint2f ( const trans2f *tr, const point2f *p,
                    point2f *q );
void TransPoint3f ( const trans3f *tr, const point3f *p,
                    point3f *q );
```

The above procedures compute the image $\mathbf{q}$ of a point $\mathbf{p}$ in an affine transformation of the two- or three-dimensional space.

```
void TransVector2f ( const trans2f *tr, const vector2f *v,
                     vector2f *w );
void TransVector3f ( const trans3f *tr, const vector3f *v,
                     vector3f *w );
```

The above procedures compute the image $\boldsymbol{w}$ of the vector $\boldsymbol{v}$ in a linear transformation, which is the linear part of the affine transformation represented by the variable `*tr`.

```
void TransContra3f ( const trans3f *tri, const vector3f *v,
                     vector3f *w );
```

The procedure `TransContra3f` computes the image $\boldsymbol{w}$ of the vector $\boldsymbol{v}$ in a linear transformation, whose matrix is the transposition of the matrix of the linear part of the affine transformation represented by the parameter `*tri`. If the vector $\boldsymbol{v}$ is the normal vector of some plane π, and the transformation represented by `*tri` is the *inverse* of some transformation A, then the computed vector $\boldsymbol{w}$ is the normal vector of the plane $A(\pi)$.

```
void Trans3Point2f ( const trans3f *tr, const point2f *p,
                     point2f *q );
```

The procedure `Trans3Point2f` applies the affine transformation `*tr` to the point $\mathbf{p} \in \mathbb{R}^3$, whose first two coordinates are the values of the fields `x` and `y` of the parameter `*p`, and the third coordinate is 0. The coordinates `x` and `y` of the image are assigned to the appropriate fields of the parameter `*q`.

```
void Trans2Point3f ( const trans2f *tr, const point3f *p,
                     point3f *q );
```

The procedure `Trans2Point3f` computes the image of a point $\mathbf{p} \in \mathbb{R}^2$, represented by homogeneous coordinates, in an affine transformation.

```
void Trans3Point4f ( const trans3f *tr, const point4f *p,
                     point4f *q );
```

The procedure `Trans3Point4f` applies the affine transformation `*tr` to the point $\mathbf{p} \in \mathbb{R}^3$, whose four homogeneous coordinates are the values of the fields of the parameter `*p`.

The homogeneous coordinates of the image (such that the weight coordinates of the point and its image are the same) are assigned to the appropriate fields of the parameter `*q`.

```
void IdentTrans2f ( trans2f *tr );
void IdentTrans3f ( trans3f *tr );
```

The procedures `IdentTrans2f` and `IdentTrans3f` initialize the structures `*tr` to the values representing the identity mappings of the two- and three-dimensional spaces respectively.

```
void CompTrans2f ( trans2f *s, trans2f *t, trans2f *u );
void CompTrans3f ( trans3f *s, trans3f *t, trans3f *u );
```

The procedures `CompTrans2f` and `CompTrans3f` compute the composition of the affine transformations represented by the parameters `*t` and `*u`, and assign it to the parameter `*s`. This composition is equivalent to the transformation `*u` *followed by* `*t`.

```
void GeneralAffineTrans3f ( trans3f *tr,
                            vector3f *v1, vector3f *v2, vector3f *v3 );
```

The procedure `GeneralAffineTrans3f` computes the composition of the transformation represented by the parameter `*tr` with the transformation, whose linear part is represented by the matrix $[\boldsymbol{v}_1, \boldsymbol{v}_2, \boldsymbol{v}_3]$ (and the translation vector is $\mathbf{0}$). The composition is assigned to the parameter `*tr`.

```
void ShiftTrans2f ( trans2f *tr, float tx, float ty );
void ShiftTrans3f ( trans3f *tr, float tx, float ty, float tz );
```

The procedures `ShiftTrans2f` and `ShiftTrans3f` compute the composition of the transformation represented by the parameter `*tr` and the translation by the vector $[t_x, t_y]^T$ or $[t_x, t_y, t_z]^T$. The composition is assigned to the parameter `*tr`.

```
void RotTrans2f ( trans2f *tr, float angle );
```

The procedure `RotTrans2f` computes the composition of the affine transformation represented by `*tr` with the rotation around the point $[0, 0]^T$ by the angle `angle`. The composition is assigned to the parameter `*tr`.

```
void Rot3f ( trans3f *tr, byte j, byte k, float angle );
```

The procedure `Rot3f` computes the composition of the transformation represented by the parameter `*tr` with the rotation around one of the axes of the system of coordinates. The axis is specified by the parameters `j` and `k`, which must be different numbers from the set $\{1, 2, 3\}$. For example the rotation in the plane xy (around the z axis) corresponds to $j = 1$, $k = 2$. The rotation angle is equal to `angle`. The composition is assigned to the parameter `*tr`.

```
#define RotXTrans3f(tr,angle) Rot3f ( tr, 2, 3, angle )
#define RotYTrans3f(tr,angle) Rot3f ( tr, 3, 1, angle )
#define RotZTrans3f(tr,angle) Rot3f ( tr, 1, 2, angle )
```

The above macros call the procedure `Rot3f` in order to compute the composition of the affine transformation represented by the parameter `*tr` with a rotation around the x, y, z axes, i.e. in the planes yz, zx and xy respectively.

```
void RotVTrans3f ( trans3f *tr, vector3f *v, float angle );
```

The procedure `RotVTrans3f` computes the composition of the affine transformation represented by the parameter `*tr` with the rotation around the line, which passes through the point $[0,0,0]^T$ and has the direction of the *unit* vector $\boldsymbol{\nu}$, by the angle `angle`. The composition is assigned to the parameter `*tr`.

```
void FindRotVEulerf ( const vector3f *v, float angle,
                      float *psi, float *theta, float *phi );
```

The procedure `FindRotVEulerf` computes the Euler angles (precession `*psi`, nutation `*theta` and revolution `*phi`), representing the rotation around the line, whose direction is specified by the *unit* vector $\boldsymbol{\nu}$ by the angle `angle`.

```
float TrimAnglef ( float angle );
```

The procedure `TrimAnglef` returns the number α, which is an element of the interval $[-\pi,\pi]$, and which differs from the parameter `angle` by an integer multiplicity of 2π plus the rounding error.

```
void CompEulerRotf ( float psi1, float theta1, float phi1,
                     float psi2, float theta2, float phi2,
                     float *psi, float *theta, float *phi );
```

The procedure `CompEulerRotf` computes the Euler angles ψ, θ, φ of the rotation, which is the composition of two rotations represented by the Euler angles ψ_1, θ_1, φ_1 and ψ_2, θ_2, φ_2 respectively.

```
void CompRotV3f ( const vector3f *v1, float a1,
                  const vector3f *v2, float a2,
                  vector3f *v, float *a );
```

The procedure `CompRotV3f` computes the composition of two rotations in $\mathbb{R}^3$, given by unit vectors of their axes, $\boldsymbol{\nu}_1$, $\boldsymbol{\nu}_2$ and the angles α_1, α_2. The procedure computes the vector $\boldsymbol{\nu}$ of the composition axis and the angle α.

```
void EulerRotTrans3f ( trans3f *tr,
                       float psi, float theta, float phi );
```

The procedure `EulerRotTrans3f` computes the composition of the affine transformation represented by the initial value of the parameter `*tr` with the rotation represented by the Euler angles ψ, θ, φ.


```
void ScaleTrans2f ( trans2f *t, float sx, float sy );
void ScaleTrans3f ( trans3f *tr, float sx, float sy, float sz );
```

The procedures `ScaleTrans2f` and `ScaleTrans3f` compute the composition of the affine transformation represented by the initial value of the parameter `*tr` with the scaling, whose coefficients are s_x and s_y or s_x, s_y and s_z.

```
void MirrorTrans3f ( trans3f *tr, vector3f *n );
```

The procedure `MirrorTrans3f` computes the composition of the affine transformation represented by the initial value of the parameter `*tr` with the symmetric reflection with respect to the plane, which contains the origin of the coordinate system and whose normal vector is $\mathbf{n}$.

```
boolean InvertTrans2f ( trans2f *tr );
boolean InvertTrans3f ( trans3f *tr );
```

The procedures `InvertTrans2f` and `InvertTrans3f` compute the inversion of the affine transformation represented by the initial value of the parameter `*tr`, if it exists. In this case the procedure returns `true`, otherwise it returns `false`.

```
void MultVector2f ( double a, const vector2f *v, vector2f *w );
void MultVector3f ( double a, const vector3f *v, vector3f *w );
void MultVector4f ( double a, const vector4f *v, vector4f *w );
```

The above procedures compute the vector $\boldsymbol{w} = a\boldsymbol{\nu}$.

```
void AddVector2f ( const point2f *p, const vector2f *v,
                   point2f *q );
void AddVector3f ( const point3f *p, const vector3f *v,
                   point3f *q );
```

The above procedures compute the point $\mathbf{q} = \mathbf{p} + \boldsymbol{\nu}$.

```
void AddVector2Mf ( const point2f *p, const vector2f *v, double t,
                    point2f *q );
void AddVector3Mf ( const point3f *p, const vector3f *v, double t,
                    point3f *q );
```

The above procedures compute the point $\mathbf{q} = \mathbf{p} + t\boldsymbol{\nu}$.

```
void SubtractPoints2f ( const point2f *p1, const point2f *p2,
                        vector2f *v );
void SubtractPoints3f ( const point3f *p1, const point3f *p2,
                        vector3f *v );
void SubtractPoints4f ( const point4f *p1, const point4f *p2,
                        vector4f *v );
```

The above procedures compute the vector $\boldsymbol{\nu} = \mathbf{p}_1 - \mathbf{p}_2$.

```
void InterPoint2f ( const point2f *p1, const point2f *p2, double t,
                    point2f *q );
void InterPoint3f ( const point3f *p1, const point3f *p2, double t,
                    point3f *q );
void InterPoint4f ( const point4f *p1, const point4f *p2, double t,
                    point4f *q );
```

The above procedures compute the point $\mathbf{q} = \mathbf{p}_1 + t(\mathbf{p}_2 - \mathbf{p}_1)$.

```
void MidPoint2f ( const point2f *p1, const point2f *p2,
                  point2f *q );
void MidPoint3f ( const point3f *p1, const point3f *p2,
                  point3f *q );
void MidPoint4f ( const point4f *p1, const point4f *p2,
                  point4f *q );
```

The above procedures compute the point $\mathbf{q} = \frac{1}{2}(\mathbf{p}_1 + \mathbf{p}_2)$.

```
void Interp3Vectors2f ( const vector2f *p0, const vector2f *p1,
                        const vector2f *p2,
                        const float *coeff, vector2f *p );
void Interp3Vectors3f ( const vector3f *p0, const vector3f *p1,
                        const vector3f *p2,
                        const float *coeff, vector3f *p );
void Interp3Vectors4f ( const vector4f *p0, const vector4f *p1,
                        const vector4f *p2,
                        const float *coeff, vector4f *p );
```

The above procedures compute the linear combination of three vectors given as parameters; the coefficients of the combination are given in the array `coeff`.

```
void NormalizeVector2f ( vector2f *v );
void NormalizeVector3f ( vector3f *v );
```

The above procedures compute $*v := \frac{1}{\|\boldsymbol{v}\|_2}\boldsymbol{v}$.

```
double DotProduct2f ( const vector2f *v1, const vector2f *v2 );
double DotProduct3f ( const vector3f *v1, const vector3f *v2 );
double DotProduct4f ( const vector4f *v0, const vector4f *v1 );
```

The above procedures compute the appropriate scalar products.


```
double det2f ( const vector2f *v1, const vector2f *v2 );
double det3f ( const vector3f *v1, const vector3f *v2,
               const vector3f *v3 );
double det4f ( const vector4f *v0, const vector4f *v1,
               const vector4f *v2, const vector4f *v3 );
```

The above procedures compute the determinants of the matrices 2×2, 3×3 and 4×4 respectively, whose columns are given as the parameters.

```
void Point3to2f ( const point3f *P, point2f *p );
void Point4to3f ( const point4f *P, point3f *p );
```

The above procedures compute the cartesian coordinates of a point $\mathbf{p}$ based on its homogeneous coordinates.

```
void Point2to3f ( const point2f *p, float w, point3f *P );
void Point3to4f ( const point3f *p, float w, point4f *P );
```

The above procedures compute the homogeneous coordinates of a point $\mathbf{p}$ with the weight coordinate w, based on the cartesian coordinates.

```
void CrossProduct3f ( const vector3f *v1, const vector3f *v2,
                      vector3f *v );
```

The procedure `CrossProduct3f` computes the vector product of the vectors $\boldsymbol{v}_1$ and $\boldsymbol{v}_2$.

```
void OrtVector2f ( const vector2f *v1, const vector2f *v2,
                   vector2f *v );
void OrtVector3f ( const vector3f *v1, const vector3f *v2,
                   vector3f *v );
```

The procedures `OrtVector2f` and `OrtVector3f` compute the vector $\boldsymbol{v} = \boldsymbol{v}_2 - \frac{\langle \boldsymbol{v}_1, \boldsymbol{v}_2 \rangle}{\langle \boldsymbol{v}_1, \boldsymbol{v}_1 \rangle}\boldsymbol{v}_1$.

```
void CrossProduct4P3f ( const vector4f *v0, const vector4f *v1,
                        const vector4f *v2, vector3f *v );
```

The procedure `CrossProduct4P3f` computes the first three coordinates of the vector in $\mathbb{R}^4$, which is the vector product of three vectors $\boldsymbol{v}_1$, $\boldsymbol{v}_2$ and $\boldsymbol{v}_3$.

```
void OutProduct4P3f ( const vector4f *v0, const vector4f *v1,
                      vector3f *v );
```

The procedure `OutProduct4P3f` computes the vector

$$\boldsymbol{v} = \begin{bmatrix} X_0 W_1 - W_0 X_1 \\ Y_0 W_1 - W_0 Y_1 \\ Z_0 W_1 - W_0 Z_1 \end{bmatrix}.$$

4.2 Boxes

Rectangles and rectangular parallelpipeds are useful in various applications, especially for estimating the locations of more complicated geometrical figures. The types defined below describe such boxes. In the future basic procedures of processing such boxes will be developed as a part of the `libpkgeom` library.

```
typedef struct Box2f {
    float x0, x1, y0, y1;
  } Box2f;

typedef struct Box3f {
    float x0, x1, y0, y1, z0, z1;
  } Box3f;
```

4.3 Finding the convex hull

The headers of the procedures of finding the convex hull (for both versions: of the single and double precision) are in the file `convh.h`.

```
void FindConvexHull2f ( int *n, point2f *p );
```

The procedure `FindConvexHull2f` finds the convex hull of a set of `n` points of the plane. These points are given in the array `p`. The initial value of the parameter `*n` is the number of the points. The final contents of the array consists of some of the points, namely the subsequent vertices of the polygon being the convex hull of the points. The number of vertices of the hull is the final value of the parameter `*n`.

5. The libcamera library

The libcamera library consists of procedures, which manage the cameras, i.e. objects, which represent projections of the 3d space onto a plane, in order to make pictures. There are two kinds of projections: perspective and parallel. The former are intended to make "photographs", the latter are better to produce technical drawings.

5.1 The camera

5.1.1 A description of the camera and the projection algorithm

The data structure and the headers of procedures are giben in the header files cameraf.h andf camerad.h. Both files may be included via the file camera.h.

```
typedef struct CameraRecf {
  boolean parallel, upside, c_fixed;
  byte magnification;
  short xmin, ymin, width, height;
  float aspect;
  point3f position;
  float psi, theta, phi;
  point3f g_centre, c_centre;
  float xscale, yscale;
  trans3f CTr, CTrInv;
  vector4f cplane[6];
  union {
    struct {
      float f;
      float xi0, eta0;
      float dxi0, deta0;
    } persp;
    struct {
      float wdt, hgh, diag;
      boolean dim_case;
    } para;
  } vd;
} CameraRecf;
```

The CameraRecf structure describes the camera, i.e. an object representing a perspective or a parallel projection.

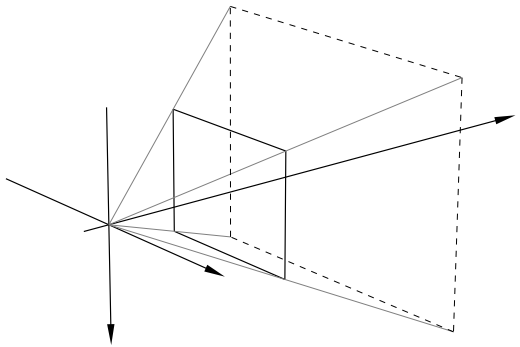


Figure 5.1. The camera system of coordinates and the frame for a perspective projection

The structure fields contain the following information:

parallel — if its value is false (0), then the projection is perspective, else it is parallel.

upside — if its value is false, then the y axis of the image system of coordinates is oriented downward (like in XWindow system windows), else it is oriented up (like in the OpenGL library or in the default PostScript system of coordinates).

c_fixed — this parameter specifies the changes of the camera rotations centre when the camera is moved — if false, then this point is fixed in the global system of coordinates, else if is fixed in the camera system.

magnification — by default this field has the value 1, which means that the axis unit in the image system of coordinate is the width or height of one pixel. Greater values select appropriately shorter units, which may be useful for supersampling.

xmin, ymin, width, height — coordinates of the upper left point of the frame and its dimensions in pixels.

aspect — the aspect factor, i.e. the ratio of the width and height of one pixel.

position — position of the view point (in global coordinates).

psi, theta, phi — Euler angles ψ, ϑ, φ, which describe the direction of the camera.

g_centre, c_centre — coordinates of the centre of rotations of the camera, i.e. the point on the axes of rotations of the camera, in global and camera coordinates respectively.

xscale, yscale — factors of scaling of the axes x and y of the camera system of coordinates.

Ctr, CTrInv — the transformation from the global to camera coordinates and its inverse.

`cplane` — representations of four halfspaces, ahose intersection is the visibility frustum. The halfspace $ax+by+cz+d>0$ is represented by the vector, whose coordinates are `a`, `b`, `c`, `d`.

So far only four halfspaces are used, the other two (near and far) are to be done.

`vd` — a union with data specific for the methods of projection. The structure `vd.persp` contains the data specific for perspective projections, while `vd.para` for the parallel ones.

`vd.persp.f` — focal length of the camera, in the units such that the diagonal of the frame has the length 1. Focal length 1 corresponds to a standard photographic objective.

`vd.persp.xi0`, `vd.persp.eta0` — shift of pixels after the perspective projection.

`vd.persp.dxi0`, `vd.persp.deta0` — coordinates `x`, `y` (in the camera system) of the frame centre. By default they are zero, and then the frame centre is located on the "optical axis" of the camera. Other values are necessary with cameras forming a stereo pair.

`vd.para.wdt`, `vd.para.hgh`, `vd.para.diag` — dimensions (width, height, diagonal) of the frame, measured in the units of the global system of coordinates.

`vd.para.dim_case` — the parameter, which specifies, which of the three above dimensions of the frame is specified by the user; 0 — diagonal, 1 — width, 2 — height. The other two dimensions will be computed by the library procedures.

By default this parameter obtains the value 0, and the parameter `vd.para.diag` is set to 1.

Projection algorithm:

The image of the point $\mathbf{p}$, represented in the global coordinates, is computed as follows:

1. The point $\mathbf{p}$ is subject to the affine transformation represented by the `CTr` attribute of the camera. In this way the camera coordinates are obtained.
2. For a perspective projection the `x` and `y` coordinates are divided by the `z` coordinate. Then the values of `xi0` and `eta0` are added to the quotients.

   For parallel projections this step is omitted.
3. If the `y` axis is oriented upward (the `upside` attribute is nonzero) then the `y` coordinate is replaced by $2y_{min}+h-y$, where y_{min} is the value of the attribute `ymin`, and `h` is the value of the attribute `height`.

Setting up the transformation to the camera coordinates:

The transformation from the global to camera coordinates (done in the first step of the algorithm describd above) is the composition of thee affine transformations:

1. Scaling of the axes `x` and `y` by the factors, being values of the attributes `xscale` and `yscale`.
2. Rottaion described with use of the Euler angles, being values of the attributes `psi`, `theta`, `phi`.
3. Translation, which sets the origin of the system at the point `position`.

The values of the attributes, which specify the above transformations, should be assigned by the procedures described later. These procedures may compute the proper values representing a composition of a series of camera movements from the default initial position.

5.1.2 Camera procedures

```
void CameraInitFramef ( CameraRecf *CPos,
                boolean parallel, boolean upside,
                short width, short height, short xmin, short ymin,
                float aspect );
```

The procedure `CameraInitFramef` sets initial values of the `*CPos` attributes, which specify the kind of projection and the size of the frame (in pixels), and the aspect factor (ratio of the width and height of one pixel).

The parameter `parallel` equal to `false` determines the prespective projection, its value `true` results in a paralell projection.

The parameter `upside` equal to `false` causes assuming the downward orientation of the `y` axis of the image, its value `true` — upward.

The parameters `width` and `height` describe the width and height of the frame (in pixels), and the parameters `xmin` and `ymin` the position of the upper left corner.

The call to this procedure should precede calling all other actions with a camera, but it is *insufficient* to fully specify the projection. This must be done by calling `CameraInitPosf` and perhaps a number of calls to the procedures changing the camera position. If the frame size is to be changed without changing the current position (e.g. after changing the size of a program window), after calling `CameraInitFramef` it is necessary to call the procedure `CameraSetMappingf`.

```
void CameraSetMagf ( CameraRecf *CPos, byte mag );
```

The procedure `CameraSetMagf` sets the magnification factor, e.g. for supersampling. The default units of the image axes are the width and height of a pixel. By calling this procedure with the parameter `mag` $= n$ (where n is a positive integer),

we decrease these units n times, which may be useful during an image synthesis with supersampling.

```
void CameraSetMappingf ( CameraRecf *CPos );
```

The procedure CameraSetMappingf computes the transformation matrices between the global and camera coordinate systems. This procedure is called by all procedures setting or changing the camera position, and therefore calling it directly from applications is usually unnecessary. One exception is after changing the size of the frame (with use of CameraInitFramef) and the current camera position is to be left unchanged.

```
void CameraProjectPoint3f ( CameraRecf *CPos, const point3f *p,
                            point3f *q );
```

The procedure CameraProjectPoint3f computes the image of a point *p in a perspective or parallel projection. The coordinates of this image are assigned to the attributes x and y of the parameter *q. Its z attribute is the depth of the point, i.e. its signed distance from the plane contatining the camera position and parallel to the projection plane. This may be needed by a hidden line or surface algorithm.

```
void CameraUnProjectPoint3f ( CameraRecf *CPos, const point3f *p,
                              point3f *q );
```

The procedure CameraUnProjectPoint3f computes the counterimage of a point p. The coordinates x, y of the point *q are specified in the image coordinates, the z coordinate is the depth (in the camera system). This is thus the inversion of the transformation computed by the procedure CameraProjectPoint3f.

The coordinates x, y, z of the counterimage are assigned to the appropriate attributes of the parameter *q.

```
void CameraProjectPoint2f ( CameraRecf *CPos, const point2f *p,
                            point2f *q );
```

The procedure CameraProjectPoint2f computes a projection of the point $\mathbf{p}$, whose coordinates x, y are these of the parameter p, and the coordinate z is 0.

The coordinates x, y of the image are assigned to the attributes of the parameter *q.

In principle using this procedure makes sense only with parallel projections.

```
void CameraUnProjectPoint2f ( CameraRecf *CPos, const point2f *p,
                              point2f *q );
```

The procedure CameraUnProjectPoint2f computes a counterimage of the point $\mathbf{p}$, whose coordinates x, y (in the image system) are given by the parameter p, and the z coordinate (in the camera system) is 0.

The coordinates of the counterimage are assigned to the attributes of the parameter *q.

This procedure may be used only with parallel projections.

```
void CameraRayOfPixelf ( CameraRecf *CPos, float xi, float eta,
                         ray3f *ray );
```

The procedure CameraRayOfPixel for a point of the projection plane, whose image coordinates are x = xi, y = eta, finds the representation of a ray, i.e. a halfline, whose origin (for a perspective projection) is the viewer position, and which intersect the projection plane at that point. For a parallel projection the ray origin is that point and its direction is the projection direction.

The ray origin is assigned to the p attribute of the structure *ray and the unit vector, representing the ray direction is assigned to the attribute v. The ray is specified in the global system of coordinates. The main use of this procedure is ray tracing.

```
void CameraInitPosf ( CameraRecf *CPos );
```

The procedure CameraInitPosf sets the camera to the default initial position, in which the axes x, y and z of the camera system of coordinates coincide with the axes x, y, z of the global system. The focal length is set to 1. Before calling this procedure it is necessary to specify the frame dimensions and the image aspect, by calling CameraInitFramef.

After calling CameraInitPosf the camera is ready to projecting points and to the manipulations with the position, direction and the focal length.

```
void CameraSetRotCentref ( CameraRecf *CPos, point3f *centre,
                           boolean global_coord, boolean global_fixed );
```

The procedure SetCameraRotCentref may be used to specify the point of axes of rotations of the camera. The parameter *centre is this point, global_coord specifies, whether its coordinates are specified in the global system (true) or in the camera system (false). The parameter global_fixed specifies, whether this point is fixed in the global (true), or in the camera (false) system, when the camera is moved.

```
void CameraMoveToGf ( CameraRecf *CPos, point3f *pos );
```

The procedure CameraMoveToGf translates (without rotation) the camera to the position *pos, specified in the global system.

```
void CameraTurnGf ( CameraRecf *CPos,
                    float psi, float theta, float phi );
```

The procedure CameraTurnGf sets the camera orientation specified by the Euler angles (precession, psi, nutation, theta, and revolution phi), in the global coordinate system.

Remark: The way of representing the camera orientation will some day be changed, and using this procedure is therefore *not recommended*.

```
void CameraMoveGf ( CameraRecf *CPos, vector3f *v );
```

The procedure `CameraMoveGf` translates the camera by the vector `v`, specified in the global system of coordinates.

```
void CameraMoveCf ( CameraRecf *CPos, vector3f *v );
```

The procedure `CameraMoveGf` translates the camera by the vector `v`, specified in the camera system of coordinates.

```
void CameraRotGf ( CameraRecf *CPos,
                   float psi, float theta, float phi );
```

The procedure `CameraRotGf` turns the camera. The rotation is specified by the Euler angles in the global system of coordinates. The axis of the rotation passes through the point set with the procedure `SetCameraRotCentref` (default is the origin of the global system of coordinates).

```
#define CameraRotXGf(Camera,angle) \
  CameraRotGf(Camera, 0.0, angle, 0.0)
#define CameraRotYGf(Camera,angle) \
  CameraRotGf(Camera, 0.5 * PI, angle, -0.5 * PI)
#define CameraRotZGf(Camera,angle) \
  CameraRotGf(Camera, angle, 0.0, 0.0)
```

Three macrodefinitions, which turn the camera around the three axes of the global system of coordinates.

```
void CameraRotVGf ( CameraRecf *CPos, vector3f *v, float angle );
```

The procedure `CameraRotVGf` turns the camera around the axis, whose direction is that of the vector `v`, by the angle `angle`. The coordinates of the vector `v` are specified in the global system of coordinates.

```
void CameraRotCf ( CameraRecf *CPos,
                   float psi, float theta, float phi );
```

The procedure `CameraRotCf` turns the camera. The rotation is specified by the Euler angles in the camera system of coordinates. The axis of the rotation passes through the point set with the procedure `SetCameraRotCentref` (default is the origin of the global system of coordinates).


```
#define CameraRotXCf(Camera,angle) \
  CameraRotCf ( Camera, 0.0, angle, 0.0 )
#define CameraRotYCf(Camera,angle) \
  CameraRotCf ( Camera, 0.5 * PI, angle, -0.5 * PI )
#define CameraRotZCf(Camera,angle) \
  CameraRotCf ( Camera, angle, 0.0, 0.0 )
```

Three macrodefinitions, which turn the camera around the three axes of the camera system of coordinates.

```
void CameraRotVCf ( CameraRecf *CPos, vector3f *v, float angle );
```

The procedure `CameraRotVCf` turns the camera around the axis, whose direction is that of the vector `v`, by the angle `angle`. The coordinates of the vector `v` are specified in the camera system of coordinates.

```
void CameraSetFf ( CameraRecf *CPos, float f );
```

The procedure `CameraSetFf` sets the focal length of the camera.

```
void CameraZoomf ( CameraRecf *CPos, float fchange );
```

The procedure `CameraZoomf` changes the focal length of the camera by the factor `fchange`, which must be positive.

```
boolean CameraClipPoint3f ( CameraRecf *CPos,
                            point3f *p, point3f *q );
```

The procedure `CameraClipPoint3f` checks, whether the image of the point `p` fits into the frame and if it does, then it computes the image. Its coordinates are passed using the parameter `q`. The value `true` indicates that the image has been computed, `false` is returned for points outside the visibility pyramid.

```
boolean CameraClipLine3f ( CameraRecf *CPos,
                      point3f *p0, float t0, point3f *p1, float t1,
                      point3f *q0, point3f *q1 );
```

The procedure `CameraClipLine3f` clips the line segment $\{(1-t)\mathbf{p}_0 + t\mathbf{p}_1 : t \in [t_0, t_1]\}$ to the visibility pyramid. If the intersection is nonempty, its end points are projected and returned with use of the parameters `q0` and `q1`. The procedure value is then `true`.

The procedure is an implementation of the Liang-Barsky algorithm.

```
boolean CameraClipPolygon3f ( CameraRecf *CPos,
                              int n, const point3f *p,
                              void (*output)(int n, point3f *p) );
```

The procedure `CameraClipPolygon3f` finds the intersection of a polygon with the visibility pyramid, using the Sutherland-Hodgman algorithm. The parameter `n`

specifies the number of vertices in the space, whose coordinates are given in the array p; the polygon boundary is one closed polyline.

The parameter output points to a procedure, which will be called if the intersection is nonempty. The parameter n of this procedure specifies the number of vertices of the intersection. Projections of those vertices are given in the array p.

5.2 Stereo camera pair

To make a stereo pair of images it is necessary to place two cameras in the space, and then to render the images using the cameras. The procedures described in this section make it easier to manipulate with such a pair of cameras; each procedure corresponds to some procedure of manipulating with one camera and it should be used *instead* of that procedure. To project points or to cast rays one should use the procedures described in the previous section for each camera of the pair.

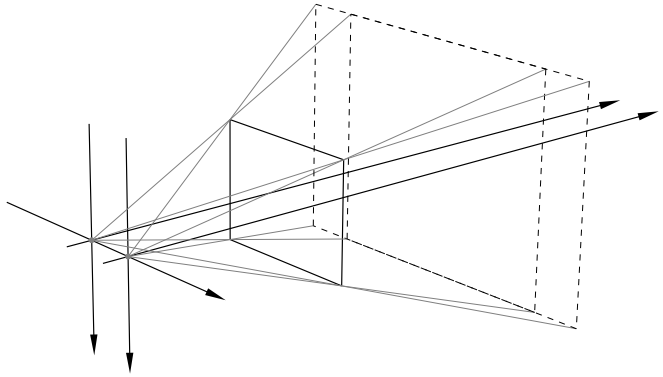


Figure 5.2. Stereo pair of cameras and their common frame

The data structure and the headers of procedures related with the stereo pair of cameras are described in the header file `stereo.h`.

```
typedef struct StereoRecf {
  point3f    position;
  float      d;
  float      l;
  CameraRecf left, right;
  trans3f    STr, STrInv;
} StereoRecf;
```

The structure `StereoRecf` contains two data structures, which describe the left and the right camera.

```
void StereoInitFramef ( StereoRecf *Stereo, boolean upside,
                short width, short height, short xmin, short ymin,
                float aspect );
```

The procedure `StereoInitFramef` initializes the dimensions of the frame of the cameras (in pixels) and the aspect factor. This is done by calling `CameraInitFramef` for each camera of the pair, with these parameters. This procedure should be called

first in the sequence of procedure calls of the camera initialization. After calling it the cameras are *not* ready to use.

The parameter upside specifies the orientation of the y axis in the image system of coordinates — see the description of the procedure CameraInitFramef.

```
void StereoSetDimf ( StereoRecf *Stereo,
                     float f, float d, float l );
```

The procedure SetStereoDimf initialize the dimensions of the cameras, using the units of length of the global system of coordinates (used to represent objects to be drawn). The parameter f specifies the focal length, i.e. the ratio of the distance of the frame from the central points of projection and the diagonal of the frame. The parameter d specifies the distance between the pupils of the eyes of the viewer (i.e. the distance between the centres of projections), and the parameter l specifies the distance of the viewer from the plane of the frame (i.e. from the monitor screen). The length of diagonal of the frame is therefore l/f units of the global system. The same length in inches depends on the monitor.

Remark. The procedure StereoInitPosf initialises the attributes $f = 1$, $d = 0$ and $l = 1$, and these values are not very useful. A more proper values must therefore be assigned, by calling StereoSetDimf.

```
void StereoSetMagf ( StereoRecf *Stereo, char mag );
```

The procedure StereoSetMagf sets the magnification factor of the cameras (e.g. for antialiasing) by calling CameraSetMagf for each camera. The default value of this factor (after this procedure has not been called) is 1.

```
void StereoSetMappingf ( StereoRecf *Stereo );
```

The procedure StereoSetMappingf computes the positions of the centres of projections of the cameras and it prepares the cameras for using (e.g. projecting points), by calling CameraSetMappingf. Before calling this procedure one should call StereoInitFramef and StereoInitPosf.

The procedures described below, which manipulate with the cameras, call this procedure, therefore typical applications need not call it directly.

```
void StereoInitPosf ( StereoRecf *Stereo );
```

The procedure StereoInitPosf moves both cameras to the default position. Both cameras get the same position, assigned by the procedure CameraInitPosf.


```
void StereoSetRotCentref ( StereoRecf *Stereo,
                           point3f *centre,
                           boolean global_coord, boolean global_fixed );
```

The procedure sets the point, through which axes of rotations of the cameras pass. The way of specifying it is the same as in case of a single camera.

```
void StereoMoveGf ( StereoRecf *Stereo, vector3f *v );
```

The procedure StereoMoveGf translates the pair of cameras (without rotating) by the vector $\boldsymbol{v}$, specified in the global system of coordinates.

```
void StereoMoveCf ( StereoRecf *Stereo, vector3f *v );
```

The procedure StereoMoveCf translates the pair of cameras (without rotating) by the vector $\boldsymbol{v}$, specified in the system of coordinates of the stereo pair.

```
void StereoRotGf ( StereoRecf *Stereo,
                   float psi, float theta, float phi );
```

The procedure StereoRotGf turns the pair of cameras. The rotation is specified by the Euler angles ψ, ϑ, φ in the global system of coordinates.

```
#define StereoRotXGf(Stereo,angle) \
  StereoRotGf ( Stereo, 0.0, angle, 0.0 )
#define StereoRotYGf(Stereo,angle) \
  StereoRotGf ( Stereo, 0.5*PI, angle, -0.5*PI )
#define StereoRotZGf(Stereo,angle) \
  StereoRotGf ( Stereo, angle, 0.0, 0.0 )
```

The above macrodefinitions turn the stereo pair around axes parallel to the axes x, y and z of the global system of coordinates.

```
void StereoRotVGf ( StereoRecf *Stereo, vector3f *v, float angle );
```

The procedure StereoRotVGf turns the stereo pair around the axis, whose direction is given by the vector $\boldsymbol{v}$, specified in the global system of coordinates.

```
void StereoRotCf ( StereoRecf *Stereo,
                   float psi, float theta, float phi );
```

The procedure StereoRotCf turns the stereo pair of cameras. The rotation is specified by the Euler angles ψ, ϑ, φ in the stereo pair system of coordinates.

```
#define StereoRotXCf(Stereo,angle) \
  StereoRotCf ( Stereo, 0.0, angle, 0.0 )
#define StereoRotYCf(Stereo,angle) \
  StereoRotCf ( Stereo, 0.5*PI, angle, -0.5*PI )
#define StereoRotZCf(Stereo,angle) \
  StereoRotCf ( Stereo, angle, 0.0, 0.0 )
```

The above macrodefinitions turn the stereo pair around axes parallel to the axes x, y and z of the stereo pair system of coordinates.

```
void StereoRotVCf ( StereoRecf *Stereo, vector3f *v, float angle );
```

The procedure `StereoRotVCf` turns the stereo pair of cameras around the axis, whose direction is given by the vector $\boldsymbol{v}$, specified in the system of coordinates of the stereo pair.

```
void StereoZoomf ( StereoRecf *Stereo, float fchange );
```

The procedure `StereoZoomf` multiplies the focal length of the cameras by the parameter `fchange`. It is better not to use it at all.

6. The libpsout library

The libpsout library consists of procedures, which write PostScript(™) code to a text file. The file represents a picture, defined by subsequent calls of the library procedures, which produce commands of drawing lines etc.

The library procedures are basic, which actually write the PostScript commands, and additional, which make it simpler to draw line segments — one can draw their parts of various widths, mark some points, draw arrows etc.

6.1 Basic procedures

The phrase „the procedure draws a line segment" or whatever is to be interpreted in such a way that based on the parameter values the procedure writes the text, whose processing by a PostScript interpreter will cause the appearance of the line segment on the picture.

```
extern short ps_dec_digits;
```

The variable ps_dec_digits determines the number of decimal digits in the fractional parts of coordinates of points written to the PostScript file. The default value is 3; if the resolution is set to 600DPI and there are no large scaling, this should be enough.

```
void ps_WriteBBox ( float x1, float y1, float x2, float y2 );
```

The procedure ps_WriteBBox called *before* opening the PostScript file sets up the dimensions of the bounding box, to be written in the preamble. The picture should (but does not have to) fit in the bounding box, which will be used by a typesetting system (TEX) to place the picture on a page. The first two parameters are coordinates of the lower left corner, and the last two parameters are coordinates of the upper right corner. The parameters are specified in "big points" (1 big point (1bp in TEX) is 1/72").

The appropriate numbers are easy to find using GhostView. The procedure call with these numbers may be added to the program producing the picture, which may then be recompiled and executed again.

```
void ps_OpenFile ( const char *filename, unsigned int dpi );
void ps_CloseFile ( void );
```

The procedure ps_OpenFile creates a file, whose name is specified by the parameter filename (if a file of than name already exists, then it is deleted) and writes a PostScript header. The header contains the coordinates of the bounding box (if the procedure ps_WriteBBox has been previously called) and a scaling command,

which sets the initial unit length. This unit is determined by the parameter dpi, e.g. if its value is 600, then the unit length is 1/600". Some procedures are written in such a way that symbols they draw (arrows etc.) look best with the unit that long.

The procedure ps_CloseFile closes the PostScript file. It should be called when the picture is finished.

```
void ps_Write_Command ( char *command );
```

The procedure ps_Write_Command writes an arbitrary text in the PostScript file. Any PostScript command may be appended to the file, even if there is no "ready" procedure in the library to write such a command. Therefore all possibilities of PostScript are available.

```
void ps_Set_Gray ( float gray );
```

The procedure ps_Set_Gray sets the specified gray level int the current graphics state. The value of the parameter gray should be in the interval $[0, 1]$.

```
void ps_Set_RGB ( float red, float green, float blue );
```

The procedure ps_Set_RGB sets the colour with the specified red, green and blue components in the current graphics state. The values of the parameters red, green and blue should be in the interval $[0, 1]$.

```
void ps_Set_Line_Width ( float w );
```

The procedure ps_Set_Line_Width sets the specified line width in the current graphics state. The parameter w should have a positive value.

```
void ps_Draw_Line ( float x1, float y1, float x2, float y2 );
```

The procedure ps_Draw_Line draws a line segment, whose end points have the coordinates x1, y1 and x2, y2. The line width, colour and other properties are determined by the current settings in the graphics state.

```
void ps_Set_Clip_Rect ( float w, float h, float x, float y );
```

The procedure ps_Set_Clip_Rect sets the clipping rectangle of dimensions w (width) and h (height), whose lower left vertex has the coordinates x, y.

The clipping is done in addition to all clipping paths specified before. Cancellation of clipping may be done only in such a way, that we save the graphics state by calling ps_GSave ();, then we set the clipping path and after drawing we restore the initial graphics state by calling ps_GRestore ();.

```
void ps_Draw_Rect ( float w, float h, float x, float y );
```

The procedure ps_Draw_Rect draws the edges of the rectangle of dimensions w (width) and h (height), whose lower left vertex has the coordinates x, y. The width and colour of the lines drawn are determined by the current graphics state.

```
void ps_Fill_Rect ( float w, float h, float x, float y );
```

The procedure ps_Fill_Rect fills the rectangle of dimensions w (width) and h (height), whose lower left vertex has the coordinates x, y. The colour of the rectangle is determined by the current graphics state.

```
void ps_Hatch_Rect ( float w, float h, float x, float y,
                     float ang, float d );
```

The procedure ps_Hatch_Rect draws a number of lines to hatch the rectangle of dimensions w (width) and h (height), whose lower left vertex has the coordinates x, y. The angle of inclination of the lines is specified by the parameter ang (in radians), and their distance is the value of the parameter d. The colour and width of the lines is determined by the current graphics state.

```
void ps_Draw_Polyline2f ( int n, const point2f *p );
void ps_Draw_Polyline2d ( int n, const point2d *p );
```

The procedures ps_Draw_Polyline2f and ps_Draw_Polyline2d draw (open) polylines consisting of $n-1$ line segments, whose vertices (n points, i.e. $2n$ floating point numbers) are given in the array p. The colour and line width are determined by the current graphics state.

```
void ps_Draw_Polyline2Rf ( int n, const point3f *p );
void ps_Draw_Polyline2Rd ( int n, const point3d *p );
```

The procedures ps_Draw_Polyline2Rf and ps_Draw_Polyline2Rd draw (open) polylines consisting of $n-1$ line segments, whose vertices (n points, i.e. $3n$ floating point numbers, the homogeneous coordinates) are given in the array p. The colour and line width are determined by the current graphics state.

```
void ps_Set_Clip_Polygon2f ( int n, const point2f *p );
void ps_Set_Clip_Polygon2d ( int n, const point2d *p );
```

The procedures ps_Set_Clip_Polygon2f and ps_Set_Clip_Polygon2d set the clipping path, being a closed polyline with n vertices given in the array p. The PostScript interpreter clips to all clipping paths set before (except for the patchs set after saving the graphics state, which has then been restored).

```
void ps_Set_Clip_Polygon2Rf ( int n, const point3f *p );
void ps_Set_Clip_Polygon2Rd ( int n, const point3d *p );
```

The procedures ps_Set_Clip_Polygon2Rf and ps_Set_Clip_Polygon2Rd set the clipping path, being a closed polyline with n vertices given in the array p, which contains their homogeneous coordinates.


```
void ps_Fill_Polygon2f ( int n, const point2f *p );
void ps_Fill_Polygon2d ( int n, const point2d *p );
```

The procedures ps_Fill_Polygon2f and ps_Fill_Polygon2d fill a polygon with n vertices given in the array p.

```
void ps_Fill_Polygon2Rf ( int n, const point3f *p );
void ps_Fill_Polygon2Rd ( int n, const point3d *p );
```

The procedures ps_Fill_Polygon2Rf and ps_Fill_Polygon2Rd fill a polygon with n vertices (the homogeneous coordinates) given in the array p.

```
void ps_Draw_BezierCf ( const point2f *p, int n );
void ps_Draw_BezierCd ( const point2d *p, int n );
```

The procedures ps_Draw_BezierCf and ps_Draw_BezierCd draw Bézier curves of degree n, whose $n+1$ control points are given in the array p. For $n > 1$ a polyline of 50 line segments is drawn.

The points of the curve are computed without using the libmultibs library.

```
void ps_Draw_Circle ( float x, float y, float r );
```

The procedure ps_Draw_Circle draws the circle with the radius r and the centre (x,y).

```
void ps_Fill_Circle ( float x, float y, float r );
```

The procedure ps_Draw_Circle fills the circle with the radius r and the centre (x,y).

```
void ps_Draw_Arc ( float x, float y, float r, float a0, float a1 );
```

The procedure ps_Draw_Arc draws an arc of a circle with the centre (x,y), radius r and the angles of beginning and end point a0 and a1. The meaning of all parameters is just like of the parameters of the PostScript operator arc, except that the angles are specified in radians (not in degrees).

```
void ps_Mark_Circle ( float x, float y );
```

The procedure ps_Mark_Circle draws a mark (small circle with a white dot) at (x,y).

```
void ps_Init_Bitmap ( int w, int h, int x, int y, byte b );
void ps_Out_Line ( byte *data );
```

The procedure ps_Init_Bitmap prepares outputting of a monochrome bitmap image (black–grey–white). The image is w pxels wide, h pixels high (the pixel width and height are 1 unit of the current system of coordinates), and the lower left corner is at (x,y). The parameter b specifies the number of bits per pixel, which has to be 1, 2, 4 or 8.

After calling the procedure ps_Init_Bitmap it is necessary to call h times the procedure ps_Out_Line, whose parameter is an array of $\lceil w/b \rceil$ bytes. Each byte describes $8/b$ packed pixels. Each call of this procedure causes writing one row of pixels to the PostScript file, from top of the image to the bottom.

The data are output in hexadecimal form, without any compression. Therefore the PostScript file with such an image may be large.

```
void ps_Init_BitmapP ( int w, int h, int x, int y );
void ps_Out_LineP ( byte *data );
```

The procedure ps_Init_BitmapP prepares outputting of a monochrome bitmap image (black–grey–white) in a packed form. The image is w pxels wide, h pixels high (the pixel width and height are 1 unit of the current system of coordinates), and the lower left corner is at (x,y). The colour of each pixel is specified by one byte.

After calling the procedure ps_Init_BitmapP it is necessary to call h times the procedure ps_Out_LineP, whose parameter is an array of w bytes. Each call of this procedure causes writing one row of pixels to the PostScript file, from top of the image to the bottom.

The data are output in hexadecimal form, with a simple run-length encoding compression. Therefore the PostScript file with such an image may smaller.

```
void ps_Init_BitmapRGB ( int w, int h, int x, int y );
void ps_Out_LineRGB ( byte *data );
```

The procedure ps_Init_BitmapRGB prepares outputting of a colour bitmap image. The image is w pxels wide, h pixels high (the pixel width and height are 1 unit of the current system of coordinates), and the lower left corner is at (x,y). The colour of each pixel is specified by three bytes.

After calling the procedure ps_Init_BitmapRGB it is necessary to call h times the procedure ps_Out_LineRGB, whose parameter is an array of $3w$ bytes. Each call of this procedure causes writing one row of pixels to the PostScript file, from top of the image to the bottom.

The data are output in hexadecimal form, without any compression.

```
void ps_Init_BitmapRGBP ( int w, int h, int x, int y );
void ps_Out_LineRGBP ( byte *data );
```

The procedure ps_Init_BitmapRGBP prepares outputting of a colour bitmap image in a packed form. The image is w pxels wide, h pixels high (the pixel width and height are 1 unit of the current system of coordinates), and the lower left corner is at (x,y). The colour of each pixel is specified by three bytes.

After calling the procedure ps_Init_BitmapRGBP it is necessary to call h times the procedure ps_Out_LineRGBP, whose parameter is an array of $3w$ bytes. Each call of this procedure causes writing one row of pixels to the PostScript file, from top of the image to the bottom.

The data are output in hexadecimal form, with a simple run-length encoding. The decompression procedure is coded in PostScript. It is not extremely fast nor effective, but often sufficient. Some day it might be replaced by something better.

```
void ps_Newpath ( void );
```

The procedure ps_Newpath causes writing the command newpath, which initializes a path. This path may then be built with the procedures ps_MoveTo and ps_LineTo, and then it may be processed by the PostScript interpreter in the way described by the procedure ps_Write_Command (it may write stroke or anything else).

```
void ps_MoveTo ( float x, float y );
void ps_LineTo ( float x, float y );
```

The procedures ps_MoveTo and ps_LineTo output the PostScript commands which build a path: respectively moveto and lineto with appropriate parameters. The path constructed with these procedures may be used in an arbitrary way.

```
void ps_ShCone ( float x, float y, float x1, float y1,
                 float x2, float y2 );
```

The procedure ps_ShCone draws a shaded (grey) cone, i.e. triangle, whose vertices are (x,y), (x+x1,y+y1) and (x+x2,y+y2).

```
void ps_GSave ( void );
void ps_GRestore ( void );
```

The procedure ps_GSave writes the command gsave, which causes saving the current graphics state (on the appropriate stack of the PostScript interpreter).

The procedure ps_GRestore writes the command grestore, which restores the graphics state previously saved on the stack.

```
void ps_BeginDict ( int n );
void ps_EndDict ( void );
```

The procedure ps_BeginDict writes the command n dict begin, where n is the number given as the parameter. For the PostScript interpreter it is the order of creating a new dictionary with the capacity of n symbols, and opening it on top of the stack of open dictionaries.

The procedure ps_EndDict writes the command end, which causes removing the dictionary from the top of the dictionary stack. Each call of ps_BeginDict should be compelemnted by a call to ps_EndDict.

```
void ps_DenseScreen ( void );
```

The procedure ps_DenseScreen causes changing the current rasterization pattern to the pattern of twice as large liniature. Thin grey lines may look better

printed with such a pattern, though the precision of reproducing grey levels is worse.

```
void ps_GetSize ( float *x1, float *y1, float *x2, float *y2 );
```

The procedure ps_GetSize may help to get the dimensions of the rectangle bounding the picture (or rather its elements drawn before calling this procedure). Howevet, this procedure does not take into account any effects of the commands output by ps_Write_Command (which may be scale, translate or drawing commands), and it does not take into account clipping. Therefore it is not a very useful procedure (using GhostView is much better).

The parameters obtain the values of coordinates of the rectangle, into which, as it seems to the library, the picture fits. The units are determined by the resolution specified when the file has been created.

6.2 Additional procedures

Additional procedures make it easier to draw line segments, whose parts have various colours and widths. They may also have some parts marked with symbols like dots, arrows, ticks etc.

```
#define tickl 10.0
#define tickw 2.0
#define tickd 6.0
#define dotr 12.0
#define arrowl 71.0
#define arroww 12.5
```

The abobe symbolic constants determine a half of length (tickl) and width (tickw) of a bar (tick) drawn across the current line, dot radius and length and half of width of arrows.

These dimensions are chosen so that the symbols look good, if the unit length (specified by the second parameter of ps_OpenFile) is 1/600".

```
void psl_SetLine ( float x1, float y1, float x2, float y2,
                   float t1, float t2 );
```

The procedure psl_SetLine sets the line, whose segments and points will be drawn and marked. The line passes through the points (x1,y1) and (x2,y2), which must be different. To these two points correspond the parameters t1 and t2, which must be different.

Setting such a line causes computing also its unit directional vector $\boldsymbol{\nu}$, which will be used by other procedures in various constructions.


```
void psl_GetPointf ( float t, float *x, float *y );
```

The procedure psl_GetPointf computes the point of the line recently set by the procedure psl_SetLine, corresponding to the parameter t. Its coordinates are assigned to the parameters *x and *y.

```
float psl_GetDParam ( float dl );
```

The procedure psl_GetDParam computes the increment of the line parameter, which corresponds to the translation by a vector of length dl.

```
void psl_GoAlong ( float s, float *x, float *y );
```

The procedure psl_GoAlong on entry gets the point $\mathbf{p}$ =(*x,*y). On return the parameters *x and *y have values of coordinates of the image of $\mathbf{p}$ in the translation along the current line by the distance s.

```
void psl_GoPerp ( float s, float *x, float *y );
```

The procedure psl_GoPerp on entry gets the point $\mathbf{p}$ =(*x,*y). On return the parameters *x and *y have values of coordinates of the image of $\mathbf{p}$ in the translation prependicular to the current line by the distance s.

```
void psl_Tick ( float t );
```

The procedure psl_Tick draws a tick on the current line, at the point corresponding to the parameter t, which is a bar perpendicular to the line.

```
void psl_BTick ( float t );
```

The procedure psl_BTick draws a tick on the current line, at the point corresponding to the parameter t, which is a bar perpendicular to the line. This bar is thicker and longer than that drawn by psl_Tick. The idea is to draw it using the background colour and then to draw the ordinary bar on that background.

```
void psl_HTick ( float t, boolean left );
```

The procedure psl_HTick draws a tick on the current line, at the point correspoding to the parameter t, which is a half of the bar drawn by the procedure psl_Tick. The parameter left determines the side of the line to draw this half tick.

```
void psl_Dot ( float t );
```

The procedure psl_Dot marks the point of the current line corresponding to the parameter t. The mark is a circle, whose radius is dotr.

```
void psl_HDot ( float t );
```

The procedure psl_HDot marks a point of the current line corresponding to the parameter t, by a circle of slightly greater radius. It is intended to draw the circle using the background colour before drawing the proper circle using psl_Dot.

```
void psl_TrMark ( float x, float y );
```

The procedure psl_TrMark marks the point (x,y) (not related with the current line) by a white isoscelses triangle with black edges. One edge is horizontal and the top vertex is at the marked point.

```
void psl_BlackTrMark ( float x, float y );
```

The procedure psl_BlackTrMark marks the point (x,y) (not related with the current line) by a black isoscelses triangle. One edge is horizontal and the top vertex is at the marked point.

```
void psl_HighTrMark ( float x, float y );
```

The procedure a psl_HighTrMark marks the point (x,y) (not related with the current line) by a white triangle with black edges. One edge is horizontal and the top vertex is at the marked point. The height of this triangle is greater than of that drawn by psl_TrMark.

```
void psl_BlackHighTrMark ( float x, float y );
```

The procedure a psl_BlackHighTrMark marks the point (x,y) (not related with the current line) by a black triangle. One edge is horizontal and the top vertex is at the marked point. The height of this triangle is greater than of that drawn by psl_BlackTrMark.

```
void psl_LTrMark ( float t );
void psl_BlackLTrMark ( float t );
void psl_HighLTrMark ( float t );
void psl_BlackHighLTrMark ( float t );
```

The above procedures mark the point of the current line corresponding to the parameter t, using the symbols drawn by psl_TrMark, psl_BlackTrMark, psl_HighTrMark and psl_BlackHighTrMark respectively. Essentially, they are appropriate for horizontal lines.

```
void psl_Arrow ( float t, boolean sgn );
```

The procedure psl_Arrow marks the point of the current line corresponding to the parameter t, by an arrow (which is a triangle) having the direction of the line. The parameter sgn selects the orientation of the arrow.

```
void psl_BkArrow ( float t, boolean sgn );
```

The procedure psl_BkArrow marks the point of the current line corresponding to the parameter t, drawing the area which is a background of the arrow to be drawn bypsl_Arrow. The parameter sgn selects the orientation of the arrow.


```
void psl_Draw ( float ta, float tb, float w );
```

The procedure psl_Draw draws a segment of the current line, between the points corresponding to the parameters ta and tb. The parameter w specifies the width of the line segment.

```
void psl_ADraw ( float ta, float tb, float ea, float eb, float w );
```

The procedure psl_ADraw draws a segment of the current line, whose end points are obtained as follows: first, the point $\mathbf{p}_a$, which corresponds to ta is computed, and then the unit vector having direction of the current line, multiplied by the parameter ea is added (see the description of the procedure psl_SetLine). The other end point of the segment is obtained in a similar way, using the parameters tb and eb. The parameter w specifies the width of the line.

```
void psl_MapsTo ( float t );
```

The procedure psl_MapsTo marks the point of the current line corresponding to the parameter t with an arrow having a shape different than that of psl_Arrow, more appropriate for commutative diagrams.

```
void psl_DrawEye ( float t, byte cc, float mag, float ang );
```

The procedure psl_DrawEye marks the point of the current line corresponding to the parameter t with an eye symbol. It may be used to denote the viewer position on various schematic pictures.

The parameter cc should be 0, 1, 2 or 3, and it determines the orientation of the symbol. The parameter mag specifies the magnification, and the parameter ang specifies the angle (in radians) of additional rotation of the image, which may be necessary to obtain a good looking effect.

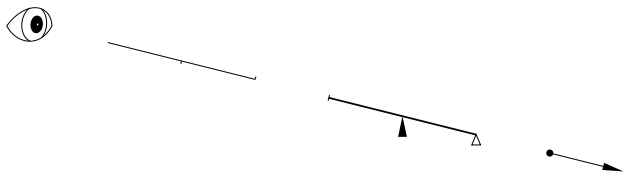


Figure 6.1. A line with marked points

Example: The program below draws the picture shown in Figure 6.1.

```
#include <string.h>
#include "psout.h"
int main ( void )
{
  ps_WriteBBox ( 12, 13, 264, 80 );
  ps_OpenFile ( "psout.ps", 600 );
  psl_SetLine ( 200, 600, 2200, 100, 0.0, 4.0 );
  psl_Draw ( 0.5, 1.5, 2.0 );
  psl_Draw ( 2.0, 3.0, 6.0 );
  psl_ADraw ( 3.5, 4.0, 0.0, -arrowl, 1.0 );
  psl_DrawEye ( 0.0, 1, 1.2, 0.15 );
  psl_HTick ( 1.0, false );
  psl_HTick ( 1.5, true );
  psl_Tick ( 2.0 );
  psl_BlackHighLTrMark ( 2.5 );
  psl_LTrMark ( 3.0 );
  psl_Dot ( 3.5 );
  psl_Arrow ( 4.0, true );
  ps_CloseFile ();
  exit ( 0 );
} /*main*/
```

7. The libmultibs library

7.1 Basic definitions and representations of curves and patches

7.1.1 Bézier curves

A Bézier curve is defined by the formula

$$\mathbf{p}(t) = \sum_{i=0}^{n} \mathbf{p}_i B_i^n(t), \tag{7.1}$$

with control points $\mathbf{p}_0, \ldots, \mathbf{p}_n$ and Bernstein polynomials

$$B_i^n(t) \stackrel{\text{def}}{=} \binom{n}{i} t^i (1-t)^{n-i}, \quad i = 0, \ldots, n. \tag{7.2}$$

The polyline, whose consecutive vertices are the points $\mathbf{p}_0, \ldots, \mathbf{p}_n$, is called the control polygon of the curve. Each control point has d coordinates and then the curve is located in the d-dimensional space. In particular, for $d = 1$ the formula (7.1) describes a polynomial of the variable t of degree at most n.

The representation of a Bézier curve consists of the number n and of the sequence of $n+1$ control points, whose coordinates ($(n+1)d$ floating-point numbers) are packed in an array (i.e. first come d coordinates of $\mathbf{p}_0$, then $\mathbf{p}_1$ etc.).

A rational Bézier curve is given by

$$\mathbf{p}(t) = \frac{\sum_{i=0}^{n} w_i \mathbf{p}_i B_i^n(t)}{\sum_{i=0}^{n} w_i B_i^n(t)}, \tag{7.3}$$

with the Bernstein polynomials, control points $\mathbf{p}_0, \ldots, \mathbf{p}_n$ and weights $w_0, \ldots, w_n$. Such a curve is located in the same space as the control points.

If $w_i = 0$ for some i, then the expression $w_i \mathbf{p}_i$ may be replaced by an arbitrary vector $\boldsymbol{\nu}_i$, thus extending the definition of the curve, but at least one weight must be nonzero.

The control points $\mathbf{p}_i$ of the rational curve are convenient for the program user, which may interactively modify them, but the procedures of this library process the homogeneous representation. For a curve in a d-dimensional space it is a polynomial curve in the space of dimension $d+1$:

$$\mathbf{P}(t) = \sum_{i=0}^{n} \mathbf{P}_i B_i^n(t), \tag{7.4}$$

whose control points $\mathbf{P}_i$ are given by

$$\mathbf{P}_i = \begin{bmatrix} w_i \mathbf{p}_i \\ w_i \end{bmatrix}. \tag{7.5}$$

The last (i.e. $d+$first) homogeneous coordinate is thus the weight. If $w_i = 0$ then

$$\mathbf{P}_i = \begin{bmatrix} \boldsymbol{\nu}_i \\ 0 \end{bmatrix}. \tag{7.6}$$

The cartesian coordinates of the point $\mathbf{p}(t)$ of a rational curve are obtained by dividing the first d coordinates of the point $\mathbf{P}(t)$ by its last coordinate.

The representation of a rational curve e.g. in a three-dimensional space consists of the number n (which determines the degree of the representation) and of an array of $4(n+1)$ floating point numbers, being the coordinates of the consecutive points $\mathbf{P}_i$. As the homogeneous curves are ordinary polynomial curves, in most cases they may be processed with the procedures appropriate for the polynomial Bézier curves.

7.1.2 Tensor product Bézier patches

A tensor product (rectangular) Bézier patch is defined by the formula

$$\mathbf{p}(u,v) = \sum_{i=0}^{n} \sum_{j=0}^{m} \mathbf{p}_{ij} B_i^n(u) B_j^m(v), \tag{7.7}$$

with the Bernstein polynomials B_i^n and B_j^m of degrees n and m respectively and with the control points $\mathbf{p}_{ij}$. By convention, the row of the control net is the polyline with the vertices $\mathbf{p}_{0j}, \ldots, \mathbf{p}_{nj}$ (for $j \in \{0, \ldots, m\}$), and the column is the polyline with the vertices $\mathbf{p}_{i0}, \ldots, \mathbf{p}_{im}$ (for all $i \in \{0, \ldots, n\}$).

The representation of a Bézier patch consists of two positive numbers n and m and of $(n+1)(m+1)$ control points, i.e. $(n+1)(m+1)d$ floating point numbers, stored in the array in the following sequence: first d coordinates of $\mathbf{p}_{00}$, then d coordinates of $\mathbf{p}_{01}$ etc. After the coordinates of the point $\mathbf{p}_{0m}$ there ought to be the coordinates of $\mathbf{p}_{10}$ etc., up to the point $\mathbf{p}_{nm}$. In other words, the control net is stored in the array columnwise.

The array described above may be seen in many different ways. For instance, to apply an affine transformation to the patch, it is necessary to transform its control points. In that case the array may be seen as a one-dimensional array of points.

We can also divide this patch using the de Casteljau algorithm, by halving the interval of the parameter u or v. In the latter case we apply the algorithm to all columns, as if they were control polygons of Bézier curves. The array contains thus $n+1$ curves and its pitch is equal to $(m+1)d$ (where d is the space dimension), i.e. the second curve representation begins $(m+1)d$ places after the first, etc.

To divide the interval of u, it is necessary to apply the de Casteljau algorithm to all rows of the control net. It turns out that the patch representation may be interpreted as a representation of a Bézier curve in the space of dimension $(m+1)d$ (each column of the control net is a point of this space). In this case we process only one Bézier curve of degree n in the $(m+1)d$-dimensional space, and the pitch is irrelevant, as there is only one curve.

A rational Bézier patch is given by the formula

$$\mathbf{p}(u,v) = \frac{\sum_{i=0}^{n}\sum_{j=0}^{m} w_{ij}\mathbf{p}_{ij}B_i^n(u)B_j^m(v)}{\sum_{i=0}^{n}\sum_{j=0}^{m} w_{ij}B_i^n(u)B_j^m(v)},$$

where apart from the Bernstein polynomials and the control points there are weights w_{ij}. The procedures processing rational Bézier patches in the `libmultibs` library, process their homogeneous representations i.e. the arrays of control points $\mathbf{P}_{ij}$ of polynomial Bézier patches

$$\mathbf{P}(u,v) = \sum_{i=0}^{n}\sum_{j=0}^{m} \mathbf{P}_{ij}B_i^n(u)B_j^m(v), \tag{7.8}$$

in the space of dimension $d+1$. The relation between the control points $\mathbf{p}_{ij}$ and weights w_{ij} of the rational patch with the points $\mathbf{P}_{ij}$ here is the same as in the case of the rational Bézier curves. The method of storing the control points $\mathbf{P}_{ij}$ of the homogeneous patch is the same as in the case of the nonrational patch (except that the space dimension, where the homogeneous patch is located is greater by 1).

7.1.3 B-spline curves

Let $n \geq 0$ and let the nondecreasing sequence of knots (real numbers) $u_0, \ldots, u_N$, such that $N > 2n$ and $u_n < u_{N-n}$ be fixed. A B-spline curve of degree n based on this knot sequence is defined by the formula

$$\mathbf{s}(t) = \sum_{i=0}^{N-n-1} \mathbf{d}_i N_i^n(t), \tag{7.9}$$

with the control points $\mathbf{d}_0, \ldots, \mathbf{d}_{N-n-1}$ and B-spline functions $N_0^n, \ldots, N_{N-n-1}^n$. These functions have a number of equivalent definitions, e.g. they may be defined by the recursive Mansfield-de Boor-Cox formula:

$$N_i^0(t) = \begin{cases} 1 & \text{dla } u_i \leq t < u_{i+1}, \\ 0 & \text{w przeciwnym razie,} \end{cases} \tag{7.10}$$

$$N_i^j(t) = \frac{t-u_i}{u_{i+j}-u_i}N_i^{j-1}(t) + \frac{u_{i+j+1}-t}{u_{i+j+1}-u_{i+1}}N_{i+1}^{j-1}(t) \quad \text{dla } j = 1, \ldots, n. \tag{7.11}$$

The domain of the curve is the interval $[u_n, u_{N-n-1})$. In each interval $[u_k, u_{k+1})$ (for $n \leq k < N-n$) the B-spline curve is a polynomial arc of degree at most n.

The representation of a B-spline curve consists of the integer numbers n and N, which determine respectively the degree and the number of the last knot, the sequence of knots (array of floating point numbers) $u_0, \ldots, u_N$ and the control points $\mathbf{d}_0, \ldots, \mathbf{d}_{N-n-1}$, located in the same space that the curve — if the dimension of this space is d, then the array of control points must contain $(N-n)d$ floating point numbers.

To describe the details of procedures the following naming convention is used: boundary knots are the knots, which bound the curve domain, i.e. u_n, u_{N-n} and all knots equal to one of the two. The boundary knots are left and right. The internal knots are all knots in the open interval (u_n, u_{N-n}); these knots have the corresponding junction points of polynomial arcs. Apart from the above there are also the external knots, which are not elements of the closed interval $[u_n, u_{N-n}]$. Apart from the above, the knots u_0 and u_N are called the extremal knots. For example, if $n = 3$, $N = 15$ and

$$u_0 < u_1 = u_2 < u_3 = u_4 = u_5 < u_6 \leq \cdots \leq u_{11} < u_{12} = u_{13} = u_{14} = u_{15},$$

then the knots u_0, u_1 i u_2 are external, the knots u_3, u_4 and $u_{12}, \ldots, u_{15}$ are boundary, and the other knots are internal. the extremal knots are u_0 and u_{15}.

The extremal knots are necessary to define the functions N_0^n and N_{N-n-1}^n, but they do not have any influence on the values of those functions in $[u_n, u_{N-n})$, and thus they do not affect the shape of the curve. Various software packages either require supplying these knots or not. The `libmultibs` library requires specifying them (it suffices that the conditions $u_0 \leq u_1$ and $u_{N-1} \leq u_N$ are satisfied).

A given spline curve may have various representations, which may differ with the degree and with the knot sequence. The construction of a representation with additional knots is called knot insertion. In particular, the representation, whose all knots have the multiplicity (the number of appearances) $n+1$ (i.e. there is $u_0 = \cdots = u_n$, $u_{n+1} = \cdots = u_{2n+1}$, $u_{2n+2} = \cdots = u_{3n+2}$ etc.), is a piecewise Bézier representation.

If the last (with the greatest index) left knot has the index $k > n$, then the initial $k-n$ knots (starting from the left extremal) and the initial $k-n$ control points are unnecessary in the curve representation and they may (must in certain situations) be rejected. Similarly, if the first (with the smallest index) right boundary knot has the index $k < N-n$, then the last $N-n-k$ knots and control points are unnecessary. The representations with unnecessary knots and control points may be the effect of knot insertion (i.e. during the conversion to the piecewise Bézier representation) or of constructing the B-spline representation of the derivatives of a B-spline curve.

A B-spline curve of degree n, whose boundary knots have the multiplicity n or greater is called the curve with clamped ends. If the last (with the greatest index) left boundary knot has the index k (the knots are numbered from 0, therefore

obviously $k \geq n$), then the control point $\mathbf{d}_{k-n}$ is the curve point corresponding to the parameter u_k, i.e. the left end of the domain. If $k = n$, then it is the point $\mathbf{d}_0$; otherwise the points $\mathbf{d}_0, \ldots, \mathbf{d}_{k-n-1}$ have no influence on the shape of the curve (and they may be rejected together with the knots $u_0, \ldots, u_{k-n-1}$). A similar rule concerns the right knot with the smallest index — if it is the knot u_{N-n} of multiplicity n or $n+1$, then the control point $\mathbf{d}_{N-n-1}$ is the end point of the curve (it corresponds to $t = u_{N-n}$).

A curve, whose boundary knots have the multiplicity less than n is called a free end curve. Each end of the curve may be clamped or free, independently on the other end.

Closed B-spline curves are represented in the same way as the other B-spline curves. To be closed, a B-spline curve of degree n must satisfy the following conditions: the knot sequence $u_1, \ldots, u_{N-1}$ has to consist of subsequent elements of an infinite sequence of numbers, such that the sequence of differences is nonnegative and periodic, with the period

$$K = N - 2n,$$

where $N > 3n$. The knot sequence must be nondecreasing and there must be a positive number T, such that

$$u_{k+K} - u_k = T \qquad \text{for } k = 1, \ldots, 2n-1.$$

The knots u_0 and u_N have no influence on the shape of the curve, but they must satisfy the conditions $u_0 \leq u_1$ and $u_{N-1} \leq u_N$.

The sequence $\mathbf{d}_0, \ldots, \mathbf{d}_{N-n-1}$ has to consist of consecutive elements of an infinite periodic sequence with the period $N - 2n$, i.e. there must be

$$\mathbf{d}_{k+K} = \mathbf{d}_k \qquad \text{for } k = 0, \ldots, n-1.$$

An application may use space saving representations of closed B-spline curves, where the knots $u_{K+1}, \ldots, u_{N-1}$ and the control points $\mathbf{d}_K, \ldots, \mathbf{d}_{N-n-1}$, possible to reproduce based on the above conditions are absent. To use the `libmultibs` procedures it is necessary to create a "working" representation, with arrays containing all the knots and control points.

Finding a point and many other computations for closed B-spline curves may be done with the procedures intended to use with "ordinary" B-spline curves with free ends. The representation changes like knot insertion or removal and degree elevation must be done with the procedures which ensure that the new representation satisfies the conditions described above. Such procedures have names with the word "`Closed`", and in most cases they still have to be written.

7.1.4 Tensor product B-spline patches

A B-spline patch is defined with the formula

$$\mathbf{s}(u,v) = \sum_{i=0}^{N-n-1} \sum_{j=0}^{M-m-1} \mathbf{d}_{ij} N_i^n(u) N_j^m(v), \tag{7.12}$$

with two sets of B-spline functions of degrees n and m (different in general), based on the knot sequences $u_0, \ldots, u_N$ and $v_0, \ldots, v_M$ respectively (also different in general, even if $n = m$). Both sequences must be nondecreasing and long enough (there must be $N > 2n$, $M > 2m$, $u_n < u_{N-n}$ and $v_m < v_{M-m}$). The terminology and remarks from the previous section apply to both these sequences.

The array of control points $\mathbf{d}_{ij}$, which together with the knots represent the patch, contains the coordinates of the points $\mathbf{d}_{00}, \mathbf{d}_{01}, \ldots, \mathbf{d}_{0,N-n-1}$, then $\mathbf{d}_{10}, \mathbf{d}_{11}, \ldots, \mathbf{d}_{1,N-n-1}$ etc., i.e. the consecutive columns of the control net of the patch.

Between the consecutive columns there may be unused spaces, which make it possible to insert knots to the sequence "v" of the initial representation. This is done as if the new knot was inserted to the representations of many B-spline curves, whose control polygons are the columns of the patch control net. After the knot insertion the length of the unused spaces is decreased by the length of d floating point numbers (where d is the dimension of the space, in which the patch is located). The pitch of the array in this case is the distance of the beginnings of consecutive columns (measured in floating point numbers).

The curves with clamped end and free end curves correspond to the patches with clamped boundary and with free boundary. For example, if the "u" knot sequence satisfies the condition $u_1 = \cdots = u_n < u_{n+1}$, then the constant parameter curve for $u = u_n$ (one of the four boundary curves of the patch) is a B-spline curve of degree m, based on the knot sequence "v", whose control polygon is the first column of the patch control net. Obviously, each of the four patch boundary curves may be clamped or free, independently of the others.

The closed curves correspond to the closed patches, which may be tubes or tori. One or both knot sequences, and the sequence of rows or columns of the control net (interpreted as points) have to satisfy the conditions formulated for closed B-spline curves.

7.1.5 NURBS curves and patches

NURBS (non-uniform rational B-spline) curves and patches are the curves and patches piecewise rational, whose relation with the B-spline curves and patches is the same as the relation of the rational Bézier curves and patches with the polynomial Bézier curves and patches. One can choose one or two knot sequences and the control points $\mathbf{d}_i$ or $\mathbf{d}_{ij}$ in the d-dimensional space and associate the weight w_i

or w_{ij} with each control point. Then the vectors in the $d+1$-dimensional space

$$\mathbf{D}_i = \begin{bmatrix} w_i \mathbf{d}_i \\ w_i \end{bmatrix} \qquad \text{or} \qquad \mathbf{D}_{ij} = \begin{bmatrix} w_{ij} \mathbf{d}_{ij} \\ w_{ij} \end{bmatrix} \tag{7.13}$$

define a curve or a patch in this space; after dividing the first d coordinates of a point of the homogeneous curve or patch by the $d+$first (weight) coordinate one obtains the cartesian coordinates of the point of the rational curve or patch.

The `libmultibs` procedures process such homogeneous representations of the rational curves and surfaces.

7.1.6 Coons patches

Coons patches of class C^k are tensor product patches defined by sufficiently smooth curves, which describe the boundary of the patch and so called cross derivatives of order $1, \ldots, k$. For any $k \in \mathbb{N}$ a Coons patch is defined by the formula

$$\mathbf{p}(u,v) = \mathbf{p}_1(u,v) + \mathbf{p}_2(u,v) - \mathbf{p}_3(u,v), \tag{7.14}$$

where

$$\mathbf{p}_1(u,v) = \mathbf{C}(u)\hat{H}(v)^T, \quad \mathbf{p}_2(u,v) = \tilde{H}(u)\mathbf{D}(v)^T, \quad \mathbf{p}_3(u,v) = \tilde{H}(u)\mathbf{P}\hat{H}(v)^T,$$

and

$$\begin{aligned}
\mathbf{C}(u) &= [\mathbf{c}_{00}(u), \mathbf{c}_{10}(u), \mathbf{c}_{01}(u), \mathbf{c}_{11}(u), \ldots, \mathbf{c}_{0k}(u), \mathbf{c}_{1k}(u)],\\
\mathbf{D}(v) &= [\mathbf{d}_{00}(v), \mathbf{d}_{10}(v), \mathbf{d}_{01}(v), \mathbf{d}_{11}(v), \ldots, \mathbf{d}_{0k}(v), \mathbf{d}_{1k}(v)],\\
\tilde{H}(u) &= [\tilde{H}_{00}(u), \tilde{H}_{10}(u), \tilde{H}_{01}(u), \tilde{H}_{11}(u), \ldots, \tilde{H}_{0k}(u), \tilde{H}_{1k}(u)],\\
\hat{H}(v) &= [\hat{H}_{00}(v), \hat{H}_{10}(v), \hat{H}_{01}(v), \hat{H}_{11}(v), \ldots, \hat{H}_{0k}(v), \hat{H}_{1k}(v)].
\end{aligned}$$

The curves $\mathbf{c}_{00}, \ldots, \mathbf{c}_{1k}$ describe two opposite boundaries of the patch and the cross derivatives at these boundaries. These curves must have the same domain, denoted here by $[a,b]$. Similarly, the curves $\mathbf{d}_{00}, \ldots, \mathbf{d}_{1k}$ describe the other pair of opposite boundaries and cross derivatives and they also must have the same domain, say $[c,d]$. The domain of the Coons patch is the rectangle $[a,b] \times [c,d]$.

The matrix $\mathbf{P}$ of dimensions $(2k+2) \times (2k+2)$ consists of the points of the given

curves and the vectors of their derivatives of order $1, \ldots, k$:

$$\mathbf{P} = \begin{bmatrix}
\mathbf{c}_{00}(a) & \mathbf{c}_{10}(a) & \ldots & \mathbf{c}_{0k}(a) & \mathbf{c}_{1k}(a) \\
\mathbf{c}_{00}(b) & \mathbf{c}_{10}(b) & \ldots & \mathbf{c}_{0k}(b) & \mathbf{c}_{1k}(b) \\
\vdots & \vdots & & \vdots & \vdots \\
\mathbf{c}_{00}^{(k)}(a) & \mathbf{c}_{10}^{(k)}(a) & \ldots & \mathbf{c}_{0k}^{(k)}(a) & \mathbf{c}_{1k}^{(k)}(a) \\
\mathbf{c}_{00}^{(k)}(b) & \mathbf{c}_{10}^{(k)}(b) & \ldots & \mathbf{c}_{0k}^{(k)}(b) & \mathbf{c}_{1k}^{(k)}(b)
\end{bmatrix} =$$
$$\begin{bmatrix}
\mathbf{d}_{00}(c) & \mathbf{d}_{00}(d) & \ldots & \mathbf{d}_{0k}(c) & \mathbf{d}_{0k}(d) \\
\mathbf{d}_{10}(c) & \mathbf{d}_{10}(d) & \ldots & \mathbf{d}_{1k}(c) & \mathbf{d}_{1k}(d) \\
\vdots & \vdots & & \vdots & \vdots \\
\mathbf{d}_{00}^{(k)}(c) & \mathbf{d}_{00}^{(k)}(d) & \ldots & \mathbf{d}_{0k}^{(k)}(c) & \mathbf{d}_{0k}^{(k)}(d) \\
\mathbf{d}_{10}^{(k)}(c) & \mathbf{d}_{10}^{(k)}(d) & \ldots & \mathbf{d}_{1k}^{(k)}(c) & \mathbf{d}_{1k}^{(k)}(d)
\end{bmatrix}. \tag{7.15}$$

The curves, which define a Coons patch, must satisfy the compatibility conditions, expressed by the equality of the matrices above.

The functions $\tilde{H}_{mj}(u)$ and $\hat{H}_{mj}(v)$ are elements of so called local Hermite bases. It is assumed that these functions are polynomials of degree $2k+1$, where $k \in \{1,2\}$, though one might use other functions of class C^k instead, e.g. spline functions of degree $k+1$. These functions are given by the formula

$$\tilde{H}_{mj}(u) = (b-a)^j H_{mj}\Big(\frac{u-a}{b-a}\Big), \qquad \hat{H}_{mj}(v) = (d-c)^j H_{mj}\Big(\frac{v-c}{d-c}\Big).$$

For $k=1$ there is

$$\begin{aligned}
H_{00}(t) &= B_0^3(t) + B_1^3(t), & H_{10}(t) &= B_2^3(t) + B_3^3(t),\\
H_{01}(t) &= \frac{1}{3}B_1^3(t), & H_{11}(t) &= -\frac{1}{3}B_2^3(t).
\end{aligned}$$

As the polynomials used to define a patch, i.e. in the interpolation of the given curves for $k=1$ in both directions are cubic, the Coons patches of class C^1 are called **bicubic Coons patches**, though such a patch may be defined with curves of any degree.

For $k=2$ the patch is defined with the polynomials of degree 5,

$$\begin{aligned}
H_{00}(t) &= B_0^5(t) + B_1^5(t) + B_2^5(t), & H_{10}(t) &= B_3^5(t) + B_4^5(t) + B_5^5(t),\\
H_{01}(t) &= \frac{1}{5}B_1^5(t) + \frac{2}{5}B_2^5(t), & H_{11}(t) &= -\frac{2}{5}B_3^5(t) - \frac{1}{5}B_4^5(t),\\
H_{02}(t) &= \frac{1}{20}B_2^5(t), & H_{12}(t) &= \frac{1}{20}B_3^5(t),
\end{aligned}$$

and therefore the Coons patches of class C^2 are called **biquintic Coons patches**.

Coons patches (bicubic and biquintic) may be defined with polynomial or spline curves. In the former case the curves are Bézier curves and the domain of the patch

is the unit square $[0,1]^2$. The individual curves defining a patch do not have to have the same degree.

The domain of a patch defined with spline curves (represented as B-spline curves) may be dan arbitrary rectangle $[a,b] \times [c,d]$. The individual curves do not need to have the same degree and they may be represented with various knot sequences (but they must have the same domain, determined by the boundary knots).

The library `libmultibs` contains procedures, which convert the Coons representation of a patch to the Bézier or B-spline form, and fast procedures computing points and derivatives of Coons patches at the points of a regular net in the patch domain; these procedures found their application in the constructions implemented in the libraries `libg1hole` and `libg2hole`.

7.1.7 Naming conventions

The naming conventions are intended to simplify the package user (the programmer) guessing the action done by a procedure and guessing the use of parameters (if two procedures have a parameter with the same name, its rôle is the same in both these procedures).

Each procedure and a macro intended to be called as a procedure in the `libmultibs` library has the name beginning with the prefix `mbs_`.

If after the prefix there is the word `multi`, the procedure is intended to process a number of curves simultaneously. The number of curves is specified by the parameter named `ncurves`.

The suffix consists of two parts. The first part may be empty (if there is the "`multi`" after the prefix), or it indicates the kind of the curve or patch processed by the procedure. The letter `C` denotes a curve, and `P` denotes a patch. The digit denotes the dimension of the space, in which the curve or the patch resides (e.g. `2` denotes a plane). The letter `R` after the digit denotes a curve or a patch in the homogeneous representation. **Caution:** the control points in this case have one coordinate more. The second part of the suffix is the letter `f`, which denotes the single precision (`float`) or `d`, which denotes the double precision[1] of the floating point arithmetic used by the procedure to represent the data and results and in the computations.

The main part of the name denotes the algorithm implemented by the procedure. The macros and procedures with the same name differ with the destination — they are universal (if there is the `multi` part) or specific for curves or patches in the space with the fixed dimension. The most important main parts of the names are

`deBoor` — computing points of B-spline curves and patches with the de Boor algorithm.

[1] In the professional applications only double precision should be used, unless even such a precision is insufficient.

`deBoorDer` — computing points of B-spline curves and patches together with the first order derivatives, using the be Boor algorithm.

`BCHorner` — computing points of Bézier curves and patches using the Horner scheme.

`BCHornerDer` — computing points of Bézier curves and patches together with the first order derivatives, using the Horner scheme.

`BCFrenet` — computing curvatures and Frenet frame vectors of Bézier curves.

`BCHornerNv` — computing the normal vector of a Bézier patch.

`KnotIns` — insertion of a single knot to B-spline curves using the Boehm algorithm.

`KnotRemove` — removing a single knot.

`*Oslo*` — procedures related with inserting and removing a number of knots simultaneously, using the Oslo algorithm.

`MaxKnotIns` — inserting knots so as to obtain a B-spline representation with all internal knots of multiplicity $n+1$ and the boundary knots of multiplicity n or $n+1$, which is a piecewise Bézier representation.

`BisectB` — division of Bézier curves into arcs, related with the division of the domain into two line segments of the same length, with the de Casteljau algorithm.

`DivideB` — division of Bézier curves into arcs, related with the division of the domain into two line segments of arbitrary lengths, with the de Casteljau algorithm.

`BCDegElev` — degree elevation of Bézier curves and patches.

`BSDegElev` — degree elevation of B-spline curves and patches.

`MultBez` — multiplication of polynomials and Bézier curves.

`MultBS` — multiplication of splines and B-spline curves.

`BezNormal` — computing normal vector Bézier patches.

`BSCubicInterp` — construction of cubic B-spline curves of interpolation.

`ConstructApproxBS` — construction of B-spline curves of approximation.

`Closed` — procedures, whose name contains this word, are intended to process closed B-spline curves.

The formal parameters of the procedures and macros may have the following names:

`spdimen` — specifies the dimension d of the space in which the curves reside, i.e. the number of coordinates of each point of this space. If the suffix of the name of a procedure or a macro contains the letter `R`, which indicates a rational object, then the control points have `spdimen` $= d+1$ coordinates.

`degree` — specifies the degree of the curve representation. The degrees of a patch with respect to its two parameters are specified by the parameters named `degreeu` and `degreev`.

`lastknot` — specifies the number N, which is the index of the last knot in the knot sequence. The knot sequence consists of $N+1$ knots. For patches there are two parameters, `lastknotu` and `lastknotv`.

`knots` — array of floating point numbers, with the knots. Two arrays with two knot sequences being parts of a B-spline patch representation are passed to the procedure as the parameters named `knotsu` and `knotsv`.

`ctlpoints` — array of control points. If the space dimension d is 1 (the procedure or macro processes scalar functions), the parameter pointing the appropriate array is called `coeff`.

`pitch` — pitch of the control point array, i.e. the difference between the indexes of the first coordinates of the first control points of two consecutive curves or columns of the control net in the `ctlpoints` array. Such arrays are always treated as arrays of *floating point numbers*, therefore the pitch unit is always the length of one floating point number (even if the formal parameter type is e.g. `point3f*`).

If the formal parameters are used to pass two representations, e.g. the procedure constructs a result representation based on the given one, the parameter names are extended by `in` and `out`. The given representations are described by the parameters, which appear in the formal parameter list *before* the parameters used to describe the result.

7.2 Knot sequence processing

This section describes procedures, which perform various auxiliary actions witj knot sequences, like searching, generating and computing their length (before generating, which is necessary for the allocation of appropriate memory blocks).

7.2.1 Searching knot sequences

Knot sequences $u_0, \ldots, u_N$ pased in arrays to all procedures must be nondecreasing and they must satisfy the condition $u_n < u_{N-n}$. The responsability for that belongs to the calling procedures, as data correctness is not verified each time in order not to slow down the computations.

```
int mbs_KnotMultiplicityf ( int lastknot, const float *knots,
                            float t );
```

The procedure `mbs_KnotMultiplicityf` obtains the array `knots`, with a nondecreasing sequence of $N+1$ numbers, where N is the value of the parameter `lastknot`. The value of the procedure is the number of appearances of the number `t` in this sequence.

```
int mbs_FindKnotIntervalf ( int degree,
                            int lastknot, const float *knots,
                            float t, int *mult );
```

The procedure `mbs_FindKnotIntervalf` obtains the array `knots`, with a nondecresaing sequence of $N+1$ numbers (the number N is the value of the parameter `lastknot`). If the parameter `degree` is equal to -1, then the procedure returns the index k, pointing the position such that `knots[k]` $\leq$ `t` $<$ `knots[k+1]`. It may also return -1 if `t` $<$ `knots[0]` or N if `t` $\geq$ `knots[N]`.

If the value n of the parameter `degree` is nonnegative, then the smallest value returned by the procedure may be n, and the greatest $N-n-1$. It is assumed that the procedure has been called in order to find the interval between two consecutive knots, which is the domain of a polynomial or a polynomial arc, which describes a spline function or a curve of degree n. After finding this interval it is possible to compute the points of the arc (e.g. with the de Boor algorithm). In this way, if $t \notin [u_n, u_{N-1})$, then points of the first or the last arc of the curve will be computed.

The parameter `mult` is used to output the multiplicity of the knot `t`. If it is `NULL`, then it is ignored. Otherwise if $t = u_k$ (for k equal to the value returned by the procedure) then the variable `*mult` is assigned the number of appearances of the number `t` in the knot sequence.

7.2.2 Generating knot sequences

```
int mbs_NumKnotIntervalsf ( int degree, int lastknot,
                            const float *knots );
```

The procedure `mbs_NumKnotIntervalsf` computes the number of intervals between the consecutive knots, which form the domain of spline functions (or curves) of degree `degree`, defined with the knot sequence of length `*lastknot+1`, given in the array `knots`.

```
int mbs_LastknotMaxInsf ( int degree, int lastknot,
                          const float *knots,
                          int *numknotintervals );
```

The procedure `mbs_LastknotMaxInsf` returns the index of the last knot of the representation of the curves, which will be constructed by the procedure `mbs_MaxKnotInsf`.

```
int mbs_NumMaxKnotsf ( int degree, int lastknot,
                       const float *knots );
```

The procedure `mbs_NumMaxKnotsf` computes the length of the knot sequence necessary to represent in the local Bernstein bases of degree `degree` spline functions or curves defined with the knot sequence of length `lastknot+1`, given in the array `knots`.

```
void mbs_SetKnotPatternf ( int lastinknot, const float *inknots,
                           int multipl,
                           int *lastoutknot, float *outknots );
```

The procedure `mbs_SetKnotPatternf` generates the knot sequence, which consists of the numbers given in the array `inknots` (of length `lastinknot` + 1), and such that all knots have the multiplicity `multipl`.

The knot sequence is stored in the array `outknots`, and the index of its last knot is returned with the parameter `*lastoutknot`.

7.2.3 Reparameterization of curves and patches

```
void mbs_TransformAffKnotsf ( int degree, int lastknot,
                              const float *inknots,
                              float a, float b, float *outknots );
```

The procedure `mbs_TransformAffKnotsf` applies an affine transformaation of the domain of a spline curve, i.e. it computes the knot sequence associated with the new domain, which is equivalent to the affine reparameterization of the curve. The original domain is the interval $[u_n, u_{N-n}]$, and the new domain is the interval $[a, b]$. The parameter `degree` specifies the degree n of the curve, the parameter `lastknot` specifies the index of the last knot, the array `inknots` contains the original knots $u_0, \ldots, u_N$.

The parameter `a`, `b` specify the interval $[a, b]$, and there must be $a < b$ (the procedure does not verify it). The new knot sequence is stored in the array `outknots`.

The parameters `inknots` and `outknots` may point two different (disjoint) arrays of length $N + 1$, or they may point the same array. In the latter case the reparameterization is done "in situ".

```
void mbs_multiReverseBSCurvef ( int degree, int lastknot,
                                float *knots,
                                int ncurves, int spdimen,
                                int pitch, float *ctlpoints );
```

The procedure `mbs_multiReverseBSCurvef` reparameterizes B-spline curves of degree n, corresponding to substituting the parameter $-t$ instead of t.

The parameter `degree` specifies the degree n of the curves. The parameters `lastknot` and `knots` specify the knot sequence of the curve representation. The parameter `ncurves` specifies the number of curves, and `spdimen` specifies the dimension of the space, in which the curves are located.

If the parameter `knots` is `NULL`, then the procedure reverses only the order of control points of the curves. Therefore it may be used also for "reversing" the Bézier curves or patches. In this case the parameter `lastknot` is ignored (a curve of degree n has $n + 1$ control points).

The parameter `pitch` specifies the pitch of the array `ctlpoints`, which contains the control points of the curves.

The computation is done "in situ", and it consists of changing the sign and reversing the knot sequence and reversing the sequences of the control points of all the curves. No rounding errors appear in this computation.

7.2.4 Knot modifications

```
int mbs_SetKnotf ( int lastknot, float *knots,
                   int knotnum, int mult, float t );
```

The procedure `mbs_SetKnotf` modifies a knot in a given sequence, with the ordering preserved. The parameters: `lastknot` — number of the last knot in the sequence, `knots` — pointer to the array with the knots, `knotnum` — index of the knot modified, `mult` — multiplicity (the new value is assigned to the entries from `knotnum-i+1` to `knotnum`), `t` — new value of the knot.

After the assignment the sequence is sorted. The return value is the new position of the knot in the sequence, i.e. the number k, such that $t = u_k < u_{k+1}$.

The return value -1 denotes invalid parameter `knotnum`; its value must be between 0 and `lastknot`.

```
int mbs_SetKnotClosedf ( int degree, int lastknot, float *knots,
                         float T, int knotnum, int mult, float t );
```

The procedure `mbs_SetKnotClosedf` modifies a knot in a given sequence, with the ordering and periodicity required by the closed B-spline curve representationpreserved. The parameters: `degree` — degree, `lastknot` — number of the last knot in the sequence, `knots` — pointer to the array with the knots, `T` — length of the curve domain (after the change there must be `T=knots[lastknot-degree]-knots[degree]`) `knotnum` — index of the knot modified, `mult` — multiplicity (the new value is assigned to the entries from `knotnum-i+1` to `knotnum`), `t` — new value of the knot.

After the assignment the sequence is sorted. The return value is the new position of the knot in the sequence, i.e. the number k, such that $t = u_k < u_{k+1}$.

The return value -1 denotes invalid parameter `knotnum`; its value must be between 0 and `lastknot`, or `lastknot`, which must be greater than `3*degree`.

7.2.5 Verifying correctness

```
boolean mbs_ClosedKnotsCorrectf ( int degree, int lastknot,
                                  float *knots,
                                  float T, int K, float tol );
```

The procedure `mbs_ClosedKnotsCorrectf` verifies the correctness of a sequence of knots intended to represent a closed B-spline curve. A correct knot sequence must be nondecreasing and satisfy the condition $u_{i+K} = u_i + T$ for $i = 1, \ldots, n$, where $K = N - 2n$, $N > 3n$. The knot multiplicitiescannot exceed the degree n. The parameter `tol` specifies the tolerance (i.e. the maximal difference $u_{i+K} - T - u_i$); it must be a small positive number, not 0 because of rounding errors.

7.3 Evaluating B-spline functions

The values of B-spline functions may be necessary in interpolation problems. The functions are evaluated based on Formulae (7.10) and (7.11).

```
void mbs_deBoorBasisf ( int degree, int lastknot,
                        const float *knots,
                        float t, int *fnz, int *nnz, float *bfv );
```

The procedure `mbs_deBoorBasisf` evaluates the B-spline functions of degree n (specified by the parameter `degree`) at the point `t`. The functions are defined by specifying a nondecreasing sequence of knots $u_0, \ldots, u_M$ in the array `knots`. The number of knots is `lastknot+1`. The parameter `t` must have the value from the interval $[u_n, u_{N-n}]$.

The computed values of the B-spline functions are stored in the array `bfv`, and the parameter `*fnz` is assigned the number of the first function, whose value is nonzero at `t`; the length of the array `bfv` must be at least `degree+1`.

The parameter `*nnz` is used to pass the information about the number of the functions having nonzero values at `t`. The contents of the array `bfv` starting from the position `*nnz` is indefinite (but the procedure may use `degree` $+ 1$ places of this array to store some intermediate results of the computations).

7.4 Computing points of curves and patches

7.4.1 The de Boor algorithm

The de Boor algorithm of computing the point $\mathbf{s}(t)$ of a curve $\mathbf{s}$ given by Formula (7.9), for $t \in [u_k, u_{k+1})$, $k \in \{n, \ldots, N-n-1\}$, is based on recursive computing the points $\mathbf{d}_i^{(j)}$ for $j = 1, \ldots, n-r$ and $i = k-n, \ldots, k-r$, using the formula

$$\mathbf{d}_i^{(j)} = (1-\alpha_i^{(j)})\mathbf{d}_{i-1}^{(j-1)} + \alpha_i^{(j)}\mathbf{d}_i^{(j-1)}, \qquad (7.16)$$

$$\text{gdzie} \quad \alpha_i^{(j)} = \frac{t-u_i}{u_{i+n+1-j}-u_i}.$$

The points $\mathbf{d}_i^{(0)} = \mathbf{d}_i$ are the curve control points, and the number r is the number of occurrences (the multiplicity) of the number t in the knot sequence $u_0, \ldots, u_N$.

```
int mbs_multideBoorf ( int degree, int lastknot,
                       const float *knots,
                       int ncurves, int spdimen,
                       int pitch, const float *ctlpoints,
                       float t, float *cpoints );
```

The procedure `mbs_multideBoorf` is an implementation of the de Boor algorithm of computing points of B-spline curves. The input data are `ncurves` B-spline curves of degree `degree`, located in the space of dimension `spdimen`. Each curve is defined with the same nondecreasing sequence of `lastknot+1` knots, given in the array `knots`.

The control polygons are given in the array `ctlpoints`; each of them is described by `(lastknot-degree)*spdimen` floating point numbers, and the beginning of description of the next polyline is `pitch` places after the previous one.

The parameter `t` specifies the argument, to which correspond the points of the curves to be computed. The procedure stores the coordinates of those points in the array `cpoints`, whose length must be at least `ncurves*spdimen`.

The value returned by the procedure is the number $n-r$, i.e. the difference of the degree of the curves anf the multiplicity of the number t in the knot sequence. If it is nonnegative then it is the minimal class of continuity of the curves in the neighbourhood of the point t.

```
#define mbs_deBoorC1f(degree,lastknot,knots,coeff,t,value) \
  mbs_multideBoorf(degree,lastknot,knots,1,1,0,coeff,t,value)
#define mbs_deBoorC2f(degree,lastknot,knots,coeff,t,value) \
  mbs_multideBoorf(degree,lastknot,knots,1,2,0,coeff,t,value)
#define mbs_deBoorC3f(degree,lastknot,knots,coeff,t,value) ...
#define mbs_deBoorC4f(degree,lastknot,knots,coeff,t,value) ...
```


The four macros above call the procedure `mbs_multideBoorf` in order to compute the value of *jednej* spline function or a B-spline curve located in the plane, in the 3D or 4D space. The parameters of the macros must satisfy the conditions given in the description of the procedure `mbs_multideBoorf`.

```
void mbs_deBoorC2Rf ( int degree,
                      int lastknot, const float *knots,
                      const point3f *ctlpoints, float t,
                      point2f *cpoint );
```

The procedure `mbs_deBoorC2Rf` computes the point of a planar rational B-spline (NURBS) curve of degree `degree`, defined for a nondecreasing sequence of `lastknot+1` knots given in the array `knots`. The array `ctlpoints` contains the *homogeneous* coordinates the control points (i.e. the control points of the homogeneous curve located in $\mathbb{R}^3$).

The number `t` is the value of the parameter `t`, and it must be and element of the interval `[knots[degree],knots[lastknot-degree]]`. The procedure stores the *cartesian* coordinates of the computed point of the curve in the array `cpoint`.

The main computation is done by the procedure `mbs_multideBoorf`.

```
void mbs_deBoorC3Rf ( int degree,
                      int lastknot, const float *knots,
                      const point4f *ctlpoints, float t,
                      point3f *cpoint );
```

The procedure `mbs_deBoorC3Rf` computes the point of a rational B-spline (NURBS) curve of degree `degree` in a 3D space, defined for a nondecreasing sequence of `lastknot+1` knots given in the array `knots`. The array `ctlpoints` contains the *homogeneous* coordinates the control points (i.e. the control points of the homogeneous curve located in $\mathbb{R}^4$).

The number `t` is the value of the parameter `t`, and it must be and element of the interval `[knots[degree],knots[lastknot-degree]]`. The procedure stores the *cartesian* coordinates of the computed point of the curve in the array `cpoint`.

The main computation is done by the procedure `mbs_multideBoorf`.

```
void mbs_deBoorP3f ( int degreeu,
                     int lastknotu, const float *knotsu,
                     int degreev,
                     int lastknotv, const float *knotsv,
                     int pitch,
                     const point3f *ctlpoints,
                     float u, float v, point3f *ppoint );
```

The procedure `mbs_deBoorP3f` computes a point of a B-spline patch in the 3D space. The degrees of the patch with respect to the parameters u and v are `degreeu`

and degreev respectively. The "u" knot sequence, of length lastknotu+1 is given in the array knotsu, and the "v" knot sequence, of length lastknotv+1 is given in the array knotsv.

The control points of the patch are given in the array ctlpoints, in the following order: the points of the first column of the control net are followed by the points of the second column etc, where each column consists of lastknotv-degreev points.

The parameters u and v specify the point of the patch domain, whose corresponding patch point is to be computed. The coordinates of the computed patch point are storedy by the procedure in the array ppoint.

The main computation is done by the procedure mbs_multideBoorf.

```
void mbs_deBoorP3Rf ( int degreeu,
                      int lastknotu, const float *knotsu,
                      int degreev,
                      int lastknotv, const float *knotsv,
                      int pitch,
                      const point4f *ctlpoints,
                      float u, float v, point3f *ppoint );
```

The procedure mbs_deBoorP3Rf computes the point of a rational B-spline patch (a NURBS patch) in the 3D space. The degrees of the patch with respect to the parameters u and v are equal to degreeu and degreev respectively. The "u" knot sequence of length lastknotu+1 is given in the array knotsu, and the "v" knot sequence of length lastknotv+1 is given in the array knotsv.

The array ctlpoints contains the control points of the *homogeneous patch*, in the following order: the cpoints of the first column are followed by the points of the second column etc. Each column consists of lastknotv-degreev control points.

The parameters u and v specify the point in the patch domain, to which corresponds the patch point to be computed. The cartesian coordinates of this point are stord by the procedure in the array ppoint.

The main computation is done by the procedure mbs_multideBoorf.

```
void mbs_deBoorP4f ( int degreeu,
                     int lastknotu, const float *knotsu,
                     int degreev,
                     int lastknotv, const float *knotsv,
                     int pitch,
                     const point4f *ctlpoints,
                     float u, float v, point4f *ppoint );
```

The procedure mbs_deBoorP4f computes a point of a B-spline patch in the 4D space. The degrees of the patch with respect to the parameters u and v are equal to degreeu and degreev respectively. The "u" knot sequence of length lastknotu+1 is given in the array knotsu, and the "v" knot sequence of length lastknotv+1 is given in the array knotsv.

The control points are given in the array ctlpoints; the points of the first column come first, followed by the points of the second column etc. Each column consists of lastknotv-degreev points.

The parameters u and v specify the point in the patch domain, to which corresponds the patch point to be computed. The cartesian coordinates of this point are stord by the procedure in the array ppoint.

The main computation is done by the procedure mbs_multideBoorf.

The derivative of a spline curve $\mathbf{s}$ defined with Formula (7.9) at the point t is equal to

$$\mathbf{s}'(t) = \frac{n}{u_{k+1} - u_k}(\mathbf{d}_{k-r}^{(n-r-1)} - \mathbf{d}_{k-r-1}^{(n-r-1)}), \tag{7.17}$$

where the points $\mathbf{d}_{k-r}^{(n-r-1)}$ and $\mathbf{d}_{k-r-1}^{(n-r-1)}$ are the intermediate results of the de Boor algorithm. The procedures described below use this algorithm to compute the point of a curve together with the derivative.

```
int mbs_multideBoorDerf ( int degree, int lastknot,
                          const float *knots,
                          int ncurves, int spdimen,
                          int pitch,
                          const float *ctlpoints,
                          float t, float *cpoints,
                          float *dervect );
```

The procedure mbs_multideBoorDerf computes points of ncurves B-spline curves of degree degree, located in the space of dimension spdimen. Additionally, the procedure computes the derivative vectors of the curves at the point t.

The data, which represents the curves is identical as in the case of the procedure mbs_multideBoorf. The computed points of the curves are stored by the procedure in the array cpoints. The derivative vectors are stored in the array dervect.

If the value t of the parameter t is equal to a knot of multiplicity degree or greater, then the procedure computes the right-side derivatives at t, except for the case, when t is the end point of the curve domain (i.e. t = knots[lastknot-degree]). In this case the procedure computes the left-side derivatives. The value returned by the procedure is the difference of the degree of the curves and the multiplicity of the number t in the knot sequence. This difference indicates the minimal class of continuity of the curves in a neighbourhood of the point t.

```
#define mbs_deBoorDerC1f(degree,lastknot,knots,ctlpoints,t,\
    cpoint,cder)\
  mbs_multideBoorDerf(degree,lastknot,knots,1,1,0,ctlpoints,t,\
    cpoint,cder)
#define mbs_deBoorDerC2f(degree,lastknot,knots,ctlpoints,t,\
    cpoint,cder)\
  mbs_multideBoorDerf(degree,lastknot,knots,1,2,0,ctlpoints,t,\
    cpoint,cder)
#define mbs_deBoorDerC3f(degree,lastknot,knots,ctlpoints,t,\
    cpoint,cder) ...
#define mbs_deBoorDerC4f(degree,lastknot,knots,ctlpoints,t,\
    cpoint,cder) ...
```

The four macros above call the procedure `mbs_multideBoorDerf` in order to compute the value of *one* spline function or the point of one curve together with the derivative at the point `t`. The parameters must satisfy the conditions given in the decription of the procedure `mbs_multideBoorDerf`.

```
int mbs_multideBoorDer2f ( int degree,
                       int lastknot, const float *knots,
                       int ncurves, int spdimen,
                       int pitch, const float *ctlpoints,
                       float t, float *p, float *d1, float *d2 );
```

The procedure `mbs_multideBoorDer2f` computes the points $\mathbf{s}_i(t)$ and the vectors $\mathbf{s}_i'(t)$ and $\mathbf{s}_i''(t)$ of B-spline curves $\mathbf{s}_i$ of degree n for a given `t`.

Input parameters: `degree` — the degree n, `lastknot` — the number N of the last knot, `knots` — array of knots, `ncurves` — the number of curves, `pitch` — the pitch of the array of control points, `ctlpoints` — array of control points, `t` — the number `t`.

Output parameters: `p` — the array, in which the procedure stores the points $\mathbf{s}_i(t)$, `d1` — the array, in which the procedure stores the vectors $\mathbf{s}_i'(t)$, `d2` — the array in which the procedure stores the vectors $\mathbf{s}_i''(t)$. The pitch of all those arrays is equal to the space dimension, `spdimen`.

The value returned by the procedure is the number $n - r$, where r is the multiplicity of the number `t` in the knot sequence. This value indicates the minimal class of continuity of the curves $\mathbf{s}_i$ in a neightbourhood of the point `t`.

If the curve or one of its derivatives is discontinuous at `t`, then the computed point or vector is the left side limit (e.g. $\lim_{x \searrow t} \mathbf{s}'(x)$).


```
#define mbs_deBoorDer2C1f(degree,lastknot,knots,coeff,t,p,d1,d2) \
  mbs_multideBoorDer2f(degree,lastknot,knots,1,1,0,coeff,t,p,d1,d2)
#define mbs_deBoorDer2C2f(degree,lastknot,knots,ctlpoints,t, \
    p,d1,d2) \
  mbs_multideBoorDer2f(degree,lastknot,knots,1,2,0, \
    (float*)ctlpoints,t,(float*)p,(float*)d1,(float*)d2)
#define mbs_deBoorDer2C3f(degree,lastknot,knots,ctlpoints,t, \
    p,d1,d2) ...
#define mbs_deBoorDer2C4f(degree,lastknot,knots,ctlpoints,t, \
    p,d1,d2) ...
```

The above macros call the procedure `mbs_multideBoorDer2f` in order to compute the point and the first two derivatives of one B-spline curve located in the space of dimension 1, 2, 3 or 4.

```
int mbs_multideBoorDer3f ( int degree,
                       int lastknot, const float *knots,
                       int ncurves, int spdimen,
                       int pitch, const float *ctlpoints, float t,
                       float *p, float *d1, float *d2, float *d3 );
```

The procedure `mbs_multideBoorDer3f` computes the points $\mathbf{s}_i(t)$ and the vectors $\mathbf{s}_i'(t)$, $\mathbf{s}_i''(t)$ and $\mathbf{s}_i'''(t)$ for B-spline curves $\mathbf{s}_i$ of degree n for a given `t`.

Input parameters: `degree` — the degree n, `lastknot` — the number N of the last knot, `knots` — array of knots, `ncurves` — the ni=umber of curves, `pitch` — the pitch of the array of control points, `ctlpoints` — array of control points, `t` — the number `t`.

Output parameters: `p` — the array in which the procedure stores the points $\mathbf{s}_i(t)$, `d1` — the aray in which the procedure stores the vectors $\mathbf{s}_i'(t)$, `d2` — the array in which the procedure stores the vectors $\mathbf{s}_i''(t)$, `d3` — the array in which the procedure stores the vectors $\mathbf{s}_i'''(t)$. All these arrays have the pitch equal to the space dimension, `spdimen`.

The value returned by the procedure is the number $n - r$, where r is the multiplicity of the number `t` in the knot sequence. This value indicates the minimal class of continuity of the curves $\mathbf{s}_i$ in a neightbourhood of the point `t`.

If the curve or one of its derivatives is discontinuous at `t`, then the computed point or vector is the left side limit (e.g. $\lim_{x \searrow t} \mathbf{s}'(x)$).

```
#define mbs_deBoorDer3C1f(degree,lastknot,knots,coeff,t, \
    p,d1,d2,d3) \
  mbs_multideBoorDer3f(degree,lastknot,knots,1,1,0,coeff,t, \
    p,d1,d2,d3)
#define mbs_deBoorDer3C2f(degree,lastknot,knots,ctlpoints,t, \
    p,d1,d2,d3) \
  mbs_multideBoorDer3f(degree,lastknot,knots,1,2,0, \
    (float*)ctlpoints,t,(float*)p,(float*)d1,(float*)d2,(float*)d3)
#define mbs_deBoorDer3C3f(degree,lastknot,knots,ctlpoints,t, \
    p,d1,d2,d3) ...
#define mbs_deBoorDer3C4f(degree,lastknot,knots,ctlpoints,t, \
    p,d1,d2,d3) ...
```

The above macros call the procedure `mbs_multideBoorDer3f` in order to compute the point and derivatives up to the order 3 of *one* B-spline curve located in the space of dimension 1, 2, 3 or 4.

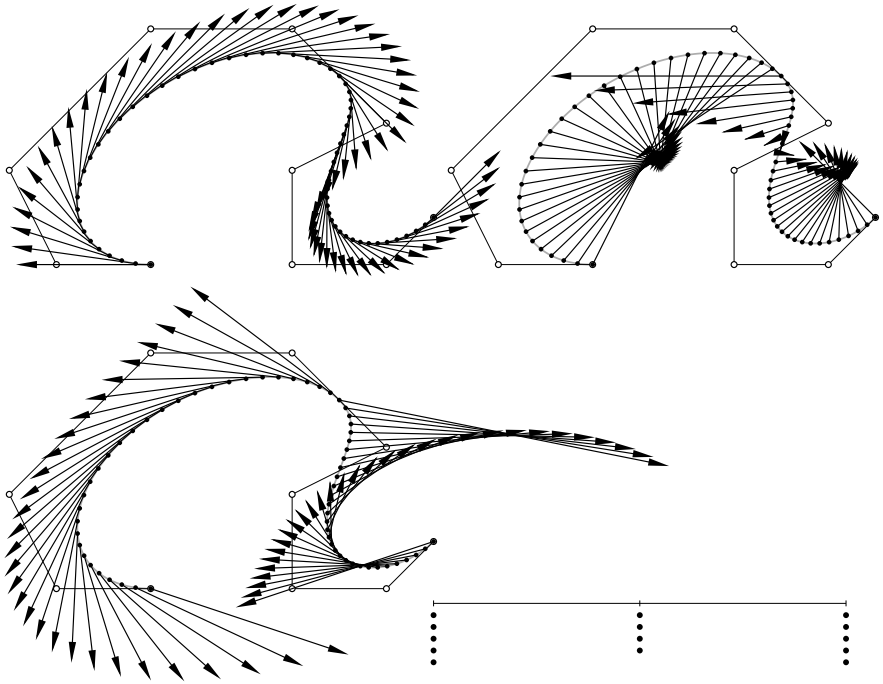


Figure 7.1. Derivative vectors of order 1, 2 and 3 of a B-spline curve of degree 5, computed by the procedures `mbs_multideBoorDerf`, `mbs_multideBoorDer2f` and `mbs_multideBoorDer3f`


```
char mbs_deBoorDerPf ( int degreeu, int lastknotu,
                       const float *knotsu,
                       int degreev, int lastknotv,
                       const float *knotsv,
                       int spdimen, int pitch, const float *ctlpoints,
                       float u, float v,
                       float *ppoint,
                       float *uder, float *vder );
```

The procedure `mbs_deBoorDerPf` computes the point of a B-spline patch, together with its first order partial derivatives.

```
char mbs_deBoorDer2Pf ( int degreeu, int lastknotu,
                       const float *knotsu,
                       int degreev, int lastknotv,
                       const float *knotsv,
                       int spdimen, int pitch, const float *ctlpoints,
                       float u, float v,
                       float *ppoint,
                       float *uder, float *vder,
                       float *uuder, float *uvder, float *vvder );
```

The procedure `mbs_deBoorDer2Pf` computes the point of a B-spline patch, together with its first and second order partial derivatives.

```
char mbs_deBoorDer3Pf ( int degreeu, int lastknotu,
      const float *knotsu,
      int degreev, int lastknotv,
      const float *knotsv,
      int spdimen, int pitch, const float *ctlpoints,
      float u, float v,
      float *ppoint,
      float *uder, float *vder,
      float *uuder, float *uvder, float *vvder,
      float *uuuder, float *uuvder, float *uvvder, float *vvvder );
```

The procedure `mbs_deBoorDer3Pf` computes the point of a B-spline patch, together with its first, second and third order partial derivatives.

7.4.2 Horner scheme for Bézier curves and patches

The Horner scheme is an algorithm of evaluating a polynomial (or computing a point of a curve), whose complexity is proportional to the degree (the cost of the de Casteljau and de Boor algorithms is proportional to the square of degree). To use this algorithm for B-spline curves (which makes sense if many points are to be com-

puted), it is necessary to find a piecewise Bézier representation, using e.g. the mbs_multiMaxKnotInsf procedure (see e.g. Section. 7.7.4).

```
void mbs_multiBCHornerf ( int degree, int ncurves,
                          int spdimen, int pitch,
                          const float *ctlpoints,
                          float t, float *cpoints );
```

The procedure mbs_multiBCHornerf computes points of ncurves Bézier curves of degree degree in the space of dimension spdimen. The control points of the curves are given in the array ctlpoints. The coordinates of the control points of each curve are stored in such a way, that the distance between the beginnings of representations of two consecutive curves is the value of the parameter pitch. The parameter t specifies the parameter of the curves to which correspond the points to be computed. The procedure stores these points in the array cpoints, whose length must be at least ncurves*spdimen.

```
#define mbs_BCHornerC1f(degree,coeff,t,value) \
  mbs_multiBCHornerf ( degree, 1, 1, 0, coeff, t, value )
#define mbs_BCHornerC2f(degree,coeff,t,value) \
  mbs_multiBCHornerf ( degree, 1, 2, 0, coeff, t, value )
#define mbs_BCHornerC3f(degree,coeff,t,value) ...
#define mbs_BCHornerC4f(degree,coeff,t,value) ...
```

The above macros call the procedure mbs_multiBCHornerf in order to evaluate a polynomial represented by its coefficients in the Bernstein basis, or to compute the point of one Bézier curve in the space of dimension 2, 3 or 4.

```
void mbs_BCHornerC2Rf ( int degree,
                        const point3f *ctlpoints,
                        float t, point2f *cpoint );
void mbs_BCHornerC3Rf ( int degree,
                        const point4f *ctlpoints,
                        float t, point3f *cpoint );
```

The above procedures compute a point of a planar or a 3D rational Bézier curve, using the Horner scheme (applied to the homogeneous representation).

```
void mbs_FindBezPatchDiagFormf ( int degreeu, int degreev,
                                 int spdimen, const float *cpoints,
                                 int k, int l, float u, float v,
                                 float *dfcp );
```

The procedure mbs_FindBezPatchDiagFormf computes the diagonal form of degree (k, l) of a Bézier patch $\mathbf{p}$ of degree (n, m) at the point (u, v); it is a tensor product Bézier patch of degree (k, l), which may be obtained after $n-k$ steps of the

de Casteljau algorithm on the rows and $m-l$ steps on the columns of the control net. This patch may be used to compute the point of the patch $\mathbf{p}$ and its derivatives of order $1, \ldots, k$ with respect to u and $1, \ldots, l$ with respect to v. Instead of the de Casteljau algorithm, this procedure uses the faster Horner scheme (by calling mbs_multiBCHornerf).

All procedures of computing points and derivatives (and also curvatures) of Bézier patches described below should call this procedure, though currently only the procedure mbs_BCHornerDer3Pf is written so (and it must still be improved).

```
void mbs_BCHornerPf ( int degreeu, int degreev, int spdimen,
                      const float *ctlpoints,
                      float u, float v, float *ppoint );
```

The procedure mbs_BCHornerPf computes the point $\mathbf{p}(u, v)$ of the Bézier patch $\mathbf{p}$ of degree (n, m), located in the space of dimension d. The parameters degreeu and degreev specify the degrees of the patch (their values are n and m). The parameter spdimen specifies the dimension d of the space containing the patch. The array ctlpoints contains the control points of the patch ($(n+1)(m+1)$d floating point numbers).

The result (the point of the patch) is stored in the array ppoint, whose length must be at least spdimen.

```
#define mbs_BCHornerP1f(degreeu,degreev,coeff,u,v,ppoint) \
  mbs_BCHornerPf ( degreeu, degreev, 1, coeff, u, v, ppoint )
#define mbs_BCHornerP2f(degreeu,degreev,ctlpoints,u,v,ppoint ) \
  mbs_BCHornerPf ( degreeu, degreev, 2, (float*)ctlpoints, \
    u, v, (float*)ppoint )
#define mbs_BCHornerP3f(degreeu,degreev,ctlpoints,u,v,ppoint ) ...
#define mbs_BCHornerP4f(degreeu,degreev,ctlpoints,u,v,ppoint ) ...
```

The above macros call mbs_BCHornerPf in order to compute a point of a Bézier patch in the space of dimension 1, 2, 3 or 4.

```
void mbs_BCHornerP3Rf ( int degreeu, int degreev,
                        const point4f *ctlpoints, float u, float v,
                        point3f *p );
```

The procedure mbs_BCHornerP3Rf computes a point of a rational Bézier patch in the 3D space, represented in the homogeneous form.

```
void mbs_multiBCHornerDerf ( int degree, int ncurves,
                             int spdimen, int pitch,
                             const float *ctlpoints,
                             float t, float *p, float *d );
```

The procedure mbs_multiBCHornerDerf uses the Horner scheme to compute the points $\mathbf{c}_i(t)$ and the vectors $\mathbf{c}_i'(t)$ for Bézier curves $\mathbf{c}_i$ located in the space of dimension d.

The parameters: degree — degree of the curve, ncurves — number of curves, spdimen — dimension d of the space, pitch — pitch of the array ctlpoints with the control points of the curves. the value of the parameter t is the number t.

The coordinates of the points $\mathbf{c}_i(t)$ and the vectors $\mathbf{c}_i'(t)$ are stored in the arrays p and d respectively. Their length must be at least ncurves*spdimen.

```
#define mbs_BCHornerDerC1f(degree,coeff,t,p,d) \
  mbs_multiBCHornerDerf ( degree, 1, 1, 0, coeff, t, p, d )
#define mbs_BCHornerDerC2f(degree,ctlpoints,t,p,d) \
  mbs_multiBCHornerDerf ( degree, 1, 2, 0, (float*)ctlpoints, t, \
    (float*)p, (float*)d )
#define mbs_BCHornerDerC3f(degree,ctlpoints,t,p,d) ...
#define mbs_BCHornerDerC4f(degree,ctlpoints,t,p,d) ...
```

The above macros call mbs_multiBCHornerDerf in order to compute a point and derivative of a Bézier curve in the space of dimension 1, 2, 3 or 4.

```
void mbs_BCHornerDerC2Rf ( int degree, const point3f *ctlpoints,
                           float t, point2f *p, vector2f *d );
void mbs_BCHornerDerC3Rf ( int degree, const point4f *ctlpoints,
                           float t, point3f *p, vector3f *d );
```

The procedures mbs_BCHornerDerC2Rf and mbs_BCHornerDerC3Rf compute the point $\mathbf{p}(t)$ and the vector $\mathbf{p}'(t)$ of a rational Bézier curve $\mathbf{p}$ in the 2D or 3D space respectively.

The parameters: degree — degree of the curves, ctlpoints — array of control points, t — the number t. The procedures assign the coordinates of the point $\mathbf{p}(t)$ to the parameter p, and the vector $\mathbf{p}'(t)$ to the parameter d.

```
void mbs_BCHornerDerPf ( int degreeu, int degreev, int spdimen,
                         const float *ctlpoints,
                         float u, float v,
                         float *p, float *du, float *dv );
```

The procedure mbs_BCHornerDerPf computes the point $\mathbf{p}(u,v)$ and the partial derivatives $\frac{\partial}{\partial u}\mathbf{p}(u,v)$ and $\frac{\partial}{\partial v}\mathbf{p}(u,v)$ of a Bézier patch $\mathbf{p}$ of degree (n,m), located in the space of dimension d.

The parameters: degreeu, degreev — specify the degrees of the patch (the numbers n and m). The parameter spdimen specifies the dimension d of the space, the array ctlpoints contains the coordinates of the control points.

The results (the coordinates of the point and derivatives) are stored in the arrays p, du and dv, whose length must be at least spdimen.


```
#define mbs_BCHornerDerP1f(degreeu,degreev,coeff,u,v,p,du,dv) \
  mbs_BCHornerDerPf ( degreeu, degreev, 1, coeff, u, v, p, du, dv )
#define mbs_BCHornerDerP2f(degreeu,degreev,ctlpoints,u,v,p,du,dv) \
  mbs_BCHornerDerPf ( degreeu,degreev,2,(float*)ctlpoints,u,v, \
    (float*)p, (float*)du, (float*)dv )
#define mbs_BCHornerDerP3f(degreeu,degreev,ctlpoints,u,v,p,du,dv) \
  ...
#define mbs_BCHornerDerP4f(degreeu,degreev,ctlpoints,u,v,p,du,dv) \
  ...
```

The above macros call mbs_BCHornerDerPf in order to compute the points and partial derivatives of Bézier patches in the space of dimension 1, 2, 3 or 4 respectively.

```
void mbs_BCHornerDerP3Rf ( int degreeu, int degreev,
                           const point4f *ctlpoints,
                           float u, float v,
                           point3f *p, vector3f *du, vector3f *dv );
```

The procedure mbs_BCHornerDerP3Rf computes the point $\mathbf{p}(u,v)$ and the partial derivative vectors of a rational Bézier patch located in the 3D space.

The parameters: degreeu, degreev — degrees with respect to the parameters u and v, ctlpoints — array of control points of the homogeneous patch, u, v — the numbers u and v, *p, *du, *dv — variables in which the results are stored.

```
void mbs_multiBCHornerDer2f ( int degree, int ncurves,
                              int spdimen, int pitch,
                              const float *ctlpoints, float t,
                              float *p, float *d1, float *d2 );
```

The procedure mbs_multiBCHornerDerf uses the Horner scheme to compute the points $\mathbf{c}_i(t)$ and the vectors $\mathbf{c}_i'(t)$ and $\mathbf{c}_i''(t)$ for Bézier curves $\mathbf{c}_i$ located in the space of dimension d.

The parameters: degree — degree of the curve, ncurves — number of curves, spdimen — dimension d of the space, pitch — pitch of the array ctlpoints with the control points of the curves. the value of the parameter t is the number t.

The coordinates of the points $\mathbf{c}_i(t)$ and the vectors $\mathbf{c}_i'(t)$ and $\mathbf{c}_i''(t)$ are stored in the arrays p, d1 and d2 respectively. The length of these arrays must not be less than ncurves*spdimen.

```
#define mbs_BCHornerDer2C1f(degree,coeff,t,p,d1,d2) \
  mbs_multiBCHornerDer2f ( degree, 1, 1, 0, coeff, t, p, d1, d2 )
#define mbs_BCHornerDer2C2f(degree,ctlpoints,t,p,d1,d2) \
  mbs_multiBCHornerDer2f ( degree, 1, 2, 0, (float*)ctlpoints, \
    t, (float*)p, (float*)d1, (float*)d2 )
#define mbs_BCHornerDer2C3f(degree,ctlpoints,t,p,d1,d2) ...
#define mbs_BCHornerDer2C4f(degree,ctlpoints,t,p,d1,d2) ...
```

The above macros call `mbs_multiBCHornerDer2f` in order to compute a point and derivatives of order 1 and 2 of a Bézier curve in the space of dimension 1, 2, 3 or 4.

```
void mbs_BCHornerDer2C2Rf ( int degree, const point3f *ctlpoints,
                 float t, point2f *p, vector2f *d1, vector2f *d2 );
void mbs_BCHornerDer2C3Rf ( int degree, const point4f *ctlpoints,
                 float t, point3f *p, vector3f *d1, vector3f *d2 );
```

The procedures `mbs_BCHornerDer2C2Rf` and `mbs_BCHornerDer2C3Rf` compute the point $\mathbf{p}(t)$ and the vectors $\mathbf{p}'(t)$ and $\mathbf{p}''(t)$ of a rational Bézier curve $\mathbf{p}$ in the 2D or 3D space respectively.

The parameters: `degree` — degree of the curves, `ctlpoints` — array of control points, `t` — the number `t`. The procedures assign the coordinates of the point $\mathbf{p}(t)$ to the parameter `p`, and the vectors $\mathbf{p}'(t)$ and $\mathbf{p}''(t)$ to the parameters `d1` and `d2`.

```
void mbs_BCHornerDer2Pf ( int degreeu, int degreev, int spdimen,
                          const float *ctlpoints,
                          float u, float v,
                          float *p, float *du, float *dv,
                          float *duu, float *duv, float *dvv );
```

The procedure `mbs_BCHornerDer2Pf` computes the point $\mathbf{p}(u,v)$ and the partial derivatives of order 1 and 2 of a Bézier patch $\mathbf{p}$ of degree (n,m), located in the space of dimension `d`.

The parameters: `degreeu`, `degreev` — specify the degrees of the patch (the numbers n and m). The parameter `spdimen` specifies the dimension `d` of the space, the array `ctlpoints` contains the coordinates of the control points.

The results (the coordinates of the point and derivatives) are stored in the arrays `p`, `du`, `dv`, `duu`, `duv` and `dvv`, whose length must be at least `spdimen`.


```
#define mbs_BCHornerDer2P1f(degreeu,degreev,coeff,u,v, \
    p,du,dv,duu,duv,dvv) \
  mbs_BCHornerDer2Pf ( degreeu, degreev, 1, coeff, u, v, \
    p, du, dv, duu, duv, dvv )
#define mbs_BCHornerDer2P2f(degreeu,degreev,ctlpoints, \
    u,v,p,du,dv,duu,duv,dvv) \
  mbs_BCHornerDer2Pf ( degreeu, degreev, 2, (float*)ctlpoints, \
    u, v, (float*)p, (float*)du, (float*)dv, \
    (float*)duu, (float*)duv, (float*)dvv )
#define mbs_BCHornerDer2P3f(degreeu,degreev,ctlpoints,u,v, \
    p,du,dv,duu,duv,dvv) ...
#define mbs_BCHornerDer2P4f(degreeu,degreev,ctlpoints,u,v, \
    p,du,dv,duu,duv,dvv) ...
```

The above macros call `mbs_BCHornerDerPf` in order to compute the points and partial derivatives of order 1 and 2 of Bézier patches in the space of dimension 1, 2, 3 or 4 respectively.

```
void mbs_BCHornerDer2P3Rf ( int degreeu, int degreev,
                            const point4f *ctlpoints,
                            float u, float v,
                            point3f *p, vector3f *du, vector3f *dv,
                            vector3f *duu, vector3f *duv,
                            vector3f *dvv );
```

The procedure `mbs_BCHornerDer2P3Rf` computes the point $\mathbf{p}(u,v)$ and the partial derivative vectors of order 1 and 2 of a rational Bézier patch located in the 3D space.

The parameters: `degreeu`, `degreev` — degrees with respect to the parameters u and v, `ctlpoints` — array of control points of the homogeneous patch, `u`, `v` — the numbers u and v, `*p`, `*du`, `*dv`, `*duu`, `*duv`, `*dvv` — variables in which the results are stored, which are the point $\mathbf{p}(u,v)$, and the vectors $\frac{\partial}{\partial u}\mathbf{p}(u,v)$, $\frac{\partial}{\partial v}\mathbf{p}(u,v)$, $\frac{\partial^2}{\partial u^2}\mathbf{p}(u,v)$, $\frac{\partial^2}{\partial u \partial v}\mathbf{p}(u,v)$ and $\frac{\partial^2}{\partial v^2}\mathbf{p}(u,v)$ respectively.

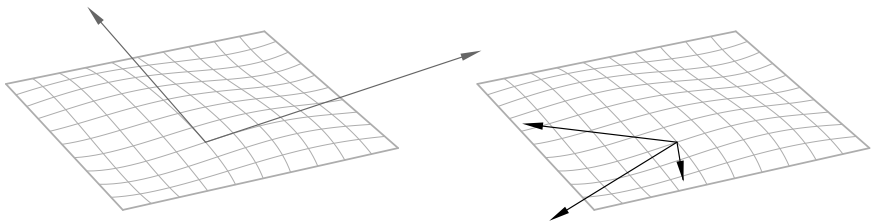


Figure 7.2. A Bézier patch and its derivative vectors of the first and second order.

```
void mbs_multiBCHornerDer3f ( int degree, int ncurves,
                        int spdimen, int pitch,
                        const float *ctlpoints, float t,
                        float *p, float *d1, float *d2, float *d3 );
```

The procedure `mbs_multiBCHornerDerf` uses the Horner scheme to compute the points $\mathbf{c}_i(t)$ and the vectors $\mathbf{c}_i'(t)$, $\mathbf{c}_i''(t)$ and $\mathbf{c}_i'''(t)$ for Bézier curves $\mathbf{c}_i$ located in the space of dimension `d`.

The parameters: `degree` — degree of the curve, `ncurves` — number of curves, `spdimen` — dimension `d` of the space, `pitch` — pitch of the array `ctlpoints` with the control points of the curves. the value of the parameter `t` is the number `t`.

The coordinates of the points $\mathbf{c}_i(t)$ and the vectors $\mathbf{c}_i'(t)$ and $\mathbf{c}_i''(t)$ are stored in the arrays `p`, `d1`, `d2` and `d3` respectively. The length of those arrays must be at least `ncurves*spdimen`.

```
#define mbs_BCHornerDer3C1f(degree,coeff,t,p,d1,d2,d3) \
  mbs_multiBCHornerDer3f ( degree, 1, 1, 0, coeff, t, \
    p, d1, d2, d3 )
#define mbs_BCHornerDer3C2f(degree,ctlpoints,t,p,d1,d2,d3) \
  mbs_multiBCHornerDer3f ( degree, 1, 2, 0, (float*)ctlpoints, \
    t, (float*)p, (float*)d1, (float*)d2, (float*)d3 )
#define mbs_BCHornerDer3C3f(degree,ctlpoints,t,p,d1,d2,d3) ...
#define mbs_BCHornerDer3C4f(degree,ctlpoints,t,p,d1,d2,d3) ...
```

The above macros call `mbs_multiBCHornerDer3f` in order to compute a point and derivatives of order 1, 2 and 3 of a Bézier curve in the space of dimension 1, 2, 3 or 4.

```
void mbs_BCHornerDer3Pf ( int degreeu, int degreev, int spdimen,
                          const float *ctlpoints,
                          float u, float v,
                          float *p, float *pu, float *pv,
                          float *puu, float *puv, float *pvv,
                          float *puuu, float *puuv, float *puvv,
                          float *pvvv );
```

The procedure `mbs_BCHornerDer3Pf` computes the point $\mathbf{p}(u,v)$ of a Bézier patch $\mathbf{p}$ located in the space of dimension `spdimen`, and its partial derivatives of order $1,\ldots,3$. The current version assumes that the degree of both parameters in not less than 3; the implementation of other cases is still to be done.


```
#define mbs_BCHornerDer3P1f(degreeu,degreev,coeff,u,v, \
    p,pu,pv,puu,puv,pvv,puuu,puuv,puvv,pvvv) \
  mbs_BCHornerDer3Pf ( degreeu, degreev, 1, coeff, u, v, \
    p, pu, pv, puu, puv, pvv, puuu, puuv, puvv, pvvv )
#define mbs_BCHornerDer3P2f(degreeu,degreev,ctlpoints,u,v, \
    p,pu,pv,puu,puv,pvv,puuu,puuv,puvv,pvvv) \
  mbs_BCHornerDer3Pf ( degreeu, degreev, 2, (float*)ctlpoints, \
    u, v, (float*)p, (float*)pu, (float*)pv, (float*)puu, \
    (float*)puv, (float*)pvv, (float*)puuu, (float*)puuv, \
    (float*)puvv, (float*)pvvv )
#define mbs_BCHornerDer3P3f(degreeu,degreev,ctlpoints,u,v, \
    p,pu,pv,puu,puv,pvv,puuu,puuv,puvv,pvvv) ...
#define mbs_BCHornerDer3P4f(degreeu,degreev,ctlpoints,u,v, \
    p,pu,pv,puu,puv,pvv,puuu,puuv,puvv,pvvv) ...
```

The above macros call `mbs_BCHormerDer3Pf` in order to compute the point and the derivatives of Bézier patches located in the spaces of dimensions $1,\ldots,4$ respectively.

7.4.3 Computing curvatures and the Frenet frames of curves

Computing curvatures and vectors of the Frenet frame is programmed only for Bézire curves. To compute the curvature of a B-spline curve, one has to do the maximal knot insertion (e.g. with the trocedure `mbs_multiMaxKnotInsf`) to obtain the Bézier representation of its polynomial arcs. The curvature will probably be computed at a number of points, and then it is better to do the conversion once. Therefore there are no procedures computing it directly for B-spline curves. The procedures described below call `mbs_multiBCHornerf`.

```
void mbs_BCFrenetC2f ( int degree, const point2f *ctlpoints,
                       float t, point2f *cpoint,
                       vector2f *fframe, float *curvature );
```

The procedure `mbs_BCFrenetC2f` computes the curvature, the tangent vector $\mathbf{t}$ and the normal vector $\mathbf{n}$ of the Frenet frame of a planar Bézier curve of degree `degree`, whose control points are given in the array `ctlpoints`. The parameter of the curve is `t`. The array `fframe` must be long enough for two vectors. In addition, the procedure computes the point of the curve, and it assigns it to the variable `*cpoint`.

```
void mbs_BCFrenetC2Rf ( int degree, const point3f *ctlpoints,
                        float t, point2f *cpoint,
                        vector2f *fframe, float *curvature );
```

The procedure `mbs_BCFrenetC2f` computes the curvature, the tangent vector $\mathbf{t}$ and the normal vector $\mathbf{n}$ of the Frenet frame of a planar rational Bézier curve of degree `degree`, whose (homogeneous) control points are given in the array `ctlpoints`. The parameter of the curve is `t`. The array `fframe` must be long enough for two vectors. In addition, the procedure computes the point of the curve and it assigns it to the variable `*cpoint`.

```
void mbs_BCFrenetC3f ( int degree, const point3f *ctlpoints,
                       float t, point3f *cpoint,
                       vector3f *fframe, float *curvatures );
```

The procedure `mbs_BCFrenetC3f` computes the curvature and the torsion of a polynomial Bézier curve of degree `degree` and the vectors of the Frenet frame: tangent $\mathbf{t}$, normal $\mathbf{n}$ and binormal $\mathbf{b}$ at the point corresponding to the parameter `t`. The array `ctlpoints` contains the control points of the curve. The curvature and torsion are stored in the array `curvatures`, and the vectors are stored in the array `fframe`. In addition the procedure computes the point of the curve and it assigns it to the variable `*cpoint`.

```
void mbs_BCFrenetC3Rf ( int degree, const point4f *ctlpoints,
                        float t, point3f *cpoint,
                        vector3f *fframe, float *curvatures );
```


The procedure `mbs_BCFrenetC3f` computes the curvature and the torsion of a rational Bézier curve of degree `degree` and the vectors of the Frenet frame: tangent $\mathbf{t}$, normal $\mathbf{n}$ and binormal $\mathbf{b}$ at the point corresponding to the parameter `t`. The array `ctlpoints` contains the control points of the homogeneous curve. The curvature and torsion are stored in the array `curvatures`, and the vectors are stored in the array `fframe`. In addition the procedure computes the point of the curve and it assigns it to the variable `*cpoint`.

7.4.4 Computing the patch normal vector

```
void mbs_BCHornerNvP3f ( int degreeu, int degreev,
                         const point3f *ctlpoints,
                         float u, float v,
                         point3f *p, vector3f *nv );
```

The procedure `mbs_BCHornerNvP3f` computes a point of a Bézier patch in the 3D space and its normal vector at this point. The normal vector is the vector product of the partial derivatives, and it may be zero, if there is a singularity, even if the tangent plane is defined.

```
void mbs_BCHornerNvP3Rf ( int degreeu, int degreev,
                          const point4f *ctlpoints,
                          float u, float v,
                          point3f *p, vector3f *nv );
```

The procedure `mbs_BCHornerNvP3f` computes a point of a rational Bézier patch in the 3D space and its normal vector at this point. The coordinates of the normal vector are the first three coordinates of the vector $\mathbf{P} \wedge \mathbf{P}_u \wedge \mathbf{P}_v$ (the product of the point of the homogeneous patch and its partial derivatives), and it may be zero, if there is a singularity, even if the tangent plane is defined.

7.4.5 Computing the fundamental forms and curvatures of patches

Computing the fundamental forms and curvatures is implemented only for Bézier patches (and not for B-spline patches), for the same reasons that these concerning the curvatures and the Frenet frames of curves.

```
void mbs_FundFormsBP3f ( int degreeu, int degreev,
                         const point3f *ctlpoints,
                         float u, float v,
                         float *firstform, float *secondform );
```

The procedure `mbs_FundFormsBP3f` computes the coefficients of the matrices of the first and the second fundamental forms of a polynomial Bézier patch in the 3D space.

The parameters: degreeu, degreev — degrees of the patch with respect to u and v, ctlpoints — array of control points (packed, i.e. without unsed areas between the consecutive columns of the control net). The parameters u and v specify the point, for which the forms are to be computed.

The parameters firstform and secondform point the arrays, (of length at least 3) in which the form coefficients are to be stored, $g_{11} = \langle \mathbf{p}_u, \mathbf{p}_u \rangle$, $g_{12} = g_{21} = \langle \mathbf{p}_u, \mathbf{p}_v \rangle$, $g_{22} = \langle \mathbf{p}_v, \mathbf{p}_v \rangle$, and $b_{11} = \langle \mathbf{n}, \mathbf{p}_{uu} \rangle$, $b_{12} = b_{21} = \langle \mathbf{n}, \mathbf{p}_{uv} \rangle$ $b_{22} = \langle \mathbf{n}, \mathbf{p}_{vv} \rangle$ respectively ($\mathbf{n}$ denotes the unit normal vector ot the patch at the point (u, v)).

```
void mbs_GMCurvaturesBP3f ( int degreeu, int degreev,
                            const point3f *ctlpoints,
                            float u, float v,
                            float *gaussian, float *mean );
```

The procedure mbs_GMCurvaturesBP3f computes the curvatures: Gaussian and mean of a polynomial Bézier patch in $\mathbb{R}^3$. The parameters degreeu, degreev, ctlpoints, u and v are identical as the corresponding parameters of the previous procedure.

The parameters *gaussian and *mean are used to return the result; the procedures assigns the curvatures to them.

```
void mbs_PrincipalDirectionsBP3f ( int degreeu, int degreev,
                                   const point3f *ctlpoints,
                                   float u, float v,
                                   float *k1, vector2f *v1,
                                   float *k2, vector2f *v2 );
```

The procedure mbs_PrincipalDirectionsBP3f computes the principal curvatures and directions of a polynomial Bézier patch in the 3D space. The parameters degreeu, degreev, ctlpoints, u and v are identical as in the case of two previous procedures.

The parameters *k1 and *k2 obtain the values of the principal curvatures, and the corresponding directions (in the space tangent to the patch domain) are assigned to the parameters *v1 and *v2.

```
void mbs_FundFormsBP3Rf ( int degreeu, int degreev,
                          const point4f *ctlpoints,
                          float u, float v,
                          float *firstform, float *secondform );
```

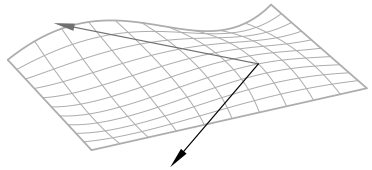



Figure 7.3. Vectors corresponding to the principal directions at a point of a Bézier patch.

```
void mbs_GMCurvaturesBP3Rf ( int degreeu, int degreev,
                             const point4f *ctlpoints,
                             float u, float v,
                             float *gaussian, float *mean );
void mbs_PrincipalDirectionsBP3Rf ( int degreeu, int degreev,
                                    const point4f *ctlpoints,
                                    float u, float v,
                                    float *k1, vector2f *v1,
                                    float *k2, vector2f *v2 );
```

The above procedures respectively compute the coefficients of the matrices of the first and second fundamental form, the Gaussian and mean curvatures and the principal curvatures and directions for a rational Bézier patch $\mathbf{p}$. The procedures directly correspond to the procedures mbs_FundFormsBP3f, mbs_GMCurvaturesBP3f and mbs_GMCurvaturesBP3f, and they have the same parameters, except for the array ctlpoints, which has to contain the coordinates of the control points of a *homogeneous* patch in $\mathbb{R}^4$.

7.5 Evaluating curves at a number of points

The procedures described below compute a sequence of points of a Béziera or B-spline curve together with their derivatives of order 1 and 2, or 1, 2 and 3, for a sequence of values of the parameter: $t_0, \ldots, t_{k-1}$. This is done by calling in a loop the appropriate procedures described before. The main application of these procedures is evaluatind Coons patches at a rectangular net by the procedures described in Section 7.19.

```
void mbs_TabBezCurveDer2f ( int spdimen, int degree,
                            const float *cp,
                            int nkn, const float *kn,
                            int ppitch,
                            float *p, float *dp, float *ddp );
```

The procedure `mbs_TabBezCurveDer2f` evaluates a Bézier curve and its derivatives of order 1 and 2 using the procedure `mbs_multiBCHornerDer2f`.

The parameters: `spdimen` — space dimension, `degree` — degree of the curve, `cp` — array of control points, `nkn` — number k, `kn` — array with k numbers (values of the curve parameter), `ppitch` — pitch of the arrays `p`, `dp` and `ddp`, in which the points and derivative vectors of order 1 and 2 respectively are to be stored. The first coordinates of the consecutive points or vectors are stored at the positions distant by the value of `ppitch`.

```
void mbs_TabBezCurveDer3f ( int spdimen, int degree,
                    const float *cp,
                    int nkn, const float *kn,
                    int ppitch,
                    float *p, float *dp, float *ddp, float *dddp );
```

The procedure `mbs_TabBezCurveDer3f` evaluates a Bézier curve and its derivatives of order 1, 2 and 3 using the procedure `mbs_multiBCHornerDer3f`.

The parameters: `spdimen` — space dimension, `degree` — degree of the curve, `cp` — array of control points, `nkn` — number k, `kn` — array with k numbers (values of the curve parameter), `ppitch` — pitch of the arrays `p`, `dp`, `ddp`, and `dddp`, in which the points and derivative vectors of order 1, 2 and 3 respectively are to be stored. The first coordinates of the consecutive points or vectors are stored at the positions distant by the value of `ppitch`.

```
void mbs_TabBSCurveDer2f ( int spdimen, int degree, int lastknot,
                           const float *knots, const float *cp,
                           int nkn, const float *kn, int ppitch,
                           float *p, float *dp, float *ddp );
```

The procedure `mbs_TabBSCurveDer2f` evaluates a B-spline curve and its derivatives of order 1 and 2 using the procedure `mbs_multideBoorDer2f`.

The parameters: `spdimen` — space dimension, `degree` — degree of the curve, `lastknot` — number of the last knot, `knots` — array of curve knots, `cp` — array of control points, `nkn` — number k, `kn` — array with k numbers (values of the curve parameter), `ppitch` — pitch of the arrays `p`, `dp` and `ddp`, in which the points and derivative vectors of order 1 and 2 respectively are to be stored. The first coordinates of the consecutive points or vectors are stored at the positions distant by the value of `ppitch`.

```
void mbs_TabBSCurveDer3f ( int spdimen, int degree, int lastknot,
                    const float *knots, const float *cp,
                    int nkn, const float *kn, int ppitch,
                    float *p, float *dp, float *ddp, float *dddp );
```

The procedure `mbs_TabBSCurveDer3f` evaluates a B-spline curve and its derivatives of order 1 2 and 3 using the procedure `mbs_multideBoorDer3f`.

The parameters: `spdimen` — space dimension, `degree` — degree of the curve, `lastknot` — number of the last knot, `knots` — array of curve knots, `cp` — array of control points, `nkn` — number k, `kn` — array with k numbers (values of the curve parameter), `ppitch` — pitch of the arrays `p`, `dp`, `ddp` and `dddp`, in which the points and derivative vectors of order 1, 2 and 3 respectively are to be stored. The first coordinates of the consecutive points or vectors are stored at the positions distant by the value of `ppitch`.

7.6 Computing the representation of derivatives

Computing the derivative vector at a point is something different than constructing a representation of the curve, which describes the derivative. The procedures in this section use the following formulae:

$$\frac{d}{dt}\sum_{i=0}^{n}\mathbf{p}_i B_i^n(t) = \sum_{i=0}^{n-1} n(\mathbf{p}_{i+1}-\mathbf{p}_i)B_{i+1}^{n-1}(t), \tag{7.18}$$

for Bézier curves, and

$$\frac{d}{dt}\sum_{i=0}^{N-n-1}\mathbf{d}_i N_i^n(t) = \sum_{i=0}^{N-n-2}\frac{n}{u_{i+n+1}-u_{i+1}}(\mathbf{d}_{i+1}-\mathbf{d}_i)N_{i+1}^{n-1}(t), \tag{7.19}$$

for B-spline curves. The B-spline functions N_i^n and N_i^{n-1} are defined with the same knot sequence.

```
void mbs_multiFindBezDerivativef ( int degree,
                                   int ncurves, int spdimen,
                                   int pitch, const float *ctlpoints,
                                   int dpitch, float *dctlpoints );
```

The procedure `mbs_multiFindBezDerivativef` computes the control points of Bézier curves of degree $n-1$, which describe the derivatives of given Bézier curves of degree n.

Input parameters: `degree` — degree n of the given curves (must be positive), `ncurves` — number of curves, `spdimen` — dimension of the space, in which the curves are located, `pitch` — pitch of the array `ctlpoints`, which is the distance between the beginnings of the representations of the consecutive curves in the array `ctlpoints`, with the control points.

The parametr `dpitch` specifies the pitch of the array `dctlpoints`, in which the procedure stores the control points of the curves representing the derivatives.

```
#define mbs_FindBezDerivativeC1f(degree,coeff,dcoeff) \
  mbs_multiFindBezDerivativef ( degree, 1, 1, 0, coeff, 0, dcoeff )
#define mbs_FindBezDerivativeC2f(degree,ctlpoints,dctlpoints) \
  mbs_multiFindBezDerivativef ( degree, 1, 2, 0, \
    (float*)ctlpoints, 0, (float*)dctlpoints )
#define mbs_FindBezDerivativeC3f(degree,ctlpoints,dctlpoints) \
  mbs_multiFindBezDerivativef ( degree, 1, 3, 0, \
    (float*)ctlpoints, 0, (float*)dctlpoints )
#define mbs_FindBezDerivativeC4f(degree,ctlpoints,dctlpoints) \
  mbs_multiFindBezDerivativef ( degree, 1, 4, 0, \
    (float*)ctlpoints, 0, (float*)dctlpoints )
```


The above macros call `mbs_multiFindBezDerivativef` in order to compute the control points of the derivative of one Bézier curve of degree n in the space of dimension 1, 2, 3, 4.

```
void mbs_multiFindBSDerivativef ( int degree, int lastknot,
                                  const float *knots,
                                  int ncurves, int spdimen,
                                  int pitch, const float *ctlpoints,
                                  int *lastdknot, float *dknots,
                                  int dpitch, float *dctlpoints );
```

The procedure `mbs_multiFindBSDerivativef` computes the control points of B-spline curves of degree $n-1$, which describe the derivatives of given B-spline curves of degree n.

Input parameters: `degree` — degree n of the given curves, `lastknot` — index N of the last knot, `knots` — array of knots $u_0,\ldots,u_N$, `ncurves` — number of curves `spdimen` — dimension of the space, `pitch` — pitch of the array `ctlpoints` (specifying the distance between the beginnings of the consecutive curves), `ctlpoints` — array with the control points of the given curves.

The output parameter `*lastdknot` takes the value $N-2$, and the procedure copies the knots $u_1,\ldots,u_{N-1}$ to the array `dknots`. The parameters `lastdknot` and `dknots` may be `NULL`, and then they are ignored.

The parameter `dpitch` specifies the pitch of the array `dctlpoints`, in which the procedure stores the control points of the curves, which describe the derivatives.

```
#define mbs_FindBSDerivativeC1f(degree,lastknot,knots,coeff, \
    lastdknot,dknots,dcoeff) \
  mbs_multiFindBSDerivativef ( degree, lastknot, knots, 1, 1, 0, \
    coeff, lastdknot, dknots, 0, dcoeff )
#define mbs_FindBSDerivativeC2f(degree,lastknot,knots,ctlpoints, \
    lastdknot,dknots,dctlpoints) \
  mbs_multiFindBSDerivativef ( degree, lastknot, knots, 1, 2, 0, \
    (float*)ctlpoints, lastdknot, dknots, 0, (float*)dctlpoints )
#define mbs_FindBSDerivativeC3f(degree,lastknot,knots,ctlpoints, \
    lastdknot,dknots,dctlpoints) ...
#define mbs_FindBSDerivativeC4f(degree,lastknot,knots,ctlpoints, \
    lastdknot,dknots,dctlpoints) ...
```

The above macros call `mbs_multiFindBSDerivativef` in order to find the representation of the derivative of one B-spline curve in the space of dimension 1, 2, 3 or 4.

7.7 Knot insertion and removal

7.7.1 The Boehm algorithm

The purpose of the procedure and the macros described below is to insert a single knot into the representation of B-spline curves, using the Boehm algorithm. The representation of the curves is *modified*, i.e. the memory area occupied by the initial representation (knot sequence and control points) after return contains the new representation, with the additional knot. If both representations are necessary, then the original representation should be copied by the application, and then the copy may be modified.

```
int mbs_multiKnotInsf ( int degree, int *lastknot,
                        float *knots,
                        int ncurves, int spdimen,
                        int inpitch, int outpitch,
                        float *ctlpoints, float t );
```

The procedure `mbs_multiKnotInsf` inserts the knot `t` to the representation of B-spline curves of degree n =`degree`. In this way a new representation of those curves is constructed, and it replaces the original representation. The number `t` must be from the interval [`knots[degree]`, `knots[lastknot-degree]`].

Initially the parameter `*lastknot` specifies the index N of the last knot of the initial knot sequence; on return it is increased by 1, which indicates the growth of the knot sequence by one number — the value of the parameter `t`, inserted into the array `knots`. Therefore this array must have the capacity at least `*lastknot`+2, to accomodate the longer knot sequence.

The parameter `ncurves` specifies the number of curves, and the parameter `spdimen` is the dimension d of the space with the curves. Each curve is initially represented by $N-n$ points in the d-dimensional space. The coordinates of those points ($(N-n)d$ numbers) are stored in the array `ctlpoints`. The first coordinate of the first control point of the first curve is at the begining of the array. As on return the representation of each curve has one control point more, there are two parameters to describe the pitch, i.e. the distance between the beginnings of representations of two consecutive curves: `inpitch` specifies the initial pitch, at least $(N-n)d$, the parameter `outpitch` specifies the final pitch, which must not be less than $(N-n+1)d$.

The value returned by the procedure is the number k of the interval $[u_k, u_{k+1})$ for the initial knot sequence, whose element is the new knot `t`. Upon return it is inserted into the array `knots` at the position $k+1$ and `*lastknot` is increased by one.

Remark: To insert a knot into the representation of a closed curve, instead of `mbs_multiKnotInsf` one should use the procedure `mbs_multiKnotInsClosedf`.


```
#define mbs_KnotInsC1f(degree,lastknot,knots,coeff,t) \
  mbs_multiKnotInsf(degree,lastknot,knots,1,1,0,0,coeff,t)
#define mbs_KnotInsC2f(degree,lastknot,knots,coeff,t) \
  mbs_multiKnotInsf(degree,lastknot,knots,1,2,0,0,coeff,t)
#define mbs_KnotInsC3f(degree,lastknot,knots,coeff,t) ...
#define mbs_KnotInsC4f(degree,lastknot,knots,coeff,t) ...
```

The four macros above call `mbs_multiKnotInsf` in order to insert a knot to the representation of *one* scalar spline function or B-spline curve in the space of dimension 2, 3 and 4. The parameters must satisfy the conditions given in the description of the procedure `mbs_multiKnotInsf`.

```
int mbs_multiKnotInsClosedf ( int degree, int *lastknot,
                              float *knots,
                              int ncurves, int spdimen,
                              int inpitch, int outpitch,
                              float *ctlpoints, float t );
```

The procedure `mbs_multiKnotInsClosedf` inserts a knot `t` to the representation of *closed* B-spline curves of degree `degree`. It may also be used to insert a knot to a closed B-spline patch (being a tube or a torus). The main conputation is done by the procedure `mbs_multiKnotInsf`. After it returns, the result is further processed in order to restore the periodicity of the curves representation.

The parameters: `degree` — degree of the curves, `*lastknot` — on entry its value is the number of the last knot in the initial sequence, on return its value is increased by 1. The array `knots` contains the knot sequences, the initial and final one respectively. The parameter `ncurves` specifies the number of curves. The parameter `spdimen` specifies the space dimension. The parameters `inpitch` and `outpitch` specify the pitch of the array `ctlpoints` with the control points, before and after the knot insertion, see the description of the procedure `mbs_multiKnotInsf`. The parameter `t` specifies the new knot, to be inserted.

```
#define mbs_KnotInsClosedC1f(degree,lastknot,knots,coeff,t) \
  mbs_multiKnotInsClosedf(degree,lastknot,knots,1,1,0,0,coeff,t)
#define mbs_KnotInsClosedC2f(degree,lastknot,knots,coeff,t) \
  mbs_multiKnotInsClosedf(degree,lastknot,knots,1,2,0,0,coeff,t)
#define mbs_KnotInsClosedC3f(degree,lastknot,knots,coeff,t) ...
#define mbs_KnotInsClosedC4f(degree,lastknot,knots,coeff,t) ...
```

The four macros above call `mbs_multiKnotInsClosedf` in order to insert a knot to the representation of *one* periodic spline function or a closed B-spline curve in the spaces of dimension 2, 3 and 4. The parameters must satisfy the condition given in the description of the procedure `mbs_multiKnotInsClosedf`.

7.7.2 Removing knots

This section describes a procedure of removing a single knot from the representation of B-spline curves, and the macros, which make it easier to use this procedure for a single curve in the spaces of dimensions 1–4. The procedure sets up a system of equations related with two representations of the curves, with the matrix corresponding to the change of representation by the Boehm algorithm, and then it solves this system as a linear least-squares problem. The curves obtained by removing a knot may differ from the original curves.

The knot removal takes place "at the spot", i.e. the memory area initially occupied by the given representation, upon return contains a new, shorter knot sequence and new control points. If both representations are necessary, then the application should copy the original representation of the curves and remove a knot from the copy.

```
int mbs_multiKnotRemovef ( int degree, int *lastknot,
                           float *knots,
                           int ncurves, int spdimen,
                           int inpitch, int outpitch,
                           float *ctlpoints,
                           int knotnum );
```

The procedure `mbs_multiKnotRemovef` removes a knot from the representation of B-spline curves of degree `degree`, located in the space of dimension `spdimen`. The representation is defined for a knot sequence of length `*lastknot+1`, given in the array `knots`. The control points of the curves are given in the array `ctlpoints`. The parameter `inpitch` specifies the pitch, i.e. the initial distance between the beginnings of the areas in the array `ctlpoints` with the control points of the consecutive curves. The parameter `outpitch` specifies the final pitch of this array (rearranged after the knot removal).

The knot to be removed is indicated by the parameter `knotnum`, whose value must be from `degree+1` to `lastknot-degree-1`.

The new representation of the curves replaces the initial one in the arrays `knots` and `ctlpoints`. The parameter `*lastknot` is decreased by 1.

If the multiplicity of the knot being removed is equal to r and the derivative of the curve of order $\texttt{degree} - r + 1$ is not continuous at this knot, then the knot removal will change the curve. The new control points are computed by solving a linear least squares problem, which is a method of solving an approximation problem (see example given below).

The value of the procedure is the number k, such that the removed knot is the element of the interval $[u_k, u_{k+1})$ determined by the *final* knot sequence. If the knot, whose number is `knotnum` is less than the next knot in the sequence, then $k = \texttt{knotnum} - 1$, but in general it may not be the case.

Remark: To remove a knot from the representation of closed curves one should call `mbs_multiKnotRemoveClosedf`. Using the procedure `mbs_multiKnotRemovef` may result in getting non-closed curves.


```
#define mbs_KnotRemoveC1f(degree,lastknot,knots,coeff,knotnum) \
  mbs_multiKnotRemovef(degree,lastknot,knots,1,1,0,0,coeff,knotnum)
#define mbs_KnotRemoveC2f(degree,lastknot,knots,ctlpoints, \
    knotnum) \
  mbs_multiKnotRemovef(degree,lastknot,knots,1,2,0,0, \
    (float*)ctlpoints,knotnum)
#define mbs_KnotRemoveC3f(degree,lastknot,knots,ctlpoints, \
    knotnum) ...
#define mbs_KnotRemoveC4f(degree,lastknot,knots,ctlpoints, \
    knotnum) ...
```

The four macros above call `mbs_multiKnotRemovef` in order to remove a knot from the representation of *one* scalar spline function or one B-spline curve in the space of dimension 2, 3 and 4. The parameters must be as described in the description of the procedure `mbs_multiKnotRemovef`.

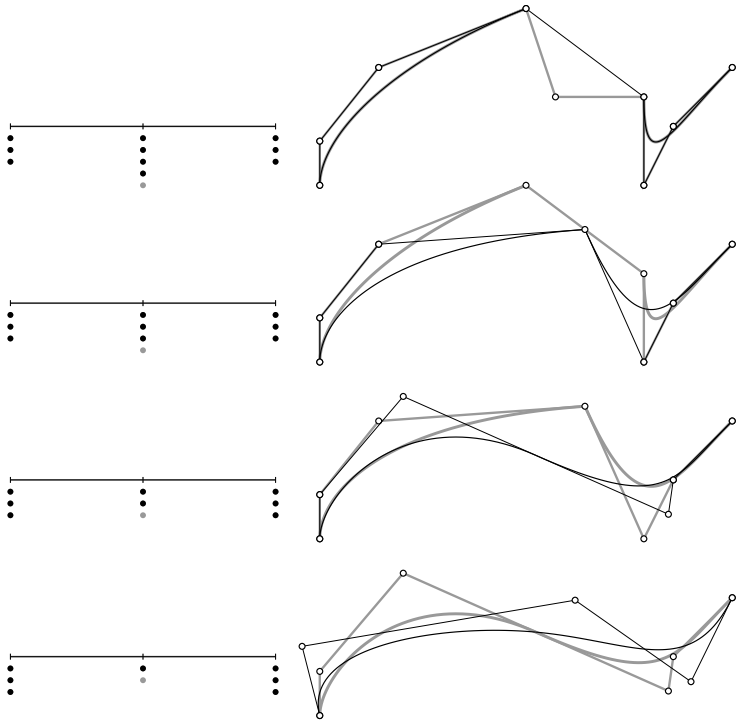


Figure 7.4. Example of knot removal.

Example of knot removal for a planar cubic B-spline curve is shown in Figure 7.4 (see the program `test/knotrem.c`). The initial multiplicity of the knot being removed is the degree of the curve plus two; the curve consists of two disjoint pieces and one of its control points does not influence its shape. Removing the knot causes rejecting this point, without changing the curve.

Removing the knot of multiplicity degree plus one causes connecting the curve pieces — two control points are replaced by one, their midpoint. Subsequent knot removal is done by solving the appropriate linear least squares problems.

```
int mbs_multiKnotRemoveClosedf ( int degree, int *lastknot,
                                 float *knots,
                                 int ncurves, int spdimen,
                                 int inpitch, int outpitch,
                                 float *ctlpoints,
                                 int knotnum );
```

The procedure `mbs_multiKnotRemoveClosedf` removes knots from representations of closed B-spline curves.

The parameters `degree`, `ncurves` and `spdimen` specify the degree and number of curves and space dimension respectively. The parameters `*lastknot` and `knots` initially describe the initial knot sequence. After return these parameters describe the final knot sequence. The parameter `knotnum` specifies the number of knot to be removed. The parameters `inpitch` and `outpitch` specify the initial and final pitch of the array of control points, `ctlpoints`, i.e. the distances between the beginnings of control polygons of consecutive curves. The array `ctlpoints` initially contains the control points of the initial representation of the curves; the procedure replaces them by the control points of the final representation, obtained by removing the knot.

```
#define mbs_KnotRemoveClosedC1f(degree,lastknot,knots,coeff, \
    knotnum) \
  mbs_multiKnotRemoveClosedf(degree,lastknot,knots,1,1,0,0,coeff, \
    knotnum)
#define mbs_KnotRemoveClosedC2f(degree,lastknot,knots,ctlpoints, \
    knotnum) \
  mbs_multiKnotRemoveClosedf(degree,lastknot,knots,1,2,0,0, \
    (float*)ctlpoints,knotnum)
#define mbs_KnotRemoveClosedC3f(degree,lastknot,knots,ctlpoints, \
    knotnum) ...
#define mbs_KnotRemoveClosedC4f(degree,lastknot,knots,ctlpoints, \
    knotnum) ...
```

The four macros above call `mbs_multiKnotRemoveClosedf` in order to remove the indicated knot from the representation of *one* periodic spline function or closed

B-spline curve in the space of dimension two, three and four. The parameters are described with the procedure `mbs_multiKnotRemoveClosedf`.

```
void mbs_multiRemoveSuperfluousKnotsf ( int ncurves,
                                        int spdimen, int degree,
                                        int *lastknot,
                                        float *knots,
                                        int inpitch, int outpitch,
                                        float *ctlpoints );
```

The procedure `mbs_multiRemoveSuperfluousKnotsf` removes knots from the representation of B-spline curves in such a way, that all the remaining knots have multiplicity of the degree (the parameter `degree`) plus one. The curves are not changed, but problems caused by the presence of such knots, may be avioded (e.g. a B-spline function, whose all knots are the same, is the zero function, hence any set of B-spline functions with a knot of multiplicity greater than $n+1$ is not a basis).

The computation is done „at the spot", i.e. the area initially occupied by the initial representation, after return contains the new representation of the curves. The knots are removed by moving the data (knots and control points) in the arrays, without any numerical computations.

7.7.3 The Oslo algorithm

The Oslo algorithm is a method of finding a representation of B-spline curves corresponding to a knot sequence $\hat{u}_0, \ldots, \hat{u}_{\hat{N}}$ given a representation based on a subsequence $u_0, \ldots, u_N$. As opposed to the Boehm algorithm (see Section 7.7.1), which inserts one knot at a time (and which may be used a number of times if necessary), here all knots are inserted at the same time.

If the control points $\mathbf{d}_i$ of a B-spline curve of degree n correspond to the knots $u_0, \ldots, u_N$, and the control points $\hat{\mathbf{d}}_l$ correspond to the knots $\hat{u}_0, \ldots, \hat{u}_{\hat{N}}$, then

$$\hat{\mathbf{d}}_l = \sum_{i=0}^{N-n-1} a_{il}^n \mathbf{d}_i, \tag{7.20}$$

where the coefficients a_{kl}^n are given by the recursive formulae

$$a_{kl}^0 = \begin{cases} 1 & \text{for } u_k \le \hat{u}_l < u_{k+1}, \\ 0 & \text{else,} \end{cases} \tag{7.21}$$

$$a_{il}^n = \frac{\hat{u}_{l+n} - u_i}{u_{i+n} - u_i} a_{il}^{n-1} + \frac{u_{i+n+1} - \hat{u}_{l+n}}{u_{i+n+1} - u_{i+1}} a_{i+1,l}^{n-1}. \tag{7.22}$$

The implementation of the Oslo algorithm in the library `libmultibs` is such that initially the matrix A, whose coefficients are a_{il}^n, is computed and then it is multiplied by the matrix of the control points $\mathbf{d}_0, \ldots, \mathbf{d}_{N-n-1}$.

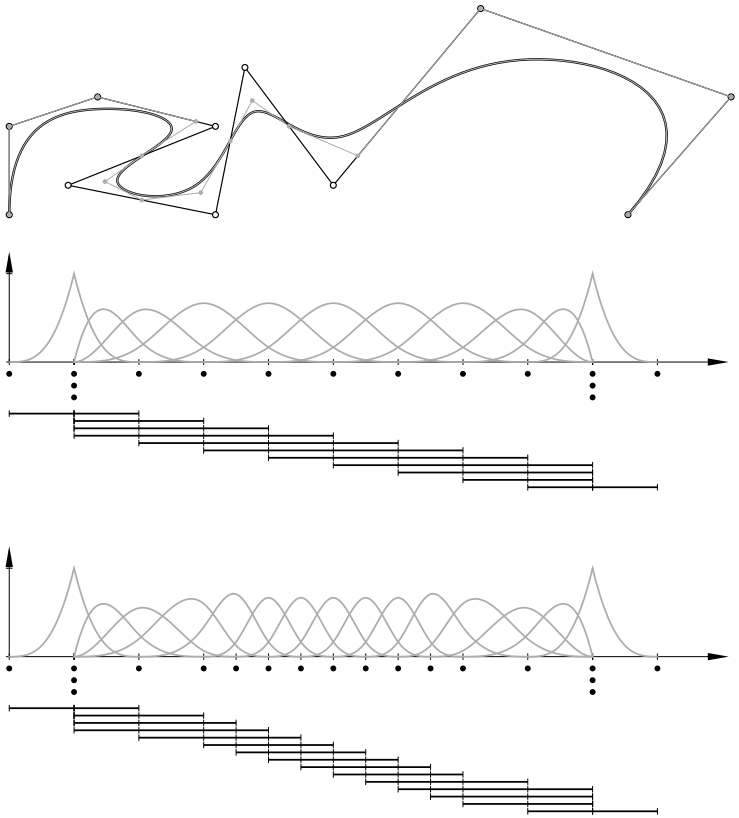


Figure 7.5. Inserting knots with the Oslo algorithm.

The matrix A makes it possible also to remove a number of knots at a time, by solving an over-definite system of linear equations (with more equations than unknowns). Such a system, even when it is consistent, is solved best as a linear least squares problem.

The matrix A is represented as a band matrix, by an array with its profile (i.e. table of positions of nonzero coefficients in consecutive columns) and an array with nonzero coefficients. A detailed description of this representation and the related procedures is in Section 3.2.

```
boolean mbs_OsloKnotsCorrectf ( int lastuknot, const float *uknots,
                                int lastvknot, const float *vknots );
```

The procedure `mbs_OsloKnotsCorrectf` verifies, whether two given sequences of knots make it possible to construct the matrix A. The conditions verified are as follows: both sequences are nondecreasing, and the first sequence of (`lastuknot`$+1$ numbers, given in the array `uknots`), is a subsequence of the second sequence of length (`lastvknot`$+1$, given in the array `vknots`). If these conditions are satisfied, the procedure returns `true`, otherwise it returns `false`.


```
int mbs_BuildOsloMatrixProfilef ( int degree,
                                  int lastuknot, const float *uknots,
                                  int lastvknot, const float *vknots,
                                  bandm_profile *prof );
```

The procedure `mbs_BuildOsloMatrixProfilef`, given the degree of the representation (`degree`) and two knot sequences (see description of the procedure `mbs_OsloKnotsCorrectf` above), constructs the profile of the matrix of the representation transformation. The profile is stored in the array `prof`, whose length must be at least `lastuknot` $-$ `degree` $+ 1$ (the number of columns plus one).

The procedure returns the number of nonzero coefficients of the matrix, which is the length of the array to be allocated for storing the coefficients.

```
void mbs_BuildOsloMatrixf ( int degree, int lastuknot,
                            const float *uknots,
                            const float *vknots,
                            const bandm_profile *prof, float *a );
```

The procedure `mbs_BuildOsloMatrixf` computes the coefficients of the matrix of curve representation transformation, using the Oslo algorithm. The parameters are: `degree` — degree of the curve, `uknots` — array of knots of the initial representation, of length `lastuknot`$+1$, `vknots` — array of knots of the final representation, whose length is determined based on the contents of the arrays, therefore there is no parameter to specify it. The knot sequences have to satisfy the conditions verified by the procedure `mbs_OsloKnotsCorrectf`.

The array `prof` contains the description of the matrix structure (the profile), which has to be found earlier, with the procedure `mbs_BuildOsloMatrixProfilef`. The coefficients computed by the `mbs_BuildOsloMatrixf` are stored in the array `a`, whose length has been computed by `mbs_BuildOsloMatrixProfilef`.

```
void mbs_multiOsloInsertKnotsf ( int ncurves, int spdimen,
                            int degree,
                            int inlastknot, const float *inknots,
                            int inpitch, float *inctlpoints,
                            int outlastknot, const float *outknots,
                            int outpitch, float *outctlpoints );
```

The procedure `mbs_multiOsloInsertKnotsf` inserts a number of knots to the representation of `ncurves` B-spline curves located in the space of dimension `spdimen`. The degree of the curves is specified by the parameter `degree`. The initial representation consists of the knots given in the array `inknots` (of length

inlastknot + 1) and the control polygons stored in the array inctlpoints, whose pitch is given by inpitch.

The final representation is based on the knot sequence of length outlastknot+1, given in the array outknots, and the initial knot sequence must be a subsequence of this sequence.

The procedure sets up the appropriate matrix with the Oslo algorithm, and then it multiplies it by the matrix of the given control points of the curves.

If the values of the parameters inlastknot and outlastknot are the same, then the procedure assumes that the knot sequences are identical (which is *not* verified) only copies data from the array inctlpoints to outctlpoints (according to the pitches of the arrays, specified by the parameters inpitch and outpitch respectively).

```
void mbs_multiOsloRemoveKnotsLSQf ( int ncurves, int spdimen,
                                    int degree,
                                    int inlastknot, const float *inknots,
                                    int inpitch, float *inctlpoints,
                                    int outlastknot, const float *outknots,
                                    int outpitch, float *outctlpoints );
```

The procedure mbs_multiOsloRemoveKnotsLSQf removes a number of knots from the representation of given ncurves B-spline curves located in the space of dimension spdimen. The degree of the curves is the value of the parameter degree. The initial representation constsis of the knot sequence stored in the array inknots (of length inlastknot + 1) and the control polygons stored in the array inctlpoints, whose pitch is inpitch.

The final representation is based on the knot sequence of length outlastknot+1, given in the array outknots, and this sequence must be a subsequence of the initial sequence. Moreover, no knot of the final knot sequence may have multiplicity greater than degree + 1 (otherwise the matrix described above would have columns linearly dependent).

The procedure constructs the appropriate matrix, and then it solves a linear least squares problem with this matrix.

If the parameters inlastknot and outlastknot have the same value, then the procedure assumes that the knot sequences are identical (which is not verified) and it only copies data from the array inctlpoints to outctlpoints according to the pitches of the arrays, specified by the parameters inpitch and outpitch respectively).

7.7.4 Maximal knot insertion

The procedures described in this section may be used to insert knots into the representation of B-spline curves and patches in such a way, that the multiplicity of

each knot be equal to the degree plus one. In this way a particular B-spline representation is obtained; it consists of the representations of polynomial arcs in local Bernstein bases, i.e. a piecewise Bézier representation. Such a representation makes it possible e.g. to quickly compute points of the curves (with the Horner scheme) and algebraic operations (like multiplication) on spline functions and curves. The procedures described here do not remove unnecessary knots (of multiplicity greater than degree+1). The procedures which do remove the unnecessary knots (thus producing a "clean" result) are described in the next section.

```
void mbs_multiMaxKnotInsf ( int ncurves, int spdimen, int degree,
                            int inlastknot, const float *inknots,
                            int inpitch, const float *inctlpoints,
                            int *outlastknot, float *outknots,
                            int outpitch, float *outctlpoints,
                            int *skipl, int *skipr );
```

The procedure mbs_multiMaxKnotInsf inserts knots to the representation of ncurves B-spline curves of degree degree in the space of dimension spdimen.

The initial representation of the curves is given by the parameters inlastknot (number of the last knot), inctlpoints (array with the knots), inpitch (pitch, i.e. the distance between the beginnings of the given control polylines), and inctlpoints (array with the control points).

The procedure constructs the representation of the curves corresponding to the knot sequence with all internal knots (see Section 7.1.3) of multiplicity degree + 1, and the boundary knots have multiplicity degree or degree + 1.

The multiplicities of the extremal knots remain unchanged, therefore the resulting representation may contain unnecessary knots and control points. The parameters *skipl and *skipr upon return indicate the number of unnecessary knots and control points from the left side and the right side respectively.

The new representation is stored in the arrays outknots (knot sequence, the index of the last knot is assigned to the parameter *outlastknot) and outctlpoints (control polygons, the pitch of this array is specified by the *input* parameter outpitch).

If the initial knot sequence contains knots of multiplicity greater than desired, the procedure begins the computations with removing them (from a copy of the data), with use of the procedure mbs_multiRemoveSuperfluousKnots). Then the procedure mbs_multiOsloInsertKnotsf) is called in order to insert the knots using the Oslo algorithm.

The lengths of the arrays necessaty to accomodate the new representation of the curves may be found with use of the procedure mbs_LastknotMaxInsf, which computes the index of the last knot of the new representation.

```
#define mbs_MaxKnotInsC1f(degree,inlastknot,inknots,incoeff, \
    outlastknot,outknots,outcoeff,skipl,skipr) \
  mbs_multiMaxKnotInsf(1,1,degree,inlastknot,inknots,0,incoeff, \
    outlastknot,outknots,0,outcoeff,skipl,skipr)
#define mbs_MaxKnotInsC2f(degree,inlastknot,inknots,inctlpoints, \
    outlastknot,outknots,outctlpoints,skipl,skipr) \
  mbs_multiMaxKnotInsf(1,2,degree,inlastknot,inknots,0, \
    (float*)inctlpoints,outlastknot,outknots,0, \
    (float*)outctlpoints,skipl,skipr)
#define mbs_MaxKnotInsC3f(degree,inlastknot,inknots,inctlpoints, \
    outlastknot,outknots,outctlpoints,skipl,skipr) ...
#define mbs_MaxKnotInsC4f(degree,inlastknot,inknots,inctlpoints, \
    outlastknot,outknots,outctlpoints,skipl,skipr) ...
```

The four macros above call `mbs_multiMaxKnotInsf` in order to construct the representation of one scalar spline function or a B-spline curve in the space of dimension 2, 3, 4, with all internal knots of multiplicity equal to the degree plus one. The parameters are described with the procedure `mbs_multiMaxKnotInsf`.

7.7.5 Conversion of curves and patches to the piecewise Bézier form

The procedures described below convert B-spline curves and patches to the piecewise Bézier form, using the procedure `mbs_multiMaxKnotInsf`. They may help to draw curves and patches.

```
void mbs_multiBSCurvesToBezf ( int spdimen, int ncurves,
                               int degree, int lastinknot,
                               const float *inknots,
                               int inpitch, const float *inctlp,
                               int *kpcs, int *lastoutknot,
                               float *outknots,
                               int outpitch, float *outctlp );
```

The procedure `mbs_multiBSCurvesToBezf` converts `ncurves` B-spline curves of degree `degree`, located in the space of dimension `spdimen` to the piecewise Bézier form.

The parameters, which describe the given representation are `lastinknot` (index of the last knot), `inknots` (array with the knot sequence), `inpitch` and `inctlp` (pitch and the array with the control points).

The value of `*kpcs` upon return from the proceure is equal to the number of polynomial arcs of each curve. The parameter `*lastoutknot` is the index of the last knot of the sequence of the resulting representation, the array `outknots` contains these knots, the *input* parameter `outpitch` specifies the pitch of the array `outctlp`, in which the procedure stores the control points of the Bézier representations of the

polynomial arcs. More precisely, to each of the B-spline curves there correspond `*kpcs*(degree+1)*spdimen` floating point numbers; each polynomial arc is represented by `(degree+1)*spdimen` consecutive numbers (coordinates of `degree+1` points); the value of the parameter `outpitch` is the distance between the beginnings of the representations of the first arcs of the consecutive B-spline curves.

If the parameter `kpcs`, `lastoutknot` or `outknots` is `NULL`, then the procedure does not output the corresponding information.

```
#define mbs_BSToBezC1f(degree,lastinknot,inknots,incoeff,kpcs, \
    lastoutknot,outknots,outcoeff) \
  mbs_multiBSCurvesToBezf(1,1,degree,lastinknot,inknots,0,incoeff,\
    kpcs,lastoutknot,outknots,0,outcoeff)
#define mbs_BSToBezC2f(degree,lastinknot,inknots,inctlp,kpcs, \
    lastoutknot,outknots,outctlp) \
  mbs_multiBSCurvesToBezf(2,1,degree,lastinknot,inknots,0, \
    (float*)inctlp,kpcs,lastoutknot,outknots,0,(float*)outctlp)
#define mbs_BSToBezC3f(degree,lastinknot,inknots,inctlp,kpcs, \
    lastoutknot,outknots,outctlp) ...
#define mbs_BSToBezC4f(degree,lastinknot,inknots,inctlp,kpcs, \
    lastoutknot,outknots,outctlp) ...
```

The above macros call `mbs_multiBSCurvesToBezf` in order to obtain the piecewise Bézier representation of *one* spline function or a B-spline curve in the space of dimension 2, 3 and 4. The parameters are described with the procedure `mbs_multiBSCurvesToBezf`.

```
void mbs_BSPatchToBezf ( int spdimen,
                         int degreeu, int lastuknot,
                         const float *uknots,
                         int degreev, int lastvknot,
                         const float *vknots,
                         int inpitch, const float *inctlp,
                         int *kupcs, int *lastoutuknot,
                         float *outuknots,
                         int *kvpcs, int *lastoutvknot,
                         float *outvknots,
                         int outpitch, float *outctlp );
```

The procedure `mbs_BSPatchToBezf` finds the representation of a B-spline patch with knots of multiplicities equal to the degree plus one, for both parameters. In other words, it is a piecewise Bézier representation. Such a represnetation may be convenient when the patch is to be drawn. The procedure may process patches with clamped boundary as well as with free boundary.

The parameter `spdimen` specifies the space dimension. The degree of the patch is given by the parameters `degreeu` and `degreev`, the knot sequences of the given

representation are given by the parameters `lastuknot`, `uknots`, `lastvknot` and `vknots`, and the control points are given in the array `inctlp`, whose pitch (distance between the beginnings of consecutive columns of the control net) specified with the parameter `inpitch`.

The parameters `*kupcs` and `*kvpcs` may be used to pass the information about the number of polynomial pieces of the patch; the patch consists of `*kupcs` "strips", each of which consisting of `*kvpcs` polynomial patches. The parameters `lastoutuknot`, `outuknots`, `lastvknot` and `outvknots` are used to output the knot sequences of the final representation. If any of the four parameters is `NULL`, then the corresponding information is not output by the procedure (it is unnecessary for drawing the patch).

The control points of the final patch representation are stored in the array `outctlp`, whose pitch is `outpitch` (it is an *input* parameter). The pitch should be greater than $d(m+1)k_v$, where d is the dimension of the space (the value of the parameter `spdimen`), m (the value of `degreev`) is the degree of the patch with respect to v, and k_v is the number of intervals between knots in the interval $[v_m, v_{M-m}]$ (the number M is the value of the parameter `lastvknot`). The number k_v is assigned by the procedure `mbs_BSPatchToBezf` to the parameter `kvpcs`, but it may be obtained before calling it, with the `mbs_NumKnotIntervalsf` procedure.

The number of columns of the final representation of the patch is $(n+1)k_u$, where n is the degree of the patch with respect to u, and k_u is the number of "strips", of which the patch consists. It may also be computed earlier, by calling `mbs_NumKnotIntervalsf`.

The main computation (mainly knot insertion) is done by the procedure `mbs_multiMaxKnotInsf`.

Example. Suppose that the patch is defined by Formula (7.12) with two non-decreasing knot sequences, $u_0, \ldots, u_N$ and $v_0, \ldots, v_M$, stored respectively in the arrays `u` and `v`. The degree is n with respect to u and m with respect to v. The control points in the d-dimensional space are orgaanized in the columns and stored in the array `cp`. The `i`-th column, for $i \in \{0, \ldots, N-n-1\}$, consists of $M-m$ points, therefore it is represented by $(M-m)d$ floating point answers.

```
ku = mbs_NumKnotIntervalsf ( n, N, u );
kv = mbs_NumKnotIntervalsf ( m, M, v );
pitch = (m+1)d*kv;
b = pkv_GetScratchMemf ( pitch*ku*(n+1) );
mbs_BSPatchToBezf ( d, n, N, u, m, M, v, d*(M-m), cp,
                    &ku, NULL, NULL, &kv, NULL, NULL, pitch, b );
```

After executing the above code the array `b` contains the Bézier control points of the polynomial pieces of the B-spline patch. To move the control points of the `j`-th Bézier patch from the `i`-th strip (counting from 0) to the array `c` (of length at least $(n+1)(m+1)d$), and obtain a "packed" control net (without unused areas between

he columns), one may use the code

```
md = (m+1)d;       /* length of one column of each Bézier patch */
start = (n+1)i*pitch + md*j;    /* position of the first point */
pkv_Selectf ( n+1, md, pitch, md, &b[start], c );
```

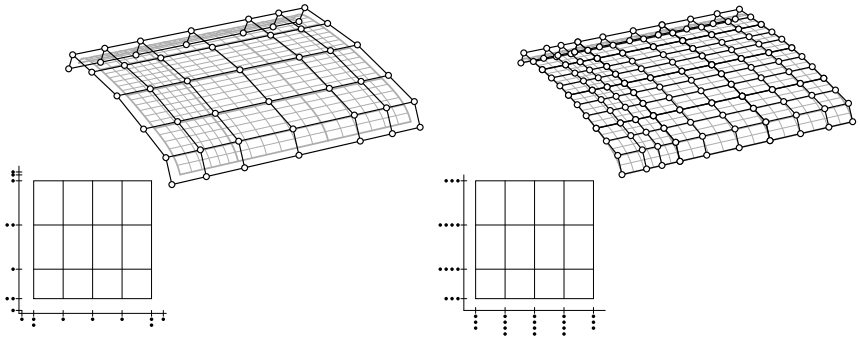


Figure 7.6. A B-spline patch and its piecewise Bézier representation.

7.8 Lane-Riesenfeld algorithm

```
boolean mbs_multiLaneRiesenfeldf ( int spdimen, int ncurves,
                    int degree,
                    int inlastknot, int inpitch, const float *incp,
                    int *outlastknot, int outpitch, float *outcp );
```

```
#define mbs_LaneRiesenfeldC1f(degree,inlastknot,incp,outlastknot, \
    outcp) \
  mbs_multiLaneRiesenfeldf ( 1, 1, degree, inlastknot, 0, incp, \
    outlastknot, 0, outcp )
#define mbs_LaneRiesenfeldC2f(degree,inlastknot,incp,outlastknot, \
    outcp) \
  mbs_multiLaneRiesenfeldf ( 2, 1, degree, inlastknot, 0, \
    (float*)incp, outlastknot, 0, (float*)outcp )
#define mbs_LaneRiesenfeldC3f(degree,inlastknot,incp,outlastknot, \
    outcp) ...
#define mbs_LaneRiesenfeldC4f(degree,inlastknot,incp,outlastknot, \
    outcp) ...
```

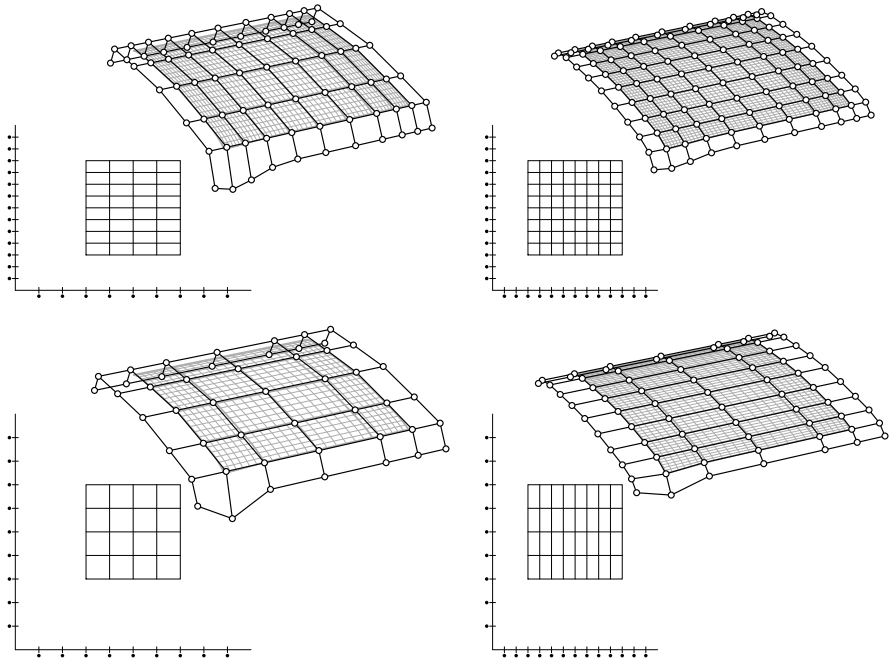


Figure 7.7. Application of the Lane-Riesenfeld algorithm to a B-spline patch

7.9 Bézier curves and patches subdivision

Bézier curves and patches may be divided into pieces with the de Casteljau algorithm, and there are two cases implemented separately. The first case is bisection of the domain (e.g. the division of the interval $[0.25, 0.5]$ into the parts $[0.25, 0.375]$ and $[0.375, 0.5]$, of equal length), and the algorithm computes only averages of pairs of numbers. The second case is general; the domain may be divided into intervals or rectangles at any place, which includes also the possibility of extrapolation.

```
void mbs_multiBisectBezCurvesf ( int degree, int ncurves,
                                 int spdimen, int pitch,
                                 float *ctlp, float *ctlq );
```

The procedure `mbs_multiBisectBezCurvesf` divides `ncurves` Bézier curves of degree `degree` in the space of dimension `spdimen`. The domain (an interval $[a, b]$) is divided into two parts of equal length.

The control points are given in the array `ctlp`, whose pitch is `pitch`. Upon return this array contains the control points of the second arc of each curve (the one corresponding to the interval $[\frac{a+b}{2}, b]$). In the array `ctlq` the procedure stores the control points of the arcs corresponding to the interval $[a, \frac{a+b}{2}]$. The pitch of this array is the same as the pitch of `ctlp` (i.e. equal to the value of the parameter `pitch`).

```
#define mbs_BisectBC1f(degree,ctlp,ctlq) \
  mbs_multiBisectBezCurvesf(degree,1,1,0,ctlp,ctlq)
#define mbs_BisectBC2f(degree,ctlp,ctlq) \
  mbs_multiBisectBezCurvesf(degree,1,2,0,(float*)ctlp,(float*)ctlq)
#define mbs_BisectBC3f(degree,ctlp,ctlq) ...
#define mbs_BisectBC4f(degree,ctlp,ctlq) ...
```

The above macros call `mbs_multiBisectBezCurvesf` in order to bisect one polynomial or Bézier curve in the space of dimension 2, 3 or 4, i.e. to compute its local representations corresponding to the halves of the initial interval.

```
#define mbs_BisectBP1uf(degreeu,degreev,ctlp,ctlq) \
  mbs_multiBisectBezCurvesf(degreeu,1,(degreev+1),0,ctlp,ctlq)
#define mbs_BisectBP1vf(degreeu,degreev,ctlp,ctlq) \
  mbs_multiBisectBezCurvesf(degreev,degreeu+1,1,degreev+1, \
  ctlp,ctlq)
#define mbs_BisectBP2uf(degreeu,degreev,ctlp,ctlq) \
  mbs_multiBisectBezCurvesf(degreeu,1,2*(degreev+1),0, \
  (float*)ctlp,(float*)ctlq)
```

```
#define mbs_BisectBP2vf(degreeu,degreev,ctlp,ctlq) \
  mbs_multiBisectBezCurvesf(degreev,degreeu+1,2,2*(degreev+1), \
  (float*)ctlp,(float*)ctlq)
#define mbs_BisectBP3uf(degreeu,degreev,ctlp,ctlq) ...
#define mbs_BisectBP3vf(degreeu,degreev,ctlp,ctlq) ...
#define mbs_BisectBP4uf(degreeu,degreev,ctlp,ctlq) ...
#define mbs_BisectBP4vf(degreeu,degreev,ctlp,ctlq) ...
```

The above macros call `mbs_multiBisectBezCurvesf` in order to divide the rectangular domain of one bivariate polynomial (given in a tensor product Bernstein basis) or a Bézier patch (of dimension 2, 3 or 4) into two equal parts and to compute the local representations of the polynomial or the patch related with the two parts. The macros with the letter `u` in the identifier bisect the interval of the first parameter, and the macros with the letter `v` bisect the interval of the second parameter of the polynomial or the patch.

```
void mbs_multiDivideBezCurvesf ( int degree, int ncurves,
                                 int spdimen, int pitch, float t,
                                 float *ctlp, float *ctlq );
```

The procedure `mbs_multiDivideBezCurvesf` divides `ncurves` Bézier curves of degree `degree` in the space of dimension `spdimen`. The domain (the interval $[0,1]$) is divided at the point `t`, specified by the parameter `t`, and if $t \notin [0,1]$, then the division is in fact an extrapolation.

The control points are given in the array `ctlp`, whose pitch is `pitch`. Upon return this array contains the control points of the second arc of each curve (related with the interval $[t,1]$). The array `ctlq` is filled with the control points of the first arcs, related with the interval $[0,t]$. The pitch of this array is the same as the pitch of `ctlp` (it is equal to the value of the parameter `pitch`).

```
#define mbs_DivideBC1f(degree,t,ctlp,ctlq) \
  mbs_multiDivideBezCurvesf(degree,1,1,0,t,ctlp,ctlq)
#define mbs_DivideBC2f(degree,t,ctlp,ctlq) \
  mbs_multiDivideBezCurvesf(degree,1,2,0,t, \
  (float*)ctlp,(float*)ctlq)
#define mbs_DivideBC3f(degree,t,ctlp,ctlq) ...
#define mbs_DivideBC4f(degree,t,ctlp,ctlq) ...
```

The above macros call `mbs_multiDivideBezCurvesf` in order to divide the domain of one polynomial or one Bézier curve (two-, three- or four-dimensional) at the proportion $t : 1-t$, where `t` is the value of the parameter `t`, and to find the local representations of the polynomial or the curve.


```
#define mbs_DivideBP1uf(degreeu,degreev,u,ctlp,ctlq) \
  mbs_multiDivideBezCurvesf(degreeu,1,(degreev)+1,0,u,ctlp,ctlq)
#define mbs_DivideBP1vf(degreeu,degreev,v,ctlp,ctlq) \
  mbs_multiDivideBezCurvesf(degreev,(degreeu)+1,1,degreev+1,v, \
    ctlp,ctlq)
#define mbs_DivideBP2uf(degreeu,degreev,u,ctlp,ctlq) \
  mbs_multiDivideBezCurvesf(degreeu,1,2*(degreev)+1,0,u, \
    (float*)ctlp,(float*)ctlq)
#define mbs_DivideBP2vf(degreeu,degreev,v,ctlp,ctlq) \
 mbs_multiDivideBezCurvesf(degreev,(degreeu)+1,2,2*(degreev)+1,v, \
    (float*)ctlp,(float*)ctlq)
#define mbs_DivideBP3uf(degreeu,degreev,u,ctlp,ctlq) ...
#define mbs_DivideBP3vf(degreeu,degreev,v,ctlp,ctlq) ...
#define mbs_DivideBP4uf(degreeu,degreev,u,ctlp,ctlq) ...
#define mbs_DivideBP4vf(degreeu,degreev,v,ctlp,ctlq) ...
```

The macros calling `mbs_multiDivideBezCurvesf` in order to divide a bivariate polynomial or a Bézier patch in the space of dimension 2, 3 or 4 — in the "u" direction (i.e. the interval of the first patch parameter is to be divided) or in the "v" direction (the interval of the second parameter is to be divided). The number u or v, which is the value of the parameter `u` or `v` is the point of division of the interval $[0,1]$ (the interval of the parameter is divided in the proportion e.g. $u : 1-u$).

7.10 Degree elevation

Degree elevation is a computation of a new representation of curves in the Bernstein or B-spline basis of degree greater by a specified amount.

7.10.1 Degree elevation of Bézier curves and patches

```
void mbs_multiBCDegElevf ( int ncurves, int spdimen,
                           int inpitch, int indegree,
                           const float *inctlpoints,
                           int deltadeg,
                           int outpitch, int *outdegree,
                           float *outctlpoints );
```

The procedure `mbs_multiBCDegElevf` performs the degree elevation of `ncurves` Bézier curves of degree `indegree` in the space of dimension `spdimen`, to the degree `indegree` + `deltadeg` (the resulting degree is assigned to the parameter `*outdegree`).

The control polygons of the curves are given in the array `inctlpoints`, whose pitch is `inpitch`. The computed control polygons are stored by the procedure in the array `outctlpoints`, with the pitch `outpitch`.

```
#define mbs_BCDegElevC1f(indegree,incoeff,deltadeg, \
    outdegree,outcoeff) \
  mbs_multiBCDegElevf ( 1, 1, 0, indegree, incoeff, deltadeg, \
    0, outdegree, outcoeff )
#define mbs_BCDegElevC2f(indegree,inctlpoints,deltadeg, \
    outdegree,outctlpoints) \
  mbs_multiBCDegElevf ( 1, 2, 0, indegree, (float*)inctlpoints, \
    deltadeg, 0, outdegree, (float*)outctlpoints )
#define mbs_BCDegElevC3f(indegree,inctlpoints,deltadeg, \
    outdegree,outctlpoints) ...
#define mbs_BCDegElevC4f(indegree,inctlpoints,deltadeg, \
    outdegree,outctlpoints) ...
```

The four macros above may be used for degree elevation of one polynomial (given by the coefficients in the Bernstein basis) or a Bézier curve in the space of dimension 2, 3 or 4. The parameters of the macros are described with the procedure `mbs_multiBCDegElevf`.


```
void mbs_BCDegElevPf ( int spdimen,
                       int indegreeu, int indegreev,
                       const float *inctlp,
                       int deltadegu, int deltadegv,
                       int *outdegreeu, int *outdegreev,
                       float *outctlp );
```

The procedure `mbs_BCDegElevPf` performs the degree elevation of a Bézier patch in the space of dimension `spdimen`, with respect to one or both parameters.

The parameters `indegu` and `indegv` specify the degree of the initial patch representation, with respect to its two parameters. The array `inctlp` contains the control points of the patch, organized in subsequent columns. The array is packed, i.e. without unused areas between consecutive columns, hence the pitch is equal to the length of the column representation: `(indegu+1)*spdimen`. The array `outctlp`, in which the procedure stores the control points of the resulting representation, is packed in a similar way.

The parameters `deltadegu` and `deltadegv` must be nonnegative. They specify the numbers, by which the degrees of the patch are to increase. The final degrees (sums of the initial degrees and the increments) are assigned to the parameters `*outdegu` and `outdegv`.

```
#define mbs_BCDegElevP1f(indegreeu,indegreev,incoeff, \
    deltadegu,deltadegv,outdegreeu,outdegreev,outcoeff) \
  mbs_BCDegElevPf ( 1, indegreeu, indegreev, incoeff, \
    deltadegu, deltadegv, outdegreeu, outdegreev, outcoeff )
#define mbs_BCDegElevP2f(indegreeu,indegreev,inctlp, \
    deltadegu,deltadegv,outdegreeu,outdegreev,outctlp) \
  mbs_BCDegElevPf ( 2, indegreeu, indegreev, (float*)inctlp, \
    deltadegu, deltadegv, outdegreeu, outdegreev, (float*)outctlp )
#define mbs_BCDegElevP3f(indegreeu,indegreev,inctlp, \
    deltadegu,deltadegv,outdegreeu,outdegreev,outctlp) ...
#define mbs_BCDegElevP4f(indegreeu,indegreev,inctlp, \
    deltadegu,deltadegv,outdegreeu,outdegreev,outctlp) ...
```

The above macros may be used for degree elevation of a bivariate polynomial or a Bézier patch in the space of dimension 2, 3 or 4. They call the procedure `mbs_BCDegElevPf`.

7.10.2 Degree elevation of B-spline curves and patches

```
void mbs_multiBSDegElevf ( int ncurves, int spdimen,
                           int indegree, int inlastknot,
                           const float *inknots,
                           int inpitch, const float *inctlpoints,
                           int deltadeg,
                           int *outdegree, int *outlastknot,
                           float *outknots,
                           int outpitch, float *outctlpoints,
                           boolean freeend );
```

The procedure `mbs_multiBSDegElevf` performs the degree elevation of `ncurves` B-spline curves of degree `indegree` in the space of dimension `spdimen`, up to the degree `indegree` + `deltadeg`, which is assigned to the parameter `*outdegree`.

The procedure is able to process curves with clamped or free ends. If the value of the parameter `freeend` is `false`, then the resulting representation of the curves has clamped ends, with the only external knots being the extremal knots (see Section 7.1.3). If the value of `freeend` is `true`, then the resulting representation has free ends. The knot sequence of this representation is obtained by increasing the multiplicities of all knots by the value of the parameter `deltadeg`, and then by rejecting knots from the beginning and end of the sequence so as to obtain $\hat{u}_n < \hat{u}_{n+1}$ and $\hat{u}_{N-n} > \hat{u}_{N-n-1}$ (n here denotes the degree of the result representation, and N is the number of its last knot).

The method of degree elevation does not depend on the value of `freeend`. The representation obtained after the degree elevation if one with clamped ends. For `freeend=true` the procedure calls `mbs_multiBSChangeLeftKnotsf` and `mbs_multiBSChangeRightKnotsf`. This involves additional rounding errors. The procedure `mbs_multiBSDegElevf` may be used for degree elevation of a closed curve; the parameter `freeend` should then be `true`, to obtain a closed representation of higher degree. In such a representation the appropriate number of initial control points coincide with the final control points *up to the rounding errors*.

The initial representation is given in the arrays `inknots` (knots, their number is `inlastknot` + 1) and `inctlpoints` (control points the pitch of this array is `inpitch`).

The resulting representation is stored in the arrays `outknots` (knots, their number is assigned to `*outlastknot`) and `outctlpoints` (control poitns, the pitch of this array is specified by `outpitch`).

It is necessary to provide the arrays long enough to accomodate the result. The rule is as follows: if the last knot of the representation of degree n has the number N, the number of polynomial arcs of the curve is l (it may be found using the procedure `mbs_NumKnotIntervalsf`), and the resulting representation has the

degree n', then the last knot of this representation has the number

$$N' = N + (l + 1 - d_0 - d_1)(n' - n),$$

where d_0 and d_1 are numbers such that

$$u_n = \cdots = u_{n+d_0} < u_{n+d_0+1} \quad \text{and} \quad u_{N-n-d_1-1} < u_{N-n-d_1} = \cdots = u_{N-n}.$$

The number of control points of the new curve representation is is $N' - n'$. For m curves in the space of dimension d it is necessary to allocate an array of length $N' + 1$ floating point numbers for the knots and an array of length $(N' - n)md$ floating point numbers for the control points.

```
#define mbs_BSDegElevC1f(indegree,inlastknot,inknots,incoeff, \
    deltadeg,outdegree,outlastknot,outknots,outcoeff,freeend) \
  mbs_multiBSDegElevf(1,1,indegree,inlastknot,inknots,0,incoeff, \
    deltadeg,outdegree,outlastknot,outknots,0,outcoeff,freeend)
#define mbs_BSDegElevC2f(indegree,inlastknot,inknots,inctlpoints, \
    deltadeg,outdegree,outlastknot,outknots,outctlpoints,freeend) \
  mbs_multiBSDegElevf(1,2,indegree,inlastknot,inknots, \
    0,(float*)inctlpoints,deltadeg, \
    outdegree,outlastknot,outknots,0,(float*)outctlpoints,freeend)
#define mbs_BSDegElevC3f(indegree,inlastknot,inknots,inctlpoints, \
  deltadeg,outdegree,outlastknot,outknots,outctlpoints,freeend) ...
#define mbs_BSDegElevC4f(indegree,inlastknot,inknots,inctlpoints, \
  deltadeg,outdegree,outlastknot,outknots,outctlpoints,freeend) ...
```

Four macros, which call `mbs_multiBSDegElevf` for degree elevation of one scalar spline function of B-spline curve in the space of dimension 2, 3 or 4.

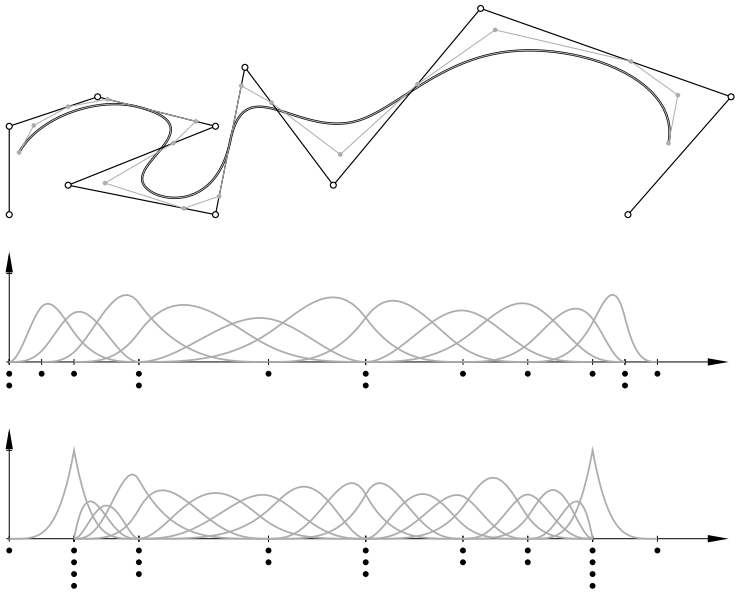


Figure 7.8. Degree elevation of a planar B-spline curve from 3 to 4.

```
void mbs_multiBSDegElevClosedf ( int ncurves, int spdimen,
              int indegree, int inlastknot, const float *inknots,
              int inpitch, const float *inctlpoints,
              int deltadeg,
              int *outdegree, int *outlastknot,
              float *outknots, int outpitch, float *outctlpoints );
```

The procedure `mbs_multiBSDegElevClosedf` performs degree elevation of closed B-spline curves.

The parameter `ncurves` specifies the number of curves, the value of the parameter `spdimen` is the dimension of the space in which they are located.

The parameters `indegree`, `inlastknot`, `inknots`, `inpitch`, `inctlpoints` describe the input data — degree n, number N of the last knot, the knot sequence $u_0, \ldots, u_N$ the pitch of the array with the control points and that array respectively. The parameter `deltadeg` (whose value must be nonnegative) specifies the degree increment.

The parameters `*outdegree` and `*outlastknot` are variables, to which the assigns the final representation degree and the number of the last knot of this representation. The final knot sequence is stored in the array `outknots`. The parameter `outpitch` specifies the pitch of the array `outctlpoints`, used to store the control points of the final representation.

The number of the last knot of the resulting representation of the closed curve is

$$N' = N + (l + 1 + r - d_0 - d_1)(n' - n),$$

where r is the multiplicity of the knot u_n in the given representation of degree n (without counting u_0), and d_0 and d_1 are nmbers such that

$$u_n = \cdots = u_{n+d_0} < u_{n+d_0+1} \quad \text{and} \quad u_{N-n-d_1-1} < u_{N-n-d_1} = \cdots = u_{N-n}.$$

```
#define mbs_BSDegElevClosedC1f(indegree,inlastknot,inknots, \
    incoeff,deltadeg,outdegree,outlastknot,outknots,outcoeff) \
  mbs_multiBSDegElevClosedf(1,1,indegree,inlastknot,inknots,0, \
    incoeff,deltadeg,outdegree,outlastknot,outknots,0,outcoeff)
#define mbs_BSDegElevClosedC2f(indegree,inlastknot,inknots, \
    inctlpoints,deltadeg,outdegree,outlastknot,outknots, \
    outctlpoints) \
  mbs_multiBSDegElevClosedf(1,2,indegree,inlastknot,inknots,0, \
    (float*)inctlpoints,deltadeg,outdegree,outlastknot,outknots, \
    0,(float*)outctlpoints)
#define mbs_BSDegElevClosedC3f(indegree,inlastknot,inknots, \
    inctlpoints,deltadeg,outdegree,outlastknot,outknots, \
    outctlpoints) ...
#define mbs_BSDegElevClosedC4f(indegree,inlastknot,inknots, \
    inctlpoints,deltadeg,outdegree,outlastknot,outknots, \
    outctlpoints) ...
```

Four macros calling the procedure `mbs_multiBSDegElevClosedf` in order to perform degree elevation of one closed B-spline curve located in the space of dimension $1, 2, 3, 4$ respectively. For the description of the parameters see the description of that procedure.

Example — degree elevation of a B-spline patch

The degree of a patch may be elevated with respect to the first („u") or the second („v") parameter. The methods of calling the appropriate procedures for both cases, shown in the example below, are based on assumption that all control nets of the patch (the initial and the final ones) are "packed", i.e. the pitch of each array with the control points is equal to the length of representation of one column.

We have the numbers n and m, which specify the degree of the initial representation of the patch, the numbers N and M, which specify the lengths of knot sequences, the arrays `uknots` and `vknots` (of length $N+1$ and $M+1$ respectively) with the knots, and the array `ctlp` with $(N-n)(M-m)d$ floating point numbers,

the coordinates of the control points of the patch. The pitch of the last array is $(M-m)d$.

To raise the degree with respect to the "u" parameter, we can see this patch as a B-spline curve in the space of dimension $(M-m)d$. Then we compute the lengths of the necessary arrays, we allocate the memory and we call the procedure of degree elevation (here the degree increment is 1):

```
ku = mbs_NumKnotIntervalsf ( n, N, uknots );
for ( d0 = 0; uknots[n+d0+1] == uknots[n]; d0++ )
  ;
for ( d1 = 0; uknots[N-n-d1-1] == unkots[N-n]; d1++ )
  ;
ua = pkv_GetScratchMemf ( N+2+ku-d0-d1 );
cpa = pkv_GetScratchMemf ( (N-n+ku-d0-d1)(M-m)d );
mbs_multiBSDegElevf ( 1, (M-m)d, n, N, uknots, 0, ctlp, 1,
                      &na, &Na, ua, 0, cpa, false );
```

The pitches of the arrays cp and cpa are irrelevant (the parameters which specify them are 0), because here only one curve is subject to the degree elevation. The variables na and Na are assigned the degree (equal to $n+1$) and the number of the last knot (equal to $N+ku-d0-d1+1$) of the resulting representation of the patch. The degree with respect to the parameter "v" and the knot sequence related with this parameter are identical as in the initial representation of the patch.

Degree elevation with respect to the parameter "v" is equivalent to the degree elevation of B-spline curves represented by the columns of the control net. The appropriate code, which raises the degree by 1, looks like this:

```
kv = mbs_NumKnotIntervalsf ( m, M, vknots );
for ( d0 = 0; vknots[m+d0+1] == vknots[m]; d0++ )
  ;
for ( d1 = 0; vknots[M-m-d1-1] == vnkots[M-m]; d1++ )
  ;
va = pkv_GetScratchMemf ( M+2+kv-d0-d1 );
cpa = pkv_GetScratchMemf ( (N-n)(M-m+kv-d0-d1)d );
pitch1 = (M-m)d;
pitch2 = (M-m+kv-d0-d1)d;
mbs_multiBSDegElevf ( N-n, d, m, M, vknots, pitch1, ctlp, 1,
                      &ma, &Ma, va, pitch2, cpa, false );
```

If degree elevation by an increment greater than 1 is needed, one can execute the code above a number of times, but it is much faster and more accurate to specify the appropriate parameter deltadeg. It is necessary then to compute correctly the lengths and pitches of the arrays to accomodate the resulting representation of the patch. The sufficient information may be found in the description of the procedure mbs_multiBSDegElevf

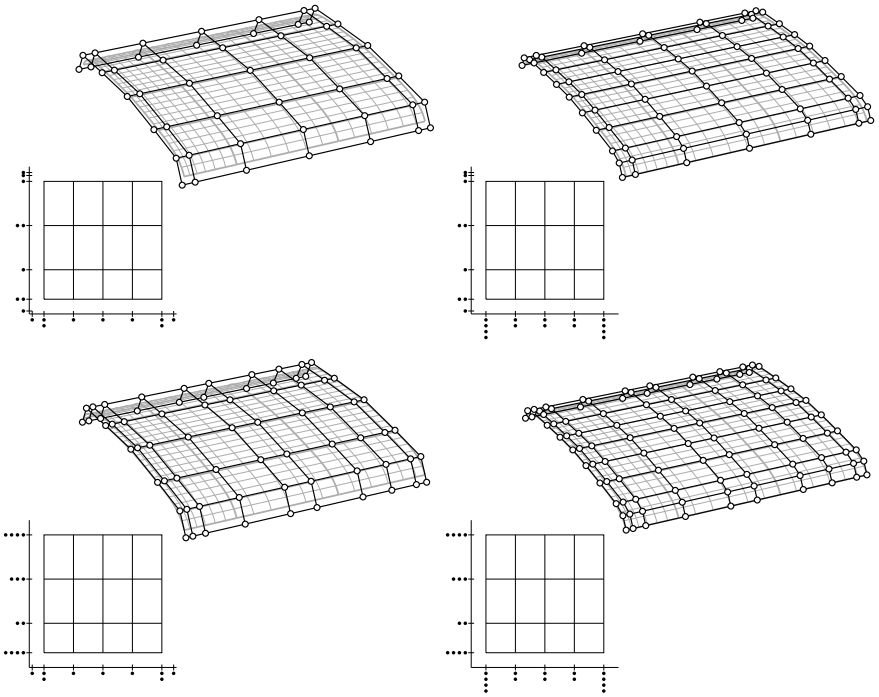


Figure 7.9. Degree elevation of a B-spline patch.

7.11 Degree reduction

Degree reduction of a B-spline curve is a problem of approximation (somewhat like knot removing). The aim is to obtain a B-spline curve $\tilde{\mathbf{s}}$ of degree $\tilde{n} = n - d$ (for $d \in \{1, \ldots, n\}$), which is close to a given curve $\mathbf{s}$ of degree n. A delicate point of this construction is the arbitrary choice of the knot sequence for the resulting curve, because of its influence on the curve shape. The following assumptions seem obvious:

- The resulting curve must have the same domain.
- If the given curve $\mathbf{s}$ was obtained by degree elevation by d of a curve $\tilde{\mathbf{s}}$ of degree n', then the result of degree reduction must be the curve $\tilde{\mathbf{s}}$.

In the constructions implemented in the procedures described in this section the set of knots of the resulting curve is a subset of the set of knots of the given curve. The rule of choosing the multiplicities of the knots is as follows: let a knot u_i of the given curve has multiplicity r. If $r <= d$, then the multiplicity $\tilde{r}$ of this knot in

the result representation is 1. If $d < r <= n+1$, then $\tilde{r} = r - d$, and if $r > n+1$, then $\tilde{r} = n - d + 1$.

For a **non-closed curve** the knot sequence obtained based on the rule above is modified so as to obtain a sequence $\tilde{u}_0, \ldots, \tilde{u}_{\tilde{N}}$ such that $\tilde{u}_{n'} < \tilde{u}_{n'+1}$ and $\tilde{u}_{\tilde{N}-n'-1} < \tilde{u}_{\tilde{N}-n'-1}$. To do this, at both ends of the sequence some knots may be rejected or appended (the first and the last knot may be appended).

The next step is to find an auxiliary knot sequence $\hat{u}_0, \ldots, \hat{u}_{\hat{N}}$, which contains all knots of the resulting sequence, with multiplicities greater by d. By knot insertion (with the Oslo algorithm, the procedure `mbs_multiOsloInsertKnotsf`) and removing knots of multiplicities exceeding $n+1$, (with the procedure `mbs_multiRemoveSuperfluousKnotsf`) the auxiliary representation of the given curve $\mathbf{s}$, based on the auxiliary knot sequence, is obtained:

$$\mathbf{s}(t) = \sum_{i=0}^{N-n-1} \mathbf{d}_i N_i^n(t) = \sum_{i=0}^{\hat{N}-n-1} \hat{\mathbf{d}}_i \hat{N}_i^n(t).$$

Then the matrix A, which describes degree elevation by d of the B-spline curve of degree n' based on the resulting knot sequence, is constructed. The control points $\tilde{\mathbf{d}}_0, \ldots, \tilde{\mathbf{d}}_{\tilde{N}-n'-1}$ of the resulting curve $\tilde{\mathbf{s}}$ are computed by solving the linear least squares problem for the system of equations

$$A\boldsymbol{x} = \mathbf{b},$$

where $\boldsymbol{x} = [\tilde{\mathbf{d}}_0, \ldots, \tilde{\mathbf{d}}_{\tilde{N}-n'-1}]^T$ and $\mathbf{b} = [\hat{\mathbf{d}}_0, \ldots, \hat{\mathbf{d}}_{\hat{N}-n-1}]^T$.

```
boolean mbs_multiBSDegRedf ( int ncurves, int spdimen,
                int indegree, int inlastknot, const float *inknots,
                int inpitch, const float *inctlpoints,
                int deltadeg,
                int *outdegree, int *outlastknot, float *outknots,
                int outpitch, float *outctlpoints );
```

The procedure `mbs_multiBSDegRedf` reduces the degree of non-closed B-spline curves, as described above. The input parameters specify: `ncurves` — the number of curves, `spdimen` — dimension of the space with the curves, `indegree` — degree n, `inlastknot` — the index N of the last knot of the given curves, `inknots` — the knot sequence of the given curves (in an array of length $N+1$), `inpitch` — pitch of the array with the given control points, `deltadeg` — the number d, by which the degree is to be reduced.

Output parameters: `*outdegree` — the variable, to which the degree n' of the result curves will be assigned, `*outlastknot` — the variable, to which the index $\tilde{N}$ of the last knot of the result knot sequence will be assigned, `outknots` — an array for storing these knots $\tilde{u}_0, \ldots, \tilde{u}_{\tilde{N}}$, `outpitch` — pitch of the array `outctlpoints`, in which the control points of the resulting curves will be stored.

Caution: Currently there is no procedure to compute the length of the resulting knot sequence, which might be called before the allocation of the arrays for the result knots and control points. Before such a procedure is implemented, one has to guess the sufficient sizes for these arrays and guess the sufficiently large pitch for the array `outctlpoints`.

The return value is `true` if the construction succeeded and `false` otherwise. However, in case of error the procedure `pkv_SignalError` is called, and its default behaviour causes the program termination.

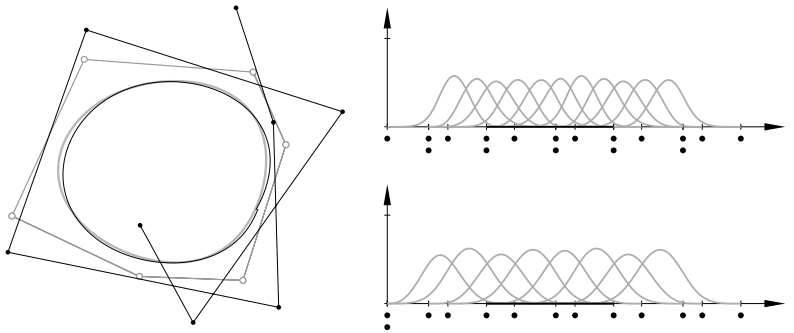


Figure 7.10. Degree reduction of a B-spline curve from 5 to 4

```
#define mbs_BSDegRedC1f(indegree,inlastknot,inknots,incoeff, \
    deltadeg,outdegree,outlastknot,outknots,outcoeff) \
  mbs_multiBSDegRedf(1,1,indegree,inlastknot,inknots,0,incoeff, \
    deltadeg,outdegree,outlastknot,outknots,0,outcoeff)
#define mbs_BSDegRedC2f(indegree,inlastknot,inknots,incpoints, \
    deltadeg,outdegree,outlastknot,outknots,outcpoints) \
  mbs_multiBSDegRedf(1,2,indegree,inlastknot,inknots,0, \
    (float*)incpoints, \
    deltadeg,outdegree,outlastknot,outknots,0,(float*)outcpoints)
#define mbs_BSDegRedC3f(indegree,inlastknot,inknots,incpoints, \
    deltadeg,outdegree,outlastknot,outknots,outcpoints) ...
#define mbs_BSDegRedC4f(indegree,inlastknot,inknots,incpoints, \
    deltadeg,outdegree,outlastknot,outknots,outcpoints) ...
```

The macros above call the procedure `mbs_multiBSDegRedf` in order to reduce the degree of one curve in the space of dimension $1, \ldots, 4$.

```
boolean mbs_multiBSDegRedClosedf ( int ncurves, int spdimen,
                int indegree, int inlastknot, const float *inknots,
                int inpitch, const float *inctlpoints,
                int deltadeg,
                int *outdegree, int *outlastknot, float *outknots,
                int outpitch, float *outctlpoints );
```

The procedure `mbs_multiBSDegRedClosedf` reduces degree of closed B-spline curves. Its parameters have identical descriptions as the parameters of the procedure `mbs_multiBSDegRedf`.

```
#define mbs_BSDegRedClosedC1f(indegree,inlastknot,inknots, \
    incoeff,deltadeg,outdegree,outlastknot,outknots,outcoeff) \
  mbs_multiBSDegRedClosedf(1,1,indegree,inlastknot,inknots,0, \
    incoeff,deltadeg,outdegree,outlastknot,outknots,0,outcoeff)
#define mbs_BSDegRedClosedC2f(indegree,inlastknot,inknots, \
    incpoints,deltadeg,outdegree,outlastknot,outknots,outcpoints) \
  mbs_multiBSDegRedClosedf(1,2,indegree,inlastknot,inknots,0, \
    (float*)incpoints, \
    deltadeg,outdegree,outlastknot,outknots,0,(float*)outcpoints)
#define mbs_BSDegRedClosedC3f(indegree,inlastknot,inknots, \
  incpoints,deltadeg,outdegree,outlastknot,outknots,outcpoints) ...
#define mbs_BSDegRedClosedC4f(indegree,inlastknot,inknots, \
  incpoints,deltadeg,outdegree,outlastknot,outknots,outcpoints) ...
```

The macros above call `mbs_multiBSDegRedClosedf` in order to reduce degree of one closed curve located in the space of dimension $1, \ldots, 4$.

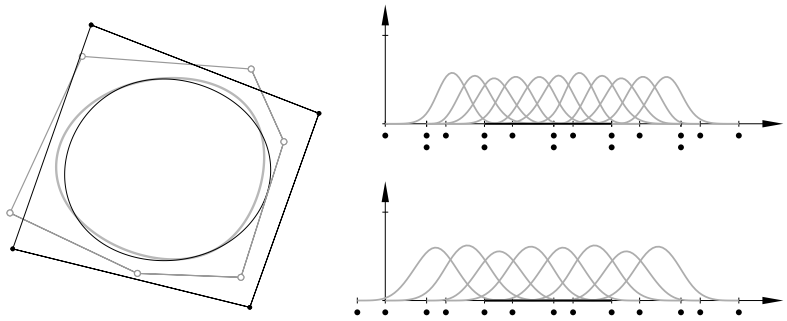


Figure 7.11. Degree reduction of a closed B-spline curve from 5 to 4

7.12 Algebraic operations on spline functions and curves

The purpose of the procedures described in this section is computing a B-spline representation of the sum of (vector) B-spline curves and the product of (scalar) functions and (vector) curves. These operations are necessary in various applications, e.g. in constructing surfaces, which prescribed order of geometric continuity.

7.12.1 Addition of splines

Adding, i.e. computing the representation of a sum of B-spline curves must be preceded by finding the degree of this representation. The degree of the sum is the greatest of degrees of the terms. The knot sequence is determined by the knot sequence of the terms, which must determine the same domain. To add two curves (which must reside in the same space), it is necessary to find the representation of those curves, with the degree and the knot sequence, which will be used to represent the result. This auxiliary task is the most complicated and costly part of the procedure of adding the spline curves. The last and the simplest part of the computation is summing the coefficients (the control points) of the terms.

```
boolean mbs_FindBSCommonKnotSequencef ( int *degree, int *lastknot,
                                float **knots, int nsequences, ... );
```

The procedure `mbs_FindBSCommonKnotSequencef` obtains `k` knot sequences, used to represent B-spline curves of given degrees. The task of this procedure is to find a minimal degree and a knot sequence suitable to represent the sum of the curves. This degree `n` is the greatest of the given degrees or the initial value of the variable `*degree`, if it is greater (it is thus possible to enforce the greater degree of the common representation of the curves). The knot sequence found by this procedure has the following properties:

- The boundary knots have multiplicities $n + 1$ (thus the external knots, including the extremal ones, coincide with the boundary knots).
- The sequence contains all internal knots from the given sequences.
- The multiplicities of the internal knots are chosen so that after the degree elevation up to `n` it is possible to represent each curve with this knot sequence.

The parameters `degree`, `lastknot` and `knots` are used to output the result (due to the C language syntax they appear at the beginning of the parameter list, which deviates from the convention assumed in the BSTools package). The variables pointed by these parameters obtain values, which are the degree, the number of the last knot and a pointer to the array with knots respectively.

Caution: The procedure allocates this array on the scratch memory stack, and the calling subprogram is responsible for its deallocation, (using `pkv_FreeScratchMem` or `pkv_SetScratchMemTop`).

The parameter nsequences specifies the number k of given knot sequences (there must be $k \geq 1$). At the calling point it must be followed by 3k parameters. The consecutive triples of parameters describe the knot sequences. The first element of a triple is the degree n_i (of type int), the second element is the index N_i of the last knot (of type int), and the third element is a pointer to the array with the knots ($N_i + 1$ floating point numbers — this parameter is of type float*).

All given knot sequences must have the same knot with the number n_i; the same concerns the knot with the number $N_i - n_i$.

The value returned is true if the computation was successful, and false in case of failure. The possible reasons of failure are invalid data or insufficient space on the stack of scratch memory.

```
boolean mbs_multiAdjustBSCRepf ( int ncurves, int spdimen,
            int indegree, int inlastknot, const float *inknots,
            int inpitch, const float *inctlpoints,
            int outdegree, int outlastknot, const float *outknots,
            int outpitch, float *outctlpoints );
```

The procedure mbs_multiAdjustBSCRepf „adjusts" the representation of B-spline curves, i.e. it finds a representation of a given degree, based on the given knot sequence. If necessary, degree elevation is done, followed by inserting knots (with the Oslo algorithm). To add k B-spline curves with different representations (but with the same domain), one has to find the degree and knot sequence suitable to represent them all (using mbs_FindBSCommonKnotSequencef), and then find the proper representation of each term, by calling mbs_multiAdjustBSCRepf.

The parameters: ncurves — number of curves, spdimen — space dimension, indegree, inlastknot, inknots — degree, index of the last knot and pointer to the array with knots of the given representation, inpitch — pitch of the array inctlpoints, with the control points of the curve.

The parameters outdegree, outlastknot and outknots describe the degree and knot sequence of the representation to be found. The control points of this representation will be stored in the array outctlpoints, whose pitch is outpitch.

The value returned is true in case of success and false in case of failure (caused by invalid data or insufficient space on the scratch memory stack).


```
#define mbs_AdjustBSCRepC1f(indegree,inlastknot,inknots, \
    inctlpoints,outdegree,outlastknot,outknots,outctlpoints) \
  mbs_multiAdjustBSCRepf (1,1,indegree,inlastknot,inknots,0, \
    inctlpoints,outdegree,outlastknot,outknots,0,outctlpoints)
#define mbs_AdjustBSCRepC2f(indegree,inlastknot,inknots, \
    inctlpoints,outdegree,outlastknot,outknots,outctlpoints) \
  mbs_multiAdjustBSCRepf (1,2,indegree,inlastknot,inknots,0, \
    (float*)inctlpoints,outdegree,outlastknot,outknots,0, \
    (float*)outctlpoints)
#define mbs_AdjustBSCRepC3f(indegree,inlastknot,inknots, \
    inctlpoints,outdegree,outlastknot,outknots,outctlpoints) ...
#define mbs_AdjustBSCRepC4f(indegree,inlastknot,inknots, \
    inctlpoints,outdegree,outlastknot,outknots,outctlpoints) ...
```

```
void mbs_multiAddBSCurvesf ( int ncurves, int spdimen,
                    int degree1, int lastknot1, const float *knots1,
                    int pitch1, const float *ctlpoints1,
                    int degree2, int lastknot2, const float *knots2,
                    int pitch2, const float *ctlpoints2,
                    int *sumdeg, int *sumlastknot, float *sumknots,
                    int sumpitch, float *sumctlpoints );
```

The procedure mbs_multiAddBSCurvesf computes the sums of ncurves pairs of B-spline curves in the space of dimension spdimen.

The first curve of each pair is described with the parameters degree1 (degree), lastknot1 (the index of the last knot), knots1 (array of knots) i ctlpoints1 (array with control points, whose pitch is pitch1).

The second curve of each pair is similarly described by the parameters degree2, lastknot2, knots2, pitch2 and ctlpoints2.

The output parameters are *sumdeg (it is assigned the degree of the sum) *sumlastknot (the index of the last knot of the sum representation) sumknots (array in which the procedure stores the knots of the sum representation), sumctlpoints (array in which the procedure stores the control points of the sums; its pitch is sumpitch).

```
#define mbs_AddBSCurvesC1f(degree1,lastknot1,knots1,ctlpoints1, \
    degree2,lastknot2,knots2,ctlpoints2, \
    sumdeg,sumlastknot,sumknots,sumctlpoints) \
  mbs_multiAddBSCurvesf (1,1,degree1,lastknot1,knots1,0, \
    ctlpoints1,degree2,lastknot2,knots2,0,ctlpoints2, \
    sumdeg,sumlastknot,sumknots,0,sumctlpoints)
#define mbs_AddBSCurvesC2f(degree1,lastknot1,knots1,ctlpoints1, \
    degree2,lastknot2,knots2,ctlpoints2, \
    sumdeg,sumlastknot,sumknots,sumctlpoints) \
  mbs_multiAddBSCurvesf (1,2,degree1,lastknot1,knots1,0, \
    (float*)ctlpoints1, \
    degree2,lastknot2,knots2,0,(float*)ctlpoints2, \
    sumdeg,sumlastknot,sumknots,0,(float*)sumctlpoints)
#define mbs_AddBSCurvesC3f(degree1,lastknot1,knots1,ctlpoints1, \
    degree2,lastknot2,knots2,ctlpoints2, \
    sumdeg,sumlastknot,sumknots,sumctlpoints) ...
#define mbs_AddBSCurvesC4f(degree1,lastknot1,knots1,ctlpoints1, \
    degree2,lastknot2,knots2,ctlpoints2, \
    sumdeg,sumlastknot,sumknots,sumctlpoints) ...
```

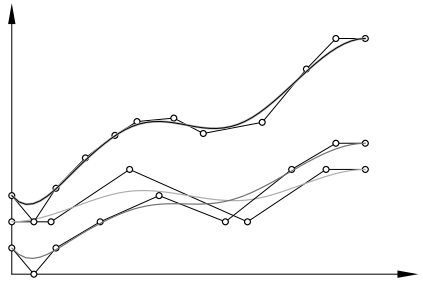


Figure 7.12. Spline functions of degrees 3 and 4 and their sum

```
void mbs_multiSubtractBSCurvesf ( int ncurves, int spdimen,
                    int degree1, int lastknot1, const float *knots1,
                    int pitch1, const float *ctlpoints1,
                    int degree2, int lastknot2, const float *knots2,
                    int pitch2, const float *ctlpoints2,
                    int *sumdeg, int *sumlastknot, float *sumknots,
                    int sumpitch, float *sumctlpoints );
```



```
#define mbs_SubtractBSCurvesC1f(degree1,lastknot1,knots1, \
    ctlpoints1,degree2,lastknot2,knots2,ctlpoints2, \
    sumdeg,sumlastknot,sumknots,sumctlpoints) \
  mbs_multiSubtractBSCurvesf (1,1,degree1,lastknot1,knots1,0, \
    ctlpoints1,degree2,lastknot2,knots2,0,ctlpoints2, \
    sumdeg,sumlastknot,sumknots,0,sumctlpoints)
#define mbs_SubtractBSCurvesC2f(degree1,lastknot1,knots1, \
    ctlpoints1,degree2,lastknot2,knots2,ctlpoints2, \
    sumdeg,sumlastknot,sumknots,sumctlpoints) \
  mbs_multiSubtractBSCurvesf (1,2,degree1,lastknot1,knots1,0, \
    (float*)ctlpoints1,degree2,lastknot2,knots2,0, \
    (float*)ctlpoints2,sumdeg,sumlastknot,sumknots,0, \
    (float*)sumctlpoints)
#define mbs_SubtractBSCurvesC3f(degree1,lastknot1,knots1, \
    ctlpoints1,degree2,lastknot2,knots2,ctlpoints2, \
    sumdeg,sumlastknot,sumknots,sumctlpoints) ...
#define mbs_SubtractBSCurvesC4f(degree1,lastknot1,knots1, \
    ctlpoints1,degree2,lastknot2,knots2,ctlpoints2, \
    sumdeg,sumlastknot,sumknots,sumctlpoints) ...
```

7.12.2 Transformation between Bernstein and scaled Bernstein bases

This section contains the description of auxiliary procedures used by the procedures of multiplication of spline functions and curves.

To multiply the polynomials given by the coefficients in the Bernstein bases, it is convenient to transform the data to the *scaled bases*. The scaled basis of degree n consists of the polynomials

$$b_i^n(t) \stackrel{\text{def}}{=} \frac{1}{\binom{n}{i}} B_i^n(t) = t^i(1-t)^{n-i}. \tag{7.23}$$

The coefficients in this basis are obtained by multiplying the coefficients of the polynomial in the Bernstein basis by $\binom{n}{i}$.

The result of multiplication of the polynomials represented in the scaled bases of degrees n and m is the sequence of coefficients in the scaled basis of degree $n+m$. Having it, we can transform it to the Bernstein basis of degree $n+m$, with the appropriate divisions.

```
void mbs_multiBezScalef ( int degree, int narcs,
                          int ncurves, int spdimen,
                          int pitch, float *ctlpoints );
```

The procedure `mbs_multiBezScalef` obtains an array of Bézier curves of degree `degree` in the space of dimension `spdimen` and it computes the coefficients of the curves in the scaled basis. An assumption is made that these curves are consecutive arcs of B-spline curves, which have been obtained by the appropriate knot insertions (e.g. with the `mbs_multiMaxKnotInsf` procedeure).

The parameters: `degree` — degree of the curves, `narcs` — the number of Bézier arcs making each B-spline curve, `ncurves` — the number of B-spline curves, `spdimen` — the dimension d of the space, in which the curves reside.

The parameter `pitch` specifies the pitch of the array `ctlpoints`, which before calling the procedure contains the control points of the curves (i.e. their coefficients in the Bernstein bases of degree $n =$ `degree`), and on return it contains the coefficients in the scaled bases. The parameter `pitch` specifies the distance between the beginnings of the first control points of consecutive *B-spline curves*. The representations of consecutive Bézier curves always occupy $(n+1)d$ places, without unused areas between them. The pitch of this array cannot be less than $(n+1)d$`*narcs`.

```
void mbs_multiBezUnscalef ( int degree, int narcs,
                            int ncurves, int spdimen,
                            int pitch, float *ctlpoints );
```

The procedure `mbs_multiBezUnscalef` obtains the array with the representations of polynomial curves in the scaled bases and it does the transformation to the Bernstein bases, i.e. to the Bézier representation. The parameters of this procedure (except for the description of the initial and final contents of the array `ctlpoints`) are identical as the parameters of the procedure `mbs_multiBezScalef`.

7.12.3 Multiplication of spline functions and curves

The procedures described in this section multiply polynomial and spline curves (i.e. vector functions) by scalar polynomials and splines. The data for the procedures consist of representations of one or more scalar functions (polynomial or splines) s_i and one or more vector functions (polynomials or splines) $\boldsymbol{v}_i$. Both these numbers have to be equal or one of them must be 1. The procedures compute the Bézier or B-spline representations of the vector functions

$$\boldsymbol{w}_i(t) = s_i(t)\boldsymbol{v}_i(t),$$

and if there is only one scalar sunction s_0 and more vector functions, then each vector function will be multiplied by s_0, and similarly if there are many scalar functions s_i and one vector function $\boldsymbol{v}_0$ then the procedures compute the products of the functions s_i with $\boldsymbol{v}_0$.

The procedures described here may be applied in various advanced constructions. The simplest is the degree elevation of a curve, by multiplying it by the constanf scalar function $s_0(t) = 1$ (the degree of representation of s_0 is the difference between the degrees of the initial and final curve representations). However, in this case it is better to use the specific procedure of degree elevation (e.g. `mbs_multiBSDegElevf`), which makes this computation in a less heavy-handed manner.

```
int mbs_BSProdRepSizef ( int degree1, int lastknot1,
                         const float *knots1,
                         int degree2, int lastknot2,
                         const float *knots2 );
```

The procedure `mbs_BSProdRepSizef` gets *two* knot sequences, `knots1` of length `lastknot1+1` and `knots2` of length `lastknot2+1`. The first sequence is the part of representation of spline functions of degree `degree1`, and the second — of the functions of degree `degree2`. The sequences should determine the same domain of the spline curves. The value of the procedure is the index of the last element of the shortest knot sequence sufficient to represent the product of any spline functions possible to represent with the two given knot sequences.

```
void mbs_SetBSProdKnotsf ( int degree1, int lastknot1,
                           const float *knots1,
                           int degree2, int lastknot2,
                           const float *knots2,
                           int *degree, int *lastknot,
                           float *knots );
```

The procedura `mbs_SetBSProdKnotsf` gets two knot sequences and it generates another sequence, which is sufficient to represent the product of splines defined with the given two knot sequences.

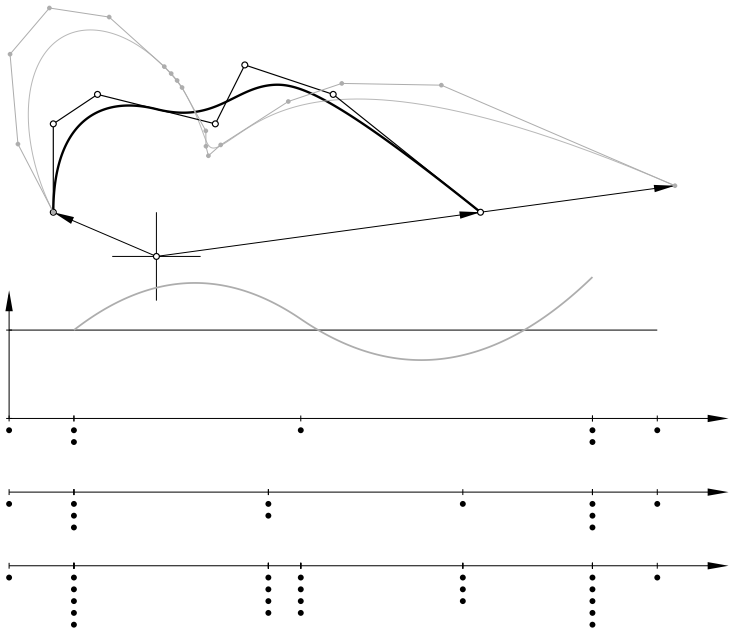


Figure 7.13. Multiplication of a planar vector B-spline curve by a spline function.

```
void mbs_multiMultBezCf ( int nscf, int degscf, int scfpitch,
                          const float *scfcoeff,
                          int spdimen,
                          int nvecf, int degvecf, int vecfpitch,
                          const float *vecfcp,
                          int *degprod, int prodpitch,
                          float *prodcp );
```

The procedure `mbs_multiMultBezCf` multiplies the polynomials represented in the Bernstein polynomial basis of degree `degscf` and polynomial vector functions (Bézier curves) of degree `degvecf`. The parameter `spdimen` specifies the dimension of the space, in which the curves reside. The number of the scalar functions is determined by the parameter `nscf`, and the number of the vector curves is specified by the parameter `nvecf`. The number of products computed by the procedure is the greater number of the two, see remarks at the beginning of this section.

The array `scfcoeff` contains the coefficients of the polynomials in the Bernstein basis; the coefficients of each polynomial occupy the consecutive places in the array, and its pitch (difference between the indexes of the first coefficients of two consecutive polynomials) is specified by the parameter `scfpitch`. Similarly, the parameter `vecfpitch` specifies the pitch of the array `vecfcp` with the vector coefficients of the

curves (each coefficient consists of `spdimen` numbers).

The products are represented in the Bernstein basis of degree equal to the sum of degrees of the arguments (i.e. `degscf` + `degvecf`); this degree is returned as the value of the parameter `degprod`. The representations of consecutive products consist of the sequences of `spdimen` $*$ (stopie"n + 1) numbers, which are stored by the procedure to the array `prodcp`, whose pitch is `prodpitch`.

```
void mbs_multiMultBSCf ( int nscf, int degscf,
                         int scflastknot, const float *scfknots,
                         int scfpitch, const float *scfcoeff,
                         int spdimen,
                         int nvecf, int degvecf,
                         int vecflastknot, const float *vecfknots,
                         int vecfpitch, const float *vecfcp,
                         int *degprod, int *prodlastknot,
                         float *prodknots,
                         int prodpitch, float *prodcp );
```

The procedure `mbs_multiMultBSCf` computes the representations of the products of `nscf` scalar spline functions s_i and `nvecf` vector spline functions $\boldsymbol{v}_i$. The numbers of the scalar functions and the vector functions may be different (one of them must be then 1), see the remarks at the beginning of this section.

The scalar functions are represented with the parameters `degscf` (representation degree), `scflastknot` and `scfknots` (index of the last knot and the array with these knots), `scfcoeff` and `scfpitch` (array with the coefficients in the B-spline basis and the pitch of this array).

The vector functions in the space of dimension `spdimen` are similarly represented by the parameters `degvecf`, `vecflastknot`, `vecfknots`, `vecfpitch` and `vecfcp`.

The result is stored in the arrays `prodknots` (knots) and `prodcp` (vector coefficients in the B-spline basis of degree equal to the sum of degrees of the factors; this degree is returned using the parameter `degprod`). The pitch of the latter array is specified by the parameter `prodpitch`. The initial value of the parameter `*prodlastknot` specifies the amount of space in the array `prodknots` (it has to be greater by 1 than the value of this parameter). It is necessary to compute this length and to allocate the array *before* calling the procedure `mbs_multiMultBSCf`. It is best to do it using the procedure `mbs_BSProdRepSizef`, which scans the knot sequences of the arguments of the multiplication, passed as its parameters.

7.12.4 Computing normal vector patches

```
void mbs_BezP3NormalDeg ( int degreeu, int degreev,
                          int *ndegu, int *ndegv );
char mbs_BezP3Normalf ( int degreeu, int degreev,
                        const point3f *ctlpoints,
                        int *ndegu, int *ndegv, vector3f *ncp );
```

The procedure `mbs_BezP3Normalf` computes the control points of the patch $\mathbf{n} = \mathbf{p}_u \wedge \mathbf{p}_v$, which describes the normal vector of a given polynomial Bézier patch $\mathbf{p}$ of degree (n, m) in $\mathbb{R}^3$. The parameters `degreeu` $= n$ and `degreev` $= m$ specify the degree of the patch $\mathbf{p}$. Its control points are given in the array `ctlpoints`, which contains the subsequent columns without unused areas between them.

The degree of the patch $\mathbf{n}$ is `*ndegu`$= 2n - 1$ with respect to u and `*ndegv`$= 2m - 1$ with respect to v, and its control points are stored by the procedure in the array `ncp` (without unused areas between the columns).

The value returned by the procedure `mbs_BezP3Normalf` is 0 in case of failure (invalid parameters or not enough scratch memory), or 1 if the computation has been successful.

The procedure `mbs_BezP3NormalDeg` computes the degree of the normal vector patch. It may be used to allocate a sufficient memory block for storing the control points of this patch.

```
void mbs_BezP3RNormalDeg ( int degreeu, int degreev,
                           int *ndegu, int *ndegv );
char mbs_BezP3RNormalf ( int degreeu, int degreev,
                         const point4f *ctlpoints,
                         int *ndegu, int *ndegv, vector3f *ncp );
```

The procedure `mbs_BezP3RNormalf` computes the control points of the polynomial Bézier patch $\mathbf{n}$, which describes the normal vector of a given rational Bézier patch $\mathbf{p}$ of degree (n, m) in $\mathbb{R}^3$. These control points are obtained by rejecting the weight coordinate of the control points of the patch $\mathbf{N} = \mathbf{P} \wedge \mathbf{P}_u \wedge \mathbf{P}_v$ in $\mathbb{R}^4$. The parameters `degreeu` $= n$ and `degreev` $= m$ specify the degree of the given patch $\mathbf{p}$. The control points of its homogeneous representation are given in the array `ctlpoints`, which contains the subsequent columns without unused areas between them.

The degree of the normal vector patch is `*ndegu`$= 3n - 2$ with respect to u and `*ndegv`$= 3m - 2$ with respect to v, and its control points are stored in the array `ncp` (without unused areas between the columns).

The procedure `mbs_BezP3RNormalf` returns 0 in case of failure (invalid parameters or not enough scratch memory), or 1, in case of success.

The procedure `mbs_BezP3RNormalDeg` computes the degree of the patch $\mathbf{n}$. It may be used to allocate a sufficient memory block for the control points of $\mathbf{n}$.

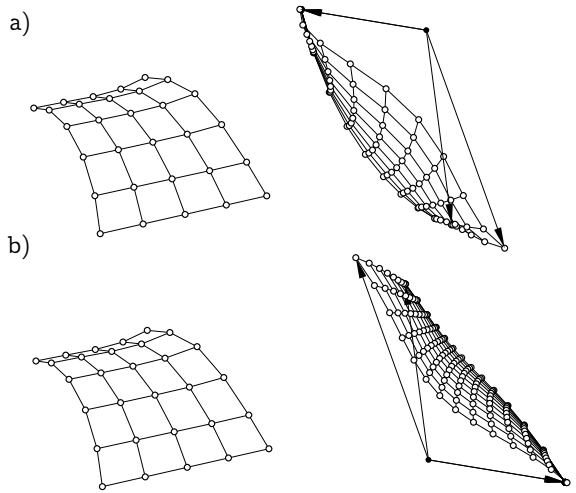


Figure 7.14. Control nets of Béziera patches and their normal vector patches: a) polynomial patch, b) rational patch.

7.13 B-spline end knots change

```
void mbs_multiBSChangeLeftKnotsf ( int ncurves, int spdimen,
                                   int degree, float *knots,
                                   int pitch, float *ctlpoints,
                                   float *newknots );
```

The procedure `mbs_multiBSChangeLeftKnotsf` changes the representation of B-spline curves of degree n, by replacing the initial $n+1$ knots by the knots given in the array `newknots`. It may cause extending or trimming the domain and in this case the first polynomial arc of each curve is extended or shortened.

The new knots must be given in the nondecreasing order, and there must be $u_n < u_{n+1}$

```
void mbs_multiBSChangeRightKnotsf ( int ncurves, int spdimen,
                                    int degree,
                                    int lastknot, float *knots,
                                    int pitch, float *ctlpoints,
                                    float *newknots );
```

The procedure `mbs_multiBSChangeRightKnotsf` changes the representation of B-spline curves of degree n, by replacing its last $n+1$ knots (i.e. the knots $u_{N-n}, \ldots, u_N$) by the knots given in the array `newknots`. It may cause extending or trimming the domain and in this case the last polynomial arc of each curve is elongated or shortened.

The new knots must be given in the nondecreasing order, and there must be $u_{N-n} > u_{N-n-1}$.

```
#define mbs_BSChangeLeftKnotsC1f(degree,knots,coeff,newknots) \
  mbs_multiBSChangeLeftKnotsf(1,1,degree,knots,0,coeff,newknots)
#define mbs_BSChangeLeftKnotsC2f(degree,knots,ctlpoints,newknots) \
  mbs_multiBSChangeLeftKnotsf(1,2,degree,knots,0, \
                              (float*)ctlpoints,newknots)
#define mbs_BSChangeLeftKnotsC3f(degree,knots,ctlpoints,newknots) \
  ...
#define mbs_BSChangeLeftKnotsC4f(degree,knots,ctlpoints,newknots) \
  ...
#define mbs_BSChangeRightKnotsC1f(degree,lastknot,knots,coeff, \
                                  newknots) ...
#define mbs_BSChangeRightKnotsC2f(degree,lastknot,knots, \
                                  ctlpoints,newknots) ...
#define mbs_BSChangeRightKnotsC3f(degree,lastknot,knots, \
                                  ctlpoints,newknots) ...
#define mbs_BSChangeRightKnotsC4f(degree,lastknot,knots, \
                                  ctlpoints,newknots)
```


The macros shown above call the procedure `mbs_multiBSChangeLeftKnotsf` and `mbs_multiBSChangeRightKnotsf` in order to change the representation of one B-spline curve in the space of dimension $1, \ldots, 4$.

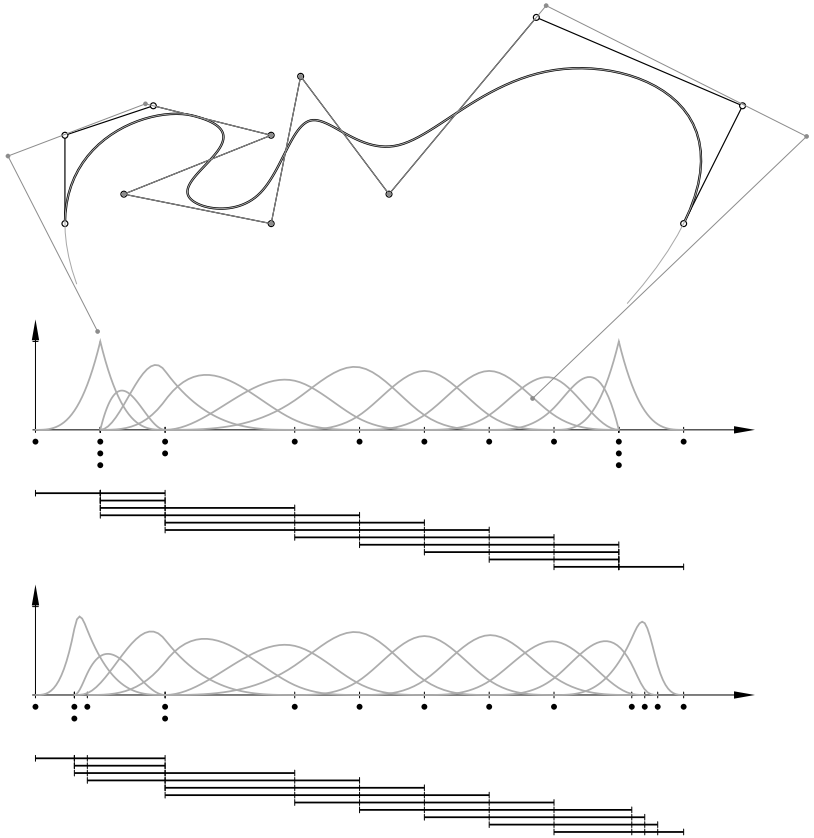


Figure 7.15. A B-spline curve before and after the end representation change.

7.14 Constructing curves of interpolation

The construction of a curve of interpolation is sometimes the main problem, and sometimes it is a part of a bigger problem, like the construction of lofted surfaces or filleting surfaces.

7.14.1 Cubic spline curves of interpolation

This section is devoted to the procedure of computing of a B-spline representation of cubic spline curves of interpolation. The interpolation knots (given as input data) will be the knots of the curves, and the first and last interpolation knots will be the curve (boundary) knots of multiplicity 3. In addition, there will be two extremal knots, necessary in the B-spline representation.

Apart from the knots and interpolation conditions it is necessary to specify *boundary conditions*. The conditions which may be processed by the current version of the procedure are described later.

```
void mbs_multiBSCubicInterpf ( int lastinterpknot,
                               float *interpknots,
                               int ncurves, int spdimen,
                               int xpitch, const float *x,
                               int ypitch,
                               char bcl, const float *ybcl,
                               char bcr, const float *ybcr,
                               int *lastbsknot,
                               float *bsknots,
                               int bspitch,
                               float *ctlpoints );
```

The procedure `mbs_multiBSCubicInterpf` constructs cubic B-spline curves of interpolation of class C^2.

The parameters: `lastinterpknot` specifies the index of the last interpolation knot, which will be denoted by N. The interpolation knots $u_0, \ldots, u_N$, which have to form an increasing sequence, are to be specified in the array `interpknots`.

The parameters `ncurves` and `spdimen` specify the number of curves and the space dimension. The array `x` contains the coordinates of points to be interpolated; for each curve it is necessary to supply `spdimen`(`lastinterpknot`$+1$) floating point numbers. The pitch of this array (i.e. the distance between the beginnings of data for consecutive curves) is specified by the parameter `xpitch`.

The parameter `ypitch` specifies the pitch of the arrays `ybcl` and `ybcr`, which contain the data describing the boundary conditions.

The parameters `bcl` and `bcr` are used to select the boundary conditions at the left and right end of the curves respectively; all the curves are constructed with the boundary conditions of the same kind, but at each end the boundary condition

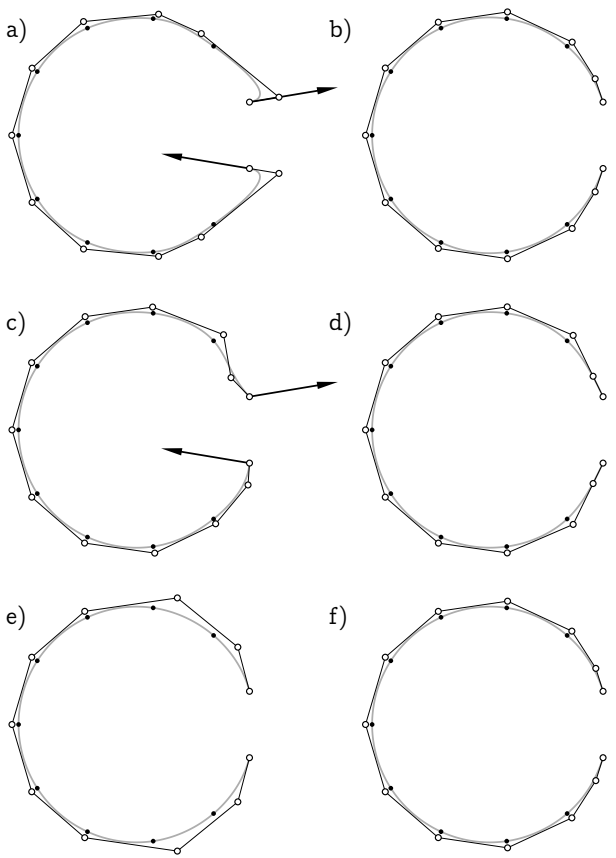


Figure 7.16. Cubic B-spline curves of interpolation.
The knots are the numbers $0, 1, \ldots, 10$. The boundary conditions are:
a) given derivatives at end points, b) Bessel end conditions,
c) given second order derivatives, d) natural spline,
e) not-a-knot condition, f) third order derivatives at end points equal to 0.

may be different. The valid values of the two parameters are defined (as macros) in the file `multibs.h`, and their current list is as follows:

`BS3_BC_FIRST_DER` — the boundary condition is given by specifying the derivative vector of each curve at the left or right interpolation knot (i.e. at u_0 or u_N). The coordinates of those vectors for all the curves must be given in the array `ybcl` (for the knot u_0) or `ybcr` (for the knot u_N). Thus the arrays `ybcl` and `ybcr` for each curve contain `spdimen` floating point numbers, being the coordinates of those vectors.

BS3_BC_FIRST_DER0 — the boundary condition is as above, with zero derivative vector at the appropriate interpolation knot. The parameter ybcl or ybcr is then ignored, so its value may be NULL.

BS3_BC_SECOND_DER — the boundary condition is given by specifying the second order derivative vector at the knot u_0 or u_N. The coordinates of this vector (or vectors, if there is more than one curve to construct) are given in the array ybcl or ybcr respectively.

BS3_BC_SECOND_DER0 — The boundary condition is as above, with the zero derivative vector at the appropriate knot. The parameter ybcl or ybcr is ignored and its value may be NULL.

A curve satisfying such a condition at both ends is called a **natural spline curve**.

BS3_BC_THIRD_DER — the boundary condition is given by specifying the third order derivative vector of the curves. Their coordinates are given in the array ybcl or ybcr.

BS3_BC_THIRD_DER0 — the boundary condition is given by requiring that the third order derivative at the end be the zero vector. As the third order derivative of a cubic polynomial arc is constant, this boundary condition means that the first or the last polynomial arc of the curve is a piece of a parabola. The parameter ybcl or ybcr for this boundary condition is ignored, and its value may be NULL.

BS3_BC_BESSEL — selects the so called Bessel boundary condition. The derivative of the curve at the first or the last interpolation knot is the derivative of the quadratic curve of interpolation for the first three or the last three knots and points.

The parameter ybcl or ybcr in case of the Bessel end condition is ignored, and its value may be NULL.

BS3_BC_NOT_A_KNOT — the not-a-knot boundary condition; the interpolation knot u_1 or u_{N-1} is not a knot of the spline curve, i.e. the polynomial arcs of the curve meet at that knot with the C^∞ continuity. The parameter ybcl or ybcr is ignored, and its value may be NULL.

The representation of the curves of interpolation constructed with this procedure is given by the following parameters: *lastbsknot — the number of the last knot of the spline curve, bsknots — array with the knots (these are the interpolation knots, but the knots u_0 and u_N in this array are of multiplicity 3, and there are two extremal knots in addition, whose presence is required by the representation). The input parameter bspitch specifies the pitch of the array ctlpoints, in which the control points of conescutive curves of interpolation are stored.

7.14.2 Hermite curves of interpolation

The procedures described in this section implement a quite particular construction: they find Bézier and B-spline curves of degree n, which satisfy the Hermite interpolation conditions imposed at two knots, 0 and 1 or u_n and u_{N-n} respectively. There is an application, in which Ineeded such procedures, and the algorithm for this case is faster than the general algorithm of solving the Hermite interpolation problem for a B-spline curve.

```
void mbs_multiInterp2knHermiteBezf ( int ncurves, int spdimen,
                          int degree,
                          int nlbc, int lbcpitch, const float *lbc,
                          int nrbc, int rbcpitch, const float *rbc,
                          int pitch, float *ctlpoints );
```

The procedure mbs_multiInterp2knHermiteBezf constructs ncurves Bézier curves of degree n (the degree is specified by the parameter degree) in the space of dimension d (specified by the parameter spdimen).

The number of interpolation conditions for each curve at the knot 0 is equal to nlbc, and at the knot 1 is nrbc. Both parameters must be nonnegative and their sum must be n + 1 (this ensures the uniqueness of solution of the interpolation problem).

The interpolation conditions are given in the arrays lbc (for the knot 0) and rbc (for the knto 1). The initial d numbers in each array specify the point of the first curve, the next d numbers are coordinates of the derivative vector, then the second order derivative etc. The data, which describe the interpolation conditions for the second curve are stored starting at the position lbcpitch and rbcpitch respectively.

The control points of the curves (i.e. the construction result) are stored in the array ctlpoints, whose pitch (the distance between the beginnings of data which describe the consecutive curves) is the value of the parameter pitch.

```
void mbs_multiInterp2knHermiteBSf ( int ncurves, int spdimen,
                          int degree,
                          int lastknot, const float *knots,
                          int nlbc, int lbcpitch, const float *lbc,
                          int nrbc, int rbcpitch, const float *rbc,
                          int pitch, float *ctlpoints );
```

The procedure mbs_multiInterp2knHermiteBSf constructs ncurves B-spline curves of degree n (the degree is the value of the parameter degree) in the space of dimension d (specified by the parameter spdimen). The curve is defined with the knot sequence of length N + 1, goven in the array knots (the number N is the value of the parameter lastknot).

The number of the interpolation conditions for each curve at the knot u_n is the value of the parameter `nlbc`, and at the knot u_{N-n} is specified by `nrbc`, and none of the parameters may have the value greater than n. Their sum must be $N-n$ to ensure that the interpolation problem has a unique solution.

The knots in the array `knots` must satisfy the conditions $u_1 = \cdots = u_n < u_{n+1}$ and $u_{N-n-1} < u_{N-n} = \cdots = u_{N-1}$, which are not verified by the procedure. The interpolation conditions are given in the arrays `lbc` (for the knot u_n) and `rbc` (for the knot u_{N-n}). The first `d` numbers in each array specify the appropriate point of the first curve, the next `d` numbers describe the derivative vector, then the second order derivative etc. The data, which decsribe the interpolation conditions for the next curve begin at the positions `lbcpitch` and `rbcpitch` respectively.

The control points of the curves (i.e. the result of the construction) are stored in the array `ctlpoints`, whose pitch (distance between the beginnings of the data for consecutive curves) is specified by the input parameter `pitch`.

7.15 Constructing curves of approximation

One can impose more interpolation conditions for a function or a curve than the dimension of the appropriate space. The resulting system of equations is overdeterminate and often inconsistent. By solving the related least squares problem we obtain a function or a curve, which satisfies the interpolation conditions with some error. Such a construction may be done with the procedures described in this section.

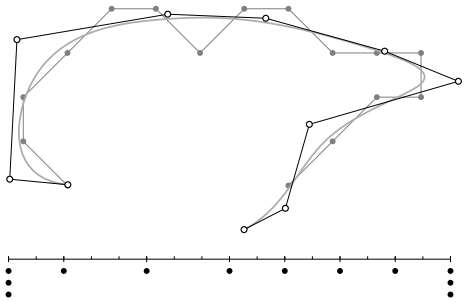


Figure 7.17. Planar B-spline curve of approximation.

A spline curve of approximation may be constructed by the procedure `mbs_multiConstructApproxBSCf` described later. The procedures described below are auxiliary and probably they will not be called directly by application programs.

```
boolean mbs_ApproxBSKnotsValidf ( int degree, int lastknot,
                                  const float *knots,
                                  int lastiknot, const float *iknots );
```

The procedure `mbs_ApproxBSKnotsValidf` verifies, whether the sequences of interpolation knots and spline curve knots satisfy the assumptions of the Schoenberg-Whitney theorem. If they do, then the construction of the curve of approximation is feasible.

```
int mbs_ApproxBSBandmSizef ( int degree, const float *knots,
                             int lastiknot, const float *iknots );
```

The procedure `mbs_ApproxBSBandmSizef` computes the length of the array needed to represent the band matrix of the system of linear equations solved as a linear least squares problem in the construction of the approximation curve.

The parameters `degree` and `knots` describe the space of spline functions, whose elements are to describe the curve (degree and the array of knots respectively; the length of this sequence is determined by the interpolation knots). The parameter `lastiknot` and `iknots` describe the *interpolation* knots of the curve — the interpolation conditions specified at these knots will be satisfied with some error.

The value returned by the procedure is the length of the array for storing the nonzero coefficients of the matrix of the system of equations.

```
boolean mbs_ConstructApproxBSProfilef ( int degree, int lastknot,
                            const float *knots,
                            int lastiknot, const float *iknots,
                            bandm_profile *prof );
```

The procedure `mbs_ConstructApproxBSProfilef` constructs the profile of the band matrix (see Section 3.2) of the system of equations solved in the construction of the approximation curve. The parameters `degree`, `lastknot`, `knots`, `lastiknot`, `iknots` describe the knots of the spline curve and the interpolation knots (see the description of the procedure `mbs_ApproxBSBandmSizef`).

The parameter `prof` points to the array of length `lastknot`−`degree`+1. The procedure stores the profile of the matrix in this array.

```
boolean mbs_ConstructApproxBSMatrixf ( int degree, int lastknot,
                              const float *knots,
                              int lastiknot, const float *iknots,
                              int *nrows, int *ncols,
                              bandm_profile *prof,
                              float *a );
```

The procedure `mbs_ConstructApproxBSMatrixf` computes the coefficients of the matrix of the system of equations, whose least squares solution represents the spline function or curve of approximation.

```
boolean mbs_multiConstructApproxBSCf ( int degree, int lastknot,
                              const float *knots,
                              int lastpknot, const float *pknots,
                              int ncurves, int spdimen,
                              int ppitch, const float *ppoints,
                              int bcpitch, float *ctlpoints );
```

The procedure `mbs_multiConstructApproxBSCf` constructs spline functions or curves of approximation, by setting up the appropriate system of linear equations and solving it as a linear least squares problem.

Parameters: `degree` — degree of the curves, `lastknot` — number of the last knot, `knots` — array of knots, `lastpknot` — number of the last interpolation knot, `pknots` — array with the interpolation knots, `ncurves` — number of curves, `spdimen` — dimension of the space with the curves.

The parameters `ppitch` and `ppoints` describe the interpolation conditions; `ppitch` is the pitch of the array `ppoints` with the points corresponding to the subsequent interpolation knots.

The parameter `bcpitch` is the pitch of the array `ctlpoints`, in which the procedure stores the control points of the curves of approximation.

The value returned by the procedure is `true` if the computation has been successful, or `false` otherwise. The reason of the failure may be that the knot sequences do not satisfy the assumptions of the Schoenberg-Whitney theorem (which leads to an irregular least squares problem), or insufficient scratch memory.

```
#define mbs_ConstructApproxBSC1f(degree,lastknot,knots,\
    lastpknot,pknots,ppoints,ctlpoints) \
  mbs_multiConstructApproxBSCf (degree,lastknot,knots,lastpknot,\
    pknots,1,1,0,(float*)ppoints,0,(float*)ctlpoints)
#define mbs_ConstructApproxBSC2f(degree,lastknot,knots,\
    lastpknot,pknots,ppoints,ctlpoints) \
  mbs_multiConstructApproxBSCf (degree,lastknot,knots,lastpknot,\
    pknots,1,2,0,(float*)ppoints,0,(float*)ctlpoints)
#define mbs_ConstructApproxBSC3f(degree,lastknot,knots,\
    lastpknot,pknots,ppoints,ctlpoints) ...
#define mbs_ConstructApproxBSC4f(degree,lastknot,knots,\
    lastpknot,pknots,ppoints,ctlpoints) ...
```

The above macros call `mbs_multiConstructApproxBSCf` in order to construct one curve of approximation in the space of dimension $1, \ldots, 4$.

7.16 Bézier curve clipping

```
boolean mbs_FindPolynomialZerosf ( int degree, const float *coeff,
                                   int *nzeros, float *zeros, float eps );
```

The procedure `mbs_FindPolynomialZerosf` computes real zeros of a polynomial of degree n in the interval $[0, 1]$.

Input parameters: `degree` — degree n of the polynomial, `coeff` — coefficients of the polynomial in the Bernstein basis of degree n, `eps` — required accuracy of the results (it must be a positive number).

Output parameters: `*nzeros` — the variable, which will be assigned the number of zeros found, `*zeros` — array in which the zeros will be stored. The length of this array must be at least n.

The value of the procedure is `true`, after the computation has been successful, and `false` otherwise, i.e. if not enough scratch memory was available.

```
void mbs_ClipBC2f ( int ncplanes, const vector3f *cplanes,
            int degree, const point2f *cpoints,
            void (*output) (int degree, const point2f *cpoints) );
```

The procedure `mbs_ClipBC2f` clips a planar polynomial Bézier curve to a convex polygon, i.e. it computes and outputs the arcs of the curve located inside that polygon.

The parameters: `ncplanes` — number of halfplanes, whose intersection is the polygon, `cplanes` — array with the representations of the halfplanes. For the halfplane $ax + by + c > 0$ the numbers a, b, c are coordinates of the appropriatre vector in the array `cplanes`.

The parameters `degree` and `cpoints` specify the curve, i.e. its degree and the control points respectively. The parameter `output` points to the procedure to be called in order to output (e.g. draw) each arc of the curve, located inside the polygon.

```
void mbs_ClipBC2Rf ( int ncplanes, const vector3f *cplanes,
            int degree, const point3f *cpoints,
            void (*output) (int degree, const point3f *cpoints) );
```

The procedure `mbs_ClipBC2Rf` clips a planar rational Bézier curve to a convex polygon, i.e. it computes and outputs the arcs of the curve located inside that polygon.

The parameters: `ncplanes` — number of halfplanes, whose intersection is the polygon, `cplanes` — array with the representations of the halfplanes. For the halfplane $ax + by + c > 0$ the numbers a, b, c are coordinates of the appropriatre vector in the array `cplanes`.

The parameters `degree` and `cpoints` specify the curve, i.e. its degree and the control points of the homogeneous curve respectively. The parameter `output` points

to the procedure to be called in order to output (e.g. draw) each arc of the curve, located inside the polygon.

```
void mbs_ClipBC3f ( int ncplanes, const vector4f *cplanes,
            int degree, const point3f *cpoints,
            void (*output) (int degree, const point3f *cpoints) );
```

The procedure `mbs_ClipBC3f` clips a polynomial Bézier curve in the 3D space to a convex polyhedron, i.e. it computes and outputs the arcs of the curve located inside that polyhedron.

The parameters: `ncplanes` — number of halfspaces, whose intersection is the polyhedron, `cplanes` — array with the representations of the halfspaces. For the halfspace $ax + by + cz + d > 0$ the numbers a, b, c, d are coordinates of the appropriatre vector in the array `cplanes`.

The parameters `degree` and `cpoints` specify the curve, i.e. its degree and the control points respectively. The parameter `output` points to the procedure to be called in order to output (e.g. draw) each arc of the curve, located inside the polygon.

```
void mbs_ClipBC3Rf ( int ncplanes, const vector4f *cplanes,
            int degree, const point4f *cpoints,
            void (*output) (int degree, const point4f *cpoints) );
```

The procedure `mbs_ClipBC3Rf` clips a rational Bézier curve in the 3D space to a convex polyhedron, i.e. it computes and outputs the arcs of the curve located inside that polyhedron.

The parameters: `ncplanes` — number of halfspaces, whose intersection is the polyhedron, `cplanes` — array with the representations of the halfspaces. For the halfspace $ax + by + cz + d > 0$ the numbers a, b, c, d are coordinates of the appropriatre vector in the array `cplanes`.

The parameters `degree` and `cpoints` specify the curve, i.e. its degree and the control points of the homogeneous curve respectively. The parameter `output` points to the procedure to be called in order to output (e.g. draw) each arc of the curve, located inside the polygon.

7.17 Polyline shape testing

```
boolean mbs_MonotonicPolylinef ( int spdimen, int npoints,
                                 int pitch, const float *points,
                                 const float *v );
boolean mbs_MonotonicPolylineRf ( int spdimen, int npoints,
                                  int pitch, const float *points,
                                  const float *v );
```

The procedures `mbs_MonotonicPolylinef` and `mbs_MonotonicPolylineRf` test, whether a polyline is monotonic with respect to the vector $\boldsymbol{v}$.

The polyline is in the space $\mathbb{R}^{\mathrm{d}}$, whose dimension is specified by the parameter `spdimen`. For the procedure `mbs_MonotonicPolylinef` its value must be d, and for the procedure `mbs_MonotonicPolylineRf` it must be $\mathrm{d}+1$.

The parameter `npoints` specifies the number of points. The cartesian coordinates (for the procedure `mbs_MonotonicPolylinef`) or the homogeneous coordinates `mbs_MonotonicPolylineRf`) of those points are given in the array `points`. The parameter `pitch` specifies the distance of the beginnings of the representations of consecutive points (which may be other than `spdimen`).

The parameter `v` points to the array with d numbers, the coordinates of the vector $\boldsymbol{v}$.

Each procedure returns `true` after detecting that the projections of consecutive points on the line, whose direction is given by the vector $\boldsymbol{v}$ are ordered along this line (and, in case of the procedure `mbs_MonotonicPolylineRf`, the weight coordinates of the points have the same sign) and `false` otherwise.

The procedures may be used to test, whether the control polylines of curves are monotonic with respect to the vector $\boldsymbol{v}$. This is a sufficient condition of monotonicity of Bézier and B-spline curves (assuming, for the rational curves, that all weight coordinates have the same sign).

7.18 Curve rasterization

The procedures of rasterization of curves represent pixels as structures of type `xpoint` and they use the buffer and its macros defined in the file `pkvaria.h`. In particular the curves of degree 1 are rasterized as line segments, using the procedure `_pkv_DrawLine` from the `libpkvaria` library.

```
void mbs_RasterizeBC2f ( int degree, const point2f *cpoints,
                         void (*output)(const xpoint *buf, int n),
                         boolean outlast );
void mbs_RasterizeBC2Rf ( int degree, const point3f *cpoints,
                          void (*output)(const xpoint *buf, int n),
                          boolean outlast );
```

The procedures `mbs_RasterizeBC2f` and `mbs_RasterizeBD2Rf` rasterize Bézier curves, i.e. they compute pixels which form eight-connected approximate images of the curves.

The parameters: `degree` — degree of the curve, `cpoints` — control points (for a rational curve must be represented with the homogeneous coordinates), `output` — an output procedurw (called to output pixels, i.e. on the screen). The parameter `outlast` specifies, whether the last pixel of the curve is to be output. Drawing a spline curve (consisting of more than one polynomial arc) or a closed curve one should not output the last pixel of each arc.

The number of calls of the `output` procedure depends on the number of pixels to draw and on the capacity of the internal buffer. The parameter `n` of the `output` procedure is the number of pixels to output.

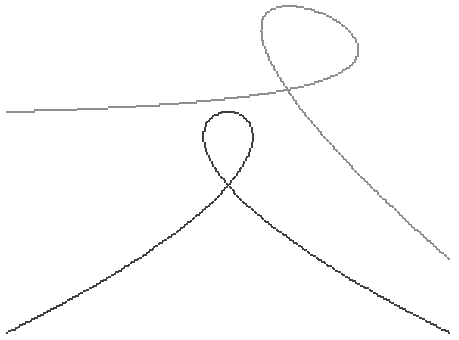


Figure 7.18. Raster images of polynomial and rational cubic Bézier curves.

```
void mbs_RasterizeBS2f ( int degree, int lastknot,
                         const float *knots,
                         const point2f *cpoints,
                         void (*output)(const xpoint *buf, int n),
                         boolean outlast );
void mbs_RasterizeBS2Rf ( int degree, int lastknot,
                          const float *knots,
                          const point3f *cpoints,
                          void (*output)(const xpoint *buf, int n),
                          boolean outlast );
```

The procedures `mbs_RasterizeBS2f` and `mbs_RasterizeBS2Rf` rasterize planar B-spline curves. The parameters `degree` (degree), `lastknot` (number of the last knot), `knots` (array with the knots) and `cpoints` (array with the control poitns) describe the curve. The parameter `output` points to the procedure which will be called in order to output or otherwise process the pixels. The parameter `outlast` specifies, whether the last pixel of the curve should be output or not.

TO DO: Clipping the curves before the rasterization. Testing, whether the curve is so short that its image consists of one pixel. Postprocessing the pixels in order to improve the smoothness of the image.

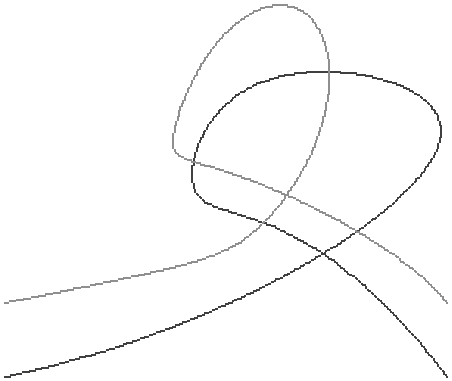


Figure 7.19. Raster images of cubic polynomial and rational B-spline curves.

7.19 Processing Coons patches

7.19.1 Polynomial patches

Polynomial Coons patches are represented with Bézier curves, whose degrees need not be the same. The domain of the patch is the square $[0,1]^2$, thus the numbers a, b, c, d discussed in Section 7.1.6 are $0, 1, 0, 1$ respectively.

```
void mbs_BezC1CoonsFindCornersf ( int spdimen,
                                  int degc00, const float *c00,
                                  int degc01, const float *c01,
                                  int degc10, const float *c10,
                                  int degc11, const float *c11,
                                  float *pcorners );
```

The procedure `mbs_BezC1CoonsFindCornersf` computes the matrix $\mathbf{P}$ of dimensions 4×4, whose elements are the points and derivatives of the curves $\mathbf{c}_{00}, \mathbf{c}_{10}, \mathbf{c}_{01}, \mathbf{c}_{11}$.

Parameters: `spdimen` — dimension d of the space with the curves and the bicubic polynomial Coons patch (of class C^1) represented by these curves. Each pair of the parameters `degc??` and `c??` describes one of the curves, its degree and the control points.

The parameter `pcorners` points to the array, in which the result is to be stored; the array length must be at least $16d$.

```
boolean mbs_BezC1CoonsToBezf ( int spdimen,
                               int degc00, const float *c00,
                               int degc01, const float *c01,
                               int degc10, const float *c10,
                               int degc11, const float *c11,
                               int degd00, const float *d00,
                               int degd01, const float *d01,
                               int degd10, const float *d10,
                               int degd11, const float *d11,
                               int *n, int *m, float *p );
```

The procedure `mbs_BezC1CoonsToBezf` finds the Bézier representation of the bicubic Coons patch (of class C^1), defined by given polynomial curves. The procedure value is `true`, if the computation was successful and `false` otherwise; the reason of failure may be insufficient space on the scratch memory stack.

The value of the parametr `spdimen` is the dimension d of the space, in which the curves and the patch are located. Each pair of parameters `degc??` and `c??` describes one of the curves $\mathbf{c}_{00}, \mathbf{c}_{01}, \mathbf{c}_{10}, \mathbf{c}_{11}$, by specifying its degree and Bézier control points. Each pair of parameters `degd??` and `d??` describe in the same

way one of the curves $\mathbf{d}_{00}, \mathbf{d}_{01}, \mathbf{d}_{10}, \mathbf{d}_{11}$. The curves must satisfy (up to rounding errors) the compatibility conditions (7.15), which *is not* verified.

The variables `*n` and `*m` obtain values, which describe the degree of the Bézier patch representation. The value n assigned to `*n` is the greatest value of the parameters `degc??` or 3 (if the number 3 is greater). Similarly the value m assigned to `*m` is the greatest value of the parameters `degd??`, or 3. The Bézier control points of the patch are stored in the array pointed by the parameter `p`; it must be long enough (at least $(n+1)(m+1)d$).

```
void mbs_TabCubicHFuncDer2f ( float a, float b,
                    int nkn, const float *kn,
                    float *hfunc, float *dhfunc, float *ddhfunc );
```

The procedure `mbs_TabCubicHFuncDer2f` evaluates the polynomials $\tilde{H}_{00}, \tilde{H}_{10}$, $\tilde{H}_{01}$ and $\tilde{H}_{11}$, being the basis of definition of bicubic Coons patches, and their derivatives of order 1 and 2. The result of this computation may be used to compute a number of points of a Coons patch corresponding to a rectangular net in the domain, using the procedure `mbs_TabBezC1CoonsDer2f` (a polynomial patch) or `mbs_TabBSC1CoonsDer2f` (a spline patch).

The parameters `a` and `b` describe the end points of the interval taken as the domain of the curves $\mathbf{c}_{ij}$ or $\mathbf{d}_{ij}$; for polynomial Coons patches these parameters must have values 0 and 1 respectively.

The parameter `nkn` specifies the number `k` of points $u_m \in [a, b]$, at which the polynomials are to be evaluated; these points (floating point numbers) are given in the array `kn`.

The values of the polynomials and their derivatives of order 1 and 2 are stored in the arrays `hfunc`, `dhfunc` and `ddhfunc` respectively. The arrays must have length at least $4k$; to each subsequent four positions in the array the values of the four polynomials or their derivatives at the subsequent point u_m are assigned.

```
void mbs_TabCubicHFuncDer3f ( float a, float b, int nkn,
                    const float *kn,
                    float *hfunc, float *dhfunc, float *ddhfunc,
                    float *dddhfunc );
```



```
boolean mbs_TabBezC1CoonsDer2f ( int spdimen,
        int nknu, const float *knu, const float *hfuncu,
        const float *dhfuncu, const float *ddhfuncu,
        int nknv, const float *knv, const float *hfuncv,
        const float *dhfuncv, const float *ddhfuncv,
        int degc00, const float *c00,
        int degc01, const float *c01,
        int degc10, const float *c10,
        int degc11, const float *c11,
        int degd00, const float *d00,
        int degd01, const float *d01,
        int degd10, const float *d10,
        int degd11, const float *d11,
        float *p, float *pu, float *pv,
        float *puu, float *puv, float *pvv );
```

The procedure `mbs_TabBezC1CoonsDer2f` performs a fast computation of points and derivatives of order 1 and 2 of a bicubic polynomial Coons patch, at the points (u_i, v_j), where $i \in \{0, \ldots, k-1\}$, $j \in \{0, \ldots, l-1\}$.

The parameter `spdimen` specifies the dimension `d` of the space with the patch. The parameter `nknu` specifies the number `k`, the array `knu` contains the numbers $u_0, \ldots, u_{k-1}$. The contents of the arrays `hfuncu`, `dhfuncu`, `ddhfuncu` must be respectively the values of the polynomials $H_{00}, H_{10}, H_{01}, H_{11}$ and their derivatives of order 1 and 2 at the points $u_0, \ldots, u_{k-1}$; these values are simplest to obtain by calling the procedure `mbs_TabCubicHFuncDer2f` (with the parameters `a` $= 0$, `b` $= 1$).

The numbers $v_0, \ldots, v_{l-1}$ and the values of the functions H_{ij} and their derivatives at v_j are analoguously specified by the parameters `nknv`, `knv`, `hfuncv`, `dhfuncv`, `ddhfuncv`.

The pairs of parameters `degc??`, `c??` and `degd??`, `d??` describe the Bézier curves, which define the patch. These curves must satisfy (up to the rounding errors) the compatibility conditions (7.15).

In the arrays pointed by the parameters `p`, `pu`, `pv`, `puu`, `puv`, `pvv` the procedure stores the computed points and derivatives of order 1 and 2; if any of the parameters is `NULL`, then the corresponding points or vectors are not computed. Otherwise the pointed array must have length at least $k^2 d$.

The value returned is `true` in case of success and `false` after failure (caused by insufficient space on the scratch memory stack).

```
boolean mbs_TabBezC1CoonsDer3f ( int spdimen,
        int nknu, const float *knu, const float *hfuncu,
        const float *dhfuncu, const float *ddhfuncu,
        const float *dddhfuncu,
        int nknv, const float *knv, const float *hfuncv,
        const float *dhfuncv, const float *ddhfuncv,
        const float *dddhfuncv,
        int degc00, const float *c00,
        int degc01, const float *c01,
        int degc10, const float *c10,
        int degc11, const float *c11,
        int degd00, const float *d00,
        int degd01, const float *d01,
        int degd10, const float *d10,
        int degd11, const float *d11,
        float *p, float *pu, float *pv,
        float *puu, float *puv, float *pvv,
        float *puuu, float *puuv, float *puvv, float *pvvv );
```

```
boolean mbs_TabBezC1Coons0Der2f ( int spdimen,
        int nknu, const float *knu, const float *hfuncu,
        const float *dhfuncu, const float *ddhfuncu,
        int nknv, const float *knv, const float *hfuncv,
        const float *dhfuncv, const float *ddhfuncv,
        int degc00, const float *c00,
        int degc01, const float *c01,
        int degd00, const float *d00,
        int degd01, const float *d01,
        float *p, float *pu, float *pv,
        float *puu, float *puv, float *pvv );
```

The procedure `mbs_TabBezC1Coons0Der2f` is a simplified version of the procedure `mbs_TabBezC1CoonsDer2f` for the case, when the curves $\mathbf{c}_{10}, \mathbf{c}_{11}, \mathbf{d}_{10}$ and $\mathbf{d}_{11}$ are null (i.e. when all their control points have all coordinates zero). Computing points of such a patch may be done in a shorter time; this procedure is used by the library `libg1hole`.

The parameters of `mbs_TabBezC1Coons0Der2f` are the same as the parameters of the procedure `mbs_TabBezC1CoonsDer2f` with the same names.


```
boolean mbs_TabBezC1Coons0Der3f ( int spdimen,
        int nknu, const float *knu, const float *hfuncu,
        const float *dhfuncu, const float *ddhfuncu,
        const float *dddhfuncu,
        int nknv, const float *knv, const float *hfuncv,
        const float *dhfuncv, const float *ddhfuncv,
        const float *dddhfuncv,
        int degc00, const float *c00,
        int degc01, const float *c01,
        int degd00, const float *d00,
        int degd01, const float *d01,
        float *p, float *pu, float *pv,
        float *puu, float *puv, float *pvv,
        float *puuu, float *puuv, float *puvv, float *pvvv );
```

```
void mbs_BezC2CoonsFindCornersf ( int spdimen,
                                  int degc00, const float *c00,
                                  int degc01, const float *c01,
                                  int degc02, const float *c02,
                                  int degc10, const float *c10,
                                  int degc11, const float *c11,
                                  int degc12, const float *c12,
                                  float *pcorners );
```

The procedure `mbs_BezC2CoonsFindCornersf` computes the matrix $\mathbf{P}$ of dimensions 6×6, whose elements are the points and derivatives of the curves $\mathbf{c}_{00}, \mathbf{c}_{10}, \mathbf{c}_{01}, \mathbf{c}_{11}, \mathbf{c}_{02}, \mathbf{c}_{12}$.

Parameters: `spdimen` — dimension `d` of the space with the curves and the biquintic polynomial Coons patch (of class C^2) represented by these curves. Each pair of the parameters `degc??` and `c??` describes one of the curves, its degree and the control points.

The parameter `pcorners` points to the array, in which the result is to be stored; the array length must be at least 36d.

```
boolean mbs_BezC2CoonsToBezf ( int spdimen,
                               int degc00, const float *c00,
                               int degc01, const float *c01,
                               int degc02, const float *c02,
                               int degc10, const float *c10,
                               int degc11, const float *c11,
                               int degc12, const float *c12,
                               int degd00, const float *d00,
                               int degd01, const float *d01,
                               int degd02, const float *d02,
                               int degd10, const float *d10,
                               int degd11, const float *d11,
                               int degd12, const float *d12,
                               int *n, int *m, float *p );
```

The procedure `mbs_BezC2CoonsToBezf` converts a biquintic Coons patch to the Bézier form. The patch is represented by 12 polynomial curves, which describe its boundary (the curves c_{00}, c_{10}, d_{00}, d_{10}) and the cross derivatives of the first (c_{01}, c_{11}, d_{01}, d_{11}) and second (c_{02}, c_{12}, d_{02}, d_{12})) order. All these curves are given in Bézier form, their degrees are specified respectively by the parameters `degc00`, ..., `degd12`, their control points (in the space of dimension `spdimen`) are given in the arrays `c00`, ..., `d12`.

The output parameters are `*n` and `*m`, which obtain the values indicating the degree and the array `p`, in which the Bézier control points of the patch are stored.

```
void mbs_TabQuinticHFuncDer3f ( float a, float b,
                                int nkn, const float *kn,
                                float *hfunc, float *dhfunc,
                                float *ddhfunc, float *dddhfunc );
```

The procedure `mbs_TabQuinticHFuncDer3f` evaluates the polynomials $\tilde{H}_{00}$, $\tilde{H}_{10}$, $\tilde{H}_{01}$, $\tilde{H}_{11}$, $\tilde{H}_{02}$ and $\tilde{H}_{12}$, being the basis of definition of biquintic Coons patches, and their derivatives of order 1, 2 and 3. The result of this computation may be used to compute a number of points of a Coons patch corresponding to a rectangular net in the domain, using the procedure `mbs_TabBezC2CoonsDer3f` (a polynomial patch) or `mbs_TabBSC2CoonsDer3f` (a spline patch).

The parameters `a` and `b` describe the end points of the interval taken as the domain of the curves $\mathbf{c}_{ij}$ or $\mathbf{d}_{ij}$; for polynomial Coons patches these parameters must have values 0 and 1 respectively.

The parameter `nkn` specifies the number `k` of points $u_m \in [a, b]$, at which the polynomials are to be evaluated; these points (floating point numbers) are given in the array `kn`.

The values of the polynomials and their derivatives of order 1, 2 and 3 are stored in the arrays `hfunc`, `dhfunc`, `ddhfunc` and `dddhfunc` respectively. The arrays must

have length at least 4k; to each subsequent four positions in the array the values of the four polynomials or their derivatives at the subsequent point u_m are assigned.

```
boolean mbs_TabBezC2CoonsDer3f ( int spdimen,
        int nknu, const float *knu, const float *hfuncu,
        const float *dhfuncu, const float *ddhfuncu,
        const float *dddhfuncu,
        int nknv, const float *knv, const float *hfuncv,
        const float *dhfuncv, const float *ddhfuncv,
        const float *dddhfuncv,
        int degc00, const float *c00,
        int degc01, const float *c01,
        int degc02, const float *c02,
        int degc10, const float *c10,
        int degc11, const float *c11,
        int degc12, const float *c12,
        int degd00, const float *d00,
        int degd01, const float *d01,
        int degd02, const float *d02,
        int degd10, const float *d10,
        int degd11, const float *d11,
        int degd12, const float *d12,
        float *p, float *pu, float *pv, float *puu,
        float *puv, float *pvv,
        float *puuu, float *puuv, float *puvv, float *pvvv );
```

The procedure `mbs_TabBezC2CoonsDer3f` performs a fast computation of points and derivatives of order 1, 2 and 3 of a biquintic polynomial Coons patch, at the points (u_i, v_j), where $i \in \{0, \ldots, k-1\}$, $j \in \{0, \ldots, l-1\}$.

The parameter `spdimen` specifies the dimension `d` of the space with the patch. The parameter `nknu` specifies the number `k`, the array `knu` contains the numbers $u_0, \ldots, u_{k-1}$. The contents of the arrays `hfuncu`, `dhfuncu`, `ddhfuncu`, `dddhfuncu` must be respectively the values of the polynomials $H_{00}, H_{10}, H_{01}, H_{11}, H_{02}, H_{12}$ and their derivatives of order 1, 2 and 3 at the points $u_0, \ldots, u_{k-1}$; these values are simplest to obtain by calling the procedure `mbs_TabQuinticHFuncDer3f` (with the parameters $a = 0$, $b = 1$).

The sequence $v_0, \ldots . v_{l-1}$ and the values of H_{ij} at the points of this sequence are analoguously represented by the parameters `nknv`, `knv`, `hfuncv`, `dhfuncv`, `ddhfuncv`, `dddhfuncv`.

The pairs of parameters `degc??`, `c??` and `degd??`, `d??` describe the Bézier curves, which define the patch. These curves must satisfy (up to the rounding errors) the compatibility conditions (7.15).

In the arrays pointed by the parameters `p`, `pu`, `pv`, `puu`, `puv`, `pvv`, `puuu`, `puuv`,

puvv, pvvv the procedure stores the computed points and derivatives of order 1, 2 and 3; if any of the parameters is NULL, then the corresponding points or vectors are not computed. Otherwise the pointed array must have length at least $k^2 d$.

The value returned is true in case of success and false after failure (caused by insufficient space on the scratch memory stack).

```
boolean mbs_TabBezC2Coons0Der3f ( int spdimen,
        int nknu, const float *knu, const float *hfuncu,
        const float *dhfuncu, const float *ddhfuncu,
        const float *dddhfuncu,
        int nknv, const float *knv, const float *hfuncv,
        const float *dhfuncv, const float *ddhfuncv,
        const float *dddhfuncv,
        int degc00, const float *c00,
        int degc01, const float *c01,
        int degc02, const float *c02,
        int degd00, const float *d00,
        int degd01, const float *d01,
        int degd02, const float *d02,
        float *p, float *pu, float *pv, float *puu, float *puv,
        float *pvv,
        float *puuu, float *puuv, float *puvv, float *pvvv );
```

The procedure mbs_TabBezC2Coons0Der3f is a simplified version of the procedure mbs_TabBezC2CoonsDer3f for the case, when the curves $\mathbf{c}_{10}, \mathbf{c}_{11}, \mathbf{c}_{12}, \mathbf{d}_{10}, \mathbf{d}_{11}$ and $\mathbf{d}_{12}$ are null (i.e. when all their control points have all coordinates zero). Computing points of such a patch may be done in a shorter time; this procedure is used by the library libg1hole.

The parameters of mbs_TabBezC2Coons0Der3f are the same as the parameters of the procedure mbs_TabBezC2CoonsDer3f with the same names.

7.19.2 Spline patches

Spline Coons patches are defined with B-spline curves; they may have different degrees and different knot sequences; the only restriction is that all curves $\mathbf{c}_{ij}$ must have the same domain (determined by their boundary knots) and the same concerns the curves $\mathbf{d}_{ij}$.

```
void mbs_BSC1CoonsFindCornersf ( int spdimen,
        int degc00, int lastknotc00, const float *knotsc00,
        const float *c00,
        int degc01, int lastknotc01, const float *knotsc01,
        const float *c01,
        int degc10, int lastknotc10, const float *knotsc10,
        const float *c10,
        int degc11, int lastknotc11, const float *knotsc11,
        const float *c11,
        float *pcorners );
```

The procedure mbs_BSC1CoonsFindCornersf computes the matrix $\mathbf{P}$ of dimensions 4×4, whose elements are the points and derivatives of the curves $\mathbf{c}_{00}, \mathbf{c}_{10}, \mathbf{c}_{01}, \mathbf{c}_{11}$.

Parameters: spdimen — dimension d of the space with the curves and the bicubic spline Coons patch (of class C^1) represented by these curves. Each quadruple of the parameters degc??, lastknotc??, knotsc?? and c?? describes one of the curves, its degree, number of the last knot, knots and the control points respectively.

The parameter pcorners points to the array, in which the result is to be stored; the array length must be at least 16d.

```
boolean mbs_BSC1CoonsToBSf ( int spdimen,
        int degc00, int lastknotc00, const float *knotsc00,
        const float *c00,
        int degc01, int lastknotc01, const float *knotsc01,
        const float *c01,
        int degc10, int lastknotc10, const float *knotsc10,
        const float *c10,
        int degc11, int lastknotc11, const float *knotsc11,
        const float *c11,
        int degd00, int lastknotd00, const float *knotsd00,
        const float *d00,
        int degd01, int lastknotd01, const float *knotsd01,
        const float *d01,
        int degd10, int lastknotd10, const float *knotsd10,
        const float *d10,
        int degd11, int lastknotd11, const float *knotsd11,
        const float *d11,
        int *degreeu, int *lastuknot, float *uknots,
        int *degreev, int *lastvknot, float *vknots, float *p );
```

The procedure mbs_BSC1CoonsToBSf finds a B-spline representation of a bicubic Coons patch (of class C^1), defined by given spline curves. The value returned is true if the computation has been successful and false otherwise (the reason of failure may be insufficient space on the scratch memory stack or incorrect knot sequences of the given curves).

The value of the parametes spdimen is the dimension d of the space with the curves and the patch. Subsequent quadruples of parameters degc??, lastknotc??, knotsc?? and c?? describe the appropriate curve of the family $\mathbf{c}_{00}, \mathbf{c}_{01}, \mathbf{c}_{10}, \mathbf{c}_{11}$, by specifying the degree, number of the last knot, knot sequence and the array of control points. The quadruples of parameters degd??, lastknotd??, knotsd?? and d?? in the same way describe the curves of the family $\mathbf{d}_{00}, \mathbf{d}_{01}, \mathbf{d}_{10}, \mathbf{d}_{11}$. The curves must satisfy (up to rounding errors) the compatibility conditions (7.15), *which is not verified.*

The variables *n and *m are assigned the values, which describe the degree of the B-spline representation of the patch. The value n of the variable *n is the greatest of the values of the parameters degc?? or 3 (if the number 3 is greater). Similarly, the value m of the variable *m is the greatest of the values of the parameters degd?? or 3. The parameters lastuknot, uknots, lastvknot i vknots are used to output the knot sequences of the B-spline representation. In the array pointed by the parameter p the procedure stores the control points.

The arrays unkots, vknots and p must be long enough; their lengths may be computed before the allocation by calling mbs_FindBSCommonKnotSequencef for the families of curves $\mathbf{c}_{ij}$ and $\mathbf{d}_{ij}$; the variables pointed by the parameter lastknot

of this procedure must have the initial value 3.

```
boolean mbs_TabBSC1CoonsDer2f ( int spdimen,
        int nknu, const float *knu, const float *hfuncu,
        const float *dhfuncu, const float *ddhfuncu,
        int nknv, const float *knv, const float *hfuncv,
        const float *dhfuncv, const float *ddhfuncv,
        int degc00, int lastknotc00, const float *knotsc00,
        const float *c00,
        int degc01, int lastknotc01, const float *knotsc01,
        const float *c01,
        int degc10, int lastknotc10, const float *knotsc10,
        const float *c10,
        int degc11, int lastknotc11, const float *knotsc11,
        const float *c11,
        int degd00, int lastknotd00, const float *knotsd00,
        const float *d00,
        int degd01, int lastknotd01, const float *knotsd01,
        const float *d01,
        int degd10, int lastknotd10, const float *knotsd10,
        const float *d10,
        int degd11, int lastknotd11, const float *knotsd11,
        const float *d11,
        float *p, float *pu, float *pv,
        float *puu, float *puv, float *pvv );
```

The procedure mbs_TabBSC1CoonsDer2f performs a fast computation of points and derivatives of order 1 and 2 of a bicubic spline Coons patch, at the points (u_i, v_j), where $i \in \{0, \ldots, k_u - 1\}$, $j \in \{0, \ldots, k_v - 1\}$.

The parameter spdimen specifies the dimension d of the space with the patch. The parameters nknu and nknv specify the numbers k_u and k_v, the arrays knu and knv contain respectively the numbers $u_0, \ldots, u_{k_u-1}$ and $v_0, \ldots, v_{k_v-1}$. The contents of the arrays hfuncu, dhfuncu, ddhfuncu must be respectively the values of the polynomials $\tilde{H}_{00}, \tilde{H}_{10}, \tilde{H}_{01}, \tilde{H}_{11}$ and their derivatives of order 1 and 2 at the points $u_0, \ldots, u_{k_u-1}$. The arrays hfuncv, dhfuncv and ddhfuncv must contain the values of the polynomials $\hat{H}_{00}, \hat{H}_{10}, \hat{H}_{01}, \hat{H}_{11}$ at $v_0, \ldots, v_{k_v-1}$; these values are simplest to obtain by calling the procedure mbs_TabCubicHFuncDer2f.

The quadruples of parameters degc??, lastknotc??, knotsc??, c?? and degd??, lastknotd??, knotsd??, d?? describe the B-spline curves, which define the patch. These curves must satisfy (up to the rounding errors) the compatibility conditions (7.15).

In the arrays pointed by the parameters p, pu, pv, puu, puv, pvv the procedure stores the computed points and derivatives of order 1 and 2; if any of the parameters

is NULL, then the corresponding points or vectors are not computed. Otherwise the pointed array must have length at least $k_u k_v d$.

The value returned is true in case of success and false after failure (caused by insufficient space on the scratch memory stack or incorrect knot sequences of the curves).

```
boolean mbs_TabBSC1Coons0Der2f ( int spdimen,
        int nknu, const float *knu, const float *hfuncu,
        const float *dhfuncu, const float *ddhfuncu,
        int nknv, const float *knv, const float *hfuncv,
        const float *dhfuncv, const float *ddhfuncv,
        int degc00, int lastknotc00, const float *knotsc00,
        const float *c00,
        int degc01, int lastknotc01, const float *knotsc01,
        const float *c01,
        int degd00, int lastknotd00, const float *knotsd00,
        const float *d00,
        int degd01, int lastknotd01, const float *knotsd01,
        const float *d01,
        float *p, float *pu, float *pv,
        float *puu, float *puv, float *pvv );
```

The procedure mbs_TabBSC1Coons0Der2f is a simplified version of the procedure mbs_TabBSC1CoonsDer2f for the case, when the curves $\mathbf{c}_{10}, \mathbf{c}_{11}, \mathbf{d}_{10}$ and $\mathbf{d}_{11}$ are null (i.e. when all their control points have all coordinates zero). Computing points of such a patch may be done in a shorter time.

The parameters of mbs_TabBSC1Coons0Der2f are the same as the parameters of the procedure mbs_TabBSC1CoonsDer2f with the same names.

```
void mbs_BSC2CoonsFindCornersf ( int spdimen,
        int degc00, int lastknotc00, const float *knotsc00,
        const float *c00,
        int degc01, int lastknotc01, const float *knotsc01,
        const float *c01,
        int degc02, int lastknotc02, const float *knotsc02,
        const float *c02,
        int degc10, int lastknotc10, const float *knotsc10,
        const float *c10,
        int degc11, int lastknotc11, const float *knotsc11,
        const float *c11,
        int degc12, int lastknotc12, const float *knotsc12,
        const float *c12,
        float *pcorners );
```


The procedure mbs_BSC2CoonsFindCornersf computes the matrix $\mathbf{P}$ of dimensions 6×6, whose elements are the points and derivatives of the curves $\mathbf{c}_{00}, \mathbf{c}_{10}, \mathbf{c}_{01}, \mathbf{c}_{11}, \mathbf{c}_{02}, \mathbf{c}_{12}$.

Parameters: spdimen — dimension d of the space with the curves and the biquintic spline Coons patch (of class C^2) represented by these curves. Each quadruple of the parameters degc??, lastknotc??, knotsc?? and c?? describes one of the curves, its degree, number of the last knot, knots and the control points respectively.

The parameter pcorners points to the array, in which the result is to be stored; the array length must be at least $36d$.

```
boolean mbs_BSC2CoonsToBSf ( int spdimen,
        int degc00, int lastknotc00, const float *knotsc00,
        const float *c00,
        int degc01, int lastknotc01, const float *knotsc01,
        const float *c01,
        Aint degc02, int lastknotc02, const float *knotsc02,
        const float *c02,
        int degc10, int lastknotc10, const float *knotsc10,
        const float *c10,
        int degc11, int lastknotc11, const float *knotsc11,
        const float *c11,
        int degc12, int lastknotc12, const float *knotsc12,
        const float *c12,
        int degd00, int lastknotd00, const float *knotsd00,
        const float *d00,
        int degd01, int lastknotd01, const float *knotsd01,
        const float *d01,
        int degd02, int lastknotd02, const float *knotsd02,
        const float *d02,
        int degd10, int lastknotd10, const float *knotsd10,
        const float *d10,
        int degd11, int lastknotd11, const float *knotsd11,
        const float *d11,
        int degd12, int lastknotd12, const float *knotsd12,
        const float *d12,
        int *degreeu, int *lastuknot, float *uknots,
        int *degreev, int *lastvknot, float *vknots, float *p );
```

The procedure mbs_BSC2CoonsToBSf finds a B-spline representation of a biquintic Coons patch (of class C^2), defined by given spline curves. The value returned is true if the computation has been successful and false otherwise (the reason of failure may be insufficient space on the scratch memory stack or incorrect knot

sequences of the given curves).

The value of the parametes spdimen is the dimension d of the space with the curves and the patch. Subsequent quadruples of parameters degc??, lastknotc??, knotsc?? and c?? describe the appropriate curve of the family $\mathbf{c}_{00}, \mathbf{c}_{01}, \mathbf{c}_{10}, \mathbf{c}_{11}$, by specifying the degree, number of the last knot, knot sequence and the array of control points. The quadruples of parameters degd??, lastknotd??, knotsd?? and d?? in the same way describe the curves of the family $\mathbf{d}_{00}, \mathbf{d}_{01}, \mathbf{d}_{10}, \mathbf{d}_{11}$. The curves must satisfy (up to rounding errors) the compatibility conditions (7.15), *which is not verified*.

The variables *n and *m are assigned the values, which describe the degree of the B-spline representation of the patch. The value n of the variable *n is the greatest of the values of the parameters degc?? or 5 (if the number 5 is greater). Similarly, the value m of the variable *m is the greatest of the values of the parameters degd?? or 5. The parameters lastuknot, uknots, lastvknot i vknots are used to output the knot sequences of the B-spline representation. In the array pointed by the parameter p the procedure stores the control points.

The arrays unkots, vknots and p must be long enough; their lengths may be computed before the allocation by calling mbs_FindBSCommonKnotSequencef for the families of curves $\mathbf{c}_{ij}$ and $\mathbf{d}_{ij}$; the variables pointed by the parameter lastknot of this procedure must have the initial value 5.


```
boolean mbs_TabBSC2CoonsDer3f ( int spdimen,
        int nknu, const float *knu, const float *hfuncu,
        const float *dhfuncu, const float *ddhfuncu,
        const float *dddhfuncu,
        int nknv, const float *knv, const float *hfuncv,
        const float *dhfuncv, const float *ddhfuncv,
        const float *dddhfuncv,
        int degc00, int lastknotc00, const float *knotsc00,
        const float *c00,
        int degc01, int lastknotc01, const float *knotsc01,
        const float *c01,
        int degc02, int lastknotc02, const float *knotsc02,
        const float *c02,
        int degc10, int lastknotc10, const float *knotsc10,
        const float *c10,
        int degc11, int lastknotc11, const float *knotsc11,
        const float *c11,
        int degc12, int lastknotc12, const float *knotsc12,
        const float *c12,
        int degd00, int lastknotd00, const float *knotsd00,
        const float *d00,
        int degd01, int lastknotd01, const float *knotsd01,
        const float *d01,
        int degd02, int lastknotd02, const float *knotsd02,
        const float *d02,
        int degd10, int lastknotd10, const float *knotsd10,
        const float *d10,
        int degd11, int lastknotd11, const float *knotsd11,
        const float *d11,
        int degd12, int lastknotd12, const float *knotsd12,
        const float *d12,
        float *p, float *pu, float *pv,
        float *puu, float *puv, float *pvv,
        float *puuu, float *puuv, float *puvv, float *pvvv );
```

The procedure mbs_TabBSC2CoonsDer3f performs a fast computation of points and derivatives of order 1, 2 and 3 of a biquintic spline Coons patch, at the points (u_i, v_j), where $i \in \{0, \ldots, k_u - 1\}$, $j \in \{0, \ldots, k_v - 1\}$.

The parameter spdimen specifies the dimension d of the space with the patch. The parameters nknu and nknv specify the numbers k_u and k_v, the arrays knu and knv contain respectively the numbers $u_0, \ldots, u_{k_u-1}$ and $v_0, \ldots, v_{k_v-1}$. The contents of the arrays hfuncu, dhfuncu, ddhfuncu and dddhfuncu must be respectively the values of the polynomials $\tilde{H}_{00}, \tilde{H}_{10}, \tilde{H}_{01}, \tilde{H}_{11}, \tilde{H}_{02}, \tilde{H}_{12}$ and their

derivatives of order 1, 2 and 3 at the points $u_0, \ldots, u_{k_u-1}$. The arrays hfuncv, dhfuncv, ddhfuncv and dddhfuncv must contain the values of the polynomials $\hat{H}_{00}, \hat{H}_{10}, \hat{H}_{01}, \hat{H}_{11}, \hat{H}_{02}, \hat{H}_{12}$ at $v_0, \ldots, v_{k_v-1}$; these values are simplest to obtain by calling the procedure mbs_TabQuinticHFuncDer3f.

The quadruples of parameters degc??, lastknotc??, knotsc??, c?? and degd??, lastknotd??, knotsd?? d?? describe the B-spline curves, which define the patch. These curves must satisfy (up to the rounding errors) the compatibility conditions (7.15).

In the arrays pointed by the parameters p, pu, pv, puu, puv, pvv, puuu, puuv, puvv and pvvv the procedure stores the computed points and derivatives of order 1, 2 and 3; if any of the parameters is NULL, then the corresponding points or vectors are not computed. Otherwise the pointed array must have length at least $k_u k_v d$.

The value returned is true in case of success and false after failure (caused by insufficient space on the scratch memory stack or incorrect knot sequences of the curves).

```
boolean mbs_TabBSC2Coons0Der3f ( int spdimen,
        int nknu, const float *knu, const float *hfuncu,
        const float *dhfuncu, const float *ddhfuncu,
        const float *dddhfuncu,
        int nknv, const float *knv, const float *hfuncv,
        const float *dhfuncv, const float *ddhfuncv,
        const float *dddhfuncv,
        int degc00, int lastknotc00, const float *knotsc00,
        const float *c00,
        int degc01, int lastknotc01, const float *knotsc01,
        const float *c01,
        int degc02, int lastknotc02, const float *knotsc02,
        const float *c02,
        int degd00, int lastknotd00, const float *knotsd00,
        const float *d00,
        int degd01, int lastknotd01, const float *knotsd01,
        const float *d01,
        int degd02, int lastknotd02, const float *knotsd02,
        const float *d02,
        float *p, float *pu, float *pv,
        float *puu, float *puv, float *pvv,
        float *puuu, float *puuv, float *puvv, float *pvvv );
```

The procedure mbs_TabBSC2Coons0Der3f is a simplified version of the procedure mbs_TabBSC2CoonsDer3f for the case, when the curves $\mathbf{c}_{10}, \mathbf{c}_{11}, \mathbf{c}_{12}, \mathbf{d}_{10}, \mathbf{d}_{11}$ and $\mathbf{d}_{12}$ are null (i.e. when all their control points have all coordinates zero). Computing points of such a patch may be done in a shorter time.

The parameters of mbs_TabBSC2Coons0Der3f are the same as the parameters of the procedure mbs_TabBSC2CoonsDer3f with the same names.

7.20 Spherical product

The spherical product of two planar parametric curves, $\mathbf{p}(t) = [x_{\mathbf{p}}(t), y_{\mathbf{p}}(t)]^T$ and $\mathbf{q}(t) = [x_{\mathbf{q}}(t), y_{\mathbf{q}}(t)]^T$, is the parametric surface in $\mathbb{R}^3$, given by

$$\mathbf{s}(u,v) = \begin{bmatrix} x_{\mathbf{p}}(u)x_{\mathbf{q}}(v) \\ y_{\mathbf{p}}(u)x_{\mathbf{q}}(v) \\ y_{\mathbf{q}}(v) \end{bmatrix}.$$

The curves $\mathbf{p}$ and $\mathbf{q}$ are called respectively equator and meridian. The procedures described below compute the control points of the B-spline representation of the spherical product of planar B-spline curves, piecewise polynomial and piecewise rational respectively.

The knot sequence of the equator is the „u" knot sequence and the knot sequence of the meridian is the „v" knot sequence of the spherical product.

```
void mbs_SphericalProductf (
      int degree_eq, int lastknot_eq, const point2f *cpoints_eq,
      int degree_mer, int lastknot_mer, const point2f *cpoints_mer,
      int pitch, point3f *spr_cp );
```

```
void mbs_SphericalProductRf (
      int degree_eq, int lastknot_eq, const point3f *cpoints_eq,
      int degree_mer, int lastknot_mer, const point3f *cpoints_mer,
      int pitch, point4f *spr_cp );
```


7.21 Drawing trimmed patches

7.21.1 Domain representation

The domain of a trimmed B-spline patch of degree (n, m), with the knots $u_0, \ldots, u_N$ and $v_0, \ldots, v_M$ is a subset of the rectangle $[u_n, u_{N-n}] \times [v_m, v_{M-m}]$. In particular, it is always a bounded area. The boundary of the domain is the sum of planar curvilinear closed polylines. Every such a polyline may consist of

- polylines (sequences of line segments),
- Bézier curves,
- B-spline curves,

called hereafter the boundary elements. The points (vertices of the polylines and control points of the curves) may be given with the cartesian coordinates (then they are of type `point2f`) or homogeneous coordinates (in this case they are of type `vector3f`).

The data describing every such a polyline must satisfy the following condition: the B-spline curves must be continuous and the end point of each boundary element (polyline or curve) is the first point of the of the element that follows (the last element is followed by the first one). If this condition is not satisfied then the procedures of drawing trimmed patches will insert the appropriate line segments.

Another condition is the absence of points at infinity. It is sufficient that all weight coordinates are positive, though it is not required. However, specifying an unbounded patch boundary may cause a program execution error. Therefore all polylines and curves must lie in the rectangle specified as the domain of the untrimmed patch.

The boundary of the domain is represented with an array containing structures of type `polycurvef`.

```
typedef struct{
    boolean closing;
    byte    spdimen;
    short   degree;
    int     lastknot;
    float   *knots;
    float   *points;
  } polycurvef;
```

The attribute `closing` specifies the boundary element following the current one. If its value is `false`, then there is a new element after the current one (and it is described by the next structure in the array). If `closing` is `true`, then the end of the current element should be connected with the beginning of the first element in

the array, or with the last of the preceding elements, preceded by an element with the `closing` attribute `true` (in this way the curvilinear polygon is closed).

The attribute `spdimen` may be equal to 2 or 3. In the former case the attribute `points` points to an array of structures `point2f` with the cartesian coordinates of points in the plane. In the latter case it points to an array of structures `vector3f`, with the homogeneous coordinates of points (the curve, whose control points are represented in this way is rational, and its representation is homogeneous).

The attribute `degree` specifies the degree n of the curve, it must not be less than 1.

The attribute `lastknot` specifies the number N of the last vertex of the polyline or the last knot of the B-spline curve.

The attribute `knots` points to an array with the knots of the B-spline curve, of length $N + 1$.

The attribute `points` points to an array with the vertices of the polyline or the control polygon. Depending on the value of the attribute `spdimen` this array contains pairs of triples of floating point numbers.

To specify a **polyline** consisting of N line segments, one should set the attributes `degree=1`, `lastknot=N`, `knots=NULL`. The array pointed by `points` must contain `spdimen`$*(N+1)$ floating point numbers (or $N+1$ structures of type `point2f` or `vector3f`).

To specify a **Bézier curve** of degree $n > 0$, the attributes should be `degree=n`, `lastknot=-1`, `knots=NULL`. The array pointed by `points` must contain `spdimen`$*(n+1)$ floating point numbers (or $n+1$ structur of type `point2f` or `vector3f`).

To specify a **B-spline curve** of degree $n > 0$, one should set up the attributes `degree=n`, `lastknot=N`. The attribute `knots` must point an array with $N + 1$ floataing point numbers, being the knots, and `points` must point an array with `spdimen`$*(N-n)$ floating point numbers, the coordinates of the control points.

It is not required that B-spline curves be represented with clamped knots, but the curve should be connected with the neighbouring boundary elements. In particular one can specify a connected part of the boundary domain as a single closed B-spline curve.

The boundary of the domain may (but it does not have to) be oriented. One can use the convention that moving along the boundary according to their natural parameterization one has the inside of the domain on the left hand (or right hand) side. The drawing procedures must be implemented in such a way that the appropriate information is available. The boundary elements may intersect, and it must not cause execution errors.

Example. The boundary of the domain in Figure 7.20 is described as follows:

```
#define n 3
#define NNt1a 10
```

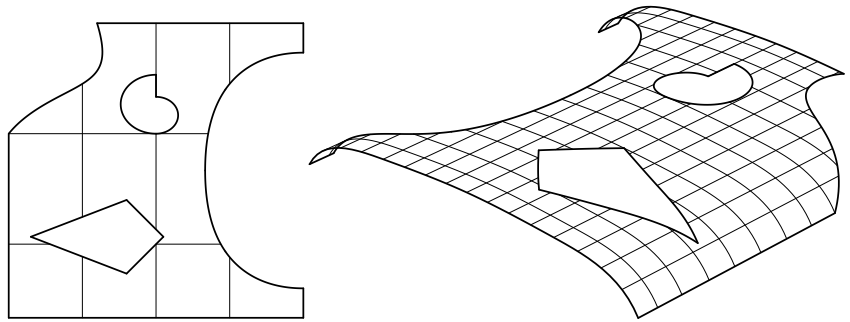



Figure 7.20. A trimmed domain and a trimmed B-spline patch.

```
float ut1a[NNt1a+1] =
  {-0.5, 0.0, 0.0, 0.0, 1.4, 2.8, 4.2, 5.6, 5.6, 5.6, 6.1};
point2f cpt1a[NNt1a-n] =
  {{4.0,0.4},{3.5,0.4},{2.8,0.8},{2.6,2.0},{2.8,3.2},{3.5,3.6},
   {4.0,3.6}};
point2f cpd1a[3] = {{4.0,3.6},{4.0,4.0},{1.2,4.0}};
point3f cpt1b[4] = {{1.2,4.0,1.0},{1.5,3.0,1.0},{0.5,2.5,0.75},
                    {0.0,2.5,1.0}};
point2f cpd1b[4] = {{0.0,2.5},{0.0,0.0},{4.0,0.0},{4.0,0.4}};
point2f cpd1c[5] = {{0.3,1.1},{1.6,1.6},{2.1,1.1},{1.6,0.6},
                    {0.3,1.1}};
point3f cpt1c[4] = {{2.0,3.3,1.0},{0.6,1.65,0.5},{0.6,1.25,0.5},
                    {2.0,2.5,1.0}};
point3f cpt1d[4] = {{2.0,2.5,1.0},{1.25,1.25,0.5},{1.25,1.5,0.5},
                    {2.0,3.0,1.0}};
point2f cpd1e[2] = {{2.0,3.0},{2.0,3.3}};
polycurvef boundary1[8] =
  {{false,2,n,NNt1a,&ut1a[0],(float*)&cpt1a[0]}, /* B-spline */
   {false,2, 1,  2, NULL,(float*)&cpd1a[0]},  /* polyline */
   {false,3, 3, -1, NULL,(float*)&cpt1b[0]},  /* Bézier curve */
   {true, 2, 1,  3, NULL,(float*)&cpd1b[0]},  /* polyline */
   {true, 2, 1,  4, NULL,(float*)&cpd1c[0]},  /* polyline */
   {false,3, 3, -1, NULL,(float*)&cpt1c[0]},  /* Bézier curve */
   {false,3, 3, -1, NULL,(float*)&cpt1d[0]},  /* Bézier curve */
   {true, 2, 1,  1, NULL,(float*)&cpd1e[0]}}; /* line segment */
```

The boundary in this example consists of four closed curves. The first is the "outer border" and it consists of four elements: a B-spline curve, a polyline of two line sements, a rational Bézier curve and a polyline of three line segments. The second curve is one closed polyline and the third curve consists of two halfcircles

(represented as rational cubic Bézier curves) and a polyline consisting of one line segment.

The **index** of a point of a plane is the number of circulations (in the counter-clockwise direction) of this point along the boundary according to the orientation of this boundary. The first two of the three closed curves above are oriented so that moving along these curves we have the inside of the domain on the left hand side. The third curve has the opposite orientation. Therefore the index of all points outside the "outer border" is 0 and the index of the points inside the polygon bounded by the second curve is 2. The domain is defined as the set of points, whose index is 1.

7.21.2 Domain boundary compilation

To draw a trimmed patch it is necessary to compute many times the common points of straight lines with the boundary of the domain. To save time the representation described in the previous section is translated into a code, which describes the polylines and Bézier curves, which form the boundary.

```
int mbs_TrimCVBoundSizef ( int nelem, const polycurvef *bound );
```

The procedure `mbs_TrimCVBoundSizef` computes the length (in bytes) of the code, which describes the boundary of the trimmed patch domain. The given boundary representation is as described in the previous section. Its parts are polylines, Bézier curves and B-spline curves, described with the elements of the array `bound` of length `nelem`.

This procedure may be used for allocation of the sufficient memory block.

```
void *mbs_CompileTrimPatchBoundf ( int nelem,
                                   const polycurvef *bound,
                                   void *buffer );
```

The procedure `mbs_CompileTrimPatchBoundf` "compiles" the boundary representation, i.e. it produces the code representing the polylines and Bézier curves (the B-spline curves are replaced with the appropriate sequences of Bézier arcs).

The parameter `nelem` specifies the length of the array `bound`, whose elements describe the domain boundary. The parameter `buffer` points to the array, in which the code is to be stored. The array must be long enough (one can compute the necessary length using `mbs_TrimCVBoundSizef`). If the parameter `buffer` is (`NULL`), then the procedure allocates a long enough memory block from the scratch memory pool (using `pkv_GetScratchMem`).

The procedure returns the pointer to the array with the code (i.e. the given value of the parameter `buffer` or the address of the allocated memory block, if `buffer` was `NULL`). It may also return `NULL`, which indicates an error (e.g. insufficient scratch memory).

7.21.3 Line pictures

A line picture of a patch or its domain consists of the curves being the images of the parts of domain boundary and of lines of constant parameters. To draw such a picture one should find these lines, i.e. find the intersections of appropriate straight lines with the domain boundary. This is done by the procedure `mbs_FindBoundLineIntersectionsf`, described below. The "higher level" procedure `mbs_DrawTrimBSPatchDomf` generates a set of straight lines and it computes their intersections with the domain. Each such an intersection is output by calling the **output procedure** given as a parameter. That procedure should draw or display in some way the line segment in the domain or its image (a piece of a curve of a constant parameter) on the patch.

```
typedef struct {
    float t;
    char  sign1, sign2;
  } signpoint1f;
```

The structure of type `signpoint1f` is used to describe an intersection point of a straight line with the domain boundary. The line is given in the parametric form, and it divides the plane into two halfplanes. The attribute `t` of the structure is set to the value of the parameter of the line corresponding to the intersection point, and the attributes `sign1` and `sign2` describe the orientation of the intersection. Their possible values are 0, -1 and $+1$, which correspond to the cases when the initial point (`sign1`) or the end point (`sign2`) of a small part of the boundary is located on the line or in one of the two halfplanes.

```
void mbs_FindBoundLineIntersectionsf ( const void *bound,
                                       const point2f *p0, float t0,
                                       const point2f* p1, float t1,
                                       signpoint1f *inters,
                                       int *ninters );
```

The procedure `mbs_FindBoundLineIntersectionsf` computes the intersection points of the straight line given by its two points, `p0` and `p1`, with the boundary of the domain of the trimmed patch, represented by the code in the array `bound` (generated by `mbs_CompileTrimPatchBoundf`). The intersection points are stored in the array `inters`. If the boundary has a common line segment with the straignt line, then it is represented in the array `inters` by two elements, corresponding to the end points of the segment. In that case the attributes `sign1` and `sign2` of the two elements are set to 0.

The numbers `t0` and `t1` are the parameters of the straight line corresponding to the points `p0` and `p1`. Both these points and their corresponding parameters must be different.

The initial value of the parameter *ninters specifies the length (capacity) of the array inters, i.e. the maximal number of the intersection points that the program expects to find. Upon exit this parameter has the value of the intersection points found. In case of error (e.g. in the code, or the overflow of the array inters), the parameter inters is assigned a negative value.

The array inters after finding all the intersection points is sorted in the order of increasing values of the attributes t.

```
void mbs_DrawTrimBSPatchDomf ( int degu, int lastuknot,
                    const float *uknots,
                    int degv, int lastvknot,
                    const float *vknots,
                    int nelem, const polycurvef *bound,
                    int nu, float au, float bu,
                    int nv, float av, float bv,
                    int maxinters,
                    void (*NotifyLine)(char,int,point2f*,point2f*),
                    void (*DrawLine)(point2f*,point2f*,int),
                    void (*DrawCurve)(int,int,const float*) );
```

The procedure mbs_DrawTrimBSPatchDomf may be used to draw a line image of the domain of a trimmed B-spline patch, or the patch itself. Its purpose is to generate a set of line segments in the domain and to pass each line segment to and output procedure. The details of further processing of the line segments (e.g. displaying on the screen or writing in a PostScript file) is thus kept away from the procedure mbs_DrawTrimBSPatchDomf.

The first 8 parameters of the procedure describe the boundary of the trimmed B-spline patch. They are: degree n of the patch with respect to the parameetr u (degu), the number N of the last knot in the knot sequence $u_0, \ldots, u_N$ (lastuknot), the array with these knots (uknots), the degree m of the patch with respect to v (degv), the number M of the last knot in the sequence $v_0, \ldots, v_M$, the array with these knots (vknots), the number of boundary elements (nelem) and the array bound, whose elements describe the domain boundary, as described in Section 7.21.1.

The six parameters that follow specify the set of lines, whose intersections with the patch boundary are to be found. The set consists of "vertical" and "horizontal" lines (lines of constant parameters u and v respectively).

The "vertical" lines correspond to the knots $u_n, \ldots, u_{N-n}$ (and thus they have nonempty intersections with the domain) and in addition to the numbers, which divide each interval $[u_i, u_{i+1}]$, $i = n, \ldots, N-n-1$, into equal subintervals. The default number of the subintervals is the value of nu, but it may be modified so that the length of each subinterval be not less than the value of the parameter au and not greater than the value of bu.

In a similar way the parameters nv, av and bv specify the set of "horizontal" lines (of constant parameter v), generated by the procedure.

The parameter maxinters is the maximal expected number of intersection points od a single line with the domain boundary. According to its value the procedure allocates a buffer for these points. In case of overflow in this buffer the procedure may fail.

The last three parameters point to the output procedures. Each of them may be NULL, which causes ignoring the appropriate result of the computations.

The first output procedure, NotifyLine, is called for each subsequent "vertical" or "horizontal" line from the set generated by the procedure. Its first parameter (of type char) is 1 if the line is vertical, and 2 if horizontal. The second parameter specifies the number of the appropriate interval between the knots, and the next two parameters are the end points of the intersection of the line with the domain of the untrimmed patch. For example, if the first parameter is 1, and the second is k, then the line is vertical, i.e. it is a line of constant parameter u, which is the number from the interval $[u_k, u_{k+1})$. This number is the value of the x coordinate of the points passed as the third and the fourth parameter.

The output procedure DrawLine is called after finding the intersections of the domain boundary with the line, for *each* pair of consecutive intersection points. The points are passed as the first two parameters. The third parameter is the index of the points inside the line segment (see Section 7.21.1). If the boundary is oriented in such a way that moving along it according to the orientation we have the inside of the domain on the left hand side, then this index will always be 1 (for the line segments inside the domain) or 0 (for the line segments outside the domain). In general the orientation of the particular closed curves, which describe the domain boundary, may be arbitrary. The decision, which line segments are inside and which are outside the domain is left to the procedure DrawLine (it may implement e.g. the parity rule: the domain contains the line segments with the index odd).

The procedure DrawCurve is called in order to draw the elements of the domain boundary. The first parameter is always $d = 2$ or 3, to distinguish between planar polynomial and rational (represented in the homoheneous form) Bézier curves. The second parameter specifies the degree n of the curve (if it is 1 then the curve is a line segment, the procedure may take advantage of that). The third parameter is an array with the control points. It contains $(n+1)d$ floating point numbers, the coordinates of the control points.

Example — output proceures for line pictures

Example procedures below were used to draw the domain and the patch shown in Figure 7.20, using PostScript.

The picture on the left hand side shows the domain of a B-spline patch, i.e. the intersections of the constant parameter lines corresponding to the knots of the patch,

and the domain boundary. The line segments are drawn with the procedure shown below. It calls some procedure MapPoint in order to map (scale and translate) the line segments.

```
void DrawLine1 ( point2f *p0, point2f *p1, int index )
{
  point2f q0, q1;
  if ( index == 1 ) {
    ps_Set_Line_Width ( 2.0 );
    MapPoint ( frame, p0, &q0 );
    MapPoint ( frame, p1, &q1 );
    ps_Draw_Line ( q0.x, q0.y, q1.x, q1.y );
  }
} /*DrawLine1*/
```

The domain boundary has been drawn by the procedure DrawCurve1, whose shortened version follows (the full version is in the file trimpatch.c):

```
void DrawCurve1 ( int dim, int degree, const float *cp )
{
#define DENS 50
  int i, size;
  float t;
  point2f *c, p;
  ps_Set_Line_Width ( 6.0 );
  if ( degree == 1 ) {
    /* A Bézier curve of degree 1 is a line segment, so this case */
    /* is treated in a special way. The array cp contains 4 */
    /* or 6 numbers, the cartesian or homogeneous coordinates */
    /* (depending on the value of dim) of the end points. */
    ...
  }
  else /* degree > 1, we draw a polyline */ {
    if ( c = pkv_GetScratchMem ( size=(DENS+1)*sizeof(point2f) ) ) {
      if ( dim == 2 ) {
        for ( i = 0; i <= DENS; i++ ) {
          t = (float)i/(float)DENS;
          mbs_BCHornerC2f ( degree, cp, t, &p );
          MapPoint ( frame, &p, &c[i] );
        }
      }
      else if ( dim == 3 ) {
        for ( i = 0; i <= DENS; i++ ) {
          t = (float)i/(float)DENS;
```



```
          mbs_BCHornerC2Rf ( degree, (point3f*)cp, t, &p );
          MapPoint ( frame, &p, &c[i] );
        }
      }
      else goto out;
      ps_Draw_Polyline ( c, DENS );
out:
      pkv_FreeScratchMem ( size );
    }
  }
#undef DENS
} /*DrawCurve1*/
```

The call of mbs_DrawTrimBSPatchDomf, which produced this picture, looks like this:

```
mbs_DrawTrimBSPatchDomf ( n1, NN1, u1, m1, MM1, v1, 8, boundary1,
                          1, 2.0, 2.0, 1, 2.0, 2.0,
                          20, NULL, DrawLine1, DrawCurve1 );
```

The first 6 parameters specify the degree and knots, and also the untrimmed domain, as described before. The domain is the rectangle $[0,4] \times [0,4]$, and the lengths of the intervals between the knots are between 1 and 1.5. Therefore the values of the parameters nu, au, bu, nv, av, bv ensure drawing only the lines of constant parameters corresponding to the knots.

To draw the picture of the trimmed patch as on the right hand side of Figure 7.20 one must map the lines in the domain onto the patch, and then to use the appropriate perspective projection. The procedure DrawLine2 used in this case is the following:

```
void DrawLine2 ( point2f *p0, point2f *p1, int index )
{
#define LGT 0.05
  void *sp;
  int i, k;
  float t, d;
  vector2f v;
  point2f q, *c;
  point3f p, r;
  if ( index == 1 ) {
    ps_Set_Line_Width ( 2.0 );
    SubtractPoints2f ( p1, p0, &v );
    d = sqrt ( DotProduct2f(&v,&v) );
    k = (int)(d/LGT+0.5);
    sp = pkv_GetScratchMemTop ();
```

```
    c = (point2f*)pkv_GetScratchMem ( (k+1)*sizeof(point2f) );
    for ( i = 0; i <= k; i++ ) {
      t = (float)i/(float)k;
      InterPoint2f ( p0, p1, t, &q );
      mbs_deBoorP3f ( n1, NN1, u1, m1, MM1, v1, 3*(MM1-m1),
                      &cp1[0][0], q.x, q.y, &p );
      PhotoPointUDf ( &CPos, &p, &r );
      c[i].x = r.x; c[i].y = r.y;
    }
    ps_Draw_Polyline ( c, k );
    pkv_SetScratchMemTop ( sp );
  }
#undef LGT
} /*DrawLine2*/
```

An explanation: instead of a curve the procedure draws a polyline. The number of its line segments depends on the length of the line segment in the domain of the patch. One can take into account also the shape of the patch, which is a bit more complicated. The procedure `DrawLine2` has an access to the representation of the patch (its knots and control points) via global variables. The points of the patch are computed using the de Boor algorithm (by the `mbs_deBoorP3f` procedure). One can decrease the computational cost, by specifying with the parameter `NotifyLine` a procedure, whose task would be to find the B-spline (or even a piecewise Bézier) representation of the curve of constant parameters u or v. The procedure specified as the parameter `DrawLine` after calling `NotifyLine` would then just draw the arcs of this curve of the constant parameter.

8. The libraybez library

The libraybez library consists of procedures, whose main (but not only) purpose is supporting ray tracing, and more precisely computing intersections of rays with Bézier patches. To do this, the procedures build trees of recursive patch subdivision, to accelerate solving the appropriate nonlinear equations, by eliminating multiple computations of the same thing.

Possible extensions of this library should include constructing trees for B-spline patches (also trimmed) and trees with additional attributes, which would help in computing intersections of surfaces.

8.1 Common definitions and procedures

```
typedef struct {
  float xmin, xmax, ymin, ymax, zmin, zmax;
} Box3f;
```

The structure Box3f represents a rectangular parallelpiped. In the representation of a piece of a patch it is used to locate this piece in the space (the piece is in the associated parallelpiped).

```
typedef struct {
  point3f  p;
  vector3f nv;
  float    u, v, t;
} RayObjectIntersf, *RayObjectIntersfp;
```

The structure RayObjecIntersf represents a point of intersection of a ray with a patch. It consists of the following fields: p — the point of intersection, nv — normal vector of the patch at this point, u, v, t — parameters pf the patch and the ray corresponding to the intersection point.

8.2 Binary subdivision trees for polynomial patches

```
typedef struct _BezPatchTreeVertexf {
  struct _BezPatchTreeVertexf
           *left, *right, *up;
  point3f   *ctlpoints;
  float     u0, u1, v0, v1;
  Box3f     bbox;
  point3f   pcent;
  float     maxder;
  short int level;
  char      divdir;
  char      pad;
} BezPatchTreeVertexf, *BezPatchTreeVertexfp;
```

The structure _BezPatchTreeVertexf represents a vertex of a binary tree of recursive subdivision of a polynomial Bézier patch $\mathbf{p}$.

The fields of this structure are used to store the following data: left, right, up — pointers to the vertices of the left and right subtrees and to the parent vertex (the vertex, whose the current vertex is the root of one of the subtrees) respectively, ctlpoints — pointer to the array with the control points of the piece represented by this vertex, u0, u1, v0, v1 — numbers which describe the domain of the piece, $[u_0, u_1] \times [v_0, v_1]$, bbox — bounding box (rectangular parallelpiped) of the piece, pcent — the point $\mathbf{p}((u_0+u_1)/2, (v_0+v_1)/2)$, maxder — upper estimation of the length of the vectors of both partial derivatives of this piece with respect to local parameters, level — level of the vertex in the tree, divdir — indicator of the direction of further division of the piece, pad — unused (it aligns the size of the structure to an even number of bytes).

```
typedef struct {
  unsigned char         n, m;
  unsigned int          cpsize;
  BezPatchTreeVertexfp root;
} BezPatchTreef, *BezPatchTreefp;
```

The structure BezPatchTreef represents a tree of recursive binary subdivision of a polynomial Bézier patch. Its fields are the following: n, m — degree of the patch with respect to the variables u and v, cpsize — amount of memory needed to store the control points, root — pointer to the root of the tree.

```
BezPatchTreefp
  rbez_NewBezPatchTreef ( unsigned char n, unsigned char m,
                          float u0, float u1, float v0, float v1,
                          point3f *ctlpoints );
```

The procedure `rbez_NewBezPatchTreef` creates a tree of binary subdivision of a polynomial Bézier patch and it returns the pointer to the structure, which represents this tree. Initially the tree consists only of the root, which represents the entire patch.

The parameter `n` and `m` specify the degree of the patch with respect to u and v respectively. The parameters `u0`, `u1`, `v0` and `v1` specify the domain of the patch, i.e. the rectangle $[u_0, u_1] \times [v_0, v_1]$ (if the patch has been obtained by dividing a B-spline patch, then these numbers should be the appropriate knots).

The parameter `ctlpoints` points to the array of control points of the patch.

The value returned by the procedure is the pointer to the structure, which describes the tree. The memory blocks for this structure and for the structures representing the vertices are allocated with `malloc`.

```
void rbez_DestroyBezPatchTreef ( BezPatchTreefp tree );
```

The procedure `rbez_DestroyBezPatchTreef` deallocates (by calling `free`) the memory blocks used to represent a tree of binary patch division. The parameter `tree` is a pointer to the structure representing the tree.

```
BezPatchTreeVertexp
  rbez_GetBezLeftVertexf ( BezPatchTreefp tree,
                           BezPatchTreeVertexfp vertex );
BezPatchTreeVertexfp
  rbez_GetBezRightVertexf ( BezPatchTreefp tree,
                            BezPatchTreeVertexfp vertex );
```

The procedures `rbez_GetBezLeftVertexf` and `rbez_GetBezRightVertexf` return pointers to the vertex of the left and right subtree of a vertex of a tree of patch subdivision respectively.

The parameters: `tree` — pointer to the structure representing the tree, `vertex` — pointer to one of the vertices of this tree.

The procedures return the pointers to the appropriate (left or right) vertices. If it does not exist, then the procedures divide the piece of the patch represented by the vertex pointed by `vertex`, they create both root vertices of the subtrees and they return the pointer to one of those new vertices. For each vertex, either both subtrees exist or both are empty.


```
int rbez_FindRayBezPatchIntersf ( BezPatchTreef *tree,
                                  ray3f *ray,
                                  int maxlevel, int maxinters,
                                  int *ninters, RayObjectIntersf *inters );
```

The procedure `FindRayBezPatchIntersf` computes the common points of a ray (a halfline) with a polynomial Bézier patch in $\mathbb{R}^3$.

The parameters: `tree` — pointer to the tree of patch subdivision; `ray` — pointer to the ray (the structure `ray3f` is defined in the header file `geomf.h`; `maxlevel` — limit of the height of the tree (the procedure will not require vertices beyond that level, therefore they will not be created if the only reason to call the procedures returning pointers to the vertices is the ray tracing); `maxinters` — capacity of the array `inters`, in which the results are to be stored. The array must have at least that length, and the procedure will terminate after computing at most that many intersections. The number of intersection points found on return is assigned to `*ninters`.

The number of intersection points is also the value of the procedure.

8.3 Binary subdivision trees for rational Bézier patches

Binary trees of recursive subdivision of rational Bézier patches are constructed and processed in an almost identical way as the trees for the polynomial patches. All structures and procedures described in the previous section have their counterparts here.

```
typedef struct _RBezPatchTreeVertexf {
  struct _RBezPatchTreeVertexf
            *left, *right, *up;
  point4f   *ctlpoints;
  float     u0, u1, v0, v1;
  Box3f     bbox;
  point3f   pcent;
  float     maxder;
  short int level;
  char      divdir;
  char      pad;
} RBezPatchTreeVertexf, *RBezPatchTreeVertexfp;
```

The structure `_BezPatchTreeVertexf` represents a vertex of a binary tree of recursive subdivision of a rational Bézier patch $\mathbf{p}$.

The fields of this structure are used to store the following data: `left`, `right`, `up` — pointers to the vertices of the left and right subtrees and to the parent vertex (the vertex, whose the current vertex is the root of one of the subtrees) respectively, `ctlpoints` — pointer to the array with the control points of the homogeneous patch

representing the piece corresponding to this vertex, u0, u1, v0, v1 — numbers which describe the domain of the piece, $[u_0, u_1] \times [v_0, v_1]$, bbox — bounding box (rectangular parallelpiped) of the piece, pcent — the point $\mathbf{p}((u_0 + u_1)/2, (v_0 + v_1)/2)$, maxder — upper estimation of the length of the vectors of both partial derivatives of this piece with respect to local parameters, level — level of the vertex in the tree, divdir — indicator of the direction of further division of the piece, pad — unused (it aligns the size of the structure to an even number of bytes).

```
typedef struct {
  unsigned char         n, m;
  unsigned int          cpsize;
  RBezPatchTreeVertexfp root;
} RBezPatchTreef, *RBezPatchTreefp;
```

The structure RBezPatchTreef representuje a binary tree of recursive subdivision of a rational Bézier patch. Its fields are as follows: n, m — degrees of the patch with respect to the cariables u and v, cpsize — amount of memory needed to store the control points, root — pointer to the root of the tree.

```
RBezPatchTreefp
  rbez_NewRBezPatchTreef ( unsigned char n, unsigned char m,
                           float u0, float u1, float v0, float v1,
                           point4f *ctlpoints );
```

The procedure rbez_NewRBezPatchTreef creates a tree of binary subdivision of a rational Bézier patch and it returns the pointer to the structure, which represents this tree. Initially the tree consists only of the root, which represents the entire patch.

The parameter n and m specify the degree of the patch with respect to u and v respectively. The parameters u0, u1, v0 and v1 specify the domain of the patch, i.e. the rectangle $[u_0, u_1] \times [v_0, v_1]$ (if the patch has been obtained by dividing a NURBS patch, then these numbers should be the appropriate knots).

The parameter ctlpoints points to the array of control points of the homogeneous patch in $\mathbb{R}^4$.

The value returned by the procedure is the pointer to the structure, which describes the tree. The memory blocks for this structure and for the structures representing the vertices are allocated with malloc.

```
void rbez_DestroyRBezPatchTreef ( RBezPatchTreefp tree );
```

The procedure rbez_DestroyRBezPatchTreef deallocates (by calling free) the memory blocks used to represent a tree of binary patch division. The parameter tree is a pointer to the structure representing the tree.


```
RBezPatchTreeVertexp
  rbez_GetRBezLeftVertexf ( RBezPatchTreefp tree,
                            RBezPatchTreeVertexfp vertex );
RBezPatchTreeVertexfp
  rbez_GetRBezRightVertexf ( RBezPatchTreefp tree,
                             RBezPatchTreeVertexfp vertex );
```

The procedures rbez_GetRBezLeftVertexf and rbez_GetRBezRightVertexf return pointers to the vertex of the left and right subtree of a vertex of a tree of patch subdivision respectively.

The parameters: tree — pointer to the structure representing the tree, vertex — pointer to one of the vertices of this tree.

The procedures return the pointers to the appropriate (left or right) vertices. If it does not exist, then the procedures divide the piece of the patch represented by the vertex pointed by vertex, they create both root vertices of the subtrees and they return the pointer to one of those new vertices. For each vertex, either both subtrees exist or both are empty.

```
int rbez_FindRayRBezPatchIntersf ( RBezPatchTreef *tree,
                                   ray3f *ray,
                                   int maxlevel, int maxinters,
                                   int *ninters, RayObjectIntersf *inters );
```

The procedure FindRayRBezPatchIntersf computes the common points of a ray (a halfline) with a rational Bézier patch in $\mathbb{R}^3$.

The parameters: tree — pointer to the tree of patch subdivision; ray — pointer to the ray (the structure ray3f is defined in the header file geomf.h; maxlevel — limit of the height of the tree (the procedure will not require vertices beyond that level, therefore they will not be created if the only reason to call the procedures returning pointers to the vertices is the ray tracing); maxinters — capacity of the array inters, in which the results are to be stored. The array must have at least that length, and the procedure will terminate after computing at most that many intersections. The number of intersection points found on return is assigned to *ninters.

The number of intersection points is also the value of the procedure.

Figure 8.1 shows an image of a rational Bézier patch of degree $(5, 5)$ obtained with use of this procedure. The full source code of the program which rendered this image is given in the file ../cpict/raybez.c.

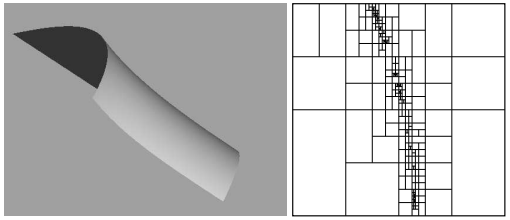


Figure 8.1. Image of a rational Bézier patch rendered by ray tracing with use of the procedure `rbez_FindRayRBezPatchIntersf`. On the right side the pieces of the domain divided during the computation are shown

9. The libeghole library

The libeghole library contains procedures of filling polygonal holes in a piecewise bicubic spline surfaces. The theory and the algorithm are described in the book *Konstrukcje powierzchni gładko wypełniających wielokątne otwory.*

9.1 Data preparation

The data for hole filling procedures consists of four elements:

- The number of hole sides, k,
- k eleven-element sequences of knots,
- $12k+1$ domain control points,
- $12k+1$ surface control points.

In addition to the data mentioned above, one can specify **constraints**, i.e. linear equations to be satisfied by the filleting surface. Their representation is described in Section 9.3.4.

The integer number k must not be less than 3 and not greater than 16. The surface constsis of 3k bicubic polynomial patches, which surround a k-sided hole.

The **knot sequences** $u_0^{(n)},\dots,u_{10}^{(n)}$, for $n=0,\dots,k-1$, must satisfy the conditions

$$u_0^{(n)} \le u_1^{(n)} < \cdots < u_9^{(n)} \le u_{10}^{(n)},$$

and

$$u_i^{(n)} - u_1^{(n)} = u_0^{(m)} - u_{10-i}^{(m)},$$

for $m=(n+2) \bmod k$ and $i=4,\dots,9$. The sequences should be given in a one-dimensional array of length 11k; the subsequent elements of the sequences must be stored without gaps.

The **domain control points**, $\mathbf{c}_0,\dots,\mathbf{c}_{12k}$ are points of a plane; they are vertices of the domain control net. A scheme of such a net and the method of numbering them is shown in Figure 9.1. This net contains k control nets of planar bicubic B-spline patches, which have common polynomial pieces.

For $n=0,\dots,k-1$ the n-th B-spline patch is represented by the knot sequences $u_0^{(n)},\dots,u_{10}^{(n)}$ and $u_0^{(m)},\dots,u_7^{(m)}$, where $m=(n+1) \bmod k$, and the control points $\mathbf{c}_{ij}^{(n)}$, $i=0,\dots,6$, $j=0,\dots,3$, such that:

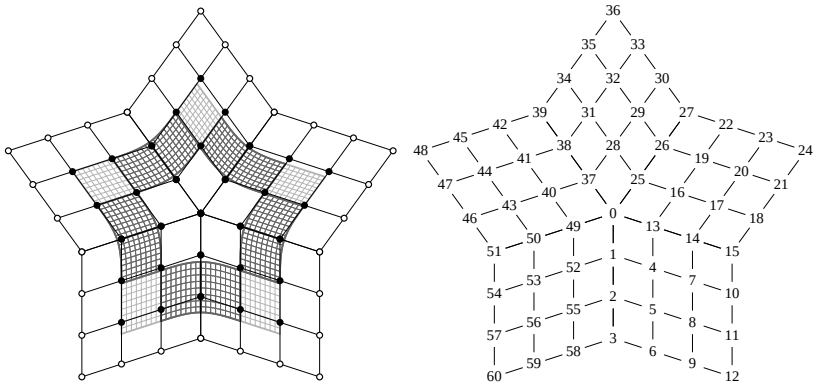


Figure 9.1. Representation of a domain of a surface filling a hole

- $\mathbf{c}_{ij}^{(n)} = \mathbf{c}_{12(n+1)-3j-i}$ for $i=0,\dots,2$, $j=0,\dots,3$
- $\mathbf{c}_{ij}^{(n)} = \mathbf{c}_{12m-3i-j}$ for $i=3,\dots,6$, $j=0,\dots,2$, where $m=(n+1) \bmod k$,
- $\mathbf{c}_{3,3}^{(n)} = \mathbf{c}_0$,
- $\mathbf{c}_{ij}^{(n)} = \mathbf{c}_{12m+i-3}$ for $i=4,\dots,6$, $j=3$, where $m=(n+2) \bmod k$.

The B-spline patches represented by these knots and control points must be regular and apart from the common pieces determined by the representation they must be disjoint.

The set of points of the B-spline patches represented by the knots and control points described above is the domain of some parameterization of the surface with the hole, and the area Ω surrounded by these patches, which is a curvilinear k-sided polygon, is the domain of a parameterization of the filleting surface to be constructed. For such a parameterization a functional F is defined; its value measures the quality (or rather *badness*) of the surface (the minimum will be searched). By changing the control points $\mathbf{c}_i$ one changes this functional, which affects the construction result.

The **surface control points**, $\mathbf{b}_0,\dots,\mathbf{b}_{12k}$, are located in the space of dimension d (usually in practice it will be $d=3$ for polynomial surfaces, or $d=4$, if the hole to be filled is in a homogeneous surface representing a piecewise rational surface). The surface control net is bulit in a similar way to the domain control net, i.e. one can distinguish in it k control nets of B-spline patches of degree $(3,3)$. For $n=0,\dots,k-1$ the n-th patch is represented by the knots $u_0^{(n)},\dots,u_{10}^{(n)}$ and $u_0^{(m)},\dots,u_7^{(m)}$, where $m=(n+1) \bmod k$, and by the control points $\mathbf{b}_{ij}^{(n)}$, $i=0,\dots,6$, $j=0,\dots,3$, being the points $\mathbf{b}_l$ with the subscripts l defined in the same way as the indexes of the control points $\mathbf{c}_{ij}^{(n)}$ of the B-spline patches surrounding the domain.

The array with the control points of the surface, which is to be passed to the procedures filling the hole, consists of $(12k+1)d$ floating point numbers — consecutive d numbers are the coordinates of one control point $\mathbf{b}_l$. The points $\mathbf{c}_{ij}^{(n)}$ and $\mathbf{b}_{ij}^{(n)}$ for $i \in \{0,6\}$ and for $j = 0$ have no influence on the construction result (i.e. on the filleting surface), as well as the knots $u_0^{(n)}$ and $u_{10}^{(n)}$, but they must be specified. In Figure 9.1 the points, which are relevant, are marked with black dots.

9.2 Theoretical background

A detailed description of theory underlying the constructions done by the procedures of the `libeghole` library may be found in the paper *Konstrukcje powierzchni gładko wypełniających wielokątne otwory* (*Constructions of surfaces filling smoothly polygonal holes*, in Polish). Below only the information necessary for correct preparation of data for the procedures are given.

9.2.1 Bases used in the constructions

To construct a surface filling a hole, the library procedures construct a basis $\phi_0, \ldots, \phi_{n+m}$ of some linear vector space, whose elements are scalar functions of class C^1 or C^2, which describe the coordinates of the hole filling surface. The surface may be described by the formula

$$\mathbf{p} = \sum_{i=0}^{n-1} \mathbf{a}_i \phi_i + \sum_{i=0}^{m-1} \mathbf{b}_i \phi_{n+i}. \tag{9.1}$$

The vectors $\mathbf{b}_i \in \mathbb{R}^d$ are given control points of the surface with the hole. The control net of this surface is a graph isomorphic with the control net of the domain shown in Figure 9.1 and its vertices are numbered analoguously. The vertices are passed to the construction procedures (in that order) in an array given as a parameter. The array contains $12k+1$ control points, of which $m = 6k+1$ influence the surface filling the hole.

The purpose of the surface construction procedures is to compute the vectors $\mathbf{a}_0, \ldots, \mathbf{a}_{n-1} \in \mathbb{R}^d$, which minimise some functionals, taken as measures of „badness" of the surface. The basis functions $\phi_0, \ldots, \phi_{n+m-1}$ are defined in the area $\Omega \in \mathbb{R}^2$, which is the hole in the planar surface represented by the domain control net, described in the previous section. The area Ω is divided into k curvilinear quadrangles $\Omega_0, \ldots, \Omega_{k-1}$, which are images of the unit square under the mappings $\mathbf{d}_0, \ldots, \mathbf{d}_{k-1}$, called **domain patches**. The function ϕ_i is defined by the formula

$$\phi_i(\mathbf{x}) = p_{li}(\mathbf{d}_l^{-1}(\mathbf{x})) \qquad \text{for } \mathbf{x} \in \Omega_i,$$

with use of the domain patches and the functions $p_{i0}, \ldots, p_{i,k-1}$, called **basis function patches**. The surface filling the hole consists of k polynomial or spline patches $\mathbf{p}_0, \ldots, \mathbf{p}_{k-1}$, given by

$$\mathbf{p}_l = \sum_{i=0}^{n-1} \mathbf{a}_i p_{li} + \sum_{i=0}^{m-1} \mathbf{b}_i p_{l,i+n}.$$

The Bézier or B-spline representation of these patches is the final result of the constructions.

The basis functions $\phi_0, \ldots, \phi_{n+m-1}$ may be divided into two subsets. The functions $\phi_0, \ldots, \phi_n$ satisfy the homogeneous boundary condition, i.e. their values and partial derivatives of order 1 (or 1 and 2) at the boundary of Ω are 0. The functions $\phi_n, \ldots, \phi_{n+m-1}$ satisfy the boundary conditions chosen in such a way, that for arbitrary vectors $\mathbf{a}_0, \ldots, \mathbf{a}_{n-1}$ the surface given by Formula (9.1) was joined with the given surface with tangent plane or curvature continuity. The functions $\phi_n, \ldots, \phi_{n+m-1}$ and their derivatives of order 1 and 2 (or $1, \ldots, 4$) at the **central point**, i.e. the comon point of all areas Ω_l are equal to 0.

The halflines tangent to the common curves of the areas $\Omega_0, \ldots, \Omega_{k-1}$ are inclined at the angles $\alpha_0, \ldots, \alpha_{k-1}$, where $\alpha_0 < \alpha_1 < \cdots < \alpha_{k-1} < \alpha_0 + 2\pi$. The set $\Delta = \{\alpha_0, \ldots, \alpha_{k-1}\}$ is called the **partition of the full angle**. Let h denote the number of pairs $\{\alpha_i, \alpha_i + \pi\} \subset \Delta$. For the case of construction of surfaces of class G^1 or G^1Q^2, let

$$n' = 3 + \max\{k, h+3\}.$$

The number n' is the number of elements of the **basic basis**, whose all elements have basis function patches being bicubic Coons patches, described by polynomials of degree 5 (thus they are polynomials of degree $(5,5)$).

One can take $n = n'$ or $n = n' + 4k$; in the former case we have so called **extended basis** of the correspondingly wider space $V_0 = \operatorname{lin}\{\phi_0, \ldots, \phi_{n-1}\} \subset V$. The basis function patches of the additional basis functions are tensor products of the Bernstein polynomials B_2^5 and B_3^5. Just like in the case of using the basic basis, the result of the construction consists of k Bézier patches of degree $(5,5)$.

There is also a possibility of filling the hole with B-spline patches of degree $(5,5)$. The basis of the appropriate space V_0, apart from the elements of the basic basis contains two subsets of functions: the functions of first subset have basis function patches being tensor products of the B-spline functions N_i^5 and N_j^5 for $i, j \in \{2, \ldots, 3 + n_k m_2\}$. There must be $1 \le m_2 \le 4$. The basis function patches of the functions from the second subset are bicubic Coons patches, determined by quintic spline functions and having $n_k m_1$ knots, where $1 \le m_1 \le 2$. The dimension of the space V_0 is then equal to $n' + k\big((2 + n_k m_2)^2 + 2 n_k m_1\big)$. The result of the construction consists of k B-spline patches of degree $(5,5)$.

For the constructions of surfaces of class G^2, let

$$n' = 6 + \max\{k, h+4\} + \max\{2k, 2h+5\}.$$

If $n = n'$, then we have the basic basis, whose all elements have basis function patches being biquintic Coons patches of degree $(9,9)$.

The extended basis has additional 16k functions, whose basis function patches are tensor products of the Bernstein polynomials $B^9_3, \ldots, By_6$. In both cases the result of the construction consists of k polynomial patches of degree $(9,9)$, represented in Bézier form.

One can use also a spline basis, defined with three parameters, n_k, m_1 i m_2; in this case the space V_0 has the dimension

$$n' + k\big((4 + n_k m_2)^2 + 3n_k m_1\big),$$

and the result of the construction consists of k B-spline patches of degree $(9,9)$. There must be $1 \le m_1 \le 3$, $1 \le m_2 \le 7$.

9.2.2 Optimisation criteria for surfaces of class G^1

The filling surfaces of class G^1 have degree $(5,5)$. There are obtained by minimisation of the folowing functionals

$$F_a(\mathbf{p}) \stackrel{\text{def}}{=} \int_\Omega \|\Delta\mathbf{p}\|_2^2 \,\mathrm{d}\Omega,$$
$$F_b(\mathbf{p}) \stackrel{\text{def}}{=} \int_\Omega H^2 \sqrt{\det G}\,\mathrm{d}\Omega,$$

where G denotes the matrix of the first fundamental form and H denotes the mean curvature. The functional F_a is a quadratic form, with a unique minimum (also for arbitrary consistent constraints). The functional F_b is nonlinear, and its value depends only on the shape of the surface. The minimisation of F_b is more troublesome and time consuming. It may fail for some surfaces.

9.2.3 Optimisation criteria for surfaces of class G^2

The vectors $\mathbf{a}_0, \ldots, \mathbf{a}_{n-1}$ are chosen so as to minimise one of the following functionals:

$$F_c(\mathbf{p}) = \int_\Omega \|\nabla\Delta\mathbf{p}\|_F^2 \,\mathrm{d}\Omega,$$
$$F_d(\mathbf{p}) = \int_\Omega \|\nabla_{\mathcal{M}} H\|_2^2 \sqrt{\det G}\,\mathrm{d}\Omega.$$

Subsequent rows of the matrix $\nabla\Delta\mathbf{p}$ are gradients of Laplacians of the d scalar functions, which describe the parameterisation $\mathbf{p}$; the symbol $\|\cdot\|_F$ denotes the Frobenius norm, i.e. square root of the sum of squares of all coefficients of the matrix.

The functional F_c is defined for surfaces in d-dimensional spaces for any d, while in case of F_d there must be $d = 3$. The symbol H denotes the mean curvature of the

surface, $\nabla_{\mathcal{M}} H$ denotes the mean curvature gradient *on the surface*, and G denotes the matrix of the first fundamental form.

The functional F_c is a quadratic form, whose minimum may be found by solving a system of linear equations

$$A\mathbf{a} = -B\mathbf{b}, \tag{9.2}$$

with the matrices $A = [a_{ij}]_{i,j}$ and $B = [b_{ij}]_{i,j}$, having dimensions $n \times n$ and $n \times m$, whose coefficients

$$a_{ij} = a(\phi_i, \phi_j), \qquad b_{ij} = a(\phi_i, \phi_{j+n}),$$

are the values of the bilinear form

$$a(f, g) = \int_\Omega \langle \nabla\Delta f, \nabla\Delta g\rangle \,\mathrm{d}\Omega.$$

The matrix $\mathbf{b}$, $m \times d$ consists of the control points of the given surface, the matrix $\mathbf{a}$, $n \times d$ consists of the unknown vectors $\mathbf{a}_0, \ldots, \mathbf{a}_{n-1}$. The number d is the dimension of the space, in which the surface is located, e.g. 3 (but it may also be 4 if a piecewise polynomial surface, being a homogeneous representation of a rational surface, is constructed).

The value of the functional F_d does not depend on the parameterisation of the surface (which must be in $\mathbb{R}^3$), it depends only on the shape. Finding its minimum is more difficult, more time consuming and not always possible (this depends on the given surface with the hole). To do it, the following nonlinear system of equations is solved

$$\nabla F(a_0, \ldots, a_{n-1}) = 0, \tag{9.3}$$

where the function F is given by

$$F(a_0, \ldots, a_{n-1}) = F_d(\mathbf{p}),$$

for a parameterisation $\mathbf{p}$ defined as

$$\mathbf{p}(u, v) = \begin{bmatrix} u \\ v \\ p(u, v) \end{bmatrix}, \qquad p(u, v) = \sum_{i=0}^{n-1} a_i \phi_i + \sum_{i=0}^{m-1} b_i \phi_{n+i}.$$

The surface with the hole is transformed to such a coordinate system uvw, that it is the graph of a scalar function, $w = q(u, v)$. The domain Ω is obtained by projecting the surface onto the plane uv. The numbers $b_0, \ldots, b_{m-1}$ are the coordinates w of the control points of the given surface.

9.2.4 Optimisation criteria for surfaces of class G^1Q^2

9.2.5 Constraint equations

The constructions make it possible to impose constraints described by linear equations, e.g. interpolation conditions. The minimua of the functional F_c or F_d may be searched in the set of surfaces, whose coefficients satisfy the system of equations

$$C\mathbf{a} = \mathbf{d}. \tag{9.4}$$

The $w \times n$ matrix C must have linearly indeopendent rows. The matrix $\mathbf{d}$, whose dimensions are $\boldsymbol{w} \times d$, describes the right hand side of the constraint equations system, where w is the number of constraints and d is the dimension of the space containing the surface; for the functional F_d there must be $d = 3$.

The subsequent rows of the unknown matrix $\mathbf{a}$ are the vectors $\mathbf{a}_0, \ldots, \mathbf{a}_{n-1}$, which appear in Formula (9.1). If the i-th constraint has the form $\mathbf{p}(\boldsymbol{x}) = \mathbf{p}_0$ (this is an interpolation condition, which fixes the point of the surface corresponding to some point $\boldsymbol{x} \in \Omega$), then the coefficients of the i-th row of the matrix C are $\phi_0(\boldsymbol{x}), \ldots, \phi_{n-1}(\boldsymbol{x})$, and the i-th row of the matrix $\mathbf{d}$ must be $\mathbf{p}_0 - \sum_{i=0}^{m-1} \mathbf{b}_i \phi_{n+i}(\boldsymbol{x})$. Similarly, fixing the value $\boldsymbol{v}$ of the partial derivative with respect to e.g. u at the point $\boldsymbol{x}$ is done by the constraint equation with the row of C consisting of the numbers $\frac{\partial}{\partial u}\phi_0(\boldsymbol{x}), \ldots, \frac{\partial}{\partial u}\phi_{n-1}(\boldsymbol{x})$, and the right hand side (i.e. the row of $\mathbf{d}$) is $\boldsymbol{v} - \sum_{i=0}^{m-1} \mathbf{b}_i \frac{\partial}{\partial u}\phi_{n+i}(\boldsymbol{x})$.

For the functional F_d, if the extended basis is used, it is possible only to impose constraints in the form of interpolation conditions at the central point of the domain (i.e. at the common point of all the areas Ω_i). The library is equipped with the procedures of computing the basis functions and their partial derivatives at this point. There are also procedures giving access to the full representation of the basis functions, which make it possible to evaluate any linear functional for all the basis functions. These values may then be used as the coefficients in the constraint equations.

The method described above imposes constraints of the same kind simultaneously and independently on all the coordinates of the surface filling the hole. An alternative form of the constraint equations is the following:

$$C_0\mathbf{a}_0 + \cdots + C_{d-1}\mathbf{a}_{d-1} = \mathbf{d}. \tag{9.5}$$

The matrices $C_0, \ldots, C_{d-1}$ have dimensions $w \times n$, the matrix $C = [C_0, \ldots, C_{d-1}]$ ($w \times nd$) must have linearly independent rows. This form of constraints is more general and it allows one to fix the value of an arbitrary linear functional for the parameterisation $\mathbf{p}$. One can e.g. fix only the value of the first coordinate of the point $\mathbf{p}(\boldsymbol{x})$, by taking the i-th row of the matrix C_0 made of the coefficients $\phi_0(\boldsymbol{x}), \ldots, \phi_{n-1}(\boldsymbol{x})$, and putting zeroes in the i-th row of $C_1, \ldots, C_{d-1}$.

9.2.6 Table of procedures of surface construction

The available procedures of surface construction are gathered in the following table:

		Coons	Bézier	B-spline	Coons	Bézier	B-spline	Coons	Bézier	B-spline
L	G^1	1.	2.	3.	4.	5.		7.	8.	
L	G^2	10.	11.	12.	13.	14.		16.	17.	
L	G^1Q^2	19.	20.	21.	22.	23.		25.	26.	
NL	G^1	28.	29.	30.	31.	32.		34.	35.	
NL	G^2	37.	38.	39.	40.	41.		43.	44.	
NL	G^1Q^2	46.	47.	48.						
		without constraints			constraints (9.4)			constraints (9.5)		

The procedures indicated in the first three rows construct the minimal surfaces of quadratic forms (by solving systems of linear equations).

The procedures in next three rows solve nonlinear equations in order to minimise the functionals independent of the parameterisation.

The procedures in the first three columns construct surfaces without constraints. The following three columns contain the procedures of constructions with constraints of the form (9.4), and the last three columns with the constraints (9.5).

At the top of each column there is the form of the basis used in the constructions. The procedure names are the following:

1. `g1h_FillHolef`.
2. `g1h_ExtFillHolef`.
3. `g1h_SplFillHolef`.
4. `g1h_FillHoleConstrf`.
5. `g1h_ExtFillHoleConstrf`.
7. `g1h_FillHoleAltConstrf`.
8. `g1h_ExtFillHoleAltConstrf`.
10. `g2h_FillHolef`.
11. `g2h_ExtFillHolef`.
12. `g2h_SplFillHolef`.
13. `g2h_FillHoleConstrf`.
14. `g2h_ExtFillHoleConstrf`.

16. g2h_FillHoleAltConstrf.
17. g2h_ExtFillHoleAltConstrf.
19. g1h_Q2FillHolef.
20. g1h_Q2ExtFillHolef.
21. g1h_Q2SplFillHolef.
22. g1h_Q2FillHoleConstrf.
23. g1h_Q2ExtFillHoleConstrf.
25. g1h_Q2FillHoleAltConstrf.
26. g1h_Q2ExtFillHoleAltConstrf.
28. g1h_NLFillHolef.
29. g1h_NLExtFillHolef.
30. g1h_NLSplFillHolef.
31. g1h_NLFillHoleConstrf.
32. g1h_NLExtFillHoleConstrf.
34. g1h_NLFillHoleAltConstrf.
35. g1h_NLExtFillHoleAltConstrf.
37. g2h_NLFillHolef.
38. g2h_NLExtFillHolef.
39. g2h_NLSplFillHolef.
40. g2h_NLFillHoleConstrf.
41. g2h_NLExtFillHoleConstrf.
43. g2h_NLFillHoleAltConstrf.
44. g2h_NLExtFillHoleAltConstrf.
46. g1h_Q2NLFillHolef.
47. g1h_Q2NLExtFillHolef.
48. g1h_Q2NLSplFillHolef.

9.3 Using the procedures

9.3.1 The basic construction

The construction of a filleting surface consists of two main parts. The first part is the construction of a basis of a linear space V, whose elements are functions, which describe a parameterization of the surface (i.e. each of its coordinates), and the computation of the matrices, which appear in the systems of linear equations solved in the second part. This computation is rather time-consuming, but the only data needed in this part represent the domain.

In the second part, based on the surface control points and perhaps constraints (in case they have been imposed), the right-hand side of the system of equation is computed. The system is then solved and the solution is used to obtain k Bézier patches of degree $(9, 9)$, which fill the hole in the surface. The second part takes much less time and in practice it may be repeated many times while the user of an interactive program manipulates with the surface control points or with the constraints (however, a modification of knots causes the necessity of repeating the first part of the construction).

The first part of the construction will be done by executing the following instructions:

```
GHoleDomainf *domain;
...
if ( !(domain = gh_CreateDomainf ( k, knots, domain_cp )) )
  exit ( 1 );
if ( !g2h_ComputeBasisf ( domain ) )
  exit ( 1 );
if ( !g2h_DecomposeMatrixf ( domain ) )
  exit ( 1 );
```

The parameter k specifies the number of sides of the hole, and the arrays knots and domain_cp contain respectively the knot sequences and the domain control points. The procedure gh_CreateDomainf allocates a data structure, used to store the representation of the domain Ω of the filleting surface parameterization, the way of decomposing it into parts (k curvilinear quadrangles) and the basis of the space V. This data structure is called the **domain record**.

After creating the domain record (before calling g2h_ComputeBasisf), the program may call the procedure g2h_SetOptionProcf in order to use construction options other than default. The procedure g2h_ComputeBasisf computes representations of functions being elements of the space V (the options affect the result of this computation), and this computation takes a moderate amount of time.

The procedure g2h_DecomposeMatrixf computes the coefficients of the matrices A and B, which appear in the system of equations (9.2). These coefficients are values of the bilinear form in the space V for pairs of the basis functions. The vector $\mathbf{b}$

consists of the surface control points (which will be introduced in the second part of the construction), and the unknown vector $\mathbf{a}$ consists of the other parameters of the filleting surface representation used in the construction. The matrix A, which is symmetric and positive-definite, is then decomposed (with the Cholesky method) into the triangular factors: $A = LL^T$. These factors will be used to solve the system. Computing the coefficients of the matrices A and B is the most time-consuming step of the construction — for $k = 8$ a Pentium IV processor with a 1.8GHz clock may spend on it about 0.15s.

The second part of the construction is done by executing the code

```
if ( !g2h_FillHolef ( domain, d, surf_cp, acoeff, output ) )
  exit ( 1 );
```

The parameter `domain` points to the domain record, for which the first part of the construction has been (successfully) completed. The parameter `d` specifies the dimension of the space, in which the surface is located, the array `surf_cp` contains the control points of the surface and the parameter `output` is a pointer to the procedure, which will be called k times. Each call is made with the parameters, which describe one Bézier patch, being part of the surface filling the hole.

The parameter `acoeff` is an array, in which the soultion of the system (9.2) should be stored. This solution is necessary, if someone is interested in the value of the functional F for the constructed surface (actually, in the sum of values of the functional for the functions, which describe the surface coordinates). This parameter may be `NULL` and then it is ignored.

9.3.2 The nonlinear construction

To obtain the filling surface, being the minimal point of the functional F_d, one should create the domain representation, construct the basis and compute the coefficients of the matrices A and B, and then call the procedure `g2h_NLFillHolef` instead of `g2h_FillHolef`.

The computations done by this procedure are much more time consuming, moreover the feasibility of this construction depends on the given surface with the hole. If this surface is not flat enough or it has singularities, the construction may fail.

9.3.3 Extending the space

The patches filling the hole, obtained in the way described above, are defined as biquintic Coons patches, whose boundary curves and cross derivatives have degree up to 9. The Coons representation used internally is eventually converted to the Bézier form. As the dimension of the space of bivariate polynomials of degree $(9, 9)$ is 100 and the polynomials representable in the biquintic Coons form form a subspace of degree 84, it is possible to extend the space used to represent the filleting

surfaces so that its dimension is increased by 16k. The surfaces obtained by minimization of the functional F in the extended space may have (and often they have indeed) a better shape.

To use this possibility, after creating the domain record with use of the procedure `gh_CreateDomainf` and perhaps after registering the options entering procedure, it is necessary to construct the basic space basis, by calling `g2h_ComputeBasisf` as before (the construction of additional functions, which are elements of a basis of the extended space does not involve any computations). Then, *instead* of `g2h_DecomposeMatrixf`, the program has to call the procedure `g2h_DecomposeExtMatrixf`, which computes the matrices A and B of the appropriately enlarged system of equations (9.2) and decomposes the matrix A into triangular factors.

The second part of the construction using the extended space is done by the procedure `g2h_ExtFillHolef`, which should be called *instead* of `g2h_FillHolef`. These procedures have identical parameter lists.

The data used in both constructions are computed and stored in the domain record independently, therefore one can compute and decompose the matrices for both constructions, and then complete them (in any order) and compare the results. The computation time for the extended space is longer, though the difference is hard to notice. However, the computations for the extended space require more memory — for the results (matrix coefficients, stored in memory blocks allocated by `malloc`) and the workspace (in the scratch memory pool, managed by the procedures described in Section 2.3). The double precision version for $k = 8$ may need about 2MB of workspace.

To construct the minimal surface of the functional F_d using the extended space, one should call the procedure `g2h_NLExtFillHolef`. The scratch memory needed for this construction may be up to 8MB.

9.3.4 Imposing constraints

To construct a surface with constraints it is necessary to create the domain representation (using `gh_CreateDomainf`), construct the basis and then to enter the matrix C of the system of constraint equations, and then call the procedure of construction with constraints.

For the **basic space** the matrix C of the system (9.4) is entered by the procedure `g2h_SetConstraintMatrixf`. The minimal surface of the functional F_c with constraints is constructed by the procedure `g2h_FillHoleConstrf`. The minimal surface of the functional F_d with constraints is constructed by the procedure `g2h_NLFillHoleConstrf`.

To enter the matrix of the system of constraint equations having the form (9.5) one should use the procedure `g2h_SetAltConstraintMatrixf`. The minimal surface of the functional F_c with such constraints is obtained by the procedure

g2h_FillHoleAltConstrf, and the minimal surface of the functional F_d is constructed by the procedure g2h_NLFillHoleAltConstrf.

For the **extended space**, to enter the matrix of the system (9.4) use the procedure g2h_SetExtConstraintMatrixf. The minimal surface of the functional F_c is constructed by the procedure g2h_ExtFillHoleConstrf, and the minimal surface of the functional F_d is constructed by the procedure g2h_NLExtFillHoleConstrf.

The matrix of the system (9.5) for the extended space is entered by the procedure g2h_SetExtAltConstraintMatrixf. The minimal surface of the functionals F_c and F_d are constructed by the procedures g2h_ExtFillHoleAltConstrf and g2h_NLExtFillHoleAltConstrf respectively.

The matrix of each of the four konds of constraints (i.e. of the form (9.4) and (9.5) for the basic and the extended spaces) may be entered independently of each other. To change the matrix of the system of constraints, one can call the appropriate procedure (one of the above) again.

9.4 Main procedures

```
#define G2H_FINALDEG 9
#define GH_MAX_K 16
```

The two symbolic constants above specify the degree of the final patches filling the hole and the maximal number of sides of this hole.

The above constants cannot simply be modified — the degree 9 results from the implemented interpolation scheme. The library procedures may use also a scheme, which produces patches of degree 10. To use this possibility, one should remove the definition of the symbol G2H_FINALDEG9 from the header file and recompile the procedures.

The domain of the parameterization of a surface filling a polygonal hole is divided into k parts. Sets of those parts are represented with short integers, i.e. 16-bit variables. To fill holes having more than 16 sides one has to modify the appropriate part of the procedures, so that e.g. 32-bit words are used, which would make it possible to fill up to thirty two-sided holes.


```
typedef struct GHoleDomainf {
    int hole_k;
    float *hole_knots;
    point2f *domain_cp;
    boolean basisG1, basisG2;
    void *privateG;
    void *privateG1;
    void *SprivateG1;
    void *privateG2;
    void *SprivateG2;
    int error_code;
  } GHoleDomainf;
```

The structure type GHoleDomainf describes the domain record, i.e. an object with all data necessary to fill holes. An application should declare only pointer variables for such structures, as the responsability for creation and consistency of data stored in such an object belongs to the library procedures.

The attribute hole_k specifies the number of sides of the hole (from 3 to 16).

The attribute hole_knots points to the array with 11k numbers, being knots of the surface representation.

The attribute domain_cp points to the array with $12k+1$ control points of the domain representation.

The attributes privateG, privateG1, SprivateG1, privateG2, SprivateG2 point to records (whose structure and contents is invisible for applications) with all other data necesary to fill the hole.

The attribute error_code is used to store the information about the success or the reason of failure of the computation.

```
GHoleDomainf* gh_CreateDomainf ( int hole_k,
                                 float *hole_knots,
                                 point2f *domain_cp );
void gh_DestroyDomainf ( GHoleDomainf *domain );
```

The procedure gh_CreateDomainf creates an object of type GHoleDomainf, which represents the domain of a surface filling a polygonal hole, and returns its address. The memory blocks for this object and for all data pointed by the pointers in it are allocated by the procedure malloc.

The parameter hole_k specifies the number k of sides of the hole (must be from 3 to 16).

The parameter hole_knots is an array with 11k floating point numbers — knots of the surface and domain representation.

The parameter domain_cp is an array with $12k+1$ control points of the domain representation. The contents of the two arrays are copied to the arrays allocated by the procedure gh_CreateDomainf.

If a sufficient memory cannot be allocated, or a data error has been detected, the procedure returns NULL.

The object just after the creation is not ready to use — the procedure does not construct the basis necessary to obtain the filleting surfaces. These computations take some time, and it may be convenient to separate them from the creation of this object in an application.

The procedure gh_DestroyDomainf deallocates (using the procedure free) the memory occupied by the domain representation (which includes all memory blocks allocated while this representation was processed).

```
void g2h_SetOptionProcf ( GHoleDomainf *domain,
    int (*OptionProc)( GHoleDomainf *domain, int query, int qn,
                       int *ndata, int **idata, float **fdata ) );
```

The procedure g2h_SetOptionProcf registers the application-supplied procedure, which selects options for the construction and transfers the necessary data. If this procedure is not called after creating the domain representation, the default procedure (which gives the default answer to all option queries) is used.

This method of introducing options was chosen in order to fix the parameter lists of the library procedures, while the construction method was under development. The advantage is that an application not using nonstandard options does not have to call the library procedures with the parameters having no significance for that application.

The principles of specifying options are described in Section 9.5.

```
boolean g2h_ComputeBasisf ( GHoleDomainf *domain );
```

The procedure g2h_ComputeBasisf constructs the basis functions, which will be used to obtain surfaces filling polygonal holes in surfaces. The parameter points to the object created by gh_CreateDomainf. The value true of the procedure indicates a success, while false is returned in case of failure.

If for the object passed as the parameter an option procedure has been registered, that procedure will be called a number of times. The responses of this procedure affect the result of the computation (i.e. the form of the basis functions), which influences the surfaces filing the holes, constructed with use of this basis.

The procedure g2h_ComputeBasisf ought to be called only once for the domain reprezentation created by gh_CreateDomainf. If it is necessary to construct bases for the same domain more than once (e.g. using various options), then the domain representation must be deallocated each time (by gh_DestroyDomainf) and created again.

The computations in this procedure take a moderate amount of time, therefore they might be done as a part of processing a single message from the system in an interactive program. The delay involved should be unnoticeable for the user.


```
boolean g2h_ComputeFormMatrixf ( GHoleDomainf *domain );
boolean g2h_DecomposeMatrixf ( GHoleDomainf *domain );
```

The procedure g2h_ComputeFormMatrixf computes the coefficients of the matrices of the system of equations solved in order to construct the filleting surface **using the basic space**. The parameter of the procedure is the domain representation created by gh_CreateDomainf, for which the procedure g2h_ComputeBasisf has successfully constructed the representation of the basis functions.

The procedure g2h_DecomposeMatrixf decomposes (with use of the Cholesky method) the matrix computed by g2h_ComputeFormMatrixf. If this matrix has not been computed yet, the procedure g2h_DecomposeMatrixf begins with computing them, by calling g2h_ComputeFormMatrixf.

Both these procedures return true to indicate the success and false to indicate a failure.

```
boolean g2h_FillHolef ( GHoleDomainf *domain,
              int spdimen, const float *hole_cp, float *acoeff,
              void (*outpatch) ( int n, int m, const float *cp ) );
```

The procedure g2h_FillHolef constructs a surface filling the polygonal hole, **using the basic space**.

The parameter domain points to a domain representation created by the procedure gh_CreateDomainf, for which the procedure g2h_ComputeBasisf has constructed (successfully) the representation of the basis functions. The number k of the hole sides and the knot sequences, which are part of the surface representation, have been specified during the call to gh_CreateDomainf.

The parameter spdimen specifies the dimension d of the space, in which the surface resides. For a polynomial surface in $\mathbb{R}^3$, this parameter will have the value 3. For a polynomial surface in $\mathbb{R}^4$, being a homogeneous representation of a rational surface in $\mathbb{R}^3$, this parameter will be 4.

The parameter hole_cp is an array with $(12k+1)d$ floating point numbers, being the coordinates of $12k+1$ control points of the surface.

The parameter acoeff may be NULL (and then it is ignored), or it may point an array, in which the procedure will store the solution of the system of equations (9.2). This array must have length at least nd, where d is the space, in which the surface resides (i.e. the value of the parameter spdimen), and n is the dimension of the basic space — it may be obtained by calling the procedure g2h_V0SpaceDimf.

The parameter outpatch is a pointer to the procedure (supplied by the application), which will be called k times in order to output the control points of k Bézier patches of degree $(9,9)$, filling the hole.

```
boolean g2h_ComputeExtFormMatrixf ( GHoleDomainf *domain );
boolean g2h_DecomposeExtMatrixf ( GHoleDomainf *domain );
```

The procedure g2h_ComputeExtFormMatrixf computes the coefficients of the matrices of the system of equations solved in order to construct the filleting surface **using the extended space**. The parameter of the procedure is the domain representation created by gh_CreateDomainf, for which the procedure g2h_ComputeBasisf has successfully constructed the representation of the basis functions.

The procedure g2h_DecomposeMatrixf decomposes (with use of the Cholesky method) the matrix computed by g2h_ComputeExtFormMatrixf. If this matrix has not been computed yet, the procedure g2h_DecomposeExtMatrixf begins with computing them, by calling g2h_ComputeExtFormMatrixf.

Both these procedures return true to indicate the success and false to indicate a failure.

```
boolean g2h_ExtFillHolef ( GHoleDomainf *domain,
              int spdimen, const float *hole_cp, float *acoeff,
              void (*outpatch) ( int n, int m, const float *cp ) );
```

The procedure g2h_ExtFillHolef constructs a surface filling the polygonal hole, **using the extended space**.

The parameter domain points to a domain representation created by the procedure gh_CreateDomainf, for which the procedure g2h_ComputeBasisf has constructed (successfully) the representation of the basis functions. The number k of the hole sides and the knot sequences, which are part of the surface representation, have been specified during the call to gh_CreateDomainf.

The parameter spdimen specifies the dimension d of the space, in which the surface resides. For a polynomial surface in $\mathbb{R}^3$, this parameter will have the value 3. For a polynomial surface in $\mathbb{R}^4$, being a homogeneous representation of a rational surface in $\mathbb{R}^3$, this parameter will be 4.

The parameter hole_cp is an array with $(12k+1)d$ floating point numbers, being the coordinates of $12k+1$ control points of the surface.

The parameter acoeff may be NULL (and then it is ignored), or it may point an array, in which the procedure will store the solution of the system of equations (9.2). This array must have length at least nd, where d is the space, in which the surface resides (i.e. the value of the parameter spdimen), and n is the dimension of the extended space — it may be obtained by calling the procedure g2h_ExtV0SpaceDimf.

The parameter outpatch is a pointer to the procedure (supplied by the application), which will be called k times in order to output the control points of k Bézier patches of degree $(9,9)$, filling the hole.

```
int g2h_GetErrorCodef ( GHoleDomainf *domain,
                        char **ErrorString );
```


The procedure g2h_GetErrorCodef may be called in case of failure of some step of the construction, in order to find out what was the reason. The value returned is the error code. If the parameter ErrorString is not NULL, the variable *ErrorString after return points to a character string, which is an error description.

9.5 Entering options

The procedure g2h_SetOptionProcf described in the previous section may register a procedure (supplied by the application), which will „answer the queries" about construction options to be used. This procedure has to have the following header (the names of the procedure and of the parameters may be different):

```
int SetOptionf ( GHoleDomainf *domain, int query, int qn,
                 int *ndata, int **idata, float **fdata );
```

During the construction of the basis this procedure will be called a number of times. Its first parameter will point to the representation of domain of the surface being constructed. The second parameter (query) is the number of option to be specified by the procedure. The parameter qn is an additional number, which will perhaps be necessary with options introduced in future versions of the library, and now it may be ignored.

The value returned is interpreted as the answer to the query about the option to be used. Possible numbers of options (i.e. the values of the parameter query) and answers are symbolic constants, whose names begin respectively with G2HQUERY_ and G2H_). They are listed below. The list may change in future versions of the library libeghole.

```
#define G2H_DEFAULT 0

#define G2HQUERY_CENTRAL_POINT 1
#define G2H_CENTRAL_POINT_GIVEN 1

#define G2HQUERY_CENTRAL_DERIVATIVES1 2
#define G2H_CENRTAL_DERIVATIVES1_ALT 1
#define G2H_CENTRAL_DERIVATIVES1_GIVEN 2

#define G2HQUERY_DOMAIN_CURVES 3
#define G2H_DOMAIN_CURVES_DEG4 1

#define G2HQUERY_BASIS 4
#define G2H_USE_RESTRICTED_BASIS 1
```

After each call, the option procedure may return the value G2H_DEFAULT. In particular, this is the value, which must be returned for each option (indicated by the

parameter query) not recognized by the procedure. This will give the application a chance of working correctly after recompilation with future versions of the library.

If the parameter query is equal to G2HQUERY_CENTRAL_POINT, then the answer (i.e. the value returned) G2H_DEFAULT will cause taking the central point of the domain (i.e. the common corner of the areas, to which the domain will be divided) at the gravity centre of the domain edge midpoints (this is the construction described in the papers). If the value returned is G2H_CENTRAL_POINT_GIVEN, then the variable *ndata must be assigned the value 2, and the variable *fdata must point to the array with two floating point numbers, being the coordinates of the central point given by the application.

If the value of the parameter query is G2HQUERY_CENTRAL_DERIVATIVES1 and the procedure returns the value G2H_DEFAULT, then the first order derivative vectors of the domain division curves and the cross derivative vectors of the auxiliary domain patches will be constructed as described in the articles. Returning G2H_CENTRAL_DERIVATIVES_ALT will cause taking the curve derivatives as before, and the cross derivative vectors will be orthogonal to the curve derivatives. Returning G2H_CENTRAL_DERIVATIVES_GIVEN means that the application produced the curve derivatives at the central point. The variable *ndata must then have the value 2k (for a k-sided hole), and the table pointed by the variable *fdata must consist of 2k floating point numbers. The consecutive pairs are the coordinates of the first order derivative vectors of the consecutive curves.

The parameter query equal to G2HQUERY_DOMAIN_CURVES denotes the query for the method of constructing derivatives of domain division curves of order higher than 1. In response to this query G2H_DEFAULT should be returned, which will cause taking zero derivatives of order 2, 3 and 4. Other options for this step of construction are still under development and need not work correctly.

If the value of the parameter query is equal to G2HQUERY_BASIS, then returning G2H_DEFAULT enables all degrees of freedom of choice of partial derivatives of the patches filling the hole at their common point (their number is the dimension of the basic space). Depending on the number of hole sides and the division of the domain, their number may range from 16 to 30 (for holes with three to eight sides). Returning G2H_USE_RESTRICTED_BASIS causes restricting the number of degrees of freedom to 15 (i.e. to the dimension of the space of bivariate polynomials of degree up to 4).

9.6 Imposing constraints

```
int g2h_V0SpaceDimf ( GHoleDomainf *domain );
int g2h_ExtV0SpaceDimf ( GHoleDomainf *domain );
```

The procedures g2h_V0SpaceDimf and g1h_ExtV0SpaceDimf compute respectively the dimensions of the basic and extended space, used in the constructions of surfaces filling the hole. The parametr domain must point to the domain record,

for which the basis of the basic space has successfully been constructed.

```
boolean g2h_GetBPDerivativesf ( GHoleDomainf *domain,
                                int cno, float *val );
```

The procedure g1h_GetBPDerivativesf evaluates the basis functions (of the basic space) at the central point and the derivatives of order $1, \ldots, 4$ of the boundary curves of the basis function patches. The number of the curve is specified by the parameter cno (it must be from 0 to $k-1$). The values computed are stored in the array val, of length $5n$, where n id the dimension of the basic space. Subsequent five-tuples of numbers stored in this array correspond to the subsequent basis functions.

```
boolean g2h_GetBFuncPatchf ( GHoleDomainf *domain,
                             int fn, int pn, float *bp );
```

The procedure g1h_GetBFuncPatchf evaluates and stores in the array bp the coefficients of the j-th patch of the i-th basis function. The parameter fn specifies the number $i \in \{0, \ldots, n-1\}$ of the basis function, and pn specifies the number $j \in \{0, \ldots, k-1\}$ of its patch. This procedure may be useful, when the constraints imposed on the surface are not interpolation conditions at the central point of the surface.

The basis function patches are bivariate polynomials of degree G2H_FINALDEG with respect to each variable. Their coefficients represent the patches in the tensor product Bernstein basis.

```
boolean g2h_SetConstraintMatrixf ( GHoleDomainf *domain,
                                   int nconstr, const float *cmat );
```

The procedure g2h_SetConstraintMatrixf associates with the domain of the surface a matrix of the system of equations, which describe constraints imposed on the surface. The parameter nconstr is the number of constraints (i.e. equations), which is the number of rows of the matrix. The number of columns is the dimension of the basic space. Subsequent rows must be given in the array cmat. They have to be linearly independent.

The procedure returns true in case of success, and false if the matrix is not rowwise-regular.

```
boolean g2h_FillHoleConstrf ( GHoleDomainf *domain,
              int spdimen, const float *hole_cp,
              int nconstr, const float *constr,
              float *acoeff,
              void (*outpatch) ( int n, int m, const float *cp ) );
```

The procedure g1h_FillHoleConstrf constructs a surface filling the hole, which satisfies the constraints imposed on it, with use of the basic space. Before calling it, the matrix of the constraint equations must be speficied (this fixes also

the number of constraints). The parameters domain, spdimen, hole_cp, acoeff and outpatch have the same meaning as for the procedure g2h_FillHolef. The patameter nconstrf specifies the number of constraints (it must match with the number given at the call of g2h_SetConstraintMatrixf. The array constr contains the right-hand side matrix of the constraint equations — nconstr rows, each with spdimen numbers.

The change of constraints (both the left- and the right-hand side may be done without reconstructing the domain record. To modify the constraints, it suffices to call again g2h_SetConstraintMatrixf and g1h_FillHoleConstrf.

```
boolean g2h_SetAltConstraintMatrixf ( GHoleDomainf *domain,
                    int spdimen,
                    int nconstr, const float *cmat );
```

The procedure g2h_SetAltConstraintMatrixf enters the matrix C of the system of constraint equations (9.5) for the construction with the basic space. The matrix has dimensions $\mathtt{n}\mathtt{d} \times w$, where n is the space dimension (it may be obtained by calling the procedure g2h_V0SpaceDimf), d is the dimension of the space, in which the surface is located (e.g. 3), and w is the number of constraints.

The parameter spdimen specifies the dimension d, the parameter nconstr specifies the number of knots. The coefficients of the matrix C are given it the array cmat. The pitch of this array is equal to the length of its row, i.e. nd. The matrix C must be rowwise regular.

The return value true of the procedure g2h_SetAltConstraintMatrixf signalls the rowwise regularity of the matrix. The value false means that the matrix is not regular (i.e. the numerical procedure classified the matrix as one with linearly dependent rows) and it is not suitable for the construction.

```
boolean g2h_FillHoleAltConstrf ( GHoleDomainf *domain,
          int spdimen, const float *hole_cp,
          int naconstr, const float *constr,
          float *acoeff,
          void (*outpatch) ( int n, int m, const float *cp ) );
```

The procedure g2h_FillHoleAltConstrf constructs thhe filling surface being the minimal point of the functional $\mathrm{F_c}$ in the set of surfaces representable with use of the basic space and satisfying the constraint equations (9.5). The matrix C of this system of equations must be entered before the call to this procedure.

The parameter spdimen specifies the dimension d of the space, in which the surface is located. The array hole_cp contains the control points of this surface. The parameter naconstr specifies the number of constraints w. In the array constr one has to supply the coefficients of the right hand side vector of the system of constraint equations (w numbers). The numbers d and w must agree with the values of the appropriate parameters of the preceding call to g2h_SetAltConstraintMatrixf.

If the parameter acoeff is not NULL, then it has to point to an array, in which the procedure will store the vectors $\mathbf{a}_0, \ldots, \mathbf{a}_{\mathrm{n}-1}$ obtained by the optimisation. The procedure pointed by the parameter output will be called k times in order to output the result of the construction, i.e. the Bézier patches filling the hole.

```
boolean g2h_SetExtConstraintMatrixf ( GHoleDomainf *domain,
                                      int nconstr, const float *cmat );
```

The procedure g2h_SetExtConstraintMatrixf associates with the domain of the surface a matrix of the system of equations, which describe constraints imposed on the surface. The parameter nconstr is the number of constraints (i.e. equations), which is the number of rows of the matrix. The number of columns is the dimension of the extended space. Subsequent rows must be given in the array cmat. They have to be linearly independent.

The procedure returns true in case of success, and false if the matrix is not rowwise-regular.

```
boolean g2h_ExtFillHoleConstrf ( GHoleDomainf *domain,
              int spdimen, const float *hole_cp,
              int nconstr, const float *constr,
              float *acoeff,
              void (*outpatch) ( int n, int m, const float *cp ) );
```

The procedure g1h_ExtFillHoleConstrf constructs a surface filling the hole, which satisfies the constraints imposed on it, with use of the extended space. Before calling it, the matrix of the constraint equations must be speficied (this fixes also the number of constraints). The parameters domain, spdimen, hole_cp, acoeff and outpatch have the same meaning as for the procedure g2h_ExtFillHolef. The patameter nconstrf specifies the number of constraints (it must match with the number given at the call of g2h_SetExtConstraintMatrixf. The array constr contains the right-hand side matrix of the constraint equations — nconstr rows, each with spdimen numbers.

The change of constraints (both the left- and the right-hand side may be done without reconstructing the domain record. To modify the constraints, it suffices to call again g2h_SetExtConstraintMatrixf and g1h_ExtFillHoleConstrf.

The constraints for the basic and extended space are specified independently.

```
boolean g2h_SetExtAltConstraintMatrixf ( GHoleDomainf *domain,
                    int spdimen,
                    int nconstr, const float *cmat );
```

The purpose of the procedure g2h_SetExtAltConstraintMatrixf is to enter the matrix C of the system of constraint equations (9.5) for the constructions of the filling surface represented with use of the extended space. The dimensions of this matrix are $\mathtt{n}\mathtt{d} \times w$, where n is the dimension of the extended space, d is the

dimension of the space in which the surface is located (e.g. 3), and the number of rows w is the number of constraints.

Parameters: `spdimen` — dimension d, `naconstr` — number of constraints w, `acmat` — array of coefficients of the matrix C. This matrix has the pitch equal to the row length, i.e. nd.

The matrix C may be divided into blocks $C_0, \ldots, C_{d-1}$, whose dimensions are $w \times d$. If the constraints are imposed on a surface, which minimises the functional F_d, then it must be $d = 3$. In each row of the each block the first 16k coefficients must be zeros, and the rows of the matrix C must be linearly independent. This limitation is a consequence of a numerical method used in the construction.

The return vaalue `true` signalls the acceptance of the matrix, and `false` means that by the numerical computation the rows of the matrix are considered linearly dependent.

```
boolean g2h_ExtFillHoleAltConstrf ( GHoleDomainf *domain,
          int spdimen, const float *hole_cp,
          int naconstr, const float *constr,
          float *acoeff,
          void (*outpatch) ( int n, int m, const float *cp ) );
```

The procedure `g2h_ExtFillHoleAltConstrf` constructs the surface filling the hole, which is the minimal point of the functional F_c in the set of surfaces satisfying the constraint equations having the form (9.5). The matrix of this system must be entered by a preceding call to the procedure `g2h_SetExtAltConstraintsf`.

The parameters `spdimen` and `naconstr` specify the dimension d of the space containing the surface and the number of constraints w. These numbers must match the values of parameters of the preceding call to `g2h_SetExtAltConstraintsf`. In the array `hole_cp` one has to specify the coordinates of the control points of the surface. In the array `constr` there must be the coefficients of the right hand side of the system (9.5). If the parameter `acoeff` is not `NULL`, then it must point to an array, in which the vectors $\mathbf{a}_0, \ldots, \mathbf{a}_{n-1}$ will be stored. The parameter `outpatch` points to a procedure called in order to output the construction result in the form of Bézier patches.

The return value `true` indicates the success of the construction, while `false` signalls failure.

```
float g2h_FunctionalValuef ( GHoleDomainf *domain, int spdimen,
                        const float *hole_cp, const float *acoeff );
float g2h_ExtFunctionalValuef ( GHoleDomainf *domain, int spdimen,
                        const float *hole_cp, const float *acoeff );
```

The procedures `g2h_FunctionalValuef` and `g2h_ExtFunctionalValuef` compute the values of the functional F_c for a filling surface given by the control points given in the array `hole_cp` and the vectors $\mathbf{a}_0, \ldots, \mathbf{a}_{n-1}$ given in the array `acoeff`,

being respectively the coefficients of the representation using the basic and the extended space.

After constructing the surface with any of the procedures described here or in the next section, one may call one of the above procedures. To do this, it is necessary to allocate an array long enough to accomodate dn `float` numbers, where d id the dimension of the space with the surface and n is the dimension of the space V_0, pass this array as the parameter `acoeff` to the construction procedure, and then pass it to the procedure computing the functional value.

9.6.1 Filling holes with B-spline patches

```
#define G2H_S_MAX_NK 4
#define G2H_S_MAX_M1 3
#define G2H_S_MAX_M2 7
```

```
boolean g2h_ComputeSplBasisf ( GHoleDomainf *domain,
                               int nk, int m1, int m2 );
```

```
boolean g2h_ComputeSplFormMatrixf ( GHoleDomainf *domain );
boolean g2h_DecomposeSplMatrixf ( GHoleDomainf *domain );
```

```
boolean g2h_SplFillHolef ( GHoleDomainf *domain,
          int spdimen, const float *hole_cp,
          float *acoeff,
          void (*outpatch) ( int n, int lknu, const float *knu,
                             int m, int lknv, const float *knv,
                             const float *cp ) );
```

9.7 Nonlinear constructions procedures

The procedures described in this section construct the filling surface by minimisation of the functional F_d. The feasibility of those constructions depends on the given surface with the hole. The constructions are also more time consuming. The surface must be located in $\mathbb{R}^3$ (therefore it cannot be e.g. a homogeneous representation of a rational surface).

```
boolean g2h_ComputeNLNormalf ( GHoleDomainf *domain,
                               const point3f *hole_cp,
                               vector3f *anv );
```

The procedure `g2h_ComputeNLNormalf` constructs the unit vector of one of the axes of the coordinate system, in which the surface will be represented during the construction; the given surface and the filling surface are supposed to form a graph

od a scalar function of two variables in this system. The input parameters are domain — pointer to the domain representation and hole_cp — array of control points of the surface. The parameter anv points to the variable, in which the result is stored.

The return value true indicates success, and false — lack of success (if the surface determined by the given control points is not flat enough). The procedure g2h_ComputeNLNormalf in principle is not intended to be called from applications.

```
boolean g2h_NLFillHolef ( GHoleDomainf *domain,
            const point3f *hole_cp, float *acoeff,
            void (*outpatch) ( int n, int m, const point3f *cp ) );
```

The procedure g2h_NLFillHolef constructs the filling surface, being the minimal point of the functional F_d in the basic space, without constraints. This procedure corresponds to g2h_FillHolef and it has the same parameters except for spdimen.

```
boolean g2h_NLFillHoleConstrf ( GHoleDomainf *domain,
            const point3f *hole_cp,
            int nconstr, const vector3f *constr,
            float *acoeff,
            void (*outpatch) ( int n, int m, const point3f *cp ) );
```

The procedureg2h_NLFillHoleConstrf constructs the filling surface, which minimises the functional F_d in the basic space, with constraints described by the systemw przestrzeni (9.4). This procedure corresponds to g2h_FillHoleConstrf.

```
boolean g2h_NLFillHoleAltConstrf ( GHoleDomainf *domain,
            const point3f *hole_cp,
            int nconstr, const float *constr,
            float *acoeff,
            void (*outpatch) ( int n, int m, const point3f *cp ) );
```

The procedure g2h_NLFillHoleAltConstrf constructs the filling surface, which minimises the functional F_d in the basic space with constraints described by the system (9.5). It corresponds to g2h_FillHoleAltConstrf.

```
boolean g2h_NLExtFillHolef ( GHoleDomainf *domain,
            const point3f *hole_cp,
            float *acoeff,
            void (*outpatch) ( int n, int m, const point3f *cp ) );
```

The procedure g2h_NLExtFillHolef constructs the filling surface, being the minimal point of the functional F_d in the extended space, without constraints. This procedure corresponds to g2h_ExtFillHolef and it has the same parameters except for spdimen.


```
boolean g2h_NLExtFillHoleConstrf ( GHoleDomainf *domain,
            const point3f *hole_cp,
            int nconstr, const vector3f *constr,
            float *acoeff,
            void (*outpatch) ( int n, int m, const point3f *cp ) );
```

The procedure g2h_NLExtFillHoleConstrf constructs the filling surface, which minimises the functional F_d in the extended space, with constraints described by the system (9.4). This procedure corresponds to g2h_ExtFillHoleConstrf.

The matrix C (entered by g2h_SetExtConstraintMatrixf) allowed for the construction done by this procedure must have the first 16k coefficients of each row equal to 0.

```
boolean g2h_NLExtFillHoleAltConstrf ( GHoleDomainf *domain,
            const point3f *hole_cp,
            int naconstr, const float *constr,
            float *acoeff,
            void (*outpatch) ( int n, int m, const point3f *cp ) );
```

The procedure g2h_NLExtFillHoleAltConstrf constructs the filling surface, which minimises the functional F_d in the extended space with constraints described by the system (9.5). It corresponds to g2h_ExtFillHoleAltConstrf.

The blocks C_0, C_1, C_2 of the matrix $C = [C_0, C_1, C_2]$ allowed for the construction done by this procedure (entered by g2h_SetExtAltConstraintMatrixf) must have the first 16k coefficients of each row equal to 0.

```
boolean g2h_NLFunctionalValuef ( GHoleDomainf *domain,
                                 const point3f *hole_cp,
                                 const vector3f *acoeff,
                                 float *funcval );
```

The procedure g2h_NLFunctionalValuef computes the value of the functional F_d for a surface represented by the control points goven in the array hole_cp and the vectors $\mathbf{a}_0, \ldots, \mathbf{a}_{n-1}$ (coefficients in the basic space basis).

```
boolean g2h_NLExtFunctionalValuef ( GHoleDomainf *domain,
                                   const point3f *hole_cp,
                                   const vector3f *acoeff,
                                   float *funcval );
```

The procedure g2h_NLExtFunctionalValuef computes the value of the functional F_d for a surface represented by the control points goven in the array hole_cp and the vectors $\mathbf{a}_0, \ldots, \mathbf{a}_{n-1}$ (coefficients in the extended space basis).

```
boolean g2h_NLSplFillHolef ( GHoleDomainf *domain,
          const point3f *hole_cp,
          float *acoeff,
          void (*outpatch) ( int n, int lknu, const float *knu,
                             int m, int lknv, const float *knv,
                             const point3f *cp ) );
```

9.8 Visualisation procedures

The name "visualisation procedures" concerns the procedures, which extract various data from the domain record. These data may be used to get insight into the construction, by showing them on various pictures.

```
void g2h_DrawDomSurrndPatchesf ( GHoleDomainf *domain,
          void (*drawpatch) ( int n, int m, const point2f *cp ) );
```

The procedure g2h_DrawDomSurrndPatchesf extracts the Bézier representations of bicubic patches surrounding the domain, i.e. the polynomial pieces of the B-spline patches represented by the knots and domain control points specified when the domain record has been created.

The parameter domain points to the domain record, the parameter drawpatch is a pointer to the procedure, which for a k-sided hole will be called 3k times, with the parameters representing consecutive domain surrounding patches.

```
void g2h_DrawDomAuxPatchesf ( GHoleDomainf *domain,
          void (*drawpatch) ( int n, int m, const point2f *cp ) );
```

The procedure g2h_DrawDomAuxPatchesf extracts the Bézier representations of the domain auxiliary patches. The parameter domain points to the domain record, and the parameter drawpatch points to the procedure to be called k times in order to output the subsequent patches.

```
void g2h_DrawBasAuxPatchesf ( GHoleDomainf *domain, int fn,
          void (*drawpatch) ( int n, int m, const float *cp ) );
```

The procedure g2h_DrawBasAuxPatchesf extracts the Béziera reprezentations of the basis function auxiliary patches. These are bivariate polynomials, for each basis function of the basic space there are k such polynomials. The dimension n of this space may be obtained by calling the procedure g2h_V0SpaceDimf.

The parameter domain points to the domain record, the parameter fn is the number of the basis function (its value must be from 0 to $n-1$), the parameter drawpatch points to the procedure, which will be called k times in order to output the subsequent auxiliary patches of the basis function, whose number is fn.


```
void g2h_DrawJFunctionf ( GHoleDomainf *domain, int i, int l,
          void (*drawpoly) ( int deg, const float *f ) );
```

The procedure g2h_DrawJFunctionf extracts the coefficients (in the Bernstein basis of the appropriate degree) of a junction function used to construct the basis functions. For each subarea Ω_i of the domain there are 16 such polynomials.

The parameter domain points to the domain record. The parameter i is the number of the subarea (it must be from 0 to $k-1$), the parameter l identifies the junction function to be output by the procedure pointed by the parameter drawpoly.

The value of the parameter l from 0 to 15 selects one of the sixteen junction functions, and from 16 to 27 the product of appropriate functions, used in the construction. For the information about numbering the junction functions, refer to the procedure source code.

```
void g2h_DrawDiPatchesf ( GHoleDomainf *domain,
          void (*drawpatch) ( int n, int m, const point2f *cp ) );
```

The procedure g2h_DrawDiPatchesf extracts the Bézier representations of degree $(9,9)$ of the domain patches. The parameter domain points to the domain record, and the parameter drawpatch points to the procedure to be called k times, in order to output the subsequent patches.

```
void g2h_ExtractPartitionf ( GHoleDomainf *domain,
          int *hole_k, int *hole_m,
          float *partition, float *part_delta, float *spart_alpha,
          float *spart_malpha, float *spart_salpha,
          float *spart_knot, float *alpha0,
          boolean *spart_sgn, boolean *spart_both );
```

The procedure g2h_ExtractPartitionf extracts the information about the the partition of the full angle at the central point of the domain Ω, divided into subareas Ω_i.

```
void g2h_ExtractCentralPointf ( GHoleDomainf *domain,
                          point2f *centp, vector2f *centder );
```

The procedure g2h_ExtractCentralPointf extracts the central point of the domain and the first order derivatives of the domain division curves at the central point.

The parameter domain points to the domain record, the parameter centp points to the variable, to which the central point is to be assigned, and the parameter centder is an array of length k, in which the derivative vectors will be stored.

```
void g2h_DrawBasAFunctionf ( GHoleDomainf *domain, int fn,
          void (*drawpatch) ( int n, int m, const point3f *cp ) );
```

The procedure g2h_DrawBasAFunctionf may be used to obtain the information about the basis function, whose number is fn$\in \{0,\ldots,n'-1\}$. The procedure *drawpatch is called k times, each time its parameters describe one of the domain patches (the coordinates x, y of the points in the array cp) and the corresponding basis function patch (the z coordinate). The parameters of this procedure describe the degree and the Bézier control ppoints in the array cp.

```
void g2h_DrawBasBFunctionf ( GHoleDomainf *domain, int fn,
          void (*drawpatch) ( int n, int m, const point3f *cp ) );
```

The procedure g2h_DrawBasBFunctionf may be used to obtain the information about the basis function, whose number is fn$\in \{n,\ldots,n+m-1\}$. The procedure *drawpatch, called to pass this information, is called in the same way as the procedure passed to g2h_DrawBasAFunctionf (and it may be just the same procedure).

```
void g2h_DrawBasCNetf ( GHoleDomainf *domain, int fn,
          void (*drawnet) ( int n, int m, const point3f *cp ) );
```

The procedure g2h_DrawBasCNetf may be used to obtain the B-spline control nets representing the functions φ_i for i =fn.

```
void g2h_DrawMatricesf ( GHoleDomainf *domain,
          void (*drawmatrix)(int nfa, int nfb,
                             float *amat, float *bmat) );
```

The procedure g2h_DrawMatricesf extracts the matrices A and B of the system of equations (9.2), for the basic space basis. As the matrix A is symmetric, its representation is packed, as described in Section 3.3.

The parameter drawmatrix points to a procedure to be called with the parameters describing the matrices; nfa is the number of rows of both matrices and the number of columns of A. The parameter nfb id the number of columns of the matrix B. The parameters amat and bmat are arrays with the coefficients.

```
void g2h_DrawExtMatricesf ( GHoleDomainf *domain,
           void (*drawmatrix)(int k, int r, int s,
                              float *Aii, float *Aki, float *Akk,
                              float *Bi, float *Bk) );
```

The procedure g2h_DrawMatricesf may be used to obtain the matrices A and B of the system of equations (9.2), for the basis of the extended space. The matrix A is symmetric, it has the block structure, which may be (and is) represented as described in Section 3.5. The matrix B is full, it is represented in a block form.

The parameter drawmatrix points to a procedure, which will be called with the parameters describing the matrices; its parameters k, r and s describe the number ans sizes of the blocks of the matrix A. In the arrays Aii, Aki and Akk there are coefficients of A, and the coefficients of B are given in the arrays Bi and Bk.


```
int g2h_DrawBFcpnf ( int hole_k, unsigned char *bfcpn );
```

The procedure g2h_DrawBFcpnf stores in the array bfcpn the indexes of these control points of the surface (and domain), which are relevant for the shape of the domain and the hole in the surface, and the tangent planes and curvatures at the hole boundary. The total number of the control points for a k-sided hole is $12k+1$, and there are $6k+1$ relevant ones; they are marked by black dots in Figure 9.1. The ordering of the numbers corresponds to the ordering of the basis functions $\phi_n,\ldots,\phi_{n+m-1}$.

The return value is $6k+1$.

```
boolean g2h_GetFinalPatchCurvesf ( GHoleDomainf *domain,
          int spdimen, const float *hole_cp, float *acoeff,
          void (*outcurve) ( int n, const float *cp ) );
boolean g2h_GetExtFinalPatchCurvesf ( GHoleDomainf *domain,
          int spdimen, const float *hole_cp, float *acoeff,
          void (*outcurve) ( int n, const float *cp ) );
boolean g2h_GetSplFinalPatchCurvesf ( GHoleDomainf *domain,
          int spdimen, const float *hole_cp, float *acoeff,
          void (*outcurve) ( int n, int lkn,
                    const float *kn, const float *cp ) );
```

10. The libbsmesh library

10.1 Mesh representation

A mesh is an object consisting of **vertices**, **edges** and **facets**. It may be used e.g. to represent a polyhedron or a spline surface. An edge is a line segment between two vertices. A facet is a closed polyline made of the edges. An edge may belong to one or two facets; in the former case it is called a **boundary edge**, and in the latter case it is an **internal edge**.

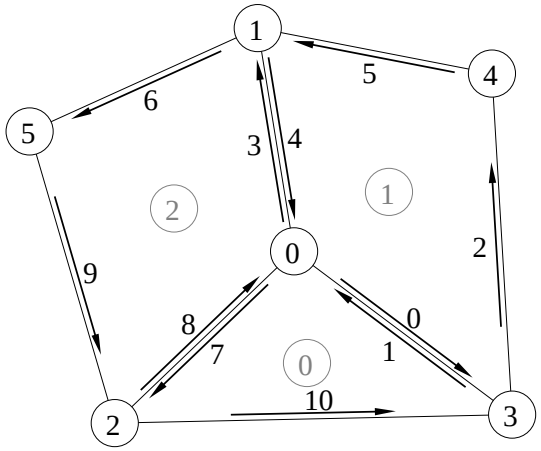
In the mesh representation processed by procedures of this library a boundary and internal edge is represented by one or two **halfedges** respectively. A halfedge is oriented; one of its vertices is the first, the other one is the second. The orientation of the other halfedge in the pair representing an internal edge is the opposite. Each halfedge is associated with one facet.

The vertices, halfedges and facets are stored in arrays (indexed from 0), and identified by the array indices. The complete representation of the mesh consists of three numbers: the number of vertices, n_v, the number of halfedges, n_h, and the number of facets, n_f, and of six arrays: the array of vertices `v`, the array of vertex positions `pos`, the array of indices of the halfedges having origins at the vertices `vhei`, the array of halfedges `he`, the array of facets `fac` and the array of indices of the halfedges forming the facets `fhei`. The vertices, halfedges and facets are described by the following structures:

```
typedef struct {
    char degree;
    int  firsthalfedge;
  } BSMfacet, BSMvertex;

typedef struct {
    int v0, v1;
    int facetnum;
    int otherhalf;
  } BSMhalfedge;
```

An example of a mesh representation is shown in Figure 10.1; the array with vertex positions is omitted. The numbers on the picture are indices to the arrays, which identify the vertices, halfedges and facets. For each halfedge its orientation is shown. The degree of the internal vertex 0 is 3. The three halfedges, whose origin is this vertex, are 7, 3 and 0—their indices are the first three numbers in the `vhei` array. Vertex 1 is a boundary vertex. It is incident with three edges, however it is the




```
int nv = 6, nhe = 11, nf = 3;
BSMvertex v[6] = {{3,0},{2,3},
    {2,5},{2,7},{1,9},{1,10}};
int vhei[11] = {7,3,0,4,6,8,
    10,1,2,5,9};
BSMhalfedge he[11] = {{0,3,1,1},
   {3,0,0,0},{3,4,1,-1},
   {0,1,2,4},{1,0,1,3},
   {4,1,1,-1},{1,5,2,-1},
   {0,2,0,8},{2,0,2,7}
   {5,2,2,-1},{2,3,0,-1}};
BSMfacet fac[3] = {{3,0},
   {4,3},{4,7}};
int fhei[11] = {1,7,10,5,4,0,2,9,8,3,6};
```

Figure 10.1. An example of a mesh

origin of only two halfedges, 4 and 6. The indices of halfedges for the i-th vertex are the `v[i].degree` numbers in the `vhei` array, from the `v[i].firsthalfedge`-th entry. Note that there is only one correct ordering of halfedge indices in the `vhei` array, as the last halfedge of a boundary vertex must be the one without a pair. The ordering of the halfedge indices for each vertex must reflect the orientation of all vertices—note that for each vertex on the picture this ordering is clockwise.

The representation of a facet is just the same as that of a vertex—only two numbers are necessary, the number of halfedges (`degree`) and the index (`firsthalfedge`) of the first entry in the `fhei` array with the indices of the halfedges. Their ordering is important—on the picture it is counterclockwise for all facets (for any part of a mesh in space, if it is flattened and drawn, the orderings of halfedges for all vertices and facets must be opposite). Currently the meshes must be orientable, so no part of a mesh may be a Möbius band.

There is no explicit list of vertices for a facet—to find them, one has to find subsequent halfedges of the facet and then take the vertices, which are the origins of the halfedges.

In the data structure representing a halfedge the fields `v0` and `v1` identify the origin and end of the halfedge, `facetnum` is the number of facet, the halfedge belongs to, and `otherhalf` is the index of the other halfedge in the pair. The orientations of the halfedges in a pair must be opposite. If the halfedge represents a boundary edge, it does not have a pair, and the value of the `otherhalf` field must be -1.

```
boolean bsm_CheckMeshIntegrity (
            int nv, const BSMvertex *v, const int *vhei,
            int nh, const BSMhalfedge *he,
            int nf, const BSMfacet *fac, const int *fhei );
```

The procedure `bsm_CheckMeshIntegrity` verifies the topological consistency of a mesh described by the parameters. The value returned is `true` is the mesh has passed the test, and `false` if errors have been detected or if the test was impossible to complete because of insufficient scratch memory. The following conditions are verified:

- The mesh has to have at least 1 facet, 3 halfedges and 3 vertices.
- The arrays `vhei` and `fhei` must contain permutations of the set of numbers $\{0, \ldots, n_h - 1\}$.
- Each halfedge must belong to some facet (its `facetnum` field must have the value from 0 to $n_f - 1$). The values of the fields `v0` and `v1` must be from 0 to $n_v - 1$.
- Each halfedge must either be a boundary halfedge (with `otherhalf` $= -1$) or make a consistent pair with another halfedge (the two halfedges must point each other with the `otherhalf` field and have opposite orientations—the origin of one of them must be the end of the other).
- Each vertex must be the origin of at least one halfedge, but if it is an internal vertex (its last halfedge has a pair), then its degree must be at least 3. Each halfedge in the list of halfedges of the i-th vertex must have the field `v0` with the value i.
- For each facet, the halfedges must form a closed polyline, i.e. the end of a halfedge must be the origin of the next halfedge of this facet (and the end of the last halfedge must be the origin of the first halfedge). The number i of the facet must be the value of the field `facetnum` of each of the facet's halfedges.
- The last thing to verify is the orientation, represented with the ordering of the halfedges and facets. The rule is briefly described with the example in Figure 10.1.

```
void bsm_TagMesh ( int nv, BSMvertex *mv, int *mvhei,
                   int nhe, BSMhalfedge *mhe,
                   int nfac, BSMfacet *mfac, int *mfhei,
                   char *vtag, char *ftag,
                   int *vi, int *vb, int *ei, int *eb );
```


The `bsm_TagMesh` procedure is an auxiliary routine, whose purpose is to find and count the boundary vertices and facets. A boundary vertex or facet has at least one boundary halfedge (having no pair). The first 8 parameters describe the mesh. The arrays `vtag` and `ftag` must have at least n_v and n_f entries respectively. Each entry is set to 0 if the vertex or facet is internal, or to 1 if it is boundary. The last four parameters must point to variables, in which the total numbers of inner and boundary vertices and inner and boundary *edges* are stored.

10.2 Mesh refinement procedures

Mesh refinement is an operation, which produces a new mesh, usually with more vertices, halfedges and facets. By repeating this operation one may get a sequence of meshes convergent to a limiting surface. The mesh refinement is a generalization of the Lane-Riesenfeld algorithm of inserting knots to B-spline surfaces (represented with uniform knots—the result is the representation with a twice denser uniform knot sequence). One can use this operation to display an approximation of the limiting surface, i.e. one of the fine meshes from the sequence obtained with refinement, or to do something else (e.g. modify by repositioning vertices or by editing the topology) with a fine mesh obtained by refinement.

Note that the numbers of vertices, halfedges and facets of the subsequent obtained by iterating of refinement meshes grow exponentially, and it is easy to exceed the capacity of the computer's memory.

The refinement operation is a composition of two more elementary operations, called `doubling` and `averaging`. One step of doubling is followed by n averaging steps, where n is a parameter. If the refinement with n averaging steps is iterated, the limiting surface consists of polynomial patches of degree (n, n).

There are two sets of procedures implementing these operations. The first set consists of procedures, which perform the operations directly, i.e. they produce the new mesh representation, in particular with an array of vertex positions in a d-dimensional space. The procedures of the second set, instead of computing the coordinates of the vertices, produce appropriate arrays. If the vertices $\boldsymbol{v}'_0, \ldots, \boldsymbol{v}'_{n'_v-1}$ of the given mesh and $\boldsymbol{v}_0, \ldots, \boldsymbol{v}_{n_v-1}$ are organised in the column matrices, then there exists a matrix R, such that

$$V = RV',$$

where

$$V' = \begin{bmatrix} \boldsymbol{v}'_0 \\ \vdots \\ \boldsymbol{v}'_{n'_f} \end{bmatrix}, \qquad V = \begin{bmatrix} \boldsymbol{v}_0 \\ \vdots \\ \boldsymbol{v}_{n_f} \end{bmatrix}.$$

The matrix R is usually sparse (e.g. each row of the doubling matrix contains one coefficient equal to 1 and zeros, and the numbers of nonzero coefficients in rows of the averaging matrix are the degrees of the facets), therefore the irregular sparse matrix representation is used (see Section 3.6). The refinement matrices may be used to compute directly the vertices of the new mesh (which is useful e.g. with a multiresolution representation of a surface) and to construct a preconditioner used by procedures of optimization of the surface shape, in the libg2blending library.

```
boolean bsm_DoublingNum ( int inv, BSMvertex *imv, int *imvhei,
                        int inhe, BSMhalfedge *imhe,
                        int infac, BSMfacet *imfac, int *imfhei,
                        int *onv, int *onhe, int *onfac );
boolean bsm_Doublingd ( int spdimen,
               int inv, BSMvertex *imv, int *imvhei, double *iptc,
               int inhe, BSMhalfedge *imhe,
               int infac, BSMfacet *imfac, int *imfhei,
               int *onv, BSMvertex *omv, int *omvhei, double *optc,
               int *onhe, BSMhalfedge *omhe,
               int *onfac, BSMfacet *omfac, int *omfhei );
```

The procedure bsm_DoublingNum computes the numbers of vertices, halfedges and facets, which will be produced by doubling. It may (and should) be used before calling the doubling procedure, to tell the application, how long arrays to allocate. The procedure returns true if the computation was successful, and false if the mesh representation is inconsistent or there was insufficient scratch memory for the computation.

The procedure bsm_Doublingd implements the doubling operation. The value returned is true after success, or false after failure, due to the inconsistency of the input mesh representation or to insufficient scratch memory.

Parameters: spdimen—dimension of the space, in which the mesh vertices reside, i.e. number of coordinates of each vertex (usually 3, but not necessarily), inv, inhe, infac—numbers of the input mesh vertices, halfedges and facets respectively; imv—array of input mesh vertices, imvhei—array with lists of indices of halfedges with origins at input mesh vertices, iptc—array with coordinates of the input mesh vertices, imhe—array with input mesh halfedges, imfac—array with input mesh facets, imfhei—indices of halfedges for the input mesh facets.

The parameters onv, onhe, onfac point to the variables, in which the numbers of vertices, halfedges and facets are stored. The arrays omv, omvhei, optc, omhe, omfac and omfhei must be allocated by the caller, which stores the mesh representation there.

```
int bsm_DoublingMatSize ( int inv, BSMvertex *imv, int *imvhei,
                        int inhe, BSMhalfedge *imhe,
```



```
                        int infac, BSMfacet *imfac, int *imfhei );
boolean bsm_DoublingMatd ( int inv, BSMvertex *imv, int *imvhei,
                        int inhe, BSMhalfedge *imhe,
                        int infac, BSMfacet *imfac, int *imfhei,
                        int *onv, BSMvertex *omv, int *omvhei,
                        int *onhe, BSMhalfedge *omhe,
                        int *onfac, BSMfacet *omfac, int *omfhei,
                        int *ndmat, index2 *dmi, double *dmc );
```

The procedure bsm_DoublingMatSize computes the number of nonzero coefficients of the doubling matrix, for a mesh represented by the parameters. This number is returned as the function value (and it is non-positive if the procedure failed, because of the input mesh inconsistency or insufficient scratch memory).

The procedure bsm_DoublingMatd is an implementation of doubling, which matches precisely the operation of the bsm_Doublingd procedure (i.e. it produces the same ordering of the output mesh vertices, halfedges and facets). Instead of computing the positions of the output mesh vertices, the procedure produces the doubling matrix (represented as a sparse matrix).

Parameters: inv, imv, imvhei, inhe, imhe, infac, imfac, imfhei—represent the input mesh, see the description of the bsm_Doublingd procedure.

The output mesh representation is stored in the variables *onv, *onhe, *onfac and in the arrays omv, omvhei, omhe, omfac and omfhei, just like in the procedure bsm_Doublingd.

The doubling matrix representation is stored in the variable *ndmat—number of nonzero coefficients, the array dmi—distributions of nonzero coefficients, and dmc—the actual coefficients. All nonzero coefficients of the doubling matrix are equal to 1.

The doubling matrix has m rows, where m is the number of the output mesh vertices, and n columns, where n is the number of the input mesh vertices.

The value returned by bsm_DoublingMatd is true in case of success, and false in case of failure, caused by the input data inconsistency or by insufficient scratch memory.

```
boolean bsm_AveragingNum ( int inv, BSMvertex *imv, int *imvhei,
                        int inhe, BSMhalfedge *imhe,
                        int infac, BSMfacet *imfac, int *imfhei,
                        int *onv, int *onhe, int *onfac );
boolean bsm_Averagingd ( int spdimen,
               int inv, BSMvertex *imv, int *imvhei, double *iptc,
               int inhe, BSMhalfedge *imhe,
               int infac, BSMfacet *imfac, int *imfhei,
               int *onv, BSMvertex *omv, int *omvhei, double
               *optc,
```

```
                int *onhe, BSMhalfedge *omhe,
                int *onfac, BSMfacet *omfac, int *omfhei );
```

The procedure bsm_AveragingNum computes the numbers of vertices, halfedges and facets, which will be produced by averaging. It may (and should) be used before calling the averaging procedure, to tell the application, how long arrays to allocate. The procedure returns true if the computation was successful, and false if the mesh representation is inconsistent or there was insufficient scratch memory for the computation.

The procedure bsm_Averagingd implements the averaging operation. The value returned is true after success, or false after failure, due to the inconsistency of the input mesh representation or to insufficient scratch memory.

Parameters: spdimen—dimension of the space, in which the mesh vertices reside, i.e. number of coordinates of each vertex (usually 3, but not necessarily), inv, inhe, infac—numbers of the input mesh vertices, halfedges and facets respectively; imv—array of input mesh vertices, imvhei—array with lists of indices of halfedges with origins at input mesh vertices, iptc—array with coordinates of the input mesh vertices, imhe—array with input mesh halfedges, imfac—array with input mesh facets, imfhei—indices of halfedges for the input mesh facets.

The parameters onv, onhe, onfac point to the variables, in which the numbers of vertices, halfedges and facets are stored. The arrays omv, omvhei, optc, omhe, omfac and omfhei must be allocated by the caller, which stores the mesh representation there.

```
int bsm_AveragingMatSize ( int inv, BSMvertex *imv, int *imvhei,
                           int inhe, BSMhalfedge *imhe,
                           int infac, BSMfacet *imfac, int *imfhei );
boolean bsm_AveragingMatd ( int inv, BSMvertex *imv, int *imvhei,
                            int inhe, BSMhalfedge *imhe,
                            int infac, BSMfacet *imfac, int *imfhei,
                            int *onv, BSMvertex *omv, int *omvhei,
                            int *onhe, BSMhalfedge *omhe,
                            int *onfac, BSMfacet *omfac, int *omfhei,
                            int *namat, index2 *ami, double *amc );
```

The procedure bsm_AveragingMatSize computes the number of nonzero coefficients of the averaging matrix, for a mesh represented by the parameters. This number is returned as the function value (and it is non-positive if the procedure failed, because of the input mesh inconsistency or insufficient scratch memory).

The procedure bsm_AveragingMatd is an implementation of averaging, which matches precisely the operation of the bsm_Averagingd procedure (i.e. it produces the same ordering of the output mesh vertices, halfedges and facets). Instead of computing the positions of the output mesh vertices, the procedure produces the

averaging matrix (represented as a sparse matrix).

Parameters: inv, imv, imvhei, inhe, imhe, infac, imfac, imfhei—represent the input mesh, see the description of the bsm_Averagingd procedure.

The output mesh representation is stored in the variables *onv, *onhe, *onfac and in the arrays omv, omvhei, omhe, omfac and omfhei, just like in the procedure bsm_Averagingd.

The averaging matrix representation is stored in the variable *ndmat—number of nonzero coefficients, the array dmi—distributions of nonzero coefficients, and dmc—the actual coefficients. Each nonzero coefficient of the averaging matrix is a fraction 1/k, where k is the degree of an input mesh facet.

The averaging matrix has m rows, where m is the number of the output mesh vertices, and n columns, where n is the number of the input mesh vertices.

The value returned by bsm_AveragingMatd is true in case of success, and false in case of failure, caused by the input data inconsistency or by insufficient scratch memory.

```
boolean bsm_RefineBSMeshd ( int spdimen, int degree,
          int inv, BSMvertex *imv, int *imvhei, double *iptc,
          int inhe, BSMhalfedge *imhe,
          int infac, BSMfacet *imfac, int *imfhei,
          int *onv, BSMvertex **omv, int **omvhei, double **optc,
          int *onhe, BSMhalfedge **omhe,
          int *onfac, BSMfacet **omfac, int **omfhei );
```

The bsm_RefineBSMeshd procedure is an implementation of the mesh refinement operation, which is the composition of doubling and n averaging steps; the number n is specified by the degree parameter. The computation is done by a call to the bsm_Doublingd procedure, followed by n calls to the bsm_Averagingd procedure.

There is no procedure to compute the lengths of arrays for the representation of the output mesh, as it is impossible to find these numbers without the actual representations of all-but-last intermediate meshes. Therefore the procedure allocates the suitable arrays using malloc (wrapped in the PKV_MALLOC macro, see Section 2.10).

Parameters: spdimen—dimension of the space, in which the mesh vertices reside, inv, imv, imvhei, iptc, inh, imh, infac, imfac, imfhei—representation of the input mesh.

The parameters onv, onhe, onfac point to the variables, to which the numbers of vertices, halfedges and facets of the output mesh are assigned. The parameters omv, omvhei, optc, omhe, omfac and omfhei point to the variables, to which the addresses of arrays allocated by the procedure bsm_RefineBSMeshd are assigned. The contents of these arrays is the representation of the output (refined) mesh.

The procedure returns true if the computation was successful, or false in case of failure, caused by insufficient memory or by failure of the doubling or averaging

procedure.

```
boolean bsm_RefinementMatd ( int degree,
                          int inv, BSMvertex *imv, int *imvhei,
                          int inhe, BSMhalfedge *imhe,
                          int infac, BSMfacet *imfac, int *imfhei,
                          int *onv, BSMvertex **omv, int **omvhei,
                          int *onhe, BSMhalfedge **omhe,
                          int *onfac, BSMfacet **omfac, int **omfhei,
                          int *nrmat, index2 **rmi, double **rmc );
```

The procedure bsm_RefinementMatd implements mesh refinement (doubling followed by n averaging operations), but instead of computing the vertices of the output mesh, it produces the refinement matrix. This is done by a call to bsm_DoublingMatd followed by n calls to bsm_AveragingMatd. The doubling and averaging matrices are multiplied by procedures described in Section 3.6.

The parameters with the same names are the same as in the bsm_RefineBSMeshd procedure; instead of arrays with coordinates of the vertices of the input and output mesh there are the following three output parameters: nrmat, which points to the variable, to which the number of nonzero coefficients is assigned, and rmi and rmc, which point to pointers to the arrays with the distribution of nonzero coefficients of the refinement matrix and the actual coefficients (all arrays, whose addresses are assigned to the variables pointed by the parameters, are allocated with PKV_MALLOC by bsm_RefinementMatd, which also computes their lengths).

The nonzero coefficients of the refinement matrix are positive, their sum in each row is 1. The number of rows of the refinement matrix is the number of vertices of the output mesh, and the number of columns is the number of vertices of the input mesh.

The procedure returns true in case of success, and false in case of failure, caused by inconsistency of the input data or by insufficient memory.

10.3 Eulerian and non-Eulerian operations

The procedures described in this section may be used to edit the meshes, which may produce meshes with the same or different topology. These procedures may be invoked by an interactive program, which allows the user e.g. to point a facet and then let the program delete this facet or to double its edges. The repertoire of these operations is rather small, and it is my intention to extend it—when I find enough time.

```
void bsm_MergeMeshesd ( int spdimen,
                int nv1, BSMvertex *mv1, int *mvhei1, double *vpc1,
                int nhe1, BSMhalfedge *mhe1,
                int nfac1, BSMfacet *mfac1, int *mfhei1,
                int nv2, BSMvertex *mv2, int *mvhei2, double *vpc2,
                int nhe2, BSMhalfedge *mhe2,
                int nfac2, BSMfacet *mfac2, int *mfhei2,
                int *onv, BSMvertex *omv, int *omvhei, double *ovpc,
                int *onhe, BSMhalfedge *omhe,
                int *onfac, BSMfacet *omfac, int *omfhei );
```

The procedure bsm_MergeMeshesd makes a mesh, which is a sum of two meshes. The numbers of vertices, halfedges and facets of the resulting mesh are respectively sums of numbers of the vertices, halfedges and facets of the two meshes. Before calling this procedure, an application must allocate suitable arrays for the result.

The result consists of a copy of the first mesh, and the "shifted" copy of the second mesh, whose vertices, halfedges and facets obtain new numbers.

Parameters: spdimen—dimension of the space (i.e. the number of coordinates of each vertex),
nv1, mv1, mvhei1, vpc1, nhe1, mhe1, nfac1, mfac1, mfhei1—representation of the first mesh,
nv2, mv2, mvhei2, vpc2, nhe2, mhe2, nfac2, mfac2, mfhei2—representation of the second mesh.

The output parameters onv, omv, omvhei, ovpc, onhe, omhe, onfac, omfac, omfhei point to the variables, to which the numbers of vertices, halfedges and facets are assigned and to arrays, in which the vertices, halfedges and facets are to be stored.

```
boolean bsm_RemoveFacetNum ( int inv, BSMvertex *imv, int *imvhei,
                          int inhe, BSMhalfedge *imhe,
                          int infac, BSMfacet *imfac, int *imfhei,
                          int nfr,
                          int *onv, int *onhe, int *onfac );
boolean bsm_RemoveFacetd ( int spdimen,
```

```
                int inv, BSMvertex *imv, int *imvhei, double *iptc,
                int inhe, BSMhalfedge *imhe,
                int infac, BSMfacet *imfac, int *imfhei,
                int nfr,
                int *onv, BSMvertex *omv, int *omvhei, double *optc,
                int *onhe, BSMhalfedge *omhe,
                int *onfac, BSMfacet *omfac, int *omfhei );
```

The procedures above may be used to remove a facet from a mesh. Removing a facet involves removing all its halfedges and all vertices, which do not belong to any other halfedges, and renumbering the remaining vertices, halfedges and facets.

The procedure `bsm_RemoveFacetNum` computes the numbers of remaining vertices, halfedges and facets, which are necessary to allocate arrays for the result.

The procedure `bsm_RemoveFacetd` removes the facet, i.e. it produces a new mesh, without the indicated facet.

Parameters: `spdimen`—dimension of the space, i.e. the number of coordinates of each vertex.
`inv`, `imv`, `imvhei`, `iptc`, `inhe`, `imhe`, `infac`, `imfac`, `imfhei`—representation of the input mesh,
`nfr`—number of the facet to remove (must be between 0 and $n_f - 1$).

The output parameters `onv`, `omv`, `omvhei`, `optc`, `onhe`, `omhe`, `onfac`, `omfac`, `omfhei` point to the variables, to which the numbers of vertices, halfedges and facets are assigned and to arrays, in which the vertices, halfedges and facets are to be stored.

```
void bsm_FacetEdgeDoublingNum ( int inv, BSMvertex *imv, int *imvhei,
                                int inhe, BSMhalfedge *imhe,
                                int infac, BSMfacet *imfac, int *imfhei,
                                int fn,
                                int *onv, int *onhe, int *onfac );
boolean bsm_FacetEdgeDoublingd ( int spdimen,
                int inv, BSMvertex *imv, int *imvhei, double *iptc,
                int inhe, BSMhalfedge *imhe,
                int infac, BSMfacet *imfac, int *imfhei,
                int fn,
                int *onv, BSMvertex *omv, int *omvhei,
                double *optc,
                int *onhe, BSMhalfedge *omhe,
                int *onfac, BSMfacet *omfac, int *omfhei );
```

Doubling edges of a facet is an Eulerian operation, which replaces each edge of the facet with a quadrangular facet degenerated to a line segment. Each vertex of the facet is replaced by two vertices (at the same position). After this operation one

can "extrude" the new facets, i.e. move the vertices of the facet, whose edges have been doubled.

The procedure `bsm_FacetEdgeDoublingNum` computes the numbers of vertices, halfedges and facets of the mesh, which is the result of this operation. It should be called in order to allocate suitable arrays for the result.

The procedure `bsm_FacetEdgeDoublingd` doubles the edges of the indicated facet, i.e. it produces the new mesh, being the result of this operation.

Parameters: `spdimen`—dimension of the space, i.e. the number of coordinates of each vertex.
`inv`, `imv`, `imvhei`, `iptc`, `inhe`, `imhe`, `infac`, `imfac`, `imfhei`—representation of the input mesh,
`fn`—number of the facet, whose edges are to be doubled (must be between 0 and $n_f - 1$).

The output parameters `onv`, `omv`, `omvhei`, `optc`, `onhe`, `omhe`, `onfac`, `omfac`, `omfhei` point to the variables, to which the numbers of vertices, halfedges and facets are assigned and to arrays, in which the vertices, halfedges and facets are to be stored.

```
void bsm_RemoveVertexNum ( int inv, BSMvertex *imv, int *imvhei,
                           int inhe, BSMhalfedge *imhe,
                           int infac, BSMfacet *imfac, int *imfhei,
                           int nvr,
                           int *onv, int *onhe, int *onfac );
boolean bsm_RemoveVertexd ( int spdimen,
                int inv, BSMvertex *imv, int *imvhei, double *iptc,
                int inhe, BSMhalfedge *imhe,
                int infac, BSMfacet *imfac, int *imfhei,
                int nvr,
                int *onv, BSMvertex *omv, int *omvhei, double *optc,
                int *onhe, BSMhalfedge *omhe,
                int *onfac, BSMfacet *omfac, int *omfhei );
```

Removing a vertex causes removing all halfedges incident with this vertex and all facets made of these halfedges. It may also cause removal of other vertices, if there are some, incident only with the halfedges to be removed.

The procedure `bsm_RemoveVertexNum` computes the number of vertices, halfedges and facets remaining in the mesh. It should be called by an application in order to allocate suitable arrays for the result.

Parameters: `spdimen`—dimension of the space, i.e. the number of coordinates of each vertex.
`inv`, `imv`, `imvhei`, `iptc`, `inhe`, `imhe`, `infac`, `imfac`, `imfhei`—representation of the input mesh,
`nvr`—number of the vertex to remove (must be between 0 and $n_v - 1$).

The output parameters onv, omv, omvhei, optc, onhe, omhe, onfac, omfac, omfhei point to the variables, to which the numbers of vertices, halfedges and facets are assigned and to arrays, in which the vertices, halfedges and facets are to be stored.

```
void bsm_ContractEdgeNum ( int inv, BSMvertex *imv, int *imvhei,
                           int inhe, BSMhalfedge *imhe,
                           int infac, BSMfacet *imfac, int *imfhei,
                           int nche,
                           int *onv, int *onhe, int *onfac );
int bsm_ContractEdged ( int spdimen,
               int inv, BSMvertex *imv, int *imvhei, double *iptc,
               int inhe, BSMhalfedge *imhe,
               int infac, BSMfacet *imfac, int *imfhei,
               int nche,
               int *onv, BSMvertex *omv, int *omvhei, double *optc,
               int *onhe, BSMhalfedge *omhe,
               int *onfac, BSMfacet *omfac, int *omfhei );
```

Edge contraction is an Eulerian operation, which removes an edge, and replaces its two vertices by one vertex. It may delete one or two facets adjacent to this edge, if any of the two facets is a triangle.

The procedure bsm_ContractEdgeNum computes the numbers of vertices, halfedges and facets of the mesh being the result of edge contraction. It should be called by an application in order to allocate suitable arrays of this result.

The procedure bsm_ContractEdged performs the contracting an edge, i.e. it produces a mesh, which is the result of this operation.

Parameters: spdimen—dimension of the space, i.e. the number of coordinates of each vertex.
inv, imv, imvhei, iptc, inhe, imhe, infac, imfac, imfhei—representation of the input mesh,
nche—number of one of the halfedges, which represent the edge to be contracted (must be between 0 and $n_h - 1$).

The output parameters onv, omv, omvhei, optc, onhe, omhe, onfac, omfac, omfhei point to the variables, to which the numbers of vertices, halfedges and facets are assigned and to arrays, in which the vertices, halfedges and facets are to be stored.

The value returned by bsm_ContractEdged is the number of the vertex, which replaced the two vertices of the edge, or -1 in case of failure.


```
int bsm_HalfedgeLoopLength ( int nv, BSMvertex *mv, int *mvhei,
                             int nhe, BSMhalfedge *mhe,
                             int he );
```

The procedure bsm_HalfedgeLoopLength counts the boundary edges making a closed polyline (boundary of the mesh). The procedure bsm_GlueHalfedgeLoopsd, described below, may join into pairs the halfedges of two such polylines, which have the same number of edges.

Parameters nv, mv, mvhei, nhe, mhe represent the vertices and halfedges of the mesh (the algorithm implemented by this procedure does not need facets).

The parameter he is the number of a halfedge, which represents a boundary edge (it is necessary to indicate, which closed polyline is of interest, as there may be more than one).

```
boolean bsm_GlueHalfedgeLoopsd ( int spdimen,
                    int inv, BSMvertex *imv, int *imvhei, double *ivc,
                    int inhe, BSMhalfedge *imhe,
                    int infac, BSMfacet *imfac, int *imfhei,
                    int he1, int he2,
                    int *onv, BSMvertex *omv, int *omvhei,
                    double *ovc,
                    int *onhe, BSMhalfedge *omhe,
                    int *onfac, BSMfacet *omfac, int *omfhei );
```

The procedure bsm_GlueHalfedgeLoopsd joins into pairs the halfedges representing boundary edges of two closed polylines, which have the same number of edges. The result is a new mesh, whose boundary consists of two polylines less. The number of facets remains unchanged, but the number of vertices is smaller, as joining the halfedges causes identification of their vertices.

The orientation of the mesh is preserved, i.e. it is possible to obtain a torus, but it is impossible to obtain the Klein bottle (which, as well as all non-orientable surfaces, is prohibited by the procedures in this library).

Parameters: spdimen—dimension of the space, i.e. the number of coordinates of each vertex.
inv, imv, imvhei, ivc, inhe, imhe, infac, imfac, imfhei—representation of the input mesh.

The parameters he1 and he2 are numbers of two halfedges to be joined (these halfedges will represent a single edge in the resulting mesh). The two halfedges must have no pairs in the input mesh, and they must belong to two different closed polylines made of boundary edges. The number of edges in these polylines must be the same. The matching of halfedges in these polylines is determined by these parameters and the orientation conditions, which must be satisfied by the mesh.

—number of one of the halfedges, which represent the edge to be contracted (must be between 0 and $n_h - 1$).

The output parameters `onv`, `omv`, `omvhei`, `ovc`, `onhe`, `omhe`, `onfac`, `omfac`, `omfhei` point to the variables, to which the numbers of vertices, halfedges and facets are assigned and to arrays, in which the vertices, halfedges and facets are to be stored.

Tre value returned is `true` in case of success or `false` in case of failure, which may be caused by invalid data (i.e. the polylines with different numbers of edges) or by insufficient scratch memory pool.

10.4 Extracting regular and special subnets

The procedures described in this section are useful when the mesh represents a spline surface and it is necessary to convert this representation so as to find polynomial patches (in B-spline representation, which may be converted to the Bézier form), the surface consists of. An extensive use of these procedures is made by the procedures of shape optimization in the `libg2blending` library (see Chapter 12).

```
boolean bsm_FindRegularSubnets ( int nv, BSMvertex *mv, int *mvhei,
                      int nhe, BSMhalfedge *mhe,
                      int nfac, BSMfacet *mfac, int *mfhei,
                      int d, void *usrptr,
                      void (*output)( int d, int *vertnum, int *mtab,
                                      void *usrptr ) );
```

The procedure `bsm_FindRegularSubnets` searches the mesh in order to find all rectangular (square) subnets made of d^2 vertices, $(d-1)^2$ facets and $2d(d-1)$ edges. Such a net is a B-spline representation (with uniform knots) of a polynomial patch of degree $(d-1, d-1)$. This representation may be converted to the Bézier representation, which may be used to obtain the picture of the patch.

The parameters `nv`, `mv`, `mvhei`, `nhe`, `mhe`, `nfac`, `mfac`, `mfhei` are the usual representation of the mesh (no array with vertex positions is needed).

The value of the parameter `d` is the number d, which determines the size of the subnets to be found.

The parameter `usrptr` is a pointer to an arbitrary data structure, which will be passed to the subprogram pointed by the parameter `output`.

The parameter `output` points to the subprogram, whose purpose is to do something with the subnets found in the mesh. This subprogram is called after finding each subnet, with the parameter `d`, whose value is the number d, two arrays with the information about the subnet and the pointer to the data structure given by the caller.

The array `vertnum` contains d^2 numbers, which are identifiers of the vertices of the subnet. This array should be seen as a square array with d columns and d rows

of the vertices (the numbers of vertices neighbouring in a column are neighbours in the array, the numbers of neighbours in a row take positions at a distance d in the array).

The array `mtab` contains $(2d-1)^2$ numbers, which are identifiers of the vertices, halfedges and facets. It is also a square array with rows and columns of length $2d-1$, numbered from 0. Let the index entry be $(2d-1)i+j$; if i and j are both even, the array entry contains a vertex number. If both numbers i and j are odd, then the array entry contains a facet number. If $i+j$ is odd, then the contents of the array entry is a halfedge number. In this way the index of a facet is surrounded by the indices of halfedges (either belonging to this facet or the facet sharing an edge) and vertices.

The return value is `true` if the computation is successful, or `false` in case of error, caused by invalid data or insufficient scratch memory.

```
boolean bsm_FindSpecialVSubnets ( int nv, BSMvertex *mv, int *mvhei,
                      int nhe, BSMhalfedge *mhe,
                      int nfac, BSMfacet *mfac, int *mfhei,
                      int d, void *usrptr,
                      void (*output)( int d, int k, int *vertnum,
                                      int *mtab, void *usrptr ) );
```

The procedure `bsm_FindSpecialVSubnets` searches the mesh described by the parameters in order to find all special elements, having the form of an internal vertex of degree $k \neq 4$ surrounded by k regular (square) subnets, each consisting of d^2 quadrangular facets. Such a subnet is called here a **Sabin net of radius** d.

If a mesh has special elements, then the spline surface represented by this mesh has polygonal (k-sided) holes, which may be filled in some special way. The procedure `bsm_FindSpecialVSubnets` may be used to find such special elements in order to construct the surfaces filling the holes.

The parameters `nv`, `mv`, `mvhei`, `nhe`, `mhe`, `nfac`, `mfac`, `mfhei` are the usual representation of the mesh (no array with vertex positions is needed).

The value of the parameter `d` is the number d, which determines the size of the Sabin nets to be found.

The parameter `usrptr` is a pointer to an arbitrary data structure, which will be passed to the subprogram pointed by the parameter `output`.

The parameter `output` points to the subprogram, whose purpose is to do something with the subnets found in the mesh. This subprogram is called after finding each subnet, with the parameter `d`, whose value is the number d, the number k, which is the degree of the central vertex of the subnet just found, and two arrays with the information about the subnet and the pointer to the data structure given by the caller.

The array `vertnum` contains $1 + kd(d+1)$ indices of the vertices of the subnet. First comes the index of the central vertex, then the surrounding vertices.

The array mtab contains $k(2d+1)^2$ numbers, which are the indices of the vertices, halfedges and facets of k square subnets around the special vertex.

The value returned is true in case of success, or false in case of failure caused by a data error or by insufficient scratch memory.

```
boolean bsm_FindSpecialFSubnets ( int nv, BSMvertex *mv, int *mvhei,
                        int nhe, BSMhalfedge *mhe,
                        int nfac, BSMfacet *mfac, int *mfhei,
                        int d, void *usrptr,
                        void (*output)( int d, int k, int *vertnum,
                                        int *mtab, void *usrptr ) );
```

Another type of a special element in a mesh is a non-quadrangular facet. The procedure bsm_FindSpecialFSubnets searches a mesh in order to find all such facets surrounded by $kd(d+1)$ quadrangular facets, forming a Sabin net of the second type (with a special facet of degree $k \neq 4$) of radius d. Such a special element also leaves a k-sided hole in the surface represented by the mesh, and special methods are needed to fill such holes.

The parameters nv, mv, mvhei, nhe, mhe, nfac, mfac, mfhei are the usual representation of the mesh (no array with vertex positions is needed).

The value of the parameter d is the number d, which determines the size of the Sabin nets to be found.

The parameter usrptr is a pointer to an arbitrary data structure, which will be passed to the subprogram pointed by the parameter output.

The parameter output points to the subprogram, whose purpose is to do something with the subnets found in the mesh. This subprogram is called after finding each subnet, with the parameter d, whose value is the number d, the number k, which is the degree of the central facet of the subnet just found, and two arrays with the information about the subnet and the pointer to the data structure given by the caller.

The array vertnum contains $k(d+1)^2$ indices of the vertices of the subnet.

The array mtab contains $k(2d+1)(2d+3)$ numbers, which are the indices of the vertices, halfedges and facets of k rectangular subnets around the special facet.

The value returned is true in case of success, or false in case of failure caused by a data error or by insufficient scratch memory.

The procedures described below make lists of special elements of a mesh, or more precisely, they search the mesh in order to find the Sabin nets of the first and second type, and store their identifiers and identifiers of their vertices in arrays.

```
typedef struct {
   byte  el_type;
   byte  degree;
```



```
   byte  snet_rad;
   short snet_nvert;
   int   first_snet_vertex;
 } bsm_special_el;

typedef struct {
   int           nspecials;
   int           nspvert;
   int           nextravert;
   bsm_special_el *spel;
   int           *spvert;
 } bsm_special_elem_list;
```

```
boolean bsm_CountSpecialVSubnets ( int nv, BSMvertex *mv, int *mvhei,
                                   int nhe, BSMhalfedge *mhe,
                                   int nfac, BSMfacet *mfac, int *mfhei,
                                   byte snet_rad,
                                   int *nspecials, int *nspvert );
boolean bsm_FindSpecialVSubnetList (
                                  int nv, BSMvertex *mv, int *mvhei,
                                  int nhe, BSMhalfedge *mhe,
                                  int nfac, BSMfacet *mfac, int *mfhei,
                                  byte snet_rad,
                                  boolean append,
                                  bsm_special_elem_list *list );
```

The procedure bsm_CountSpecialVSubnets searches the mesh in order to count the Sabin net of the first type (with the central vertex of degree $k \neq 4$) of radius d and their vertices. One should use this procedure before calling bsm_FindSpecialVSubnetList in order to allocate the arrays for the list of the special elements. The addresses of the first elements of those arrays must be assigned to the fields spel and spvert of the data structure of type bsm_special_elem_list, passed to bsm_FindSpecialVSubnetList with use of the parameetr list.

The procedure bsm_FindSpecialVSubnetList searches the mesh and finds all Sabin nets of the first type of radius d =snet_rad. The information about the Sabin nets found is stored in the arrays pointed by the fields spel and spvert of the structure pointed by the parameter list.

The parameters nv, mv, mvhei, nhe, mhe, nfac, mfac, mfhei are the usual representation of the mesh (no array with vertex positions is needed).

The return value true signals a success, and false indicates a failure of the computation.

```
boolean bsm_CountSpecialFSubnets ( int nv, BSMvertex *mv, int *mvhei,
                                   int nhe, BSMhalfedge *mhe,
                                   int nfac, BSMfacet *mfac, int *mfhei,
                                   byte snet_rad,
                                   int *nspecials, int *nspvert );
boolean bsm_FindSpecialFSubnetLists (
                                int nv, BSMvertex *mv, int *mvhei,
                                int nhe, BSMhalfedge *mhe,
                                int nfac, BSMfacet *mfac, int *mfhei,
                                boolean append,
                                byte snet_rad,
                                bsm_special_elem_list *list );
```

The procedure `bsm_CountSpecialFSubnets` searches the mesh in order to count the Sabin net of the second type (with the central facet of degree $k \neq 4$) of radius `d` and their vertices. One should use this procedure before calling `bsm_FindSpecialFSubnetList` in order to allocate the arrays for the list of the special elements. The addresses of the first elements of those arrays must be assigned to the fields `spel` and `spvert` of the data structure of type `bsm_special_elem_list`, passed to `bsm_FindSpecialFSubnetList` with use of the parameetr `list`.

The procedure `bsm_FindSpecialFSubnetList` searches the mesh and finds all Sabin nets of the second type of radius `d` =`snet_rad`. The information about the Sabin nets found is stored in the arrays pointed by the fields `spel` and `spvert` of the structure pointed by the parameter `list`.

The parameters `nv`, `mv`, `mvhei`, `nhe`, `mhe`, `nfac`, `mfac`, `mfhei` are the usual representation of the mesh (no array with vertex positions is needed).

The return value `true` signals a success, and `false` indicates a failure of the computation.

10.5 Other procedures

```
void bsm_TagBoundaryZoneVertices ( int nv, BSMvertex *mv, int *mvhei,
                                   int nhe, BSMhalfedge *mhe,
                                   char d, char *vtag );
```

The purpose of the procedure `bsm_TagBoundaryZoneVertices` is to find all vertices, whose distance from the mesh boundary is less than a number `d`. The boundary vertices (incident with halfedges having no twins) are at the distance 0 from the boundary, and the distance between two different vertices is defined as the number of edges of the shortest path between them. If the mesh represents a spline surface made of patches of degree (d, d), then the vertices, whose distance from the mesh

boundary is less than `d` determine the boundary of the surface (and cross derivatives of the surface up to the order $d-1$; if $d = 2$ then this determines the tangent plane and curvature at each point of the boundary).

The procedure, for each vertex, stores in the array `vtag` the smaller of the two numbers: `d` or the distance of the vertex from the boundary.

```
boolean bsm_FindVertexDistances1 ( int nv, BSMvertex *mv, int *mvhei,
                                   int nhe, BSMhalfedge *mhe,
                                   int nfac, BSMfacet *mfac, int *mfhei,
                                   int v, int *dist );
boolean bsm_FindVertexDistances2 ( int nv, BSMvertex *mv, int *mvhei,
                                   int nhe, BSMhalfedge *mhe,
                                   int nfac, BSMfacet *mfac, int *mfhei,
                                   int v, int *dist );
```

The two procedures above find distances of all vertices of the mesh from a given vertex. The distances are defined using two metrics, defined as follows:

For two vertices, v_1 and v_2, which are end points of an edge, the metric ρ_1 takes the value 1.

For two vertices, v_1 and v_2, which belong to one facet, the metric ρ_2 takes the value 1.

Both metrics are defined as the maximal functions defined in the set of pairs of the vertices of a mesh, which satisfy the conditions above and the triangle's inequality.

For each vertex v_i, the procedures above store in the array `dist` the value of the metric $\rho_1(v, v_i)$ or $\rho_2(v, v_i)$ respectively. If this value is infinite (which is possible, if the mesh is not connected), then the number of vertices n_v is stored.

11. The libg1blending library

This chapter is waiting until I have enough time to write it. A great part of this library has been written by Mateusz Markowski.

```
boolean g1bl_SetupBiharmAMatrixf ( int lastknotu, int lastknotv,
                 int *n, int **prof, float **Amat, float ***arow );
boolean g1bl_SetupBiharmRHSf ( int lastknotu, int lastknotv,
                      int spdimen, int pitch, const float *cpoints,
                      float *rhs );
```

```
int g1bl_NiSize ( int nkn );
int g1bl_NijSize ( int nkn );
int g1bl_MijSize ( int nkn );
```

```
void g1bl_TabNid ( int nkn, double *bf, double *dbf, double *ddbf,
                 double *Nitab );
void g1bl_TabNijd ( int nkn, double *bf, double *dbf, double *ddbf,
                  double *Nijtab );
double g1bl_UFuncd ( int nkn, const double *qcoeff, double *Nitab,
               int lastknotu, int lastknotv, int pitch, point3d *cp,
               char *dirty,
               double tC, double *ftab );
```

```
double g1bl_QFuncd ( int nkn, const double *qcoeff, double *Nitab,
              int lastknotu, int lastknotv, int pitch, point3d *cp,
              char *dirty,
              double tC, double *ftab );
double g1bl_biharmFuncd ( int nkn, const double *qcoeff,
              double *Nitab,
              int lastknotu, int lastknotv, int pitch, point3d *cp,
              char *dirty,
              double tC, double *ftab );
```

```
void g1bl_UFuncGradd ( int nkn, const double *qcoeff, double *Nitab,
                     int lastknotu, int lastknotv,
                     int pitch, point3d *cp, char *dirty,
                     double tC, double *ftab, double *gtab,
                     double *func, double *grad );
void g1bl_UFuncGradHessiand ( int nkn, const double *qcoeff,
```



```
                 double *Nitab, double *Nijtab, double *Mijtab,
                 int lastknotu, int lastknotv,
                 int pitch, point3d *cp, char *dirty,
                 double tC, double *ftab, double *gtab,
                 double *htab, double *func, double *grad,
                 int hsize, const int *prof, double **hrows );
double g1bl_SurfNetDiameterSqd ( int lastknotu, int lastknotv,
                               int pitch, const point3d *cp );
```

```
boolean g1bl_InitBlSurfaceOptLMTd ( int lastknotu, int lastknotv,
                                 int pitch, point3d *cp,
                                 double C, double dO, double dM,
                                 int nkn1, int nkn2,
                                 void **data );
boolean g1bl_IterBlSurfaceOptLMTd ( void *data, boolean *finished );
void g1bl_OptLMTDeallocated ( void **data );
boolean g1bl_FindBlSurfaceLMTd ( int lastknotu, int lastknotv,
                               int pitch, point3d *cp,
                               double C, double dO, double dM,
                               int maxit, int nkn1, int nkn2 );
```

```
boolean g1bl_ClosedInitBlSurfaceConstrOptLMTd (
               int lastknotu, int lastknotv, int pitch, point3d *cp,
               int nconstr, double *constrmat, double *constrrhs,
               double C, double dO, double dM, int nkn1, int nkn2,
               void **data );
boolean g1bl_ClosedIterBlSurfaceConstrOptLMTd ( void *data,
               boolean *finished );
void g1bl_ClosedConstrOptLMTDeallocated ( void **data );
boolean g1bl_ClosedFindBlSurfaceConstrLMTd (
               int lastknotu, int lastknotv, int pitch, point3d *cp,
               int nconstr, double *constrmat, double *constrrhs,
               double C, double dO, double dM,
               int maxit, int nkn1, int nkn2 );
```

```
boolean g1bl_SetupULConstraintsd ( int lastknotu, int lastknotv,
                    int spdimen, int ppitch, double *cp,
                    int nucurv, double *ucknots,
                    int cpitch, double *uccp,
                    int *nconstr, double *cmat, double *crhs );

boolean g1bl_SetupUNLConstraintsd ( int lastknotu, int lastknotv,
```

```
                        int ppitch, point3d *cp,
                        int nucurv, double *ucknots,
                        int cpitch, point3d *uccp,
                        int *nconstr, double *cmat, double *crhs );

boolean g1bl_SetupClosedULConstraintsd (
                      int lastknotu, int lastknotv,
                      int spdimen, int ppitch, double *cp,
                      int nucurv, double *ucknots,
                      int cpitch, double *uccp,
                      int *nconstr, double *cmat, double *crhs );
```

```
boolean g1bl_SetupClosedUNLConstraintsd (
                      int lastknotu, int lastknotv,
                      int ppitch, point3d *cp,
                      int nucurv, double *ucknots,
                      int cpitch, point3d *uccp,
                      int *nconstr, double *cmat, double *crhs );


boolean g1bl_FuncTSQFd ( int nkn,
                int lastknotu, int lastknotv, int pitch, point3d *cp,
                double tC,
                double *fT, double *fS, double *fQ, double *fF );
```

12. The libg2blending library

The libg2blending library is made of procedures whose purpose is to optimise the shape of spline surfaces of class G^2. If the surface is represented by a parameterization $\mathbf{p}$, whose domain is Ω, then the simplest quality measure (i.e. badness measure, which grows with the surface undulations) is described by the following functional:

$$T(\mathbf{p}) = \int_\Omega \|\nabla\Delta\mathbf{p}\|_F^2 \, d\Omega. \tag{12.1}$$

The Euler-Lagrange equation for this functional is the homogeneous triharmonic equation

$$-\Delta^3\mathbf{p} = \mathbf{0}, \tag{12.2}$$

and the minimal surface of the functional T is found by solving this equation with the Dirichlet boundary condition

$$\begin{cases} \mathbf{p}|_{\partial\Omega} = \mathbf{q}|_{\partial\Omega}, \\ \dfrac{\partial\mathbf{p}}{\partial\mathbf{n}}\Big|_{\partial\Omega} = \dfrac{\partial\mathbf{q}}{\partial\mathbf{n}}\Big|_{\partial\Omega}, \\ \dfrac{\partial^2\mathbf{p}}{\partial\mathbf{n}^2}\Big|_{\partial\Omega} = \dfrac{\partial^2\mathbf{q}}{\partial\mathbf{n}^2}\Big|_{\partial\Omega}, \end{cases} \tag{12.3}$$

where $\mathbf{n}$ is the unit normal vector of the boundary $\partial\Omega$ of the domain Ω, and $\mathbf{q}$ is some fixed parameterization.

The functional T is actually a badness measure for a parameterization, and a good surface shape, if obtained by minimization of T, is only a side effect. On the other hand, the triharmonic equation is a linear differential equation, which is reduced by the finite element method (FEM) to a system of linear algebraic equations. The procedures of the libg2blending library allow one to use this criterion for bicubic B-spline patches with uniform knots. As only a direct numerical method of solving a system of linear equations (Cholesky's decomposition) is used, the number of control points of the patches must be limited. These procedures of finding triharmonic patches are described in Section 12.1.

The second criterion explicitly depends on the surface shape:

$$S(\mathbf{p}) = \int_{\mathcal{M}} \|\nabla_{\mathcal{M}} H\|_2^2 \, dM. \tag{12.4}$$

Finding a minimum of this functional is a nonlinear problem. There is an additional trouble, as any surface may be represented using various parameterizations, which leads to ill-posed numerical problems. This difficulty is solved in two ways. The

optimization criterion is modified by adding a regularization term, and a minimum of the following functional is searched

$$F(\mathbf{p}) = S(\mathbf{p}) + cQ(\mathbf{p}). \tag{12.5}$$

The presence of the term $cQ(\mathbf{p})$, where c is a positive constant,

$$Q(\mathbf{p}) = \int_\Omega \|P\nabla\Delta\mathbf{p}\|_F^2 \, d\mathbf{u}, \tag{12.6}$$

and P is the orthogonal projection of $\mathbb{R}^3$ onto the tangent plane of the surface $\mathcal{M}$ at the point $\mathbf{p}(\mathbf{u})$, where $\mathbf{u} \in \Omega$, discriminates parameterizations. The term $cQ(\mathbf{p})$ is a penalty imposed on undulations of curves of constant parameters of the parameterization $\mathbf{p}$. Minimization of the functional F is often a well-posed problem (that depends on the boundary conditions).

The presence of the regularization term affects the result (the projection P was introduced in order to decrease as much as possible that effect). It is possible to get rid of this term, by restricting the space in which the minimum is searched. This approach has been implemented for surfaces represented by meshes. The optimization procedures for the meshes are described in Section 12.3.

The algorithms, whose implementations are the procedures of this library, are described in the following publications:

[1] Kiciak P.: Bicubic B-spline blending patches with optimized shape, *Computer-Aided Design* 43 (2011), p. 133–144,

[2] Kiciak P.: Shape optimization of smooth surfaces of arbitrary topology, *IMProVE 2011 Conference Proceedings*, Venice, Italy, June 15–17, 2011,

[3] Kiciak P.: Spline surfaces of arbitrary topology with continuous curvature and optimized shape, *CAD-D-11-00233*, a paper submitted to *Computer-Aided Design*.

12.1 Triharmonic tensor product B-spline patches

Minimization of the functional T is done by solving linear equations, obtained by applying the finite element method approach to the triharmonic equation. Currently only rather small size problems may be solved in this way, as the method of solving linear equations, which may be used along with the procedures described below is the Cholesky's decomposition.

```
boolean g2bl_SetupTriharmAMatrixd ( int lastknotu, int lastknotv,
               int *n, int **prof, double **Amat, double ***arow );
boolean g2bl_SetupTriharmRHSd ( int lastknotu, int lastknotv,
               int spdimen, int pitch, const double *cpoints,
               double *rhs );
```

The procedures g2bl_SetupTriharmAMatrixd and g2bl_SetupTriharmRHSd set up respectively the matrix A and the right-hand side vector $\mathbf{b}$ of the system of equations, which is a discterized (with FEM) triharmonic equation for a tensor product bicubic B-spline patch with uniform knots.

Parameters: lastknotu and lastknotv are the numbers N and M, which determine the sequences $0,\dots,N$ and $0,\dots,M$ of equidistant knots, being parts of the patch representation. The domain Ω of the patch is the rectangle $[3,N-3]\times[3,M-3]$. The numbers N and M must be greater than 9.

The parameter n points to a variable, to which the number of equations is assigned; it is $(N-9)(M-9)$.

.....

```
boolean g2bl_SetupClosedTriharmAMatrixd (
               int lastknotu, int lastknotv,
               int *n, int **prof, double **Amat, double ***arow );
boolean g2bl_SetupClosedTriharmRHSd ( int lastknotu, int lastknotv,
               int spdimen, int pitch, const double *cpoints,
               double *rhs );
```


12.2 Tensor product patches optimized using a shape-dependent functional

12.2.1 Main procedures

```
boolean g2bl_InitBlSurfaceOptLMTd ( int lastknotu, int lastknotv,
                      int pitch, point3d *cp,
                      double C, double dO, double dM,
                      int nkn1, int nkn2,
                      void **data );
boolean g2bl_IterBlSurfaceOptLMTd ( void *data, boolean *finished );
void g2bl_OptLMTDeallocated ( void **data );
boolean g2bl_FindBlSurfaceLMTd ( int lastknotu, int lastknotv,
                      int pitch, point3d *cp,
                      double C, double dO, double dM,
                      int maxit, int nkn1, int nkn2 );
```

```
boolean g2bl_InitBlSurfaceConstrOptLMTd (
                     int lastknotu, int lastknotv, int pitch,
                     point3d *cp,
                     int nconstr, double *constrmat,
                     double *constrrhs,
                     double C, double dO, double dM,
                     int nkn1, int nkn2,
                     void **data );
boolean g2bl_IterBlSurfaceConstrOptLMTd ( void *data,
                     boolean *finished );
void g2bl_ConstrOptLMTDeallocated ( void **data );
boolean g2bl_FindBlSurfaceConstrLMTd (
                     int lastknotu, int lastknotv, int pitch,
                     point3d *cp,
                     int nconstr, double *constrmat,
                     double *constrrhs,
                     double C, double dO, double dM,
                     int maxit, int nkn1, int nkn2 );
```

```
boolean g2bl_ClosedInitBlSurfaceOptLMTd (
                     int lastknotu, int lastknotv, int pitch,
                     point3d *cp,
                     double C, double dO, double dM,
                     int nkn1, int nkn2,
```

```
                    void **data );
boolean g2bl_ClosedIterBlSurfaceOptLMTd ( void *data,
                    boolean *finished );
void g2bl_ClosedOptLMTDeallocated ( void **data );
boolean g2bl_ClosedFindBlSurfaceLMTd (
                    int lastknotu, int lastknotv, int pitch,
                    point3d *cp,
                    double C, double dO, double dM,
                    int maxit, int nkn1, int nkn2 );
```

```
boolean g2bl_ClosedInitBlSurfaceConstrOptLMTd (
                int lastknotu, int lastknotv, int pitch, point3d *cp,
                int nconstr, double *constrmat, double *constrrhs,
                double C, double dO, double dM, int nkn1, int nkn2,
                void **data );
boolean g2bl_ClosedIterBlSurfaceConstrOptLMTd ( void *data,
                boolean *finished );
void g2bl_ClosedConstrOptLMTDeallocated ( void **data );
boolean g2bl_ClosedFindBlSurfaceConstrLMTd (
                int lastknotu, int lastknotv, int pitch, point3d *cp,
                int nconstr, double *constrmat, double *constrrhs,
                double C, double dO, double dM,
                int maxit, int nkn1, int nkn2 );
```

12.2.2 Auxiliary procedures

```
int g2bl_NiSize ( int nkn );
int g2bl_NijSize ( int nkn );
int g2bl_MijSize ( int nkn );
```

The procedures above return the sizes of arrays necessary to hold the values of the expressions

$$N^i_\alpha = \frac{d^{\alpha_1}}{du^{\alpha_1}} N^3_{i_1}(u) \frac{d^{\alpha_2}}{dv^{\alpha_2}} N^3_{i_2}(v),$$
$$N^{ij}_{\alpha\beta} = N^i_\alpha N^j_\beta + N^j_\alpha N^i_\beta,$$
$$M^{ij}_{\alpha\beta} = N^i_\alpha N^j_\beta - N^j_\alpha N^i_\beta,$$

where $\boldsymbol{\alpha} = (\alpha_1, \alpha_2)$, $\boldsymbol{\beta}$ and $\boldsymbol{\gamma}$ are biindices and the functions N^3_i are cubic B-spline functions with uniform knots being consecutive integers, at the n^2 quadrature knots, where n is the value of the parameter nkn. These expressions are

evaluated and stored in the arrays to accelerate the computations during the actual optimization.

```
int _g2bl_SetupHessian1Profile ( int lastknotu, int lastknotv,
                                 int *prof );
```

```
double g2bl_UFuncd ( int nkn, const double *qcoeff, double *Nitab,
                int lastknotu, int lastknotv, int pitch, point3d *cp,
                char *dirty,
                double tC, double *ftab );
void g2bl_UFuncGradd ( int nkn, const double *qcoeff, double *Nitab,
                    int lastknotu, int lastknotv,
                    int pitch, point3d *cp, char *dirty,
                    double tC, double *ftab, double *gtab,
                    double *func, double *grad );
void g2bl_UFuncGradHessiand (
                 int nkn, const double *qcoeff, double *Nitab,
                 double *Nijtab, double *Mijtab,
                 int lastknotu, int lastknotv,
                 int pitch, point3d *cp, char *dirty,
                 double tC, double *ftab, double *gtab, double *htab,
                 double *func, double *grad,
                 int hsize, const int *prof, double **hrows );
void g2bl_ClosedUFuncGradd (
                 int nkn, const double *qcoeff, double *Nitab,
                 int lastknotu, int lastknotv,
                 int pitch, point3d *cp, char *dirty,
                 double tC, double *ftab, double *gtab,
                 double *func, double *grad );
void g2bl_ClosedUFuncGradHessiand (
                 int nkn, const double *qcoeff, double *Nitab,
                 double *Nijtab, double *Mijtab,
                 int lastknotu, int lastknotv,
                 int pitch, point3d *cp, char *dirty,
                 double tC, double *ftab, double *gtab, double *htab,
                 double *func, double *grad,
                 int hsize, const int *prof, double **hrows );
```

```
double g2bl_SurfNetDiameterSqd ( int lastknotu, int lastknotv,
                                 int pitch, const point3d *cp );
double g2bl_ClosedSurfNetDiameterSqd ( int lastknotu, int lastknotv,
                                       int pitch, const point3d *cp );
```

```
boolean g2bl_SetupULConstraintsd (
                int lastknotu, int lastknotv, int spdimen,
                int ppitch, double *cp,
                int nucurv, double *ucknots,
                int cpitch, double *uccp,
                int *nconstr, double *cmat, double *crhs );
boolean g2bl_SetupUNLConstraintsd ( int lastknotu, int lastknotv,
                int ppitch, point3d *cp,
                int nucurv, double *ucknots,
                int cpitch, point3d *uccp,
                int *nconstr, double *cmat, double *crhs );
```

```
boolean g2bl_SetupClosedULConstraintsd (
                int lastknotu, int lastknotv, int spdimen,
                int ppitch, double *cp,
                int nucurv, double *ucknots,
                int cpitch, double *uccp,
                int *nconstr, double *cmat, double *crhs );
boolean g2bl_SetupClosedUNLConstraintsd (
                int lastknotu, int lastknotv,
                int ppitch, point3d *cp,
                int nucurv, double *ucknots,
                int cpitch, point3d *uccp,
                int *nconstr, double *cmat, double *crhs );
```

```
boolean g2bl_FuncTSQFd ( int nkn,
               int lastknotu, int lastknotv, int pitch, point3d *cp,
               double tC,
               double *fT, double *fS, double *fQ, double *fF );
```


12.3 Optimization of surfaces represented by irregular meshes

12.3.1 Overview

The optimization of a surface represented by a mesh is done as follows: an application prepares the mesh, and then it calls an optimization procedure. The mesh is represented as described in Section 10.1. The mesh must have a boundary, made of one or more closed polylines; the surface represented by such a mesh has a boundary made of the same number of closed curves. The boundary of the surface is fixed, by fixing all vertices, whose distance from the mesh boundary (measured by the number of edges from the closest boundary vertex) is less than 3. These vertices also determine the tangent plane and curvature at each point of the surface boundary. The optimization procedures will not modify these vertices.

Apart from the mesh, the application may pass an array of bytes—one byte for each vertex. A nonzero value in this array marks the corresponding vertex as fixed in addition to the vertices, which determine the boundary conditions for the surface. In this way one can impose `constraints`. The other vertices, whose positions may be modified by the optimization procedures, are called non-fixed in the following text.

Instead of calling a single procedure, which does the entire job, the application may call a preparation procedure, which creates an auxiliary data structure holding all necessary data, and then, in a loop, call the procedure, which makes a single iteration of the numerical optimization algorithm. In this way the application has access to the positions of mesh vertices after each iteration. As the computations with fine meshes, having thousands of vertices take much time, it makes sense to display the mesh after each iteration, and allow the user to have an insight of the computations. When the optimization is complete, the application should call a procedure of deallocation of the data structure; it consists of a number of arrays, which take a considerable amount of memory, and not doing that would cause a massive memory leakage.

The following sets of procedures are available now:

- `g2mbl_InitBlSurfaceOptLMTd`—preparation,
  `g2mbl_IterBlSurfaceOptLMTd`—one iteration,
  `g2mbl_OptLMTDeallocated`—deallocation of the auxiliary data structure,
  `g2mbl_FindBlSurfaceLMTd`—full optimization procedure, calling the former three procedures.

  These procedures optimize the shape of surfaces by finding minima of the functional F given by Formula (12.5).

  The algorithm implemented by these procedures is appropriate for meshes with a rather small number of non-fixed vertices—up to 5000. The optimization is done using the Newton method, accompanied by minimization along

the Levenberg-Marquardt trajectories, and all systems of linear equations are solved using the Cholesky's decomposition, which is inefficient beyond that limit.

- g2mbl_InitBlSurfaceOptAltBLMTd—preparation,
  g2mbl_IterBlSurfaceOptAltBLMTd—one iteration,
  g2mbl_OptLMTDeallocated— deallocation of the auxiliary data structure,
  g2mbl_FindBlSurfaceAltBLMTd—full optimization procedure, calling the former three procedures.

  These procedures optimize the shape of surfaces by finding minima of the functional F given by Formula (12.5).

  The algorithm implemented by these procedures is appropriate for finer meshes, whose numbers of non-fixed vertices are between 3500 and 20000. The algorithm uses blocks, which are overlapping subsets covering the set of non-fixed vertices of the mesh. The number of blocks, specified by the caller, must be between 2 and the maximal number of blocks, which is the constant hidden beyond the symbolic name G2MBL_MAX_BLOCKS (currently 32, it may change in future). It is best to specify the number of blocks so as to obtain blocks having about 3000 vertices (the blocks may consist of different numbers of vertices).

- g2mbl_InitBlCMPSurfaceOptd—preparation,
  g2mbl_IterBlSurfaceOptAltBLMTd—one iteration,
  g2mbl_OptLMTDeallocated—deallocation of the auxiliary data structure.

  Two of the above procedures are the same as in the previous set of optimization procedures. A minimum of the functional F given by Formula 12.5 is searched.

  The algorithm implemented by these procedures is appropriate for meshes having between 3500 and 30000 vertices; the algorithm uses blocks. The difference is using another preconditioner, constructed with a refinement matrix. This preconditioner is used in the final stage of optimization, when the Newton method is applied to the entire system of nonlinear equations, whose solution is the minimal point of the functional F; the Newton method steps use the conjugate gradient method to solve systems of linear equations (the procedure pkn_PCGd procedure is used, see Section 3.7). This preconditioner may be used if the mesh to optimise has been obtained from a coarse mesh by one or more refinement steps (and optional repositioning of vertices), and the matrix, which describes this operation (the refinement or the composition of the refinement steps) is available. If the number of blocks is 4 or greater, then the convergence of the conjugate gradient method may be considerably faster.

  The procedure bsm_RefineBSMeshd may be used to obtain the refinement matrix bsm_RefineBSMeshd (see Section 10.2). The composition of subsequent

refinements is represented by the product of the appropriate matrices, which may be computed using the procedure pkn_SPMmultMMCd (see Section 3.6.2).

- g2mbl_MLOptInitd—preparation,
  g2mbl_MLOptIterd—one iteration,
  g2mbl_MLOptDeallocated—deallocation of the auxiliary data structure.

  These procedures use the multilevel algorithm to find a minimum of the functional F. The set of non-fixed vertices is recursicely divided into overlapping subsets, called blocks, which form a balanced binary tree. For large blocks the conjugate gradient method is used to solve the systems of linear equations in the Newton method iterations. The preconditioner is constructed using small blocks.

  In particular, if the height of the block tree is 1, the algorithm implemented by these procedures is the non-block one; the number of non-fixed vertices should be rather small (up to 5000, but preferably not greater than 3500). For very fine meshes the height of the tree should be chosen so as to obtain the smallest blocks having no more than 3500 vertices. The height of the block tree (which determines the number of blocks) may be taken as suggested by the proceure g2mbl_MLSuggestNLevels.

- g2mbl_MLCMPOptInitD—preparation,
  g2mbl_MLOptIterd—one iteration,
  g2mbl_MLOptDeallocated—deallocation of the auxiliary data structure.

  These procedures use the multilevel algorithm to find a minimum of the functional F. For large blocks the conjugate gradient method is used to solve the systems of linear equations in the Newton method iterations. The preconditioner is constructed using small blocks and a refinement matrix, which must be available to use this possibility.

  The procedure g2mbl_MLCPSuggestNLevels may be used to suggest the height of the block tree (the block overlaps in this case are smaller, which may result in a lower tree than that suggested by g2mbl_MLSuggestNLevels).

  In one of experiments these procedures found a minimum of a function of 216027 variables (coordinates of 72009 non-fixed vertices of a mesh), which took about 6 hours and 20 minutes.

- g2mbl_MLSOptInitd—preparation,
  g2mbl_MLSOptIterd—one iteration,
  g2mbl_MLOptDeallocated—deallocation of the auxiliary data structure.

  These procedures use the multilevel algorithm to find a minimum of the function S in a restricted space. It is recommended that the starting point for the minimisation be a mesh, which is a minimal point of teh functional F, found using one of the sets of procedures described above.

If the tree height is 1, the algorithm is just the non-block algorithm, which solves a global system of equations in each iteration. The number of non-fixed vertices in this case should not exceed 15000, but it better be not greater than 10000. If there are large blocks, such that the linear equations have to be solved using the conjugater gradient method, a preconditioner constructed with small blocks is used.

The procedure `g2mbl_MLSSuggestNLevels` may be used to establish the height of the block tree.

- `g2mbl_MLSCMPOptInitD`—preparation,
  `g2mbl_MLSOptIterd`—one iteration,
  `g2mbl_MLOptDeallocated`—deallocation of the auxiliary data structure.

  These procedures use the multilevel algorithm to find a minimum of the function S in a restricted space. It is recommended that the starting point (i.e. initial mesh vertices) be a minimal point of the function F, found by one of the sets of procedures described above.

  The preconditioner used by the conjugate gradient method for large blocks is constructed using the small blocks and the refinement matrix, which must be available, if this set of optimisation procedures is to be used.

A simple example showing how to use the optimization procedures mentioned above is the program `optblmesh`, which is briefly described in Section 12.3.6; its source code (see the file `test/optblmesh/optblmesh.c`) is a recommended lecture accompanying the documentation below.

12.3.2 Nonblock algorithm

```
boolean g2mbl_InitBlSurfaceOptLMTd (
                  int nv, BSMvertex *mv, int *mvhei,
                  point3d *mvcp, int nhe, BSMhalfedge *mhe,
                  int nfac, BSMfacet *mfac, int *mfhei,
                  byte *mkcp,
                  double C, double d0, double dM,
                  int nkn1, int nkn2, void **data );
boolean g2mbl_IterBlSurfaceOptLMTd ( void *data, boolean *finished );
void g2mbl_OptLMTDeallocated ( void **data );
boolean g2mbl_FindBlSurfaceLMTd ( int nv, BSMvertex *mv, int *mvhei,
                        point3d *mvcp, int nhe, BSMhalfedge *mhe,
                        int nfac, BSMfacet *mfac, int *mfhei,
                        byte *mkcp,
                        double C, double d0, double dM,
                        int maxit, int nkn1, int nkn2 );
```


The procedures above may be used to find a minimum of F using the simplest (nonblock) algorithm, which is appropriate for meshes with a rather small number of non-fixed vertices (up to 3500). The procedure `g2mbl_FindBlSurfaceLMTd` calls the other three, which may also be called directly by an application.

Parameters: `nv`, `nhe`, `nfac`—numbers n_v, n_h and n_f of vertices, halfedges and facets, respectively, `mv`—array of vertices, `mvhei`—array of indices of halfedges having origins at subsequent vertices, `mhe`—array of halfedges, `mfac`—array of facets, `mfhei`—array of indoces of halfedges forming the facets.

The parameter `mkcp` may be `NULL` or it may point to an array of length n_v, which marks (by nonzero values) vertices fixed in addition to the ones, which determine boundary conditions.

The parameters `C`, `d0` and `dM` are used to compute the constant c in Formula (12.5). Their values are the numbers C, D_Ω and $D_\mathcal{M}$ respectively. The number C is a user-specified positive constant; in many experiments good results were obtained with $C = 0$. The number D_Ω should be the diameter of the domain Ω of a parameterization of the surface, and $D_\mathcal{M}$ should be the diameter of the surface. If these two parameters are less than or equal to 0, the procedure will compute approximations of these diameters, which is preferable.

The parameter `maxit` specifies the limit of number of iterations made by the `g2mbl_FindBlSurfaceLMTd` proceure.

The parameters `nkn1` and `nkn2` determine the orders of quadratures used to evaluate the function F and its gradient and Hessian. Their values, n_1 and n_2 must be between 4 and 10, and there should be $n_1 \le n_2$. The quadratures are tensor product Gauss-Legendre quadratures with n_1^2 and n_2^2 knots in each domain square—the domain Ω of a parameterization of the surface represented by the mesh is a manifold made of these squares. Greater values of these parameters mean greater accuracy and longer computation times. The quadrature with n_1^2 knots is used to compute the Hessian coefficients, and also to compute the function value and its gradient in the beginning of search of the minimum. In the final phase, the quadrature with n_2^2 knots in each domain square is used to get a better accuracy.

The parameter `data` of the procedures `g2mbl_InitBlSurfaceOptLMTd` and `g2mbl_OptLMTDeallocated` points to a pointer to the auxiliary data structure, created by the first and destroyed by the second of the two procedures. The pointer to this data structure is the parameter `data` of the procedure `g2mbl_IterBlSurfaceOptLMTd`, which makes a single iteration.

The variable pointed by the parameter `finished` is assigned `true` when the stop criterion of the procedure is satisfied. One should not continue iterations after that event.

The non-`void` procedures return `true` in case of success, or `false` in case of failure of their missions.

12.3.3 Two-level block algorithm

```
boolean g2mbl_InitBlSurfaceOptAltBLMTd (
                  int nv, BSMvertex *mv, int *mvhei,
                  point3d *mvcp, int nhe, BSMhalfedge *mhe,
                  int nfac, BSMfacet *mfac, int *mfhei,
                  byte *mkcp,
                  double C, double d0, double dM,
                  int nkn1, int nkn2, int nbl,
                  void **data );
boolean g2mbl_InitBlCMPSurfaceOptd (
               int fnv, BSMvertex *fmv, int *fmvhei, point3d *fmvcp,
               int fnhe, BSMhalfedge *fmhe,
               int fnfac, BSMfacet *fmfac, int *fmfhei,
               byte *fmkcp,
               int cnv,
               int rmnnz, index2 *rmnzi, double *rmnzc,
               double C, double d0, double dM,
               int nkn1, int nkn2, int nbl,
               void **data );
boolean g2mbl_IterBlSurfaceOptAltBLMTd ( void *data,
                                    boolean *finished );
boolean g2mbl_FindBlSurfaceAltBLMTd (
                  int nv, BSMvertex *mv, int *mvhei,
                  point3d *mvcp, int nhe, BSMhalfedge *mhe,
                  int nfac, BSMfacet *mfac, int *mfhei,
                  byte *mkcp,
                  double C, double d0, double dM,
                  int maxit, int nkn1, int nkn2, int nbl );
```

The auxiliary data structure for the two-level block algorithm may be created in two alternative ways: by calling the procedure `g2mbl_InitBlSurfaceOptAltBLMTd` or `g2mbl_InitBlCMPSurfaceOptd`. In the former case the preconditioner used by the conjugate gradient method is made of solvers of systems of linear equations of small blocks. In the latter case the preconditioner has a term defined with a refinement matrix. For meshes with many non-fixed vertices this preconditioner is more efficient.

The procedure `g2mbl_IterBlSurfaceOptAltBLMTd` should be called in a loop, after creating the auxiliary data structure, until it assigns `true` to the variable pointed by the parameter `finished`. Then the non-fixed mesh vertices are moved to the positions, which correspond to the minimal point of the functional F. The auxiliary data structure must be destroyed by calling `g2mbl_OptLMTDeallocated`.

Parameters: `nv`, `mv`, `mvhei`, `mvcp`, `nhe`, `mhe`, `nfac`, `mfac`, `mfhei`—representation of the mesh, `mkcp`—if not `NULL`, then is is an array which specifies the vertices fixed in addition to the ones describing boundary conditions, `C`, `d0`, `dM`—three numbers used to calculate the constant c in Formula (12.5), `maxit`—limit of number of iterations, `nkn1`, `nkn2`—specify the quadratures. More details about these parameters are in Section 12.3.2.

The parameter `nbl` specifies the number of small blocks used by the algorithm. This number must be between 2 and `G2MBL_MAX_BLOCKS`. It should be chosen in such a way that the blocks have about 3500 vertices; in this algorithm small blocks differ in size. Perhaps a good choice of the number of blocks is $\lceil n_{\mathrm{nfv}}/2500 \rceil$, where n_{nfv} is the number of non-fixed vertices of the mesh.

The parameters `fnv`, `fmv`, `fmvhei`, `fmvcp`, `fnhe`, `fmhe`, `fnfac`, `fmfac`, `fmfhei` of the procedure `g2mbl_InitBlCMPSurfaceOptd` describe the fine mesh to be optimised. The parameter `cnv` is the number of vertices of a coarse mesh such that the fine mesh topology was obtained by one or more refinement operations. The coarse mesh itself is unnecessary, all that is needed is the refinement matrix (represented as a sparse matrix with nonzero coefficients distributed irregularly, see Section 3.6.2). The procedure `g2mbl_InitBlCMPSurfaceOptd` prepares a preconditioner for the conjugate gradient method with a term defined with the refinement matrix, which is much more efficient than the preconditioner without this term, especially when the number of small blocks is big (say, greater than four).

The return value of each of the above procedures is `true` in case of success, or `false` after a failure.

The procedure `g2mbl_FindBlSurfaceAltBLMTd` calls `g2mbl_InitBlSurfaceOptAltBLMTd` to create the auxiliary data structure, then it calls `g2mbl_IterBlSurfaceOptAltBLMTd` in a loop, and then it calls `g2mbl_OptLMTDeallocated` to clean up.

12.3.4 Multilevel algorithm

Main procedures

```
boolean g2mbl_MLOptInitd (
                  int nv, BSMvertex *mv, int *mvhei, point3d *mvcp,
                  int nhe, BSMhalfedge *mhe,
                  int nfac, BSMfacet *mfac, int *mfhei,
                  byte *mkcp,
                  double C, double d0, double dM,
                  int nkn1, int nkn2, short nlevels, void **data );
boolean g2mbl_MLCMPOptInitd (
               int fnv, BSMvertex *fmv, int *fmvhei, point3d *fmvcp,
               int fnhe, BSMhalfedge *fmhe,
               int fnfac, BSMfacet *fmfac, int *fmfhei,
```

```
                byte *fmkcp,
                int cnv,
                int rmnnz, index2 *rmnzi, double *rmnzc,
                double C, double d0, double dM,
                int nkn1, int nkn2, short nlevels,
                void **data );
boolean g2mbl_MLOptIterd ( void *data, boolean *finished );
void g2mbl_MLOptDeallocated ( void **data );
```

The procedures above implement the multilevel algorithm of finding a minimum of the functional F. The first two, `g2mbl_MLOptInitd` and `g2mbl_MLCMPOptInitd`, create the auxiliary data structure, which must eventually be destroyed by `g2mbl_MLOptDeallocated`. The difference between the two procedures is the preparation of a preconditioner for the conjugate gradient method, with a term usnig the refinement matrix.

The procedure `g2mbl_MLOptIterd` makes one iteration of the algorithm. If the stop condition is satisfied, then the variable pointed by the parameter `finished` is assigned `true`, which ought to terminate the computations.

The parameters, which describe the mesh and the other data are the same as for the procedures described in the preceding sections. A new parameter is `nlevels`, which is the height of the binary tree of blocks used by the algorithm. If this parameter is 1, the procedures work as a nonblock algorithm. To choose the height of the block tree one may call the procedure `g2mbl_MLSuggestNLevels` or `g2mbl_MLCPSuggetsNLevels`, described below.

The procedure `g2mbl_MLCMPOptInitd` should be used if the height of the block tree is at least 3.

The procedures return `true` to signal a success, or `false` to signal a failure.

```
boolean g2mbl_MLSOptInitd (
                  int nv, BSMvertex *mv, int *mvhei, point3d *mvcp,
                  int nhe, BSMhalfedge *mhe,
                  int nfac, BSMfacet *mfac, int *mfhei,
                  byte *mkcp,
                  int nkn1, int nkn2, short nlevels, void **data );
boolean g2mbl_MLSCMPOptInitd (
             int fnv, BSMvertex *fmv, int *fmvhei, point3d *fmvcp,
             int fnhe, BSMhalfedge *fmhe,
             int fnfac, BSMfacet *fmfac, int *fmfhei,
             byte *fmkcp,
             int cnv,
             int rmnnz, index2 *rmnzi, double *rmnzc,
             int nkn1, int nkn2, short nlevels,
             void **data );
```



```
boolean g2mbl_MLSOptIterd ( void *data, boolean *finished );
```

The procedures above implement the multilevel algorithm of finding a minimum of the functional S. The first two, `g2mbl_MLSOptInitd` and `g2mbl_MLSCMPOptInitd`, create the auxiliary data structure, which must eventually be destroyed by `g2mbl_MLOptDeallocated`. The difference between the two procedures is the preparation of a preconditioner for the conjugate gradient method, with a term usnig the refinement matrix.

The procedure `g2mbl_MLSOptIterd` makes one iteration of the algorithm. If the stop condition is satisfied, then the variable pointed by the parameter `finished` is assigned `true`, which ought to terminate the computations.

The parameters, which describe the mesh and the other data are the same as for the procedures described in the preceding sections. Note that there are no parameters `C`, `d0` and `dM`, which are not necessary to define the functional S. The parameter `nlevels` is the height of the binary tree of blocks used by the algorithm. If this parameter is 1, the procedures work as a nonblock algorithm. To choose the height of the block tree one may call the procedure `g2mbl_MLSSuggestNLevels` or `g2mbl_MLCPSSuggetsNLevels`, described below.

Auxiliary procedures

```
boolean g2mbl_MLSuggestNLevels ( int nv, BSMvertex *mv, int *mvhei,
                                 int nhe, BSMhalfedge *mhe,
                                 int nfac, BSMfacet *mfac, int *mfhei,
                                 byte *mkcp,
                                 int *minlev, int *maxlev );
boolean g2mbl_MLCPSuggestNLevels ( int nv, BSMvertex *mv, int *mvhei,
                                   int nhe, BSMhalfedge *mhe,
                                   int nfac, BSMfacet *mfac, int *mfhei,
                                   byte *mkcp,
                                   int *minlev, int *maxlev );
```

The two procedures above count the number of non-fixed vertices in a mesh and then they compute the numbers, which seem to define a good range for the height of the block tree for the multilevel block algorithm of finding a minimum of the function F. These numbers are assigned to the variables pointed by the parameters `minlev` and `maxlev`. It seems better to choose the lower bound of this range for the tree height.

The first of the two procedures is appropriate if no refinement matrix is to be used by the preconditioner for the conjugate gradient method. The second procedure is supposed to be appropriate if such a matrix is available and intended to use. As the block overlaps necessary in that case are smaller, the block tree in some cases may be lower.

The value returned is `true` to indicate a success, and `false` in case of failure.

```
boolean g2mbl_MLSSuggestNLevels (
                        int nv, BSMvertex *mv, int *mvhei,
                        int nhe, BSMhalfedge *mhe,
                        int nfac, BSMfacet *mfac, int *mfhei,
                        byte *mkcp,
                        int *minlev, int *maxlev );
boolean g2mbl_MLCPSSuggestNLevels (
                        int nv, BSMvertex *mv, int *mvhei,
                        int nhe, BSMhalfedge *mhe,
                        int nfac, BSMfacet *mfac, int *mfhei,
                        byte *mkcp,
                        int *minlev, int *maxlev );
```

The two procedures above count the number of non-fixed vertices in a mesh and then they compute the numbers, which seem to define a good range for the height of the block tree for the multilevel block algorithm of finding a minimum of the function S. These numbers are assigned to the variables pointed by the parameters `minlev` and `maxlev`. It seems better to choose the lower bound of this range for the tree height.

The first of the two procedures is appropriate if no refinement matrix is to be used by the preconditioner for the conjugate gradient method. The second procedure is supposed to be appropriate if such a matrix is available and intended to use. As the block overlaps necessary in that case are smaller, the block tree in some cases may be lower.

The value returned is `true` to indicate a success, and `false` in case of failure.

```
int g2mbl_MLGetLastBlockd ( void *data );
```

The multilevel algorithm solves the local optimization problem for each block, from the last one to the first, which is the global system of equations for all non-fixed vertices. Each time one iteration is made, and then a decision is taken, whether to advance to the next block, or not. The value returned is the number of the block, which was processed in the last recent iteration. This information may be used by the application to monitor the state of the computation.

```
void g2mbl_MLSetNextBlock ( void *data, int nbl );
```

Normally the multilevel algorithm begins the computation from the last small block. This procedure may be used to override it, by explicitly setting the block number, from which to start the next iteration.


```
boolean g2mbl_MLGetBlockVCPNumbersd ( void *data, int bl,
                                int *nvcp, int **vncpi, int *seed );
```

After creating the auxiliary data structure for the multilevel algorithm, this procedure may be used to obtain the information about blocks created for the multilevel algorithm. The first parameter points to the auxiliary data structure, the second is the block number, the parameter `nvcp` points to the variable, which is assigned the number of vertices of the block, the parameter `vncpi` points to the variable, which will be set to an array with indices of the vertices of the block (the application may read this array, but it must not alter its contents), and the parameter `seed` points to the variable, which is assigned the number of the vertex chosen for the seed of the discrete Voronoi diagram used to choose the vertices for this block.

```
void g2mbl_MLGetTimes ( void *data,
              float *time_prep, float *time_h, float *time_cg );
```

After the optimization using the multilevel algorithm is complete, but before destroying the auxiliary data structure, this procedure may be called to obtain the total times (in seconds), spent for the preparation (creating blocks etc.), computing the Hessian coefficients, and solving the linear equations using the conjugate gradient method. These times are assigned to the variables pointed by the parameters.

12.3.5 Additional procedures

```
extern GHoleDomaind *g2mbl_domaind[GH_MAX_K-3];
extern double *g2mbl_patchmatrixd[GH_MAX_K-3];

boolean g2mbl_SetupHolePatchMatrixd ( int k );
void g2mbl_CleanupHoleDomainsd ( void );
```

The construction of the function space for representation of the surface involves constructing binonic polynomials filling k-sided holes in the piecewise bicubic surface represented by the mesh (where k may be any number from 3 to 16, except 4). This is done using procedures of the `libeghole` library. For each type of special element present in the mesh, an auxiliary data structure defined in the `libeghole` library is created (these types correspond to different numbers k). After the optimization these data structures remain allocated, and they may be reused to optimise a next mesh.

The procedure `g2mbl_SetupHolePatchMatrixd` may be called to demand creating the data structure for filling the k-sided hole by the appropriate procedure from the `libeghole` library. The value `true` returned denotes a success.

The procedure `g2mbl_CleanupHoleDimaonsd` destroys all data structures for filling k-sided holes in order to free the memory they occupy. If a mesh is optimised

after that, the data structures must be created again, which takes some time, though insignificant, when compared with the times of optimization of the meshes.

```
int g2mbl_GetNvcp ( int nv, BSMvertex *mv, int *mvhei,
                    int nhe, BSMhalfedge *mhe,
                    int nfac, BSMfacet *mfac, int *mfhei,
                    byte *mkcp );
```

This procedure returns the number of non-fixed vertices in a mesh. It may be useful to choose the number of blocks for the two-level optimization block algorithm.

```
extern void (*g2mbl_outputnzdistr)( int nbl, int blnum,
                                    boolean final,
                                    int nvcp, int n, byte *nzdistr );
```

This pointer is `NULL` by default. An application may set it to point a procedure, which will be called during the preparation of the auxiliary data structure for optimization, to pass to the application the distribution of nonzero coefficients in the Hessian matrices for small blocks. This possibility was helpful for writing and debugging of the procedures of creating blocks.

12.3.6 Example of using the optimization procedures

An example of using the mesh optimization procedures is the program `optblmesh`, whose source file is `test/optblmesh/optblmesh.c`. The program works in batch mode; it reads the file specified by the command line, performs the optimization and writes the result to a file. It does not make any pictures, but the files may be read in by the demonstration program `pozwalaj`, which allows one to examine the surfaces at will, edit them and interactively create meshes, which may be optimized (see Section 16.6).

The program `pozwalaj` also has the optimization procedures built in, but the source code of `optblmesh` is much shorter (about 650 lines) and easier to explore. It is intended to serve as an example, how to prepare data and to call properly the optimization procedures. There are not too many comments, but most of the program are instructions of calling library procedures, which usually have significant names. The parameters and jobs of these procedures are described in this documentation.

13. The libbsfile library

```
#define BSF_SYMB_EOF 0
#define BSF_SYMB_ERROR 1
#define BSF_SYMB_INTEGER 2
#define BSF_SYMB_FLOAT 3
#define BSF_SYMB_LBRACE 4
#define BSF_SYMB_RBRACE 5
#define BSF_SYMB_PLUS 6
#define BSF_SYMB_MINUS 7
#define BSF_SYMB_STRING 8
#define BSF_SYMB_COMMA 9
```

```
#define BSF_FIRST_KEYWORD 10
#define BSF_SYMB_BCURVE 10
#define BSF_SYMB_BPATCH 11
#define BSF_SYMB_BSCURVE 12
#define BSF_SYMB_BSHOLE 13
#define BSF_SYMB_BSPATCH 14
#define BSF_SYMB_CLOSED 15
#define BSF_SYMB_CPOINTS 16
#define BSF_SYMB_DEGREE 17
#define BSF_SYMB_DIM 18
#define BSF_SYMB_DOMAIN 19
#define BSF_SYMB_KNOTS 20
#define BSF_SYMB_KNOTS_U 21
#define BSF_SYMB_KNOTS_V 22
#define BSF_SYMB_NAME 23
#define BSF_SYMB_RATIONAL 24
#define BSF_SYMB_SIDES 25
#define BSF_SYMB_UNIFORM 26
```

```
#define BSF_NKEYWORDS 17
extern const char *bsf_keyword[BSF_NKEYWORDS];
```

```
extern FILE *bsf_input, *bsf_output;
```

```
extern int bsf_nextsymbol;
extern int bsf_nextint;
extern double bsf_nextfloat;
```



```
boolean bsf_OpenInputFile ( char *filename );
void bsf_CloseInputFile ( void );
void bsf_GetNextSymbol ( void );
```

```
void bsf_PrintErrorLocation ( void );
```

```
boolean bsf_ReadDoubleNumber ( double *number );
boolean bsf_ReadPointd ( int maxspdimen, double *point, int
*spdimen );
int bsf_ReadPointsd ( int maxspdimen, int maxnpoints,
double *points, int *spdimen );
```

```
boolean bsf_ReadSpaceDim ( int maxdim, int *spdimen );
boolean bsf_ReadCurveDegree ( int maxdeg, int *degree );
boolean bsf_ReadPatchDegree ( int maxdeg, int *udeg, int *vdeg );
```

```
boolean bsf_ReadKnotSequenced ( int maxlastknot, int *lastknot,
double *knots,
boolean *closed );
```

```
boolean bsf_ReadBezierCurve4d ( int maxdeg, int *deg, point4d
*cpoints,
int *spdimen, boolean *rational );
```

```
boolean bsf_ReadBSplineCurve4d ( int maxdeg, int maxlastknot, int
maxncpoints,
int *deg, int *lastknot, double *knots,
boolean *closed, point4d *cpoints,
int *spdimen, boolean *rational );
```

```
boolean bsf_ReadBezierPatch4d ( int maxdeg, int maxlastknot, int
maxncpoints,
int *udeg, int *vdeg,
int *pitch, point4d *cpoints,
int *spdimen, boolean *rational );
```

```
boolean bsf_ReadBSplinePatch4d ( int maxdeg, int maxlastknot, int
maxncpoints,
int *udeg, int *lastknotu, double *knotsu,
int *vdeg, int *lastknotv, double *knotsv,
boolean *closed_u, boolean *closed_v,
int *pitch, point4d *cpoints,
int *spdimen, boolean *rational );
```

```
boolean bsf_ReadBSplineHoled ( int maxk, int *hole_k, double
*knots,
point2d *domain_cp, point3d *hole_cp );
```

```
boolean bsf_OpenOutputFile ( char *filename );
void bsf_CloseOutputFile ( void );
```

```
void bsf_WriteComment ( char *comment );
void bsf_WriteDoubleNumber ( double x );
void bsf_WritePointd ( int spdimen, const double *point );
void bsf_WritePointsd ( int spdimen, int cols, int rows, int pitch,
const double *points );
```

```
void bsf_WriteSpaceDim ( int spdimen );
void bsf_WriteCurveDegree ( int degree );
void bsf_WritePatchDegree ( int udeg, int vdeg );
```

```
void bsf_WriteKnotSequenced ( int lastknot, const double *knots,
boolean closed );
```

```
boolean bsf_WriteBezierCurved ( int spdimen, boolean rational,
int deg, const double *cpoints );
```

```
boolean bsf_WriteBSplineCurved ( int spdimen, boolean rational,
int deg, int lastknot, const double *knots,
boolean closed,
double *cpoints );
```

```
boolean bsf_WriteBezierPatchd ( int spdimen, boolean rational,
int udeg, int vdeg,
int pitch, const double *cpoints );
```



```
boolean bsf_WriteBSplinePatchd ( int spdimen, boolean rational,
int udeg, int lastknotu, const double *knotsu,
int vdeg, int lastknotv, const double *knotsv,
boolean closed_u, boolean closed_v,
int pitch, const double *cpoints );
```

```
boolean bsf_WriteBSplineHoled ( int hole_k, const double *knots,
const point2d *domain_cp,
const point3d *hole_cp );
```

14. The `libmengerc` library

This library consists of procedures, whose purpose is to find minimal curves of the integral Menger curvature, a functional defined with the formula

$$K_p(\mathcal{C}) = \int\int\int_{\mathcal{C}^3} K(\mathbf{p}_1, \mathbf{p}_2, \mathbf{p}_3)^p \, d\mu(\mathcal{C}) \, d\mu(\mathcal{C}) \, d\mu(\mathcal{C})$$

where $K(\mathbf{p}_1, \mathbf{p}_2, \mathbf{p}_3)$ is the Menger curvature of the triple of points of the curve $\mathcal{C}$ (in $\mathbb{R}^3$), p is an exponent, which should be greater than 3 (in practice: from 4 to 20) and the integral is taken over all triples of points of the curve, with respect to the arc length measure.

The curve $\mathcal{C}$ is a closed B-spline curve of degree at least 3 with uniorm knots. Such a curve is a knot in $\mathbb{R}^3$. Given an initial curve, the procedures search local minima of the integral Menger curvature in the set of curves, whose length is that of the initial curve. The local minimum is a knot topologically equivalent to the initial curve. The problem is discretized by defining a function, whose arguments are Cartesian coordinates of the control points of the curve, and whose value is the value of the functional to minimize.

The optimization method is described in the paper *Shape optimization of closed B-spline curves by minimization of the integral Menger curvature*, in preparation.

Actually, minima of the following functionals are searched:

$$\tilde{K}_p(\mathcal{C}) = L(\mathcal{C})^{p-3} K_p(\mathcal{C}), \quad \text{or}$$
$$\hat{K}_p(\mathcal{C}) = L(\mathcal{C}) K_p(\mathcal{C})^{1/(p-3)},$$

where $L(\mathcal{C})$ is the length of the curve $\mathcal{C}$. These functionals are an invariants of geometric similarities, i.e. homotetiae and isometries. In any set of curves of a fixed length the functionals K_p, $\tilde{K}_p$ and $\hat{K}_p$ differ by constant factors. Their minimization problems are still ill-posed; to obtain a well posed minimization problem, five penalty terms, described in detail in the paper, are added.

The functional $\hat{K}_p$ is more convenient in the numerical computations, as $\tilde{K}_p$ grows fast with the growth of the exponent p. Both were used in experiments and $\hat{K}_p$ is chosen by default, though the code using $\tilde{K}_p$ is still present in the library.

14.1 Demo programs in the package

A batch-mode program reading the curve and searching for a minimum may be found in the directory `bstools/test/mengerc`.

The procedures are also built in the demonstration program `pozwalaj`. To experiment, create or read in a closed cubic B-spline curve with uniform knots, click the `Options` button and the `Menger curv.` switch, set the parameters and click the `optimize` button.

14.2 Library contents

14.2.1 Symbolic constants

```
#define MENGERC_MIN_NQKN 2
#define MENGERC_MAX_NQKN 10
```

The integral is approximated by the composite Gauss-Legendre quadrature; as the curves processed by procedures in this library are B-splines, the domain of the curve is divided into intervals between curve knots (as the curve knots are consecutive integers, these are unit length intervals). The restriction of the curve to each of those is a polynomial curve (of class C^∞). The Gauss-Legendre quadrature of order $2k$ in each unit interval is used. The symbolic constants above specify the range for the number of quadrature knots, k.

```
#define MENGERC_NPPARAM 5
#define MENGERC_OPT_NONE 0
#define MENGERC_OPT_FULL1 1
#define MENGERC_OPT_FULL2 2
#define MENGERC_OPT_PART 3
```

To make the optimization problem regular, five penalty terms are added to the integral Menger curvature, as described in the paper *Shape optimization ...*. These terms are multiplied by positive constants, which are nontrivial to choose so as to achieve the convergence of the optimization method.

The constants (penalty parameters) are given in an array passed to the procedure `mengerc_InitMCOptimization` or `mengerc_OptimizeMengerCurvature` as the parameter `penalty_param`.

The last four symbolic constant are used to specify the method of choosing the parameters; they are supposed to be passed as the value of the parameter `opt`.

`MENGERC_OPT_NONE` selects no automatic choice of the penalty parameters—the responsibility for choosing them is left to the caller.

`MENGERC_OPT_FULL1`, `MENGERC_OPT_FULL2` and `MENGERC_OPT_PART` let the optimization procedure choose the penalty parameters by minimization of a function, which depends on the greatest and the smallest eigenvalue of the Hessian matrix of the function to minimize—the sum of the Hessian of the discretized integral Menger curvature and the Hessians of the five penalty terms. The goal is to obtain a function (chosen heuristically based on numerical experiments), whose Hessian (for the B-spline curve being the current approximation of the minimal curve) is

positive-definite, with a moderate condition number. The minimization is done using the same numerical procedure, which is used to find minimal curves of the integral Menger curvature—minimization along the Levenberg–Marquardt trajectories. In many cases the optimization gives good results, though more experiments and theoretical research are needed.

14.2.2 Data structure

```
typedef struct {
    ...
  } mengerc_data;
```

This structure has a number of fields to store the information necessary for the optimization, like the pointer to the array of curve control points, arrays of quadrature knots and coefficients, values of the B-spline functions and their derivatives at the quadrature knots etc. In fact these are private data of hardly any interest to applications (but potentially meaningful for debugging purposes and experiments involving the algorithm modifications). The structure is filled with information by the procedure `mengerc_InitMCOptimization` and then passed as the parameter to the procedure `mengerc_IterMCOptimization` making the optimization step, which should be executed in a loop.

14.2.3 Main optimization procedures

```
boolean mengerc_InitMCOptimization ( int deg, int lkn,
                                double *knots, point3d *cpoints,
                                double w,
                                double penalty_param[MENGERC_NPPARAM],
                                int nqkn, int npthr, int opt,
                                mengerc_data *md );
```

This procedure prepares the data necessary for the numerical optimization, ahoch involves the alloation of necessary arrays, generating the quadrature knots and coefficients, evaluating the necessary B-spline functions and their derivatives at the quadrature knots etc. It ought to be called once before the optimization.

Parameters: `deg`, `lkn`, `knots`, `cpoints` specify the curve which is the starting point for he optimization. The degree (`deg`) must be at least 3. The parameter `lkn` must be greater than 3 times the degree. The curve knots in the array `knots` must be consecutive integers from 0 to `lkn`.

The parameter `w` is the exponent p, which must be greater than 3.

The array `penalty_param` contains the penalty parameters—five positive constants. Depending on the value of the parameter `opt` these are used without modificaton, or changed between some optimization steps.

The parameter `nqkn` is the number k of quadrature knots in each unit length

interval between the spline curve knots. The Gauss-Legendre quadrature of order 2k is used to approximate the integrals.

The parameter `npthr` specifies the number of threads. If greater than 1, some computations (e.g. the evaluation of quadratures) are done in parallel, which shortens the computation time on multiprocessor (or multicore) computers.

The parameter `md` points to the structure in which the data are stored. This structure must then be passed to the procedure `mengerc_IterMCOptimization`.

The value returned is `true` in case of success and `false` after failure, which may be caused by incorrect input data or insufficient memory.

```
boolean mengerc_IterMCOptimization ( mengerc_data *md,
                                     boolean *finished );
```

This procedure makes one step of the numerical minimization. The first parameter points to the data structure prepared by `mengerc_InitMCOptimization`. The second parameter points to a variable, which is set to `true` after the termination condition is satisfied.

The optimization step is either a Newton method step (a zero of the function gradient is searched) or a minimization along one Levenberg-Marquardt trajectory.

The return value `true` indicates a success, and `false` a failure.

```
boolean mengerc_OptimizeMengerCurvature (
            int deg, int lkn, double *knots, point3d *cpoints,
            double w, double penalty_param[MENGERC_NPPARAM],
            int nqkn, int npthr, int opt, int maxit,
            void (*outiter)(void *usrdata,
                            boolean ppopt, int mdi,
                            int it, int itres, double f, double g),
            void *usrdata );
```

This procedure calls `mengerc_InitMCOptimization` and then in a loop the procedure `mengerc_IterMCOptimization`, until a stop criterion is satisfied or an error occurs. The first nine parameters are the same as the parameters of the procedure `mengerc_InitMCOptimization`.

The parameter `maxit` is the limit of the number of iterations.

The procedure pointed by `outiter` is called after each iteration and it may output the result of this iteration, so that the application may visualise it. The parameter `usrdata` is a pointer passed to the output procedure, allowing for a communication with the application without global variables.

14.2.4 Auxiliary and private procedures

The procedures described below are of less interest to applications, though some of them might be called to obtain a detailed information about a curve (like the value of the functional). Most of them should be private to the library, i.e. have headers

moved to a private header file (mengercprivate.h in the src directory), and not the one intended to be included in application source files. In future some of the headers will be moved to the private header file, after I decide that the library is beyond the experimental stage.

```
boolean mengerc_TabBasisFunctions ( int deg, int nqkn,
                                    mengerc_data *md );
```

This procedure evaluates B-spline functions of degree specified by the parameter deg, with uniform (integer) knots at the knots of the Gauss-Legendre quadrature with nqkn knots in the interval $[0,1]$. The result is stored in the data structure pointed by md.

```
boolean mengerc_BindACurve ( mengerc_data *md,
                    int deg, int lkn, double *knots,
                    point3d *cpoints,
                    int nqkn, double w, double *penalty_param,
                    boolean alt_scale );
```

This procedure binds the curve specified by the parameters deg, lkn, knots and cpoints to the data structure pointed by md, and also calls the procedure mengerc_TabBasisFunctions to prepare the numerical integration.

The parameter alt_scale choses between the minimization of the functional $\tilde{\mathrm{K}}_\mathrm{p}$ and $\hat{\mathrm{K}}_\mathrm{p}$; mathematically they are equivalent, but with large exponent p (given as the parameter w) the former one takes very big values, which is troublesome in numerical computations. Therefore alt_scale should be true.

```
void mengerc_UntieTheCurve ( mengerc_data *md );
```

This procedure disposes of the arrays allocated by mengerc_BindACurve. It is called after the minimization is done successfully or with a failure; it ought to be called by an application, if the application has the loop, in which it calls mengerc_IterMCOptimization, after breaking the loop.

```
boolean mengerc_intF ( mengerc_data *md,
                int lkn, double *knots, point3d *cpoints,
                double *func );
boolean mengerc_gradIntF ( mengerc_data *md,
                int lkn, double *knots, point3d *cpoints,
                double *intf, double *grad );
boolean mengerc_hessIntF ( mengerc_data *md,
                int lkn, double *knots, point3d *cpoints,
                double *intf, double *grad, double *hess );
```



```
boolean _mengerc_intF ( mengerc_data *md, double *func );
boolean _mengerc_gradIntF ( mengerc_data *md, double *func,
                            double *grad );
boolean _mengerc_hessIntF ( mengerc_data *md, double *func,
                            double *grad, double *hess );
```

The six procedures above evaluate the function obtained by discretization of $\tilde{\mathrm{K}}_\mathrm{p}$ or $\hat{\mathrm{K}}_\mathrm{p}$ and its gradient and Hessian. The last three functions are wrappers for the first three. The function arguments are the Cartesian coordinates of the control points of the B-spline curve; the curve of degree n with $N+1$ knots has $N-n$ control points, and the last n of them are the same that the first n. Thus the number of function arguments is $N-2n$.

```
boolean mengerc_intD ( mengerc_data *md,
                int lkn, double *knots, point3d *cpoints,
                double *dl, double *acp );
boolean mengerc_gradIntD ( mengerc_data *md,
                int lkn, double *knots, point3d *cpoints,
                double *dl, double *grdl, double *acp,
                double *gracp );
boolean mengerc_hessIntD ( mengerc_data *md,
                int lkn, double *knots, point3d *cpoints,
                double *dl, double *grdl, double *hesdl,
                double *acp, double *gracp, double *hesacp );
```

The three functions above evaluate the functional $\mathrm{L}(\mathcal{C})$, which is the length of the curve $\mathcal{C}$, and its gradient and Hessian. These are used to evaluate the penalty terms, making the minimization problem regular.

```
boolean mengerc_IntegralMengerf ( int n, void *usrdata, double *x,
                    double *f );
boolean mengerc_IntegralMengerfg ( int n, void *usrdata, double *x,
                    double *f, double *g );
boolean mengerc_IntegralMengerfgh ( int n, void *usrdata,
                    double *x, double *f, double *g, double *h );
```

These procedures evaluate the functional, which is actually minimized, i.e. the sum of $\tilde{\mathrm{K}}_\mathrm{p}$ or $\hat{\mathrm{K}}_\mathrm{p}$ and the five penalty terms, and its gradient and Hessian. These procedures are passed as parameters to the numerical optimization procedure pkn_NLMIterd, which makes one minimization step; either the Newton method step, or minimization along one Levenberg–Marquardt trajectory. Therefore their headers have the form required by the optimization procedure.

```
boolean mengerc_IntegralMengerTransC ( int n, void *usrdata,
                                       double *x );
```

This procedure transforms the curve so as to minimize (annihilate) the penalty terms, for which this is trivial. The transformation involves scaling of the curve to obtain the curve of desired length, translating it so as to obtain the gravity centre of the control points at the origin of the coordinate system and rotating it.

```
boolean mengerc_HomotopyTest ( int n, void *usrdata,
                    double *x0, double *x1, boolean *went_out );
```

If a curve has a self-intersection, then the integral Menger curvature with the exponent greater than 3 is infinite. However, its approximation using a quadrature may be finite. The test done by this procedure indicates, whether there exists a self-intersection of a curve being the linear interpolant between two curves, whose control points are given in the arrays `x0` and `x1`. In that case `x1` must not be taken as the next approximation of the minimal point, because it indicates "untying" the knot. The failure of the test is signalled by assigning `true` to the variable pointed by `went_out`. The return value `false` signalls an error, i.e. lack of memory.

```
int mengerc_FindRemotestPoint ( int np, point3d *cpoints,
                                point3d *sc );
int mengerc_ModifyRemotestPoint ( int np, point3d *cpoints,
                                  point3d *sc, int mdi );
```

One of the penalty terms introduced to obtain a regular problem (i.e. with locally unique minimal points) is a parameter value corresponding to the point of the curve most distant from the gravity centre of the curve. Instead of the gravity centre of the curve, the gravity centre of the set of control points is used, and to determine the necessary parameter value, the control point most distant from the gravity centre is searched.

The two procedures with the headers above take care of finding the most distant point and taking decisione, whether the parameter corresponding to the most distant point is to be changed.

```
boolean mengerc_OptPenaltyParams1 ( mengerc_data *md,
                                    boolean wide );
boolean mengerc_OptPenaltyParams2 ( mengerc_data *md );
boolean mengerc_OptPenaltyParams3 ( mengerc_data *md );
```

The procedures with headers above use different methods of choosing the five penalty parameters. I guess that plenty of effort (theoretical research and experiments) are needed to replace these by something more reliable.

15. The libxgedit library

The procedures of the libxgedit library support the interaction between an application and a user, via the XWindow system. The library makes it possible to open a number of windows and create various widgets for user interaction, and it is used *instead* of more sophisticated packages built on top of XWindow. The library is intended mainly for use with the demonstration programs of the BSTools packages and perhaps it is good for nothing else.

15.1 Overview

15.2 Auxiliary #definitions

```
typedef unsigned int xgecolour_int;

#define xge_MAX_WINDOWS      8
#define xge_MAX_CURSORS     16

#define xge_MAX_WIDTH /*1024*/ 1280
#define xge_MAX_HEIGHT /* 768*/ 960
#define xge_WIDTH          480
#define xge_HEIGHT         360

#define XGE_AUTO_ASPECT
#ifndef XGE_AUTO_ASPECT
#define XGE_DEFAULT_ASPECT 1.0
#endif

#define xge_CHAR_WIDTH       6
#define xge_CHAR_HEIGHT     13

#define xge_RECT_NONE       -1
#define xge_MINDIST          8

#define xge_FOCUS_DEPTH      8

#define xge_MAX_STRING_LENGTH 512
```



```
#define xgemouse_LBUTTON_DOWN (1 << 0)
#define xgemouse_LBUTTON_CHANGE (1 << 1)
#define xgemouse_RBUTTON_DOWN (1 << 2)
#define xgemouse_RBUTTON_CHANGE (1 << 3)
#define xgemouse_MBUTTON_DOWN (1 << 4)
#define xgemouse_MBUTTON_CHANGE (1 << 5)
#define xgemouse_WHEELFW_DOWN (1 << 6)
#define xgemouse_WHEELFW_CHANGE (1 << 7)
#define xgemouse_WHEELBK_DOWN (1 << 8)
#define xgemouse_WHEELBK_CHANGE (1 << 9)

#define xgemsg_NULL                         0
#define xgemsg_INIT                     0x100

#define xgemsg_KEY                      0x101
#define xgemsg_SPECIAL_KEY              0x102
#define xgemsg_MMOVE                    0x103
#define xgemsg_MCLICK                   0x104
#define xgemsg_OTHEREVENT               0x105

#define xgemsg_ENTERING                 0x106
#define xgemsg_EXITING                  0x107
#define xgemsg_RESIZE                   0x108
#define xgemsg_MOVE                     0x109
#define xgemsg_BUTTON_COMMAND           0x10A
#define xgemsg_SWITCH_COMMAND           0x10B
#define xgemsg_SLIDEBAR_COMMAND         0x10C
#define xgemsg_SLIDEBAR2_COMMAND        0x10D
#define xgemsg_DIAL_COMMAND             0x10E
#define xgemsg_TEXT_EDIT_VERIFY         0x10F
#define xgemsg_TEXT_EDIT_ENTER          0x110
#define xgemsg_TEXT_EDIT_ESCAPE         0x111
#define xgemsg_INT_WIDGET_COMMAND       0x112
#define xgemsg_LISTBOX_ITEM_SET         0x113
#define xgemsg_LISTBOX_ITEM_PICK        0x114
#define xgemsg_QUATROTBALL_COMMAND      0x115

#define xgemsg_2DWIN_RESIZE             0x116
#define xgemsg_2DWIN_PROJCHANGE         0x117
#define xgemsg_2DWIN_PICK_POINT         0x118
#define xgemsg_2DWIN_MOVE_POINT         0x119
#define xgemsg_2DWIN_SELECT_POINTS      0x11A
```

```
#define xgemsg_2DWIN_UNSELECT_POINTS    0x11B
#define xgemsg_2DWIN_SPECIAL_SELECT     0x11C
#define xgemsg_2DWIN_SPECIAL_UNSELECT   0x11D
#define xgemsg_2DWIN_CHANGE_TRANS       0x11E
#define xgemsg_2DWIN_SAVE_POINTS        0x11F
#define xgemsg_2DWIN_TRANSFORM_POINTS   0x120
#define xgemsg_2DWIN_TRANSFORM_SPECIAL  0x121
#define xgemsg_2DWIN_FIND_REFBBOX       0x122
#define xgemsg_2DWIN_UNDO               0x123
#define xgemsg_2DWIN_KEY                0x124
#define xgemsg_2DWIN_ERROR              0x125

#define xgemsg_3DWIN_RESIZE             0x126
#define xgemsg_3DWIN_PROJCHANGE         0x127
#define xgemsg_3DWIN_PICK_POINT         0x128
#define xgemsg_3DWIN_MOVE_POINT         0x129
#define xgemsg_3DWIN_SELECT_POINTS      0x12A
#define xgemsg_3DWIN_UNSELECT_POINTS    0x12B
#define xgemsg_3DWIN_SPECIAL_SELECT     0x12C
#define xgemsg_3DWIN_SPECIAL_UNSELECT   0x12D
#define xgemsg_3DWIN_CHANGE_TRANS       0x12E
#define xgemsg_3DWIN_SAVE_POINTS        0x12F
#define xgemsg_3DWIN_TRANSFORM_POINTS   0x130
#define xgemsg_3DWIN_TRANSFORM_SPECIAL  0x131
#define xgemsg_3DWIN_FIND_REFBBOX       0x132
#define xgemsg_3DWIN_UNDO               0x133
#define xgemsg_3DWIN_KEY                0x134
#define xgemsg_3DWIN_ERROR              0x135

#define xgemsg_KNOTWIN_CHANGE_KNOT      0x136
#define xgemsg_KNOTWIN_INSERT_KNOT      0x137
#define xgemsg_KNOTWIN_REMOVE_KNOT      0x138
#define xgemsg_KNOTWIN_CHANGE_ALTKNOT   0x139
#define xgemsg_KNOTWIN_INSERT_ALTKNOT   0x13A
#define xgemsg_KNOTWIN_REMOVE_ALTKNOT   0x13B
#define xgemsg_KNOTWIN_MCLICK           0x13C
#define xgemsg_KNOTWIN_MMOVE            0x13D
#define xgemsg_KNOTWIN_CHANGE_MAPPING   0x13E
#define xgemsg_KNOTWIN_ERROR            0x13F

#define xgemsg_T2KNOTWIN_RESIZE         0x140
#define xgemsg_T2KNOTWIN_PROJCHANGE     0x141
```



```
#define xgemsg_T2KNOTWIN_CHANGE_KNOT_U  0x142
#define xgemsg_T2KNOTWIN_CHANGE_KNOT_V  0x143
#define xgemsg_T2KNOTWIN_INSERT_KNOT_U  0x144
#define xgemsg_T2KNOTWIN_INSERT_KNOT_V  0x145
#define xgemsg_T2KNOTWIN_REMOVE_KNOT_U  0x146
#define xgemsg_T2KNOTWIN_REMOVE_KNOT_V  0x147
#define xgemsg_T2KNOTWIN_CHANGE_ALTKNOT_U 0x148
#define xgemsg_T2KNOTWIN_CHANGE_ALTKNOT_V 0x149
#define xgemsg_T2KNOTWIN_INSERT_ALTKNOT_U 0x14A
#define xgemsg_T2KNOTWIN_INSERT_ALTKNOT_V 0x14B
#define xgemsg_T2KNOTWIN_REMOVE_ALTKNOT_U 0x14C
#define xgemsg_T2KNOTWIN_REMOVE_ALTKNOT_V 0x14D
#define xgemsg_T2KNOTWIN_SELECT_POINTS  0x14E
#define xgemsg_T2KNOTWIN_UNSELECT_POINTS 0x14F
#define xgemsg_T2KNOTWIN_CHANGE_MAPPING 0x150
#define xgemsg_T2KNOTWIN_ERROR          0x151

#define xgemsg_POPUP_REMOVED            0x152
#define xgemsg_POPUPS_REMOVED           0x153

#define xgemsg_USER_MESSAGE_DISMISSED   0x154

#define xgemsg_IDLE_COMMAND             0x155

#define xgemsg_CHILD_MESSAGE            0x156
#define xgemsg_CHILD_FAILURE            0x157

#define xgemsg_LAST_MESSAGE xgemsg_CHILD_FAILURE

#define xgestate_NOTHING                 0
#define xgestate_MOVINGSLIDE             1
#define xgestate_MOVINGSLIDE2A           2
#define xgestate_MOVINGSLIDE2B           3
#define xgestate_TURNINGDIAL             4
#define xgestate_QUATROT_TURNING1        5
#define xgestate_QUATROT_TURNING2        6
#define xgestate_QUATROT_TURNING3        7
#define xgestate_MESSAGE                 8
#define xgestate_RESIZING_X              9
#define xgestate_RESIZING_Y             10
#define xgestate_RESIZING_XY            11
#define xgestate_TEXT_EDITING           12
```

```
#define xgestate_2DWIN_MOVINGPOINT      13
#define xgestate_2DWIN_PANNING          14
#define xgestate_2DWIN_ZOOMING          15
#define xgestate_2DWIN_SELECTING        16
#define xgestate_2DWIN_UNSELECTING      17
#define xgestate_2DWIN_MOVING_GEOM_WIDGET 18
#define xgestate_2DWIN_USING_GEOM_WIDGET 19
#define xgestate_2DWIN_ALTUSING_GEOM_WIDGET 20
#define xgestate_2DWIN_USING_SPECIAL_WIDGET 21
#define xgestate_3DWIN_MOVINGPOINT      22
#define xgestate_3DWIN_PARPANNING       23
#define xgestate_3DWIN_PARZOOMING       24
#define xgestate_3DWIN_TURNING_VIEWER   25
#define xgestate_3DWIN_PANNING          26
#define xgestate_3DWIN_ZOOMING          27
#define xgestate_3DWIN_SELECTING        28
#define xgestate_3DWIN_UNSELECTING      29
#define xgestate_3DWIN_MOVING_GEOM_WIDGET 30
#define xgestate_3DWIN_USING_GEOM_WIDGET 31
#define xgestate_3DWIN_ALTUSING_GEOM_WIDGET 32
#define xgestate_3DWIN_USING_SPECIAL_WIDGET 33
#define xgestate_KNOTWIN_MOVINGKNOT     34
#define xgestate_KNOTWIN_PANNING        35
#define xgestate_KNOTWIN_ZOOMING        36
#define xgestate_T2KNOTWIN_MOVINGKNOT_U 37
#define xgestate_T2KNOTWIN_MOVINGKNOT_V 38
#define xgestate_T2KNOTWIN_MOVING_POINT 39
#define xgestate_T2KNOTWIN_PANNING      40
#define xgestate_T2KNOTWIN_ZOOMING      41
#define xgestate_T2KNOTWIN_SELECTING    42
#define xgestate_T2KNOTWIN_UNSELECTING  43
#define xgestate_LISTBOX_PICKING        44
#define xgestate_LAST xgestate_LISTBOX_PICKING

#define xgeCURSOR_CROSSHAIR xgecursor[0]
#define xgeCURSOR_HAND   xgecursor[1]
#define xgeCURSOR_PENCIL xgecursor[2]
#define xgeCURSOR_FLEUR  xgecursor[3]
#define xgeCURSOR_ARROW  xgecursor[4]
#define xgeCURSOR_WATCH  xgecursor[5]
#define xgeCURSOR_CIRCLE xgecursor[6]
#define xgeCURSOR_DEFAULT xgecursor[7]
```



```
#define xgeCURSOR_INVISIBLE xgecursor[8]
```

15.3 Colours

```
#ifndef XGERGB_H
#include "xgergb.h"
#endif

#define xgec_MENU_BACKGROUND    xgec_Grey5
#define xgec_INFOMSG_BACKGROUND xgec_Grey4
#define xgec_ERRORMSG_BACKGROUND xgec_Red
#define xgec_WARNINGMSG_BACKGROUND xgec_DarkMagenta
```

```
xgecolour_int xge_PixelColourf ( float r, float g, float b );
xgecolour_int xge_PixelColour ( byte r, byte g, byte b );
void xge_GetPixelColour ( xgecolour_int pixel,
                          byte *r, byte *g, byte *b );
void xge_GetPixelColourf ( xgecolour_int pixel,
                           float *r, float *g, float *b );
```

15.4 Xlib procedure wrappers

```
#define xgeSetBackground(colour) \
  XSetBackground(xgedisplay,xgegc,colour)
#define xgeSetForeground(colour) \
  XSetForeground(xgedisplay,xgegc,colour)
#define xgeSetLineAttributes(width,line_style,\
    cap_style,join_style) \
  XSetLineAttributes(xgedisplay,xgegc,width,line_style,cap_style, \
    join_style)
#define xgeSetDashes(n,dash_list,offset) \
  XSetDashes(xgedisplay,xgegc,offset,dash_list,n)

#define xgeDrawRectangle(w,h,x,y) \
  XDrawRectangle(xgedisplay,xgepixmap,xgegc,x,y,w,h)
#define xgeFillRectangle(w,h,x,y) \
  XFillRectangle(xgedisplay,xgepixmap,xgegc,x,y,w,h)
#define xgeDrawString(string,x,y) \
  XDrawString(xgedisplay,xgepixmap,xgegc,x,y,string,strlen(string))
```

```
#define xgeDrawLine(x0,y0,x1,y1) \
  XDrawLine(xgedisplay,xgepixmap,xgegc,x0,y0,x1,y1)
#define xgeDrawLines(n,p) \
  XDrawLines(xgedisplay,xgepixmap,xgegc,p,n,CoordModeOrigin)
#define xgeDrawArc(w,h,x,y,a0,a1) \
  XDrawArc(xgedisplay,xgepixmap,xgegc,x,y,w,h,a0,a1)
#define xgeFillArc(w,h,x,y,a0,a1) \
  XFillArc(xgedisplay,xgepixmap,xgegc,x,y,w,h,a0,a1)
#define xgeDrawPoint(x,y) \
  XDrawPoint(xgedisplay,xgepixmap,xgegc,x,y)
#define xgeDrawPoints(n,p) \
  XDrawPoints(xgedisplay,xgepixmap,xgegc,p,n,CoordModeOrigin)
#define xgeFillPolygon(shape,n,p) \
  XFillPolygon(xgedisplay,xgepixmap,xgegc,p,n,shape,CoordModeOrigin)

#define xgeCopyRectOnScreen(w,h,x,y) \
  XCopyArea(xgedisplay,xgepixmap,xgewindow,xgegc,x,y,w,h,x,y)
#define xgeRaiseWindow() \
  XRaiseWindow(xgedisplay,xgewindow)
#define xgeResizeWindow(w,h) \
  XResizeWindow(xgedisplay,xgewindow,w,h)
#define xgeMoveWindow(x,y) \
  XMoveWindow(xgedisplay,xgewindow,x,y)
```

15.5 Global variables

```
extern int  xge_winnum;
extern unsigned int xge_mouse_buttons;
extern int  xge_mouse_x, xge_mouse_y;
extern short xge_xx, xge_yy;

extern Display   *xgedisplay;
extern XVisualInfo *xgevisualinfo;
extern Colormap  xgecolormap;
extern int       xgescreen;
extern Window    xgeroot;
extern Window    xgewindow;
extern Pixmap    xgepixmap;
extern GC        xgegc;
extern Visual    *xgevisual;
extern XSizeHints xgehints;
```



```
extern Cursor    xgecursor[];
extern KeySym    xgekeysym;
extern XEvent    xgeevent;

typedef struct {
        unsigned short r_bits, g_bits, b_bits;
        char          nr_bits, ng_bits, nb_bits;
        unsigned char r_shift, g_shift, b_shift;
        unsigned char r_mask, g_mask, b_mask;
        float         ar, ag, ab;
      } xge_rgbmap_bits;
extern xge_rgbmap_bits xge_rgbmap;

extern float    xge_aspect;

extern unsigned int xge_current_width, xge_current_height;

extern char     *xge_p_name;
extern char     xge_done;
extern short    xge_prevx, xge_prevy;

extern xgecolour_int xge_foreground, xge_background;
extern int         xge_nplanes;
extern xgecolour_int *xge_palette;
extern const char  *xge_colour_name[];
extern xge_widget  *xge_null_widget;
```

15.6 Widgets

```
typedef struct xge_widget {
         int  id;
         short w, h, x, y;
         void *data0, *data1, *data2;
         short xofs, yofs;
         char rpos;
         char window_num;
         short state;
         boolean (*msgproc)(struct xge_widget*,int,int,short,short);
         void (*redraw) ( struct xge_widget*, boolean );
         struct xge_widget *next, *prev, *up;
       } xge_widget;
```

```
typedef boolean (*xge_message_proc) ( struct xge_widget *er,
                                int msg, int key, short x, short y );
typedef void (*xge_redraw_proc) ( struct xge_widget *er,
                                boolean onscreen );
```

15.6.1 Generic widget constructor

```
xge_widget *xge_NewWidget (
        char window_num, xge_widget *prev, int id,
        short w, short h, short x, short y,
        void *data0, void *data1,
        boolean (*msgproc)(xge_widget*, int, int, short, short),
        void (*redraw)(xge_widget*, boolean) );
void xge_SetWidgetPositioning ( xge_widget *edr,
                               char rpos, short xofs, short yofs );
```

15.6.2 Empty widget

```
void xge_DrawEmpty ( xge_widget *er, boolean onscreen );
boolean xge_EmptyMsg ( xge_widget *er,
                      int msg, int key, short x, short y );
xge_widget *xge_NewEmptyWidget ( char window_num,
                               xge_widget *prev, int id,
                               short w, short h, short x, short y );
```

15.6.3 Menu widgets

```
void xge_DrawMenu ( xge_widget *er, boolean onscreen );
boolean xge_MenuMsg ( xge_widget *er,
                     int msg, int key, short x, short y );
boolean xge_PopupMenuMsg ( xge_widget *er,
                          int msg, int key, short x, short y );
xge_widget *xge_NewMenu ( char window_num, xge_widget *prev,
                        int id,
                        short w, short h, short x, short y,
                        xge_widget *widgetlist );
void xge_DrawFMenu ( xge_widget *er, boolean onscreen );
xge_widget *xge_NewFMenu ( char window_num, xge_widget *prev,
```



```
                        int id,
                        short w, short h, short x, short y,
                        xge_widget *widgetlist );
void xge_SetMenuWidgets ( xge_widget *menu, xge_widget *widgetlist,
                         boolean redraw );
```

15.6.4 Switch widget

```
void xge_DrawSwitch ( xge_widget *er, boolean onscreen );
boolean xge_SwitchMsg ( xge_widget *er,
                       int msg, int key, short x, short y );
xge_widget *xge_NewSwitch ( char window_num, xge_widget *prev,
                          int id,
                          short w, short h, short x, short y,
                          char *title, boolean *sw );
```

15.6.5 Button widget

```
void xge_DrawButton ( xge_widget *er, boolean onscreen );
boolean xge_ButtonMsg ( xge_widget *er,
                       int msg, int key, short x, short y );
xge_widget *xge_NewButton ( char window_num, xge_widget *prev,
                          int id,
                          short w, short h, short x, short y,
                          char *title );
```

15.6.6 Slidebar widgets

```
void xge_DrawSlidebarf ( xge_widget *er, boolean onscreen );
boolean xge_SlidebarfMsg ( xge_widget *er,
                          int msg, int key, short x, short y );
xge_widget *xge_NewSlidebarf ( char window_num, xge_widget *prev,
                             int id,
                             short w, short h, short x, short y,
                             float *data );

void xge_DrawSlidebarfRGB ( xge_widget *er, boolean onscreen );
xge_widget *xge_NewSlidebarfRGB ( char window_num, xge_widget *prev,
                                int id,
```

```
                              short w, short h, short x, short y,
                              float *data );

void xge_DrawVSlidebarf ( xge_widget *er, boolean onscreen );
boolean xge_VSlidebarfMsg ( xge_widget *er,
                            int msg, int key, short x, short y );
xge_widget *xge_NewVSlidebarf ( char window_num, xge_widget *prev,
                              int id,
                              short w, short h, short x, short y,
                              float *data );

void xge_DrawSlidebar2f ( xge_widget *er, boolean onscreen );
boolean xge_Slidebar2fMsg ( xge_widget *er,
                            int msg, int key, short x, short y );
xge_widget *xge_NewSlidebar2f ( char window_num, xge_widget *prev,
                              int id,
                              short w, short h, short x, short y,
                              float *data );

void xge_DrawVSlidebar2f ( xge_widget *er, boolean onscreen );
boolean xge_VSlidebar2fMsg ( xge_widget *er,
                             int msg, int key, short x, short y );
xge_widget *xge_NewVSlidebar2f ( char window_num, xge_widget *prev,
                               int id,
                               short w, short h, short x, short y,
                               float *data );

float xge_LinSlidebarValuef ( float xmin, float xmax, float t );
float xge_LogSlidebarValuef ( float xmin, float xmax, float t );
```

15.6.7 Dial widget

```
void xge_DrawDialf ( xge_widget *er, boolean onscreen );
boolean xge_DialfMsg ( xge_widget *er,
                       int msg, int key, short x, short y );
xge_widget *xge_NewDialf ( char window_num, xge_widget *prev,
                           int id,
                           short w, short h, short x, short y,
                           char *title, float *data );
```


15.6.8 Quaternion ball widget

```
typedef struct xge_quatrotballf {
   xge_widget *er;
   quaternionf *q;
   trans3f    *tr;
   vector3f   axis;
   short      xc, yc, r1, r2, R;
   boolean    axis_ok, insert;
 } xge_quatrotballf;

void xge_DrawQuatRotBallf ( xge_widget *er, boolean onscreen );
boolean xge_QuatRotBallfMsg ( xge_widget *er,
                              int msg, int key, short x, short y );
xge_widget *xge_NewQuatRotBallf ( char window_num, xge_widget *prev,
                          int id,
                          short w, short h, short x, short y, short R,
                          xge_quatrotballf *qball, quaternionf *q,
                          trans3f *tr, char *title );
```

15.6.9 Text output widget

```
void xge_DrawText ( xge_widget *er, boolean onscreen );
xge_widget *xge_NewTextWidget ( char window_num, xge_widget *prev,
                                int id,
                                short w, short h, short x, short y,
                                char *text );
```

15.6.10 Colour sample widget

```
void xge_DrawRGBSamplef ( xge_widget *er, boolean onscreen );
xge_widget *xge_NewRGBSamplef ( char window_num, xge_widget *prev,
                                int id,
                                short w, short h, short x, short y,
                                float *data );
```

15.6.11 Text editing widget

```
typedef struct xge_string_ed {
   xge_widget *er;
   short maxlength, /* maximal string length */
         chdisp,    /* number of characters displayed */
         start,     /* first character displayed */
         pos;       /* text cursor position */
  } xge_string_ed;

void xge_DrawStringEd ( xge_widget *er, boolean onscreen );
boolean xge_StringEdMsg ( xge_widget *er,
                          int msg, int key, short x, short y );
xge_widget *xge_NewStringEd ( char window_num, xge_widget *prev,
                  int id,
                  short w, short h, short x, short y,
                  short maxlength, char *text, xge_string_ed *ed );
```

15.6.12 Integer widget

```
typedef struct xge_int_widget {
   xge_widget *er;
   int       minvalue, maxvalue;
   char      *title;
  } xge_int_widget;

void xge_DrawIntWidget ( xge_widget *er, boolean onscreen );
boolean xge_IntWidgetMsg ( xge_widget *er,
                           int msg, int key, short x, short y );
xge_widget *xge_NewIntWidget ( char window_num, xge_widget *prev,
                  int id,
                  short w, short h, short x, short y,
                  int minvalue, int maxvalue,
                  xge_int_widget *iw, char *title, int *valptr );
```


15.6.13 List widgets

```
#define xge_LISTDIST 16

typedef struct xge_listbox {
   xge_widget  *er;
   char         dlistnpos;  /* number of positions displayed */
   char         maxitl;     /* maximal item length, in characters */
   short        nitems;     /* current number of list elements */
   short        fditem;     /* first displayed item */
   short        currentitem; /* current item */
   int          *itemind;   /* indexes to the item strings */
   char         *itemstr;   /* item strings */
   xgecolour_int bk0, bk1; /* background colours for the items */
  } xge_listbox;

void xge_DrawListBox ( xge_widget *er, boolean onscreen );
boolean xge_ListBoxMsg ( xge_widget *er,
                         int msg, int key, short x, short y );
xge_widget *xge_NewListBox ( char window_num, xge_widget *prev,
                           int id,
                           short w, short h, short x, short y,
                           xge_listbox *listbox );
void xge_ClearListBox ( xge_listbox *lbox );
void xge_ShortenString ( const char *s, char *buf, int maxlen );
boolean xge_GetCurrentListBoxString ( xge_listbox *lbox,
                                      char *string );
int xge_MoveInListBox ( xge_listbox *lbox, short amount );
```

```
boolean xge_SetupFileList ( xge_listbox *lbox, const char *dir,
                            const char *filter );
boolean xge_SetupDirList ( xge_listbox *lbox, const char *dir,
                           const char *filter, const char *prevdir );
boolean xge_FilterMatches ( const char *name, const char *filter );
```

15.6.14 2D geometry editing widget

```
#define xge_2DWIN_MIN_ZOOM         0.01
#define xge_2DWIN_MAX_ZOOM       100.0

#define xge_2DWIN_NO_TOOL            0
#define xge_2DWIN_MOVING_TOOL        1
#define xge_2DWIN_SCALING_TOOL       2
#define xge_2DWIN_ROTATING_TOOL      3
#define xge_2DWIN_SHEAR_TOOL         4
#define xge_2DWIN_SELECTING_TOOL     5
#define xge_2DWIN_PANNING_TOOL       6
#define xge_2DWIN_SPECIAL_SELECTING_TOOL 7
#define xge_2DWIN_SPECIAL_TRANS_TOOL 8
```

```
typedef struct xge_2Dwinf {
   xge_widget *er;
   CameraRecf CPos;
   Box2f      DefBBox, RefBBox;
   boolean    panning, selecting_mode, special_selecting_mode;
   boolean    display_coord, inside;
   boolean    moving_tool, scaling_tool, rotating_tool, shear_tool,
              special_trans_tool;
   char       current_tool;
   char       tool_mode;
   int        current_tab, current_point;
   short      xx, yy;
   float      zoom;
   Box2s      selection_rect;
   point2f    saved_centre;
   point2f    scaling_centre;
   vector2f   scaling_factors;
   short      scaling_size;
   point2f    rotating_centre;
   short      rotating_radius;
   vector2f   trans_params;
   point2f    shear_centre;
   vector2f   shear_axis[2];
   float      shear_radius;
   trans2f    gwtrans;
 } xge_2Dwinf;
```



```
boolean xge_2DwinfMsg ( xge_widget *er,
                        int msg, int key, short x, short y );
xge_widget *xge_New2Dwinf ( char window_num, xge_widget *prev,
                            int id,
                            short w, short h, short x, short y,
                            xge_2Dwinf *_2Dwin,
                            void (*redraw)(xge_widget*, boolean) );
```

```
void xge_2DwinfSetDefBBox ( xge_2Dwinf *_2Dwin,
                            float x0, float x1, float y0, float y1 );
void xge_2DwinfSetupProjection ( xge_2Dwinf *_2Dwin );
void xge_2DwinfPan ( xge_widget *er, short x, short y );
void xge_2DwinfZoom ( xge_widget *er, short y );
void xge_2DwinfInitProjection ( xge_2Dwinf *_2Dwin,
                            float x0, float x1, float y0, float y1 );
void xge_2DwinfResetGeomWidgets ( xge_2Dwinf *_2Dwin );
void xge_2DwinfResetGeomWidgetPos ( xge_2Dwinf *_2Dwin );
void xge_2DwinfEnableGeomWidget ( xge_2Dwinf *_2Dwin, char tool );
void xge_2DwinfDrawGeomWidgets ( xge_widget *er );
char xge_2DwinfIsItAGeomWidget ( xge_2Dwinf *_2Dwin,
                                 int key, short x, short y );
void xge_2DwinfMoveGeomWidget ( xge_2Dwinf *_2Dwin,
                                short x, short y );
boolean xge_2DwinfApplyGeomWidget ( xge_2Dwinf *_2Dwin,
                                   short x, short y, boolean alt );
void xge_2DwinfExitWidgetMode ( xge_2Dwinf *_2Dwin );
void xge_2DwinfResetGeomWidget ( xge_2Dwinf *_2Dwin );
void xge_2DwinfDrawCursorPos ( xge_2Dwinf *_2Dwin,
                               short x, short y );
```

15.6.15 Four window widget

```
typedef struct xge_fourww {
   xge_widget *er;
   xge_widget *win[4];
   float     xsfr, ysfr;
   short     splitx, splity;
   boolean   resized;
   char      zoomwin;
 } xge_fourww;
```

```
boolean xge_CompSizeFourWW ( xge_widget *er, char cs );
void xge_DrawFourWW ( xge_widget *er, boolean onscreen );
boolean xge_FourWWMsg ( xge_widget *er,
                        int msg, int key, short x, short y );
xge_widget *xge_NewFourWW ( char window_num, xge_widget *prev,
                            int id,
                            short w, short h, short x, short y,
                            xge_widget *ww, xge_fourww *fwwdata );
```

15.6.16 3D geometry editing widget

```
#define xge_3DWIN_MIN_PARZOOM 0.01
#define xge_3DWIN_MAX_PARZOOM 100.0
#define xge_3DWIN_MIN_ZOOM   0.1
#define xge_3DWIN_MAX_ZOOM 1000.0

#define xge_3DWIN_NO_TOOL            xge_2DWIN_NO_TOOL
#define xge_3DWIN_MOVING_TOOL        xge_2DWIN_MOVING_TOOL
#define xge_3DWIN_SCALING_TOOL       xge_2DWIN_SCALING_TOOL
#define xge_3DWIN_ROTATING_TOOL      xge_2DWIN_ROTATING_TOOL
#define xge_3DWIN_SHEAR_TOOL         xge_2DWIN_SHEAR_TOOL
#define xge_3DWIN_SELECTING_TOOL     xge_2DWIN_SELECTING_TOOL
#define xge_3DWIN_PANNING_TOOL       xge_2DWIN_PANNING_TOOL
#define xge_3DWIN_SPECIAL_SELECTING_TOOL \
                              xge_2DWIN_SPECIAL_SELECTING_TOOL
#define xge_3DWIN_SPECIAL_TRANS_TOOL xge_2DWIN_SPECIAL_TRANS_TOOL
```

```
typedef struct xge_3Dwinf {
    xge_fourww fww;       /* this must be the first component */
    xge_widget *cwin[4];
    CameraRecf CPos[5];
    Box3f      DefBBox, RefBBox, WinBBox, PerspBBox;
    boolean    panning, selecting_mode, special_selecting_mode;
    boolean    display_coord;
    signed char CoordWin;
    boolean    moving_tool, scaling_tool, rotating_tool, shear_tool,
               special_trans_tool;
    char       current_tool;
    char       tool_mode;
    int        current_tab, current_point;
    short      xx, yy;
```



```
    float      perspzoom;
    Box2s      selection_rect;
    point3f    saved_centre;
    point3f    scaling_centre;
    vector3f   scaling_factors;
    short      scaling_size;
    point3f    rotating_centre;
    short      rotating_radius;
    vector3f   trans_params;
    point3f    shear_centre;
    vector3f   shear_axis[3];
    float      shear_radius;
    trans3f    gwtrans;
  } xge_3Dwinf;
```

```
xge_widget *xge_New3Dwinf ( char window_num, xge_widget *prev,
                        int id,
                        short w, short h, short x, short y,
                        xge_3Dwinf *_3Dwin,
                        void (*pararedraw)(xge_widget*, boolean),
                        void (*perspredraw)(xge_widget*, boolean) );

void xge_3DwinfSetDefBBox ( xge_3Dwinf *_3Dwin, float x0, float x1,
                            float y0, float y1, float z0, float z1 );
void xge_3DwinfSetupParProj ( xge_3Dwinf *_3Dwin, Box3f *bbox );
void xge_3DwinfSetupPerspProj ( xge_3Dwinf *_3Dwin,
                                boolean resetpos );
void xge_3DwinfUpdatePerspProj ( xge_3Dwinf *_3Dwin );
void xge_3DwinfPanParWindows ( xge_widget *er, short x, short y );
void xge_3DwinfZoomParWindows ( xge_widget *er, short y );
void xge_3DwinfPanPerspWindow ( xge_widget *er, short x, short y );
void xge_3DwinfInitProjections ( xge_3Dwinf *_3Dwin,
      float x0, float x1, float y0, float y1, float z0, float z1 );
void xge_3DwinfResetGeomWidgets ( xge_3Dwinf *_3Dwin );
void xge_3DwinfResetGeomWidgetPos ( xge_3Dwinf *_3Dwin );
void xge_3DwinfEnableGeomWidget ( xge_3Dwinf *_3Dwin, char tool );
void xge_3DwinfDrawCursorPos ( xge_3Dwinf *_3Dwin,
                               int id, short x, short y );
void xge_3DwinfDrawSelectionRect ( xge_widget *er );
void xge_3DwinfDrawGeomWidgets ( xge_widget *er );
char xge_3DwinfIsItAGeomWidget ( xge_3Dwinf *_3Dwin, int id,
                                 int key, short x, short y );
```

```
void xge_3DwinfMoveGeomWidget ( xge_3Dwinf *_3Dwin,
                            int id, short x, short y );
boolean xge_3DwinfApplyGeomWidget ( xge_3Dwinf *_3Dwin,
                      int id, short x, short y, boolean alt );
void xge_3DwinfExitWidgetMode ( xge_3Dwinf *_3Dwin );
void xge_3DwinfResetGeomWidget ( xge_3Dwinf *_3Dwin );
void xge_3DwinfSavePerspCamera ( xge_3Dwinf *_3Dwin );
void xge_3DwinfSwapPerspCameras ( xge_3Dwinf *_3Dwin );
```

15.6.17 Knot sequence editing widget

```
#define xge_KNOTWIN_MIN_SCALE 0.01
#define xge_KNOTWIN_MAX_SCALE 100.0
#define xge_KNOT_EPS       1.0e-4
```

```
typedef struct {
   xge_widget   *er;
   boolean      panning, display_coord, moving_many, locked;
   boolean      closed, altkn, switchkn;
   float        akm, bkm, umin, umax, knotscf, knotshf;
   int          clcK;
   float        clcT;
   unsigned char current_mult;
   short        xx;
   int          maxknots, lastknot, degree;
   float        *knots;
   int          maxaltknots, lastaltknot, altdegree;
   float        *altknots;
   float        newknot;
   int          current_knot;
  } xge_KnotWinf;
```

```
void xge_DrawKnotWinf ( xge_widget *er, boolean onscreen );
boolean xge_KnotWinfMsg ( xge_widget *er,
                        int msg, int key, short x, short y );
xge_widget *xge_NewKnotWinf ( char window_num, xge_widget *prev,
                    int id,
                    short w, short h, short x, short y,
                    xge_KnotWinf *knw, int maxknots, float *knots );

void xge_KnotWinfDrawCursorPos ( xge_KnotWinf *knw );
```



```
void xge_KnotWinfDrawAxis ( xge_KnotWinf *knw );
void xge_KnotWinfDrawKnots ( xge_KnotWinf *knw );
void xge_KnotWinfInitMapping ( xge_KnotWinf *knw,
                             float umin, float umax );
void xge_KnotWinfZoom ( xge_KnotWinf *knw, float scf );
void xge_KnotWinfPan ( xge_KnotWinf *knw, int dxi );
void xge_KnotWinfFindMapping ( xge_KnotWinf *knw );
void xge_KnotWinfResetMapping ( xge_KnotWinf *knw );
short xge_KnotWinfMapKnot ( xge_KnotWinf *knw, float u );
float xge_KnotWinfUnmapKnot ( xge_KnotWinf *knw, short xi );
boolean xge_KnotWinfFindNearestKnot ( xge_KnotWinf *knw,
                                   int x, int y );
boolean xge_KnotWinfSetKnot ( xge_KnotWinf *knw, short x );
boolean xge_KnotWinfInsertKnot ( xge_KnotWinf *knw, short x );
boolean xge_KnotWinfRemoveKnot ( xge_KnotWinf *knw );
void xge_KnotWinfSetAltKnots ( xge_KnotWinf *knw,
        int altmaxkn, int lastaltkn, int altdeg, float *altknots );
void xge_KnotWinfSwitchAltKnots ( xge_KnotWinf *knw );
```

15.6.18 Two knot sequences editing widget

```
typedef struct {
   xge_widget *er;
   CameraRecf CPos;
   Box2f      DefBBox, RefBBox;
   point3f    centre;
   boolean    panning, selecting_mode, moving_many,
              locked_u, locked_v;
   boolean    display_coord, inside;
   unsigned char current_mult;
   int        current_item;
   short      knot_margin;
   short      xx, yy;
   float      zoom;
   Box2s      selection_rect;
   boolean    closed_u, closed_v;
   int        clcKu, clcKv;
   float      clcTu, clcTv;
   int        maxknots_u, lastknot_u, degree_u;
   float      *knots_u;
   int        maxknots_v, lastknot_v, degree_v;
```

```
   float      *knots_v;
   float      newknot;
   boolean    altknu, switchknu, altknv, switchknv;
   int        altmaxknots_u, altlastknot_u, altdeg_u;
   float      *altknots_u;
   int        altmaxknots_v, altlastknot_v, altdeg_v;
   float      *altknots_v;
 } xge_T2KnotWinf;
```

```
boolean xge_T2KnotWinfMsg ( xge_widget *er,
                        int msg, int key, short x, short y );
xge_widget *xge_NewT2KnotWinf ( char window_num, xge_widget *prev,
                            int id,
                            short w, short h, short x, short y,
                            short knot_margin,
                            xge_T2KnotWinf *T2win,
                            void (*redraw)(xge_widget*, boolean),
                            int maxknots_u, float *knots_u,
                            int maxknots_v, float *knots_v );

void xge_T2KnotWinfDrawKnots ( xge_T2KnotWinf *T2win );
void xge_T2KnotWinfSetupMapping ( xge_T2KnotWinf *T2win );
void xge_T2KnotWinfInitMapping ( xge_T2KnotWinf *T2win,
                    float umin, float umax, float vmin, float vmax );
void xge_T2KnotWinfZoom ( xge_T2KnotWinf *T2win, short y );
boolean xge_T2KnotWinfPan ( xge_T2KnotWinf *T2win,
                        short x, short y );
void xge_T2KnotWinfFindMapping ( xge_T2KnotWinf *T2win );
void xge_T2KnotWinfResetMapping ( xge_T2KnotWinf *T2win );

char xge_T2KnotWinfFindDomWinRegion ( xge_T2KnotWinf *T2win,
                                  int x, int y );
char xge_T2KnotWinfFindNearestKnot ( xge_T2KnotWinf *T2win,
                                 int x, int y );
short xge_T2KnotWinfMapKnotU ( xge_T2KnotWinf *T2win, float u );
float xge_T2KnotWinfUnmapKnotU ( xge_T2KnotWinf *T2win, short xi );
short xge_T2KnotWinfMapKnotV ( xge_T2KnotWinf *T2win, float v );
float xge_T2KnotWinfUnmapKnotV ( xge_T2KnotWinf *T2win, short eta );
boolean xge_T2KnotWinfSetKnotU ( xge_T2KnotWinf *T2win, short x );
boolean xge_T2KnotWinfInsertKnotU ( xge_T2KnotWinf *T2win,
                                short x );
boolean xge_T2KnotWinfRemoveKnotU ( xge_T2KnotWinf *T2win );
```



```
boolean xge_T2KnotWinfSetKnotV ( xge_T2KnotWinf *T2win, short y );
boolean xge_T2KnotWinfInsertKnotV ( xge_T2KnotWinf *T2win,
                                  short y );
boolean xge_T2KnotWinfRemoveKnotV ( xge_T2KnotWinf *T2win );
void xge_T2KnotWinfSelect ( xge_T2KnotWinf *T2win,
                          short x0, short x1, short y0, short y1 );
void xge_T2KnotWinfUnselect ( xge_T2KnotWinf *T2win,
                            short x0, short x1, short y0, short y1 );

void xge_T2KnotWinfSetAltKnots ( xge_T2KnotWinf *T2win,
              int altmaxknu, int lastaltknu, int altdegu,
              float *altknotsu,
              int altmaxknv, int lastaltknv, int altdegv,
              float *altknotsv );
void xge_T2KnotWinfSwitchAltKnots ( xge_T2KnotWinf *T2win,
              boolean altu, boolean altv );
void xge_T2KnotWinfDrawCursorPos ( xge_T2KnotWinf *T2win,
                                 short x, short y );
```

15.6.19 Scrolling widget

```
typedef struct {
   xge_widget *er;
   xge_widget *contents, *clipw, *xsl, *ysl;
   float     x, y;
   boolean   xslon, yslon;
 } xge_scroll_widget;

void xge_SetupScrollWidgetPos ( xge_widget *er );
void xge_DrawScrollWidget ( xge_widget *er, boolean onscreen );
boolean xge_ScrollWidgetMsg ( xge_widget *er,
                            int msg, int key, short x, short y );
xge_widget *xge_NewScrollWidget ( char window_num, xge_widget *prev,
                    int id,
                    short w, short h, short x, short y,
                    xge_scroll_widget *sw, xge_widget *contents );
```

15.7 Input focus processing

```
void xge_GrabFocus ( xge_widget *er, boolean all );
void xge_ReleaseFocus ( xge_widget *er );
xge_widget *xge_GetFocusWidget ( char win );
```

15.8 Popup widgets

```
void xge_AddPopup ( xge_widget *er );
void xge_RemovePopup ( boolean redraw );
void xge_RemovePopups ( boolean redraw );
boolean xge_IsPopupOn ( xge_widget *er );

void _xge_DisplayErrorMessage ( char *message,
                            xgecolour_int bk, int key );
void xge_DisplayErrorMessage ( char *message, int key );
void xge_DisplayWarningMessage ( char *message, int key );
void xge_DisplayInfoMessage ( char **msglines, int key );
```

15.9 Application initialisation, message loop and closing

```
void xge_Init ( int argc, char *argv[],
              int (*callback)(xge_widget*,int,int,short,short),
              char *title );
void xge_MessageLoop ( void );
void xge_Cleanup ( void );
```

15.10 Other procedures

```
void xge_OutPixels ( const xpoint *buf, int n );
void xge_DrawBC2f ( int n, const point2f *cp );
void xge_DrawBC2Rf ( int n, const point3f *cp );
void xge_DrawBC2d ( int n, const point2d *cp );
void xge_DrawBC2Rd ( int n, const point3d *cp );

int    xge_NewWindow ( char *title );
```



```
boolean xge_SetWindow ( int win );
int    xge_CurrentWindow ( void );
void   xge_SetWinEdRect ( xge_widget *edr );

int  xge_NewCursor ( int shape );
void xge_SetWindowCursor ( int win, Cursor cursor );
void xge_SetCurrentWindowCursor ( Cursor cursor );
void xge_SetOtherWindowsCursor ( Cursor cursor );

void xge_RedrawPopups ( void );
void xge_Redraw ( void );
void xge_RedrawAll ( void );

boolean xge_PointInRect ( xge_widget *edr, short x, short y );
void xge_BoundPoint ( xge_widget *er, short *x, short *y );
boolean xge_RectanglesIntersect (
             short wa, short ha, short xa, short ya,
             short wb, short hb, short xb, short yb );

boolean xge_IntersectXRectangles ( XRectangle *r1, XRectangle *r2 );
boolean xge_SetClipping ( xge_widget *edr );
void xge_ResetClipping ( void );

void xge_RepositionWidgets ( short w, short h, short x, short y,
                           xge_widget *edr );

void xge_DrawVShadedRect ( short w, short h, short x, short y,
                  xgecolour_int c0, xgecolour_int c1, short nb );
void xge_DrawHShadedRect ( short w, short h, short x, short y,
                  xgecolour_int c0, xgecolour_int c1, short nb );

void xge_OrderSelectionRect ( Box2s *sel_rect );
void xge_DrawGeomWinBackground ( xge_widget *er );
void xge_DrawGeomWinFrame ( xge_widget *er, boolean onscreen );
void xge_DrawGeomWinSelectionRect ( xge_widget *er,
                                  Box2s *sel_rect );

void xge_GetWindowSize ( void );
void xge_PostIdleCommand ( unsigned int key, short x, short y );

void xge_dispatch_message ( unsigned int msg, unsigned int key,
                          short x, short y );
```

```
void xge_get_message ( unsigned int *msg, unsigned int *key,
                       short *x, short *y );
```

15.11 OpenGL support

```
#define xgleCopyGLRect(w,h,x,y) \
  XCopyArea(xgedisplay,xglepixmap,xgepixmap,xgegc,0,\
    xge_MAX_HEIGHT-h,w,h,x,y)
#define xgleClearColor3fv(rgb) \
  glClearColor(rgb[0],rgb[1],rgb[2],1.0)
```

```
extern Pixmap    xglepixmap;
extern XID       _xglepixmap;
extern void      *xglecontext;

extern GLfloat xgle_palette[XGLE_PALETTE_LENGTH][3];
```

```
void xgle_Init ( int argc, char *argv[],
                 int (*callback)(xge_widget*,int,int,short,short),
                 char *title,
                 boolean depth, boolean accum, boolean stencil );
void xgle_Cleanup ( void );
```

```
void xgle_SetIdentMapping ( xge_widget *er );
```

```
boolean xgle_SetGLCameraf ( CameraRecf *CPos );
boolean xgle_SetGLCamerad ( CameraRecd *CPos );
boolean xgle_SetGLaccCameraf ( CameraRecf *CPos,
                       float pixdx, float pixdy,
                       float eyedx, float eyedy, float focus );
boolean xgle_SetGLaccCamerad ( CameraRecd *CPos,
                       double pixdx, double pixdy,
                       double eyedx, double eyedy, double focus );
```

```
void xgle_MultMatrix3f ( trans3f *tr );

void xgleDrawPoint ( int x, int y );
void xgleDrawPoints ( int n, XPoint *p );
void xgleDrawLine ( int x0, int y0, int x1, int y1 );
void xgleDrawRectangle ( int w, int h, int x, int y );
void xgleDrawString ( char *string, int x, int y );
```



```
void xgle_DrawGeomWinBackground ( xge_widget *er, GLbitfield mask );

void xgle_2DwinfDrawCursorPos ( xge_2Dwinf *_2Dwin,
                                short x, short y );

void xgle_3DwinfDrawCursorPos ( xge_3Dwinf *_3Dwin,
                                int id, short x, short y );

void xgle_T2KnotWinfDrawCursorPos ( xge_T2KnotWinf *T2win,
                                    short x, short y );
```

15.12 Interprocess communication

15.12.1 Overview

15.12.2 Common variables

```
extern pid_t xge_parent_pid, xge_child_pid;
extern int xge_pipe_in[2], xge_pipe_out[2];
extern Window xgeparentwindow, xgechildwindow;
```

15.12.3 Parent side procedures

```
boolean xge_MakeTheChild ( const char *name,
                           const char *suffix, int magic );
boolean xge_ChildIsActive ( void );
void xge_CallTheChild ( int cmd, int size );
void xge_SignalTheChild ( void );
void xge_ParentFlushPipe ( void );
```

15.12.4 Child side procedures

```
extern void (*xge_childcallback) ( int msg, int size );

void xge_CallTheParent ( int cmd, int size );
void xge_ChildCallYourself ( int cmd );
void xge_ChildMessageLoop ( void );
void xge_ChildFlushPipe ( void );
```

```
boolean xge_ChildInit ( int argc, char **argv, int magic,
                        void (*callback) ( int msg, int size ) );
```

16. Demonstration programs

The demonstration programs accompanying the libraries (in the `demo` directory subtree) are meant to reach three goals. Firstly, they display movable pictures, which may help to learn about the curves and surfaces. They are also tests of the library procedures (plenty of errors were extincted after detecting them in these programs). Finally, the demonstration programs may serve as sources of information on the way of using the library procedures in new applications.

The programs were developed in a not very systematic way, with new possibilities added when I fancied them. Therefore **they are not** examples of a particularly good programming style. The basic assumption was that apart from a working XWindow environment, no special libraries or packages (e.g. Gnome, KDE, Athena, Motif, OpenGL) are present. The programs use only the `Xlib` interface, and therefore it should be possible to compile and execute them on any computer equipped with XWindow. The people unsatisfied with the possibilities or with the user interface are welcome to write their own programs; no doubt that they will be much better.

16.1 The `pokrzyw` program

The program `pokrzyw`[1] (directory `demo/pokrzyw`) displays and makes it possible to bend a planar (polynomial or rational) B-spline curve of degree from 1 to 8. Two windows on the right side of the screen (of the program window opened by XWindow) show the image of the curve (with the control polygon and other objects) and the knot sequence.

Most of commands are given using the left button of the mouse. It serves for "picking" and "holding" the control points and knots, as well as for clicking the widgets (buttons etc.) in the menu on the left side.

The right button is used only for inserting new knots. To move a knot or a control point, point it with the cursor, then press the *left* mouse button, then move the mouse. To remove a knot, point it with the cursor, press the *left* button, and type the key R.

On the left side of the screen there is a menu, i.e. a collection of widgets, to issue other commands. To halt the program, click (with the left mouse button) the button labelled Quit, or type the key Q.

Typing M causes the window to take the maximal size, and typing m minimizes it. Other commands are easy to guess after reading the labels of the widgets.

[1] The program name in Polish means "bend it" and also "nettle".

16.2 The `pognij` program

The program `pognij`[2] (the directory `demo/pognij`) displays and makes it possible to model a three-dimensional B-spline curve. The main difference between this program and the previous one is the presence of four windows showing the curve, instead of one.

Three of those windows show the images of the curve in orthographic projections onto the planes xy, yz, zx of the coordinate system. It is possible to change the control points via these three windows (using the left mouse button). The fourth window shows the curve in a perspective projection. The left mouse button pressed in this window makes it possible to rotate the viewer position around the curve, and the right button may be used for zooming (one should press it and then move the mouse, forward or back).

Typing R while the cursor is in the perspective view window resets the initial viewer position.

The four windows may be resized. To do it, place the cursor in the narrow area between the windows (this causes changing the cursor shape), press the left button and move the mouse. Typing R while the cursor is altered resets the default equal dimensions of the four windows.

Other details of using this program are the same as for the `pokrzyw` program.

16.3 The `pomnij` program

.....

16.4 The `polep` program

The program `polep` demonstrates the effects of using a simple procedure of filling a polygonal hole in a spline surface made of bicubic patches. This procedure is described in Section 17.1. A detailed description (in Polish) of the algorithm is in my book *Podstawy modelowania krzywych i powierzchni*. Four windows in which three orthographic projections and one perspective projection of the surface are displayed, are used in exactly the same way as the similar four windows of the programs `pognij` and `pozwalaj`.

The control points of the surface may be modified with the mouse (via the orthographic projection windows). The program has a built-in data generator of "ready" data, controlled by three slidebars in the upper part of the menu.

[2] The name of this program in Polish means "bend it" or "get rotten a little".

16.5 The `policz` program

.....

16.6 The `pozwalaj` program

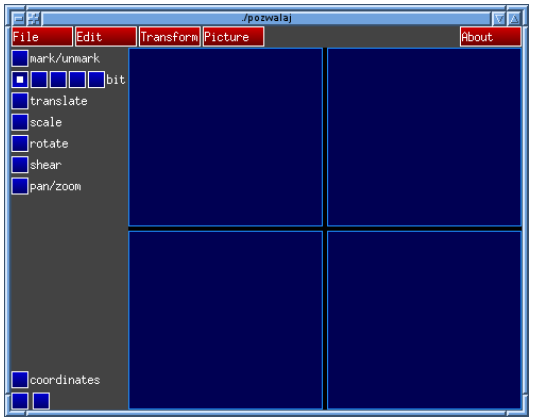
.....

The executable file, `pozwalaj`, by default is located in the directory `demo/bin/` where there is also the file `pozwalaj_proc`, containing the shape optimization procedures; as the computations may take a long time, they are performed independently of the interaction provided by the main executable file. The program `pozwalaj_proc` is supposed to be executed only when invoked by the program `pozwalaj` (and run from the command line it will immediately terminate).

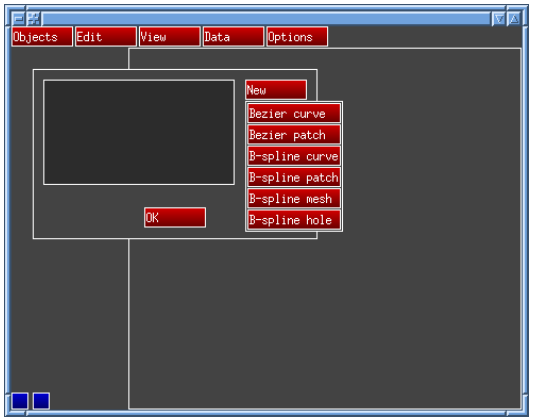
16.6.1 A session log

This section contains screen dumps and comments written during a session with the program `pozwalaj`. During this session a blending surface has been designed, using the interactive tools of the program and one of the built in shape optimization procedures for such surfaces.

After invoking, the program displays two windows (some XWindow managers may place them initially in such a way that one window obscures the other one). The first window displays the geometric data (curves, surfaces and their control polygons and meshes). If 3D objects are displayed, the geometry window is divided into four areas. Three of them show orthogonal projections of the geometric objects onto the xz, yz and xy planes, and one (lower right) shows a perspective projection. A user may change the projection centre by moving the cursor into this area, pressing the left button and moving the cursor.



One of the geometric objects (curves or surfaces, none are present at the beginning of program execution) is distinguished as current. The second window allows the user to make actions specific for the current object, via menus specific for that object. After clicking the button labelled Objects and then New in the popup menu, the window looks like this:

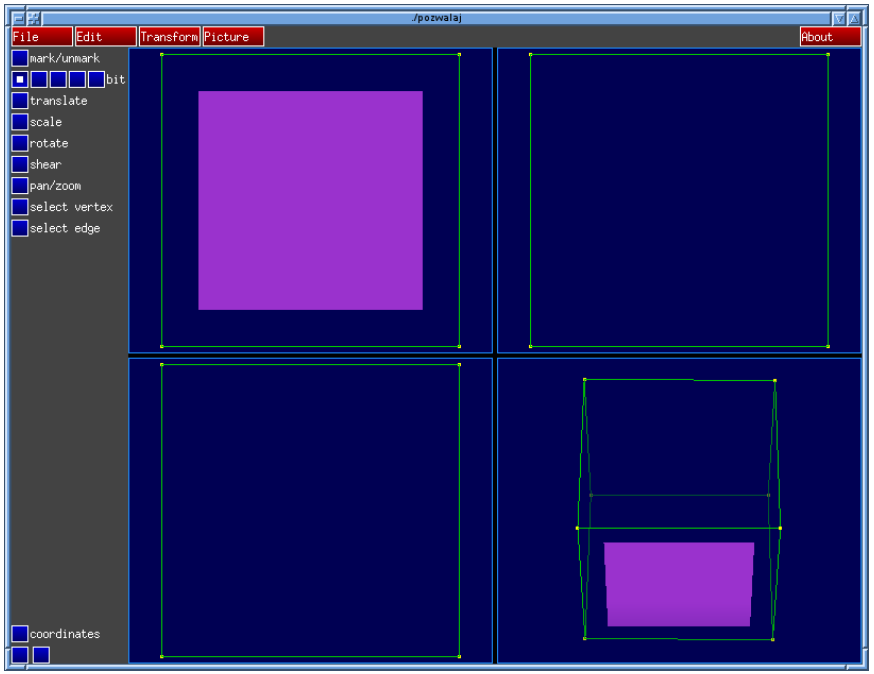


Then, after clicking the button B-spline mesh and then Add in yet another popup, the program creates a new object, which is a spline surface represented by a mesh. Initially this mesh has one facet with four vertices and edges.

After clicking the buttons Data and cube, we choose a mesh, whose facets form the boundary of a cube; the length of its edges is 2. Then clicking the Edit button returns to the menu making it possible to edit the mesh topology. Resizing the window (making it slightly higher) causes all widgets of this menu to fit in.

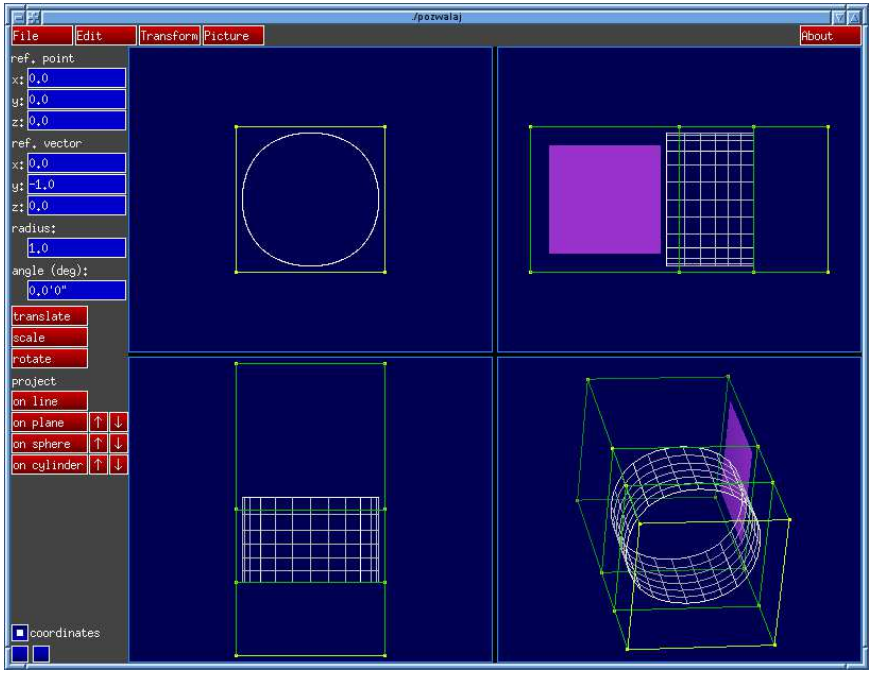
With the cursor back in the first window, on the object images, typing F makes the program find a bounding cube of the object and fit it in the visible area.

Now we edit the mesh. In the second window, click twice (using the left mouse button) the green widget labelled facet (to decrease the number, use the right button, also the mouse wheel works here). This will distinguish the facet number 1, which will be displayed as follows:

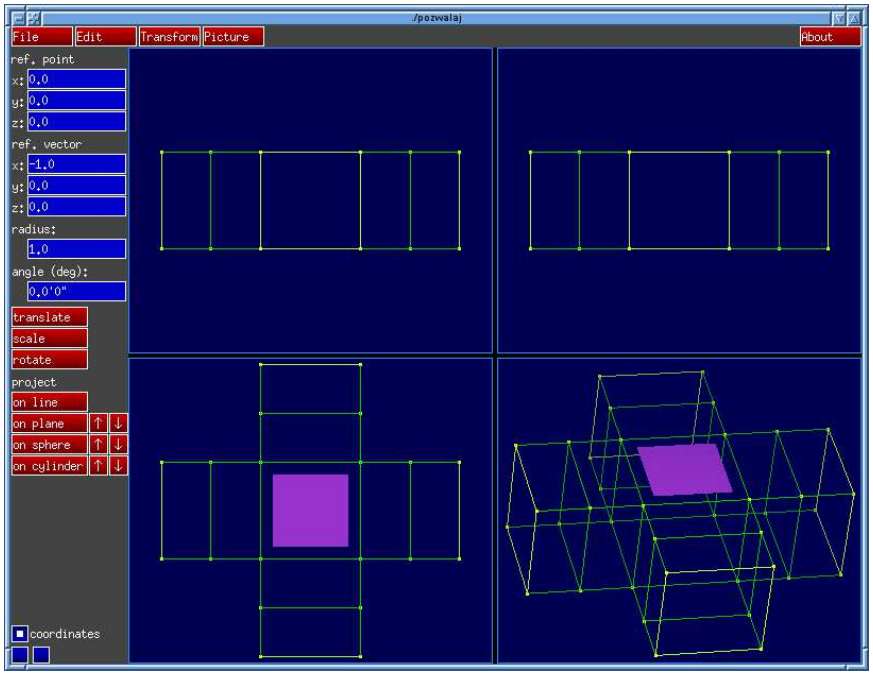


In the second window, click the double edges button. This executes the Eulerian operation, which produces four new facets surrounding the distinguished facet. The new facets are quadrangular, but they are degenerated to line segments.

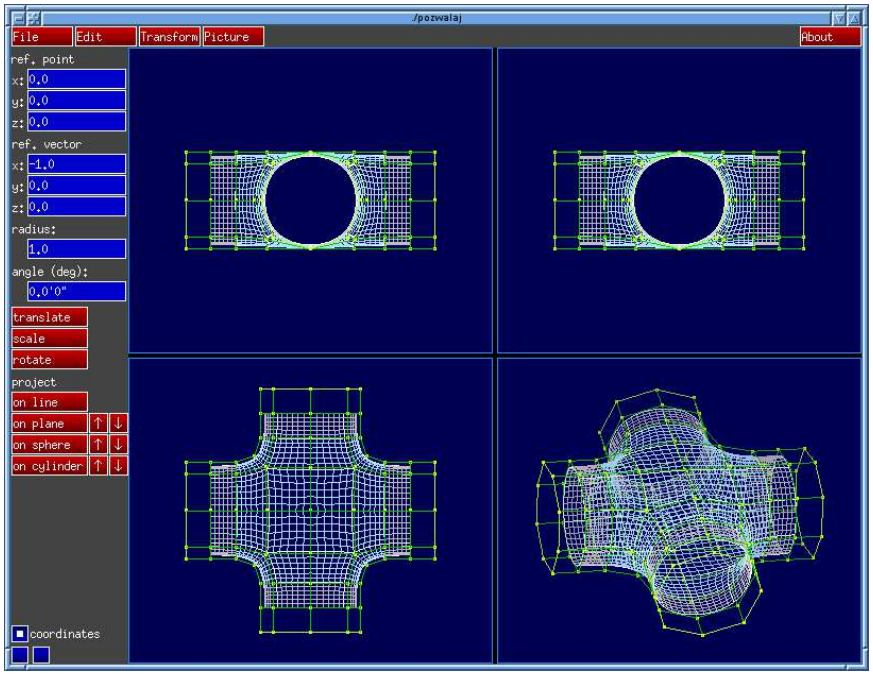
In the first window, click the Transform button. Then, using the text editing widgets (the blue ones, they are activated by clicking on them, and deactivated by clicking aside), type in the coordinates of the reference vector $[0, -1, 0]$, and then click the translate button. Then, in the second window, click double edges and in the first window click translate again. Then click the remove button below the facet number widget. The window now looks as follows:



After removing the facet, another facet became number 1. We click the button double edges, then in the first window we enter the reference vector $[1, 0, 0]$ and click translate, we double edges and translate once more and again we remove the facet. After removing it, we double the edges of the new facet number one, translate its vertices by the vector $[0, 1, 0]$, again we double and translate and remove the facet. For the fourth facet, which became number 1, twice we double the edges and translate the vertices by the vector $[-1, 0, 0]$, then we remove the facet. The result is shown on the next picture:



The mesh has now four closed boundaries, each formed by four edges adjacent to the removed facets. Now, in the second window, we click the refine button. It invokes the procedure implementing the mesh refinement operator, the composition of mesh doubling followed by three averaging operations (actually, the number of averagings is the surface degree, 3 by default). Here is what we obtain:

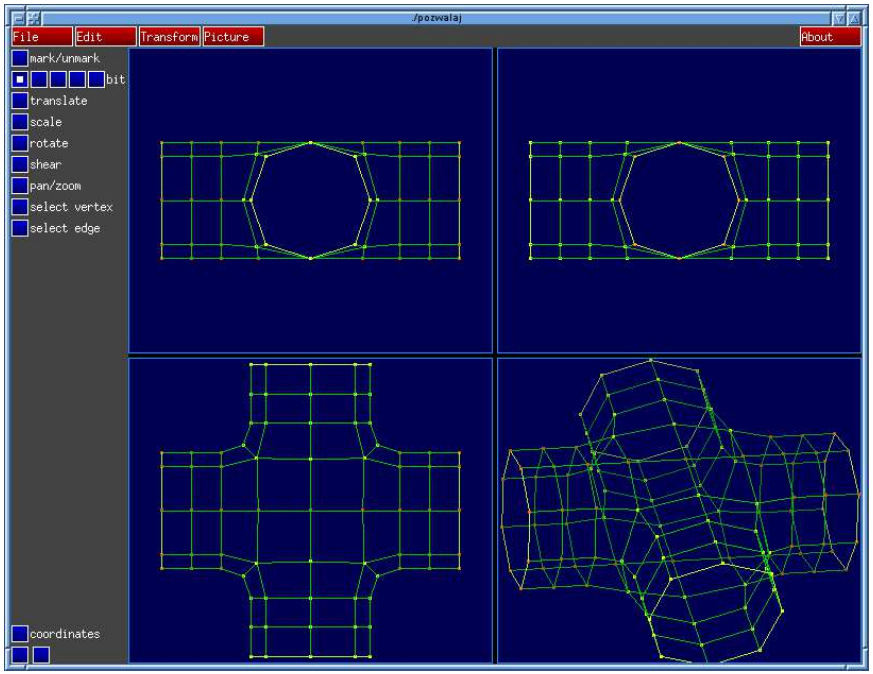


The light grey lines are constant parameter curves of the bicubic patches, corresponding to the regular elements of the surface domain. The binonic patches, represented by special elements of the mesh, are drawn in light blue. For convenience, we may click the View button in the second window and turn off (by clicking) the switches which control displaying the surface (i.e. the bicubic patches) and the hole filling (binonic patches), thus leaving only the mesh vertices and edges on the picture.

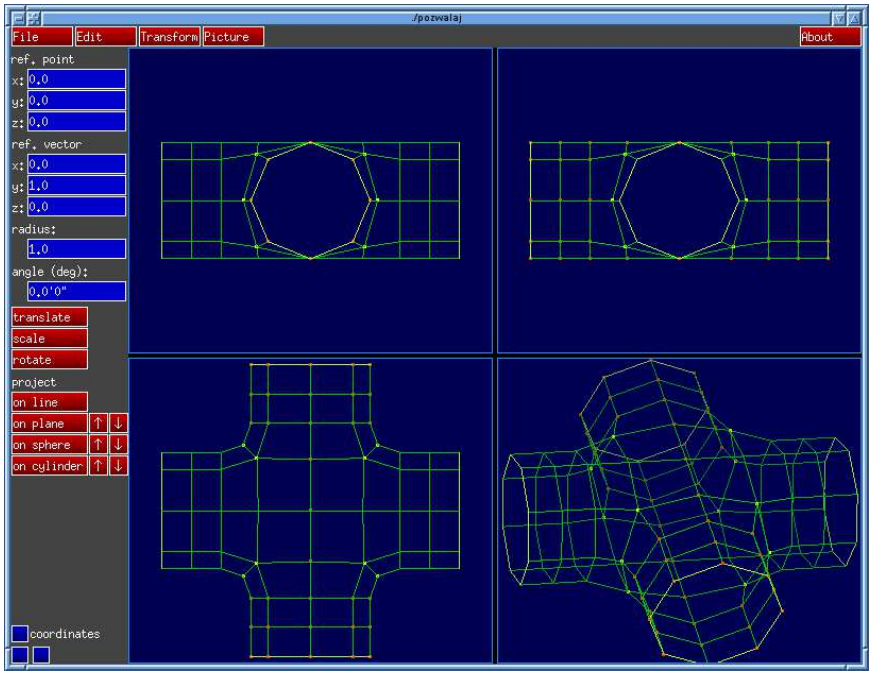
We are going to obtain a blending surface, which is a junction of two crossing cylindric tubes. Of course, bicubic splines cannot represent cylinders of revolution exactly, but if the mesh is dense enough and the vertices are located on a cylinder of revolution, the spline surface may approximate a cylinder with an arbitrarily small error. Therefore in the next step the mesh vertices will be projected onto cylinders. Here is the method: click the Edit button in the top menu of the first window. Then click the mark/unmark switch to turn it on. Now move the cursor to one of the object image windows. Vertices may be marked individually by clicking with

the left mouse button and unmarked by clicking with the right button. Also it is possible to press the button, move the cursor and release the button in order to mark or unmark all vertices in the rectangular area indicated by this mouse movement.

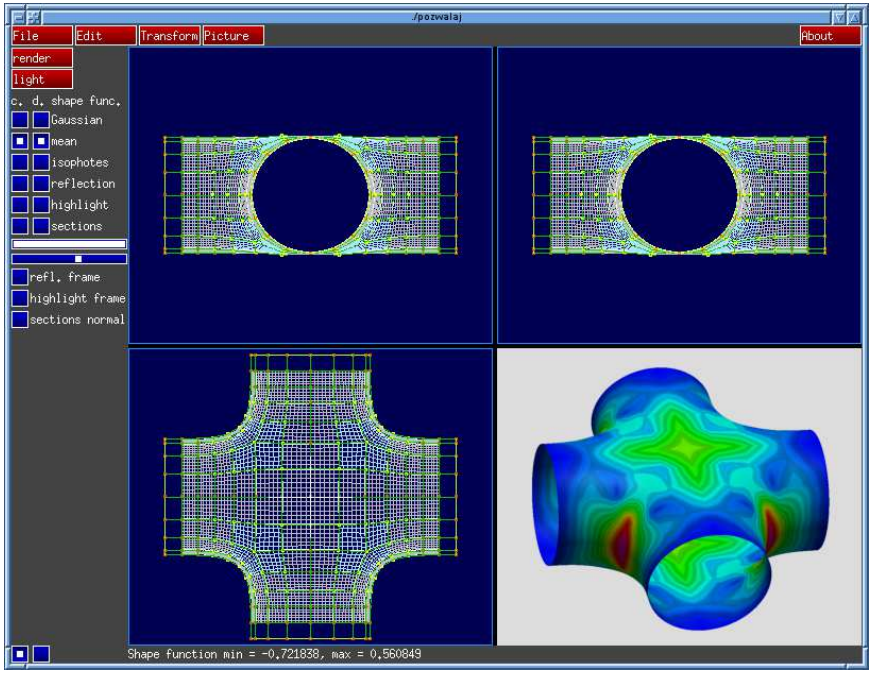
The marking of each vertex consists of five bits. They may be selected for manipulating by five switches just below the mark/unmark switch. We process two groups of vertices, so we need two bits. At first we mark the vertices shown below (the marked vertices are red):



Then we click the Transform button. There we have the coordinates of the reference point $[0,0,0]$ and reference vector $[-1,0,0]$, which determine the axis of the cylinder, and radius 1. Clicking the project on cylinder button makes the program project all the marked vertices on this cylinder. Then we click Edit again, and choose the second bit to mark/unmark (and we turn off switching the first bit). We mark the second set of vertices, then we click Transform, we enter the reference vector (direction of the cylinder axis) $[0,1,0]$ and again we project the vertices on the second cylinder. The result is as on the picture:

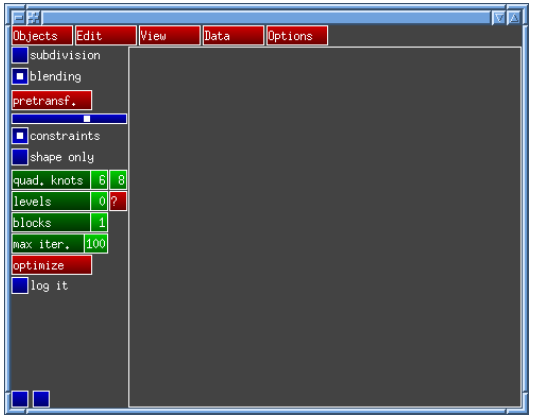


Now we click the Edit button in the second window and then refine. The refinement clears any vertex marking, therefore we go to the first window and in the similar way we mark the vertices and then we project them onto two cylinders, but this time we choose the cylinder radius 0.9267795297. After projecting the vertices we may inspect the surface. To do this, we click the View button in the second window, where we turn on displaying the surface and patches filling the holes in it. Then in the first window we click the button labelled Picture, then shape f., and we may choose the shape function to visualise. Clicking render starts the rendering process (which is ray tracing). An image of mean curvature obtained in this way is as follows:

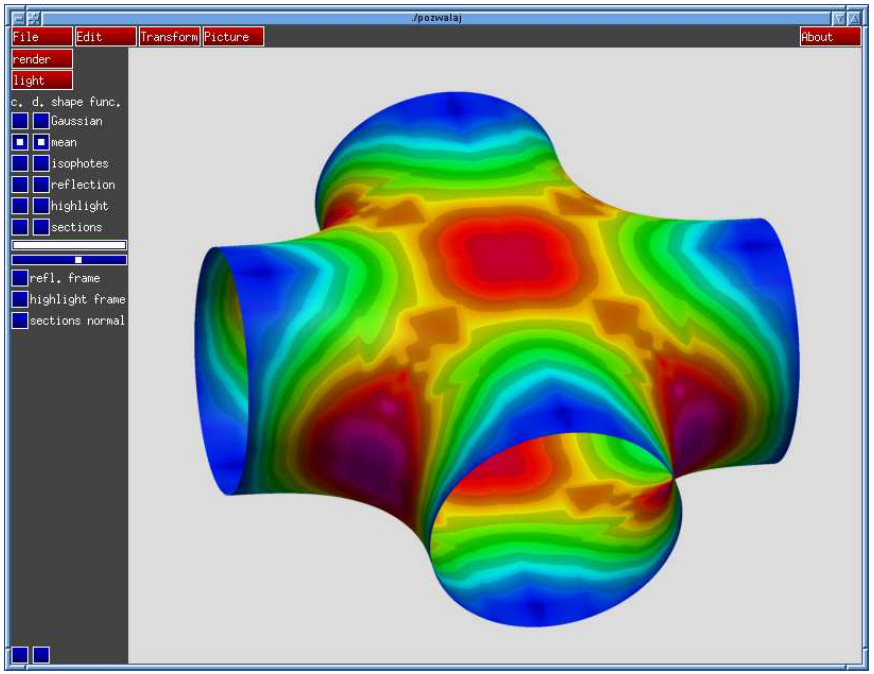


Time to optimise the surface shape. To do this, we need both marked bits to be selected in the editing menu, as the constraints, which we want, are imposed by fixing all vertices having one of the currently selected bits set. The boundary vertices (which we also marked in order to project them on the cylinders) are always fixed for the optimization procedure.

In the second window we click Options and then we turn on the blending switch. Then we turn on the switch labelled constraints. Here is, how the second window should look:



Now we click the optimize button. Some data are written out in the terminal, from which the program has been invoked. Intermediate results (after subsequent iterations of the optimization procedure) are displayed in the first window, and the user may still interact, e.g. in order to look at these results from different sides (this is useful during the optimization of surfaces represented by meshes with large numbers of vertices). The computations on a PC with 3.0GHz Intel Core 2 processor took less than 20 seconds, after which we may render the surface again. To obtain a bigger picture, before doing that we may move the cursor to the perspective image area and type S. The result is the following:



17. Obsolete projects

Once upon a time I wrote a very simple procedure of filling polygonal holes in a bicubic spline surface with biquintic patches so as to obtain the tangent plane (G^1) continuity. This procedure is described in detail in my book *Podstawy modelowania krzywych i powierzchni*, and it may have some educational value, which is why I left it, even if the procedures of filling polygonal holes in the `libeghole` library produce much better results.

17.1 Filling polygonal holes

The demonstration program `polep` uses the procedure of filling a polygonal hole in a generalized bicubic B-spline surface, with Bézier patches of degree $(5,5)$, joined with each other and with the patches around the hole with tangent plane (G^1) continuity. A detailed description of this procedure and the underlying theory is in my book *Podstawy modelowania krzywych i powierzchni* (in Polish). The construction carried out by this procedure is much simpler and less general than the constructions performed by the procedures collected in the libraries `libg1hole` and `libg2hole`; it was developed much earlier and the experience gathered then helped in developing the constructions implemented in these libraries. Below is a description of the representation of data for this procedure and its parameters.

The single precision source code is in the file `g1holef.c`, and its header file is `g1holef.h`. The corresponding double precision source files are `g1holed.c` and `g1holed.h` respectively.

```
boolean FillG1Holef ( int hole_k, point3f*(*GetBezp)(int i, int j),
                      float beta1, float beta2,
                      point3f *hpcp );
```

The procedure `FillG1Holef` constructs a surface filling a k-sided hole in a surface. The parameters are as follows:

The parameter `hole_k` specifies the number k of sides (and corners) of the hole. Its value has to be 3, 5, 6, 7 or 8.

The parameter `GetBezp` is a pointer of a procedure called by `FillG1Holef` in order to get the data, i.e. the control points of bicubic Bézier patches around the hole. The procedure has to return the pointer to an array with the control points of the appropriate patch.

The patches around the hole are numbered with pairs of numbers (i,j), as shown in Figure 17.1; the variable `i` (the parameter `i`) has the value from 0 to $k-1$, the variable `j` (the parameter `j`) is either 1 or 2. The figure shows also the order in

which the control points are to be stored in the array. For each patch it is necessary to supply only 8 points, whose numbers are shown.

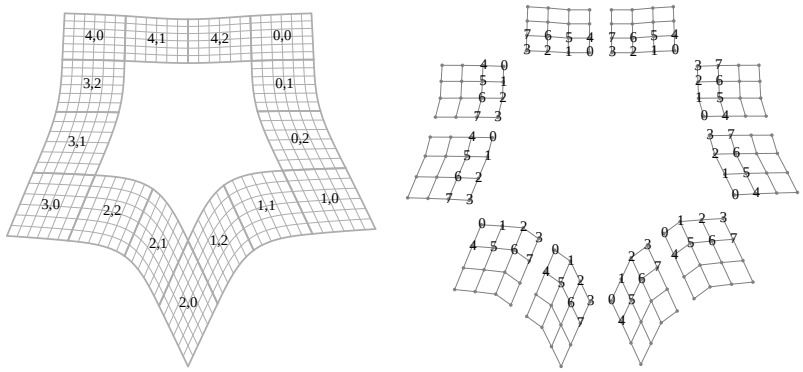


Figure 17.1. The numbering scheme of the patches around the hole and the order of their control points in the arrays

The patches surrounding the hole have to satisfy the compatibility conditions given below, which concern their corners, partial and mixed partial derivatives. The m-th control point of the patch (i,j) is denoted by $\mathbf{p}_m^{(i,j)}$. In addition, $l = i+1 \bmod k$.

- Corner compatibility conditions:

$$\mathbf{p}_0^{(i,1)} = \mathbf{p}_3^{(i,2)} \quad \text{oraz} \quad \mathbf{p}_0^{(i,2)} = \mathbf{p}_3^{(l,1)}.$$

- Partial derivatives compatibility conditions:

$$\begin{aligned}
\mathbf{p}_0^{(i,1)} - \mathbf{p}_1^{(i,1)} &= \mathbf{p}_2^{(i,2)} - \mathbf{p}_3^{(i,2)},\\
\mathbf{p}_0^{(i,2)} - \mathbf{p}_1^{(i,2)} &= \mathbf{p}_7^{(l,1)} - \mathbf{p}_3^{(l,1)},\\
\mathbf{p}_0^{(i,2)} - \mathbf{p}_4^{(i,2)} &= \mathbf{p}_2^{(l,1)} - \mathbf{p}_3^{(l,1)}.
\end{aligned}$$

- Mixed partial derivatives compatibility conditions:

$$\begin{aligned}
\mathbf{p}_4^{(i,1)} - \mathbf{p}_5^{(i,1)} &= \mathbf{p}_6^{(i,2)} - \mathbf{p}_7^{(i,2)},\\
\mathbf{p}_0^{(i,2)} - \mathbf{p}_1^{(i,2)} - \mathbf{p}_4^{(i,2)} + \mathbf{p}_5^{(i,2)} &= \mathbf{p}_6^{(l,1)} - \mathbf{p}_7^{(l,1)} - \mathbf{p}_2^{(l,1)} + \mathbf{p}_3^{(l,1)}.
\end{aligned}$$

The parameters `beta1` and `beta2` are factors, by which some vectors constructed by the procedure `FillG1Holef` are multiplied. By default, their value should be 1, but they may be modified to improve the filleting surface shape, if necessary (i.e. if the effect of using 1 is unsatisfactory).

The parameter hpcp points to the array, in which the procedure is supposed to store the constructed control points of the Bézier patches filling the hole.

The procedure returns true if the construction has been successful, or false otherwise.

```
extern void (*G1OutCentralPointf)( point3f *p );
extern void (*G1OutAuxCurvesf)( int ncurves, int degree,
                                const point3f *accp, float t );
extern void (*G1OutStarCurvesf)( int ncurves, int degree,
                                 const point3f *sccp );
extern void (*G1OutAuxPatchesf)( int npatches, int degu, int degv,
                                 const point3f *apcp );
```

The variables above are "hooks" for procedures, which may output the partial results of the construction. Their initial value is NULL. If the address of the appropriate procedure is assigned to any of those variables before calling FillG1Hole, then this procedure will be called and it may output the data given by the parameters or draw the appropriate picture.

The procedure pointed by the variable G1OutCentralPointf obtains as the parameter the pointer to the "central" point of the filleting surface (i.e. the common point of the patches filling the hole). This procedure is allowed to modify this point, i.e. assign it new coordinates, as such an interference with the construction is possible and sometimes desirable.

The other procedures must not modify the data they get. The procedure pointed by the variable G1OutAuxCurvesf is called with the parameters, which describe the so called auxiliary curves — Bézier curves of degree degree (here it is always 3). Each curve is given in a separate call of this procedure.

The procedure pointed by the variable G1OutStarCurvesf is called with the parameters, which describe the common boundary curves of the patches filling the hole (the construction of those curves is one of the first steps of the algorithm). The procedure parameters describe the Bézier representation of degree 3 of the curves. Each call is made to output one curve.

The procedure pointed by the variable G1OutAuxPatchesf is called with the parameters, which describe the so called auxiliary patches, which determine the tangent planes at all points of the common curves of the patches filling the hole. The auxiliary patches are of degree $(3, 1)$ (the parameters ndegu and ndegv are equal to 3 and 1 respectively). The parameters apcp is a pointer of an array with the control points of one auxiliary patch.

Caution: The current version of the procedure does not contain any code to handle exceptional situations, which might cause the safe return in case of failure. Such a code has to be added and tested if the procedure is to be built into a "production" software, to be used in industrial applications. Moreover, most of the library procedures are not prepared to deal with exceptional situations (exit is called in

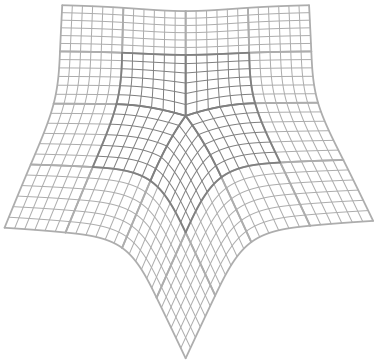


Figure 17.2. A surface with a hole filled by the procedure FillG1Holef

case of error), and therefore the package may be used mainly for experiments (and this is the cause, which made me write the package).