

# BSTools

## biblioteki procedur

Przemysław Kiciak

Wersja 0.29, 20 kwietnia 2012,  
T<sub>E</sub>X-owane 9 grudnia 2019.

**Uwaga:** ta dokumentacja jest niekompletna i wymaga gruntownej rewizji.

Rozpowszechnianie oprogramowanie, którego dotyczy ta dokumentacja, odbywa się na zasadach licencji GNU opracowanych przez Free Software Foundation. Procedury, których kody źródłowe znajdują się w podkatalogach `../src` i `../include` są rozpowszechniane na licencji *GNU Lesser General Public License*, której pełny tekst znajduje się w pliku `COPYING.LIB`, zaś programy demonstracyjne i testowe oraz generujące obrazki do tej dokumentacji (w katalogach `../demo`, `../test` i `../pict`) na licencji *GNU General Public License*, której tekst jest podany w pliku `COPYING`.

Copyright © by Przemysław Kiciak, 2005–2012.

# Spis treści

<b>1</b>	<b>Przegląd</b>	<b>1</b>
1.1	Wstęp . . . . .	1
1.2	Krótki opis bibliotek . . . . .	2
1.3	Kompilacja pakietu . . . . .	3
1.4	Pliki nagłówkowe . . . . .	3
1.5	Kolejność linkowania . . . . .	4
1.6	Zasady modyfikowania procedur . . . . .	5
<b>2</b>	<b>Biblioteka libpkvaria</b>	<b>1</b>
2.1	Różne drobiazgi . . . . .	1
2.2	Boksy . . . . .	2
2.3	Obsługa pamięci pomocniczej . . . . .	2
2.4	Kwadratowa miara kąta . . . . .	4
2.5	Zamiana danych . . . . .	4
2.6	Sortowanie . . . . .	5
2.6.1	Algorytm CountSort . . . . .	5
2.6.2	Algorytm QuickSort . . . . .	6
2.6.3	Kopiec z kolejką priorytetową . . . . .	7
2.7	Obsługa tablic wielowymiarowych . . . . .	8
2.8	Rasteryzacja odcinków . . . . .	13
2.9	Obsługa sytuacji wyjątkowych . . . . .	15
2.10	Opakowania procedur malloc i free . . . . .	17
2.11	Odpluskwanie . . . . .	19
<b>3</b>	<b>Biblioteka libpknum</b>	<b>1</b>
3.1	Działania na wektorach i macierzach pełnych . . . . .	1
3.1.1	Działania elementarne . . . . .	1
3.1.2	Rozwiązywanie układów równań liniowych . . . . .	5
3.1.3	Rozkład QR i liniowe zadania najmniejszych kwadratów . . . . .	7
3.2	Obsługa macierzy wstęgowych . . . . .	11

3.2.1	Reprezentacja i podstawowe procedury . . . . .	11
3.2.2	Rozwiązanie liniowych zadań najmniejszych kwadratów . .	15
3.2.3	Rozwiązanie zadań regularnych z więzami . . . . .	16
3.2.4	Rozwiązanie dualnych zadań najmniejszych kwadratów . .	18
3.2.5	Odpluskwianie . . . . .	20
3.3	Obsługa „spakowanych” macierzy symetrycznych i trójkątnych . . .	21
3.4	Obsługa symetrycznych i trójkątnych macierzy o nieregularnej wstędze . . . . .	26
3.5	Obsługa blokowych macierzy symetrycznych . . . . .	31
3.5.1	Macierze o strukturze blokowej pierwszego rodzaju . . . . .	31
3.5.2	Macierze o strukturze blokowej drugiego rodzaju . . . . .	33
3.5.3	Macierze o strukturze blokowej trzeciego rodzaju . . . . .	37
3.6	Nieregularne macierze rzadkie . . . . .	39
3.7	Rozwiązanie równań nieliniowych . . . . .	45
3.8	Optymalizacja . . . . .	46
3.9	Obliczanie pochodnych funkcji złożonej . . . . .	47
3.9.1	Obliczanie macierzy przekształceń pochodnych . . . . .	48
3.9.2	Obliczanie pochodnych funkcji złożonej . . . . .	49
3.9.3	Obliczanie pochodnych złożenia z funkcją odwrotną . . . . .	50
3.9.4	Obliczanie pochodnych funkcji odwrotnej . . . . .	52
3.10	Kwadratury . . . . .	53
<b>4</b>	<b>Biblioteka libpkgeom</b>	<b>1</b>
4.1	Działania na punktach i wektorach . . . . .	1
4.2	Boksy . . . . .	9
4.3	Znajdowanie otoczki wypukłej . . . . .	9
<b>5</b>	<b>Biblioteka libcamera</b>	<b>1</b>
5.1	Kamera . . . . .	1
5.1.1	Opis kamery i algorytm rzutowania . . . . .	1
5.1.2	Procedury obsługi kamery . . . . .	4
5.2	Para kamer do obrazów stereoskopowych . . . . .	10
<b>6</b>	<b>Biblioteka libpsout</b>	<b>1</b>
6.1	Procedury podstawowe . . . . .	1
6.2	Biblioteka dodatkowa . . . . .	7
<b>7</b>	<b>Biblioteka libmultibs</b>	<b>1</b>
7.1	Podstawowe definicje i sposoby reprezentowania krzywych i płatów .	1
7.1.1	Krzywe Béziera . . . . .	1
7.1.2	Prostokątne płaty Béziera . . . . .	2
7.1.3	Krzywe B-sklejane . . . . .	3
7.1.4	Płaty B-sklejane . . . . .	6

7.1.5	Krzywe i płaty NURBS . . . . .	7
7.1.6	Płaty Coonsa . . . . .	7
7.1.7	Nazwy procedur i parametrów . . . . .	9
7.2	Operacje na ciągach węzłów . . . . .	12
7.2.1	Przeszukiwanie ciągu węzłów . . . . .	12
7.2.2	Tworzenie ciągów węzłów . . . . .	13
7.2.3	Reparametryzacja krzywych i płatów . . . . .	14
7.2.4	Modyfikowanie węzłów . . . . .	15
7.2.5	Sprawdzanie poprawności . . . . .	15
7.3	Obliczanie wartości funkcji B-sklejanych . . . . .	16
7.4	Wyznaczanie punktów krzywych i płatów . . . . .	17
7.4.1	Algorytm de Boora . . . . .	17
7.4.2	Schemat Hornera dla krzywych i płatów Béziera . . . . .	24
7.4.3	Obliczanie krzywizn i układu Freneta krzywej . . . . .	32
7.4.4	Obliczanie wektora normalnego płata . . . . .	33
7.4.5	Obliczanie form podstawowych i krzywizn płatów . . . . .	33
7.5	Tablicowanie krzywych . . . . .	36
7.6	Znajdowanie reprezentacji pochodnych . . . . .	38
7.7	Wstawianie i usuwanie węzłów . . . . .	40
7.7.1	Algorytm Boehma . . . . .	40
7.7.2	Usuwanie węzłów . . . . .	42
7.7.3	Algorytm Oslo . . . . .	45
7.7.4	Maksymalne wstawianie węzłów . . . . .	48
7.7.5	Konwersja krzywych i płatów do postaci kawałkami Béziera . . . . .	50
7.8	Algorytm Lane'a-Riesenfelda . . . . .	53
7.9	Podział krzywych i płatów Béziera na części . . . . .	55
7.10	Podwyższanie stopnia . . . . .	58
7.10.1	Podwyższanie stopnia krzywych i płatów Béziera . . . . .	58
7.10.2	Podwyższanie stopnia krzywych i płatów B-sklejanych . . . . .	60
7.11	Obniżanie stopnia . . . . .	65
7.12	Działania algebraiczne na funkcjach i krzywych sklejanych . . . . .	69
7.12.1	Dodawanie funkcji i krzywych sklejanych . . . . .	69
7.12.2	Przejście między bazami Bernsteina i skalowanymi . . . . .	74
7.12.3	Mnożenie funkcji i krzywych sklejanych . . . . .	75
7.12.4	Wyznaczanie płata opisującego wektory normalne . . . . .	78
7.13	Zmiana węzłów krzywych B-sklejanych na końcach . . . . .	80
7.14	Konstrukcje krzywych interpolacyjnych . . . . .	82
7.14.1	Konstrukcja krzywych interpolacyjnych trzeciego stopnia . . . . .	82
7.14.2	Konstrukcja krzywych interpolacyjnych Hermite'a . . . . .	85
7.15	Konstrukcja krzywych aproksymacyjnych . . . . .	87
7.16	Obcinanie krzywych Béziera . . . . .	90
7.17	Badanie kształtu łamanych . . . . .	92

7.18	Rasteryzacja krzywych . . . . .	93
7.19	Przetwarzanie płatów Coonsa . . . . .	95
7.19.1	Płaty wielomianowe . . . . .	95
7.19.2	Płaty sklejane . . . . .	103
7.20	Produkt sferyczny . . . . .	112
7.21	Rysowanie płatów obciętych . . . . .	113
7.21.1	Reprezentacja dziedziny . . . . .	113
7.21.2	Kompilacja brzegu dziedziny . . . . .	116
7.21.3	Wykonywanie obrazków kreskowych . . . . .	117
<b>8</b>	<b>Biblioteka libraybez</b>	<b>1</b>
8.1	Deklaracje i procedury wspólne . . . . .	1
8.2	Drzewa binarne dla wielomianowych płatów Beziera . . . . .	2
8.3	Drzewa binarne dla wymiennych płatów Béziera . . . . .	4
<b>9</b>	<b>Biblioteka libeghole</b>	<b>1</b>
9.1	Przygotowanie danych . . . . .	1
9.2	Minimum teorii . . . . .	3
9.2.1	Bazy używane w konstrukcjach . . . . .	3
9.2.2	Kryteria optymalizacji powierzchni klasy $G^1$ . . . . .	5
9.2.3	Kryteria optymalizacji powierzchni klasy $G^2$ . . . . .	5
9.2.4	Kryteria optymalizacji dla powierzchni klasy $G^1Q^2$ . . . . .	7
9.2.5	Równania więzów . . . . .	7
9.2.6	Tabela procedur konstrukcji powierzchni . . . . .	8
9.3	Sposób użycia procedur . . . . .	10
9.3.1	Konstrukcja podstawowa . . . . .	10
9.3.2	Konstrukcja nieliniowa . . . . .	11
9.3.3	Konstrukcje z przestrzenią rozszerzoną . . . . .	11
9.3.4	Konstrukcje z więzami . . . . .	12
9.4	Procedury podstawowe . . . . .	13
9.5	Wprowadzanie opcji . . . . .	18
9.6	Nakładanie więzów . . . . .	20
9.6.1	Wypełnianie otworów płatami B-sklejanymi . . . . .	24
9.7	Procedury konstrukcji nieliniowych . . . . .	25
9.8	Procedury wizualizacji . . . . .	28
<b>10</b>	<b>Biblioteka libbsmesh</b>	<b>1</b>
10.1	Reprezentacja siatki . . . . .	1
10.2	Procedury zagęszczania siatki . . . . .	3
10.3	Operacje Eulerowskie i nie-Eulerowskie . . . . .	5
10.4	Wyszukiwanie podsiatek regularnych i specjalnych . . . . .	7
10.5	Inne procedury . . . . .	8

---

<b>11 Biblioteka libg1blending</b>	<b>1</b>
<b>12 Biblioteka libg2blending</b>	<b>1</b>
12.1 Tensorowe B-sklejane płyty trójharmiczne . . . . .	1
12.2 Optymalizacja płytów tensorowych z kryterium jakości kształtu . . .	1
12.3 Optymalizacja powierzchni reprezentowanych przez siatki nieregularne	4
<b>13 Biblioteka libbsfile</b>	<b>1</b>
<b>14 Biblioteka libxgedit</b>	<b>1</b>
<b>15 Programy demonstracyjne</b>	<b>1</b>
15.1 Program pokrzyw . . . . .	1
15.2 Program pognij . . . . .	2
15.3 Program pomnij . . . . .	2
15.4 Program polep . . . . .	2
<b>16 Projekty</b>	<b>1</b>
16.1 Wypełnianie wielokątnych otworów . . . . .	1





# 1. Przegląd

## 1.1 Wstęp

Pakiet procedur BSTools napisałem w celu przeprowadzania eksperymentów stanowiących część mojej pracy naukowej, a także dla rozrywki. Główna część pakietu składa się z procedur przetwarzania krzywych i powierzchni Béziera i B-sklejanych, stąd nazwa. Własności krzywych i powierzchni oraz teoretyczne podstawy działania procedur ich przetwarzania są opisane w mojej książce

*Podstawy modelowania krzywych i powierzchni  
zastosowania w grafice komputerowej*

wydanej przez Wydawnictwa Naukowo-Techniczne<sup>1</sup>. Pakiet BSTools (wersja 0.12) jest dodatkiem do drugiego wydania tej książki (z roku 2005). *W odróżnieniu* od książki (której kopiowanie, nawet fragmentów, *musi* być poprzedzone uzyskaniem zgody WNT), pakiet ten *wolno* kopiować i swobodnie rozpowszechniać, a także modyfikować i używać w dowolnych programach, na zasadach Mniejszej Licencji GNU, której pełny tekst znajduje się w pliku COPYING.LIB.

W mojej drugiej książce,

*Konstrukcje powierzchni gładko wypełniających  
wielokątne otwory*

wydanej przez Oficynę Wydawniczą Politechniki Warszawskiej (prace naukowe, Elektronika, z. 159, 2007) są opisane konstrukcje powierzchni klasy  $G^1$  i  $G^2$ , których implementacje są w bibliotece libeghole w tym pakiecie (w wersji 0.18 dołączonej do książki są dwie osobne biblioteki, libg1hole i libg2hole, które później zostały połączone i znacznie rozbudowane).

Procedury z pakietu BSTools mogą być użyte w dowolnym (oby godziwym) celu, na przykład do napisania systemu modelowania lub dowolnego programu graficznego. W tym celu trzeba je „uodpornić”, tj. dopracować pełny system wykrywania i obsługi błędów, oraz przeprowadzić stosowne testy. Jak wiadomo, osobą o najmniejszych kompetencjach do testowania dowolnego programu jest jego autor (co go zresztą nie usprawiedliwia, jeśli tego nie robi). Osoby chętne do udziału w tym przedsięwzięciu, a także do rozwijania pakietu i wykorzystywania go w zastosowaniach, będą mile widziane.

---

<sup>1</sup>Oprócz mojej istnieje jeszcze wiele innych książek, w których są przedstawione algorytmy realizowane przez opisane tu procedury; zgodnie z moją wiedzą żadna z książek poświęconych w całości tej tematyce nie została jeszcze (do początku roku 2005) przetłumaczona na język polski.

## 1.2 Krótki opis bibliotek

Pakiet BSTools obecnie składa się z następujących bibliotek:

- `libpkvaria` — Różności, jak to: obsługa pamięci pomocniczej, sortowanie i inne.
- `libpknum` — Procedury numeryczne używane w różnych konstrukcjach krzywych B-sklejanych, ale nadające się do wykorzystania także w dowolnym innym celu.
- `libpkgeom` — Procedury geometryczne.
- `libcamera` — Rzutowanie perspektywiczne i równoległe.
- `libpsout` — Tworzenie plików z rysunkami w języku PostScript<sup>(TM)</sup>.
- `libmultibs` — Obsługa krzywych i powierzchni Béziera i B-sklejanych.
- `libraybez` — Śledzenie promieni (wyznaczanie przecięć promienia z płatem).
- `libeghole` — Wypełnianie wielokątnych otworów w kawałkami bikubicznych powierzchni B-sklejanych z ciągłością  $G^1$ ,  $G^2$  i  $G^1Q^2$ .
- `libbsmesh` — Procedury obsługi siatek reprezentujących powierzchnie.
- `libg1blending` — Procedury optymalizacji kształtu powierzchni B-sklejanych stopnia  $(2,2)$ , klasy  $G^1$ .
- `libg2blending` — Procedury optymalizacji kształtu bikubicznych powierzchni B-sklejanych i reprezentowanych przez siatki, klasy  $G^2$ .
- `libbsfile` — Pisanie i czytanie plików z danymi opisującymi krzywe i powierzchnie.
- `libxgedit` — Obsługa dialogu (poprzez okna i wihajstry) aplikacji w systemie XWindow (na potrzeby programów demonstracyjnych).

Procedury są napisane w czystym języku C, bez żadnych zależności sprzętowych ani systemowych, z jednym wyjątkiem: procedura sortowania w bibliotece `libpkvaria` jest napisana przy założeniu, że procesor umieszcza poszczególne bajty słowa maszynowego w kolejności little-endian. Ewentualne przeniesienie pakietu na komputer wyposażony w procesor taki jak Motorola wymaga przepisania tej procedury (to zostało zrobione, ale jeszcze nie zostało przetestowane).

## 1.3 Kompilacja pakietu

Dostarczone pliki Makefile są przystosowane do działania w systemie Linux. Aby skompilować pakiet, dokumentację i programy demonstracyjne, należy mieć zainstalowane:

- program GNU make,
- kompilator gcc i program ar,
- biblioteki XWindow i OpenGL (dla programów demonstracyjnych),
- system  $\text{\TeX}$  (do skompilowania dokumentacji — potrzebny jest pakiet  $\text{\LaTeX}$  i zestawy fontów Concrete Roman i Euler),
- a ponadto programy Ghostscript i Ghostview, które służą do wygodnego oglądania dokumentacji i obrazków wygenerowanych przez programy testowe.

Aby skompilować cały pakiet, wystarczy w głównym katalogu rozpakowanego pakietu wydać polecenie make. Można to poprzedzić poleceniem make clean, aby wymusić kompilację wszystkich plików źródłowych.

Programy demonstracyjne można uruchomić w systemie XWindow, przy czym nie ma żadnych wymagań co do konkretnego menagera okien, nie są też wymagane żadne specjalistyczne biblioteki (takie jak Motif itd.). Niektóre programy demonstracyjne korzystają z OpenGL-a, potrzebne są biblioteki libGL, libGLU i libGLX.

## 1.4 Pliki nagłówkowe

Pliki nagłówkowe bibliotek znajdują się w katalogu ../include. Każda biblioteka może mieć więcej niż jeden plik nagłówkowy, co umożliwia skrócenie kompilacji programów nie odwołujących się do wszystkich procedur w danej bibliotece.

libpkvaria — plik pkvaria.h.

libpknum — pliki pknumf.h i pknumd.h, zawierające nagłówki procedur odpowiednio w wersjach pojedynczej (IEEE-754 single) i podwójnej (IEEE-754 double) precyzji arytmetyki zmiennopozycyjnej. Użycie pliku pknum.h powoduje włączenie obu tych plików, dzięki czemu można wygodniej kompilować programy wykonujące obliczenia w obu precyzjach.

libpkgeom — pliki pkgeomf.h i pkgeomd.h oraz pkgeom.h, które umożliwiają korzystanie z procedur pojedynczej i podwójnej oraz obu precyzji.

Procedury znajdowania otoczki wypukłej zbioru punktów mają osobny plik convh.h, który zawiera nagłówki wersji dla obu precyzji.

`libcamera` — pliki `cameraf.h`, `camerad.h` i `camera.h` zawierają opisy kamer, tj. obiektów opisujących rzutowanie perspektywiczne i równoległe, w wersji pojedynczej, podwójnej i obu precyzji.

Pliki `stereof.h`, `stereod.h` i `stereo.h` zawierają opisy pary takich kamer, której można użyć do wykonania pary obrazów stereoskopowych.

`libpsout` — plik `psout.h` zawiera nagłówki wszystkich procedur w tej bibliotece.

`libmultibs` — pliki `multibsf.h`, `multibsd.h` i `multibs.h` zawierają opisy procedur w pojedynczej, podwójnej oraz obu precyzjach.

`libraybez` — pliki `raybezf.h` (pojedyncza precyzja), `raybezd.h` (podwójna precyzja) i `raybez.h` (obie wersje).

`libeghole` — pliki `eg1holef.h`, `eg2holef.h` (pojedyncza precyzja), `eg1holed.h` i `eg2holed.h` (podwójna precyzja). Nie ma plików dla obu wersji jednocześnie.

`libbsmesh` — plik `bsmesh.h`

`libg1blending` — pliki `g1blendingf.h`, `g1blendingd.h` (procedury pojedynczej i podwójnej precyzji).

`libg2blending` — pliki `g2blendingf.h`, `g2blendingd.h`, `g2mblendingd.h`. Niektóre procedury są tylko w wersji podwójnej precyzji, ponieważ zakres liczb pojedynczej precyzji jest niewystarczający do przeprowadzenia obliczeń.

`libbsfile` — plik `bsfile.h`, procedury są tylko dla wersji podwójnej precyzji.

`libxgedit` — pliki `xgedit.h` i `xgledit.h`; drugi plik zawiera nagłówki procedur dla aplikacji używającej OpenGL-a. Dodatkowe pliki, `xgergb.h` i `xglergb.h`, których nie należy włączać bezpośrednio zawierają definicje kolorów odpowiadających angielskim nazwom.

Procedury są skompilowane jako programy w C, ale powyższe pliki nagłówkowe zawierają odpowiednie fragmenty powodujące kompilowanie programu w C++ tak, aby dało się go zlinkować z tymi bibliotekami.

## 1.5 Kolejność linkowania

Procedury w poszczególnych bibliotekach wywołują procedury z innych bibliotek, w związku z czym linkowanie programu wymaga podania tych bibliotek we właściwej kolejności (w przeciwnym razie kompilator nie znajdzie potrzebnej procedury w bibliotece, której przeszukiwanie zakończył wcześniej). Jedną z właściwych kolejności jest taka:

---

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

i należy ją zachować pisząc np. własne pliki Makefile. Oczywiście, biblioteki, których dany program nie używa, można pominąć.

## 1.6 Zasady modyfikowania procedur

Licencja GNU nie nakłada żadnych ograniczeń na zmiany, jakich ktoś chciałby dokonać w oprogramowaniu (ale fakt dokonania zmiany musi być zaznaczony w kodzie, tak aby nie było wątpliwości, że to nie autor oryginalnego programu coś sknocił). Dlatego wszelkie zasady mogą być tylko życzeniami autora.

1. Z wyjątkiem tworzenia plików postscriptowych, oraz biblioteki xgedit, wszystkie procedury są niezależne od wszelkich środowisk, w jakich mogłyby być użyte (i tak powinno pozostać).
2. Dokonując zmiany procedury w wersji pojedynczej lub podwójnej precyzji trzeba wprowadzić analogiczną zmianę w tej drugiej wersji (jeśli istnieje; jeśli nie, wskazane jest napisanie jej, z użyciem identycznego algorytmu, choć dla niektórych algorytmów pojedyncza precyzja ma niewystarczający zakres).
3. Po dokonaniu zmiany wskazane jest uaktualnienie dokumentacji i zrealizowanie odpowiedniego programu testowego. Będę wdzięczny za informacje o zmianach (i mogę je włączać do następnych wersji pakietu).



## 2. Biblioteka libpkvaria

Nagłówki procedur z biblioteki libpkvaria są opisane w pliku pkvaria.h.

### 2.1 Różne drobiazgi

```
#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;
```

Jeśli jakaś dana jest boolowska, to lepiej jest to zaznaczyć używając nazwy boolean niż unsigned char dla jej typu i pisać true i false zamiast 0 i 1. Sposób potraktowania tej reguły w bibliotekach pozostawia niestety wiele do życzenia.

Warto jest ustalić pewne konwencje dotyczące miar stosowanych w programie. Wiadomo, że jak coś można zrobić na kilka sposobów, to każdy zrobi to inaczej. W ten sposób w jednym programie pisanym przez zespół występują procedury, które otrzymują parametry — długości w metrach i centymetrach albo calach. Wiadomo, że dla ludzi wygodniejsze są stopnie, a w kodzie programu — radiany. Ja wymyśliłem, że będę przestrzegał właśnie tej konwencji, ale np. w PostScriptcie i w OpenGL-u jest inna.

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

Dwa bardzo użyteczne makra.

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

Procedura pkv\_rpower oblicza  $x^e$ .

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

Procedura pkv\_HexByte wyznacza reprezentację wartości parametru b w postaci szesnastkowej. Cyfry szesnastkowe są umieszczane w tablicy s, która musi mieć długość co najmniej 3.

## 2.2 Boksy

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

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

## 2.3 Obsługa pamięci pomocniczej

Wiele procedur w opisanych tu bibliotekach używa pamięci pomocniczej do przechowywania danych, przy czym bloki tej pamięci są zwalniane w kolejności odwrotnej do kolejności rezerwowania. Dlatego gospodarka tą pamięcią jest zaimplementowana za pomocą stosu, co działa bardzo szybko.

Pamięci pomocniczej obsługiwanej przez procedury opisane niżej mogą używać też inne procedury — jedyne warunki to zarezerwowanie na początku działania programu dostatecznie dużej puli oraz używanie tej pamięci w ściśle „stosowy” sposób.

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

Procedura `pkv_InitScratchMem` rezerwuje (za pomocą procedury `malloc`) blok pamięci pomocniczej o wielkości `size` bajtów i przygotowuje gospodarkę pamięcią w tym bloku. Wartość procedury 0 oznacza brak pamięci, zaś 1 wskazuje sukces.

Procedura ta musi być wywołana przed użyciem wszelkich procedur, które korzystają z puli pamięci pomocniczej (tj. wywołują procedury `pkv_GetScratchMem` lub `pkv_GetScratchMemTop` i `pkv_FreeScratchMem` lub `pkv_SetScratchMemTop`).

Obecnie nie ma powiększania puli pamięci pomocniczej, jeśli podczas wywołania procedury `pkv_GetScratchMem` okaże się, że tej pamięci zabrakło. Nakłada to na programistę obowiązek dobrego oszacowania wielkości tego bloku wystarczającej do przeprowadzenia obliczeń. Pewną pomocą może być użycie w eksperymentach funkcji `pkv_MaxScratchTaken`.

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

Procedura `pkv_DestroyScratchMem` zwalnia (przez wywołanie procedury `free`) blok pamięci pomocniczej zarezerwowany przez `pkv_InitScratchMem`.

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

Procedura `pkv_GetScratchMem` rezerwuje obszar o wielkości `size` bajtów w bloku pamięci pomocniczej i zwraca wskaźnik do tego bloku. Jeśli nie ma tyle pamięci, to wartością procedury jest wskaźnik pusty (`NULL`).



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

Procedura `pkv_FreeScratchMem` zwalnia ostatnie `size` bajtów zarezerwowanych przez wcześniejsze wywołania procedury `pkv_GetScratchMem`.

Kolejno rezerwowane bloki muszą być zwalniane w odwrotnej kolejności, przy czym można po zarezerwowaniu kilku bloków zwolnić je wszystkie naraz, podając parametr `size` o wartości równej sumie długości (w bajtach) tych bloków.

```
#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))
```

Powyższe makra mają na celu wygodne wywoływanie procedur rezerwowania i zwalniania pamięci na potrzeby tablic liczb zmiennopozycyjnych. Dzięki ich zastosowaniu można skrócić i uczynić kod programu.

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

Alternatywą dla zapamiętania liczby zarezerwowanych bajtów (które można zwolnić wywołując `pkv_FreeScratchMem`) jest zapamiętanie wskaźnika końca zajętego obszaru pamięci pomocniczej przez wywołanie funkcji `pkv_GetScratchMemTop`. Następnie można zarezerwować pamięć (za pomocą jednego lub więcej wywołań `pkv_GetScratchMem`), a po zakończeniu obliczeń można przypisać zapamiętaną wartość wskaźnika za pomocą `pkv_SetScratchMemTop`.

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

Wartością funkcji `pkv_ScratchMemAvail` jest aktualna liczba wolnych bajtów w puli. Próba zarezerwowania większego bloku nie uda się (`pkv_GetScratchMem` zwróci wskaźnik `NULL`).

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

Wartością funkcji `pkv_MaxScratchTaken` jest maksymalna liczba zarezerwowanych jednocześnie bajtów od utworzenia puli (za pomocą `pkv_InitScratchMem`) do chwili wywołania tej funkcji.

## 2.4 Kwadratowa miara kąta

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

Funkcja `pkv_SqAngle` oblicza pewną miarę kąta między wektorem  $[x, y]$  a osią  $Ox$ . Miara ta jest obliczana za pomocą paru działań arytmetycznych, a zatem szybciej niż funkcje cyklotometryczne. Funkcja przyjmuje wartość z przedziału  $[0, 4]$ .

Własności tej miary: jeśli dwa wektory tworzą kąt prosty, to różnica wartości funkcji `pkv_SqAngle` jest równa  $\pm 1$ . Podobnie, dla kąta półpełnego to jest  $\pm 2$ . Miara kąta pełnego jest równa 4.

## 2.5 Zamiana danych

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

Procedura `pkv_Exchange` przestawia zawartości obszarów pamięci o długości `size` wskazywanych przez parametry `x` i `y`. Obszary te muszą być rozłączne.

Procedura korzysta z bufora o długości co najwyżej 1KB, rezerwowanego za pomocą procedury `pkv_GetScratchMem`, a zatem aby z niej skorzystać należy utworzyć odpowiednio dużą pulę pamięci pomocniczej (wywołując `pkv_InitScratchMem` na początku programu).

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

Procedury `pkv_Sort2f` i `pkv_Sort2d` wymieniają wartościami zmienne `*a` i `*b`, jeśli pierwsza z nich jest większa od drugiej.

## 2.6 Sortowanie

### 2.6.1 Algorytm CountSort

Procedury opisane niżej służą do sortowania tablic zawierających rekordy z danymi liczbowymi (kluczami) całkowitymi lub zmiennopozycyjnymi. Metoda sortowania jest zależna od liczby rekordów (długości tablicy). Dla małej liczby rekordów jest używany algorytm sortowania przez wstawianie, a dla dużej (co najmniej kilkadziesiąt) — sortowanie licznikowe.

Metoda sortowania jest stabilna, tj. nie zamienia kolejności rekordów, które zawierają klucz o tej samej wartości, z wyjątkiem przestawiania liczby zmiennopozycyjnej  $+0.0$  za  $-0.0$ .

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

Powyżej są wyliczone identyfikatory dopuszczalnych typów kluczy. Typy `short` i `unsigned short` to liczby całkowite szesnastobitowe. Typy `int` i `unsigned int` to liczby całkowite trzydziestodwubitowe. Typy `float` i `double` to liczby zmiennopozycyjne trzydziestodwu- i sześćdziesięcioczerobitowe, zdefiniowane w standardzie IEEE-754. Procedury sortowania są oparte na założeniu, że kolejność bajtów jest *little-endian* (czyli tak jak w procesorach Intel).

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

Zdefiniowane wyżej stałe są możliwymi wartościami procedur sortujących opisanych niżej. Jeśli podczas wykonania nie nastąpił błąd, to procedura zwraca wartość `SORT_OK`. Inne możliwe wartości procedury wskazują na brak pamięci pomocniczej lub na niepoprawne dane.

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

Procedura `pkv_SortKernel` znajduje właściwą kolejność elementów w tablicy `ndata`, tj. permutację, zgodnie z którą należy poprzerastawiać elementy, aby je posortować. Tablica `ndata` składa się z rekordów o długości `item_length` bajtów. Klucz, tj. liczba całkowita lub zmiennopozycyjna (typ jest określony przez parametr `num_type`), względem której należy uporządkować rekordy, znajduje się w każdym rekordzie `num_offset` bajtów od początku rekordu. Liczba `n` rekordów (tj. długość tablicy `ndata`) jest określona przez parametr `num_data`.

Tablica permut zawiera liczby od 0 do  $n - 1$ . Po wyjściu z procedury (jeśli nie wystąpił błąd), tablica permut zawiera te same liczby, które reprezentują właściwą permutację. Początkowa kolejność liczb w tablicy permut jest istotna wtedy, gdy dane mają być posortowane kolejno względem kilku kluczy. Jeśli np. rekordy składają się z dwóch pól liczbowych, np.  $x$  i  $y$ , i mają być posortowane względem wartości pola  $x$ , a w przypadku, gdy pola  $x$  są równe, to względem  $y$ , to należy wpisać do tablicy permut liczby od 0 do  $n - 1$  w dowolnej kolejności, następnie wywołać procedurę `pkv_SortKernel` dwukrotnie: najpierw w celu posortowania tablicy względem wartości pól  $y$ , a następnie względem  $x$ . Potem można wywołać procedurę `pkv_SortPermute` w celu odpowiedniego poprzestawiania rekordów w tablicy.

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

Procedura `pkv_SortPermute` przestawia rekordy w tablicy `ndata` zgodnie z zawartością tablicy `permut`, która musi zawierać liczby całkowite od 0 do  $n - 1$ . Liczba rekordów  $n$  jest wartością parametru `num_data`, długość rekordu w bajtach jest określona przez parametr `item_length`.

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

Procedura `pkv_SortFast` sortuje tablicę `ndata`, która zawiera `num_data` rekordów o długości `item_length`, z których każdy zawiera klucz liczbowy typu określonego przez parametr `num_type`, w odległości `num_offset` bajtów od początku rekordu.

### 2.6.2 Algorytm QuickSort

Opisana niżej procedura sortuje elementy danego ciągu za pomocą porównań. Jest ona implementacją algorytmu QuickSort, która dwie podstawowe operacje — porównywanie i przestawianie — wykonuje za pomocą procedur (w aplikacji) podanych przy użyciu parametrów. Algorytm QuickSort nie jest stabilny, tj. w ciągu nierównowartościowym elementy takie same (ze względu na porządek) mogą po posortowaniu mieć zamienioną kolejność.

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

Parametr `n` określa długość ciągu do posortowania; jego elementy mają indeksy od 0 do  $n - 1$ . Parametr `less` jest wskaźnikiem procedury, której wartość `true` oznacza, że element  $i$ -ty ciągu jest mniejszy niż element  $j$ -ty (gdzie liczby  $i$  i  $j$  są wartościami parametrów tej procedury). Parametr `swap` jest wskaźnikiem procedury, której zadaniem jest przestawienie elementów ciągu o numerach będących wartościami parametrów.

### 2.6.3 Kopiec z kolejką priorytetową

Kolejka priorytetowa zaimplementowana przy użyciu procedur opisanych w tym punkcie jest tablicą wskaźników do dowolnych obiektów; zarówno wstawienie obiektu do kolejki, jak i jego usunięcie, polega na dopisaniu lub usunięciu wskaźnika w tablicy. Priorytety są określone przez aplikację, która dostarcza procedurę `cmp`, z dwoma parametrami wskaźnikowymi. Procedura zwraca wartość `true` wtedy, gdy priorytet obiektu wskazywanego przez pierwszy parametr jest większy.

```
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 Obsługa tablic wielowymiarowych

Procedury przetwarzania krzywych i powierzchni zawarte w bibliotece libmultibs przetwarzają punkty kontrolne pobierając je i wstawiając do jednowymiarowych tablic liczb zmiennopozycyjnych. Taka tablica może być zadeklarowana jako tablica np.  $n$  punktów w przestrzeni trójwymiarowej, ale obszar pamięci komputera zajęty przez tę tablicę zawiera  $3n$  upakowanych obok siebie liczb.

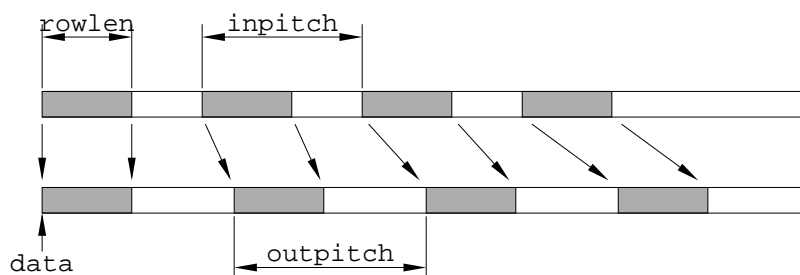
Tablice dwuwymiarowe punktów mają podobną zawartość i zwykle służą do przechowywania prostokątnych siatek kontrolnych płatów. W takiej tablicy znajdują się kolejno współrzędne punktów w pierwszej kolumnie siatki, następnie w drugiej itd. Podstawowym parametrem umożliwiającym procedurom dostęp do odpowiednich miejsc w takiej tablicy jest jej *podziałka* (ang. *pitch*), czyli odległość (której jednostką jest długość reprezentacji jednej liczby) między pierwszą współrzędną pierwszego punktu w dowolnej kolumnie siatki i pierwszą współrzędną pierwszego punktu w kolumnie następnej. Oczywiście, podziałka nie ma znaczenia dla tablic jednowymiarowych.

Z punktu widzenia procedur przetwarzania tablic lepiej jest interpretować je jako tablice dwuwymiarowe (bez wnikania w to, ile jest punktów i jaki jest wymiar przestrzeni, której to są punkty). W tablicy są przechowywane *wiersze* o ustalonej długości (nie większej niż podziałka). Po każdym wierszu (z wyjątkiem być może ostatniego) znajduje się obszar nieużywany, którego długość dodana do długości wiersza jest równa podziałce. Procedury i makra opisane niżej służą do zmieniania podziałki (tj. „rozsuwania” lub „dosuwania” wierszy połączonego ze zmianą długości obszarów nieużywanych), przepisywania danych z jednej tablicy do drugiej (o innej podziałce) oraz „przesuwania” wierszy w tablicy (bez zmiany podziałki).

Dla celów specjalnych może być potrzebne przetwarzanie w taki sposób tablic bajtów (czyli najmniejszych bezpośrednio adresowanych przez procesor komórek pamięci). Dlatego procedury w C są zrealizowane dla tablic elementów typu char. Tablice liczb typu float i double mogą być przetwarzane za pomocą makr, które mnożą długości wierszy i podziałki przez odpowiednie liczby bajtów reprezentacji typu float lub double.

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

Procedura pkv\_Rearrange przestawia dane w tablicy w celu zmieniania podziałki. W tablicy jest *nrows* wierszy. Każdy z nich zawiera *rowlen* bajtów. Parametr *inpitch* określa początkową podziałkę (odległość początków wierszy). Parametr *outpitch* to podziałka docelowa. Obie podziałki nie mogą być mniejsze niż długość wiersza.

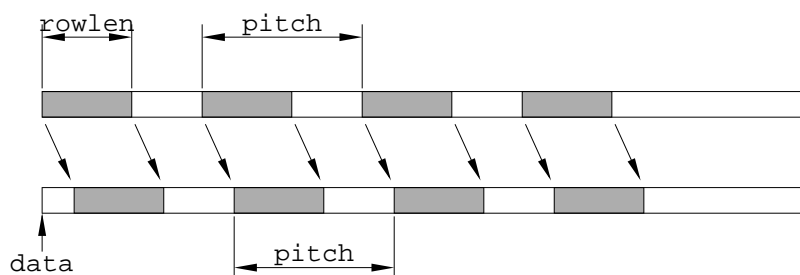


Rys. 2.1. Działanie procedur pkv\_Rearrangec i pkv\_Selectc

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

Procedura pkv\_Selectc przepisuje dane z tablicy indata do tablicy outdata. Dane są umieszczone w nrows wierszach o długości rowlen. Podziałka tablicy indata jest określona przez parametr inpitch, a podziałka tablicy outdata przez outpitch. Tablice te muszą zajmować rozłączne obszary pamięci.

Działanie procedury pkv\_Selectc można zilustrować tym samym rysunkiem (rys. 2.1) co działanie procedury pkv\_Rearrangec, pamiętając, że dane są w tym przypadku przenoszone do *innej* tablicy.



Rys. 2.2. Działanie procedury pkv\_Movec

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

Procedura pkv\_Movec dokonuje „przesunięcia” danych w tablicy o shift miejsc. Parametr nrows określa liczbę wierszy, rowlen — długość każdego wiersza, pitch — podziałkę (która nie zostaje zmieniona). Parametr data jest wskaźnikiem początku pierwszego wiersza przed przesuwaniem. Wartość parametru shift może być dodatnia lub ujemna.

Dane w tablicy między wierszami, o ile nie są nadpisywane przez dane, które procedura pkv\_Movec „przesuwa” na ich miejsce, pozostają niezmienione. Umożliwia to

w szczególności „rozszerzenie” wszystkich wierszy (kosztem obszarów nieużywanych) w celu zrobienia w nich miejsca na dodatkowe elementy, lub „usunięcie” z każdego wiersza pewnych elementów (połączone z wydłużeniem obszaru nieużywanego po każdym wierszu).

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

Procedura `pkv_ZeroMatc` inicjalizuje macierz bajtową, przypisując wartość 0 wszystkim elementom. Zawartość obszarów nieużywanych w tablicy przechowującej wiersze macierzy nie jest zmieniana.

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

Procedura `pkv_ReverseMatc` przestawia wiersze macierzy bajtowej w odwrotnej kolejności. Parametry `nrows` i `rowlen` określają wymiary tej macierzy. Parametr `pitch` opisuje podziałkę tablicy `data` z danymi.

```
#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_Moveef(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_Movevec(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 )

```

Powyższe makra służą do przetwarzania tablic liczb typu float i double w sposób opisany wcześniej.

Makra `pkv_Rearrange` i `pkv_Rearranged` zmieniają podziałkę tablic.

Makra `pkv_Selectf` i `pkv_Selectd` przepisują dane między tablicami o różnych podziałkach.

Makra `pkv_Movef` i `pkv_Moved` przesuwają wiersze w tablicy.

Makra `pkv_ZeroMatf` i `pkv_ZeroMatd` inicjalizują zawartość tablicy przypisując wszystkim elementom wartość 0 (zmiennopozycyjne — to jest trik oparty na fakcie, że wszystkie bity reprezentacji zera zmiennopozycyjnego są równe 0).

Makra `pkv_ReverseMatf` i `pkv_ReverseMatd` odwracają kolejność wierszy macierzy zmiennopozycyjnych.

```

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 );

```

Powyższe procedury przepisują dane (liczby zmiennopozycyjne) z jednej tablicy do drugiej, podobnie jak procedura `pkv_Selectf`. Różnica polega na konwersji; tablica z danymi jest typu float albo double, zaś tablica, do której dane są przepisywane, jest typu double albo float.

Jednostki podziałek tablic (tj. odległości początków kolejnych wierszy) są odpowiednio długościami reprezentacji liczb przechowywanych w danej tablicy.

Procedura `pkv_Selectfd` powinna działać dobrze dla dowolnych danych, które reprezentują liczby. W drugiej z procedur może wystąpić nadmiar lub niedomiar zmiennopozycyjny, a ponadto na ogół wystąpią błędy zaokrągleń, co wynika stąd, że zbiór liczb zmiennopozycyjnych typu float jest podzbiorem zbioru liczb typu double.

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

Procedura `pkv_TransposeMatrixc` dokonuje transpozycji macierzy  $m \times n$ . Parametry `nrows` i `ncols` określają liczby  $m$  i  $n$ . Wielkość (w bajtach) elementu macierzy jest wartością parametru `elemsize`. Kolejne wiersze macierzy wejściowej (z elementami upakowanymi bez przerw) są podane w tablicy `indata` o podziałce (mierzonej w bajtach) `inpitch`. Procedura wpisuje kolejne wiersze macierzy transponowanej do tablicy `outdata` o podziałce `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) ...
```

Powyższe dwa makra służą do wygodnego transponowania macierzy liczbowych, tj. złożonych z elementów typu `float` lub `double`. Parametry tych makr odpowiadają parametrom procedury `pkv_TransposeMatrixc` (z wyjątkiem parametru `elemsize`). Jednostka podziałki tablic jest długością reprezentacji zmiennych typu `float` albo `double`.

## 2.8 Rasteryzacja odcinków

Procedura rasteryzacji odcinków jest umieszczona w bibliotece libpkvaria, ponieważ nie było lepszego miejsca. W przyszłości przyda się napisanie procedury rasteryzacji wielokątów i jeśli to się rozbuduje, to może warto będzie zrobić osobną bibliotekę.

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

Struktura typu xpoint jest przeznaczona do reprezentowania pikseli; ma ona identyczną budowę jak struktura Xpoint zdefiniowana w pliku Xlib.h. Dzięki temu piksele obliczone przez procedury rasteryzacji odcinków (oraz krzywych, z biblioteki libmultibs) można wyświetlać w aplikacji systemu XWindow bez dodatkowej konwersji. Z drugiej strony, dzięki powtórzeniu tej definicji nie jest konieczne korzystanie z pliku Xlib.h, w związku z czym można używać niżej opisanych procedur w programach, które nie są aplikacjami X-ów.

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

Powyższe zmienne służą do rasteryzacji; są to kolejno wskaźnik procedury wywoływania pikseli (która musi być częścią aplikacji), wskaźnik bufora pikseli i licznik pikseli w buforze. Aplikacja do tych zmiennych nie powinna bezpośrednio się odwoływać.

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

Powyższe makra definiują pojemność bufora (256 oznacza rezerwację 1KB na ten cel) oraz realizują jego obsługę. Są one udostępnione w pliku nagłówkowym na potrzeby procedur rasteryzacji krzywych w bibliotece libmultibs.

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

Procedury pomocnicze, z których pierwsza tworzy bufor na piksele, a druga go zwalnia. Bufor jest rezerwowany w pamięci pomocniczej (przez wywołanie procedury pkv\_GetScratchMem), a zatem między wywołaniami tych dwóch procedur trzeba zwolnić tyle pamięci pomocniczej, ile się zarezerwowało.

```
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) );
```

Procedura `_pkv_DrawLine` realizuje algorytm Bresenhama rasteryzacji odcinka. Działa ona przy założeniu, że bufor na piksele jest utworzony (przez wywołanie `_pkv_InitPixelBuffer`), zaś zmienna `_pkv_OutputPixels` wskazuje odpowiednią procedurę wyprowadzającą (np. na ekran) piksele.

Do wywoływania przez aplikację przeznaczona jest procedura `pkv_DrawLine`, której parametry: `x1`, `y1`, `x2`, `y2` określają współrzędne końców odcinka, zaś parametr `output` wskazuje procedurę wyprowadzania pikseli. Procedura `pkv_DrawLine` tworzy i inicjalizuje bufor oraz przypisuje wartość parametru `output` zmiennej `_pkv_OutputPixels`, po czym wywołuje `_pkv_DrawLine`, opróżnia bufor i zwalnia go.

Procedura wskazywana przez parametr `output` jest wywoływana z parametrami, z których pierwszy wskazuje początek bufora (tablicy z pikselami), a drugi określa liczbę pikseli w buforze. Procedura ta może rezerwować pamięć, ale musi ją zwolnić co do jednego bajtu.

## 2.9 Obsługa sytuacji wyjątkowych

Podczas wykonywania programu występują sytuacje błędne, z którymi program musi sobie radzić. Typowy przykład to brak pamięci; w razie niemożności przydzielenia odpowiednio dużego bloku pamięci program musi wykonać co najmniej jedną z następujących czynności:

- Zatrzymać się (przez wywołanie `exit`); gdyby tego nie zrobił, to za chwilę zostałby przerwany przez system operacyjny, z powodu nieodpowiedniego zachowania się, czyli próby dostępu do pamięci pod przypadkowym adresem.
- Poinformować użytkownika przed zatrzymaniem o przyczynie i miejscu wystąpienia sytuacji awaryjnej. W przeciwnym razie użytkownik nie będzie wiedział, dlaczego program mu się wykrzaczył.
- Przewrócić obliczenia niewykonalne z powodu wystąpienia błędu, ale bez zatrzymywania programu. W takim przypadku również należy zwykle poinformować użytkownika, że program nie był w stanie wykonać pewnego obliczenia, ale za to może zrobić coś innego.

W razie wystąpienia sytuacji błędnej procedury biblioteczne wywołują procedurę `pkv_SignalError`. Jej domyślne działanie polega na wypisaniu stosownego komunikatu i zatrzymaniu programu. Aplikacje mogą (za pomocą procedury `pkv_SetErrorHandler`) instalować własne procedury obsługi sytuacji wyjątkowych, które mogą powodować wyświetlanie komunikatów dla użytkownika np. w boksie dialogowym i które mogą doprowadzić (przy użyciu procedur `setjmp` i `longjmp`, przeczytaj opis w poleceniu `man`) do przerwania obliczeń, podczas których wiele wywołanych procedur nie zakończyło jeszcze działania (i ma to zrobić teraz).

```
#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
```

Powyższe nazwy symboliczne identyfikują bibliotekę, do której należy procedura, która sygnalizuje błąd. Aplikacja może określić własne identyfikatory, najlepiej różne od powyższych.

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

Procedura `pkv_SignalError` domyślnie wypisuje komunikat o błędzie na plik `stderr` i zatrzymuje program (wywołując `exit ( 1 );`). Komunikat zawiera nu-

mer błędu (wewnętrzny dla modułu, tj. biblioteki) będący wartością parametru `errno`, numer modułu (parametr `module`) i tekst komunikatu (parametr `errstr`).

Jeśli jest zainstalowana (za pomocą `pkv_SetErrorHandler`) procedura obsługi sytuacji wyjątkowej, to procedura `pkv_SignalError` wywoła tę procedurę, przekazując jej swoje parametry.

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

Procedura `pkv_SetErrorHandler` instaluje procedurę obsługi błędu, która będzie odtąd wywoływana przez `pkv_SignalError`. Podanie parametru `ehandler` o wartości `NULL` powoduje „odinstalowanie” procedury obsługi błędu, tj. przywrócenie domyślnego działania procedury `pkv_SignalError`.

## 2.10 Opakowania procedur malloc i free

W jednym (na razie) programie demonstracyjnym (pomnij) jest tworzony proces potomny (za pomocą procedur `fork` i `exec`), którego zadaniem jest wykonywanie długotrwałych obliczeń numerycznych, podczas gdy interakcja z programem odbywa się normalnie. W szczególności jest możliwość przerywania obliczeń przed ich zakończeniem, w tym celu do procesu potomnego jest wysyłany sygnał `SIGUSR1`, który może pojawić się w zupełnie dowolnej chwili. Przerwanie obliczeń (i powrót procesu potomnego do stanu gotowości przyjęcia następnych poleceń) jest wykonywane za pomocą procedur `setjmp` i `longjmp`.

Dynamiczna alokacja pamięci jest miejscem krytycznym w obliczeniach; nie wolno wykonać skoku (tj. wykonać instrukcji `longjmp`) w trakcie działania procedury `malloc` lub `free`, a także po zakończeniu działania `malloc`, ale przed przypisaniem zwróconej wartości do zmiennej; może to uszkodzić listę wolnych obszarów obsługiwanych przez `malloc` i `free`, a ponadto spowodować wyciekanie pamięci. Dodatkowo, jeśli adresy alokowanych dynamicznie bloków są przechowywane tylko w zmiennych lokalnych jakiejś procedury, to przerywanie działania tej procedury przez `longjmp` musi wiązać się ze zwolnieniem tych bloków. To samo dotyczy bloków wskazywanych przez wskaźnikowe zmienne globalne.

Zamiast bezpośrednich wywołań procedur `malloc` i `free`, należy wywoływać podane niżej makra `PKV_MALLOC` i `PKV_FREE`; **Uwaga:** na razie nie wszystkie procedury biblioteczne robią to. Odpowiednie zmienne globalne zadeklarowane w bibliotece `libpkv` umożliwiają dołożenie obsługi zdarzeń związanych z obsługą przerwania. Jeśli wartości tych zmiennych mają domyślne wartości początkowe, to makra po prostu wywołują `malloc` i `free`.

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

Zmienna `pkv_signal_handler` ma domyślną wartość `NULL`; aplikacja może przypisać tej zmiennej adres procedury, która powinna być wywołana w celu obsłużenia przerwania, ale nie w trakcie działania `malloc` i `free`.

Procedura obsługi sygnału programu (rejestrowana za pomocą `signal`) powinna zacząć działanie od zbadania wartości zmiennej `pkv_critical`. Jeśli zmienna ta ma wartość `true`, to należy tylko wykonać przypisanie `pkv_signal = true`; Jeśli wartością zmiennej `pkv_critical` jest `false`, to można wywołać procedurę wskazywaną przez `pkv_signal_handler`, która wykona `longjmp`. Jeśli po wyjściu z obszaru krytycznego zmienna `pkv_signal` ma wartość `true`, to makro wywołuje tę procedurę (czyli obsługa przerwania jest odrobinę opóźniona, ale następuje).

Zmienna `pkv_register_memblock`, jeśli nie ma wartości `NULL`, wskazuje procedurę, której przekazywany jest adres każdego bloku rezerwowanego lub zwalnianego przez makra `PKV_MALLOC` i `PKV_FREE` (tylko jeśli `pkv_signal_handler` nie ma

wartości NULL). Umożliwia to tworzenie listy zaalokowanych bloków do zwolnienia w razie przerwania obliczeń.

```
#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; \
    } \
}
```

Makro PKV\_FREE oprócz zwolnienia bloku wskazywanego przez parametr makra (za pomocą free), przypisuje temu parametrowi wartość NULL.



## 2.11 Odpluskwanie

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

Powyższe procedury mogą być użyte podczas uruchamiania programu metodą wydruków kontrolnych. Każda z procedur wypisuje na stdout napis name oraz lgt liczb zmiennopozycyjnych z tablicy tab.

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

Powyższe procedury można wywoływać zamiast malloc i free, jeśli zachodzi podejrzenie, że program pisze coś poza obszarami zarezerwowanymi. Procedura DMalloc rezerwuje (za pomocą malloc) blok pamięci większy o 16 bajtów, wypełnia go zerami, wpisuje (w pierwszych czterech bajtach) rozmiar i zwraca wskaźnik do ósmego bajtu za początkiem zarezerwowanego bloku.

Procedura DFree sprawdza, czy bajty 4, ..., 7 oraz ostatnie 8 bajtów zwalnianego bloku ma wartość 0 i wypisuje odpowiednie ostrzeżenie.



### 3. Biblioteka libpknum

W tej bibliotece są procedury realizujące różne metody numeryczne. Chwilowo metody te dotyczą prawie wyłącznie niektórych zadań algebry liniowej, ale to może się zmienić (jeśli trzeba będzie zaspokoić nowe potrzeby).

**DOZRO:** Rzeczą do zrobienia w przyszłości jest przeprowadzenie optymalizacji opisanych tu procedur lub nawet zrealizowanie ich jako interfejsu do procedur BLAS z pakietu LAPACK, co spowoduje przyspieszenie obliczeń. Wymaga to jednak dokonania odpowiednich zmian także w bibliotece libmultibs, której wiele procedur „liczy na własną rękę” zamiast wywoływać odpowiednie procedury z tej biblioteki.

### 3.1 Działania na wektorach i macierzach pełnych

Procedury opisane w tym punkcie przetwarzają macierze reprezentowane w postaci zwykłych tablic zawierających wszystkie współczynniki. Macierze rzadkie, tj. takie, których większość współczynników jest równa 0, najczęściej można i warto reprezentować w inny sposób. W punkcie 3.2 są opisane procedury przetwarzania tzw. macierzy wstęgowych, tj. pewnego rodzaju macierzy rzadkich.

Wiersze i kolumny macierzy  $m \times n$  indeksujemy od 0 do  $m-1$  i od 0 do  $n-1$ .

### 3.1.1 Działania elementarne

[illegible]

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

Powyższe procedury obliczają macierze

$A + B$	<code>pkn_AddMatrixf,</code>
$A - B$	<code>pkn_SubtractMatrixf,</code>
$A + aB$	<code>pkn_AddMatrixMf,</code>
$a(A - B)$	<code>pkn_MatrixMDifferencef,</code>
$aA + bB$	<code>pkn_MatrixLinComb.</code>

Obie macierze dane i wynik mają `nrows` wierszy i `rowlen` kolumn. Współczynniki macierzy  $A$  i  $B$  są podane w tablicach `indata1` i `indata2`. Wynik jest umieszczany w tablicy `outdata`. Podziały tablic są równe odpowiednio `inpitch1`, `inpitch2` i `outpitch`.

**Uwaga:** Istotną cechą procedur przetwarzających macierze jest to, że jeśli między wierszami macierzy umieszczonymi w tablicy występują nieużywane obszary pamięci, to ich zawartość nie jest zmieniana. Na tej własności procedur opierają się różne inne procedury, które mogą w obszarach tych przechowywać dowolne dane, bez obawy, że jakaś procedura na przykład dokona inicjalizacji tablicy wpisując zera gdzie popadnie. Dodatkowo, jest rzeczą dopuszczalną stosowanie ujemnych podziałek, byleby tylko nie prowadziło to do pisania lub czytania elementów macierzy poza obszarem na ten cel zarezerwowanym.

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

Procedura `pkn_MultMatrixNumf` oblicza iloczyn macierzy  $A$  o wymiarach  $m \times n$  ( $m = \text{nrows}$ ,  $n = \text{rowlen}$ ), i liczby  $a$ .

Współczynniki macierzy są podane w tablicy `indata` o podziale `inpitch`, zaś wynik jest wpisywany do tablicy `outdata` o podziale `outpitch`. Liczba  $a$  jest wartością parametru  $a$ .

Jeśli podziałka tablicy `outdata` jest większa niż długość wiersza, to zawartość obszarów nieużywanych między wierszami nie ulega zmianie.

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

Procedura `pkn_MultArrayf` mnoży współczynniki macierzy  $A$  i  $B$ , tj. oblicza  $c_{ij} = a_{ij}b_{ij}$ . Macierze te oraz macierz iloczynów  $C$  mają wymiary  $nrows \times rowlen$ . Podziały tablic  $a$ ,  $b$  i  $c$  ze współczynnikami macierzy  $A$ ,  $B$  i  $C$  są równe odpowiednio `pitch_a`, `pitch_b` i `pitch_c`.

```
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 );
```

Procedura `pkn_MultMatrixf` dokonuje mnożenia macierzy prostokątnych, tj. oblicza macierz  $C = AB$ , gdzie  $A \in \mathbb{R}^{m,n}$ ,  $B \in \mathbb{R}^{n,l}$ , a zatem  $C \in \mathbb{R}^{m,l}$ .

Parametry `nrows_a`, `rowlen_a` i `rowlen_b` mają wartości odpowiednio  $m$ ,  $n$  i  $l$ . Współczynniki macierzy  $A$  i  $B$  należy podać w tablicach  $a$  i  $b$  o podziałkach `pitch_a` i `pitch_b`. Parametr `pitch_c` określa podziałkę tablicy  $c$ , do której procedura wpisuje wynik.

```
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 );
```

Procedura `pkn_MultMatrixAddf` oblicza sumę macierzy i iloczynu macierzy prostokątnych, tj. oblicza macierz  $D = C + AB$ , gdzie  $A \in \mathbb{R}^{m,n}$ ,  $B \in \mathbb{R}^{n,l}$ , oraz  $C, D \in \mathbb{R}^{m,l}$ .

Procedura `pkn_MultMatrixSubf` oblicza macierz  $D = C - AB$ , dla macierzy o wymiarach jak wyżej.

Parametry `nrows_a`, `rowlen_a` i `rowlen_b` mają wartości odpowiednio  $m$ ,  $n$  i  $l$ . Współczynniki macierzy  $A$  i  $B$  należy podać w tablicach  $a$  i  $b$  o podziałkach `pitch_a` i `pitch_b`. Parametr `pitch_c` określa podziałkę tablicy  $c$ , która początkowo zawiera współczynniki macierzy  $C$  i do której procedura wpisuje wynik (współczynniki macierzy  $D$ ).

```
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 );
```

Procedura `pkn_MultTMatrixf` dokonuje mnożenia macierzy prostokątnych, tj. oblicza macierz  $C = A^T B$ , gdzie  $A \in \mathbb{R}^{m,n}$ ,  $B \in \mathbb{R}^{m,l}$ , a zatem  $C \in \mathbb{R}^{n,l}$ .

Parametry `nrows_a`, `rowlen_a` i `rowlen_b` mają wartości odpowiednio  $m$ ,  $n$  i  $l$ . Współczynniki macierzy  $A$  i  $B$  należy podać w tablicach  $a$  i  $b$  o podziałkach `pitch_a` i `pitch_b`. Parametr `pitch_c` określa podziałkę tablicy  $c$ , do której procedura wpisuje wynik.

```
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 );
```

Wartością procedury jest iloczyn skalarny wektorów  $a$  i  $b$  w przestrzeni  $\mathbb{R}^n$ , gdzie wymiar  $n$  jest wartością parametru `spdimen`.

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

Wartością procedury jest norma druga (pierwiastek z sumy kwadratów współrzędnych) wektora  $b$ , w przestrzeni  $\mathbb{R}^n$  o wymiarze  $n = \text{spdimen}$ .

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

Wartością procedury jest wyznacznik macierzy  $A$  o wymiarach  $n \times n$ . Parametr  $n$  określa wymiary macierzy, a w tablicy  $a$  należy podać  $n^2$  współczynników tej macierzy (w kolejnych wierszach lub kolumnach). Procedura niszczy zawartość tej tablicy.

Wyznacznik jest obliczany metodą eliminacji Gaussa z pełnym wyborem elementu głównego.

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

Procedury `pkn_MVectorSumf` i `pkn_MVectorLinComb` obliczają odpowiednio sumę i kombinację liniową  $m$  wektorów w  $\mathbb{R}^n$ . Parametry  $m$  i  $n$  określają liczby  $m$  i  $n$ , które muszą być dodatnie. Parametr `sum` jest wskaźnikiem tablicy, w której

zostanie umieszczony wynik. Po nim, w wywołaniu procedury `pkn_MVectorSumf` należy podać `m` wskaźników do tablic elementów typu `float`, których sumy mają być obliczone.

W wywołaniu procedury `pkn_MVectorLinCombf` po parametrze `sum` należy podać `m` par parametrów; para składa się ze wskaźnika tablicy (typu `float*`) oraz współczynnika kombinacji liniowej typu `double`.

### 3.1.2 Rozwiązywanie układów równań liniowych

Układ równań liniowych  $Ax = b$  z pełną nieosobliwą macierzą kwadratową można rozwiązać za pomocą metody eliminacji Gaussa; procedury opisane w tym punkcie realizują ten algorytm z pełnym wyborem elementu głównego.

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

Procedura `pkn_GaussDecomposePLUQf` oblicza czynniki rozkładu macierzy kwadratowej  $A = P^{-1}LUQ$  o wymiarach  $n \times n$ . Czynniki te to: macierz permutacji  $P^{-1}$ , macierzy trójkątna dolna  $L$  z jedynkami na diagonalu, macierz trójkątna górna  $U$  i macierz permutacji  $Q$ .

Parametr `n` określa wymiary macierzy. Jej współczynniki należy podać w tablicy `a` o długości  $n^2$ ; zawiera ona kolejne wiersze. Procedura umieszcza w tej tablicy obliczone współczynniki macierzy  $L$  i  $U$ . Macierze permutacji  $P$  i  $Q$  są reprezentowane przez liczby wstawione do tablic `P` i `Q` o długości  $n - 1$ .

Wartością procedury jest `true`, jeśli obliczenie zakończyło się sukcesem, albo `false`, w przypadku gdy macierz  $A$  jest osobliwa.

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

Procedura `pkn_multiSolvePLUQf` rozwiązuje układ równań liniowych  $AX = B$ , gdzie macierz  $A$  o wymiarach  $n \times n$  jest kwadratowa nieosobliwa, macierz  $B$  ma wymiary  $n \times d$ .

Macierz  $A$  jest reprezentowana za pomocą jej czynników rozkładu znalezionych przez procedurę `pkn_GaussDecomposePLUQf`. Parametr `n` określa jej wymiary. Parametr `spdimen` określa liczbę kolumn  $d$  macierzy  $B$  i  $X$ . Współczynniki macierzy  $B$  są dane w tablicy `b` o podziałce `pitch`. W tablicy tej procedura umieszcza obliczone rozwiązanie.

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

Procedura `pkn_multiGaussSolveLinEqf` rozwiązuje układ równań  $AX = B$  z nieosobliwą macierzą  $A$  ze względu na macierz  $X$ . W tym celu procedura ta

tworzy kopię tablicy `a` (aby nie psuć jej zawartości) i wywołuje kolejno procedury `pkn_GaussDecomposePLUQf` i `pkn_multiSolvePLUQf`. Wartością procedury jest `true` jeśli obliczenie zakończyło się sukcesem, albo `false`, jeśli nie. Przyczyną niepowodzenia może być osobliwa macierz  $A$ , albo brak pamięci pomocniczej.

Ponieważ dla  $d < n$  największy koszt rozwiązywania wiąże się ze znalezieniem rozkładu macierzy  $A$ , więc jeśli trzeba kolejno rozwiązać wiele układów z tą samą macierzą (i różnymi prawymi stronami), to lepiej jest nie korzystać z tej procedury. Zamiast tego należy raz dokonać rozkładu, a następnie dla każdej kolejnej macierzy  $B$  wywoływać procedurę `pkn_multiSolvePLUQf`.

<pre>boolean pkn_GaussInvertMatrixf ( int n, float *a );</pre>
--

Procedura `pkn_GaussInvertMatrixf` oblicza odwrotność danej macierzy  $A$  o wymiarach  $n \times n$ . Najlepiej jej nie używać wcale.



### 3.1.3 Rozkład QR i liniowe zadania najmniejszych kwadratów

W tym punkcie są opisane procedury dokonujące rozkładu macierzy prostokątnej  $A$  na czynniki ortogonalny  $Q$  i trójkątny górny  $R$ , oraz procedury wykorzystujące ten rozkład do rozwiązywania liniowych zadań najmniejszych kwadratów dla układów równań  $Ax = b$  z pełną macierzą  $A$ .

Macierz ortogonalna  $Q$  reprezentuje przekształcenie, które jest złożeniem ciągu odbić symetrycznych względem pewnych hiperpłaszczyzn; właściwa metoda reprezentowania takiej macierzy polega na przechowywaniu wektorów normalnych hiperpłaszczyzn tych odbić.

Odbicie względem hiperpłaszczyzny jest przekształceniem  $\mathbb{R}^m \rightarrow \mathbb{R}^m$ , którego macierz jest opisana wzorem

$$H_i = I_m - w_i \gamma_i w_i^T, \quad \text{gdzie} \quad \gamma_i = \frac{2}{w_i^T w_i}.$$

Macierz  $I_m$  jest jednostkowa  $m \times m$ . Wektorem normalnym hiperpłaszczyzny odbicia jest wektor  $w_i$ . Odbicia konstruowane w celu otrzymania opisywanego tu rozkładu macierzy to tzw. **odbicia Householdera**. Są one dobierane tak, aby obrazami kolejnych kolumn były kolumny macierzy trójkątnej. Dla przyspieszenia obliczeń w następnych etapach rozwiązywania liniowych zadań najmniejszych kwadratów, oprócz wektorów  $w_i$  należy przechowywać liczby  $\gamma_i$ , których ponowne obliczanie jest możliwe, ale zabiera czas.

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

Procedura `pkn_QRDecomposeMatrixf` dokonuje rozkładu macierzy  $A$ , która ma `nrows` wierszy i `ncols` kolumn, na czynniki ortogonalny  $Q$  i trójkątny  $R$ . Współczynniki macierzy  $A$  należy podać w tablicy `a`, o długości `nrows×ncols`, która zawiera kolejne wiersze macierzy  $A$ .

Po zakończeniu działania procedury tablica `a` zawiera reprezentację znalezionych czynników rozkładu. Współczynniki macierzy  $R$  na i nad diagonalą są umieszczane w odpowiednich miejscach tablicy (współczynnik  $r_{ij}$  dla  $i \leq j$  zajmuje miejsce współczynnika  $a_{ij}$ ). Macierz ortogonalna  $Q$  jest reprezentowana w postaci ciągu wektorów normalnych hiperpłaszczyzn odbić Householdera, które przekształcają macierz  $A$  na  $R$ . Do przechowywania współrzędnych tych wektorów są wykorzystane miejsca w tablicy `a`, w których początkowo były współczynniki  $a_{ij}$  dla  $i > j$ . Ponieważ nie ma tam dość miejsca, pozostałe `ncols` współrzędnych, a także dodatkowe `ncols` współczynników  $\gamma_i$  procedura wstawia do tablicy `aa`. Sposób przechowywania współczynników macierzy  $R$  i reprezentacji odbić jest pokazany na rysunku.

Wartością procedury jest `true`, jeśli obliczenie zakończyło się sukcesem, albo `false`, jeśli kolumny macierzy są liniowo zależne. W tym przypadku zawartości tablic `a` i `aa` są nieokreślone.

$$\begin{aligned}
 \mathbf{a} &= \{r_{00}, r_{01}, r_{02}, r_{03}, \\
 &\quad w_{10}, r_{11}, r_{12}, r_{13}, \\
 &\quad w_{20}, w_{21}, r_{22}, r_{23}, \\
 &\quad w_{30}, w_{31}, w_{32}, r_{33}, \\
 &\quad w_{40}, w_{41}, w_{42}, w_{43}, \\
 &\quad w_{50}, w_{51}, w_{52}, w_{53}\}; \\
 \mathbf{aa} &= \{w_{00}, w_{11}, w_{22}, w_{33}, \\
 &\quad \gamma_0, \gamma_1, \gamma_2, \gamma_3\};
 \end{aligned}
 \quad
 \mathbf{w}_0 = \begin{bmatrix} w_{00} \\ w_{10} \\ w_{20} \\ w_{30} \\ w_{40} \\ w_{50} \end{bmatrix}, \mathbf{w}_1 = \begin{bmatrix} 0 \\ w_{11} \\ w_{21} \\ w_{31} \\ w_{41} \\ w_{51} \end{bmatrix}, \dots$$

Rys. 3.1. Przechowywanie reprezentacji macierzy Q i R dla macierzy  $6 \times 4$ 

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

Procedura `pkg_multiReflectVectorf` oblicza iloczyn macierzy  $Q^{-1}$  i B; czynnik Q jest macierzą ortogonalną, której reprezentacja znaleziona przez procedurę `pkg_QRDecomposeMatrixf` (w postaci ciągu odbić Householdera) jest przechowywana w tablicach a i aa. Macierz B o wymiarach ncols wierszy i spdimen kolumn jest przechowywana w tablicy b. Podziałka tablicy b, czyli odległość pierwszych współczynników kolejnych wierszy, jest wartością parametru pitch.

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

Procedura `pkg_multiInvReflectVectorf` oblicza iloczyn macierzy Q i B; czynnik Q jest macierzą ortogonalną, której reprezentacja znaleziona przez procedurę `pkg_QRDecomposeMatrixf` (w postaci ciągu odbić Householdera) jest przechowywana w tablicach a i aa. Macierz B o wymiarach ncols wierszy i spdimen kolumn jest przechowywana w tablicy b. Podziałka tablicy b, czyli odległość pierwszych współczynników kolejnych wierszy, jest wartością parametru pitch.

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

Procedura `pkg_multiMultUTVectorf` oblicza iloczyn macierzy R i B; czynnik R jest macierzą trójkątną, której reprezentacja znaleziona np. przez procedurę `pkg_QRDecomposeMatrixf` jest przechowywana w tablicy a. Macierz B o wymiarach nrows wierszy i spdimen kolumn jest przechowywana w tablicy b. Podziałka tablicy b, czyli odległość pierwszych współczynników kolejnych wierszy, jest wartością parametru bpitch.

Wynik mnożenia jest umieszczany w tablicy x, o podziałce xpitch.

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

Procedura `pkn_multiMultInvUTVectorf` oblicza iloczyn macierzy  $R^{-1}$  i  $B$ ; macierz  $R$  jest macierzą trójkątną, której reprezentacja znaleziona np. przez procedurę `pkn_QRDecomposeMatrixf` jest przechowywana w tablicy  $a$ . Macierz  $B$  o wymiarach  $nrows$  wierszy i  $spdimen$  kolumn jest przechowywana w tablicy  $b$ . Podziałka tablicy  $b$ , czyli odległość pierwszych współczynników kolejnych wierszy, jest wartością parametru  $bpitch$ .

Wynik mnożenia jest umieszczany w tablicy  $x$ , o podziałce  $xpitch$ .

```
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 );
```

Procedura `pkn_multiSolveRLSQf` rozwiązuje liniowe zadanie najmniejszych kwadratów  $AX = B$ , tj. dokonuje rozkładu QR macierzy  $A$ , (która musi być kolumnowo-regularna), po czym oblicza macierz  $Y = Q^{-1}B$ , a następnie  $X = R_1^{-1}Y$ , gdzie macierz kwadratowa  $R_1$  jest górnym blokiem macierzy  $R$ .

Liczby wierszy i kolumn macierzy  $A$  to  $nrows$  i  $ncols$ . Jej współczynniki należy podać w tablicy  $a$  (należy podać kolejne wiersze bez przerw między nimi). Wymiary macierzy  $B$  to  $nrows$  wierszy i  $spdimen$  kolumn. Jej współczynniki należy podać w tablicy  $b$ , o podziałce  $bpitch$ .

Wynik, tj. współczynniki macierzy  $X$  o wymiarach  $ncols \times spdimen$  procedura wstawia do tablicy  $x$  o podziałce  $xpitch$ .

Jeśli obliczenie zakończyło się sukcesem, to wartością procedury jest `true`. Porażka, sygnalizowana przez wartość `false`, oznacza, że kolumny macierzy  $A$  są liniowo zależne, czyli że zadanie jest nieregularne.

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

Procedura `pkn_QRGetReflectionf` „wyjmuje” reprezentację jednego odbicia Householdera z tablic  $a$  i  $aa$ , w których reprezentacja ta została umieszczona przez procedurę `pkn_QRDecomposeMatrixf`.

Parametry `nrows` i `ncols` opisują wymiary macierzy  $A$ , której czynniki rozkładu QR są podane w tablicach `a` i `aa`. Parametr `nrefl` o wartości  $i$  od 0 do `ncols`-1 określa numer odbicia. Współrzędne wektora normalnego hiperpłaszczyzny odbicia  $w$  są wpisywane do tablicy `w` o długości  $l = \text{nrows} - i$ ; to jest ostatnie  $l$  z `nrows` współrzędnych tego wektora, ponieważ początkowe  $i$  współrzędnych jest równe 0.

Zmienna `*gamma` otrzymuje wartość parametru  $\gamma_i$ . Można też wywołać procedurę z parametrem `gamma=NULL`, który jest wtedy ignorowany.

## 3.2 Obsługa macierzy wstęgowych

### 3.2.1 Reprezentacja i podstawowe procedury

Macierz wstęgowa  $m \times n$  spełnia taki warunek: istnieje liczba  $w$  i dwa niemalejące ciągi liczb,  $j_0 < \dots < j_{m-1}$  oraz  $k_0 < \dots < k_{m-1}$ , takie że współczynnik  $a_{ij}$  (w  $i$ -tym wierszu i  $j$ -tej kolumnie) jest równy 0, jeśli  $j < j_i$  lub  $j \geq k_i$ , przy czym dla każdego  $i$  jest  $k_i - j_i \leq w$ . Liczba  $w$  jest nazywana *szerokością wstęgi* i jeśli jest znacznie mniejsza niż liczba kolumn  $n$ , to do reprezentowania takiej macierzy potrzebne jest znacznie mniej pamięci, a ponadto wiele algorytmów przetwarzania takiej macierzy może wykonać zadanie znacznie mniejszym kosztem niż w przypadku pełnej macierzy.

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

Parametry opisujące macierz wstęgową to: liczba kolumn i ewentualnie wierszy oraz dwie tablice. Pierwsza z nich, `prof`, o długości  $n + 1$  (o 1 większej niż liczba kolumn) zawiera struktury typu `bandm_profile`, opisujące kolejne kolumny macierzy. W drugiej tablicy, `a`, przechowywane są współczynniki macierzy, zgodnie z opisem w pierwszej tablicy.

Wartość `prof[j].firstnz` (od 0 do  $m - 1$ ) jest równa indeksowi pierwszego wiersza, w którym w  $j$ -tej kolumnie współczynnik macierzy jest różny od 0. Wartość `prof[j].ind` jest indeksem miejsca w tablicy współczynników `a`, w którym jest przechowywany ten współczynnik. W kolejnych miejscach tej tablicy przechowywane są kolejne współczynniki tej kolumny. Liczba współczynników z tej kolumny jest równa `prof[j+1].ind - prof[j].ind`. Przykład takiej reprezentacji jest przedstawiony na rys. 3.2.

Aby reprezentować ciąg odbić względem hiperpłaszczyzn, których wektorami normalnymi są  $w_0, \dots, w_{n-1}$  należy utworzyć tablice `a` i `prof`, takie jak dla macierzy wstęgowej. W tablicy `a` przechowywuje się liczby  $\gamma_i$ , a po nich niezerowe współrzędne wektorów  $w_i$  (tak jakby to były kolumny macierzy wstęgowej). Tablica `prof` umożliwia odnalezienie tych współrzędnych. Przykład jest pokazany na rys. 3.3.

Opisana reprezentacja ma na celu oszczędność pamięci w przypadku, gdy pewne odbicia są skonstruowane w celu rozwiązywania zadań najmniejszych kwadratów dla układów równań z macierzą wstęgową  $A$ . Złożenie wszystkich odbić w kolejności uporządkowania kolumn jest przekształceniem o macierzy ortogonalnej  $Q^T$ . Składając te odbicia w odwrotnej kolejności otrzymamy macierz  $Q$ : macierz  $R$ , taka że  $A = QR$ , jest trójkątna górna.

$a_0$	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	$a_{26}$

```
int ncols = 5;
```

```
bandm_profile prof[6] =
```

```
{ {0,0}, {1,4}, {1,10}, {5,17}, {8,23}, {*,27} };
```

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

Rys. 3.2. Przykład macierzy wstęgowej i opisujących ją tablic

$w_5$	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	$w_{36}$

```
int ncols = 5;
```

```
bandm_profile prof[6] =
```

```
{ {0,5}, {1,9}, {2,15}, {3,21},  
  {4,29}, {*,37} };
```

```
float a[37] =
```

```
{  $\gamma_0, \dots, \gamma_4, w_5, \dots, w_{36}$  };
```

↑    ↑    ↑    ↑    ↑  
 $w_0$   $w_1$   $w_2$   $w_3$   $w_4$

Rys. 3.3. Reprezentacja przykładowej macierzy opisującej złożenie odbić.

Kolumny macierzy z lewej strony są wektorami normalnymi hiperpłaszczyzn odbić

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

Procedura `pkn_BandmFindQRMSizes` oblicza długości tablic potrzebnych do przechowywania współczynników macierzy  $Q$  oraz  $R$  — czynników rozkładu ortogonalno-trójkątnego danej macierzy wstęgowej  $A$ . Macierz  $R$  będzie reprezentowana jako macierz wstęgowa w „zwykły” sposób (tj. odpowiednia tablica zawiera jej współczynniki, tak samo jak rozkładana macierz  $A$ ), a macierz  $Q$ , która jest

iloczynem macierzy odbić Householdera będzie reprezentowana za pomocą wektorów określających kolejne odbicia. Oba sposoby reprezentowania tych macierzy są opisane wyżej.

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

Procedura `pkn_BandmQRDecomposeMatrixf` znajduje czynniki rozkładu macierzy wstęgowej  $A$ , tj. macierz ortogonalną  $Q$  i macierz trójkątną  $R$ . Macierz  $A$  o wymiarach  $nrows \times ncols$  jest reprezentowana za pomocą tablicy `aprof`, która opisuje rozmieszczenie niezerowych współczynników w jej kolumnach i macierzy `a`, w której są przechowywane te współczynniki.

Obliczona macierz trójkątna górna  $R$  jest również wstęgowa. Procedura umieszcza jej reprezentację w tablicach `rprof` i `r`. Pierwsza z tych tablic musi mieć długość co najmniej  $ncols+1$ . Druga z nich musi mieć co najmniej długość obliczoną przez wywołaną wcześniej procedurę `pkn_BandmFindQRSizes`.

Macierz ortogonalna  $Q$  jest iloczynem macierzy odbić Householdera sprowadzających macierz  $A$  do postaci trójkątnej. Liczba tych odbić jest równa  $ncols$ , a zatem tablica `qprof` musi mieć długość co najmniej  $ncols+1$ . Długość tablicy `q` do przechowywania współrzędnych wektorów reprezentujących odbicia powinna również być nie mniejsza niż podana przez procedurę `pkn_BandmFindQRSizes`.

**Uwaga:** Liczba kolumn, `ncols`, musi być *mniejsza* niż liczba wierszy, `nrows`; macierze kwadratowe są rozkładane z błędem (do poprawienia kiedyś w przyszłości).

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

Procedura `pkn_multiBandmReflectVectorf` dokonuje  $ncols$  odbić kolumn macierzy  $B$ , która ma `spdimen` kolumn. Kolejne wiersze tej macierzy są podane w tablicy `b`, w której po wykonaniu obliczenia znajduje się wynik. Kolejność wykonywanych odbić jest zgodna z ich uporządkowaniem w tablicach (tj. najpierw względem hiperpłaszczyzny prostopadłej do  $w_0$ , potem  $w_1$  itd.).

Reprezentacja odbić jest podawana w tablicach `qprof` i `q`, zgodnie z wcześniejszym opisem.

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

Procedura `pkn_multiBandmReflectVectorf` dokonuje  $\text{ncols}$  odbić kolumn macierzy  $B$ , która ma  $\text{spdimen}$  kolumn. Kolejne wiersze tej macierzy są podane w tablicy  $b$ , w której po wykonaniu obliczenia znajduje się wynik. Kolejność wykonywanych odbić jest odwrotna do ich uporządkowaniem w tablicach (tj. jeśli  $n = \text{ncols}$ , to najpierw wykonywane jest odbicie względem hiperpłaszczyzny prostopadłej do  $w_{n-1}$ , potem  $w_{n-2}$  itd.).

Reprezentacja odbić jest podawana w tablicach  $qprof$  i  $q$ , zgodnie z wcześniejszym opisem.

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

Procedura `pkn_multiBandmMultVectorf` wykonuje mnożenie macierzy wstęgowej  $A$  o wymiarach  $\text{nrows} \times \text{ncols}$ , reprezentowanej za pomocą tablic  $aprof$  i  $a$  i macierzy  $X$  o wymiarach  $\text{ncols} \times \text{spdimen}$ . Wynik — macierz  $Y = AX$  o wymiarach  $\text{nrows} \times \text{spdimen}$  jest wpisywany do tablicy  $y$ . W tablicach  $x$  i  $y$  są kolejne wiersze macierzy  $X$  i  $Y$ .

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

Procedura `pkn_multiBandmMultInvUTMVectorf` oblicza macierz  $Y = A^{-1}X$ . Macierz  $A$  o wymiarach  $\text{nrows} \times \text{nrows}$  musi być nieosobliwa trójkątna górna. Macierz  $X$  o wymiarach  $\text{nrows} \times \text{spdimen}$  jest reprezentowana za pomocą tablicy  $x$ , w której znajdują się kolejne wiersze. Wynik jest wpisywany do tablicy  $y$ .

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

Procedura `pkn_multiBandmMultTrVectorf` wykonuje mnożenie transpozycji macierzy wstęgowej  $A$  o wymiarach  $m \times n$ , reprezentowanej za pomocą tablic  $aprof$  i  $a$  i macierzy  $X$  o wymiarach  $n \times d$ . Wynik — macierz  $Y = A^T X$  o wymiarach



$nrows \times spdimen$  jest wpisywany do tablicy  $y$ . W tablicach  $x$  i  $y$  są kolejne wiersze macierzy  $X$  i  $Y$ .

Liczba  $m$  jest reprezentowana przez profil macierzy  $A$ ,  $n$  jest wartością parametru  $ncols$ , a  $d$  jest wartością parametru  $spdimen$ .

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

Procedura `pkn_multiBandmMultInvTrUTMVectorf` oblicza macierz  $Y = A^{-T}X$ . Macierz  $A$  o wymiarach  $nrows \times nrows$  musi być nieosobliwa trójkątna górna. Macierz  $X$  o wymiarach  $nrows \times spdimen$  jest reprezentowana za pomocą tablicy  $x$ , w której znajdują się kolejne wiersze. Wynik jest wpisywany do tablicy  $y$ .

### 3.2.2 Rozwiązywanie liniowych zadań najmniejszych kwadratów

Przykład użycia powyższych procedur do rozwiązania regularnego liniowego zadania najmniejszych kwadratów  $Ax = b$ , z kolumnowo-regularną macierzą wstęgową  $A$ :

1. Utwórz reprezentację macierzy  $A$ .
2. Wywołaj procedurę `pkn_BandmFindQRMSizes` i zarezerwuj tablice o obliczonych przez tę procedurę długościach na przechowanie reprezentacji czynników rozkładu  $Q$  i  $R$  macierzy  $A$ .
3. Wywołaj `pkn_BandmQRDecomposeMatrixf` w celu dokonania rozkładu macierzy  $A$ .
4. Oblicz wektor  $y = Q^T b$  za pomocą `pkn_multiBandmReflectVectorf`.
5. Oblicz  $x = R_1^{-1} y_1$ , gdzie macierz  $R_1$  jest blokiem  $n \times n$  złożonym z początkowych wierszy macierzy  $R$ , a wektor  $y_1$  składa się z pierwszych  $n$  współrzędnych wektora  $y$ .  
W tym celu wywołaj procedurę `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 );

```

Procedura `pkn_multiBandmSolveRLSQf` rozwiązuje w opisany wyżej sposób z liniowych zadań najmniejszych kwadratów, opisanych wspólnie przez układ równań

$$A[x_0, \dots, x_{z-1}] = [b_0, \dots, b_{z-1}].$$

Macierz wstęgowa  $A$  o wymiarach  $m \times n$  (podanych jako parametry `nrows` i `ncols`) jest reprezentowana przez tablice `aprof` i `a`. Tablica `b` o długości `bpitch`×`z` opisuje prawe strony układów, tj. macierze  $b_0, \dots, b_{z-1}$  z których każda ma wymiary  $m \times d$  i jej kolejne wiersze są umieszczone w tablicy po kolei (bez przerw). Pozycje pierwszych współczynników kolejnych macierzy różnią się o wartość parametru `bpitch`. Każda z `d` kolumn każdej macierzy jest jednym wektorem prawej strony układu (czyli w rzeczywistości procedura rozwiązuje  $z \times d = \text{nrsides} \times \text{spdimen}$  zadań, z tą samą macierzą  $A$  i różnymi prawymi stronami).

Rozwiązania zadań są kolumnami macierzy  $x_0, \dots, x_{z-1}$ , których współczynniki (kolejne wiersze, bez przerw) procedura wpisuje do tablicy `x`. Tablica ta musi mieć długość co najmniej `xpitch`×`z`. Parametr `xpitch` określa odległość w tablicy `x` początkowych współczynników kolejnych macierzy  $x_i$ .

### 3.2.3 Rozwiązywanie zadań regularnych z więzami

Regularne liniowe zadanie najmniejszych kwadratów z więzami polega na znalezieniu wektora  $x$  spełniającego układ równań liniowych

$$Cx = d,$$

zwany układem więzów, i takiego, że wektor  $r = Ax - b$  ma najmniejszą normę drugą, przy czym macierz  $A \in \mathbb{R}^{m,n}$  jest kolumnowo regularna, a macierz  $C \in \mathbb{R}^{w,n}$  jest wierszowo regularna.

Liniowa niezależność wierszy macierzy  $C$  zapewnia niesprzeczność układu więzów i postawione wyżej zadanie ma jednoznaczne rozwiązanie. Można je znaleźć, rozwiązując układ równań liniowych

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

Numeryczna metoda rozwiązywania przebiega w następujących krokach:

1. Dokonaj rozkładu macierzy  $A$  na czynniki  $Q$  i  $R$ , takie że  $Q$  jest macierzą ortogonalną, a  $R$  trójkątną górną.

2. Rozwiąż układ równań liniowych  $R^T E = C^T$ .
3. Dokonaj rozkładu macierzy  $E$  na czynniki  $U$  i  $F$ , takie że  $U$  jest macierzą ortogonalną, a  $F$  trójkątną górną. Oznacz symbolem  $F_1$  macierz  $w \times w$ , która jest blokiem macierzy  $F$  składającym się z jej początkowych  $w$  wierszy.
4. Korzystając z rozkładu QR macierzy  $A$  oblicz rozwiązanie  $x_0$  regularnego zadania najmniejszych kwadratów, rozwiązując układ równań  $R_1 x_0 = y_1$  (zobacz opis w p. 3.2.2).
5. Rozwiąż układy równań  $F_1^T e = d - Cx_0$  oraz  $F_1 f = e$ .
6. Rozwiąż układy  $R_1^T g = C^T f$  i  $R_1 h = g$ .
7. Oblicz  $x = x_0 + h$ .

```
void pkn_multiBandmSolveCRLSf ( 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 );
```

Procedura `pkn_multiBandmSolveCRLSf` rozwiązuje z regularnych liniowych zadań najmniejszych kwadratów z więzami, opisanych łącznie przez układ równań

$$A[x_0, \dots, x_{z-1}] = [b_0, \dots, b_{z-1}],$$

z macierzą wstęgową  $A$  przy czym więzy są opisane przez układ równań

$$C[x_0, \dots, x_{z-1}] = [d_0, \dots, d_{z-1}],$$

w którym macierz  $C$  jest pełna.

Parametry: `nrows`, `ncols` — liczba wierszy  $m$  i kolumn  $n$  macierzy  $A$ , `aprof`, `a` — profil (tj. reprezentacja sposobu rozmieszczenia niezerowych współczynników) i tablica z niezerowymi współczynnikami macierzy  $A$ , `nconstr` — liczba w więzów (musi być mniejsza niż  $n$ ), `cpitch` — podziałka (tj. odległość początków kolejnych wierszy) tablicy `c` ze współczynnikami macierzy  $C$ , `nrsides` — liczba  $z$ , `spdimen` — długość wiersza każdej macierzy  $b_i$  oraz  $x_i$  i  $d_i$ , `bpitch` — podziałka tablicy `b` ze współczynnikami macierzy  $b_0, \dots, b_{z-1}$  (tj. odległość początkowych współczynników kolejnych macierzy; wiersze każdej z nich są przechowywane bez przerw), `dpitch` — podziałka tablicy `d` ze współczynnikami macierzy  $d_0, \dots, d_{z-1}$ , `xpitch` — podziałka tablicy `x`, w której ma być umieszczony wynik.

Parametr `d` może być równy `NULL` — wtedy układ równań więzów jest jednorodny.

### 3.2.4 Rozwiązywanie dualnych zadań najmniejszych kwadratów

Dualne zadanie najmniejszych kwadratów polega na znalezieniu rozwiązania  $\mathbf{x}$  układu równań  $A\mathbf{x} = \mathbf{b}$  z macierzą  $A \in \mathbb{R}^{m,n}$  wierszowo-regularną, takiego że dla wskazanego wektora  $\mathbf{x}_0 \in \mathbb{R}^n$  liczba  $\|\mathbf{x} - \mathbf{x}_0\|_2$  jest najmniejsza. Poprzednio opisane procedury mogą być użyte do rozwiązania zadania, jeśli program utworzy reprezentację wstęgową macierzy  $A^T$ .

1. Utwórz reprezentację wstęgową macierzy  $A^T$ .
2. Wywołaj procedurę `pkn_BandmFindQRSizes` i zarezerwuj tablice o odpowiednich długościach na przechowanie reprezentacji czynników  $Q$  i  $R$  rozkładu macierzy  $A^T$ .
3. Wywołaj `pkn_BandmQRDecomposeMatrixf` w celu dokonania rozkładu macierzy  $A^T$  (co jest równoważne rozłożeniu macierzy  $A$  na czynniki  $R^T$  i  $Q^T$ ).
4. Oblicz wektor  $\mathbf{z}_0 = Q^T \mathbf{x}_0$ , wywołując `pkn_multiBandmReflectVectorf`. Jeśli  $\mathbf{x}_0 = \mathbf{0}$ , to można zamiast tego przypisać (bez liczenia)  $\mathbf{z}_0 = \mathbf{0}$ .
5. Rozwiąż układ równań liniowych  $R_1^T \mathbf{z}_1 = \mathbf{b}$ , wykonując w tym celu procedurę `pkn_multiBandmMultInvTrUTMVectorf`. Jeśli  $\mathbf{b} = \mathbf{0}$ , to można przypisać (bez liczenia)  $\mathbf{z}_1 = \mathbf{0}$ . Oblicz wektor  $\mathbf{z}$ , którego początkowe  $m$  współrzędnych jest równych odpowiednim współrzędnym wektora  $\mathbf{z}_1$ , a pozostałe to współrzędne wektora  $\mathbf{z}_0$ .
6. Wykonaj procedurę `pkn_multiBandmInvReflectVectorf` w celu obliczenia rozwiązania zadania, tj. wektora  $\mathbf{x} = Q\mathbf{z}$ .

```
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 );
```

Procedura `pkn_multiBandmSolveDLSQf` rozwiązuje dualne zadania najmniejszych kwadratów w sposób opisany wyżej. Parametry `nrows` (liczba wierszy,  $n$ ), `ncols` (liczba kolumn,  $m$ ), `atprof` (profil) i `at` (tablica współczynników) opisują macierz  $A^T$ .

Prawe strony układu są opisane przez `z = nrsides` macierz  $\mathbf{b}$ , które mają  $m$  wierszy i `d = spdimen` kolumn; każda kolumna jest prawą stroną jednego zadania (zatem procedura rozwiązuje  $z$  zadań z tą samą macierzą  $A$ ). Kolejne wiersze macierzy  $\mathbf{b}$

---

należy podać w tablicy  $b$ . Pozycje pierwszych współczynników kolejnych macierzy  $b$  różnią się o wartość parametru  $bpitch$ . Parametr  $b$  może też mieć wartość NULL, co oznacza, że prawe strony układów w zadaniu są wektorem zerowym.

Przybliżenia rozwiązań zadań są kolumnami macierzy  $x_0$  o wymiarach  $n \times d$ . Kolejne wiersze tej macierzy należy podać w tablicy  $x0$ . Pozycje pierwszych współczynników kolejnych macierzy  $x_0$  różnią się o wartość parametru  $x0pitch$ . Jeśli parametr  $x0$  ma wartość NULL, to oznacza to, że macierze  $x_0$  są zerowe.

Rozwiązania zadań są kolumnami macierzy  $x$ . Kolejne wiersze tych macierzy procedura umieszcza w tablicy  $x$ , która musi mieć długość co najmniej  $xpitch \times z$ .

### 3.2.5 Odpluskwianie

W tym punkcie są opisane procedury, które wypisują na stdout macierze w postaci tekstowej. Takie procedury nieraz bardzo pomagają znaleźć błąd w programie.

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

Procedura `pkn_PrintMatf` wypisuje współczynniki macierzy  $A$ , reprezentowanej w postaci jawnej w tablicy.

Parametry `nrows` i `ncols` opisują odpowiednio liczbę wierszy i kolumn. Tablica `a` zawiera współczynniki macierzy  $A$  — kolejno pierwszy wiersz, potem drugi itd.

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

Procedura `pkn_PrintBandmf` wypisuje macierz wstęgową reprezentowaną przez tablice `aprof` i `a`. Może ona być użyteczna podczas uruchamiania programu.

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

Procedura `pkn_PrintBandmRowSumf` wypisuje macierz wstęgową reprezentowaną przez tablice `aprof` i `a`. Na końcu każdego wiersza procedura wypisuje sumę współczynników w tym wierszu.

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

Procedura `pkn_PrintProfile` wypisuje zawartość tablicy `prof`, czyli profil macierzy wstęgowej.

### 3.3 Obsługa „spakowanych” macierzy symetrycznych i trójkątnych

Kwadratowa macierz symetryczna  $n \times n$  może być reprezentowana za pomocą  $\frac{1}{2}(n+1)n$  liczb, czyli prawie dwa razy mniej niż macierz dowolna o tych samych wymiarach. Także macierze trójkątne dolne i górne mogą być reprezentowane bez potrzeby przechowywania współczynników, o których wiadomo, że są równe 0. Poniżej opisane procedury służą do przetwarzania oszczędnej reprezentacji takich macierzy.

$$A = \begin{bmatrix} a_{00} & a_{10} & a_{20} & a_{30} \\ a_{10} & a_{11} & a_{21} & a_{31} \\ a_{20} & a_{21} & a_{22} & a_{32} \\ a_{30} & a_{31} & a_{32} & a_{33} \end{bmatrix} \quad \begin{array}{l} \text{int } n = 4; \\ \text{float } a[] = \{a_{00}, a_{10}, a_{11}, a_{20}, a_{21}, a_{22}, \\ \quad a_{30}, a_{31}, a_{32}, a_{33}\}; \end{array}$$

Rys. 3.4. Reprezentacja macierzy symetrycznej

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

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

Rys. 3.5. Reprezentacja macierzy trójkątnych

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

Makro `pkn_SymMatIndex` służy do obliczania indeksu współczynnika  $a_{ij}$  macierzy symetrycznej  $A$  w tablicy, w której są przechowywane jej współczynniki. Jednocześnie jest to indeks współczynnika  $l_{ij}$  macierzy trójkątnej dolnej  $L$ , jeśli  $i \geq j$  (w przeciwnym razie  $l_{ij} = 0$ ).

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

Procedura `pkn_CholeskyDecompf` dokonuje rozkładu macierzy symetrycznej dodatnio określonej  $A$  (tj. takiej, że  $A^T = A$  oraz  $\forall_{x \neq 0} x^T A x > 0$ ) na czynniki trójkątne:  $A = LL^T$ . Współczynniki macierzy trójkątnej dolnej  $L$  są umieszczane w tablicy `a`, w której początkowo znajdują się współczynniki macierzy  $A$ .

Parametr `n` określa wymiary macierzy. Wartością procedury jest `true`, jeśli rozkład został dokonany z powodzeniem, zaś `false`, jeśli w trakcie obliczeń okazało

się, że macierz  $A$  nie jest dodatnio określona. W tym przypadku zawartość tablicy  $a$  jest nieokreślona.

```
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 );
```

Procedury `pkn_SymMatrixMultf`, `pkn_LowerTrMatrixMultf` oraz `pkn_UpperTrMatrixMultf` obliczają iloczyn  $X$  macierzy odpowiednio symetrycznej, trójkątnej dolnej i trójkątnej górnej (reprezentowanej w opisany wyżej oszczędny sposób)  $A$  o wymiarach  $n \times n$  i macierzy  $B$  o wymiarach  $n \times d$ , której reprezentacja jest „pełna”.

Parametry  $n$  i  $spdimen$  określają wymiary  $n$  i  $d$  macierzy. Parametr  $a$  lub  $l$  jest wskaźnikiem tablicy współczynników macierzy  $A$ , spakowanych w „oszczędny” sposób. Parametr  $b$  jest wskaźnikiem tablicy współczynników macierzy  $B$ , o podziałce  $bpitch$ . Parametry  $x$  i  $xpitch$  to odpowiednio wskaźnik i podziałka tablicy, w której procedura ma umieścić wynik.

**Uwaga.** Nie ma specjalnej procedury mnożenia macierzy symetrycznych, ponieważ iloczyn w ogólności nie jest macierzą symetryczną. Nie ma też mnożenia dwóch macierzy trójkątnych dolnych lub trójkątnych górnych, bo póki co nie miałem potrzeby, żeby je napisać. W razie czego można dokonać konwersji reprezentacji do postaci pełnej (za pomocą procedur opisanych dalej) i użyć procedur mnożenia macierzy pełnych, lub napisać własne procedury.

```
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 );
```

Procedury `pkn_LowerTrMatrixSolvef` i `pkn_UpperTrMatrixSolvef` rozwiązują układy równań odpowiednio z macierzą trójkątną dolną  $L$  i górną  $L^T$ , reprezentowanymi w „oszczędnej” postaci, czyli efektywnie dokonują mnożenia macierzy prawej strony  $B$  przez macierz  $L^{-1}$  lub  $L^{-T}$ .

Parametry  $n$  i  $spdimen$  określają wymiary macierzy  $L$ :  $n \times n$  i  $B$  oraz  $X$ :  $n \times d$ . Tablice  $l$  i  $b$  zawierają współczynniki macierzy  $L$  i  $B$ . Procedury wpisują wynik do tablicy  $x$ . Parametry  $bpitch$  i  $xpitch$  określają podziałki tablic  $b$  i  $x$ .



Można podać tę samą tablicę jako oba parametry:  $\mathbf{b}$  i  $\mathbf{x}$ ; w tym przypadku wynik obliczeń będzie umieszczony w tablicy zamiast początkowej zawartości (czyli wektora prawej strony), ale wtedy parametry  $\text{bpitch}$  i  $\text{xpitch}$  muszą mieć identyczne wartości. W razie podania różnych tablic, zawartość tablicy  $\mathbf{b}$  pozostanie niezmieniona.

Aby rozwiązać układ równań liniowych  $\mathbf{Ax} = \mathbf{b}$  z macierzą symetryczną dodatnio określoną  $\mathbf{A}$ , można użyć procedury `pkn_CholeskyDecomp`, która obliczy współczynniki macierzy  $\mathbf{L}$  takiej, że  $\mathbf{A} = \mathbf{LL}^T$ , a następnie rozwiązać układ równań liniowych  $\mathbf{Ly} = \mathbf{b}$  za pomocą `pkn_LowerTrMatrixSolve` i  $\mathbf{L}^T\mathbf{x} = \mathbf{y}$  za pomocą `pkn_UpperTrMatrixSolve`. Całe to postępowanie jest metodą szybszą niż użycie odpowiedniego algorytmu dla macierzy dowolnej (np. eliminacji Gaussa lub odbić Householdera).

```
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 );
```

Powyższe procedury (i jedno makro) służą do dokonywania konwersji macierzy symetrycznych i trójkątnych, między reprezentacjami „oszczędnymi” i pełnymi.

```
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 );
```

Procedury `pkn_ComputeQSQTf` i `pkn_ComputeQTSQf` obliczają odpowiednio iloczyny macierzy

$$\mathbf{QSQ}^T \text{ oraz } \mathbf{Q}^T\mathbf{SQ},$$

gdzie  $\mathbf{S}$  jest macierzą symetryczną  $m \times m$ , reprezentowaną w postaci spakowanej, zaś macierz  $\mathbf{Q}$  jest ortogonalna,  $m \times m$ . Macierz  $\mathbf{Q}$  reprezentuje złożenie  $n$  odbić Householdera, otrzymanych podczas rozkładu ortogonalno-trójkątnego macierzy  $\mathbf{A}$  o wymiarach  $m \times n$ , zgodnie z opisem w p. 3.1.3.

Parametry  $m$  i  $n$  opisują wymiary macierzy  $S$  i  $A$ . Tablica  $s$  zawiera współczynniki macierzy  $S$ . Tablice  $a$  i  $aa$  zawierają reprezentację odbić (tj. macierzy  $Q$ ), zgodnie z opisem w p. 3.1.3. Współczynniki iloczynu macierzy, który jest macierzą symetryczną, są wpisywane do tablicy  $b$ . Można podać parametry  $b=s$ ; wtedy współczynniki macierzy  $S$  zostaną w tablicy zastąpione przez współczynniki iloczynu.

Zastosowany w procedurach algorytm Ortegi-Householdera jest następujący; dla każdego kolejnego odbicia, reprezentowanego przez macierz  $H_i = I_m - v_i \beta_i v_i^T$ , obliczane są kolejno dla  $i = 0, \dots, n-1$

$$\begin{aligned} \text{wektor } u &= B_{i-1} v_i \beta_i, \\ \text{wektor } p &= u - v_i v_i^T u \beta_i / 2, \\ \text{macierz } B_i &= B_{i-1} - (v_i p^T + p v_i^T), \end{aligned}$$

przy czym  $B_0 = S$ , zaś  $v_i = w_i$ ,  $\beta_i = \gamma_i$  dla procedury `pkn_ComputeQTSQf`, oraz  $v_i = w_{n-i-1}$ ,  $\beta_i = \gamma_{n-i-1}$  dla procedury `pkn_ComputeQSQTf`. Końcowy wynik to macierz  $B_{n-1}$ .

```
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 Obsługa symetrycznych i trójkątnych macierzy o nieregularnej wstędze

Macierze  $n \times n$  symetryczne i trójkątne o nieregularnej wstędze są reprezentowane za pomocą dwóch tablic: profilu i tablicy współczynników. Profil jest tablicą liczb całkowitych o długości  $n$ ; jej  $i$ -ty element jest indeksem pierwszego niezerowego współczynnika macierzy w  $i$ -tym wierszu (wiersze i kolumny są indeksowane od 0). Przykłady są na rysunkach 3.6 i 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};
```

Rys. 3.6. Reprezentacja macierzy symetrycznej o nieregularnej wstędze

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

```
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};
```

Rys. 3.7. Reprezentacja macierzy trójkątnych o nieregularnej wstędze

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

Procedura `pkn_NRBArraySize` oblicza długość tablicy współczynników na podstawie profilu.

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

Procedura `pkn_NRBFindRowsf` umieszcza w tablicy `row` o długości  $n$  wskaźniki początków wirtualnych wierszy; dostęp do współczynnika  $a_{ij}$ , daje wyrażenie `row[i][j]`, ale musi być spełniony warunek `prof[i] ≤ j ≤ i`.

Opisane niżej procedury mają parametr `row`, który może być równy `NULL`; wtedy wywoływana jest przez nie procedura `pkn_NRBFindRowsf`. Można również jednorazowo utworzyć tablicę `row`, wywołać `pkn_NRBFindRowsf`, a następnie przekazywać

ją jako parametr tych procedur. Oszczędza to trochę czasu (ale może się to przydać również w celu obliczenia współczynników macierzy przez aplikację).

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

Procedura pkg\_NRBCholeskyDecompf dokonuje rozkładu metodą Choleskiego macierzy symetrycznej dodatnio określonej  $A$  na czynniki trójkątne  $L$  i  $L^T$ . Współczynniki macierzy  $L$  są wpisywane do tablicy  $a$ , w której zastępują współczynniki danej macierzy  $A$ . Profil macierzy  $L$  jest taki sam jak profil  $A$ .

```
boolean pkg_NRBSymMultf ( int n, const int *prof,
                          const float *a, const float **row,
                          int spdimen, int xpitch, const float *x,
                          int ypitch, float *y );
boolean pkg_NRBLowerTrMultf ( int n, const int *prof,
                              const float *a, const float **row,
                              int spdimen, int xpitch, const float *x,
                              int ypitch, float *y );
boolean pkg_NRBUpperTrMultf ( int n, const int *prof,
                              const float *a, const float **row,
                              int spdimen, int xpitch, const float *x,
                              int ypitch, float *y );
```

Procedury pkg\_NRBSymMultf, pkg\_NRBLowerTrMultf i pkg\_NRBUpperTrMultf obliczają odpowiednio iloczyn macierzy symetrycznej, trójkątnej dolnej lub trójkątnej górnej o nieregularnej wstędze i macierzy pełnej  $X$ .

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

Procedury pkg\_NRBLowerTrSolvef i pkg\_NRBUpperTrSolvef rozwiązują odpowiednio układ równań liniowych z macierzą trójkątną dolną lub górną o nieregularnej wstędze.

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

Procedura `pkn_NRBSymFindEigenvalueIntervalf` znajduje, na podstawie twierdzenia Gershgorina, przedział, w którym znajdują się wszystkie wartości własne macierzy symetrycznej o nieregularnej wstędze.

Parametry `n`, `prof`, `a` i `row` reprezentują macierz.

Parametry `amin`, `amax` są wskaźnikami do zmiennych, którym procedura ma przypisać znalezione liczby.

Wartość powrotna `true` oznacza sukces, `false` — porażkę, która może być spowodowana brakiem pamięci, jeśli parametr `row` ma wartość `NULL` i procedura musi sama utworzyć tablicę wskaźników wierszy na podstawie profilu (błąd może być wtedy wykryty przez procedurę `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 );
```

Danymi dla powyżej opisanych procedur są: macierz symetryczna  $S$  o wymiarach  $n \times n$  i o nieregularnej wstędze, oraz macierz ortogonalna  $Q$ , reprezentowana w postaci ciągu  $w$  odbić Householdera przestrzeni  $\mathbb{R}^n$  (gdzie  $w < n$ ). Macierz  $Q$  może być otrzymana z rozkładu ortogonalno-trójkątnego macierzy  $B$  o wymiarach  $n \times w$ , za pomocą procedury `pkn_QRDecomposeMatrixf`.

Zadaniem procedury `pkn_NRBComputeQTSQf` jest obliczenie macierzy  $C = Q^T S Q$ .

Zadaniem procedury `pkn_NRBComputeQSQTf` jest obliczenie macierzy  $D = Q S Q^T$ .

W obu przypadkach wynik obliczeń jest reprezentowany jako macierz symetryczna o nieregularnej wstędze.

Parametry wejściowe: `n`, `prof`, `Amat`, `Arows` — reprezentacja macierzy  $S$ .

**Uwaga:** w obecnej wersji parametr `Arows` nie może być wskaźnikiem pustym, musi wskazywać tablicę  $n$  wskaźników wirtualnych wierszy.

Parametry `n`, `w`, `Bmat`, `bb` reprezentują macierz  $Q$  w sposób opisany w p. 3.1.3. Liczba `w` jest liczbą odbić, kolumny macierzy w tablicy `Bmat` zawierają współrzędne wektorów normalnych hiperpłaszczyzn odbić  $w_i$  (oprócz początkowych zerowych i pierwszego niezerowego), w tablicy `bb` są podane pierwsze niezerowe współrzędne wektorów  $w_i$  i liczby  $\gamma_i$ .

Parametry wyjściowe: `qaprof` — wskaźnik do tablicy o długości  $n$ , w której zostanie umieszczony profil macierzy  $C$  albo  $D$  (tablica ta ma być zaalokowana przez program przed wywołaniem procedury), `QArows` — wskaźnik do tablicy o długości

ci  $n$ , w której zostaną umieszczone wskaźniki wirtualnych wierszy macierzy wynikowej. Współczynniki tej macierzy są umieszczone w tablicy zarezerwowanej przy użyciu procedury `malloc`; adres początku tej tablicy (do zdealokowania za pomocą `free`) jest adresem pierwszego wirtualnego wiersza.

Wartość `true` procedury oznacza sukces, wartość `false` porażkę, która może być spowodowana brakiem dostatecznej pamięci w puli pamięci pomocniczej lub w stercie obsługiwanej przez `malloc` i `free`.

**Uwaga:** w zastosowaniach praktycznych bardziej pożyteczne mogą się okazać opisane niżej procedury `pkn_NRBCComputeQTSQblf` i `pkn_NRBCComputeQSQTblf`, które rozwiązują te same zadania, ale tworzą reprezentację wyniku podzielonego na bloki.

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

Danymi dla powyżej opisanych procedur są: macierz symetryczna  $S$  o wymiarach  $n \times n$  i o nieregularnej wstędze, oraz macierz ortogonalna  $Q$ , reprezentowana w postaci ciągu w odbić Householdera przestrzeni  $\mathbb{R}^n$  (gdzie  $w < n$ ). Macierz  $Q$  może być otrzymana z rozkładu ortogonalno-trójkątnego macierzy  $B$  o wymiarach  $n \times w$ , za pomocą procedury `pkn_QRDecomposeMatrixf`.

Zadaniem procedury `pkn_NRBCComputeQTSQblf` jest obliczenie bloków macierzy  $C = Q^T S Q$ .

Zadaniem procedury `pkn_NRBCComputeQSQTblf` jest obliczenie bloków macierzy  $D = Q S Q^T$ .

Wynik, np. macierz  $C$ , składa się z bloków:

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

Bloki  $C_{11}$  i  $C_{22}$ , o wymiarach odpowiednio  $w \times w$  i  $n - w \times n - w$ , są macierzami symetrycznymi i są reprezentowane jako macierze o nieregularnej wstędze. Blok  $C_{21}$  o wymiarach  $n - w \times w$  jest reprezentowany jako macierz pełna.

Parametry wejściowe są identyczne, jak dla dwóch procedur opisanych poprzednio:  $n$ ,  $prof$ ,  $Amat$ ,  $Arows$  — reprezentacja macierzy  $S$ .

**Uwaga:** w obecnej wersji parametr  $Arows$  nie może być wskaźnikiem pustym, musi wskazywać tablicę  $n$  wskaźników wirtualnych wierszy.

Parametry  $n$ ,  $w$ ,  $Bmat$ ,  $bb$  reprezentują macierz  $Q$  w sposób opisany w p. 3.1.3. Liczba  $w$  jest liczbą odbić, kolumny macierzy w tablicy  $Bmat$  zawierają współrzędne wektorów normalnych hiperpłaszczyzn odbić  $w_i$  (oprócz początkowych zerowych i pierwszego niezerowego), w tablicy  $bb$  są podane pierwsze niezerowe współrzędne wektorów  $w_i$  i liczby  $\gamma_i$ .

Parametry wyjściowe: `qa11prof` i `qa22prof` — wskaźniki do tablicy o długościach odpowiednio  $w$  i  $n - w$ , w których zostaną umieszczone profile macierzy  $C_{11}$  i  $C_{22}$  (tablice te mają być zaalokowane przez program przed wywołaniem procedury), `QA11rows` i `QA22rows` — wskaźniki do tablic o długościach  $w$  i  $n - w$ , w których zostaną umieszczone wskaźniki wirtualnych wierszy macierzy  $C_{11}$  i  $C_{22}$ . Parametr `QA21` jest adresem zmiennej, która otrzyma wartość wskazującą tablicę ze współczynnikami bloku  $C_{12}$  (reprezentowaną wiersz po wierszu, bez przerw).

Wszystkie współczynniki macierzy wynikowej są umieszczone w tablicy zarezerwowanej przy użyciu procedury `malloc`; adres początku tej tablicy (do zdealokowania za pomocą `free`) jest adresem pierwszego wirtualnego wiersza bloku  $C_{11}$ .

Wartość `true` procedury oznacza sukces, wartość `false` porażkę, która może być spowodowana brakiem dostatecznej pamięci w puli pamięci pomocniczej lub w sterwie obsługiwanej przez `malloc` i `free`.



## 3.5 Obsługa blokowych macierzy symetrycznych

### 3.5.1 Macierze o strukturze blokowej pierwszego rodzaju

Procedury wypełniania wielokątnych otworów w bibliotece `libg2hole` muszą rozwiązywać układy równań liniowych z symetrycznymi macierzami dodatnio określonymi o strukturze blokowej — z blokami zerowymi poza diagonalą i ostatnim wierszem oraz kolumną. Przykład takiej macierzy jest na rysunku 3.8

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

Rys. 3.8. Struktura symetrycznej macierzy blokowej i blokowej macierzy trójkątnej dolnej

Struktura macierzy jest określona za pomocą trzech liczb. Pierwsza ( $k$ ) jest liczbą bloków na diagonalu bez ostatniego, druga ( $r$ ) określa wymiary tych bloków, a trzecia ( $s$ ) określa wymiary ostatniego bloku na diagonalu. Macierz ma zatem wymiary  $(kr + s) \times (kr + s)$ .

Współczynniki takiej macierzy są przechowywane w tablicy `A`. Bloki diagonalne są reprezentowane w „spakowanej” postaci, opisanej w poprzednim punkcie, bloki poddiagonalne w postaci pełnej. Tablica `A` musi mieć zatem długość  $kr(r+1)/2 + s(s+1)/2 + krs$ .

Większość obliczeń procedury opisane niżej wykonują wywołując procedury obsługi macierzy pełnych i „spakowanych” macierzy symetrycznych.

```
int pkn_Block1ArraySize ( int k, int r, int s );
int pkn_Block1FindBlockPos ( int k, int r, int s, int i, int j );
int pkn_Block1FindElemPos ( int k, int r, int s, int i, int j );
```

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

Procedura `pkn_Block1CholeskyDecompMf` dokonuje rozkładu macierzy blokowej `A` na czynniki trójkątne `L` i `LT`. Współczynniki macierzy trójkątnej dolnej `L` są wstawiane do tablicy `A` na miejsca zajmowane początkowo przez współczynniki macierzy `A`. Jest to możliwe, ponieważ macierz `L` ma bloki zerowe tam, gdzie macierz `A` ma bloki zerowe.

Wartością procedury jest `true`, jeśli obliczenie zakończyło się sukcesem, a `false` w przeciwnym razie. Przyczyną niepowodzenia może być brak dodatniej określoności macierzy `A`, lub tak złe jej uwarunkowanie, że skutki błędów zaokrągleń wyglądają, jakby macierz ta nie była dodatnio określona.

```

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 );

```

Powyższe procedury rozwiązują układy równań liniowych odpowiednio  $Lx = b$  i  $L^T x = b$ . Prawa strona układu i rozwiązanie to macierze o wymiarach  $n \times d$  (gdzie  $n = kr + s$ ). Procedury zastępują współczynniki prawej strony dane w tablicy  $x$  o podziałce  $xpitch$  przez współczynniki rozwiązania.

Aby rozwiązać układ równań z macierzą blokową  $A$ , należy najpierw dokonać jej rozkładu na czynniki (za pomocą procedury `pkn_Block1CholeskyDecompMf`), a następnie wywołać kolejno powyższe dwie procedury.

```

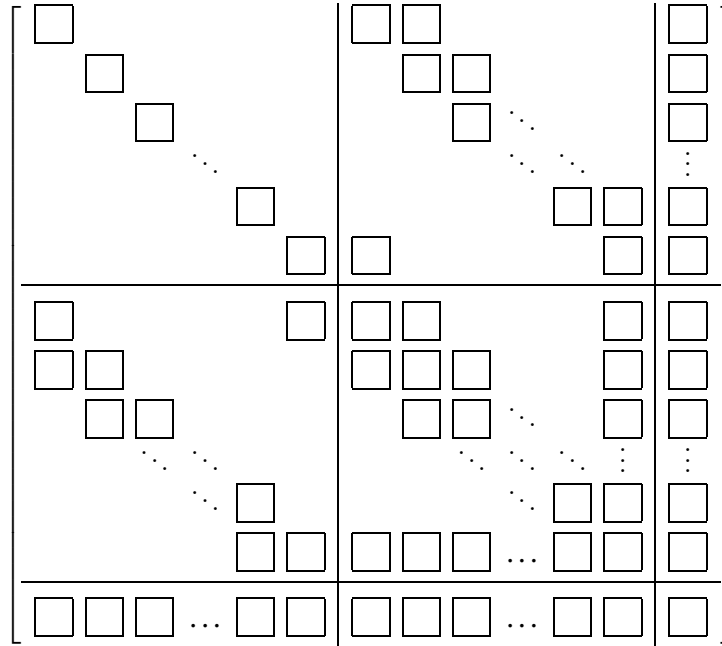
void pkn_Block1SymMatrixMultf ( int k, int r, int s,
                                float *A,
                                int spdimen, int xpitch, float *x,
                                int ypitch, float *y );

```

Procedura `pkn_Block1SymMatrixMultf` oblicza iloczyn macierzy:  $y = Ax$ , gdzie macierz  $A$  jest blokowa symetryczna  $n \times n$  (gdzie  $n = kr + s$ ), a macierz  $x$  (oraz  $y$ ) jest pełna, o wymiarach  $n \times d$ . Tablica  $x$  zawiera współczynniki macierzy  $x$ . Do tablicy  $y$  procedura wpisuje wynik. Podziałyki tych tablic (tj. odległości początków kolejnych wierszy) są równe odpowiednio  $xpitch$  i  $ypitch$ . Parametr `spdimen` ma wartość  $d$ .

### 3.5.2 Macierze o strukturze blokowej drugiego rodzaju

Struktura blokowa drugiego rodzaju obsługiwanego przez bibliotekę `libpcknum` jest przedstawiona na rysunku 3.9. Macierze takie są symetryczne i składają się z  $2k+1 \times 2k+1$  bloków, gdzie  $k \geq 3$ .



Rys. 3.9. Struktura blokowa drugiego rodzaju macierzy symetrycznej

Bloki niezerowe są rozmieszczone jak na rysunku, przy czym wiersze i kolumny bloków są numerowane od 0 do  $2k$ :

- Bloki  $A_{00}, \dots, A_{k-1,k-1}$  mają wymiary  $r \times r$ .
- Bloki  $A_{kk}, \dots, A_{2k-1,2k-1}$  mają wymiary  $s \times s$ .
- Blok  $A_{2k,2k}$  ma wymiary  $t \times t$ .

Jeśli macierz  $A$  jest dodatnio określona, to macierz trójkątna dolna  $L$ , taka że  $LL^T = A$ , ma zerowe bloki odpowiadające zerowym blokom macierzy  $A$ .

Cała macierz ma zatem wymiary  $k(r+s)+t \times k(r+s)+t$ . Do zapamiętania jej dolnego trójkąta potrzeba:

- $k \cdot \frac{1}{2}(r+1)r$  liczb dla bloków diagonalnych  $A_{00}, \dots, A_{k-1,k-1}$ ,
- $k \cdot \frac{1}{2}(s+1)s$  liczb dla bloków diagonalnych  $A_{kk}, \dots, A_{2k-1,2k-1}$ ,

- $\frac{1}{2}(t+1)t$  liczb dla bloku diagonalnego  $A_{2k,2k}$ ,
- $2k \cdot rs$  liczb dla bloków  $A_{k,k-1}, A_{k,0}, A_{k+1,k}, A_{k+1,k+1}, \dots, A_{2k-1,2k-2}, A_{2k-1,2k-1}$ ,
- $(2k-3) \cdot s^2$  liczb dla bloków  $A_{k+1,k}, \dots, A_{2k-2,2k-3}$  oraz  $A_{2k-1,k}, \dots, A_{2k-1,2k-2}$ .
- $k \cdot (r+s)t$  liczb dla bloków  $A_{2k,0}, \dots, A_{2k,2k-1}$ .

W sumie trzeba zarezerwować tablicę o długości

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

Obliczanie indeksów początków bloków

Mając dane indeksy  $i, j \in \{0, \dots, 2k\}$ , gdzie  $i \geq j$ , należy obliczyć indeks  $p$  pierwszego elementu bloku  $A_{ij}$  w tablicy.

1. Jeśli  $i = j < k$ , to  $p = i\frac{1}{2}(r+1)r$ .
2. Jeśli  $k \leq i = j < 2k$ , to  $p = k\frac{1}{2}(r+1)r + i\frac{1}{2}(s+1)s$ .
3. Jeśli  $i = j = 2k$ , to  $p = \frac{1}{2}k((r+1)r + (s+1)s)$ .
4. Niech  $N_1 = \frac{1}{2}(k(r+1)r + k(s+1)s + (t+1)t)$ .  
Jeśli  $k \leq i < 2k$ ,  $0 \leq j < k$  oraz  $i - j \bmod k \in \{0, 1\}$ , to  
 $p = N_1 + (2(i-k) + 1 - (i-j) \bmod k)rs$ .
5. Niech  $N_2 = N_1 + 2krs$ .  
Jeśli  $k < i < 2k-1$  oraz  $j = i-1$ , to  $p = N_2 + (i-k-1)s^2$ .
6. Jeśli  $i = 2k-1$  oraz  $k \leq j < 2k-2$ , to  $p = N_2 + (j-2)s^2$ .
7. Niech  $N_3 = N_2 + (2k-3)s^2$ .  
Jeśli  $i = 2k$  oraz  $j < k$ , to  $p = N_3 + jrt$ .
8. Jeśli  $i = 2k$  oraz  $k \leq j < 2k$ , to  $p = N_3 + krt + (j-k)st$ .
9. W przeciwnym razie blok  $A_{ij}$  jest blokiem zerowym, którego współczynniki nie są przechowywane w tablicy.

Dla bloków diagonalnych są przechowywane tylko elementy na i pod diagonalą. Elementy bloków poddiagonalnych są przechowywane wiersz po wierszu.

Rozkład trójkątny macierzy symetrycznej  $A$

Jeśli macierz  $L$  jest trójkątna dolna i składa się z bloków  $L_{i,j}$  (tj. dla  $i < j$  blok  $L_{i,j}$  jest zerowy), to macierz  $A = LL^T$  składa się z bloków

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

Bloki macierzy  $L$  można obliczyć przy użyciu następującego algorytmu:  
Kolejno dla  $i = 0, \dots, 2k$  oblicz (metodą Choleskiego) blok trójkątny dolny  $L_{i,i}$ , taki że  $L_{i,i} L_{i,i}^T = A_{i,i} - \sum_{l=0}^{i-1} L_{i,l} L_{l,i}^T$ , a następnie dla  $j = i+1, \dots, 2k$  oblicz blok  $L_{j,i} = (A_{j,i} - \sum_{l=0}^{i-1} L_{j,l} L_{l,i}^T) L_{i,i}^{-T}$ .

Dla macierzy  $A$  o rozpatrywanej strukturze wystarczy obliczyć

1. Dla  $i = 0, \dots, k-1$  macierz  $L_{i,i}$  taką że  $L_{i,i} L_{i,i}^T = A_{i,i}$ ,  
a następnie  $L_{j,i} = A_{j,i} L_{i,i}^{-T}$ , gdzie  $j \in \{i+k, i+(k+1) \bmod k, 2k\}$ .
2. Macierz  $L_{k,k}$  taką że  $L_{k,k} L_{k,k}^T = A_{k,k} - L_{k,0} L_{0,k}^T - L_{k,k-1} L_{k,k-1}^T$ ,  
a następnie  $L_{k+1,k}$ ,  $L_{2k-1,k}$  i  $L_{2k,k}$ .
3. Dla  $i = k+1, \dots, 2k-3$  macierz  $L_{i,i}$  taką, że  
 $L_{i,i} L_{i,i}^T = A_{i,i} - L_{i,i-k-1} L_{i,i-k-1}^T - L_{i,i-k} L_{i,i-k}^T - L_{i,i-1} L_{i,i-1}^T$ ,  
a następnie  $L_{i+1,i}$ ,  $L_{2k-1,i}$  i  $L_{2k,i}$ .
4. Macierz  $L_{2k-2,2k-2}$  taką że  
 $L_{2k-2,2k-2} L_{2k-2,2k-2}^T = A_{2k-2,2k-2} - L_{2k-2,k-3} L_{2k-2,k-3}^T - L_{2k-2,k-2} L_{2k-2,k-2}^T - L_{2k-2,2k-3} L_{2k-2,2k-3}^T$ ,  
a następnie  $L_{2k-1,2k-2} = (A_{2k-1,2k-2} - L_{2k-1,k-2} L_{2k-1,k-2}^T - L_{2k-1,2k-3} L_{2k-1,2k-3}^T) L_{2k-2,2k-2}^{-T}$   
i  $L_{2k,2k-2} = (A_{2k,2k-2} - L_{2k,k-2} L_{2k,k-2}^T - L_{2k,2k-3} L_{2k,2k-3}^T) L_{2k-2,2k-2}^{-T}$ .
5. Macierz  $L_{2k-1,2k-1}$  taką że  
 $L_{2k-1,2k-1} L_{2k-1,2k-1}^T = A_{2k-1,2k-1} - \sum_{l=k-2}^{2k-2} L_{2k-1,l} L_{2k-1,l}^T$ ,  
a następnie  $L_{2k,2k-1} = (A_{2k,2k-1} - \sum_{l=k-2}^{2k-2} L_{2k,l} L_{2k-1,l}^T) L_{2k-1,2k-1}^{-T}$ .
6. Macierz  $L_{2k,2k}$  taką że  $L_{2k,2k} L_{2k,2k}^T = A_{2k,2k} - \sum_{l=0}^{2k-1} L_{2k,l} L_{2k,l}^T$ .

Procedury

```
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 );
```









### 3.6 Nieregularne macierze rzadkie

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

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

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

```
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 );
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 );
```

```
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_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_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_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_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_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_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_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_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_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_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 );
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 );
```

```
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 );
```

```
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 );
```

```
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_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_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_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_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_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_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_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_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_SpSubMFindMTMnnzR ( int nra, int nca, int ncb,
                                int nnza, index3 *ai, int *apermut, int *acols,
                                int nnzb, index3 *bi, int *bpermut, int *brows,

```



```

        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 );

```

```

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 );

```

### 3.7 Rozwiązywanie równań nieliniowych

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

Procedura `pkn_SolveSqEqf` oblicza miejsca zerowe wielomianu  $x^2 + 2px + q$  o rzeczywistych współczynnikach. Parametry `p` i `q` mają wartości współczynników `p` i `q`. Parametry `x1` i `x2` służą do wyprowadzenia wyników.

Jeśli miejsca zerowe są *rzeczywiste*, to wartością procedury jest `true`. Procedura przypisuje zmiennej `*x1` wartość mniejszego, a zmiennej `*x2` większego miejsca zerowego.

Jeśli miejsca zerowe są *zespolone*, to wartością procedury jest `false`. Procedura przypisuje zmiennej `*x1` część rzeczywistą, a zmiennej `*x2` wartość bezwzględną części urojonej miejsc zerowych.

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

Procedura `pkn_Illinoisf` znajduje z dokładnością  $\varepsilon$  miejsce zerowe funkcji rzeczywistej `f` w przedziale `[a, b]`. Funkcja musi być ciągła w przedziale `[a, b]` i mieć na jego końcach przeciwne znaki. Jeśli funkcja `f` ma więcej niż jedno miejsce zerowe, to procedura obliczy jedno z nich. Metoda numeryczna realizowana przez procedurę dla funkcji gładkich o zerach jednokrotnych działa zwykle szybciej niż bisekcja.

Parametr `f` jest procedurą obliczającą wartość funkcji `f` dla podanego argumentu. Parametry `a` i `b` określają przedział `[a, b]`, w którym jest poszukiwane rozwiązanie. Parametr `eps` określa żadaną dokładność  $\varepsilon$  obliczenia rozwiązania (musi to być liczba dodatnia i nie powinna być mniejsza niż maksymalna graniczna dokładność, która zależy od błędów zaokrągleń w obliczaniu wartości funkcji `f`). Parametr `error` na wyjściu z procedury ma wartość `false` jeśli nie został wykryty błąd podczas obliczeń, albo `true` jeśli znak funkcji `f` na obu końcach przedziału `[a, b]` jest taki sam.

Obliczone miejsce zerowe jest zwracane jako wartość procedury.

## 3.8 Optymalizacja

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

Procedura `pkn_GoldenRatf` znajduje metodą złotych podziałów minimum funkcji rzeczywistej  $f$  jednej zmiennej w przedziale  $[a, b]$ . Parametry  $a, b$  określają końce tego przedziału, parametr `eps` określa żadaną dokładność (jego wartość musi być liczbą dodatnią), procedura `*f` ma obliczać wartość funkcji  $f$  w punkcie podanym jako parametr.

Parametr `*error` otrzymuje wartość `true`, jeśli procedura nie wykryje żadnego błędu w obliczeniach (obecnie nie jest w stanie żadnego błędu wykryć, ale po testach może to się przyda).

Wartość procedury jest znalezionym punktem minimum; jest to przybliżenie jakiegoś minimum lokalnego w przedziale  $[a, b]$ .



### 3.9 Obliczanie pochodnych funkcji złożonej

Procedury opisane w tym punkcie służą do obliczania pochodnych cząstkowych rzędu  $1, \dots, 4$  funkcji  $\mathbf{h}$ , będącej złożeniem funkcji  $\mathbf{f}: \mathbb{R}^2 \rightarrow \mathbb{R}^2$  i  $\mathbf{g}: \mathbb{R}^2 \rightarrow \mathbb{R}^d$  na podstawie pochodnych cząstkowych tych funkcji. Rzecz dotyczy więc funkcji dwóch zmiennych, ale zastosowane podejście może być użyte do funkcji, które mają inną niż 2 liczbę argumentów (ale jak dotąd nie miałem takiej potrzeby).

Niech  $\mathbf{f}(u, v) = [x(u, v), y(u, v)]^T$ . Wzory opisujące pochodne funkcji złożonej coraz wyższych rzędów można wyprowadzać rekurencyjnie, na podstawie wzorów dla pochodnych pierwszego rzędu:

$$\mathbf{h}_u = x_u \mathbf{g}_x + y_u \mathbf{g}_y,$$

$$\mathbf{h}_v = x_v \mathbf{g}_x + y_v \mathbf{g}_y,$$

i wzorów na pochodne iloczynu funkcji. Wychodzą stąd wzory

$$\mathbf{h}_{uu} = x_u^2 \mathbf{g}_{xx} + 2x_u y_u \mathbf{g}_{xy} + y_u^2 \mathbf{g}_{yy} + x_{uu} \mathbf{g}_x + y_{uu} \mathbf{g}_y,$$

$$\mathbf{h}_{uv} = x_u x_v \mathbf{g}_{xx} + (x_u y_v + x_v y_u) \mathbf{g}_{xy} + y_u y_v \mathbf{g}_{yy} + x_{uv} \mathbf{g}_x + y_{uv} \mathbf{g}_y,$$

$$\mathbf{h}_{vv} = x_v^2 \mathbf{g}_{xx} + 2x_v y_v \mathbf{g}_{xy} + y_v^2 \mathbf{g}_{yy} + x_{vv} \mathbf{g}_x + y_{vv} \mathbf{g}_y,$$

a następnie wzory na pochodne wyższych rzędów, które są coraz dłuższe. Cechą charakterystyczną wszystkich tych wzorów jest możliwość przedstawienia ich w postaci macierzowej, np.

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

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

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

$$\begin{bmatrix} \mathbf{h}_{uuuu} \\ \mathbf{h}_{uuuv} \\ \mathbf{h}_{uuvv} \\ \mathbf{h}_{uvvv} \\ \mathbf{h}_{vvvv} \end{bmatrix} = A_{41} \begin{bmatrix} \mathbf{g}_x \\ \mathbf{g}_y \end{bmatrix} + A_{42} \begin{bmatrix} \mathbf{g}_{xx} \\ \mathbf{g}_{xy} \\ \mathbf{g}_{yy} \end{bmatrix} + A_{43} \begin{bmatrix} \mathbf{g}_{xxx} \\ \mathbf{g}_{xxy} \\ \mathbf{g}_{xyy} \\ \mathbf{g}_{yyy} \end{bmatrix} + A_{44} \begin{bmatrix} \mathbf{g}_{xxxx} \\ \mathbf{g}_{xxxxy} \\ \mathbf{g}_{xxxyy} \\ \mathbf{g}_{xxyyy} \\ \mathbf{g}_{yyyyy} \end{bmatrix}. \quad (3.4)$$

Współczynniki macierzy  $A_{11}, \dots, A_{44}$  są wyrażeniami zależnymi od pochodnych cząstkowych funkcji  $x$  i  $y$ .

### 3.9.1 Obliczanie macierzy przekształceń pochodnych

```

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 xvuv, float yvuv,
    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 xuuvv, float yuuvv,
    float xuvvv, float yuvvv, float xuvvv, float yuvvv,
    float xvuvv, float yvuvv, 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 xvuv, float yvuv,
    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 );

```

Powyższe procedury obliczają macierze występujące we wzorach (3.1)–(3.4). Parametry typu float opisują wartości pochodnych funkcji  $x$  i  $y$ , np. wartość parametru  $x_u$  jest równa  $x_u$ , czyli  $\frac{\partial x}{\partial u}$ , wartość parametru  $y_{uuv}$  jest równa  $y_{uuv}$ , czyli  $\frac{\partial^3 y}{\partial^2 u \partial v}$  itd. Współczynniki macierzy są wpisywane do tablic wskazywanych przez parametry A11...A44.

### 3.9.2 Obliczanie pochodnych funkcji złożonej

```

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 xvuv, 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 xvuv, float yvvv,
    float xuuuu, float yuuuu, float xuuvv, float yuuvv,
    float xuvvv, float yuvvv, float xvuvv, 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 *gxxxxy, const float *gxxyy,
    const float *gxyyy, const float *gyyyy,
    float *huuuu, float *huuuv, float *huuvv,
    float *huvvv, float *hvvvv );

```

Powyższe procedury obliczają pochodne rzędu 1, ..., 4 funkcji  $h = f \circ g$  na podstawie pochodnych cząstkowych funkcji  $f: \mathbb{R}^2 \rightarrow \mathbb{R}^2$ , opisanej za pomocą dwóch

funkcji skalarnych,  $x(u, v)$  i  $y(u, v)$ , oraz funkcji  $\mathbf{g}: \mathbb{R}^2 \rightarrow \mathbb{R}^d$ .

Wymiar  $d$  przestrzeni, która jest przeciwdziedzina funkcji  $\mathbf{g}$  jest we wszystkich procedurach określany przez wartość parametru `spdimen`.

Nazwy pozostałych parametrów odpowiadają znaczeniu ich wartości. Na przykład wartość parametru `xu` jest równa wartości pochodnej funkcji  $x$  względem  $u$ ; podobnie, parametr `yuvv` ma wartość  $y_{uuvv} = \frac{\partial^4 y}{\partial^2 u \partial^2 v}$  itp.

Podobnie, parametr `gx` wskazuje tablicę  $d$  współrzędnych wektora  $\mathbf{g}_x = \frac{\partial \mathbf{g}}{\partial x}$ , zaś parametr `hu` jest wskaźnikiem tablicy, do której procedura wstawi  $d$  współrzędnych wektora  $\frac{\partial \mathbf{h}}{\partial u}$  itd.

Ponieważ do obliczenia pochodnych funkcji  $\mathbf{h}$  rzędu  $n$  potrzebne są tylko macierze  $A_{n1}, \dots, A_{nn}$  (zobacz wzory (3.1)–(3.4)), więc każda z powyższych procedur oblicza pochodne tylko jednego rzędu — odpowiednio 1, 2, 3 i 4. W obliczeniu tych pochodnych są istotne pochodne funkcji  $\mathbf{f}$  i  $\mathbf{g}$  rzędu  $1, \dots, n$ .

**Uwaga:** Jeśli funkcja  $\mathbf{f}$  jest przekształceniem afinicznym, to jej pochodne rzędu większego niż 1 są równe 0. W tym przypadku macierze  $A_{ij}$  dla  $j < i$  są macierzami zerowymi i lepiej (bo odrobinę szybciej) jest obliczać pochodne rzędu  $n$  funkcji  $\mathbf{h}$  obliczając macierz  $A_{nn}$  (za pomocą odpowiedniej procedury opisanej w p. 3.9.1), a następnie mnożąc ją przez macierz, której wiersze są odpowiednimi pochodnymi rzędnu  $n$  funkcji  $\mathbf{g}$ .

### 3.9.3 Obliczanie pochodnych złożenia z funkcją odwrotną

```
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 xvuv, 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 xvuv, float yvvv,
    float xuuuv, float yuuuv, float xuuvv,
    float yuuvv, float xuvvv, float yuvvv, 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 *gxxxxy, float *gxxxyy,
    float *gxxyyy, float *gyyyy );

```

Powyższe procedury obliczają pochodne cząstkowe funkcji  $\mathbf{g} = \mathbf{f}^{-1} \circ \mathbf{h}$ , która jest złożeniem funkcji  $\mathbf{f}^{-1}: \mathbb{R}^2 \rightarrow \mathbb{R}^2$  i funkcji  $\mathbf{h}: \mathbb{R}^2 \rightarrow \mathbb{R}^d$ . Funkcja  $\mathbf{f}$ , opisana przez dwie funkcje skalarne,  $x(u, v)$  i  $y(u, v)$ , musi być regularna (tj. jej pochodne cząstkowe pierwszego rzędu muszą być liniowo niezależne), przy czym funkcje  $\mathbf{f}$  i  $\mathbf{h}$  muszą być odpowiednio gładkie.

Parametr `spdimen` we wszystkich procedurach określa wymiar  $d$  przeciwdziedziny funkcji  $\mathbf{g}$  i  $\mathbf{h}$ . Pozostałe parametry mają nazwy opisujące niesione wartości. Na przykład parametr `yu` ma wartość pochodnej  $y_u = \frac{\partial y}{\partial u}$  itp. Podobnie, parametr `huv` jest wskaźnikiem tablicy zawierającej  $d$  współrzędnych wektora  $\mathbf{h}_{uv} = \frac{\partial^2 \mathbf{h}}{\partial u \partial v}$ , zaś parametr `gyyy` jest wskaźnikiem tablicy, do której procedura ma wstawić  $d$  współrzędnych wektora  $\mathbf{g}_{yyy} = \frac{\partial^3 \mathbf{g}}{\partial y^3}$ .

Metoda obliczenia polega na potraktowaniu wzorów (3.1)–(3.4)) jako układów

równań liniowych z niewiadomymi pochodnymi funkcji  $g$ . Równania te są rozwiązywane za pomocą procedury `pkn_multiGaussSolveLinEqf`, która realizuje metodę eliminacji Gaussa z pełnym wyborem elementu głównego. Ponieważ aby obliczyć pochodne rzędu  $n$  funkcji  $g$  trzeba wcześniej obliczyć pochodne rzędu niższego niż  $n$ , więc procedury mają parametry — wskaźniki tablic, do których wpisywane są wszystkie te pochodne (to jest różnica w porównaniu z procedurami opisanymi w poprzednim punkcie).

### 3.9.4 Obliczanie pochodnych funkcji odwrotnej

```
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 xvuv, float yvuv,
    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 xvuv, float yvuv,
    float xuuuv, float yuuuv, float xuuvv, float yuuvv,
    float xuvvv, float yuvvv, float xvuvv, float yvuvv,
    float *gx, float *gy, float *gxx, float *gxy, float *gyy,
    float *gxxx, float *gxxy, float *gxyy, float *gyyy,
    float *gxxxx, float *gxxxxy, float *gxxyy,
    float *gxxyy, float *gyyyy );
```

Powyższe procedury obliczają pochodne cząstkowe funkcji  $g = f^{-1}$ , gdzie  $f: \mathbb{R}^2 \rightarrow \mathbb{R}^2$  jest funkcją regularną i odpowiednio gładką, opisaną za pomocą dwóch funkcji skalarnych,  $x(u, v)$  i  $y(u, v)$ . Procedury te wywołują odpowiednio opisane







## 4. Biblioteka libpkgeom

W bibliotece libpkgeom są zebrane procedury realizujące podstawowe działania na punktach i wektorach w płaszczyźnie i przestrzeni trój- i czterowymiarowej. Działania te to dodawanie, odejmowanie, mnożenie i interpolacja, a także przekształcenia afiniczne. Ponadto w bibliotece tej jest procedura znajdowania otoczki wypukłej zbioru punktów na płaszczyźnie. Inne procedury z geometrii obliczeniowej też będą w tej bibliotece.

Wszystkie nazwy typów danych mają na końcu literę f albo d, co wskazuje na reprezentację współrzędnych — pojedynczej (float) albo podwójnej (double) precyzji.

### 4.1 Działania na punktach i wektorach

```
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;
```

Punkty i wektory reprezentuje się za pomocą par, trójek lub czwórek liczb. Istotny w tej reprezentacji jest brak wszelkich dodatkowych danych, dzięki czemu np. tablica  $n$  punktów na płaszczyźnie może być przekazywana jako parametr do procedury, która przetwarza tablicę  $2n$  liczb. Dlatego nie należy robić z tego klasy języka C++, ani tym bardziej definiować podklas z dodatkowymi atrybutami.

Struktura typu point3f może reprezentować punkt w przestrzeni trójwymiarowej albo punkt na płaszczyźnie. W tym ostatnim przypadku pola  $x$ ,  $y$ ,  $z$  opisują współrzędne jednorodnego tego punktu — jego współrzędne kartezjańskie są równe  $x/z$  i  $y/z$ . Podobnie, struktura typu point4f składa się z pól opisujących współrzędne jednorodnego punktu w przestrzeni trójwymiarowej.

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

Struktura typu ray3f służy do reprezentowania promieni, tj. półprostych o początku w punkcie p i o kierunku określonym przez wektor 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;
```

Struktury typu trans2f i trans3f reprezentują przekształcenia afiniczne przestrzeni dwu- i trójwymiarowej. Reprezentacja składa się z macierzy  $3 \times 3$  albo  $4 \times 4$ , której ostatni wiersz ma zawsze postać  $[0, 0, 1]$  albo  $[0, 0, 0, 1]$ , w związku z czym nie jest pamiętany. Pole detsgn ma wartość znaku wyznacznika macierzy.

```
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 )
```

Powyższe procedury i makra służą do inicjalizacji punktów i wektorów.

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

Powyższe procedury obliczają obraz  $q$  punktu  $p$  w przekształceniu afinicznym odpowiednio przestrzeni dwu- i trójwymiarowej.

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

Powyższe procedury obliczają obraz  $w$  wektora  $v$  w przekształceniu liniowym, które jest częścią liniową przekształcenia afinicznego reprezentowanego przez parametr  $*tr$ .

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

Procedura `TransContra3f` oblicza obraz  $w$  wektora  $v$  w przekształceniu liniowym, którego macierz jest transpozycją macierzy części liniowej przekształcenia afinicznego reprezentowanego przez parametr  $*tri$ . Jeśli wektor  $v$  jest wektorem normalnym pewnej płaszczyzny  $\pi$ , a przekształcenie reprezentowane przez parametr  $*tri$  jest *odwrotnością* pewnego przekształcenia  $A$ , to obliczony wektor  $w$  jest wektorem normalnym obrazu płaszczyzny  $\pi$  w przekształceniu  $A$ .

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

Procedura `Trans3Point2f` poddaje przekształceniu afinicznemu  $*tr$  punkt  $p \in \mathbb{R}^3$ , którego początkowe dwie współrzędne są wartościami pól  $x$  i  $y$  parametru  $*p$ , a trzecia jest równa 0. Współrzędne  $x$  i  $y$  obrazu są przypisywane odpowiednim polom parametru  $*q$ .

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

Procedura `Trans2Point3f` poddaje przekształceniu afinicznemu punkt  $p \in \mathbb{R}^2$ , reprezentowany za pomocą współrzędnych jednorodnych.

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

Procedura `Trans3Point4f` poddaje przekształceniu afinicznemu  $*tr$  punkt  $p \in \mathbb{R}^3$ , którego cztery współrzędne jednorodne są wartościami odpowiednich pól parametru  $*p$ .

Współrzędne jednorodne obrazu (takie że współrzędna wagowa punktu i obrazu jest taka sama) są przypisywane odpowiednim polom parametru  $*q$ .

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

Procedury IdentTrans2f i IdentTrans3f dokonują inicjalizacji struktury \*tr, po której reprezentuje ona przekształcenie tożsamościowe przestrzeni dwu- albo trójwymiarowej.

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

Procedury CompTrans2f i CompTrans3f wyznaczają złożenie przekształceń afinicznych reprezentowanych przez parametry \*t i \*u i przypisują je parametrowi \*s. Przekształcenie to jest równoważne wykonaniu najpierw przekształcenia \*u, a następnie \*t.

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

Procedura GeneralAffineTrans3f oblicza złożenie przekształcenia reprezentowanego przez parametr \*tr z przekształceniem, którego część liniowa jest reprezentowana przez macierz  $[v_1, v_2, v_3]$  (a wektor przesunięcia jest zerowy). Obliczone złożenie jest przypisywane parametrowi \*tr.

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

Procedury ShiftTrans2f i ShiftTrans3f wyznaczają złożenie przekształcenia reprezentowanego przez parametr \*tr i przesunięcia o wektor  $[t_x, t_y]^T$  albo  $[t_x, t_y, t_z]^T$ . Obliczone złożenie jest przypisywane parametrowi \*tr.

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

Procedura RotTrans2f oblicza złożenie przekształcenia płaszczyzny reprezentowanego przez parametr \*tr z obrotem wokół punktu  $[0, 0]^T$  o kąt angle. Obliczone złożenie jest przypisywane parametrowi \*tr.

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

Procedura Rot3f oblicza złożenie przekształcenia reprezentowanego przez parametr \*tr z obrotem w jednej z płaszczyzn układu. Płaszczyzna ta jest określona przez parametry j i k, które muszą mieć różne wartości ze zbioru  $\{1, 2, 3\}$ . Na przykład obrotowi w płaszczyźnie xy (wokół osi z) odpowiada  $j = 1, k = 2$ . Kąt obrotu jest równy angle. Obliczone złożenie przekształceń jest przypisywane parametrowi \*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 )
```

Powyższe trzy makra wywołują procedurę Rot3f w celu złożenia przekształcenia reprezentowanego przez parametr \*tr z obrotem wokół osi x, y, z, czyli odpowiednio w płaszczyznach yz, zx i xy.

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

Procedura RotVTrans3f oblicza złożenie przekształcenia afinicznego reprezentowanego przez początkową wartość parametru \*tr z obrotem wokół prostej przechodzącej przez punkt  $[0,0,0]^T$  o kierunku *jednostkowego* wektora  $v$  o kąt angle. Obliczone złożenie przekształceń jest przypisywane parametrowi \*tr.

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

Procedura FindRotVEulerf oblicza kąty Eulera (precesji \*psi, nutacji \*theta i obrotu właściwego \*phi) reprezentujące obrót wokół prostej o kierunku *jednostkowego* wektora  $v$  o kąt angle.

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

Wartością procedury TrimAnglef jest liczba  $\alpha$ , która należy do przedziału  $[-\pi, \pi]$  i która różni się od wartości parametru angle o całkowitą wielokrotność liczby  $2\pi$  i o błąd zaokrąglenia.

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

Procedura CompEulerRotf oblicza kąty Eulera  $\psi$ ,  $\theta$ ,  $\varphi$  obrotu, który jest złożeniem dwóch obrotów reprezentowanych odpowiednio przez kąty Eulera  $\psi_1$ ,  $\theta_1$ ,  $\varphi_1$  oraz  $\psi_2$ ,  $\theta_2$ ,  $\varphi_2$ .

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

Procedura CompRotV3f oblicza złożenie dwóch obrotów w  $\mathbb{R}^3$ , danych za pomocą wektorów jednostkowych osi obrotu,  $v_1$ ,  $v_2$  i kątów  $\alpha_1$ ,  $\alpha_2$ . Wyznaczany jest wektor  $v$  osi złożenia i kąt  $\alpha$ .

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

Procedura EulerRotTrans3f wyznacza złożenie przekształcenia afinicznego reprezentowanego przez początkową wartość parametru \*tr i obrotu reprezentowanego przez kąty Eulera  $\psi$ ,  $\theta$ ,  $\varphi$ .

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

Procedury ScaleTrans2f i ScaleTrans3f wyznaczają złożenie przekształcenia afinicznego reprezentowanego przez początkową wartość parametru \*tr i skalowania o współczynniki odpowiednio  $s_x$  i  $s_y$  albo  $s_x$ ,  $s_y$  i  $s_z$ .

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

Procedura MirrorTrans3f wyznacza złożenie przekształcenia afinicznego reprezentowanego przez początkową wartość parametru \*tr i odbicia symetrycznego względem płaszczyzny zawierającej początek układu współrzędnych, której wektorem normalnym jest wektor  $\mathbf{n}$ .

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

Procedury InvertTrans2f i InvertTrans3f wyznaczają przekształcenie odwrotne do przekształcenia afinicznego reprezentowanego przez początkową wartość parametru \*tr, jeśli istnieje. Wartością procedury jest wtedy true, a w przeciwnym razie 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 );
```

Powyższe procedury obliczają wektor  $\mathbf{w} = a\mathbf{v}$ .

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

Powyższe procedury obliczają punkt  $\mathbf{q} = \mathbf{p} + \mathbf{v}$ .

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

Powyższe procedury obliczają punkt  $\mathbf{q} = \mathbf{p} + t\mathbf{v}$ .

```
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 );
```

Powyższe procedury obliczają wektor  $\mathbf{v} = \mathbf{p}_1 - \mathbf{p}_2$ .

```

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 );

```

Powyższe procedury obliczają punkt  $q = p_1 + t(p_2 - p_1)$ .

```

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 );

```

Powyższe procedury wyznaczają punkt  $q = \frac{1}{2}(p_1 + p_2)$ .

```

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 );

```

Powyższe procedury obliczają kombinację liniową trzech wektorów danych jako parametry; współczynniki kombinacji są podane w tablicy coeff.

```

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

```

Powyższe procedury obliczają  $*v := \frac{1}{\|v\|_2}v$ .

```

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

```

Powyższe procedury obliczają odpowiednie iloczyny skalarne.

```
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 );
```

Powyższe procedury obliczają wyznaczniki macierzy o wymiarach odpowiednio  $2 \times 2$ ,  $3 \times 3$  i  $4 \times 4$ , których kolumny są wektorami podanymi jako parametry.

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

Powyższe procedury obliczają współrzędne kartezjańskie punktu  $\mathbf{p}$  na podstawie współrzędnych jednorodnych.

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

Powyższe procedury obliczają współrzędne jednorodne punktu  $\mathbf{p}$  z wagą  $w$  na podstawie współrzędnych kartezjańskich.

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

Procedura CrossProduct3f oblicza iloczyn wektorowy wektorów  $\mathbf{v}_1$  i  $\mathbf{v}_2$ .

```
void OrtVector2f ( const vector2f *v1, const vector2f *v2,
                  vector2f *v );
void OrtVector3f ( const vector3f *v1, const vector3f *v2,
                  vector3f *v );
void OrtVector4f ( const vector4f *v1, const vector4f *v2,
                  vector4f *v );
```

Procedury OrtVector2f, OrtVector3f i OrtVector4f obliczają wektor  $\mathbf{v} = \mathbf{v}_2 - \frac{\langle \mathbf{v}_1, \mathbf{v}_2 \rangle}{\langle \mathbf{v}_1, \mathbf{v}_1 \rangle} \mathbf{v}_1$ .

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

Procedura CrossProduct4P3f oblicza pierwsze trzy współrzędne wektora, który jest iloczynem wektorowym w  $\mathbb{R}^4$  wektorów  $\mathbf{v}_1$ ,  $\mathbf{v}_2$  i  $\mathbf{v}_3$ .

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

Procedura OutProduct4P3f oblicza wektor

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



```
void ProjectPointOnLine2f ( const point2f *p0, const point2f *p1,
                           point2f *q );
void ProjectPointOnLine3f ( const point3f *p0, const point3f *p1,
                           point3f *q );
void ProjectPointOnLine4f ( const point4f *p0, const point4f *p1,
                           point4f *q );
```

```
void ProjectPointOnPlane3f ( const point3f *p0, const point3f *p1,
                            const point3f *p2, point3f *q );
void ProjectPointOnPlane4f ( const point4f *p0, const point4f *p1,
                            const point4f *p2, point4f *q );
```

## 4.2 Boksy

Prostokąty i prostopadłościany przydają się w wielu zastosowaniach, zwłaszcza w szacowaniu położenia bardziej skomplikowanych figur geometrycznych. Zdefiniowane niżej typy opisują takie boksy. W przyszłości podstawowe procedury wykonujące działania na boksach mają być częścią biblioteki libpkggeom.

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

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

## 4.3 Znajdowanie otoczki wypukłej

Nagłówki procedur znajdowania otoczki wypukłej (w wersji dla pojedynczej i podwójnej precyzji) znajdują się w pliku convh.h.

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

Procedura FindConvexHull2f znajduje otoczkę wypukłą zbioru  $n$  punktów na płaszczyźnie. Na wejściu punkty te są podane w tablicy  $p$ . Początkowa wartość parametru  $*n$  jest liczbą tych punktów. Końcowa zawartość tej tablicy to niektóre z tych punktów, mianowicie kolejne wierzchołki wielokąta, który jest otoczką zbioru punktów danych. Liczba wierzchołków otoczki jest końcową wartością parametru  $*n$ .



## 5. Biblioteka libcamera

Biblioteka libcamera zawiera procedury obsługujące kamery, czyli obiekty, które służą do odwzorowania (rzutowania) punktów przestrzeni trójwymiarowej na płaszczyznę, oczywiście w celu wykonywania obrazków. Są dwa rodzaje rzutów: perspektywiczne i równoległe. Te pierwsze służą do wykonywania „fotografii”, a te drugie do „rysunków technicznych”.

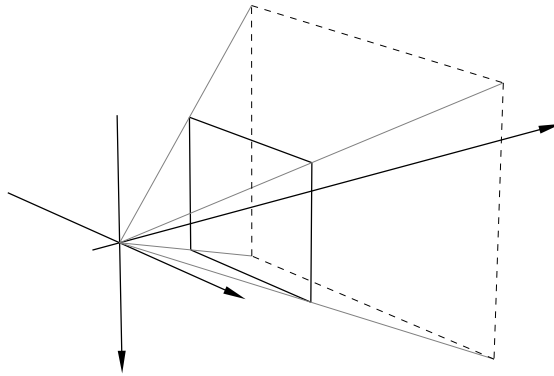
### 5.1 Kamera

#### 5.1.1 Opis kamery i algorytm rzutowania

Struktura danych i nagłówki procedur obsługujących rzutowanie są opisane w plikach cameraf.h i camerad.h. Oba pliki można włączyć do programu za pośrednictwem pliku 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;
```

Struktura `CameraRecf` opisuje kamerę, tj. obiekt, który określa rzutowanie perspektywiczne lub równoległe na płaszczyznę.



Rys. 5.1. Układ współrzędnych kamery i klatka dla rzutu perspektywicznego

Pola struktury mają następujące znaczenie:

`parallel` — jeśli pole to ma wartość `false` (0), to rzutowanie jest perspektywiczne, a w przeciwnym razie równoległe.

`upside` — jeśli pole to ma wartość `false`, to oś `y` układu współrzędnych na obrazie jest zorientowana do dołu (tak jak w oknach systemu XWindow), a w przeciwnym razie — do góry (tak jak w bibliotece OpenGL i w domyślnym układzie w języku PostScript).

`c_fixed` — parametr określający zmiany środka obrotu kamery przy przesuwaniu. Wartość `false` oznacza, że punkt ten jest nieruchomy w układzie globalnym, zaś `true` oznacza, że jest nieruchomy w układzie kamery.

`magnification` — domyślnie pole to ma wartość 1, co oznacza, że jednostki osi w układzie klatki są równe szerokości i wysokości piksela. Jeśli pole to ma większą wartość, to jednostki są o ten czynnik krótsze, co może być użyteczne, jeśli obraz ma być wykonany z nadpróbkowaniem.

`xmin`, `ymin`, `width`, `height` — współrzędne górnego lewego rogu klatki oraz jej szerokość i wysokość w pikselach.

`aspect` — współczynnik aspekt rastra, tj. iloraz szerokości i wysokości piksela.

`position` — położenie obserwatora (współrzędne w układzie globalnym).

`psi`, `theta`, `phi` — kąty Eulera  $\psi$ ,  $\vartheta$ ,  $\varphi$  określające położenie kamery (tj. kierunek).

`g_centre`, `c_centre` — współrzędne środka obrotu kamery, tj. punktu, który leży na osi obrotu kamery podczas obracania jej.

`xscale`, `yscale` — współczynniki skalowania osi `x` i `y` układu kamery.

`Ctr`, `CTrInv` — przekształcenia opisujące przejścia od układu globalnego do układu kamery i odwrotne.

`cplane` — reprezentacje czterech półprzestrzeni, których przecięciem jest ostrosłup widzenia. Półprzestrzeń  $ax + by + cz + d > 0$  jest reprezentowana przez wektor o współrzędnych  $a$ ,  $b$ ,  $c$ ,  $d$ .

Na razie są oprogramowane cztery półprzestrzenie, docelowo ma być 6.

`vd` — unia, która zawiera dane specyficzne dla rodzaju rzutowania. Struktura `vd.persp` zawiera dane dla rzutów perspektywicznych, `vd.para` dla równoległych.

`vd.persp.f` — długość ogniskowej kamery, mierzona w jednostkach, w których przekątna klatki ma długość 1. Długość 1 ogniskowej odpowiada obiektywowi standardowemu.

`vd.persp.xi0`, `vd.persp.eta0` — przesunięcie pikseli po rzutowaniu perspektywicznym.

`vd.persp.dxi0`, `vd.persp.deta0` — współrzędne  $x$ ,  $y$  (w układzie kamery) środka klatki. Domyślnie mają wartość 0, co oznacza, że środek klatki leży na „osi optycznej” kamery. Nadanie innej wartości jest potrzebne w programie obsługującym parę kamer w celu tworzenia obrazów stereoskopowych.

`vd.para.wdt`, `vd.para.hgh`, `vd.para.diag` — wymiary (szerokość, wysokość i długość przekątnej) klatki mierzona w jednostkach układu globalnego.

`vd.para.dim_case` — parametr określający, który z powyższych trzech wymiarów klatki jest podany przez użytkownika; 0 — długość przekątnej, 1 — szerokość, 2 — wysokość. Na podstawie jednego z tych wymiarów podczas obliczania macierzy przekształcenia zostaną obliczone pozostałe dwa.

Domyślnie pole to otrzymuje wartość 0, a pole `vd.para.diag` wartość 1.

Algorytm rzutowania:

Obraz punktu  $p$  reprezentowanego za pomocą współrzędnych w układzie globalnym jest obliczany tak:

1. Punkt  $p$  jest poddawany przekształceniu afinicznemu reprezentowanemu przez pole `CTr`. Krok ten jest przejściem do układu współrzędnych kamery.
2. Jeśli rzutowanie jest perspektywiczne, to obliczone współrzędne  $x$  i  $y$  w układzie kamery są dzielone przez współrzędną  $z$ , co jest właściwym rzutowaniem perspektywicznym, a następnie obliczane są współrzędne  $x$ ,  $y$  punktu na obrazie, przez dodanie wartości pól `xi0` i `eta0`.

Jeśli rzutowanie jest równoległe, to ten krok obliczeń jest pomijany.

3. Jeśli oś  $y$  na obrazie jest zorientowana do góry (pole upside ma wartość niezerową), to współrzędna  $y$  jest zastępowana przez  $2y_{\min} + h - y$ , gdzie  $y_{\min}$  jest wartością pola `ymin`, a  $h$  jest wartością pola `height`.

Określanie przejścia do układu kamery:

Przekształcenie, które opisuje przejście od globalnego układu współrzędnych do układu kamery (wykonywane w pierwszym kroku rzutowania opisanego wyżej) jest złożeniem trzech przekształceń afinicznych:

1. Skalowania osi  $x$  i  $y$  o czynniki będące wartościami odpowiednio pól `xscale` i `yscale`.
2. Obrotu opisanego za pomocą kątów Eulera, będących wartościami pól `psi`, `theta`, `phi`.
3. Przesunięcia, które umieszcza początek układu w punkcie określonym przez pole `position`.

Nadawanie wartości powyższych pól powinno odbywać się w zasadzie wyłącznie za pomocą opisanych dalej procedur, które w szczególności obliczają położenie kamery jako wynik ciągu jej przemieszczeń od domyślnego położenia początkowego.

### 5.1.2 Procedury obsługi kamery

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

Procedura `CameraInitFramef` inicjalizuje w strukturze `*CPos` pola opisujące sposób rzutowania i wielkość klatki kamery (w pikselach) oraz aspekt rastra (iloraz szerokości i wysokości piksela).

Parametr `parallel` o wartości `false` określa rzutowanie perspektywiczne, a wartość `true` spowoduje określenie rzutowania równoległego.

Parametr `upside` o wartości `false` spowoduje przyjęcie na obrazie układu współrzędnych z osią  $y$  skierowaną do dołu, a `true` — do góry.

Parametry `width` i `height` opisują szerokość i wysokość klatki, zaś parametry `xmin` i `ymin` położenie górnego lewego rogu klatki.

Wywołanie tej procedury powinno poprzedzić wszystkie dalsze akcje z użyciem kamery, ale *nie wystarczy* do określenia rzutowania. To powinno być zrobione przez wywołanie `CameraInitPosf` i ewentualnie pewnej liczby wywołań procedur zmieniających położenie kamery. Jeśli trzeba zmienić wielkość klatki (np. w celu dostosowania jej do nowych wymiarów okna zmienionego przez użytkownika programu na ekranie), bez zmieniania położenia obserwatora, to po wywołaniu `CameraInitFramef` należy wywołać procedurę `CameraSetMappingf`.

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

Procedura `CameraSetMagf` służy do określenia powiększenia rozdzielczości rastera; domyślnie jednostkami osi układu kamery są szerokość i wysokość jednego piksela. Wywołując tę procedurę z parametrem `mag = n` (gdzie `n` jest całkowitą liczbą dodatnią) zmniejszamy te jednostki `n` razy, co może się przydać podczas tworzenia obrazu z nadpróbkowaniem (ang. *supersampling*).

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

Procedura `CameraSetMappingf` ma na celu obliczenie macierzy wykorzystywanych przez procedury rzutowania. Procedura ta jest wywoływana przez wszystkie procedury nadające lub zmieniające położenie kamery, tak więc jej wywoływanie przez aplikacje jest zbędne, z wyjątkiem sytuacji, gdy wielkość klatki została zmieniona (za pomocą `CameraInitFramef`) i określone wcześniej położenie kamery ma pozostać niezmienione.

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

Procedura `CameraProjectPoint3f` służy do obliczenia obrazu punktu `*p` w rzucie perspektywicznym lub równoległym. Współrzędne tego obrazu to współrzędne `x` i `y` punktu `*q`. Jego współrzędna `z` określa głębokość punktu, tj. jego odległość (ze znakiem) od płaszczyzny równoległej do rzutni, zawierającej położenie obserwatora. Może ona się przydać do rozstrzygania widoczności w algorytmach linii lub powierzchni zasłoniętej.

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

Procedura `CameraUnProjectPoint3f` oblicza przeciwobraz punktu `p` w rzucie. Współrzędne `x`, `y` punktu `*q` są podane w układzie obrazu, współrzędna `z` jest głębokością (w układzie kamery). Jest to więc przekształcenie odwrotne do przekształcenia dokonywanego przez procedurę `CameraProjectPoint3f`.

Współrzędne `x`, `y`, `z` przeciwobrazu są przypisywane do odpowiednich pól parametru `*q`.

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

Procedura `CameraProjectPoint2f` dokonuje rzutowania punktu `p`, którego współrzędne `x`, `y` są dane za pomocą parametru `p`, a współrzędna `z` jest równa 0.

Współrzędne `x`, `y` rzutu są przypisywane polom parametru `*q`.

W zasadzie stosowanie tej procedury ma sens tylko dla rzutowania równoległego.

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

Procedura `CameraUnProjectPoint2f` znajduje przeciwobraz punktu  $p$ , którego współrzędne  $x$ ,  $y$  (w układzie obrazu) są dane za pomocą parametru  $p$ , a współrzędna  $z$  (w układzie kamery) jest równa 0.

Współrzędne przeciwobrazu są przypisywane polom parametru  $*q$ .

Procedurę tę można stosować tylko dla rzutowania równoległego.

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

Procedura `CameraRayOfPixel` dla punktu o współrzędnych na obrazie  $x = xi$ ,  $y = eta$ , znajduje reprezentację promienia, tj. półprostej, której początek (dla rzutowania perspektywicznego) jest położeniem obserwatora, i która przechodzi przez podany punkt rzutni. Dla rzutowania równoległego początek promienia jest punktem rzutni, a jego kierunek jest kierunkiem rzutowania.

Pole  $p$  struktury  $*ray$  otrzymuje wartość opisującą początek promienia (we współrzędnych w układzie globalnym), a przypisana przez procedurę wartość pola  $v$  to jednostkowy wektor kierunkowy promienia. Procedura jest napisana na głównie potrzeby śledzenia promieni.

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

Procedura `CameraInitPosf` przypisuje kamerze domyślne położenie, w którym osie  $x$ ,  $y$  i  $z$  układu kamery pokrywają się z osiami  $x$ ,  $y$ ,  $z$  układu globalnego. Długość ogniskowej kamery otrzymuje wartość 1. Przed wywołaniem tej procedury należy określić wielkość klatki i aspekt obrazu, za pomocą procedury `CameraInitFramef`.

Po wywołaniu procedury `CameraInitPosf` kamera jest gotowa do rzutowania punktów, a także do zmieniania położenia i ogniskowej.

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

Procedura `CameraSetRotCentref` służy do określania punktu, przez który przechodzą osie następnie wykonywanych obrotów kamery. Parametr  $*centre$  określa ten punkt, parametr `global_coord` określa, czy współrzędne tego punktu są podane w układzie globalnym (jeśli ma wartość `true`), czy w układzie kamery (jeśli `false`). Parametr `global_fixed` określa, czy przy przesunięciach kamery punkt ten jest nieruchomy w układzie globalnym (jeśli `true`), czy w układzie kamery (jeśli `false`).

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

Procedura `CameraMoveToGf` przesuwa (bez obrotu) kamerę do położenia  $*pos$ , wyspecyfikowanego w układzie globalnym.



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

Procedura CameraTurnGf nadaje kamerze położenie kątowe określone przez podanie kątów Eulera (precesji, psi, nutacji, theta, i obrotu właściwego, phi), w globalnym układzie współrzędnych.

**Uwaga:** Przewiduję zmianę sposobu określania położenia kąтового kamery, w związku z czym bezpośrednio używanie tej procedury jest *niewskazane*.

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

Procedura CameraMoveGf poddaje kamerę przesunięciu o wektor v, podany w globalnym układzie współrzędnych.

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

Procedura CameraMoveGf poddaje kamerę przesunięciu o wektor v, podany w układzie współrzędnych kamery.

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

Procedura CameraRotGf dokonuje obrotu kamery. Obrót jest określony za pomocą kątów Eulera w układzie globalnym. Oś obrotu przechodzi przez początek globalnego układu współrzędnych lub przez punkt określony za pomocą procedury SetCameraRotCentref.

```
#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)
```

Trzy makra dokonują obrotów kamery wokół trzech osi układu globalnego.

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

Procedura CameraRotVGf dokonuje obrotu kamery wokół osi o kierunku wektora v i kąt angle. Współrzędne wektora v są dane w globalnym układzie współrzędnych.

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

Procedura CameraRotCf dokonuje obrotu kamery. Obrót jest określony za pomocą kątów Eulera w układzie kamery. Oś obrotu przechodzi przez początek *globalnego* układu współrzędnych lub przez punkt określony za pomocą procedury SetCameraRotCentref.

```
#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 )
```

Trzy makra dokonują obrotów kamery wokół trzech osi układu kamery.

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

Procedura CameraRotVCf dokonuje obrotu kamery wokół osi o kierunku wektora  $v$  i kąt  $angle$ . Współrzędne wektora  $v$  są dane w układzie współrzędnych kamery.

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

Procedura CameraSetFf ustawia „długość ogniskowej” kamery.

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

Procedura CameraZoomf zmienia „długość ogniskowej” kamery, mnożąc długość dotychczasową przez parametr  $fchange$ , który musi być dodatni.

```
boolean CameraClipPoint3f ( CameraRecf *CPos,
                           point3f *p, point3f *q );
```

Procedura CameraClipPoint3f sprawdza, czy obraz punktu  $p$  mieści się w klatce i jeśli tak, to dokonuje rzutowania tego punktu. Współrzędne obrazu punktu są przekazywane za pomocą parametru  $q$ . Wartość `true` oznacza, że obraz punktu jest widoczny, `false`, że nie.

```
boolean CameraClipLine3f ( CameraRecf *CPos,
                          point3f *p0, float t0, point3f *p1, float t1,
                          point3f *q0, point3f *q1 );
```

Procedura CameraClipLine3f obcina do ostrosłupa widoczności odcinek  $\{(1-t)p_0 + tp_1 : t \in [t_0, t_1]\}$ . Jeśli przecięcie odcinka z ostrosłupem widoczności jest niepuste, to końce tej części są rzutowane i wyprowadzane za pomocą parametrów  $q_0$  i  $q_1$ . Wartością procedury jest wtedy `true`.

Procedura jest implementacją algorytmu Lianga-Barsky’ego.

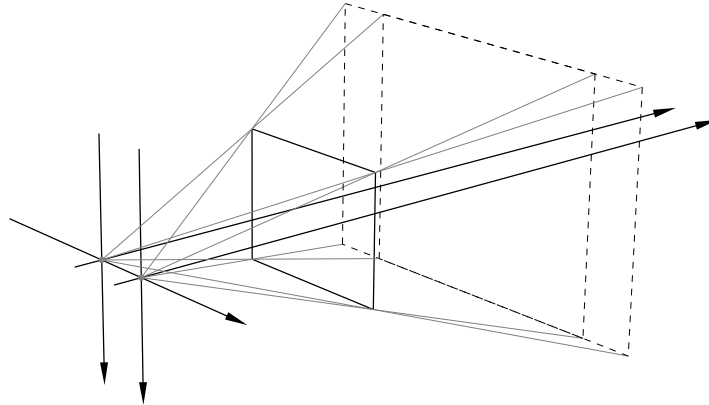
```
boolean CameraClipPolygon3f ( CameraRecf *CPos,
                             int n, const point3f *p,
                             void (*output)(int n, point3f *p) );
```

Procedura CameraClipPolygon3f dokonuje obcinania wielokąta do ostrosłupa widzenia, przy użyciu algorytmu Sutherlanda-Hodgmana. Parametr  $n$  określa liczbę wierzchołków wielokąta w przestrzeni, podanych w tablicy  $p$ ; brzeg tego wielokąta jest jedną łamaną zamkniętą.

Parametr `output` wskazuje procedurę, która zostanie wywołana, jeśli część wspólna wielokąta z ostrosłupem widzenia jest niepusta. Parametr `n` tej procedury określa liczbę wierzchołków obciętego wielokąta. W tablicy `p` są podane rzuty tych wierzchołków.

## 5.2 Para kamer do obrazów stereoskopowych

Aby wykonać parę obrazów stereoskopowych trzeba umieścić w przestrzeni dwie kamery, a następnie wykonać obrazy, dokonując rzutowania za pomocą każdej z nich. Opisane niżej procedury ułatwiają manipulowanie taką parą kamer; każda z nich odpowiada pewnej procedurze manipulacji jedną kamerą. Procedur tych używa się *zamiast* opisanych wyżej procedur ustawiania kamer w przestrzeni. Natomiast podczas tworzenia obrazów będą używane procedury rzutowania lub generowania promieni, dla każdej kamery niezależnie od drugiej.



Rys. 5.2. Para kamer stereo i ich wspólna klatka

Struktura danych i nagłówki procedur obsługujących parę kamer stereoskopowych są opisane w pliku nagłówkowym `stereo.h`.

```
typedef struct StereoRecf {
    point3f    position;
    float      d;
    float      l;
    CameraRecf left, right;
    trans3f    STr, STrInv;
} StereoRecf;
```

Struktura typu `StereoRecf` zawiera opisy dwóch kamer (lewej i prawej).

```
void StereoInitFramef ( StereoRecf *Stereo, boolean upside,
                        short width, short height, short xmin, short ymin,
                        float aspect );
```

Procedura `StereoInitFramef` służy do określenia wymiarów klatek kamer w pikselach i współczynnika `aspect`. W tym celu procedura wywołuje dla każdej z kamer procedurę `CameraInitFramef` z tymi samymi parametrami. Procedurę tę

należy wywołać jako pierwszą w procesie inicjalizacji pary kamer. Po jej wywołaniu jeszcze procedura *nie jest* gotowa do użycia.

Parametr *upside* określa zwrot osi *y* w układzie współrzędnych obrazu — zobacz opis procedury `CameraInitFramef`.

```
void StereoSetDimf ( StereoRecf *Stereo,
                    float f, float d, float l );
```

Procedura `StereoSetDimf` służy do określenia wymiarów pary kamer w jednostkach długości związanych z podstawowym układem współrzędnych, w którym opisuje się obiekty do narysowania. Parametr *f* określa bezwymiarową długość ogniskowej, tj. iloraz odległości płaszczyzny klatki od środka rzutowania i przekątnej klatki. Parametr *d* określa odległość między źrenicami oczu obserwatora (tj. między środkami rzutowania kamer), a parametr *l* określa odległość środków rzutowania od płaszczyzny klatki (tj. ekranu). Przekątna klatki ma zatem długość  $l/f$  w jednostkach związanych ze stosowanym układem. Długość przekątnej klatki w calach zależy od monitora.

**Uwaga:** Procedura `StereoInitPosf` przypisuje atrybuty  $f = 1$ ,  $d = 0$  i  $l = 1$ , które nie są zbyt użyteczne. Trzeba im zatem nadać bardziej sensowne wartości, właśnie przez wywołanie procedury `StereoSetDimf`.

```
void StereoSetMagf ( StereoRecf *Stereo, char mag );
```

Procedura `SetStereoMagf` określa współczynnik powiększenia rozdzielczości kamer (np. dla celów antyaliasowania) przez wywołanie procedury `CameraSetMagf` dla obu kamer. Domyślna wartość tego współczynnika (bez wywoływania tej procedury) jest równa 1.

```
void StereoSetMappingf ( StereoRecf *Stereo );
```

Procedura `StereoSetMappingf` określa położenie środków rzutowania kamer i przygotowuje je do użycia (tj. rzutowania punktów itd.), przez wywołanie procedury `CameraSetMappingf`. Przed wywołaniem tej procedury należy wywołać `StereoInitFramef`, `StereoInitPosf` i `StereoSetDimf`.

Opisane niżej procedury ustawiające i zmieniające pozycję pary kamer wywołują tę procedurę, a więc typowe aplikacje nie muszą jej wywoływać bezpośrednio.

```
void StereoInitPosf ( StereoRecf *Stereo );
```

Procedura `StereoInitPosf` ustawia parę kamer w domyślnym położeniu wyjściowym. Obie kamery są ustawiane jednakowo, w takiej pozycji, jak przez procedurę `CameraInitPosf`.

```
void StereoSetRotCentref ( StereoRecf *Stereo,
                          point3f *centre,
                          boolean global_coord, boolean global_fixed );
```

Procedura określa punkt, przez który przechodzi oś obrotu pary kamer podczas zmian jej położenia wykonywanego przez procedury opisane niżej. Sposób jego określania i przetwarzania jest identyczny jak w przypadku jednej kamery.

```
void StereoMoveGf ( StereoRecf *Stereo, vector3f *v );
```

Procedura StereoMoveGf dokonuje przemieszczenia pary kamer (bez zmiany kierunku) o wektor  $v$ , określony w układzie globalnym.

```
void StereoMoveCf ( StereoRecf *Stereo, vector3f *v );
```

Procedura StereoMoveCf dokonuje przemieszczenia pary kamer (bez zmiany kierunku) o wektor  $v$ , określony w układzie kamer.

```
void StereoRotGf ( StereoRecf *Stereo,
                  float psi, float theta, float phi );
```

Procedura StereoRotGf dokonuje obrotu pary kamer, określonego za pomocą kątów Eulera  $\psi$ ,  $\vartheta$ ,  $\varphi$  w układzie globalnym.

```
#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 )
```

Powyższe makra dokonują obrotu pary kamer wokół osi równoległych odpowiednio do osi  $x$ ,  $y$  i  $z$  globalnego układu współrzędnych.

```
void StereoRotVGf ( StereoRecf *Stereo, vector3f *v, float angle );
```

Procedura StereoRotVGf dokonuje obrotu pary kamer wokół osi o kierunku wektora  $v$ , określonego w układzie globalnym.

```
void StereoRotCf ( StereoRecf *Stereo,
                  float psi, float theta, float phi );
```

Procedura StereoRotCf dokonuje obrotu pary kamer określonego przez kąty Eulera  $\psi$ ,  $\vartheta$ ,  $\varphi$  w układzie współrzędnych pary kamer.

```
#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 )
```

Powyższe makra dokonują obrotu pary kamer wokół osi równoległych odpowiednio do osi  $x$ ,  $y$  i  $z$  układu współrzędnych pary kamer.

```
void StereoRotVCf ( StereoRecf *Stereo, vector3f *v, float angle );
```

Procedura StereoRotVCf dokonuje obrotu pary kamer wokół osi o kierunku wektora  $v$ , określonego w układzie współrzędnych pary kamer.

```
void StereoZoomf ( StereoRecf *Stereo, float fchange );
```

Procedura StereoZoomf mnoży długość ogniskowej kamer przez wartość parametru  $fchange$ . Lepiej jej nie używać.





## 6. Biblioteka libpsout

Biblioteka libpsout zawiera procedury generujące plik w języku PostScript<sup>(TM)</sup>. Plik ten opisuje obrazek, określony przez kolejno wywołane procedury rysowania linii itd.

Procedury biblioteczne dzielą się na podstawowe, które generują odpowiedni kod postscriptowy i pomocnicze, których zadaniem jest ułatwienie rysowania odpowiednio opisanych odcinków — można rysować pododcinki o różnej grubości i zaznaczać pewne punkty odcinka przy użyciu markerów, strzałek itd.

### 6.1 Procedury podstawowe

Sformułowanie „procedura rysuje odcinek” lub cokolwiek należy interpretować w ten sposób, że procedura na podstawie podanych parametrów wypisuje do pliku postscriptowego tekst, który podczas interpretowania go (np. w procesie drukowania) spowoduje pojawienie się na obrazie odcinka, czy innej figury.

```
extern short ps_dec_digits;
```

Zmienna `ps_dec_digits` określa liczbę cyfr dziesiętnych współrzędnych punktów wypisywanych do pliku PostScriptowego. Domyślna wartość jest równa 3 i jeśli plik jest tworzony w rozdzielczości 600DPI i nie zawiera dużych powiększeń, to to wystarczy.

```
void ps_WriteBBox ( float x1, float y1, float x2, float y2 );
```

Procedura `ps_WriteBBox` wywołana *przed* utworzeniem pliku postscriptowego określa boks otaczający obrazek, tj. prostokąt, w którym obrazek powinien (ale nie musi) się mieścić i który jest podstawą do umieszczenia obrazka w tekście przez system składu tekstu (np. przez  $\text{\TeX}$ -a). Pierwsze dwa parametry to współrzędne dolnego lewego rogu obrazka, a następne dwa — prawego górnego. Parametry te są podawane w „dużych punktach” (1 duży punkt (1bp w  $\text{\TeX}$ -u) to 1/72 cala).

Odpowiednie liczby można znaleźć za pomocą programu GhostView, a następnie dopisać wywołanie tej procedury do programu i wykonać go ponownie.

```
void ps_OpenFile ( const char *filename, unsigned int dpi );  
void ps_CloseFile ( void );
```

Procedura `ps_OpenFile` tworzy plik, którego nazwa jest określona przez parametr `filename` (ewentualnie kasuje istniejący plik o takiej nazwie) i wypisuje do niego nagłówek. Nagłówek ten zawiera opis boku otaczającego rysunek (jeśli wcześniej została wywołana procedura `ps_WriteBBox`) i skalowanie, które ustala długość jednostki, w której podawane są współrzędne punktów. Jednostka ta jest określona

przez parametr dpi, jeśli np. parametr ten ma wartość 600, to długość jednostki jest równa 1/600 cala. Niektóre procedury są napisane tak, że generowane przez nie symbole (np. strzałki) wyglądają dobrze dla właśnie takiej jednostki.

Procedura `ps_CloseFile` zamyka plik postscriptowy. Powinna być wywołana po utworzeniu obrazka.

```
void ps_Write_Command ( char *command );
```

Procedura `ps_Write_Command` służy do wyprowadzania do pliku postscriptowego dowolnych napisów. Dzięki niej nawet nie mając „gotowej” procedury w opisaną tu bibliotecę, można otrzymać dowolny efekt na obrazku.

```
void ps_Set_Gray ( float gray );
```

Procedura `ps_Set_Gray` ustawia w bieżącym stanie grafiki poziom szarości rysowania, określony przez parametr `gray`, który powinien mieć wartość z przedziału  $[0, 1]$ .

```
void ps_Set_RGB ( float red, float green, float blue );
```

Procedura `ps_Set_RGB` ustawia w bieżącym stanie grafiki kolor o składowych czerwonej, zielonej i niebieskiej określonych przez parametry `red`, `green` i `blue`. Powinny one mieć wartości z przedziału  $[0, 1]$ .

```
void ps_Set_Line_Width ( float w );
```

Procedura `ps_Set_Line_Width` ustawia w bieżącym stanie grafiki szerokość linii, określoną przez parametr `w`, który ma mieć wartość dodatnią.

```
void ps_Draw_Line ( float x1, float y1, float x2, float y2 );
```

Procedura `ps_Draw_Line` rysuje odcinek, którego końce mają współrzędne `x1`, `y1` oraz `x2`, `y2`. Grubość, kolor i inne własności narysowanej linii są określone przez ustawiony w danej chwili stan grafiki.

```
void ps_Set_Clip_Rect ( float w, float h, float x, float y );
```

Procedura `ps_Set_Clip_Rect` ustawia obcinanie do prostokąta o wymiarach `w` (szerokość) i `h` (wysokość), którego dolny lewy wierzchołek ma współrzędne `x`, `y`.

Obcinanie następuje w dodatku do obcinania ustawionego wcześniej. „Odwołanie” obcinania może być przeprowadzone tylko tak, że najpierw zachowujemy stan grafiki wywołując `ps_GSave ()`; , następnie ustawiamy obcinanie i rysujemy, a potem wywołujemy `ps_GRestore ()`; w celu przywrócenia początkowego stanu grafiki.

```
void ps_Draw_Rect ( float w, float h, float x, float y );
```

Procedura `ps_Draw_Rect` rysuje brzeg prostokąta o wymiarach `w` (szerokość) i `h` (wysokość), którego dolny lewy wierzchołek ma współrzędne `x`, `y`. Grubość i kolor narysowanych linii jest określona przez bieżący stan grafiki.

```
void ps_Fill_Rect ( float w, float h, float x, float y );
```

Procedura `ps_Fill_Rect` wypełnia prostokąt o wymiarach `w` (szerokość) i `h` (wysokość), którego dolny lewy wierzchołek ma współrzędne `x`, `y`. Kolor prostokąta jest określony przez bieżący stan grafiki.

```
void ps_Hatch_Rect ( float w, float h, float x, float y,
                    float ang, float d );
```

Procedura `ps_Hatch_Rect` zakreskowuje (rysując linie ukośne) prostokąt o wymiarach `w` (szerokość) i `h` (wysokość), którego dolny lewy wierzchołek ma współrzędne `x`, `y`. Kąt nachylenia linii jest określony przez parametr `ang` (w radianach), a ich odstęp jest określony przez parametr `d`. Kolor i grubość linii jest określona przez bieżący stan grafiki.

```
void ps_Draw_Polyline2f ( int n, const point2f *p );
void ps_Draw_Polyline2d ( int n, const point2d *p );
```

Procedury `ps_Draw_Polyline2f` i `ps_Draw_Polyline2d` rysują łamaną (otwartą) złożoną z `n-1` odcinków, której wierzchołki (`n` punktów, tj. `2n` liczb zmiennopozycyjnych) są podane w tablicy `p`. Kolor i grubość narysowanych odcinków jest określona przez bieżący stan grafiki.

```
void ps_Draw_Polyline2Rf ( int n, const point3f *p );
void ps_Draw_Polyline2Rd ( int n, const point3d *p );
```

Procedury `ps_Draw_Polyline2Rf` i `ps_Draw_Polyline2Rd` rysują łamaną złożoną z `n-1` odcinków, której wierzchołki (`n` punktów, tj. `3n` liczb zmiennopozycyjnych, będących współrzędnymi jednorodnymi) są podane w tablicy `p`. Kolor i grubość narysowanych odcinków jest określona przez bieżący stan grafiki.

```
void ps_Set_Clip_Polygon2f ( int n, const point2f *p );
void ps_Set_Clip_Polygon2d ( int n, const point2d *p );
```

Procedury `ps_Set_Clip_Polygon2f` i `ps_Set_Clip_Polygon2d` ustawiają ścieżkę obcinania do wielokąta o `n` wierzchołkach danych w tablicy `p`. Interpreter PostScriptu obcina do wszystkich ścieżek ustawionych wcześniej (z wyjątkiem ścieżek ustawionych po zapamiętaniu i przed przywróceniem bieżącego stanu grafiki).

```
void ps_Set_Clip_Polygon2Rf ( int n, const point3f *p );
void ps_Set_Clip_Polygon2Rd ( int n, const point3d *p );
```

Procedury `ps_Set_Clip_Polygon2Rf` i `ps_Set_Clip_Polygon2Rd` ustawiają ścieżkę obcinania do wielokąta o `n` wierzchołkach danych w tablicy `p`, za pomocą współrzędnych jednorodnych.

```
void ps_Fill_Polygon2f ( int n, const point2f *p );
void ps_Fill_Polygon2d ( int n, const point2d *p );
```

Procedury `ps_Fill_Polygon2f` i `ps_Fill_Polygon2d` wypełniają wielokąt o  $n$  wierzchołkach podanych w tablicy `p`.

```
void ps_Fill_Polygon2Rf ( int n, const point3f *p );
void ps_Fill_Polygon2Rd ( int n, const point3d *p );
```

Procedury `ps_Fill_Polygon2f` i `ps_Fill_Polygon2d` wypełniają wielokąt o  $n$  wierzchołkach podanych w tablicy `p`, za pomocą współrzędnych jednorodnych.

```
void ps_Draw_BezierCf ( const point2f *p, int n );
void ps_Draw_BezierCd ( const point2d *p, int n );
```

Procedury `ps_Draw_BezierCf` i `ps_Draw_BezierCd` rysują krzywą Béziera stopnia  $n$ , której  $n + 1$  punktów kontrolnych jest podane w tablicy `p`. Dla  $n > 1$  jest w rzeczywistości rysowana łamana złożona z 50 odcinków.

Obliczanie punktów krzywej odbywa się bez używania procedur z biblioteki `libmultibs`.

```
void ps_Draw_Circle ( float x, float y, float r );
```

Procedura `ps_Draw_Circle` rysuje okrąg o promieniu  $r$  i środku w punkcie  $(x, y)$ .

```
void ps_Fill_Circle ( float x, float y, float r );
```

Procedura `ps_Draw_Circle` rysuje koło o promieniu  $r$  i środku w punkcie  $(x, y)$ .

```
void ps_Draw_Arc ( float x, float y, float r, float a0, float a1 );
```

Procedura `ps_Draw_Arc` rysuje łuk okręgu o środku w punkcie  $(x, y)$ , o promieniu  $r$  i kątach początku i końca  $a_0$  i  $a_1$ . Znaczenie wszystkich parametrów jest takie, jak w postscriptowym operatorze `arc`, z tym wyjątkiem, że kąty podaje się w radianach (a nie w stopniach).

```
void ps_Mark_Circle ( float x, float y );
```

Procedura `ps_Mark_Circle` rysuje znaczek (małe koło z białą kropką) w punkcie  $(x, y)$ .

```
void ps_Init_Bitmap ( int w, int h, int x, int y, byte b );
void ps_Out_Line ( byte *data );
```

Procedura `ps_Init_Bitmap` przygotowuje wyprowadzanie do pliku postscriptowego jednobarwnego (czarny-szary-biały) obrazka rastrowego. Obrazek ten ma szerokość  $w$  pikseli, wysokość  $h$  pikseli (wymiar piksela są  $1 \times 1$  w jednostkach określonych przez bieżący układ współrzędnych), a lewy dolny róg jest w punkcie

(x,y). Parametr b określa liczbę bitów na piksel, która musi być równa 1, 2, 4 lub 8.

Po wywołaniu procedury `ps_Init_Bitmap` należy h razy wywołać procedurę `ps_Out_Line`, której parametrem jest tablica  $\lceil w/b \rceil$  bajtów, z których każdy zawiera  $8/b$  upakowanych pikseli. Każde wywołanie tej procedury ma na celu wypisanie do pliku postscriptowego jednego wiersza pikseli, zaczynając od góry.

Dane są wyprowadzane w postaci szesnastkowej, bez kompresji, a zatem wielkość pliku postscriptowego zawierającego obrazek wyprowadzony w ten sposób może być znaczna.

```
void ps_Init_BitmapP ( int w, int h, int x, int y );
void ps_Out_LineP ( byte *data );
```

Procedury `ps_InitBitmapP` i `ps_Out_LineP` służą do wyprowadzenia do pliku postscriptowego jednobarwnego (czarny-szary-biały) obrazka rastrowego. Obrazek ten ma wymiary i położenie określone tak samo, jak obrazek wyprowadzany przez procedury `ps_Init_Bitmap` i `ps_Out_Line`. Liczba bitów na piksel jest równa 8.

Kolejne wiersze pikseli (wyprowadzane przez procedurę `ps_Out_LineP`) są kompresowane (w dość prymitywny sposób), dzięki czemu objętość pliku postscriptowego z obrazkiem wyprowadzonym przy użyciu tych procedur może być mniejsza.

```
void ps_Init_BitmapRGB ( int w, int h, int x, int y );
void ps_Out_LineRGB ( byte *data );
```

Procedury `ps_InitBitmapRGB` i `ps_Out_LineRGB` służą do wyprowadzenia do pliku postscriptowego kolorowego obrazka rastrowego. Obrazek ten ma wymiary i położenie określone tak samo, jak obrazek wyprowadzany przez procedury `ps_Init_Bitmap` i `ps_Out_Line`. Liczba bitów na piksel jest równa 24, tj. każdy piksel jest reprezentowany przez 3 bajty opisujące składowe czerwoną, zieloną i niebieską.

Kolejne wiersze pikseli (wyprowadzane przez procedurę `ps_Out_LineRGB`, której parametrem jest tablica o długości  $3w$  bajtów) nie są kompresowane, przez co plik z tak wyprowadzonym obrazkiem może być dość duży.

```
void ps_Init_BitmapRGBP ( int w, int h, int x, int y );
void ps_Out_LineRGBP ( byte *data );
```

Procedury `ps_InitBitmapRGBP` i `ps_Out_LineRGBP` służą do wyprowadzenia do pliku postscriptowego kolorowego obrazka rastrowego w postaci skompresowanej. Obrazek ten ma wymiary i położenie określone tak samo, jak obrazek wyprowadzany przez procedury `ps_Init_Bitmap` i `ps_Out_Line`. Liczba bitów na piksel jest równa 24, tj. każdy piksel jest reprezentowany przez 3 bajty opisujące składowe czerwoną, zieloną i niebieską.

Kolejne wiersze pikseli (wyprowadzane przez procedurę `ps_Out_LineRGBP`, której parametrem jest tablica o długości  $3w$  bajtów) są kompresowane za pomocą pewnej

odmiany algorytmu *run-length encoding*. Procedura dekompresji jest zrealizowana w PostScriptcie, przez co nie jest to algorytm szczególnie szybki, ale w moich dotychczasowych zastosowaniach był wystarczający. Kiedyś warto będzie zrealizować lepszy algorytm kompresji.

```
void ps_Newpath ( void );
```

Procedura `ps_Newpath` powoduje wypisanie do pliku postscriptowego komendy `newpath`, która inicjalizuje ścieżkę. Ścieżkę tę można dalej rozbudowywać za pomocą procedur `ps_MoveTo` i `ps_LineTo`, a następnie określić sposób jej przetwarzania przez interpreter PostScriptu za pomocą procedury `ps_Write_Command` (np. można podać komendę `stroke`).

```
void ps_MoveTo ( float x, float y );
void ps_LineTo ( float x, float y );
```

Procedury `ps_MoveTo` i `ps_LineTo` generują postscriptowe komendy konstrukcji ścieżki `moveto` i `lineto` z odpowiednimi parametrami. Można ich użyć do skonstruowania ścieżki, która następnie będzie przetwarzana w dowolny sposób.

```
void ps_ShCone ( float x, float y, float x1, float y1,
                 float x2, float y2 );
```

Procedura `ps_ShCone` służy do narysowania pocieniowanego (szarego) trójkąta o wierzchołkach  $(x,y)$ ,  $(x+x1,y+y1)$  i  $(x+x2,y+y2)$ .

```
void ps_GSave ( void );
void ps_GRestore ( void );
```

Procedura `ps_GSave` powoduje wypisanie do pliku postscriptowego komendy `gsave` zapamiętującej (na odpowiednim stosie interpretera PostScriptu) bieżący stan grafiki.

Procedura `ps_GRestore` powoduje wypisanie do pliku postscriptowego komendy `grestore` przywracającej stan grafiki uprzednio zapamiętany przez interpreter.

```
void ps_BeginDict ( int n );
void ps_EndDict ( void );
```

Procedura `ps_BeginDict` wypisuje do pliku postscriptowego tekst `n dict begin`, gdzie `n` jest ciągiem cyfr reprezentujących wartość parametru `n`. Dla interpretera PostScriptu jest to polecenie utworzenia nowego słownika o pojemności `n` symboli i umieszczenie go na stosie słowników.

Procedura `ps_EndDict` wypisuje do pliku postscriptowego napis `end`, który jest poleceniem usunięcia ze stosu słownika. Procedur `ps_BeginDict` i `ps_EndDict` należy używać w parach.

```
void ps_DenseScreen ( void );
```

Procedura `ps_DenseScreen` powoduje zmianę maski rastra na dwa razy gęstsza. Dzięki temu cienkie szare linie wydrukowane na papierze są gładzsze, ale odwzorowanie szarości jest mniej dokładne.

```
void ps_GetSize ( float *x1, float *y1, float *x2, float *y2 );
```

Procedura `ps_GetSize` umożliwia orientacyjne określenie wymiarów prostokąta, w którym mieści się obrazek (tj. jego elementy narysowane przed wywołaniem tej procedury). Procedura ta jednak nie uwzględnia żadnych skutków interpretowania komend wyprowadzonych przy użyciu procedury `ps_Write_Command` (np. takich jak `scale`, `translate` lub komendy rysujące), a także nie uwzględnia ewentualnego obcinania. Dlatego procedura ta jest raczej mało użyteczna.

Parametry na wyjściu otrzymują wartości współrzędnych prostokąta, w którym biblioteka myśli, że mieści się obrazek, w jednostkach określonych przez rozdzielczość podaną przy otwieraniu pliku. Zamiast używać tę procedurę, lepiej jest obrazek obejrzeć przy użyciu programu `Ghostview`, który wyświetla współrzędne punktów wskazanych przez kursor. Odczytane odpowiednie liczby można następnie dopisać jako parametry procedury `ps_WriteBBox` wywoływanej *bezpośrednio przed* procedurą `ps_OpenFile`.

## 6.2 Biblioteka dodatkowa

Procedury dodatkowe mają na celu ułatwienie rysowania odcinków, których pododcinki mogą mieć różne grubości i kolory, a ponadto mogą mieć pewne punkty poznaczane symbolami takimi jak strzałki, kreski itp.

```
#define tickl 10.0
#define tickw 2.0
#define tickd 6.0
#define dotr 12.0
#define arrowl 71.0
#define arroww 12.5
```

Powyższe stałe symboliczne określają połowę długości (`tickl`) i szerokość (`tickw`) kreski poprzecznej do rysowanej linii, szerokość tła takiej kreski, promień kropki oraz długość i połowę szerokości strzałki.

Wymiary te są dobrane tak, aby wspomniane symbole dobrze wyglądały, jeśli jednostka długości (określona przez parametr procedury `ps_OpenFile`) była równa 1/600 cala.

```
void psl_SetLine ( float x1, float y1, float x2, float y2,
                  float t1, float t2 );
```

Procedura `psl_SetLine` określa prostą, której odcinki i punkty będą rysowane i oznaczane. Prosta przechodzi przez punkty  $(x1, y1)$  i  $(x2, y2)$ , które muszą być różne. Punktom tym odpowiadają wartości parametru  $t1$  i  $t2$ , które muszą być różne.

Określenie bieżącej prostej powoduje ustalenie jej jednostkowego wektora kierunkowego  $v$ , za pomocą którego różne procedury dokonują rozmaitych konstrukcji.

```
void psl_GetPointf ( float t, float *x, float *y );
```

Procedura `psl_GetPointf` oblicza punkt prostej określonej przez ostatnie wywołanie procedury `psl_SetLine`, odpowiadający wartości parametru  $t$ . Jego współrzędne są przypisywane parametrom  $*x$  i  $*y$  procedury.

```
float psl_GetDParam ( float dl );
```

Procedura `psl_GetDParam` oblicza przyrost parametru prostej odpowiadający przesunięciu o wektor o długości  $dl$ .

```
void psl_GoAlong ( float s, float *x, float *y );
```

Procedura `psl_GoAlong` otrzymuje na wejściu punkt  $p = (*x, *y)$ . Na wyjściu parametry  $*x$  i  $*y$  mają wartości współrzędnych punktu otrzymanego przez przesunięcie punktu  $p$  w kierunku ustalonej prostej na odległość  $s$ .

```
void psl_GoPerp ( float s, float *x, float *y );
```

Procedura `psl_GoPerp` otrzymuje na wejściu punkt  $p = (*x, *y)$ . Na wyjściu parametry  $*x$  i  $*y$  mają wartości współrzędnych punktu otrzymanego przez przesunięcie punktu  $p$  w kierunku prostopadłym do ustalonej prostej na odległość  $s$ .

```
void psl_Tick ( float t );
```

Procedura `psl_Tick` zaznacza na ustalonej prostej punkt odpowiadający parametrowi  $t$ , za pomocą kreski prostopadłej do tej prostej.

```
void psl_BTick ( float t );
```

Procedura `psl_BTick` zaznacza na ustalonej prostej punkt odpowiadający parametrowi  $t$ , za pomocą kreski prostopadłej do tej prostej. Kreska ta jest grubsza i dłuższa niż kreska rysowana przez procedurę `psl_Tick`. Pomyślane jest to w ten sposób, aby przed narysowaniem obrazka prostej z zaznaczonymi punktami narysować ten obrazek grubszymi liniami i w kolorze tła (np. białym).

```
void psl_HTick ( float t, boolean left );
```

Procedura `psl_HTick` rysuje oznaczenie punktu na bieżącej prostej odpowiadającego parametrowi  $t$  w postaci kreski prostopadłej do bieżącej prostej, przy czym



jeden koniec kreski jest oznaczanym punktem, a długość kreski jest równa tickl. Parametr left określa stronę prostej, po której znajduje się drugi koniec kreski.

```
void psl_Dot ( float t );
```

Procedura psl\_Dot rysuje oznaczenie punktu na bieżącej prostej odpowiadającego parametrowi t. Oznaczenie to jest kółkiem o promieniu dotr.

```
void psl_HDot ( float t );
```

Procedura psl\_HDot rysuje oznaczenie punktu na bieżącej prostej odpowiadającego parametrowi t. Oznaczenie to jest kółkiem o promieniu nieco większym niż dotr. Przeznaczeniem tej procedury jest narysowanie tła (np. białego) dla (czarnej) kropki rysowanej później przez procedurę psl\_Dot.

```
void psl_TrMark ( float x, float y );
```

Procedura psl\_TrMark rysuje oznaczenie punktu (x,y) (nie związanego z bieżącą prostą) w postaci białego trójkątka równoramiennego o czarnych krawędziach, którego podstawa jest pozioma, a najwyższy wierzchołek to oznaczany punkt.

```
void psl_BlackTrMark ( float x, float y );
```

Procedura psl\_BlackTrMark rysuje oznaczenie punktu (x,y) (nie związanego z bieżącą prostą) w postaci czarnego trójkątka równoramiennego, którego podstawa jest pozioma, a najwyższy wierzchołek to oznaczany punkt.

```
void psl_HighTrMark ( float x, float y );
```

Procedura psl\_HighTrMark rysuje oznaczenie punktu (x,y) (nie związanego z bieżącą prostą) w postaci białego trójkątka równoramiennego o czarnych krawędziach, którego podstawa jest pozioma, a najwyższy wierzchołek to oznaczany punkt. Wysokość tego trójkątka jest większa niż wysokość trójkątka rysowanego przez procedurę psl\_TrMark.

```
void psl_BlackHighTrMark ( float x, float y );
```

Procedura psl\_BlackHighTrMark rysuje oznaczenie punktu (x,y) (nie związanego z bieżącą prostą) w postaci czarnego trójkątka równoramiennego którego podstawa jest pozioma, a najwyższy wierzchołek to oznaczany punkt. Wysokość tego trójkątka jest większa niż wysokość trójkątka rysowanego przez procedurę psl\_BlackTrMark.

```
void psl_LTrMark ( float t );
void psl_BlackLTrMark ( float t );
void psl_HighLTrMark ( float t );
void psl_BlackHighLTrMark ( float t );
```

Powyższe procedury rysują oznaczenia punktu bieżącej prostej, który odpowiada parametrowi t, za pomocą procedur psl\_TrMark, psl\_BlackTrMark,

psl\_HighTrMark i psl\_BlackHighTrMark. W zasadzie nadają się one tylko do oznaczania punktów na prostych poziomych.

```
void psl_Arrow ( float t, boolean sgn );
```

Procedura psl\_Arrow oznacza punkt bieżącej prostej odpowiadający parametrowi  $t$  za pomocą strzałki (a dokładniej trójkątka stanowiącego „grot” strzałki) o kierunku bieżącej prostej. Parametr  $sgn$  określa zwrot strzałki.

```
void psl_BkArrow ( float t, boolean sgn );
```

Procedura psl\_BkArrow rysuje tło (np. białe) dla strzałki, którą można następnie narysować za pomocą procedury psl\_Arrow. Parametry tej procedury mają identyczne znaczenie jak parametry procedury psl\_Arrow.

```
void psl_Draw ( float ta, float tb, float w );
```

Procedura psl\_Draw rysuje odcinek bieżącej prostej, którego końce są określone przez parametry  $ta$  i  $tb$ . Parametr  $w$  określa grubość rysowanej kreski.

```
void psl_ADraw ( float ta, float tb, float ea, float eb, float w );
```

Procedura psl\_ADraw rysuje odcinek bieżącej prostej, którego końce są określone następująco: najpierw wyznaczany jest punkt  $p_a$  odpowiadający parametrowi  $ta$ , a następnie punkt otrzymany przez dodanie do  $p_a$  iloczynu jednostkowego wektora kierunkowego bieżącej prostej (zobacz opis procedury psl\_SetLine) i wartości parametru  $ea$ . Drugi koniec odcinka jest wyznaczany podobnie, za pomocą parametrów  $tb$  i  $eb$ . Parametr  $w$  określa grubość rysowanej kreski.

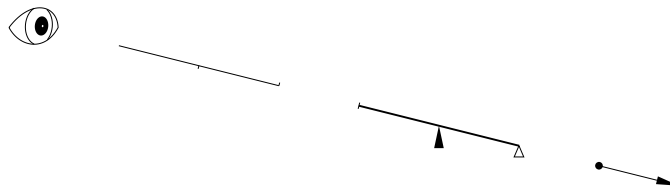
```
void psl_MapsTo ( float t );
```

Procedura psl\_MapsTo rysuje w punkcie  $t$  bieżącej prostej strzałkę innego rodzaju, bardziej odpowiednią do rysowania np. diagramów przemennych.

```
void psl_DrawEye ( float t, byte cc, float mag, float ang );
```

Procedura psl\_DrawEye rysuje symbol oczka. Na różnych rysunkach schematycznych symbol ten może się przydać do zaznaczania położenia obserwatora.

Parametr  $t$  określa punkt bieżącej linii, który ma być oznaczony symbolem oczka. Parametr  $cc$  powinien mieć wartość od 0 do 3, która określa orientację oczka. Parametr  $mag$  określa wielkość oczka, zaś parametr  $ang$  określa dodatkowy kąt (w radianach), o który oczko powinno zostać obrócone w celu otrzymania poprawnego efektu.



Rys. 6.1. Linia z pozaznaczanymi punktami

**Przykład:** Poniższy program generuje obrazek pokazany na rys. 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, -arrow1, 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. Biblioteka libmultibs

## 7.1 Podstawowe definicje i sposoby reprezentowania krzywych i płatów

### 7.1.1 Krzywe Béziera

Krzywa Béziera jest określona wzorem

$$\mathbf{p}(t) = \sum_{i=0}^n \mathbf{p}_i B_i^n(t), \quad (7.1)$$

w którym występują punkty kontrolne  $\mathbf{p}_0, \dots, \mathbf{p}_n$  oraz wielomiany Bernsteina

$$B_i^n(t) \stackrel{\text{def}}{=} \binom{n}{i} t^i (1-t)^{n-i}, \quad i = 0, \dots, n. \quad (7.2)$$

Łamana, której kolejnymi wierzchołkami są punkty  $\mathbf{p}_0, \dots, \mathbf{p}_n$ , nazywa się łamaną kontrolną krzywej. Każdy punkt kontrolny ma  $d$  współrzędnych i wtedy krzywa leży w przestrzeni  $d$ -wymiarowej. W szczególności dla  $d = 1$  wzór (7.1) określa wielomian zmiennej  $t$  stopnia co najwyżej  $n$ .

Reprezentacja krzywej Béziera składa się z liczby  $n$  i ciągu  $n + 1$  punktów kontrolnych, których współrzędne (w sumie  $(n + 1)d$  liczb zmiennopozycyjnych) należy umieścić w tablicy „po kolei” (tj. najpierw  $d$  współrzędnych punktu  $\mathbf{p}_0$ , potem  $\mathbf{p}_1$  itd.).

Wymierna krzywa Béziera jest określona wzorem

$$\mathbf{p}(t) = \frac{\sum_{i=0}^n w_i \mathbf{p}_i B_i^n(t)}{\sum_{i=0}^n w_i B_i^n(t)}, \quad (7.3)$$

w którym występują wielomiany Bernsteina, punkty kontrolne  $\mathbf{p}_0, \dots, \mathbf{p}_n$  i wagi  $w_0, \dots, w_n$ . Krzywa taka leży w tej samej przestrzeni, w której leżą punkty kontrolne.

Jeśli  $w_i = 0$  dla pewnego  $i$ , to wyrażenie  $w_i \mathbf{p}_i$  możemy zastąpić przez dowolny wektor  $\mathbf{v}_i$ , rozszerzając definicję krzywej, ale przynajmniej jedna waga musi być różna od zera.

Punkty kontrolne  $\mathbf{p}_i$  krzywej wymiernej są wygodne dla użytkownika programu, który może interakcyjnie je dobierać, ale procedury w tej bibliotece przetwarzają tzw. reprezentację jednorodną. Dla krzywej w przestrzeni  $d$ -wymiarowej jest nią krzywa wielomianowa położona w przestrzeni o wymiarze  $d + 1$ :

$$\mathbf{P}(t) = \sum_{i=0}^n \mathbf{P}_i B_i^n(t), \quad (7.4)$$

której punkty kontrolne  $\mathbf{P}_i$  są dane wzorem

$$\mathbf{P}_i = \begin{bmatrix} w_i \mathbf{p}_i \\ w_i \end{bmatrix}. \quad (7.5)$$

Ostatnia (tj.  $d + \text{pierwsza}$ ) współrzędna jednorodna jest więc odpowiednią współrzędną wagową. Jeśli  $w_i = 0$  to

$$\mathbf{P}_i = \begin{bmatrix} \mathbf{v}_i \\ 0 \end{bmatrix}. \quad (7.6)$$

Współrzędne kartezjańskie punktu  $\mathbf{p}(t)$  krzywej wymiernej otrzymuje się przez podzielenie pierwszych  $d$  współrzędnych punktu  $\mathbf{P}(t)$  przez jego ostatnią współrzędną jednorodną.

Reprezentacja krzywej wymiernej np. w przestrzeni trójwymiarowej składa się z liczby  $n$  (która określa stopień reprezentacji) i tablicy  $4(n+1)$  liczb zmiennopozycyjnych, będących współrzędnymi kolejnych punktów  $\mathbf{P}_i$ . Ponieważ krzywe jednorodne są zwykłymi krzywymi wielomianowymi, więc do ich przetwarzania w większości przypadków służą te same procedury, co do przetwarzania wielomianowych krzywych Béziera.

### 7.1.2 Prostokątne płaty Béziera

Prostokątny płat Béziera jest określony wzorem

$$\mathbf{p}(u, v) = \sum_{i=0}^n \sum_{j=0}^m \mathbf{p}_{ij} B_i^n(u) B_j^m(v), \quad (7.7)$$

w którym występują wielomiany Bernsteina  $B_i^n$  i  $B_j^m$  stopni odpowiednio  $n$  i  $m$  oraz punkty kontrolne  $\mathbf{p}_{ij}$ . Dla ustalenia uwagi wierszem siatki kontrolnej płata nazywamy łamaną o wierzchołkach w punktach  $\mathbf{p}_{0j}, \dots, \mathbf{p}_{nj}$  (dla  $j \in \{0, \dots, m\}$ ), a kolumną jest to łamana o wierzchołkach  $\mathbf{p}_{i0}, \dots, \mathbf{p}_{im}$  (dla każdego  $i \in \{0, \dots, n\}$ ).

Reprezentacja płata składa się zatem z liczb dodatnich  $n$  i  $m$  oraz  $(n+1)(m+1)$  punktów kontrolnych, czyli  $(n+1)(m+1)d$  liczb zmiennopozycyjnych, umieszczonych w tablicy w kolejności następującej: najpierw  $d$  współrzędnych punktu  $\mathbf{p}_{00}$ , potem  $d$  współrzędnych punktu  $\mathbf{p}_{01}$  itd. Po współrzędnych punktu  $\mathbf{p}_{0m}$  należy podać współrzędne punktu  $\mathbf{p}_{10}$  i tak dalej, aż do punktu  $\mathbf{p}_{nm}$ . Inaczej mówiąc, w tablicy są umieszczone kolejne kolumny siatki kontrolnej.

Na opisaną wyżej tablicę można patrzeć na wiele różnych sposobów. Na przykład, aby poddać płat ustalonemu przekształceniu afinicznemu wystarczy zastosować to przekształcenie do wszystkich jego punktów kontrolnych. Będziemy więc wtedy widzieć tę tablicę jako jednowymiarową tablicę punktów.

Możemy też dokonać podziału płata za pomocą algorytmu de Casteljau na dwie części, dzieląc na połowy przedział zmienności parametru  $u$  lub  $v$ . W tym ostatnim

przypadku wystarczy zastosować ten algorytm do wszystkich kolumn tak jak gdyby były to łamane kontrolne krzywe Béziera. Tablica zawiera zatem reprezentację  $n + 1$  krzywych Béziera stopnia  $m$  i pierwsza współrzędna pierwszego punktu następnej krzywej znajduje się o  $(m + 1)d$  miejsc za początkiem reprezentacji krzywej danej. Zatem przyjmujemy, że podziałka tej tablicy jest równa  $d(m + 1)$  (gdzie  $d$  jest wymiarem przestrzeni, w której leży płąt).

Aby podzielić przedział zmienności parametru  $u$ , należy zastosować algorytm de Casteljau do wszystkich wierszy statki kontrolnej płata. Okazuje się, że reprezentację płata można potraktować jak reprezentację krzywej Béziera stopnia  $n$ , położonej w przestrzeni o wymiarze  $(m + 1)d$  (każda kolumna siatki kontrolnej jest jednym punktem tej przestrzeni). W tym przypadku przetwarzamy *jedną krzywą* Béziera stopnia  $n$  w przestrzeni  $(m + 1)d$ -wymiarowej, a podziałka tablicy jest nieistotna, ponieważ krzywa jest tylko jedna.

Wymierny płąt Béziera jest określony wzorem

$$p(u, v) = \frac{\sum_{i=0}^n \sum_{j=0}^m w_{ij} p_{ij} B_i^n(u) B_j^m(v)}{\sum_{i=0}^n \sum_{j=0}^m w_{ij} B_i^n(u) B_j^m(v)},$$

w którym oprócz wielomianów Bernsteina i punktów kontrolnych występują wagi  $w_{ij}$ . Procedury przetwarzające wymierne płaty Béziera w bibliotece libmultibs przetwarzają reprezentację jednorodną, czyli tablicę punktów kontrolnych  $P_{ij}$  wielomianowego płata Béziera

$$P(u, v) = \sum_{i=0}^n \sum_{j=0}^m P_{ij} B_i^n(u) B_j^m(v), \quad (7.8)$$

położonego w przestrzeni o wymiarze  $d + 1$ . Związek punktów kontrolnych  $p_{ij}$  i wag  $w_{ij}$  płata wymiernego z punktami  $P_{ij}$  jest taki sam jak w przypadku wymiernych krzywych Béziera. Sposób przechowywania w tablicy punktów kontrolnych  $P_{ij}$  płata jednorodnego jest taki sam jak w przypadku płata wielomianowego (tylko wymiar przestrzeni, w której leży płąt jednorodny i jego punkty kontrolne jest o 1 większy).

### 7.1.3 Krzywe B-sklejane

Przyjmijmy  $n \geq 0$  i niemalejący ciąg węzłów (liczb rzeczywistych)  $u_0, \dots, u_N$ , taki że  $N > 2n$  i  $u_n < u_{N-n}$ . Krzywa B-sklejana stopnia  $n$  oparta na tym ciągu węzłów jest określona wzorem

$$s(t) = \sum_{i=0}^{N-n-1} d_i N_i^n(t), \quad (7.9)$$

w którym występują punkty kontrolne  $d_0, \dots, d_{N-n-1}$  oraz funkcje B-sklejane  $N_0^n, \dots, N_{N-n-1}^n$ . Funkcje te mają kilka równoważnych definicji, m.in. można je

zdefiniować rekurencyjnym wzorem Mansfielda-deBoora-Coxa:

$$N_i^0(t) = \begin{cases} 1 & \text{dla } u_i \leq t < u_{i+1}, \\ 0 & \text{w przeciwnym razie,} \end{cases} \quad (7.10)$$

$$N_i^j(t) = \frac{t - u_i}{u_{i+j} - u_i} N_i^{j-1}(t) + \frac{u_{i+j+1} - t}{u_{i+j+1} - u_{i+1}} N_{i+1}^{j-1}(t) \quad \text{dla } j = 1, \dots, n. \quad (7.11)$$

Dziedzina krzywej B-sklejanej jest przedział  $[u_n, u_{N-n-1}]$ . W każdym przedziale  $[u_k, u_{k+1}]$  (dla  $n \leq k < N - n$ ) krzywa B-sklejana jest łukiem wielomianowym stopnia co najwyżej  $n$ .

Reprezentacja krzywej B-sklejanej składa się z liczb całkowitych  $n$  i  $N$ , określających odpowiednio stopień i indeks ostatniego węzła, ciągu węzłów (tablicy liczb zmiennopozycyjnych)  $u_0, \dots, u_N$  oraz ciągu punktów kontrolnych  $d_0, \dots, d_{N-n-1}$ , położonych w tej samej przestrzeni co krzywa — jeśli wymiarem tej przestrzeni jest  $d$ , to w odpowiedniej tablicy należy podać  $(N - n)d$  liczb zmiennopozycyjnych.

Dla ułatwienia wyjaśniania niuansów działania procedur przyjąłem następującą terminologię: węzły brzegowe to te, które wyznaczają brzeg dziedziny krzywej, czyli  $u_n, u_{N-n}$  i wszystkie węzły równe jednemu z tych dwóch; węzły brzegowe dzielimy na lewe i prawe. Węzły wewnętrzne to wszystkie węzły należące do przedziału  $(u_n, u_{N-n})$ ; węzłom tym odpowiadają punkty połączenia łuków wielomianowych krzywej. Oprócz tego mamy węzły zewnętrzne, które nie należą do przedziału  $[u_n, u_{N-n}]$ . Niezależnie od tego węzły  $u_0$  i  $u_N$  będą nazywane węzłami skrajnymi. Na przykład, jeśli  $n = 3$ ,  $N = 15$  i

$$u_0 < u_1 = u_2 < u_3 = u_4 = u_5 < u_6 \leq \dots \leq u_{11} < u_{12} = u_{13} = u_{14} = u_{15},$$

to węzły  $u_0$ ,  $u_1$  i  $u_2$  są zewnętrzne, węzły  $u_3$ ,  $u_4$  i  $u_{12}, \dots, u_{15}$  są brzegowe, a pozostałe węzły są wewnętrzne. Węzły skrajne to  $u_0$  i  $u_{15}$ .

Węzły skrajne są potrzebne w definicji funkcji B-sklejanych  $N_0^n$  oraz  $N_{N-n-1}^n$ , ale nie mają one wpływu na wartości tych funkcji w przedziale  $[u_n, u_{N-n}]$ , a więc także na kształt krzywej. W różnych pakietach oprogramowania węzły te są wymagane lub nie; w bibliotece libmultibs węzły te należy podawać (wystarczy, że spełniają warunki  $u_0 \leq u_1$  oraz  $u_{N-1} \leq u_N$ ).

Ustalona krzywa sklejana może mieć różne reprezentacje; mogą one różnić się stopniem lub ciągiem węzłów. Konstruowanie reprezentacji, której ciąg węzłów zawiera dodatkowe liczby nazywa się wstawianiem węzłów. W szczególności reprezentacja, w której wszystkie węzły mają krotność (liczbę wystąpień)  $n + 1$  (tj. jest  $u_0 = \dots = u_n$ ,  $u_{n+1} = \dots = u_{2n+1}$ ,  $u_{2n+2} = \dots = u_{3n+2}$  itd.), jest reprezentacją krzywej kawałkami Béziera.

Jeśli ostatni (o największym indeksie) lewy węzeł brzegowy ma indeks  $k > n$ , to początkowe  $k - n$  węzłów (zaczynając od skrajnego lewego) i początkowe  $k - n$  punktów jest w reprezentacji krzywej zbędnych i można (dla pewnych celów trzeba) je pominąć. Podobnie, jeśli pierwszy (o najmniejszym indeksie) prawy węzeł



brzegowy ma indeks  $k < N - n$ , to ostatnie  $N - n - k$  węzłów i punktów kontrolnych jest zbędnych. Reprezentacje ze zbędnymi węzłami i punktami kontrolnymi mogą powstać w wyniku wstawiania węzłów (np. podczas konwersji do postaci kawałkami Béziera), oraz w wyniku wyznaczania B-sklejanej reprezentacji pochodnej krzywej.

Krzywą B-sklejaną stopnia  $n$ , której węzły brzegowe mają krotność większą lub równą  $n$ , nazywamy krzywą o końcach zaczepionych. Jeśli ostatni (o największym indeksie) lewy węzeł brzegowy ma indeks  $k$  (ponieważ węzły liczymy od 0, więc oczywiście  $k \geq n$ ), to punkt kontrolny  $\mathbf{d}_{k-n}$  jest punktem krzywej odpowiadającym parametrowi  $u_k$ , tj. lewemu końcowi dziedziny. Jeśli  $k = n$ , to jest to punkt  $\mathbf{d}_0$ ; w przeciwnym razie punkty  $\mathbf{d}_0, \dots, \mathbf{d}_{k-n-1}$  nie mają wpływu na kształt krzywej (i wraz z węzłami  $u_0, \dots, u_{k-n-1}$  są zbędne). Podobna reguła dotyczy prawego węzła brzegowego o najmniejszym indeksie — jeśli jest to węzeł  $u_{N-n}$  o krotności  $n$  lub  $n+1$ , to punkt kontrolny  $\mathbf{d}_{N-n-1}$  jest punktem końcowym krzywej (odpowiada on parametrowi  $u_{N-n}$ ).

Krzywa, której węzły brzegowe mają krotność mniejszą niż stopień, nazywa się krzywą o końcach swobodnych. Każdy koniec krzywej może być zaczepiony lub swobodny niezależnie od drugiego końca.

Zamknięte krzywe B-sklejane są reprezentowane tak samo jak wszystkie krzywe B-sklejane. Aby krzywa B-sklejana stopnia  $n$  była zamknięta, muszą być spełnione następujące warunki: ciąg węzłów  $u_1, \dots, u_{N-1}$  musi składać się z kolejnych elementów nieskończonego ciągu liczb, takiego że ciąg różnic jest nieujemny i okresowy, o okresie

$$K = N - 2n,$$

przy czym musi być spełniony warunek  $N > 3n$ . Ciąg węzłów musi być niemalejący i musi istnieć liczba dodatnia  $T$ , taka że

$$u_{k+K} - u_k = T \quad \text{dla } k = 1, \dots, 2n - 1.$$

Węzły  $u_0$  i  $u_N$  są nieistotne dla kształtu krzywej, ale muszą spełniać warunki  $u_0 \leq u_1$  i  $u_{N-1} \leq u_N$ .

Ciąg  $\mathbf{d}_0, \dots, \mathbf{d}_{N-n-1}$  musi składać się z kolejnych elementów nieskończonego okresowego ciągu punktów o okresie  $N - 2n$ , a zatem muszą być spełnione równości

$$\mathbf{d}_{k+K} = \mathbf{d}_k \quad \text{dla } k = 0, \dots, n - 1.$$

Aplikacja może wykorzystywać „oszczędną” reprezentację zamkniętej krzywej B-sklejanej, w której węzły  $u_{K+1}, \dots, u_{N-1}$  oraz punkty kontrolne  $\mathbf{d}_K, \dots, \mathbf{d}_{N-n-1}$ , możliwe do odtworzenia na podstawie powyższych warunków, są nieobecne. Jednak aby użyć procedur z biblioteki libmultibs trzeba utworzyć „roboczą” reprezentację krzywej, składającą się z tablic zawierających wszystkie węzły i punkty kontrolne.

Obliczanie punktu i wiele innych obliczeń dla krzywych zamkniętych może być wykonywane przez procedury biblioteki libmultibs przeznaczone do przetwarzania „zwykłych” krzywych o końcach swobodnych. Zmiany reprezentacji takie jak

wstawianie i usuwanie węzłów oraz podwyższanie stopnia wymaga użycia procedur, które zapewnią spełnienie podanych wyżej warunków przez nową reprezentację krzywej. Procedury takie mają w nazwie słowo „Closed”, przy czym w większości przypadków czekają dopiero na napisanie.

#### 7.1.4 Płaty B-sklejane

Płat B-sklejany jest określony wzorem

$$s(u, v) = \sum_{i=0}^{N-n-1} \sum_{j=0}^{M-m-1} d_{ij} N_i^n(u) N_j^m(v), \quad (7.12)$$

w którym występują dwa układy funkcji B-sklejanych stopni (w ogólności różnych)  $n$  i  $m$ , oparte odpowiednio na (w ogólności różnych, nawet jeśli  $n = m$ ) ciągach węzłów  $u_0, \dots, u_N$  i  $v_0, \dots, v_M$ . Każdy z tych ciągów musi być niemalejący i dostatecznie długi (ma być  $N > 2n$ ,  $M > 2m$ ,  $u_n < u_{N-n}$  i  $v_m < v_{M-m}$ ). Do każdego z tych ciągów stosuje się terminologia i wszystkie uwagi podane w poprzednim punkcie.

Tablica punktów kontrolnych  $d_{ij}$ , które razem z węzłami stanowią reprezentację płata, zawiera kolejno współrzędne punktów  $d_{00}, d_{01}, \dots, d_{0, N-n-1}$ , a następnie  $d_{10}, d_{11}, \dots, d_{1, N-n-1}$  itd., czyli kolejne kolumny siatki kontrolnej płata.

Między tymi kolumnami mogą występować w tablicy obszary nieużywane, które umożliwiają np. wstawianie węzła do ciągu  $(v_i)$  węzłów początkowej reprezentacji płata. Odbywa się to tak, jakby węzeł był wstawiany do reprezentacji wielu krzywych B-sklejanych, których łamanymi kontrolnymi są kolumny siatki kontrolnej płata. Po wstawieniu węzła długość każdego obszaru nieużywanego zmniejsza się o  $d$  miejsc na liczby zmiennopozycyjne (gdzie  $d$  jest wymiarem przestrzeni, w której leży płat). Podziałką tablicy w tym przypadku jest odległość początków ciągów współrzędnych kolejnych kolumn.

Odpowiednikami krzywych o końcach zaczepionych i krzywych o końcach swobodnych są płaty o brzegach zaczepionych i swobodnych. Na przykład, jeśli ciąg węzłów „ $u$ ” spełnia warunek  $u_1 = \dots = u_n < u_{n+1}$ , to krzywa stałego parametru  $u = u_n$  (jedna z czterech krzywych brzegowych płata) jest krzywą B-sklejaną stopnia  $m$ , opartą na ciągu węzłów „ $v$ ”, której łamana kontrolna jest pierwszą kolumną siatki kontrolnej płata. Oczywiście, każdy z czterech brzegów płata może być zaczepiony albo swobodny niezależnie od pozostałych.

Odpowiednikami krzywych zamkniętych są płaty zamknięte, które mogą być rurkami lub torusami. Jeden lub oba ciągi węzłów, a także ciąg wierszy lub kolumn siatki kontrolnej (traktowanych jako punkty) muszą spełniać warunki dla zamkniętych krzywych B-sklejanych.

### 7.1.5 Krzywe i płaty NURBS

Krzywe i płaty NURBS (ang. *non-uniform rational B-splines*) są to krzywe i płaty powierzchni kawałkami wymiernych, których związek z krzywymi i płatami B-sklejanymi jest taki sam, jak związek wymiernych krzywych i płatów Béziera z wielomianowymi krzywymi i płatami Béziera. Zatem, można wybrać jeden lub dwa ciągi węzłów oraz układ punktów kontrolnych  $\mathbf{d}_i$  lub  $\mathbf{d}_{ij}$  w przestrzeni  $d$ -wymiarowej i każdemu punktowi przyporządkować wagę  $w_i$  albo  $w_{ij}$ . Następnie wystarczy określić wektory w przestrzeni  $d + 1$ -wymiarowej

$$\mathbf{D}_i = \begin{bmatrix} w_i \mathbf{d}_i \\ w_i \end{bmatrix} \quad \text{albo} \quad \mathbf{D}_{ij} = \begin{bmatrix} w_{ij} \mathbf{d}_{ij} \\ w_{ij} \end{bmatrix} \quad (7.13)$$

i obliczać punkty krzywej lub płata B-sklejanego (którego to są punkty kontrolne) położonego w tej przestrzeni, po czym dzielić pierwsze  $d$  współrzędnych każdego takiego punktu przez współrzędną  $d + 1$  (tzw. wagową), co daje w wyniku współrzędne kartezjańskie odpowiednich punktów krzywej lub płata wymiernego.

Procedury w bibliotece libmultibs przetwarzają właśnie taką reprezentację jednorodną.

### 7.1.6 Płaty Coonsa

Płaty Coonsa klasy  $C^k$  są to płaty tensorowe określone za pomocą odpowiednio gładkich krzywych opisujących brzeg i tzw. pochodne poprzeczne rzędu  $1, \dots, k$ . Dla dowolnego  $k \in \mathbb{N}$  płat Coonsa jest określony wzorem

$$\mathbf{p}(u, v) = \mathbf{p}_1(u, v) + \mathbf{p}_2(u, v) - \mathbf{p}_3(u, v), \quad (7.14)$$

w którym

$$\mathbf{p}_1(u, v) = \mathbf{C}(u)\hat{\mathbf{H}}(v)^T, \quad \mathbf{p}_2(u, v) = \tilde{\mathbf{H}}(u)\mathbf{D}(v)^T, \quad \mathbf{p}_3(u, v) = \tilde{\mathbf{H}}(u)\mathbf{P}\hat{\mathbf{H}}(v)^T,$$

przy czym

$$\begin{aligned} \mathbf{C}(u) &= [\mathbf{c}_{00}(u), \mathbf{c}_{10}(u), \mathbf{c}_{01}(u), \mathbf{c}_{11}(u), \dots, \mathbf{c}_{0k}(u), \mathbf{c}_{1k}(u)], \\ \mathbf{D}(v) &= [\mathbf{d}_{00}(v), \mathbf{d}_{10}(v), \mathbf{d}_{01}(v), \mathbf{d}_{11}(v), \dots, \mathbf{d}_{0k}(v), \mathbf{d}_{1k}(v)], \\ \tilde{\mathbf{H}}(u) &= [\tilde{\mathbf{H}}_{00}(u), \tilde{\mathbf{H}}_{10}(u), \tilde{\mathbf{H}}_{01}(u), \tilde{\mathbf{H}}_{11}(u), \dots, \tilde{\mathbf{H}}_{0k}(u), \tilde{\mathbf{H}}_{1k}(u)], \\ \hat{\mathbf{H}}(v) &= [\hat{\mathbf{H}}_{00}(v), \hat{\mathbf{H}}_{10}(v), \hat{\mathbf{H}}_{01}(v), \hat{\mathbf{H}}_{11}(v), \dots, \hat{\mathbf{H}}_{0k}(v), \hat{\mathbf{H}}_{1k}(v)]. \end{aligned}$$

Krzywe  $\mathbf{c}_{00}, \dots, \mathbf{c}_{1k}$  opisują dwa przeciwległe brzegi płata i pochodne poprzeczne na tych brzegach. Krzywe te muszą mieć identyczną dziedzinę, którą oznaczmy  $[a, b]$ . Podobnie, krzywe  $\mathbf{d}_{00}, \dots, \mathbf{d}_{1k}$  opisują drugą parę przeciwległych brzegów i pochodne poprzeczne i również muszą mieć identyczną dziedzinę,  $[c, d]$ . Dziedziną płata Coonsa jest prostokąt  $[a, b] \times [c, d]$ .

Macierz  $\mathbf{P}$  ma wymiary  $(2k+2) \times (2k+2)$  i składa się z punktów danych krzywych oraz wektorów ich pochodnych rzędu  $1, \dots, k$ :

$$\mathbf{P} = \begin{bmatrix} \mathbf{c}_{00}(\mathbf{a}) & \mathbf{c}_{10}(\mathbf{a}) & \dots & \mathbf{c}_{0k}(\mathbf{a}) & \mathbf{c}_{1k}(\mathbf{a}) \\ \mathbf{c}_{00}(\mathbf{b}) & \mathbf{c}_{10}(\mathbf{b}) & \dots & \mathbf{c}_{0k}(\mathbf{b}) & \mathbf{c}_{1k}(\mathbf{b}) \\ \vdots & \vdots & & \vdots & \vdots \\ \mathbf{c}_{00}^{(k)}(\mathbf{a}) & \mathbf{c}_{10}^{(k)}(\mathbf{a}) & \dots & \mathbf{c}_{0k}^{(k)}(\mathbf{a}) & \mathbf{c}_{1k}^{(k)}(\mathbf{a}) \\ \mathbf{c}_{00}^{(k)}(\mathbf{b}) & \mathbf{c}_{10}^{(k)}(\mathbf{b}) & \dots & \mathbf{c}_{0k}^{(k)}(\mathbf{b}) & \mathbf{c}_{1k}^{(k)}(\mathbf{b}) \end{bmatrix} = \begin{bmatrix} \mathbf{d}_{00}(\mathbf{c}) & \mathbf{d}_{00}(\mathbf{d}) & \dots & \mathbf{d}_{0k}(\mathbf{c}) & \mathbf{d}_{0k}(\mathbf{d}) \\ \mathbf{d}_{10}(\mathbf{c}) & \mathbf{d}_{10}(\mathbf{d}) & \dots & \mathbf{d}_{1k}(\mathbf{c}) & \mathbf{d}_{1k}(\mathbf{d}) \\ \vdots & \vdots & & \vdots & \vdots \\ \mathbf{d}_{00}^{(k)}(\mathbf{c}) & \mathbf{d}_{00}^{(k)}(\mathbf{d}) & \dots & \mathbf{d}_{0k}^{(k)}(\mathbf{c}) & \mathbf{d}_{0k}^{(k)}(\mathbf{d}) \\ \mathbf{d}_{10}^{(k)}(\mathbf{c}) & \mathbf{d}_{10}^{(k)}(\mathbf{d}) & \dots & \mathbf{d}_{1k}^{(k)}(\mathbf{c}) & \mathbf{d}_{1k}^{(k)}(\mathbf{d}) \end{bmatrix}. \quad (7.15)$$

Krzywe określające płat muszą spełniać warunki zgodności, przedstawione wyżej w postaci równości macierzy.

Funkcje  $\tilde{\mathbf{H}}_{mj}(\mathbf{u})$  i  $\hat{\mathbf{H}}_{mj}(\mathbf{v})$  tworzą tzw. lokalne bazy Hermite'a. Jest przyjęte założenie, że funkcje te są wielomianami stopnia  $2k+1$  dla  $k \in \{1, 2\}$ , choć można by użyć zamiast nich innych funkcji klasy  $C^k$ , np. funkcji sklepanych stopnia  $k+1$ . Funkcje te są określone wzorem

$$\tilde{\mathbf{H}}_{mj}(\mathbf{u}) = (\mathbf{b} - \mathbf{a})^j \mathbf{H}_{mj}\left(\frac{\mathbf{u} - \mathbf{a}}{\mathbf{b} - \mathbf{a}}\right), \quad \hat{\mathbf{H}}_{mj}(\mathbf{v}) = (\mathbf{d} - \mathbf{c})^j \mathbf{H}_{mj}\left(\frac{\mathbf{v} - \mathbf{c}}{\mathbf{d} - \mathbf{c}}\right).$$

Dla  $k = 1$  jest

$$\begin{aligned} \mathbf{H}_{00}(\mathbf{t}) &= \mathbf{B}_0^3(\mathbf{t}) + \mathbf{B}_1^3(\mathbf{t}), & \mathbf{H}_{10}(\mathbf{t}) &= \mathbf{B}_2^3(\mathbf{t}) + \mathbf{B}_3^3(\mathbf{t}), \\ \mathbf{H}_{01}(\mathbf{t}) &= \frac{1}{3}\mathbf{B}_1^3(\mathbf{t}), & \mathbf{H}_{11}(\mathbf{t}) &= -\frac{1}{3}\mathbf{B}_2^3(\mathbf{t}). \end{aligned}$$

Ponieważ wielomiany użyte do określenia płata, tj. interpolacji w obu kierunkach między zadanymi krzywymi, są dla  $k = 1$  trzeciego stopnia, więc płaty Coonsa klasy  $C^1$  są nazywane **płatami bikubicznymi Coonsa**, choć płat taki może być określony przez krzywe dowolnego stopnia.

Dla  $k = 2$  płat jest określony przy użyciu wielomianów piątego stopnia,

$$\begin{aligned} \mathbf{H}_{00}(\mathbf{t}) &= \mathbf{B}_0^5(\mathbf{t}) + \mathbf{B}_1^5(\mathbf{t}) + \mathbf{B}_2^5(\mathbf{t}), & \mathbf{H}_{10}(\mathbf{t}) &= \mathbf{B}_3^5(\mathbf{t}) + \mathbf{B}_4^5(\mathbf{t}) + \mathbf{B}_5^5(\mathbf{t}), \\ \mathbf{H}_{01}(\mathbf{t}) &= \frac{1}{5}\mathbf{B}_1^5(\mathbf{t}) + \frac{2}{5}\mathbf{B}_2^5(\mathbf{t}), & \mathbf{H}_{11}(\mathbf{t}) &= -\frac{2}{5}\mathbf{B}_3^5(\mathbf{t}) - \frac{1}{5}\mathbf{B}_4^5(\mathbf{t}), \\ \mathbf{H}_{02}(\mathbf{t}) &= \frac{1}{20}\mathbf{B}_2^5(\mathbf{t}), & \mathbf{H}_{12}(\mathbf{t}) &= \frac{1}{20}\mathbf{B}_3^5(\mathbf{t}), \end{aligned}$$

w związku z czym płaty Coonsa klasy  $C^2$  są nazywane **dwupiętnymi płatami Coonsa**.

Płaty Coonsa (bikubiczne i dwupiętne) mogą być określone za pomocą krzywych wielomianowych albo sklejanych. W pierwszym przypadku krzywe te są krzywymi Béziera, a za dziedzinę płata przyjmuje się kwadrat  $[0, 1]^2$ . Poszczególne krzywe mogą mieć różne stopnie.

Dziedzina płatów określonych za pomocą krzywych sklejanych (o reprezentacji B-sklejanej) może być dowolnym prostokątem  $[a, b] \times [c, d]$ . Krzywe określające płat mogą mieć reprezentacje różnych stopni, reprezentacje te mogą również mieć różne ciągi węzłów (ale muszą mieć identyczne dziedziny, wyznaczone przez węzły brzegowe).

Biblioteka libmultibs zawiera procedury wyznaczające reprezentację Béziera lub B-sklejaną płata Coonsa na podstawie określających go krzywych, a także procedury szybkiego tablicowania płatów Coonsa razem z pochodnymi; procedury te znalazły zastosowanie w bibliotekach libg1hole i libg2hole.

### 7.1.7 Nazwy procedur i parametrów

Jako ułatwienie dla użytkownika jest pomyślany sposób nazywania procedur (których nazwy mają ułatwić odgadnięcie spełnianej funkcji) i ich parametrów formalnych (które, jeśli mają taką samą nazwę w różnych procedurach, to oznaczają to samo).

Każda procedura i makro przeznaczone do wywoływania jako procedura w bibliotece libmultibs ma nazwę zaczynającą się od prefiksu mbs\_.

Jeśli bezpośrednio po prefiksie występuje ciąg znaków multi, to procedura służy do przetwarzania jednej lub wielu krzywych (liczba krzywych jest wartością parametru, który ma nazwę ncurves).

Końcówka nazwy składa się z dwóch części. Pierwsza część może być pusta, jeśli po prefiksie jest „multi”, albo określa rodzaj krzywej lub płata przetwarzanego przez daną procedurę lub makro. Litera C oznacza krzywą zaś P płat. Cyfra określa wymiar przestrzeni (np. 2 oznacza płaszczyznę), w której leży krzywa lub płat. Litera R po cyfrze oznacza krzywą lub płat wymierny w reprezentacji jednorodnej. **Uwaga:** punkty kontrolne mają wtedy o jedną współrzędną więcej. Druga część końcówki jest literą f albo d i wskazuje odpowiednio precyzję (pojedynczą, float, albo podwójną, double<sup>1</sup>) arytmetyki zmiennopozycyjnej, w której procedura otrzymuje dane i wyprowadza wyniki.

Środkowa część nazwy określa funkcję wykonywaną przez procedurę. Makra i procedury o tej samej nazwie różnią się zastosowaniem — uniwersalnym (jeśli jest przedrostek „multi”) albo do krzywych lub płatów w przestrzeni o wymiarze określonym przez prefiks. Lista tych nazw zawiera m.in.

deBoor — obliczanie punktów krzywych i płatów B-sklejanych za pomocą algorytmu de Boora.

<sup>1</sup>W „poważnych” zastosowaniach należy stosować tylko podwójną precyzję, chyba, że i ona nie wystarczy.

deBoorDer — obliczanie punktów i pochodnych krzywych i płatów B-sklejanych za pomocą algorytmu de Boora.

BCHorner — obliczanie punktów krzywych i płatów Béziera za pomocą schematu Hornera.

BCHornerDer — obliczanie punktów i pochodnych krzywych i płatów Béziera za pomocą schematu Hornera.

BCFrenet — obliczanie krzywizn i wektorów układu Freneta w danym punkcie krzywej Béziera.

BCHornerNv — obliczanie wektora normalnego w danym punkcie płata Béziera.

KnotIns — wstawianie (jednego) węzła przy użyciu algorytmu Boehma.

KnotRemove — usuwanie (jednego) węzła.

\*Oslo\* — procedury związane z wstawianiem i usuwaniem (w jednym kroku) wielu węzłów przy użyciu algorytmu Oslo.

MaxKnotIns — wstawianie węzłów w celu otrzymania reprezentacji B-sklejanej z wszystkimi węzłami wewnętrznymi o krotności  $n + 1$  i brzegowymi o krotności  $n$  lub  $n + 1$ , czyli reprezentacji kawałkami Béziera.

BisectB — podział krzywych Béziera na łuki związany z podziałem dziedziny na dwa odcinki o tej samej długości, algorytmem de Casteljau.

DivideB — podział krzywych Béziera na łuki związany z podziałem dziedziny na dwa odcinki o dowolnych długościach, algorytmem de Casteljau.

BCDegElev — podwyższanie stopnia krzywych i płatów Béziera.

BSDegElev — podwyższanie stopnia krzywych i płatów B-sklejanych.

MultBez — mnożenie wielomianów i krzywych Béziera.

MultBS — mnożenie funkcji i krzywych sklejanych.

BezNormal — wyznaczanie płata Béziera opisującego wektor normalny danego płata Béziera.

BSCubicInterp — konstrukcja B-sklejanych interpolacyjnych krzywych trzeciego stopnia.

ConstructApproxBS — konstrukcja B-sklejanych krzywych aproksymacyjnych.

Closed — procedura, w której nazwie występuje to słowo służy do przetwarzania krzywych zamkniętych.

Parametry formalne procedur i makr mają następujące nazwy:

spdimen — określa wymiar  $d$  przestrzeni, w której leżą krzywe, czyli liczbę współrzędnych każdego punktu tej przestrzeni. Jeśli końcówka nazwy procedury lub makra zawiera literę R, która wskazuje na obiekt wymierny, to punkty kontrolne mają o 1 więcej współrzędną.

`degree` — określa stopień reprezentacji krzywej. Stopień płata ze względu na każdy z jego parametrów jest określany za pomocą parametrów, które mają nazwy `degreeu` i `degreev`.

`lastknot` — określa liczbę  $N$ , która jest indeksem ostatniego węzła, a zatem ciąg węzłów składa się z  $N + 1$  węzłów.

`knots` — tablica liczb zmiennopozycyjnych, w której podaje się węzły. Dwie tablice z ciągami węzłów składającymi się na reprezentację płata B-sklejanego przekazuje się przy użyciu parametrów o nazwach `knotsu` i `knotsv`.

`ctlpoints` — tablica punktów kontrolnych. W przypadku, gdy wymiar  $d$  przestrzeni jest równy 1 (procedura lub makro służy do przetwarzania funkcji skalarnych), parametr poprzez który przekazuje się odpowiednią tablicę nazywa się `coeff`.

`pitch` — podziałka tablicy punktów kontrolnych, tj. różnica indeksów pierwszych współrzędnych pierwszych punktów kolejnych łamanych kontrolnych lub kolumn siatki kontrolnej w tablicy `ctlpoints`. Zawsze tablice takie są traktowane jako tablice *liczb zmiennopozycyjnych*, a zatem jednostka podziałki jest to długość reprezentacji zmiennopozycyjnej jednej liczby (nawet jeśli parametr formalny `ctlpoints` jest typu np. `point3f`).

Jeśli parametry procedury służą do przekazania dwóch reprezentacji, np. procedura na podstawie reprezentacji wejściowej konstruuje reprezentację wynikową, to odpowiednie parametry mają nazwy rozszerzone o fragment `in` albo `out`. Parametry służące do przekazania danych wejściowych występują w listach parametrów *przed* parametrami opisującymi dane wyjściowe.

## 7.2 Operacje na ciągach węzłów

W tym punkcie są zamieszczone opisy procedur dokonujących różnych pomocniczych działań na ciągach węzłów, takich jak wyszukiwanie, tworzenie nowych ciągów i obliczanie ich długości (przed utworzeniem, co przydaje się w celu zarezerwowania tablic o odpowiedniej długości).

### 7.2.1 Przeszukiwanie ciągu węzłów

Ciągi węzłów  $u_0, \dots, u_N$  przekazywane w tablicach jako parametry wszystkich procedur muszą być niemalejące i muszą spełniać warunek  $u_n < u_{N-n}$ . Odpowiedzialność za spełnienie tych warunków spoczywa na procedurach wywołujących (bo każdorazowe sprawdzanie zabiera czas).

```
int mbs_KnotMultiplicityf ( int lastknot, const float *knots,
                           float t );
```

Procedura `mbs_KnotMultiplicityf` otrzymuje tablicę `knots`, która zawiera niemalejący ciąg liczb o długości  $N + 1$ , gdzie  $N$  jest wartością parametru `lastknot`. Wartością procedury jest liczba wystąpień liczby  $t$  w tym ciągu.

```
int mbs_FindKnotIntervalf ( int degree,
                           int lastknot, const float *knots,
                           float t, int *mult );
```

Procedura `mbs_FindKnotIntervalf` otrzymuje tablicę `knots`, która zawiera niemalejący ciąg liczb o długości  $N + 1$  (liczba  $N$  jest wartością parametru `lastknot`). Jeśli parametr `degree` ma wartość  $-1$ , to wartością procedury jest indeks  $k$  do tablicy, taki że  $knots[k] \leq t < knots[k + 1]$ . Może też być  $-1$  jeśli  $t < knots[0]$  lub  $N$  jeśli  $t \geq knots[N]$ .

Jeśli wartość  $n$  parametru `degree` jest nieujemna, to najmniejsza zwracana wartość procedury może być równa  $n$ , a największa  $N - n - 1$ . Następuje domniemanie, że procedura jest wywołana w celu wyznaczenia przedziału między węzłami, któremu odpowiada pewien wielomian lub łuk wielomianowy opisujący funkcję lub krzywą sklejaną stopnia  $n$ . Po znalezieniu numeru tego łuku można obliczać jego punkty (np. algorytmem de Boora). W ten sposób jeśli  $t \notin [u_n, u_{N-1})$ , to będą obliczane punkty pierwszego albo ostatniego łuku opisującego krzywą.

Parametr `mult` służy do przekazania krotności węzła  $t$ . Jeśli jego wartością jest wskaźnik pusty (NULL), to parametr ten jest ignorowany. W przeciwnym razie jeśli  $t = u_k$  (dla  $k$  równego wartości procedury), to zmienna `*mult` otrzymuje wartość równą liczbie wystąpień liczby  $t$  w podanym ciągu węzłów.



### 7.2.2 Tworzenie ciągów węzłów

```
int mbs_NumKnotIntervalsf ( int degree, int lastknot,
                           const float *knots );
```

Procedura `mbs_NumKnotIntervalsf` oblicza, z ilu przedziałów między węzłami składa się dziedzina funkcji (albo krzywych) sklejanych stopnia `degree`, określonych dla ciągu węzłów o długości `*lastknot+1`, podanego w tablicy `knots`.

```
int mbs_LastknotMaxInsf ( int degree, int lastknot,
                          const float *knots,
                          int *numknotintervals );
```

Procedura `mbs_LastknotMaxInsf` oblicza indeks ostatniego węzła reprezentacji krzywych, która zostanie utworzona przez procedurę `mbs_MaxKnotInsf`.

```
int mbs_NumMaxKnotsf ( int degree, int lastknot,
                       const float *knots );
```

Procedura `mbs_NumMaxKnotsf` oblicza długość ciągu węzłów potrzebną do reprezentowania w lokalnych bazach Bernsteina stopnia `degree` funkcji lub krzywej sklejanej określonej dla ciągu węzłów o długości `lastknot+1`, podanego w tablicy `knots`.

```
void mbs_SetKnotPatternf ( int lastinknot, const float *inknots,
                           int multipl,
                           int *lastoutknot, float *outknots );
```

Procedura `mbs_SetKnotPatternf` służy do wygenerowania ciągu węzłów, który składa się z liczb podanych w tablicy `inknots` (o długości `lastinknot + 1`), ale w którym każdy węzeł ma krotność `multipl`.

Ciąg węzłów jest wpisywany do tablicy `outknots`, a indeks ostatniego węzła jest zwracana poprzez parametr `*lastoutknot`.

### 7.2.3 Reparametryzacja krzywych i płatów

```
void mbs_TransformAffKnotsf ( int degree, int lastknot,
                             const float *inknots,
                             float a, float b, float *outknots );
```

Procedura `mbs_TransformAffKnotsf` dokonuje przekształcenia afinicznego dziedziny krzywej B-sklejanej, tj. oblicza ciąg węzłów związany z nową dziedziną, co jest równoważne reparametryzacji krzywej. „Stara” dziedziną jest przedział  $[u_n, u_{N-n}]$ , zaś nową przedział  $[a, b]$ . Parametr `degree` określa stopień  $n$  krzywej, parametr `lastknot` określa indeks ostatniego węzła, tablica `inknots` zawiera ciąg węzłów  $u_0, \dots, u_N$ .

Parametry `a`, `b` określają brzegi przedziału  $[a, b]$ , przy czym powinien być spełniony warunek  $a < b$  (procedura go nie sprawdza). Nowy ciąg węzłów procedura wpisuje do tablicy `outknots`.

Parametry `inknots` i `outknots` mogą wskazywać dwie różne (rozłączne) tablice o długości  $N + 1$ , lub też tę samą tablicę. W drugim przypadku procedura dokonuje zmiany reprezentacji (reparametryzacji) krzywej „w miejscu”.

```
void mbs_multiReverseBSCurvef ( int degree, int lastknot,
                                float *knots,
                                int ncurves, int spdimen,
                                int pitch, float *ctlpoints );
```

Procedura `mbs_multiReverseBSCurvef` dokonuje reparametryzacji krzywych B-sklejanych stopnia  $n$ , polegającej na podstawieniu parametru  $-t$  w miejsce  $t$ .

Parametr `degree` określa stopień  $n$  krzywych. Parametry `lastknot` i `knots` opisują ciąg węzłów, na którym oparta jest reprezentacja krzywych. Parametr `ncurves` określa liczbę krzywych, a `spdimen` wymiar przestrzeni, w której leżą krzywe.

Jeśli parametr `knots` jest równy `NULL`, to procedura tylko odwraca kolejność punktów kontrolnych. Dzięki temu można jej użyć do „odwracania” krzywych (lub płatów) Béziera. W tym przypadku wartość parametru `lastknot` jest ignorowana (krzywa stopnia  $n$  ma  $n + 1$  punktów kontrolnych).

Parametr `pitch` określa podziałkę tablicy `ctlpoints`, w której są podane punkty kontrolne krzywych.

Obliczenie realizowane jest „w miejscu” i polega na zmianie znaku na przeciwny i odwróceniu kolejności węzłów oraz na odwróceniu kolejności punktów kontrolnych każdej krzywej. Obliczenie to jest wykonywane bez błędów zaokrągleń.

### 7.2.4 Modyfikowanie węzłów

```
int mbs_SetKnotf ( int lastknot, float *knots,
                  int knotnum, int mult, float t );
```

Procedura `mbs_SetKnotf` zmienia węzeł w danym ciągu, z zachowaniem uporządkowania. Parametry: `lastknot` — numer ostatniego węzła w ciągu, `knots` — wskaźnik tablicy z ciągiem węzłów, `knotnum` — indeks zmienianego węzła w ciągu, `mult` — krotność (nową wartość otrzymują węzły o indeksach `knotnum-i+1 ... knotnum`), `t` — nowa wartość węzła.

Po zmianie węzłów ciąg jest sortowany. Wartością procedury jest nowy indeks węzła, a dokładniej liczba  $k$ , taka że  $t = u_k < u_{k+1}$ .

Wartość procedury  $-1$  oznacza błędną wartość parametru `knotnum`; musi ona być od 0 do `lastknot`.

```
int mbs_SetKnotClosedf ( int degree, int lastknot, float *knots,
                        float T, int knotnum, int mult, float t );
```

Procedura `mbs_SetKnotClosedf` zmienia węzeł w danym ciągu, z zachowaniem uporządkowania i okresowości wymaganej przez reprezentację zamkniętej krzywej B-sklejanej. Parametry: `degree` — stopień krzywej, `lastknot` — numer ostatniego węzła w ciągu, `knots` — wskaźnik tablicy z ciągiem węzłów, `T` — długość dziedziny krzywej (po zmianie ma być `knots[lastknot-degree]-knots[degree]`) `knotnum` — indeks zmienianego węzła w ciągu, `mult` — krotność (nową wartość otrzymają węzły o indeksach `knotnum-i+1 ... knotnum`), `t` — nowa wartość węzła.

Po zmianie węzłów ciąg jest sortowany. Wartością procedury jest nowy indeks węzła, a dokładniej liczba  $k$ , taka że  $t = u_k < u_{k+1}$ .

Wartość procedury  $-1$  oznacza błędną wartość parametru `knotnum`; musi ona być od 0 do `lastknot`, lub `lastknot`, która musi być większa niż  $3 \cdot \text{degree}$ .

### 7.2.5 Sprawdzanie poprawności

```
boolean mbs_ClosedKnotsCorrectf ( int degree, int lastknot,
                                  float *knots,
                                  float T, int K, float tol );
```

Procedura `mbs_ClosedKnotsCorrectf` sprawdza poprawność ciągu węzłów przeznaczonego do reprezentowania zamkniętej krzywej B-sklejanej. Poprawny ciąg musi być niemalejący i spełniać warunek  $u_{i+K} = u_i + T$  dla  $i = 1, \dots, n$ , gdzie  $K = N - 2n$ ,  $N > 3n$ . Krotności węzłów nie mogą przekraczać stopnia  $n$ . Parametr `tol` określa tolerancję (tj. dopuszczalną różnicę  $u_{i+K} - T - u_i$ ); musi to być mała liczba dodatnia, nie może być 0 z powodu błędów zaokrągleń.

### 7.3 Obliczanie wartości funkcji B-sklejanych

Wyznaczenie wartości funkcji B-sklejanych może być potrzebne do rozwiązywania zadań interpolacyjnych. Funkcje te są obliczane na podstawie wzorów (7.10) i (7.11).

```
void mbs_deBoorBasisf ( int degree, int lastknot,  
                        const float *knots,  
                        float t, int *fnz, int *nnz, float *bfv );
```

Procedura `mbs_deBoorBasisf` oblicza wartości funkcji B-sklejanych stopnia  $n$  (parametr `degree`) w punkcie  $t$ . Funkcje są określone przez podanie niemalejącego ciągu węzłów  $u_0, \dots, u_M$  w tablicy `knots`. Liczba węzłów jest równa `lastknot+1`. Wartość parametru  $t$  musi być liczbą z przedziału  $[u_n, u_{N-n}]$ .

Obliczone wartości funkcji B-sklejanych są umieszczane w tablicy `bfv`, przy czym wartość parametru `*fnz` na wyjściu jest równa numerowi pierwszej funkcji różnej od 0 dla podanego  $t$ ; tablica `bfv` musi mieć długość `degree+1`.

Parametr `*nnz` służy do przekazania informacji o liczbie funkcji bazowych różnych od 0 w punkcie  $t$ . Zawartość miejsc w tablicy `bfv` zaczynając od miejsca `*nnz` jest nieokreślona (ale procedura może wpisywać pośrednie wyniki swoich obliczeń do pierwszych `degree + 1` miejsc w tej tablicy).

## 7.4 Wyznaczanie punktów krzywych i płatów

### 7.4.1 Algorytm de Boora

Algorytm de Boora obliczania punktu  $s(t)$  krzywej  $s$  danej wzorem (7.9), dla  $t \in [u_k, u_{k+1})$ ,  $k \in \{n, \dots, N - n - 1\}$ , polega na rekurencyjnym obliczeniu punktów  $d_i^{(j)}$  dla  $j = 1, \dots, n - r$  oraz  $i = k - n, \dots, k - r$ , na podstawie wzoru

$$d_i^{(j)} = (1 - \alpha_i^{(j)})d_{i-1}^{(j-1)} + \alpha_i^{(j)}d_i^{(j-1)}, \quad (7.16)$$

gdzie  $\alpha_i^{(j)} = \frac{t - u_i}{u_{i+n+1-j} - u_i}$ .

Punkty  $d_i^{(0)} = d_i$  są punktami kontrolnymi krzywej, zaś liczba  $r$  jest liczbą wystąpień (krotnością) liczby  $t$  w ciągu węzłów  $u_0, \dots, u_N$ .

```
int mbs_multideBoorf ( int degree, int lastknot,
                      const float *knots,
                      int ncurves, int spdimen,
                      int pitch, const float *ctlpoints,
                      float t, float *cpoints );
```

Procedura `mbs_multideBoorf` jest implementacją algorytmu de Boora obliczania punktu na krzywej B-sklejanej. Dane dla procedury to `ncurves` krzywych B-sklejanych stopnia `degree`, położonych w przestrzeni o wymiarze `spdimen`. Każda z tych krzywych jest określona za pomocą tego samego niemalejącego ciągu `lastknot+1` węzłów przechowywanych w tablicy `knots`.

Łamane kontrolne są przekazywane w tablicy `ctlpoints`; każda z nich jest opisana przez  $(\text{lastknot} - \text{degree}) * \text{spdimen}$  liczb zmiennopozycyjnych, przy czym początek opisu kolejnej łamanej jest `pitch` miejsc po poprzedniej.

Parametr `t` ma wartość parametru, dla którego należy obliczyć punkt na każdej z krzywych. Współrzędne tych punktów są wpisywane do tablicy `cpoints`, która musi mieć długość co najmniej `ncurves * spdimen`.

Wartością procedury jest liczba  $n - r$ , tj. różnica stopnia krzywych i krotności liczby  $t$  w ciągu węzłów. Jeśli liczba ta jest nieujemna, to określa minimalną klasę ciągłości krzywych w otoczeniu  $t$ .

```
#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) ...
```

Cztery makra, które wywołują procedurę `mbs_multideBoorf` w celu obliczenia wartości *jednej* skalarnej funkcji sklejanej lub B-sklejanej krzywej płaskiej, prze-

strzennej lub czterowymiarowej. Parametry makra powinny spełniać warunki podane w opisie procedury `mbs_multideBoorf`.

```
void mbs_deBoorC2Rf ( int degree,
                     int lastknot, const float *knots,
                     const point3f *ctlpoints, float t,
                     point2f *cpoint );
```

Procedura `mbs_deBoorC2Rf` oblicza punkt wymiernej B-sklejanej krzywej płaskiej stopnia `degree`, określonej dla niemalejącego ciągu `lastknot+1` węzłów podanych w tablicy `knots`. W tablicy `ctlpoints` należy podać punkty kontrolne *krzywej jednorodnej* (położonej w  $\mathbb{R}^3$ ).

Argument krzywej jest wartością parametru `t` i musi należeć do przedziału `[knots[degree], knots[lastknot-degree]]`. Współrzędne obliczonego punktu krzywej są przekazywane za pomocą parametru `cpoint`.

Właściwe obliczenie wykonuje procedura `mbs_multideBoorf`.

```
void mbs_deBoorC3Rf ( int degree,
                     int lastknot, const float *knots,
                     const point4f *ctlpoints, float t,
                     point3f *cpoint );
```

Procedura `mbs_deBoorC3Rf` oblicza punkt wymiernej krzywej B-sklejanej stopnia `degree` w przestrzeni trójwymiarowej, określonej dla niemalejącego ciągu `lastknot+1` węzłów podanych w tablicy `knots`.

W tablicy `ctlpoints` należy podać punkty kontrolne *krzywej jednorodnej* (położonej w  $\mathbb{R}^4$ ). Argument krzywej jest wartością parametru `t` i musi należeć do przedziału `[knots[degree], knots[lastknot-degree]]`. Współrzędne obliczonego punktu krzywej są przekazywane za pomocą parametru `cpoint`.

Właściwe obliczenie wykonuje procedura `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 );
```

Procedura `mbs_deBoorP3f` oblicza punkt płata B-sklejanego w przestrzeni trójwymiarowej. Stopień płata ze względu na parametry `u` i `v` są równe odpowiednio `degreeu` i `degreev`. Ciąg węzłów „`u`” o długości `lastknotu+1` jest podany w tablicy `knotsu`, podobnie ciąg węzłów „`v`” o długości `lastknotv+1` jest podany w tablicy `knotsv`.

Punkty kontrolne płata są podane w tablicy `ctlpoints`, przy czym ich kolejność jest następująca: najpierw są punkty z pierwszej kolumny siatki, potem z drugiej itd, przy czym kolumna składa się z `lastknotv-degreev` punktów.

Parametry `u` i `v` określają punkt w dziedzinie, dla którego należy obliczyć punkt płata. Współrzędne tego punktu są zwracane za pomocą parametru `ppoint`.

Właściwe obliczenie polega na wywołaniu procedury `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 );
```

Procedura `mbs_deBoorP3Rf` wyznacza punkt wymiennego płata B-sklejanego w przestrzeni trójwymiarowej. Stopień płata ze względu na parametry `u` i `v` są równe odpowiednio `degreeu` i `degreev`. Ciąg węzłów „u” o długości `lastknotu+1` jest podany w tablicy `knotsu`, podobnie ciąg węzłów „v” o długości `lastknotv+1` jest podany w tablicy `knotsv`.

W tablicy `ctlpoints` należy podać punkty kontrolne *płata jednorodnego*, przy czym ich kolejność jest następująca: najpierw są punkty z pierwszej kolumny siatki, potem z drugiej itd, przy czym kolumna składa się z `lastknotv-degreev` punktów.

Parametry `u` i `v` określają punkt w dziedzinie, dla którego należy obliczyć punkt płata. Współrzędne tego punktu są zwracane za pomocą parametru `ppoint`.

Właściwe obliczenie polega na wywołaniu procedury `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 );
```

Procedura `mbs_deBoorP4f` oblicza punkt płata B-sklejanego w przestrzeni czterowymiarowej. Stopień płata ze względu na parametry `u` i `v` są równe odpowiednio `degreeu` i `degreev`. Ciąg węzłów „u” o długości `lastknotu+1` jest podany w tablicy `knotsu`, podobnie ciąg węzłów „v” o długości `lastknotv+1` jest podany w tablicy `knotsv`.

Punkty kontrolne płata są podane w tablicy `ctlpoints`, przy czym ich kolejność jest następująca: najpierw są punkty z pierwszej kolumny siatki, potem z drugiej itd, przy czym kolumna składa się z `lastknotv-degreev` punktów.

Parametry `u` i `v` określają punkt w dziedzinie, dla którego należy obliczyć punkt płata. Współrzędne tego punktu są zwracane za pomocą parametru `ppoint`.

Właściwe obliczenie polega na wywołaniu procedury `mbs_multideBoorf`.

Pochodna krzywej sklejanej  $s$  danej wzorem (7.9) w punkcie  $t$  wyraża się wzorem

$$s'(t) = \frac{n}{u_{k+1} - u_k} (d_{k-r}^{(n-r-1)} - d_{k-r-1}^{(n-r-1)}), \quad (7.17)$$

w którym występują punkty  $d_{k-r}^{(n-r-1)}$  i  $d_{k-r-1}^{(n-r-1)}$  obliczane w algorytmie de Boora. Opisane niżej procedury realizują ten algorytm uzupełniony o obliczenie pochodnej.

```
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 );
```

Procedura `mbs_multideBoorDerf` służy do obliczenia punktów na `ncurves` krzywych B-sklejanych stopnia `degree`, położonych w przestrzeni o wymiarze `spdimen`. Dodatkowo procedura oblicza wektory pochodnych tych krzywych dla podanego argumentu  $t$ .

Sposób reprezentowania krzywych jest taki sam jak sposób reprezentowania krzywych dla potrzeb procedury `mbs_multideBoorf`. Obliczone współrzędne punktów krzywych są wpisywane (w taki sam sposób) do tablicy `cpoints`, a współrzędne wektorów pochodnych do tablicy `dervect`.

Jeśli wartość  $t$  parametru  $t$  jest równa węzłowi o krotności `degree` lub większej, to procedura oblicza wartości pochodnych prawostronnych krzywych w punkcie  $t$ , z wyjątkiem przypadku, gdy liczba  $t$  określa koniec dziedziny krzywych (to znaczy gdy  $t = \text{knots}[\text{lastknot} - \text{degree}]$ ). W tym przypadku obliczane są pochodne lewostronne. Wartością procedury jest różnica stopnia krzywych i krotności liczby  $t$  w ciągu węzłów, która określa klasę ciągłości krzywych w otoczeniu punktu  $t$ .

```
#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) ...
```



Cztery makra wywołujące procedurę `mbs_multideBoorDerf` w celu obliczenia wartości *jednej* funkcji skleianej lub punktu krzywej B-sklejanej wraz pochodną w punkcie  $t$ . Parametry muszą spełniać warunki podane w opisie procedury `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 );
```

Procedura `mbs_multideBoorDer2f` oblicza punkty  $s_i(t)$  oraz wektory  $s'_i(t)$  i  $s''_i(t)$  krzywych B-sklejanych  $s_i$  stopnia  $n$  dla ustalonego  $t$ .

Parametry wejściowe: `degree` — stopień  $n$ , `lastknot` — numer  $N$  ostatniego węzła, `knots` — tablica węzłów, `ncurves` — liczba krzywych, `pitch` — podziałka tablicy punktów kontrolnych, `ctlpoints` — tablica punktów kontrolnych,  $t$  — liczba  $t$ .

Parametry wyjściowe:  $p$  — tablica, do której procedura wstawia punkty  $s_i(t)$ ,  $d1$  — tablica, do której procedura wstawia wektory  $s'_i(t)$ ,  $d2$  — tablica, do której procedura wstawia wektory  $s''_i(t)$ . Podziałka wszystkich tych tablic jest równa wymiarowi przestrzeni, `spdimen`.

Wartością procedury jest liczba  $n-r$ , gdzie  $r$  oznacza krotność (liczbę wystąpień) liczby  $t$  w ciągu węzłów. Określa ona klasę ciągłości krzywych  $s_i$  w otoczeniu punktu  $t$ .

Jeśli krzywa lub któraś pochodna jest nieciągła w otoczeniu  $t$ , to obliczony punkt lub wektor jest równy granicy lewostronnej (np.  $\lim_{x \rightarrow t^-} s'(x)$ ).

```
#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) ...
```

Powyższe makra służą do wywoływania procedury `mbs_multideBoorDer2f` w sytuacji, gdy należy obliczyć punkt i pochodne rzędu 1 i 2 jednej krzywej B-sklejanej położonej w przestrzeni o wymiarze 1, 2, 3 lub 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 );
```

Procedura `mbs_multideBoorDer3f` oblicza punkty  $s_i(t)$  oraz wektory  $s'_i(t)$ ,  $s''_i(t)$  i  $s'''_i(t)$  krzywych B-sklejanych  $s_i$  stopnia  $n$  dla ustalonego  $t$ .

Parametry wejściowe: `degree` — stopień  $n$ , `lastknot` — numer  $N$  ostatniego węzła, `knots` — tablica węzłów, `ncurves` — liczba krzywych, `pitch` — podziałka tablicy punktów kontrolnych, `ctlpoints` — tablica punktów kontrolnych, `t` — liczba  $t$ .

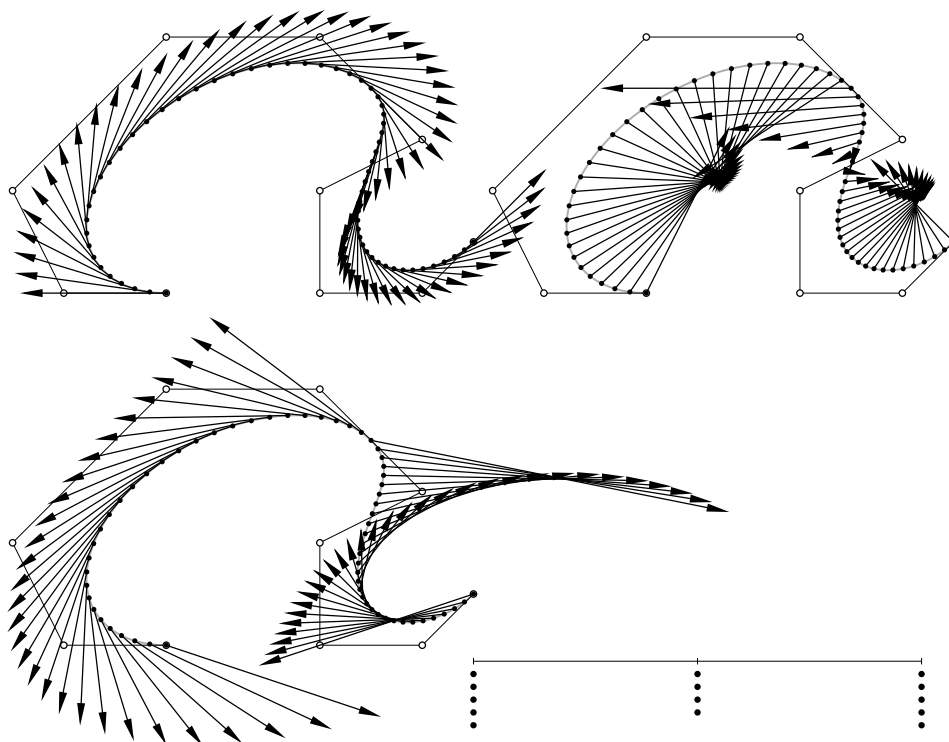
Parametry wyjściowe: `p` — tablica, do której procedura wstawia punkty  $s_i(t)$ , `d1` — tablica, do której procedura wstawia wektory  $s'_i(t)$ , `d2` — tablica, do której procedura wstawia wektory  $s''_i(t)$ , `d3` — tablica, do której procedura wstawia wektory  $s'''_i(t)$ . Podziałka wszystkich tych tablic jest równa wymiarowi przestrzeni, `spdimen`.

Wartością procedury jest liczba  $n-r$ , gdzie  $r$  oznacza krotność (liczbę wystąpień) liczby  $t$  w ciągu węzłów. Określa ona klasę ciągłości krzywych  $s_i$  w otoczeniu punktu  $t$ .

Jeśli krzywa lub któraś pochodna jest nieciągła w otoczeniu  $t$ , to obliczony punkt lub wektor jest równy granicy lewostronnej ( $\lim_{x \rightarrow t^-} s'(x)$ ).

```
#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) ...
```

Powyższe makra służą do wywoływania procedury `mbs_multideBoorDer3f` w sytuacji, gdy należy obliczyć punkt i pochodne rzędu 1, 2 i 3 jednej krzywej B-sklejanej położonej w przestrzeni o wymiarze 1, 2, 3 lub 4.



Rys. 7.1. Wektory pochodnych rzędu 1, 2 i 3 krzywej B-sklejanej piątego stopnia, obliczone przez procedury `mbs_multideBoorDerf`, `mbs_multideBoorDer2f` i `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 );
```

Procedura `mbs_deBoorDerPf` służy do obliczenia punktu płata B-sklejanego, odpowiadającego punktowi dziedziny o współrzędnych  $u$  i  $v$ , oraz wektorów pochodnych cząstkowych w tym punkcie.

```
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 );
```

Procedura `mbs_deBoorDer2Pf` służy do obliczenia punktu płata B-sklejanego, odpowiadającego punktowi dziedziny o współrzędnych  $u$  i  $v$ , oraz wektorów pochodnych cząstkowych pierwszego i drugiego rzędu w tym punkcie.

```
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 );
```

Procedura `mbs_deBoorDer3Pf` służy do obliczenia punktu płata B-sklejanego, odpowiadającego punktowi dziedziny o współrzędnych  $u$  i  $v$ , oraz wektorów pochodnych cząstkowych pierwszego, drugiego i trzeciego rzędu w tym punkcie.

#### 7.4.2 Schemat Hornera dla krzywych i płatów Béziera

Schemat Hornera jest algorytmem obliczania wartości wielomianu (lub punktu krzywej) o złożoności proporcjonalnej do stopnia (algorytmy de Casteljau i de Boora mają złożoność proporcjonalną do kwadratu stopnia). Aby go stosować do krzywych B-sklejanych (co opłaca się wtedy, gdy obliczamy wiele punktów), trzeba przejść do reprezentacji kawałkami Béziera, przy użyciu procedury `mbs_multiMaxKnotInsf` (zobacz p. 7.7.4).

```
void mbs_multiBCHornerf ( int degree, int ncurves,
                          int spdimen, int pitch,
                          const float *ctlpoints,
                          float t, float *cpoints );
```

Procedura `mbs_multiBCHornerf` oblicza punkty na  $n$  krzywych Béziera stopnia `degree` w przestrzeni o wymiarze `spdimen`. Punkty kontrolne tych krzywych są podane w tablicy `ctlpoints`. Współrzędne punktów kontrolnych każdej

krzywej są upakowane w tej tablicy, przy czym odległość między początkami reprezentacji kolejnych dwóch krzywych jest wartością parametru pitch. Parametr  $t$  określa parametr krzywych, dla którego mają być obliczone punkty. Obliczone punkty są wstawiane do tablicy cpoints, która musi mieć długość co najmniej  $n_{\text{curves}} * \text{spdimen}$ .

```
#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) ...
```

Cztery makra, które wywołują procedurę `mbs_multiBCHornerf` w celu obliczenia wartości wielomianu danego w bazie Bernsteina albo punktu na jednej krzywej Béziera w przestrzeni dwu-, trój- i czterowymiarowej.

```
void mbs_BCHornerC2Rf ( int degree,
                        const point3f *ctlpoints,
                        float t, point2f *cpoint );
void mbs_BCHornerC3Rf ( int degree,
                        const point4f *ctlpoints,
                        float t, point3f *cpoint );
```

Powyższe procedury obliczają punkt płaskiej lub trójwymiarowej wymiernej krzywej Béziera, za pomocą schematu Hornera (zastosowanego do reprezentacji jednorodnej).

```
void mbs_FindBezPatchDiagFormf ( int degreeu, int degreev,
                                int spdimen, const float *cpoints,
                                int k, int l, float u, float v,
                                float *dfcp );
```

Procedura `mbs_FindBezPatchDiagFormf` wyznacza formę diagonalną stopnia  $(k, l)$  płata Béziera  $p$  stopnia  $(n, m)$  w punkcie  $(u, v)$ ; jest to prostokątny płat Béziera stopnia  $(k, l)$ , który można otrzymać wykonując  $n - k$  kroków algorytmu de Casteljau na wierszach i  $m - l$  kroków na kolumnach; płat ten umożliwia obliczenie punktu płata  $p$  i jego pochodnych rzędu  $1, \dots, k$  ze względu na  $u$  i  $1, \dots, l$  ze względu na  $v$ . Zamiast algorytmu de Casteljau procedura używa szybszego schematu Hornera (procedury `mbs_multiBCHornerf`).

Wszystkie procedury obliczania punktu i pochodnych (a także krzywizn) płata Béziera opisane dalej w tym punkcie powinny być zrealizowane przy użyciu tej procedury, ale na razie tylko procedura `mbs_BCHornerDer3Pf` jest taka (i trzeba ją jeszcze dopracować).

```
void mbs_BCHornerPf ( int degreeu, int degreev, int spdimen,
                     const float *ctlpoints,
                     float u, float v, float *ppoint );
```

Procedura `mbs_BCHornerPf` oblicza punkt  $p(u,v)$  płata Béziera  $p$  stopnia  $(n,m)$ , położonego w przestrzeni o wymiarze  $d$ . Parametry `degreeu` i `degreev` określają stopień płata (ich wartościami są liczby  $n$  i  $m$ ). Parametr `spdimen` określa wymiar  $d$  przestrzeni, w której leży płat. W tablicy `ctlpoints` należy podać punkty kontrolne  $((n+1)(m+1)d)$  liczb).

Procedura umieszcza obliczony wynik (współrzędne punktu płata) w tablicy `ppoint`, która musi mieć długość co najmniej `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) ...
```

Powyższe makra służą do obliczania punktu płata Béziera w przestrzeni o wymiarze 1, 2, 3 i 4, za pomocą procedury `mbs_BCHornerPf`.

```
void mbs_BCHornerP3Rf ( int degreeu, int degreev,
                      const point4f *ctlpoints, float u, float v,
                      point3f *p );
```

Procedura `mbs_BCHornerP3Rf` oblicza punkt wymiennego płata Béziera w przestrzeni trójwymiarowej, reprezentowanego w postaci jednorodnej.

```
void mbs_multiBCHornerDerf ( int degree, int ncurves,
                             int spdimen, int pitch,
                             const float *ctlpoints,
                             float t, float *p, float *d );
```

Procedura `mbs_multiBCHornerDerf` oblicza za pomocą schematu Hornera punkty  $c_i(t)$  i wektory pochodnej  $c'_i(t)$  krzywych Béziera  $c_i$  położonych w przestrzeni o wymiarze  $d$ .

Parametry: `degree` — określa stopień krzywej, `ncurves` — liczbę krzywych, `spdimen` — wymiar  $d$  przestrzeni, `pitch` — podziałkę tablicy `ctlpoints` zawierającej punkty kontrolne. Parametr `t` ma wartość  $t$ .

Współrzędne punktów  $c_i(t)$  i wektorów  $c'_i(t)$  procedura wpisuje odpowiednio do tablic `p` i `d`, które muszą mieć długość co najmniej `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) ...

```

Powyższe makra służą do obliczenia punktu i pochodnej w punkcie  $t$  jednej krzywej Béziera w przestrzeni o wymiarze 1, 2, 3 lub 4, przez wywołanie procedury `mbs_multiBCHornerDerf`.

```

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 );

```

Procedury `mbs_BCHornerDerC2Rf` i `mbs_BCHornerDerC3Rf` obliczają punkt  $\mathbf{p}(t)$  i wektor  $\mathbf{p}'(t)$  wymiernej krzywej Béziera  $\mathbf{p}$  odpowiednio w przestrzeni dwu- i trój-wymiarowej.

Parametry: `degree` — stopień krzywej, `ctlpoints` — tablica punktów kontrolnych krzywej jednorodnej, `t` — liczba  $t$ . Procedury przypisują obliczony punkt  $\mathbf{p}(t)$  parametrowi `p`, a wektor  $\mathbf{p}'(t)$  parametrowi `d`.

```

void mbs_BCHornerDerPf ( int degreeu, int degreev, int spdimen,
                        const float *ctlpoints,
                        float u, float v,
                        float *p, float *du, float *dv );

```

Procedura `mbs_BCHornerDerPf` oblicza punkt  $\mathbf{p}(u,v)$  oraz pochodne cząstkowe  $\frac{\partial}{\partial u}\mathbf{p}(u,v)$  i  $\frac{\partial}{\partial v}\mathbf{p}(u,v)$  płata Béziera  $\mathbf{p}$  stopnia  $(n,m)$ , położonego w przestrzeni o wymiarze `d`.

Parametry: `degreeu`, `degreev` — określają stopień płata (liczby  $n$  i  $m$ ). Parametr `spdimen` określa wymiar  $d$  przestrzeni, tablica `ctlpoints` zawiera współrzędne punktów kontrolnych.

Wyniki (punkt płata i wektory pochodnych) procedura wpisuje do tablic `p`, `du` i `dv`, które muszą mieć długość co najmniej `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) \
    ...
```

Powyższe makra służą do obliczania punktów i pochodnych cząstkowych płatów Béziera położonych w przestrzeni o wymiarze 1, 2, 3 lub 4.

```
void mbs_BCHornerDerP3Rf ( int degreeu, int degreev,
                           const point4f *ctlpoints,
                           float u, float v,
                           point3f *p, vector3f *du, vector3f *dv );
```

Procedura `mbs_BCHornerDerP3Rf` oblicza punkt  $p(u,v)$  i wektory pochodnych cząstkowych wymiennego płata Béziera położonego w przestrzeni trójwymiarowej.

Parametry: `degreeu`, `degreev` — stopień płata ze względu na parametry  $u$  i  $v$ , `ctlpoints` — tablica punktów kontrolnych płata *jednorodnego*,  $u$ ,  $v$  — liczby  $u$  i  $v$ ,  $*p$ ,  $*du$ ,  $*dv$  — struktury, do których procedura wpisuje wyniki.

```
void mbs_multiBCHornerDer2f ( int degree, int ncurves,
                              int spdimen, int pitch,
                              const float *ctlpoints, float t,
                              float *p, float *d1, float *d2 );
```

Procedura `mbs_multiBCHornerDer2f` oblicza za pomocą schematu Hornera punkty  $c_i(t)$  i wektory pochodnych  $c'_i(t)$  i  $c''_i(t)$  krzywych Béziera  $c_i$  położonych w przestrzeni o wymiarze  $d$ .

Parametry: `degree` — określa stopień krzywej, `ncurves` — liczbę krzywych, `spdimen` — wymiar  $d$  przestrzeni, `pitch` — podziałkę tablicy `ctlpoints` zawierającej punkty kontrolne. Parametr  $t$  ma wartość  $t$ .

Obliczone współrzędne punktów  $c_i(t)$  i wektorów  $c'_i(t)$  i  $c''_i(t)$  procedura wpisuje odpowiednio do tablic  $p$ ,  $d1$  i  $d2$ , które muszą mieć długość co najmniej `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) ...
```



Powyższe makra służą do obliczania punktów i wektorów pochodnych pierwszego i drugiego rzędu jednej krzywej Béziera położonych w przestrzeni o wymiarze 1, 2, 3 lub 4, za pomocą procedury `mbs_multiBCHornerDer2f`.

```
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 );
```

Procedury `mbs_BCHornerDer2C2Rf` i `mbs_BCHornerDer2C3Rf` obliczają punkt  $\mathbf{p}(t)$  wymiernej krzywej Béziera odpowiednio w przestrzeni dwu- i trójwymiarowej i jej pochodne pierwszego i drugiego rzędu w punkcie  $t$ .

Parametry: `degree` — stopień krzywej, `ctlpoints` — tablica punktów kontrolnych wielomianowej krzywej jednorodnej,  $t$  — liczba  $t$ , `*p`, `*d1` i `*d2` — struktury, do których procedura przypisze odpowiednio punkt  $\mathbf{p}(t)$  i wektory  $\mathbf{p}'(t)$  i  $\mathbf{p}''(t)$ .

```
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 );
```

Procedura `mbs_BCHornerDer2Pf` służy do obliczania punktu  $\mathbf{p}(u,v)$  i wektorów pochodnych cząstkowych rzędu 1 i 2 płata Béziera  $\mathbf{p}$  położonego w przestrzeni o wymiarze  $d$ .

Parametry: `degreeu`, `degreev` — określają stopień płata (liczby  $n$  i  $m$ ). Parametr `spdimen` określa wymiar  $d$  przestrzeni, tablica `ctlpoints` zawiera współrzędne punktów kontrolnych.

Wyniki (punkt płata i wektory pochodnych) procedura wpisuje do tablic `p`, `du`, `dv`, `duu`, `duv` i `dvv`, które muszą mieć długość co najmniej `spdimen`.

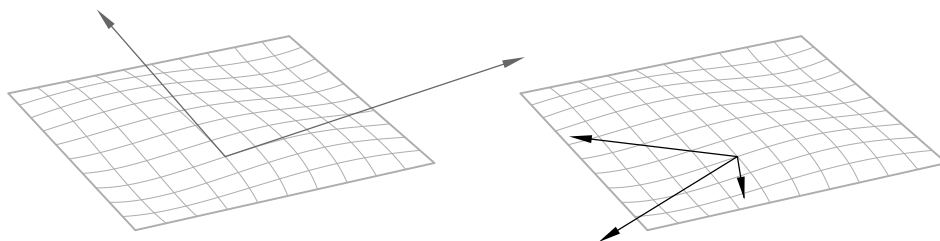
```
#define mbs_BCHornerDer2P1f(degreeu,degreev,coeff,u,v, \
    p,du,dv,duu,duv,dvv) \
    mbs_BCHornerDer2Pf ( degreeu, degreev, 1, coeff, u, v, \
    p, du, dv, duu, duv, dvv )
#define mbs_BCHornerDer2P2f(degreeu,degreev,ctlpoints, \
    u,v,p,du,dv,duu,duv,dvv) \
    mbs_BCHornerDer2Pf ( degreeu, 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) ...
```

Powyższe makra służą do obliczania punktów i wektorów pochodnych rzędu 1 i 2 płatów Béziera położonych w przestrzeniach o wymiarach 1, 2, 3 i 4, za pomocą procedury `mbs_BCHornerDer2Pf`.

```
void mbs_BCHornerDer2P3Rf ( int degree, int degreev,
                           const point4f *ctlpoints,
                           float u, float v,
                           point3f *p, vector3f *du, vector3f *dv,
                           vector3f *duu, vector3f *dudv,
                           vector3f *dvv );
```

Procedura `mbs_BCHornerDer2P3Rf` oblicza punkt wymiennego płata Béziera  $\mathbf{p}$ , położonego w przestrzeni trójwymiarowej, oraz wektory pochodnych rzędu 1 i 2.

Parametry: `degree`, `degreev` — stopień płata ze względu na parametry  $u$  i  $v$ , `ctlpoints` — tablica punktów kontrolnych płata *jednorodnego*,  $u$ ,  $v$  — liczby  $u$  i  $v$ , `*p`, `*du`, `*dv`, `*duu`, `*dudv`, `*dvv` — struktury, do których procedura ma wpisać wyniki, odpowiednio punkt  $\mathbf{p}(u, v)$ ,  $\frac{\partial}{\partial u}\mathbf{p}(u, v)$ ,  $\frac{\partial}{\partial v}\mathbf{p}(u, v)$ ,  $\frac{\partial^2}{\partial u^2}\mathbf{p}(u, v)$ ,  $\frac{\partial^2}{\partial u \partial v}\mathbf{p}(u, v)$  i  $\frac{\partial^2}{\partial v^2}\mathbf{p}(u, v)$ .



Rys. 7.2. Płat Béziera i jego wektory pochodnych cząstkowych pierwszego i drugiego rzędu.

```
void mbs_multiBCHornerDer3f ( int degree, int ncurves,
                              int spdimen, int pitch,
                              const float *ctlpoints, float t,
                              float *p, float *d1, float *d2, float *d3 );
```

Procedura `mbs_multiBCHornerDer3f` oblicza za pomocą schematu Hornera punkty  $\mathbf{c}_i(t)$  i wektory pochodnych  $\mathbf{c}'_i(t)$ ,  $\mathbf{c}''_i(t)$  i  $\mathbf{c}'''_i(t)$  krzywych Béziera  $\mathbf{c}_i$  położonych w przestrzeni o wymiarze  $d$ .

Parametry: `degree` — określa stopień krzywej, `ncurves` — liczbę krzywych, `spdimen` — wymiar  $d$  przestrzeni, `pitch` — podziałkę tablicy `ctlpoints` zawierającej punkty kontrolne. Parametr  $t$  ma wartość  $t$ .

Obliczone współrzędne punktów  $\mathbf{c}_i(t)$  i wektorów  $\mathbf{c}'_i(t)$ ,  $\mathbf{c}''_i(t)$  i  $\mathbf{c}'''_i(t)$  procedura wpisuje odpowiednio do tablic `p`, `d1`, `d2` i `d3`, które muszą mieć długość co najmniej `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) ...
```

Powyższe makra służą do obliczania punktów i wektorów pochodnych pierwszego, drugiego i trzeciego rzędu jednej krzywej Béziera położonych w przestrzeni o wymiarze 1, 2, 3 lub 4, za pomocą procedury `mbs_multiBCHornerDer3f`.

```
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 );
```

Procedura `mbs_BCHornerDer3Pf` oblicza punkt  $p(u,v)$  płata Béziera  $p$  położonego w przestrzeni o wymiarze `spdimen` i jego pochodne rzędu  $1, \dots, 3$  w tym punkcie. Obecna wersja procedury działa przy założeniu, że stopień płata ze względu na oba parametry jest nie mniejszy niż 3 — zaimplementowanie obsługi pozostałych przypadków pozostaje do zrobienia.

```
#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) ...
```

Podane wyżej makra służą do wywoływania procedury `mbs_BCHornerDer3Pf` w celu obliczenia punktu i pochodnych płatów położonych w przestrzeniach o wymiarach odpowiednio  $1, \dots, 4$ .

### 7.4.3 Obliczanie krzywizn i układu Freneta krzywej

Obliczanie krzywizn i wektorów układu Freneta jest oprogramowane tylko dla krzywych Béziera. Chcąc obliczyć krzywiznę krzywej B-sklejanej trzeba dokonać maksymalnego wstawienia węzłów (np. za pomocą procedury `mbs_multiMaxKnotInsf`) w celu otrzymania reprezentacji Béziera poszczególnych łuków. Zwykle krzywizna będzie tablicowana, a zatem wykonamy jedną taką konwersję, a potem wiele obliczeń krzywizny w różnych punktach. Dlatego nie ma procedur obliczających bezpośrednio krzywizny krzywych B-sklejanych. Procedury opisane poniżej wykonują obliczenia za pomocą procedury `mbs_multiBCHornerf`.

```
void mbs_BCFrenetC2f ( int degree, const point2f *ctlpoints,
                      float t, point2f *cpoint,
                      vector2f *fframe, float *curvature );
```

Procedura `mbs_BCFrenetC2f` oblicza krzywiznę i wektory styczny  $t$  i normalny  $n$  układu Freneta płaskiej krzywej Béziera stopnia `degree`, której punkty kontrolne są podane w tablicy `ctlpoints`. Parametr krzywej jest równy  $t$ . Tablica `fframe` musi pomieścić dwa wektory. Ponadto procedura oblicza punkt krzywej i przypisuje go do parametru `*cpoint`.

```
void mbs_BCFrenetC2Rf ( int degree, const point3f *ctlpoints,
                       float t, point2f *cpoint,
                       vector2f *fframe, float *curvature );
```

Procedura `mbs_BCFrenetC2f` oblicza krzywiznę i wektory styczny  $t$  i normalny  $n$  układu Freneta płaskiej wymiernej krzywej Béziera stopnia `degree`, której punkty kontrolne (w reprezentacji jednorodnej) są podane w tablicy `ctlpoints`. Parametr krzywej jest równy  $t$ . Tablica `fframe` musi pomieścić dwa wektory. Ponadto procedura oblicza punkt krzywej i przypisuje go do parametru `*cpoint`.

```
void mbs_BCFrenetC3f ( int degree, const point3f *ctlpoints,
                      float t, point3f *cpoint,
                      vector3f *fframe, float *curvatures );
```

Procedura `mbs_BCFrenetC3f` oblicza krzywiznę i skręcenie wielomianowej krzywej Béziera stopnia `degree` oraz wektory układu Freneta: styczny  $t$ , normalny  $n$  i binormalny  $b$  w punkcie odpowiadającym danemu parametrowi  $t$ . Tablica `ctlpoints` zawiera punkty kontrolne krzywej. Krzywizna i skręcenie są wpisywane do tablicy `curvatures`, a wektory do tablicy `fframe`. Ponadto procedura oblicza punkt krzywej i przypisuje do parametru `*cpoint`.

```
void mbs_BCFrenetC3Rf ( int degree, const point4f *ctlpoints,
                       float t, point3f *cpoint,
                       vector3f *fframe, float *curvatures );
```

Procedura `mbs_BCFrenetC3f` oblicza krzywiznę i skręcenie wymiernej krzywej Béziera stopnia `degree` oraz wektory układu Freneta: styczny  $\mathbf{t}$ , normalny  $\mathbf{n}$  i binormalny  $\mathbf{b}$  w punkcie odpowiadającym danemu parametrowi  $t$ . Tablica `ctlpoints` zawiera punkty kontrolne krzywej w reprezentacji jednorodnej. Krzywizna i skręcenie są wpisywane do tablicy `curvatures`, a wektory do tablicy `fframe`. Ponadto procedura oblicza punkt krzywej i przypisuje go do parametru `*cpoint`.

#### 7.4.4 Obliczanie wektora normalnego płata

```
void mbs_BCHornerNvP3f ( int degreeu, int degreev,
                        const point3f *ctlpoints,
                        float u, float v,
                        point3f *p, vector3f *nv );
```

Procedura `mbs_BCHornerNvP3f` służy do obliczenia punktu płata Béziera położonego w przestrzeni trójwymiarowej i jego wektora normalnego w tym punkcie. Obliczony wektor jest iloczynem pochodnych cząstkowych i może być wektorem zerowym jeśli w danym punkcie jest osobliwość, nawet jeśli w tym punkcie i jego otoczeniu płaszczyzna styczna do płata jest jednoznacznie określona.

```
void mbs_BCHornerNvP3Rf ( int degreeu, int degreev,
                        const point4f *ctlpoints,
                        float u, float v,
                        point3f *p, vector3f *nv );
```

Procedura `mbs_BCHornerNvP3Rf` służy do obliczania punktu wymiernego płata Béziera położonego w przestrzeni trójwymiarowej i jego wektora normalnego w tym punkcie. Współrzędne obliczonego wektora normalnego to pierwsze trzy współrzędne iloczynu wektorowego  $\mathbf{P} \wedge \mathbf{P}_u \wedge \mathbf{P}_v$  (iloczynu punktu płata jednorodnego i jego pochodnych cząstkowych). Wektor ten może być wektorem zerowym, jeśli płat ma osobliwość, nawet jeśli płaszczyzna styczna w danym punkcie jest określona jednoznacznie.

#### 7.4.5 Obliczanie form podstawowych i krzywizn płatów

Obliczanie form podstawowych i krzywizn jest oprogramowane tylko dla płatów Béziera; powody, dla których nie ma procedur bezpośrednio obliczających te obiekty dla płatów B-sklejanych są takie same jak powody, dla których w bibliotece są tylko procedury obliczania krzywizn i układu Freneta krzywych Béziera.

```
void mbs_FundFormsBP3f ( int degreeu, int degreev,
                        const point3f *ctlpoints,
                        float u, float v,
                        float *firstform, float *secondform );
```

Procedura `mbs_FundFormsBP3f` oblicza współczynniki macierzy pierwszej i drugiej formy podstawowej wielomianowego płata Béziera w przestrzeni trójwymiarowej.

Parametry: `degreeu`, `degreev` — stopień płata ze względu na parametry  $u$  i  $v$ , `ctlpoints` — tablica punktów kontrolnych (spakowana, tj. bez obszarów nieużywanych między danymi opisującymi kolejne kolumny siatki kontrolnej). Parametry  $u$  i  $v$  określają punkt, w którym procedura ma obliczyć formy.

Parametry `firstform` i `secondform` są wskaźnikami tablic o długości co najmniej 3. Procedura wpisuje do tych tablic współczynniki macierzy form, odpowiednio  $g_{11} = \langle \mathbf{p}_u, \mathbf{p}_u \rangle$ ,  $g_{12} = g_{21} = \langle \mathbf{p}_u, \mathbf{p}_v \rangle$  i  $g_{22} = \langle \mathbf{p}_v, \mathbf{p}_v \rangle$ , oraz  $b_{11} = \langle \mathbf{n}, \mathbf{p}_{uu} \rangle$ ,  $b_{12} = b_{21} = \langle \mathbf{n}, \mathbf{p}_{uv} \rangle$  i  $b_{22} = \langle \mathbf{n}, \mathbf{p}_{vv} \rangle$  (gdzie  $\mathbf{n}$  oznacza jednostkowy wektor normalny płata w punkcie  $(u, v)$ ).

```
void mbs_GMCurvaturesBP3f ( int degreeu, int degreev,
                             const point3f *ctlpoints,
                             float u, float v,
                             float *gaussian, float *mean );
```

Procedura `mbs_GMCurvaturesBP3f` oblicza krzywiznę gaussowską i średnią wielomianowego płata Béziera w  $\mathbb{R}^3$ . Parametry `degreeu`, `degreev`, `ctlpoints`,  $u$  i  $v$  są identyczne jak odpowiednie parametry poprzedniej procedury.

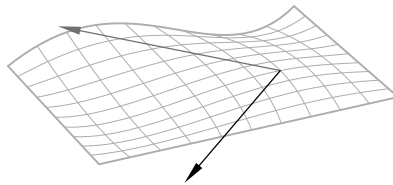
Parametry `*gaussian` i `*mean` służą do wyprowadzenia wyników; procedura przypisuje im odpowiednio obliczoną wartość krzywizny gaussowskiej i średniej.

```
void mbs_PrincipalDirectionsBP3f ( int degreeu, int degreev,
                                    const point3f *ctlpoints,
                                    float u, float v,
                                    float *k1, vector2f *v1,
                                    float *k2, vector2f *v2 );
```

Procedura `mbs_PrincipalDirectionsBP3f` oblicza krzywizny i kierunki główne wielomianowego płata Béziera w przestrzeni trójwymiarowej. Parametry `degreeu`, `degreev`, `ctlpoints`,  $u$  i  $v$  są identyczne jak odpowiednie parametry poprzednich dwóch procedur.

Parametrom `*k1` i `*k2` procedura przypisuje wartości krzywizn głównych płata, a odpowiednie kierunki główne (w przestrzeni stycznej do dziedziny płata) są przypisywane parametrom `*v1` i `*v2`.

```
void mbs_FundFormsBP3Rf ( int degreeu, int degreev,
                          const point4f *ctlpoints,
                          float u, float v,
                          float *firstform, float *secondform );
```



Rys. 7.3. Wektory odpowiadające kierunkom głównym w pewnym punkcie płata Béziera

```
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 );
```

Powyższe procedury obliczają odpowiednio współczynniki macierzy pierwszej i drugiej formy podstawowej, krzywizny gaussowską i średnią oraz krzywizny i kierunki główne wymiennego płata Béziera  $\mathbf{p}$ . Procedury te są dokładnymi odpowiednikami procedur `mbs_FundFormsBP3f`, `mbs_GMCurvaturesBP3f` i `mbs_GMCurvaturesBP3f` i mają takie same parametry, z wyjątkiem `ctlpoints`, który jest tablicą punktów kontrolnych płata *jednorodnego* w przestrzeni  $\mathbb{R}^4$ .

## 7.5 Tablicowanie krzywych

Opisane niżej procedury obliczają ciąg punktów krzywej Béziera lub B-sklejanej oraz pochodne rzędu 1 i 2 albo 1, 2 i 3 dla ciągu wartości parametru  $t_0, \dots, t_{k-1}$ , wywołując w pętli odpowiednie procedury opisane wcześniej. Głównym ich zastosowaniem jest wykorzystanie do tablicowania płatów Coonsa (przez procedury opisane w p. 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 );
```

Procedura `mbs_TabBezCurveDer2f` tablicuje krzywą Béziera i jej pochodne rzędu 1 i 2 przy użyciu procedury `mbs_multiBCHornerDer2f`.

Parametry: `spdimen` — wymiar przestrzeni, `degree` — stopień krzywej, `cp` — tablica punktów kontrolnych, `nkn` — liczba  $k$ , `kn` — tablica zawierająca  $k$  liczb (wartości parametru), `ppitch` — podziałka tablic `p`, `dp` i `ddp`, do których mają być wpisane współrzędne punktów krzywej, wektorów pochodnej i wektorów pochodnej drugiego rzędu. Pierwsze współrzędne kolejnych punktów i wektorów są wpisywane do tablic na miejsca odległe od siebie o wartość parametru `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 );
```

Procedura `mbs_TabBezCurveDer3f` tablicuje krzywą Béziera i jej pochodne rzędu 1, 2 i 3 przy użyciu procedury `mbs_multiBCHornerDer3f`.

Parametry: `spdimen` — wymiar przestrzeni, `degree` — stopień krzywej, `cp` — tablica punktów kontrolnych, `nkn` — liczba  $k$ , `kn` — tablica zawierająca  $k$  liczb (wartości parametru), `ppitch` — podziałka tablic `p`, `dp`, `ddp` i `dddp`, do których mają być wpisane odpowiednio współrzędne punktów krzywej, i wektorów pochodnej rzędu 1, 2 i 3. Pierwsze współrzędne kolejnych punktów i wektorów są wpisywane do tablic na miejsca odległe od siebie o wartość parametru `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 );
```

Procedura `mbs_TabBSCurveDer2f` tablicuje krzywą B-sklejaną i jej pochodne rzędu 1, i 2 przy użyciu procedury `mbs_multideBoorDer2f`.



Parametry: `spdimen` — wymiar przestrzeni, `degree` — stopień krzywej, `lastknot` — numer ostatniego węzła, `knots` — tablica węzłów, `cp` — tablica punktów kontrolnych, `nkn` — liczba  $k$ , `kn` — tablica zawierająca  $k$  liczb (wartości parametru), `ppitch` — podziałka tablic `p`, `dp` i `dpp`, do których mają być wpisane odpowiednio współrzędne punktów krzywej, i wektorów pochodnej rzędu 1, i 2. Pierwsze współrzędne kolejnych punktów i wektorów są wpisywane do tablic na miejsca odległe od siebie o wartość parametru `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 *dpp, float *dddp );
```

Procedura `mbs_TabBSCurveDer3f` tablicuje krzywą B-sklejaną i jej pochodne rzędu 1, 2 i 3 przy użyciu procedury `mbs_multideBoorDer3f`.

Parametry: `spdimen` — wymiar przestrzeni, `degree` — stopień krzywej, `lastknot` — numer ostatniego węzła, `knots` — tablica węzłów, `cp` — tablica punktów kontrolnych, `nkn` — liczba  $k$ , `kn` — tablica zawierająca  $k$  liczb (wartości parametru), `ppitch` — podziałka tablic `p`, `dp`, `dpp` i `dddp`, do których mają być wpisane odpowiednio współrzędne punktów krzywej, i wektorów pochodnej rzędu 1, 2 i 3. Pierwsze współrzędne kolejnych punktów i wektorów są wpisywane do tablic na miejsca odległe od siebie o wartość parametru `ppitch`.

## 7.6 Znajdowanie reprezentacji pochodnych

Czym innym jest obliczanie wektora pochodnej krzywej w ustalonym punkcie, a czym innym znalezienie krzywej reprezentującej pochodną krzywej danej. Opisane niżej procedury obliczają takie reprezentacje na podstawie wzorów

$$\frac{d}{dt} \sum_{i=0}^n p_i B_i^n(t) = \sum_{i=0}^{n-1} n(p_{i+1} - p_i) B_{i+1}^{n-1}(t), \quad (7.18)$$

dla krzywych Béziera, oraz

$$\frac{d}{dt} \sum_{i=0}^{N-n-1} d_i N_i^n(t) = \sum_{i=0}^{N-n-2} \frac{n}{u_{i+n+1} - u_{i+1}} (d_{i+1} - d_i) N_{i+1}^{n-1}(t), \quad (7.19)$$

dla krzywych B-sklejanych. Funkcje B-sklejane  $N_i^n$  oraz  $N_{i+1}^{n-1}$  są określone dla tego samego ciągu węzłów.

```
void mbs_multiFindBezDerivativef ( int degree,
                                   int ncurves, int spdimen,
                                   int pitch, const float *ctlpoints,
                                   int dpitch, float *dctlpoints );
```

Procedura `mbs_multiFindBezDerivativef` oblicza punkty kontrolne krzywych Béziera stopnia  $n - 1$  opisujących pochodne danych krzywych Béziera stopnia  $n$ .

Parametry wejściowe: `degree` — stopień  $n$  krzywych danych (musi być dodatni), `ncurves` — liczba krzywych danych, `spdimen` — wymiar przestrzeni, w której leżą krzywe, `pitch` — podziałka tablicy `ctlpoints` określająca odległość początków reprezentacji krzywych, `ctlpoints` — tablica zawierająca współrzędne punktów kontrolnych krzywych danych.

Parametr `dpitch` opisuje podziałkę tablicy `dctlpoints`, do której procedura wpisuje obliczone punkty kontrolne krzywych opisujących pochodne.

```
#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 )
```

Powyższe makra wywołują procedurę `mbs_multiFindBezDerivativef` w celu obliczenia punktów kontrolnych jednej krzywej Béziera stopnia  $n$  położonej w przestrzeni o wymiarze 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 );
```

Procedura `mbs_multiFindBSDerivativef` oblicza punkty kontrolne krzywych B-sklejanych stopnia  $n - 1$  opisujących pochodne danych krzywych B-sklejanych stopnia  $n$ .

Parametry wejściowe: `degree` — stopień  $n$  krzywych danych, `lastknot` — indeks  $N$  ostatniego węzła, `knots` — tablica węzłów  $u_0, \dots, u_N$ , `ncurves` — liczba krzywych danych, `spdimen` — wymiar przestrzeni, w której leżą krzywe, `pitch` — podziałka tablicy `ctlpoints` określająca odległość początków reprezentacji krzywych, `ctlpoints` — tablica zawierająca współrzędne punktów kontrolnych krzywych danych.

Parametr wyjściowy `*lastdknot` otrzymuje wartość  $N - 2$ , zaś do tablicy `dknots` procedura przepisuje węzły  $u_1, \dots, u_{N-1}$ . Parametry `lastdknot` i `dknots` mogą mieć wartość `NULL` i wtedy są ignorowane.

Parametr `dpitch` opisuje podziałkę tablicy `dctlpoints`, do której procedura wpisuje obliczone punkty kontrolne krzywych opisujących pochodne.

```
#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) ...
```

Powyższe makra wywołują procedurę `mbs_multiFindBSDerivativef` w celu znalezienia reprezentacji pochodnej jednej krzywej B-sklejanej położonej w przestrzeni o wymiarze 1, 2, 3 lub 4.

## 7.7 Wstawianie i usuwanie węzłów

### 7.7.1 Algorytm Boehma

Poniżej jest opisana procedura i makra, które służą do wstawiania pojedynczego węzła do reprezentacji B-sklejanej krzywych, za pomocą algorytmu Boehma. Reprezentacja krzywych jest *zmieniana*, tj. obszar pamięci zajmowany początkowo przez daną reprezentację krzywych (ciąg węzłów i tablica punktów kontrolnych), po wykonaniu procedury zawiera nową reprezentację, z dodatkowym węzłem. Jeśli potrzebne są obie reprezentacje, to reprezentację daną trzeba skopiować i wykonać wstawianie węzła do kopii.

```
int mbs_multiKnotInsf ( int degree, int *lastknot,
                        float *knots,
                        int ncurves, int spdimen,
                        int inpitch, int outpitch,
                        float *ctlpoints, float t );
```

Procedura `mbs_multiKnotInsf` wstawia węzeł `t` do reprezentacji krzywych B-sklejanych stopnia  $n = \text{degree}$ . W ten sposób powstaje nowa reprezentacja tych krzywych, przechowywana w miejscu reprezentacji początkowej. Wartość parametru `t` musi być liczbą z przedziału  $[\text{knots}[\text{degree}], \text{knots}[\text{lastknot} - \text{degree}]]$ .

Na wejściu parametr `*lastknot` określa indeks `N` ostatniego węzła w ciągu węzłów reprezentacji początkowej; na wyjściu parametr ten jest zwiększany o 1, co wiąże się z wydłużeniem ciągu węzłów o jedną liczbę — wartość parametru `t`, wstawianą do tablicy `knots`. Tablica ta musi mieć zatem długość co najmniej `*lastknot + 2`, aby pomieścić wydłużony ciąg węzłów.

Parametr `ncurves` określa liczbę krzywych, zaś wartość `d` parametru `spdimen` jest wymiarem przestrzeni, w której te krzywe są położone. Każda krzywa jest początkowo reprezentowana przez  $N - n$  punktów w przestrzeni  $d$ -wymiarowej. Współrzędne tych punktów ( $(N - n)d$  liczb) są przechowywane w tablicy `ctlpoints`. Pierwsza współrzędna pierwszego punktu kontrolnego pierwszej krzywej jest na początku tablicy. Ponieważ na wyjściu reprezentacja każdej krzywej ma o jeden punkt więcej, więc są dwa parametry opisujące podziałkę, czyli odległość w tablicy między początkami reprezentacji kolejnych krzywych: `inpitch` określa podziałkę początkową, co najmniej  $(N - n)d$ , parametr `outpitch` określa podziałkę końcową, która nie może być mniejsza niż  $(N - n + 1)d$ .

Wartością procedury jest numer `k` przedziału  $[u_k, u_{k+1})$  początkowego ciągu węzłów, w którym leży nowy węzeł `t`.

**Uwaga:** Aby wstawić węzeł do krzywej zamkniętej, zamiast `mbs_multiKnotInsf` należy użyć procedury `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) ...
```

Cztery makra wywołujące procedurę `mbs_multiKnotInsf` w celu wstawienia węzła do reprezentacji *jednej* funkcji skalarnej lub krzywej B-sklejanej w przestrzeni dwu-, trój- i czterowymiarowej. Parametry muszą spełniać warunki podane w opisie procedury `mbs_multiKnotInsf`.

```
int mbs_multiKnotInsClosedf ( int degree, int *lastknot,
                             float *knots,
                             int ncurves, int spdimen,
                             int inpitch, int outpitch,
                             float *ctlpoints, float t );
```

Procedura `mbs_multiKnotInsClosedf` wstawia węzeł `t` do reprezentacji *zamkniętych* krzywych B-sklejanych stopnia `degree`. Może to być wykorzystane również do wstawiania węzła do zamkniętego płata B-sklejanego (będącego rurką lub torusem). Zasadnicze obliczenie wykonuje procedura `mbs_multiKnotInsf`, po wykonaniu której wynik jest „porządkowany” w celu przywrócenia okresowości reprezentacji krzywej.

Parametry: `degree` — stopień  $n$  krzywej, `*lastknot` — przed wywołaniem procedury ma wartość indeksu  $N$  ostatniego węzła w początkowym ciągu, a po wywołaniu indeksu ostatniego węzła w ciągu wynikowym. Tablica `knots` zawiera ciąg węzłów, odpowiednio początkowy i końcowy, przed i po wykonaniu procedury. Parametr `ncurves` określa liczbę krzywych. Parametr `spdimen` określa wymiar  $d$  przestrzeni. Parametry `inpitch` i `outpitch` opisują podziałki tablicy `ctlpoints`, w której znajdują się punkty kontrolne, przed i po wstawieniu węzła (zobacz opis procedury `mbs_multiKnotInsf`). Parametr `t` określa nowy węzeł, który procedura ma wstawić.

```
#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) ...
```

Cztery makra wywołujące procedurę `mbs_multiKnotInsClosedf` w celu wstawienia węzła do reprezentacji *jednej* okresowej funkcji skalarnej lub zamkniętej krzywej B-sklejanej w przestrzeni dwu-, trój- i czterowymiarowej. Parametry muszą spełniać warunki podane w opisie procedury `mbs_multiKnotInsClosedf`.

### 7.7.2 Usuwanie węzłów

W tym punkcie jest opisana procedura usuwania pojedynczego węzła z reprezentacji krzywych B-sklejanych i makra, które służą wygodnemu wywoływaniu tej procedury w przypadku jednej krzywej w przestrzeniach o wymiarach 1–4. Procedura tworzy układ równań wiążący dwie reprezentacje krzywej, z macierzą równoważną zmianie reprezentacji przy użyciu algorytmu Boehma wstawiania węzła, a następnie rozwiązuje ten układ metodą najmniejszych kwadratów. Krzywe mogą w wyniku usuwania węzła ulec zmianie.

Usuwanie węzła odbywa się „w miejscu”, tj. obszar pamięci zajmowany początkowo przez daną reprezentację, po wykonaniu procedury zawiera nowy, krótszy ciąg węzłów i punktów kontrolnych. Jeśli potrzebne są obie reprezentacje, tj. reprezentacja początkowa i reprezentacja z usuniętym węzłem, to reprezentację początkową trzeba skopiować, a następnie wykonać usuwanie węzła na kopii.

```
int mbs_multiKnotRemovef ( int degree, int *lastknot,
                           float *knots,
                           int ncurves, int spdimen,
                           int inpitch, int outpitch,
                           float *ctlpoints,
                           int knotnum );
```

Procedura `mbs_multiKnotRemovef` służy do usuwania węzła z reprezentacji krzywych B-sklejanych stopnia `degree`, położonych w przestrzeni o wymiarze `spdimen`. Reprezentacja krzywych jest określona dla ciągu węzłów o długości `*lastknot+1`, podanego w tablicy `knots`. Punkty kontrolne krzywych są podane w tablicy `ctlpoints`. Parametr `inpitch` określa podziałkę, czyli odległość między początkami obszarów w tablicy `ctlpoints` zawierających współrzędne danych punktów kontrolnych kolejnych krzywych. Parametr `outpitch` określa podziałkę tej tablicy ustaloną po usunięciu węzła.

Węzeł do usunięcia jest określony przez parametr `knotnum`, który musi mieć wartość od `degree+1` do `lastknot-degree-1`.

Nowe reprezentacje krzywych są umieszczane w tablicach `knots` i `ctlpoints`. Parametr `*lastknot` jest zmniejszany o 1.

Jeśli krotność usuwanego węzła jest równa  $r$  i pochodna krzywej rzędu  $degree - r + 1$  nie jest w tym węźle ciągła, to krzywa po usunięciu węzła zmieni się. Nowe punkty kontrolne są obliczane przez rozwiązanie liniowego zadania najmniejszych kwadratów dla układu równań liniowych opisującego zmianę bazy, co jest pewnym sposobem rozwiązania aproksymacyjnego (zobacz przykład podany dalej).

Wartością procedury jest liczba  $k$ , taka że usunięty węzeł należy do przedziału  $[u_k, u_{k+1})$  *wynikowego* ciągu węzłów. Jeśli węzeł o numerze `knotnum` jest mniejszy niż węzeł następny, to  $k = \text{knotnum} - 1$ , ale w ogólności nie musi tak być.

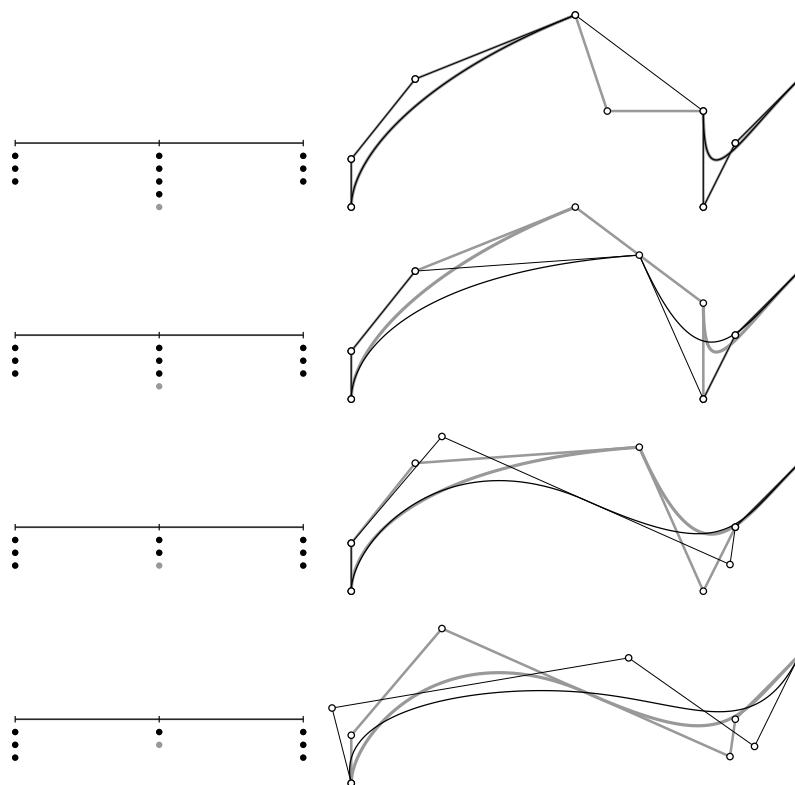
**Uwaga:** Aby usunąć węzeł z reprezentacji krzywej zamkniętej należy użyć procedury `mbs_multiKnotRemoveClosedf`. Użycie procedury `mbs_multiKnotRemovef`

może spowodować powstanie krzywej nie-zamkniętej.

```
#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) ...
```

Cztery makra wywołujące procedurę `mbs_multiKnotRemovef` w celu usunięcia wskazanego węzła z reprezentacji *jednej* funkcji sklejaney lub krzywej B-sklejanej w przestrzeni dwu-, trój- lub czterowymiarowej. Parametry muszą spełniać warunki podane w opisie procedury `mbs_multiKnotRemovef`.

Na rysunku 7.4 jest przykład usuwania węzłów, dla płaskiej krzywej B-sklejanej



Rys. 7.4. Przykład usuwania węzłów

trzeciego stopnia (zobacz program `test/knotrem.c`). Usuwany węzeł ma początkową krotność większą o 2 od stopnia, w związku z czym krzywa składa się z osobnych łuków i jeden z punktów kontrolnych nie ma wpływu na jej kształt. Usunięcie węzła powoduje odrzucenie tego punktu, bez zmiany kształtu krzywej.

Podczas usuwania węzła o krotności o 1 większej niż stopień łuki muszą zostać połączone — dwa punkty kontrolne zostają zastąpione przez jeden, leżący w połowie odległości między nimi. Dalsze usuwanie węzła polega na rozwiązywaniu odpowiednich liniowych zadań najmniejszych kwadratów.

```
int mbs_multiKnotRemoveClosedf ( int degree, int *lastknot,
                                float *knots,
                                int ncurves, int spdimen,
                                int inpitch, int outpitch,
                                float *ctlpoints,
                                int knotnum );
```

Procedura `mbs_multiKnotRemoveClosedf` służy do usuwania węzła z reprezentacji zamkniętych krzywych B-sklejanych.

Parametry `degree`, `ncurves` i `spdimen` opisują odpowiednio stopień krzywych, ich liczbę i wymiar przestrzeni, w której krzywe leżą. Parametry `*lastknot` i `knots` opisują początkowo początkowy ciąg węzłów. Po wykonaniu procedury parametry te opisują końcowy ciąg węzłów. Parametr `knotnum` określa numer węzła, który należy usunąć. Parametry `inpitch` i `outpitch` określają początkową i końcową podziałkę tablicy punktów kontrolnych `ctlpoints`, tj. odległości początków łamanych kontrolnych poszczególnych krzywych. W tablicy `ctlpoints` należy podać punkty kontrolne początkowej reprezentacji krzywych; procedura umieszcza w niej obliczoną reprezentację krzywych z usuniętym węzłem.

```
#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) ...
```

Cztery makra wywołujące procedurę `mbs_multiKnotRemoveClosedf` w celu usunięcia wskazanego węzła z reprezentacji *jednej* zamkniętej funkcji skleianej lub krzywej B-sklejanej w przestrzeni dwu-, trój- lub czterowymiarowej. Parametry muszą spełniać warunki podane w opisie procedury `mbs_multiKnotRemoveClosedf`.



```

void mbs_multiRemoveSuperfluousKnotsf ( int ncurves,
                                         int spdimen, int degree,
                                         int *lastknot,
                                         float *knots,
                                         int inpitch, int outpitch,
                                         float *ctlpoints );

```

Procedura `mbs_multiRemoveSuperfluousKnotsf` służy do takiego usunięcia z reprezentacji krzywych węzłów, aby krotność żadnego z pozostałych węzłów nie przekraczała stopnia (parametr `degree`) plus 1. Nie powoduje to zmiany krzywej, natomiast pozwala uniknąć kłopotów związanych z występowaniem takich węzłów (m.in. funkcja B-sklejana, której wszystkie węzły są tą samą liczbą, jest tożsamościowo równa 0, a zatem układ funkcji B-sklejanych stopnia  $n$  opartych na ciągu węzłów, który zawiera węzeł o krotności większej niż  $n + 1$ , nie jest bazą).

Obliczenie jest wykonywane „w miejscu”, tj. obszar zajmowany początkowo przez reprezentację daną, po wykonaniu procedury zawiera nową reprezentację krzywych. Usuwanie węzłów polega na „przemieszczaniu” danych (węzłów i punktów kontrolnych) w tablicach, bez żadnych obliczeń numerycznych.

### 7.7.3 Algorytm Oslo

Algorytm Oslo jest metodą przejścia od opartej na ciągu węzłów  $u_0, \dots, u_N$  reprezentacji B-sklejanej krzywej do reprezentacji z dodatkowymi węzłami (które razem z poprzednimi tworzą ciąg  $\hat{u}_0, \dots, \hat{u}_N$ ). Inaczej niż w algorytmie Boehma (zobacz p. 7.7.1), który służy do wstawiania jednego węzła (i w razie potrzeby należy go stosować wielokrotnie), tu wszystkie węzły są wstawiane jednocześnie.

Jeśli punkty kontrolne  $\mathbf{d}_i$  krzywej B-sklejanej stopnia  $n$  odpowiadają ciągowi węzłów  $u_0, \dots, u_N$ , zaś punkty kontrolne  $\hat{\mathbf{d}}_l$  odpowiadają ciągowi  $\hat{u}_0, \dots, \hat{u}_N$ , to

$$\hat{\mathbf{d}}_l = \sum_{i=0}^{N-n-1} a_{il}^n \mathbf{d}_i, \quad (7.20)$$

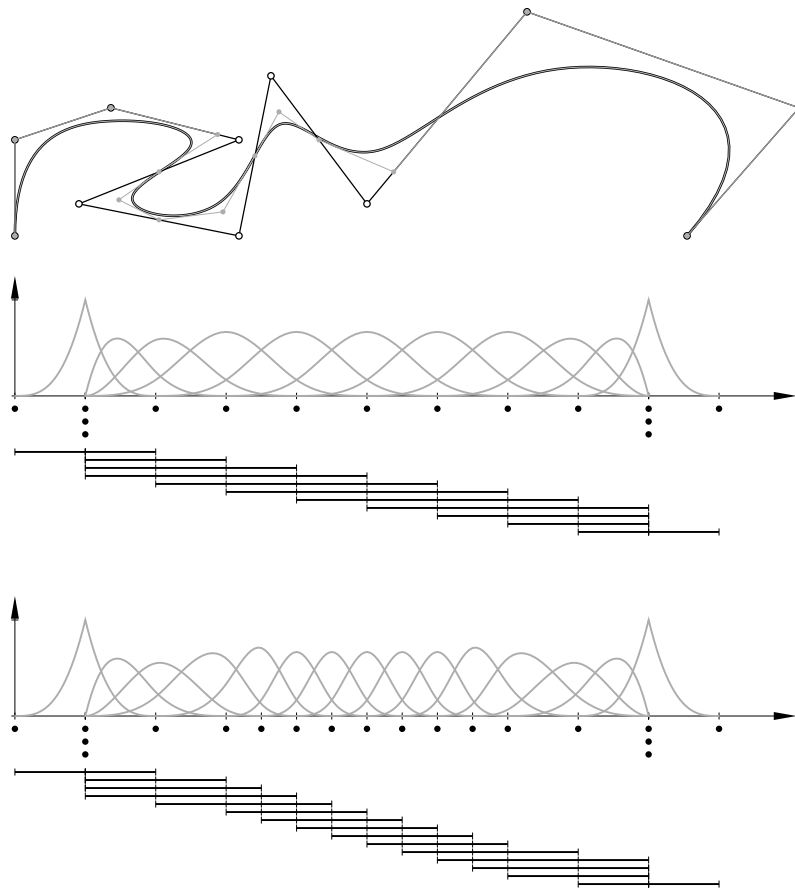
gdzie współczynniki  $a_{kl}^n$  są określone za pomocą rekurencyjnych wzorów

$$a_{kl}^0 = \begin{cases} 1 & \text{dla } u_k \leq \hat{u}_l < u_{k+1}, \\ 0 & \text{w przeciwnym razie,} \end{cases} \quad (7.21)$$

$$a_{il}^n = \frac{\hat{u}_{l+n} - u_i}{u_{i+n} - u_i} a_{il}^{n-1} + \frac{u_{i+n+1} - \hat{u}_{l+n}}{u_{i+n+1} - u_{i+1}} a_{i+1,l}^{n-1}. \quad (7.22)$$

Implementacja algorytmu Oslo w bibliotece `libmultibs` polega na tym, że najpierw jest obliczana macierz  $A$  współczynników  $a_{il}^n$ , a następnie jest ona mnożona przez macierz punktów kontrolnych  $\mathbf{d}_0, \dots, \mathbf{d}_{N-n-1}$ .

Macierz  $A$  umożliwia również usunięcie w jednym kroku wielu węzłów, przez rozwiązywanie nadokreślonego układu równań liniowych (w którym jest więcej równań



Rys. 7.5. Wstawianie wielu węzłów za pomocą algorytmu Oslo

niż niewiadomych). Taki układ równań, nawet jeśli jest niesprzeczny, najlepiej jest rozwiązywać jako liniowe zadanie najmniejszych kwadratów.

Macierz zmiany reprezentacji jest reprezentowana jako macierz wstęgowa, za pomocą tablicy opisującej profil (tj. położenia niezerowych współczynników w kolejnych kolumnach) i tablicy z niezerowymi współczynnikami. Szczegółowy opis tej reprezentacji i procedur jej przetwarzania jest treścią p. 3.2.

```
boolean mbs_OsloKnotsCorrectf ( int lastuknot, const float *uknots,
                                int lastvknot, const float *vknots );
```

Procedura `mbs_OsloKnotsCorrectf` sprawdza, czy dane dwa ciągi węzłów umożliwiają skonstruowanie macierzy przejścia. Sprawdzane warunki są takie: oba ciągi są niemalejące, a ponadto pierwszy ciąg (`lastuknot + 1` liczb podanych w tablicy `uknots`) jest podciągiem drugiego ciągu (`lastvknot + 1` liczb w tablicy `vknots`). Jeśli warunki te są spełnione, to wartością procedury jest `true` (czyli 1),

a w przeciwnym razie false (czyli 0).

```
int mbs_BuildOsloMatrixProfilef ( int degree,
                                   int lastuknot, const float *uknots,
                                   int lastvknot, const float *vknots,
                                   bandm_profile *prof );
```

Procedura `mbs_BuildOsloMatrixProfilef` konstruuje na podstawie parametrów (określających stopień reprezentacji, `degree`) i dwa ciągi węzłów (zobacz wyżej opis procedury `mbs_OsloKnotsCorrectf`) profil (tj. opis rozmieszczenia niezerowych współczynników) macierzy zmiany reprezentacji. Profil ten jest umieszczany w tablicy `prof`, która musi mieć długość co najmniej `lastuknot - degree + 1` (o 1 większą niż liczba kolumn macierzy).

Wartością procedury jest liczba niezerowych współczynników macierzy, tj. długość tablicy potrzebnej do ich przechowywania.

```
void mbs_BuildOsloMatrixf ( int degree, int lastuknot,
                            const float *uknots,
                            const float *vknots,
                            const bandm_profile *prof, float *a );
```

Procedura `mbs_BuildOsloMatrixf` oblicza współczynniki macierzy zmiany reprezentacji, za pomocą algorytmu Oslo. Parametry procedury to: `degree` — stopień reprezentacji, `uknots` — tablica węzłów reprezentacji początkowej, o długości `lastuknot + 1`, `vknots` — tablica węzłów reprezentacji z dodatkowymi węzłami. Liczba tych węzłów jest wyznaczana na podstawie zawartości tablic, dlatego nie ma określającego ją parametru. Ciągi węzłów w tych tablicach muszą spełniać warunki sprawdzane przez procedurę `mbs_OsloKnotsCorrectf`.

Tablica `prof` zawiera opis struktury macierzy, który musi być skonstruowany wcześniej, za pomocą procedury `mbs_BuildOsloMatrixProfilef`. Do tablicy `a` o długości obliczonej przez procedurę `mbs_BuildOsloMatrixProfilef` procedura `mbs_BuildOsloMatrixf` wpisuje współczynniki macierzy przejścia.

```
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 );
```

Procedura `mbs_multiOsloInsertKnotsf` służy do jednoczesnego wstawienia *wielu* węzłów do reprezentacji `ncurves` krzywych B-sklejanych położonych w przestrzeni o wymiarze `spdimen`. Stopień krzywych jest określony przez wartość parametru `degree`. Reprezentacja początkowa składa się z węzłów umieszczonych w tablicy `inknots` (jest ich `inlastknot + 1`) oraz łamanych kontrolnych umieszczonych w tablicy `inctlpoints` o podziałce `inpitch`.

Końcowa reprezentacja krzywych ma być oparta na ciągu węzłów o długości  $\text{outlastknot} + 1$  podanym w tablicy `outknots`, przy czym ciąg węzłów początkowej reprezentacji musi być podciągiem tego ciągu.

Działanie procedury polega na wyznaczeniu za pomocą algorytmu Oslo odpowiedniej macierzy, a następnie pomnożeniu jej przez macierz utworzoną z punktów kontrolnych danych krzywych.

Jeśli wartości parametrów `inlastknot` i `outlastknot` są równe, to procedura, przy założeniu, że oba ciągi węzłów są identyczne (co *nie jest* sprawdzane), kopiuje dane z tablicy `inctlpoints` do `outctlpoints` (z uwzględnieniem podziałek tych tablic, określonych przez parametry `inpitch` i `outpitch`).

```
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 );
```

Procedura `mbs_multiOsloRemoveKnotsLSQf` służy do jednoczesnego usunięcia *wielu* węzłów z reprezentacji `ncurves` krzywych B-sklejanych położonych w przestrzeni o wymiarze `spdimen`. Stopień krzywych jest określony przez wartość parametru `degree`. Reprezentacja początkowa składa się z węzłów umieszczonych w tablicy `inknots` (jest ich  $\text{inlastknot} + 1$ ) oraz łamanych kontrolnych umieszczonych w tablicy `inctlpoints` o podziałce `inpitch`.

Końcowa reprezentacja krzywych ma być oparta na ciągu węzłów o długości  $\text{outlastknot} + 1$  podanym w tablicy `outknots`, przy czym musi to być podciąg ciągu początkowego. Ponadto krotność żadnego węzła w ciągu końcowym nie może być większa niż  $\text{degree} + 1$  (ponieważ w przeciwnym razie macierz opisana wyżej miałaby kolumny liniowo zależne).

Działanie procedury polega na wyznaczeniu za pomocą algorytmu Oslo odpowiedniej macierzy, a następnie rozwiązaniu liniowego zadania najmniejszych kwadratów z tą macierzą.

Jeśli wartości parametrów `inlastknot` i `outlastknot` są równe, to procedura, przy założeniu, że oba ciągi węzłów są identyczne (co *nie jest* sprawdzane), kopiuje dane z tablicy `inctlpoints` do `outctlpoints` (z uwzględnieniem podziałek tych tablic, określonych przez parametry `inpitch` i `outpitch`).

#### 7.7.4 Maksymalne wstawianie węzłów

Procedury opisane w tym punkcie służą do wstawienia węzłów do reprezentacji krzywych i płatów B-sklejanych w taki sposób, aby krotność każdego węzła wewnętrznego była o 1 większa od stopnia. W ten sposób powstaje szczególna reprezentacja B-sklejana, która składa się z reprezentacji każdego łuku wielomianowego

w bazie Bernsteina lokalnej zmiennej, tj. reprezentacja kawałkami Béziera. Reprezentacja taka umożliwia m.in. szybkie obliczanie punktów krzywych (za pomocą schematu Hornera) i działania algebraiczne (mnożenie) funkcji sklejanych i krzywych. Procedury te nie usuwają zbędnych węzłów i punktów kontrolnych, które mogą pozostać po wstawieniu węzłów. Procedury, które to robią (czyli dają wynik „czysty”) są opisane w następnym punkcie.

```
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 );
```

Procedura `mbs_multiMaxKnotInsf` wstawia węzły do reprezentacji `ncurves` krzywych B-sklejanych stopnia `degree` w przestrzeni o wymiarze `spdimen`.

Początkowa reprezentacja krzywych jest opisana przez parametry `inlastknot` (indeks ostatniego węzła), `inctlpoints` (tablica zawierająca węzłów), `inpitch` (podziałka, tj. odległość początków danych łamanych kontrolnych) i `inctlpoints` (tablica zawierająca współrzędne punktów kontrolnych kolejnych krzywych).

Procedura wyznacza reprezentacje krzywych odpowiadające ciągowi węzłów, w którym każdy węzeł wewnętrzny (zobacz p. 7.1.3) ma krotność `degree+1`, a węzły brzegowe mają krotności `degree` lub `degree + 1`.

Krotności węzłów zewnętrznych nie ulegają zmianie, w związku z czym może powstać reprezentacja, która zawiera zbędne węzły i punkty kontrolne. Parametry `*skipl` i `*skipr` na wyjściu z procedury określają liczbę zbędnych węzłów i punktów kontrolnych odpowiednio z lewej i prawej strony.

Nowa reprezentacja krzywej jest umieszczana w tablicach `outknots` (ciąg węzłów, indeks ostatniego węzła jest zwracany poprzez parametr `outlastknot`) i `outctlpoints` (łamane kontrolne, początki łamanych kolejnych krzywych są od siebie oddzielone liczbą, która jest wartością parametru *wejściowego* `outpitch`).

Jeśli w początkowej reprezentacji krzywych występują węzły o krotności większej niż docelowa, to w pierwszym kroku są one usuwane (z roboczej kopii reprezentacji krzywych), za pomocą procedury `mbs_multiRemoveSuperfluousKnots`). Następnie jest wywoływana procedura `mbs_multiOsloInsertKnotsf`), która dokonuje wstawienia węzłów za pomocą algorytmu Oslo.

Długości tablic potrzebnych do pomieszczenia nowej reprezentacji krzywej można obliczyć za pomocą procedury `mbs_LastknotMaxInsf`, która oblicza indeks ostatniego węzła nowej reprezentacji.

```

#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) ...

```

Cztery makra wywołujące procedurę `mbs_multiMaxKnotInsf` w celu wyznaczenia reprezentacji B-sklejanej funkcji lub krzywej skleianej w przestrzeni dwu-, trój- i czterowymiarowej, w której wszystkie węzły wewnętrzne mają krotność o 1 większą niż stopień, czyli reprezentacji kawałkami Béziera. Parametry muszą spełniać warunki podane w opisie procedury `mbs_multiMaxKnotInsf`.

### 7.7.5 Konwersja krzywych i płatów do postaci kawałkami Béziera

Opisane tu procedury dokonują konwersji krzywych i płatów B-sklejanych do postaci kawałkami Béziera, za pomocą procedury `mbs_multiMaxKnotInsf`. Można ich używać do wygodnego rysowania krzywych lub płatów.

```

void mbs_multiBSCurvesToBez ( 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 );

```

Procedura `mbs_multiBSCurvesToBez` dokonuje konwersji `ncurves` krzywych B-sklejanych stopnia `degree`, położonych w przestrzeni o wymiarze `spdimen` do postaci kawałkami Béziera.

Parametry opisujące daną reprezentację krzywych to `lastinknot` (indeks ostatniego węzła), `inknots` (tablica z ciągiem węzłów), `inpitch` i `inctlp` (podziałka i tablica z punktami kontrolnymi krzywych).

Wartość parametru `*kpcs` po wyjściu z procedury jest równa liczbie wielomianowych łuków, z których składa się każda krzywa. Parametr `*lastoutknot` jest indeksem ostatniego węzła w ciągu należącym do wynikowej reprezentacji krzywych, tablica `outknots` zawiera ten ciąg węzłów, parametr *wejściowy* `outpitch` określa

podziałkę tablicy outctlp, w której procedura umieszcza punkty kontrolne Béziera poszczególnych łuków wielomianowych. Dokładniej, w tablicy tej każdej z krzywych B-sklejanych odpowiada  $*kpcs*(degree+1)*spdimen$  liczb zmiennopozycyjnych; każdy łuk wielomianowy jest reprezentowany przez kolejne  $(degree+1)*spdimen$  liczb (opisujących  $degree+1$  punktów); wartość parametru outpitch określa odległość w tablicy początków reprezentacji pierwszego łuku każdej z krzywych B-sklejanych.

Jeśli parametr kpcs, lastoutknot lub outknots jest wskaźnikiem pustym, to procedura pomija przykazywanie odpowiedniej informacji.

```
#define mbs_BSToBezC1f(degree,lastinknot,inknots,incoeff,kpcs, \
    lastoutknot,outknots,outcoeff) \
    mbs_multiBSCurvesToBez(1,1,degree,lastinknot,inknots,0,incoeff,\
    kpcs,lastoutknot,outknots,0,outcoeff)
#define mbs_BSToBezC2f(degree,lastinknot,inknots,inctlp,kpcs, \
    lastoutknot,outknots,outctlp) \
    mbs_multiBSCurvesToBez(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) ...
```

Podane wyżej makra wywołują procedurę mbs\_multiBSCurvesToBez w celu otrzymania reprezentacji kawałkami Béziera *jednej* funkcji skleianej lub krzywej B-sklejanej w przestrzeni dwu-, trój- lub czterowymiarowej. Parametrów tych makr mają takie samo znaczenie jak parametry procedury mbs\_multiBSCurvesToBez o tych samych nazwach.

```
void mbs_BSPatchToBez ( 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 );
```

Procedura mbs\_BSPatchToBez znajduje reprezentację płata B-sklejanego z węzłami o krotnościach o 1 większych od stopni płata ze względu na oba parametry, czyli reprezentacje Béziera wielomianowych kawałków płata. Reprezentacja taka

nadaje się m.in. do wygodnego wyświetlania płata. Procedura dopuszcza płyty o brzegach swobodnych.

Parametr `spdimen` określa wymiar przestrzeni, w której jest położony płat. Stopień płata jest określony przez parametry `degreeu` i `degreev`, ciągi węzłów reprezentacji danej są reprezentowane przez parametry `lastuknot`, `uknots`, `lastvknot` i `vknots`, a punkty kontrolne są podane w tablicy `inctlp` o podziałce (odległości początków kolejnych kolumn siatki kontrolnej) będącej wartością parametru `inpitch`.

Parametry `*kupcs` i `*kvpcs` służą do przekazania informacji o liczbie wielomianowych kawałków płata; płat składa się z `*kupcs` „pasów”, z których każdy składa się z `*kvpcs` płatów wielomianowych. Parametry `lastoutuknot`, `outuknots`, `lastvknot` i `outvknots` służą do przekazania przez procedurę ciągów węzłów końcowej reprezentacji płata. Jeśli dowolny z tych parametrów jest wskaźnikiem pustym (NULL), to odpowiednia informacja nie jest wyprowadzana przez procedurę (dla potrzeb rysowania płatów idane te są zbędne).

Punkty kontrolne wynikowej reprezentacji płata są umieszczane w tablicy `outctlp` o podziałce `outpitch` (Uwaga: to jest parametr wejściowy). Podziałka ta powinna być większa lub równa  $d(m+1)k_v$ , gdzie liczba  $d$  jest wymiarem przestrzeni (wartość parametru `spdimen`),  $m$  (wartość parametru `degreev`) jest stopniem płata ze względu na parametr  $v$ , zaś  $k_v$  jest liczbą odcinków, na które węzły w ciągu „ $v$ ” dzielą przedział  $[v_m, v_{M-m}]$  (liczba  $M$  jest wartością parametru `lastvknot`). Liczba  $k_v$  jest przypisywana przez procedurę parametrowi `kvpcs`, ale można ją otrzymać wcześniej, za pomocą procedury `mbs_NumKnotIntervalsf`.

Liczba kolumn końcowej reprezentacji płata jest równa  $(n+1)k_u$ , gdzie  $n$  jest stopniem płata ze względu na parametr  $u$ , zaś  $k_u$  jest liczbą „pasów”, z których składa się płat. Też można ją obliczyć zawczasu, wywołując procedurę `mbs_NumKnotIntervalsf`.

Właściwe obliczenia (przede wszystkim wstawianie węzłów) wykonuje procedura `mbs_multiMaxKnotInsf`.

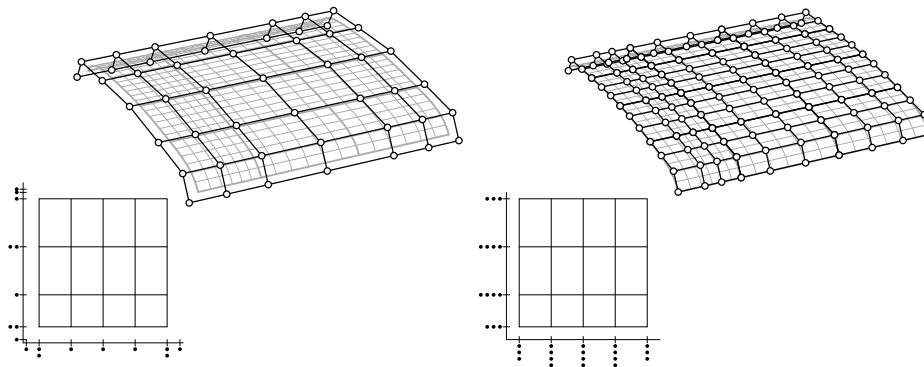
**Przykład.** Założymy, że płat jest określony wzorem (7.12) za pomocą dwóch niemalejących ciągów węzłów,  $u_0, \dots, u_N$  i  $v_0, \dots, v_M$ , umieszczonych odpowiednio w tablicach `u` i `v`. Stopień płata jest równy  $n$  ze względu na parametr  $u$  i  $m$  ze względu na parametr  $v$ . Punkty kontrolne płata leżą w przestrzeni  $d$ -wymiarowej (tej samej co płat) i są ustawione w kolumnach, w tablicy `cp`. Kolumna  $i$ -ta dla  $i \in \{0, \dots, N-n-1\}$  składa się z  $M-m$  punktów, a zatem jest ona reprezentowana przez  $(M-m)d$  liczb zmiennopozycyjnych.

```
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_BSPatchToBez ( d, n, N, u, m, M, v, d*(M-m), cp,
                  &ku, NULL, NULL, &kv, NULL, NULL, pitch, b );
```



Po wykonaniu przedstawionego wyżej fragmentu programu w tablicy *b* mamy punkty kontrolne Béziera płatów wielomianowych, z których składa się dany płat B-sklejany. Aby do tablicy *c* (o długości co najmniej  $(n+1)(m+1)d$  liczb zmien-nopozycyjnych) przenieść „spakowaną” siatkę kontrolną (tj. punkty kontrolne w kolejnych kolumnach bez przerw) *j*-tego płata z *i*-tego pasa (licząc od zera), można wykonać instrukcje

```
md = (m + 1)d;          /* długość kolumny płata Béziera */
start = (n + 1)i*pitch + md*j; /* położenie pierwszego punktu */
pkv_Selectf ( n + 1, md, pitch, md, &b[start], c );
```



Rys. 7.6. Płat B-sklejany i jego reprezentacja kawałkami Béziera

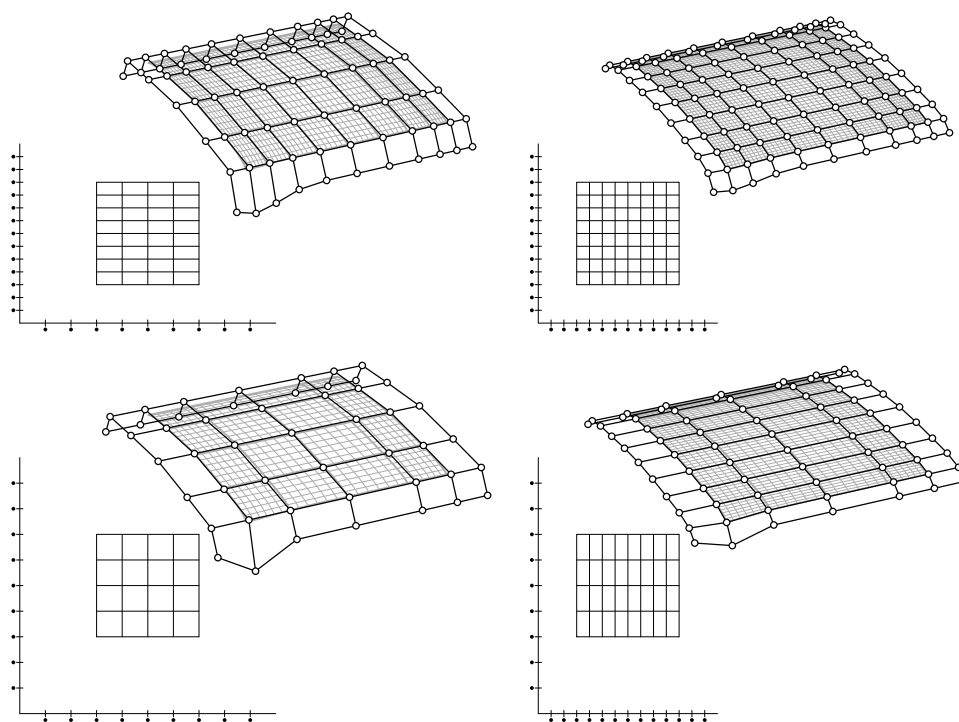
## 7.8 Algorytm Lane’a-Riesenfelda

```
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) ...

```



Rys. 7.7. Zastosowanie algorytmu Lane'a-Riesenfelda do płata B-sklejanego

## 7.9 Podział krzywych i płatów Béziera na części

Podział krzywych i płatów Béziera na części można wykonać za pomocą algorytmu de Casteljau, przy czym są dwa przypadki zaprogramowane osobno. W pierwszym następuje podział dziedziny na dwie równe części (np. podział odcinka  $[0.25, 0.5]$  na odcinki  $[0.25, 0.375]$  i  $[0.375, 0.5]$ ), dzięki czemu cały algorytm polega na obliczaniu tylko średnich arytmetycznych liczb. W drugim przypadku dziedzina może być podzielona na odcinki lub prostokąty w dowolnym miejscu, co umożliwia również dokonanie ekstrapolacji.

```
void mbs_multiBisectBezCurvesf ( int degree, int ncurves,
                                int spdimen, int pitch,
                                float *ctlp, float *ctlq );
```

Procedura `mbs_multiBisectBezCurvesf` dokonuje podziału `ncurves` krzywych Béziera stopnia `degree` w przestrzeni o wymiarze `spdimen`. Dziedzina (odcinek  $[a, b]$ ) jest dzielona w połowie.

Punkty kontrolne podaje się w tablicy `ctlp` o podziałce `pitch`. Po wykonaniu obliczenia tablica ta zawiera punkty kontrolne drugiego łuku (związanego z przedziałem  $[\frac{a+b}{2}, b]$ ). W tablicy `ctlq` procedura umieszcza punkty kontrolne reprezentacji krzywych związanej z odcinkiem  $[0, 0.5]$ . Podziałka tej tablicy jest taka sama jak podziałka tablicy `ctlp` (czyli równa wartości parametru `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) ...
```

Powyższe makra wywołują procedurę `mbs_multiBisectBezCurvesf` w celu podzielenia dziedziny jednego wielomianu lub krzywej Béziera (dwu-, trój- lub cztero-wymiarowej) na połowy i wyznaczenia lokalnych reprezentacji tego wielomianu lub krzywej.

```
#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) ...
```

Powyższe makra wywołują procedurę `mbs_multiBisectBezCurvesf` w celu podzielenia prostokątnej dziedziny jednego wielomianu dwóch zmiennych (danego w bazie tensorowej Bernsteina) lub płata Béziera (dwu-, trój- lub czterowymiarowego) na przystające części i wyznaczenia lokalnych reprezentacji tego wielomianu lub płata. Makra z literą *u* w nazwie służą do podziału przedziału zmienności pierwszego argumentu wielomianu lub płata, a makra z literą *v* do podziału przedziału zmienności drugiego argumentu.

```
void mbs_multiDivideBezCurvesf ( int degree, int ncurves,
                                int spdimen, int pitch, float t,
                                float *ctlp, float *ctlq );
```

Procedura `mbs_multiDivideBezCurvesf` dokonuje podziału `ncurves` krzywych Béziera stopnia `degree` w przestrzeni o wymiarze `spdimen`. Dziedzina (odcinek  $[0, 1]$ ) jest dzielona w punkcie  $t$ , który jest wartością parametru  $t$ , przy czym jeśli  $t \notin [0, 1]$ , to podział jest w rzeczywistości ekstrapolacją.

Punkty kontrolne podaje się w tablicy `ctlp` o podziałce `pitch`. Po wykonaniu obliczenia tablica ta zawiera punkty kontrolne drugiego łuku (związanego z przedziałem  $[t, 1]$ ). W tablicy `ctlq` procedura umieszcza punkty kontrolne reprezentacji krzywych związanej z odcinkiem  $[0, t]$ . Podziałka tej tablicy jest taka sama jak podziałka tablicy `ctlp` (czyli równa wartości parametru `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) ...
```

Powyższe makra wywołują procedurę `mbs_multiDivideBezCurvesf` w celu podzielenia dziedziny jednego wielomianu lub krzywej Béziera (dwu-, trój- lub czterowymiarowej) w proporcji  $t : 1 - t$ , gdzie  $t$  jest wartością parametru  $t$ , i wyznaczenia lokalnych reprezentacji tego wielomianu lub krzywej.

```

#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) ...

```

Makra wywołujące procedurę `mbs_multiDivideBezCurvesf` w celu podzielenia wielomianu dwóch zmiennych lub płata Béziera w przestrzeni dwu-, trój- i czterowymiarowej na dwa kawałki — w kierunku „u” (tj. podziałowi ulega przedział zmienności pierwszego argumentu) albo „v” (zostaje podzielony przedział zmienności drugiego argumentu). Liczba `u` lub `v`, która jest wartością parametru `u` albo `v` określa punkt podziału odcinka  $[0, 1]$  — w takich proporcjach jest dzielony przedział zmienności odpowiedniego argumentu.

## 7.10 Podwyższanie stopnia

Podwyższenie stopnia jest obliczeniem nowej reprezentacji krzywej w bazie wielomianów Bernsteina lub B-sklejanej stopnia wyższego o wskazany przyrost.

### 7.10.1 Podwyższanie stopnia krzywych i płatów Béziera

```
void mbs_multiBCDegElevf ( int ncurves, int spdimen,
                           int inpitch, int indegree,
                           const float *inctlpoints,
                           int deltadeg,
                           int outpitch, int *outdegree,
                           float *outctlpoints );
```

Procedura `mbs_multiBCDegElevf` dokonuje podwyższenia stopnia `ncurves` krzywych Béziera stopnia `indegree` w przestrzeni o wymiarze `spdimen` do stopnia `indegree + deltadeg` (końcowy stopień jest przypisywany do parametru `*outdegree`).

Łamane kontrolne krzywych danych są podane w tablicy `inctlpoints`, z podziałką `inpitch`. Obliczone łamane kontrolne krzywych procedura wpisuje do tablicy `outctlpoints` z podziałką `outpitch`.

```
#define mbs_BCDElevC1f(indegree,incoeff,deltadeg, \
    outdegree,outcoeff) \
    mbs_multiBCDegElevf ( 1, 1, 0, indegree, incoeff, deltadeg, \
        0, outdegree, outcoeff )
#define mbs_BCDElevC2f(indegree,inctlpoints,deltadeg, \
    outdegree,outctlpoints) \
    mbs_multiBCDegElevf ( 1, 2, 0, indegree, (float*)inctlpoints, \
        deltadeg, 0, outdegree, (float*)outctlpoints )
#define mbs_BCDElevC3f(indegree,inctlpoints,deltadeg, \
    outdegree,outctlpoints) ...
#define mbs_BCDElevC4f(indegree,inctlpoints,deltadeg, \
    outdegree,outctlpoints) ...
```

Powyższe cztery makra służą do podwyższenia stopnia jednego wielomianu (określonego za pomocą współczynników w bazie Bernsteina) lub krzywej Béziera położonej w przestrzeni dwu-, trój- lub czterowymiarowej. Parametry tych makr są przedstawione w opisie procedury `mbs_multiBCDegElevf`.

```
void mbs_BCDegElevPf ( int spdimen,
                      int indegreeu, int indegreev,
                      const float *inctlp,
                      int deltadegu, int deltadegv,
                      int *outdegreeu, int *outdegreev,
                      float *outctlp );
```

Procedura `mbs_BCDegElevPf` dokonuje podwyższenia stopnia płata Béziera położonego w przestrzeni o wymiarze `spdimen`, ze względu na jeden lub oba parametry.

Parametry `indeg` i `indegv` określają początkowy stopień płata ze względu na każdy z parametrów. Tablica `inctlp` zawiera współczynniki wielomianu lub punkty kontrolne płata, należące do kolejnych kolumn. Tablica ta jest spakowana, tj. bez obszarów nieużywanych, a zatem jej podziałka, czyli odległość początków kolejnych kolumn jest równa długości kolumny:  $(\text{indeg} + 1) * \text{spdimen}$ . W podobny sposób jest pakowana tablica `outctlp`, zawierająca obliczone przez procedurę punkty kontrolne reprezentacji płata o wyższym stopniu.

Parametry `deltadegu` i `deltadegv` muszą być nieujemne. Określają one o ile ma być podwyższony stopień płata ze względu na każdy z jego parametrów. Końcowy stopień (suma stopnia początkowego i przyrostu) jest przypisywany do parametrów `*outdeg` i `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) ...
```

Powyższe makra dokonują podwyższenia stopnia wielomianu dwóch zmiennych danego w bazie tensorowej Bernsteina lub płata Béziera położonego w przestrzeni dwu-, trój- lub czterowymiarowej, wywołując procedurę `mbs_BCDegElevPf`.

### 7.10.2 Podwyższanie stopnia krzywych i płatów B-sklejanych

```
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 );
```

Procedura `mbs_multiBSDegElevf` dokonuje podwyższenia stopnia `ncurves` krzywych B-sklejanych stopnia `indegree` w przestrzeni o wymiarze `spdimen`, do stopnia `indegree + deltadeg`, który jest przypisywany do parametru `*outdegree`.

Procedura dopuszcza krzywe o końcach zaczepionych lub swobodnych jako dane wejściowe. Jeśli parametr `freeend` ma wartość `false`, to wynikiem jest krzywa o końcach zaczepionych, w której reprezentacji jedynymi węzłami zewnętrznymi są węzły skrajne (zobacz p. 7.1.3). Jeśli wartość parametru `freeend` jest `true`, to reprezentacja wynikowa jest krzywa o końcach swobodnych. Ciąg węzłów tej reprezentacji powstaje przez zwiększenie krotności wszystkich węzłów o wartość parametru `deltadeg`, a następnie odrzucenie tylu węzłów z początku i końca otrzymanego ciągu, aby otrzymać  $\hat{u}_n < \hat{u}_{n+1}$  i  $\hat{u}_{N-n} > \hat{u}_{N-n-1}$  ( $n$  oznacza tu stopień reprezentacji wynikowej, a  $N$  oznacza indeks ostatniego węzła tej reprezentacji).

Sposób podwyższania stopnia nie zależy od wartości parametru `freeend`. Reprezentacja istniejąca bezpośrednio po podwyższeniu stopnia jest o końcach zaczepionych. Dla `freeend=true` procedura wywołuje `mbs_multiBSChangeLeftKnotsf` i `mbs_multiBSChangeRightKnotsf`. Wiąże się to z dodatkowymi błędami zaokrągleń. Procedura `mbs_multiBSDegElevf` może być użyta do podwyższenia stopnia krzywej zamkniętej; parametr `freeend` powinien mieć wtedy wartość `true`, dzięki czemu wynikowa reprezentacja krzywej będzie zamknięta, co oznacza m.in., że odpowiednia liczba początkowych punktów kontrolnych pokrywa się z punktami końcowymi, z dokładnością do błędów zaokrągleń.

Początkowa reprezentacja znajduje się w tablicach `inknots` (węzły, jest ich `inlastknot + 1`) oraz `inctlpoints` (punkty kontrolne, podziałka tej tablicy jest równa `inpitch`).

Końcowa reprezentacja jest wpisywana do tablic `outknots` (węzły, ich liczba jest przypisywana do `*outlastknot`) i `outctlpoints` (punkty kontrolne, podziałka tej tablicy jest równa `outpitch`).

Należy zadbać o dostateczną pojemność tablic, w których ma być umieszczony wynik. Reguła jest taka: jeśli ostatni węzeł krzywej stopnia  $n$  ma indeks  $N$ , liczba wielomianowych łuków krzywej jest równa  $l$  (można ją znaleźć przez wywołanie procedury `mbs_NumKnotIntervalsf`), zaś reprezentacja którą chcemy znaleźć ma



stopień  $n'$ , to ostatni węzeł tej reprezentacji ma indeks

$$N' = N + (l + 1 - d_0 - d_1)(n' - n),$$

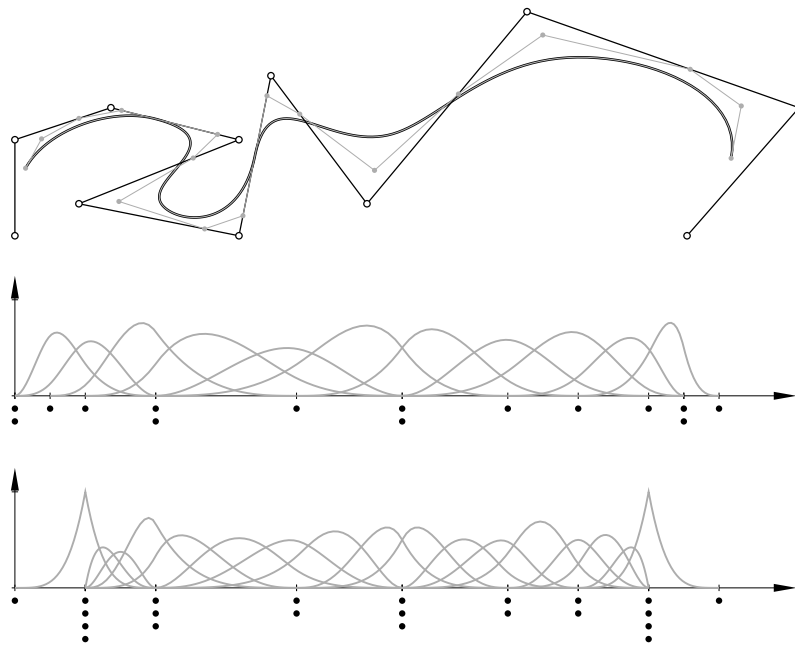
gdzie  $d_0$  i  $d_1$  są liczbami, takimi że

$$u_n = \dots = u_{n+d_0} < u_{n+d_0+1} \quad \text{oraz} \quad u_{N-n-d_1-1} < u_{N-n-d_1} = \dots = u_{N-n}.$$

Liczba punktów kontrolnych nowej reprezentacji krzywej jest równa  $N' - n'$ . Dla  $m$  krzywych położonych w przestrzeni o wymiarze  $d$  należy zarezerwować tablicę o pojemności  $N' + 1$  liczb na węzły i tablicę o pojemności  $(N' - n)md$  liczb (na współrzędne punktów kontrolnych).

```
#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) ...
```

Cztery makra, które wywołują procedurę `mbs_multiBSDegElevf` w celu podwyższenia stopnia jednej skalarnej funkcji sklejaney lub krzywej B-sklejaney w przestrzeni dwu-, trój- lub czterowymiarowej.



Rys. 7.8. Podwyższenie stopnia płaskiej krzywej B-sklejanej z 3 do 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 );
```

Procedura `mbs_multiBSDegElevClosedf` dokonuje podwyższenia stopnia zamkniętych krzywych B-sklejanych.

Parametr `ncurves` określa liczbę krzywych, parametr `spdimen` określa wymiar przestrzeni, w której one są położone.

Parametry `indegree`, `inlastknot`, `inknots`, `inpitch`, `inctlpoints` opisują dane wejściowe — odpowiednio stopień  $n$ , indeks ostatniego węzła  $N$ , ciąg węzłów  $u_0, \dots, u_N$ , podziałkę tablicy punktów kontrolnych i punkty kontrolne.

Parametr `deltadeg` (musi mieć nieujemną wartość) określa różnicę stopnia reprezentacji wynikowej i danej.

Parametry `*outdegree` i `*outlastknot` są zmiennymi, do których procedura wpisze stopień reprezentacji wynikowej i numer jej ostatniego węzła. W tablicy `outknots` procedura umieści ciąg węzłów tej reprezentacji. Parametr `outpitch` określa podziałkę tablicy `outctlpoints`, w której procedura umieści punkty kontrolne krzywych w tej reprezentacji.

Numer ostatniego węzła wynikowej reprezentacji krzywej zamkniętej jest równy

$$N' = N + (l + 1 + r - d_0 - d_1)(n' - n),$$

gdzie liczba  $r$  jest krotnością węzła  $u_n$  w danej reprezentacji stopnia  $n$  (z pominięciem węzła  $u_0$ ), a  $d_0$  i  $d_1$  są liczbami, takimi że

$$u_n = \dots = u_{n+d_0} < u_{n+d_0+1} \quad \text{oraz} \quad u_{N-n-d_1-1} < u_{N-n-d_1} = \dots = u_{N-n}.$$

```
#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) ...
```

Cztery makra, które wywołują procedurę `mbs_multiBSDegElevClosedf` w celu podwyższenia stopnia jednej zamkniętej krzywej B-sklejanej położonej w przestrzeni o wymiarze odpowiednio 1,2,3,4. Parametry makr odpowiadają parametrom procedury o tych samych nazwach, w związku z czym ich opisy można znaleźć w opisie procedury.

### Przykład — podwyższenie stopnia płata B-sklejanego

Stopień płata można podwyższyć ze względu na pierwszy parametr („u”) lub drugi („v”). Sposoby wywoływania odpowiednich procedur w obu przypadkach, pokazane w przykładzie poniżej, opierają się na założeniu, że wszystkie siatki kontrolne płata (tj. początkowa i docelowa) są „spakowane”, czyli podziałka każdej tablicy, w której są przechowywane punkty kontrolne jest równa długości reprezentacji (liczbie liczb zmiennopozycyjnych) jednej kolumny.

Mamy zatem liczby  $n$  i  $m$  określające stopień początkowej reprezentacji płata, liczby  $N$  i  $M$  określające długości ciągów węzłów dla tej reprezentacji, tablice `uknots` i `vknots` (o długościach  $N + 1$  i  $M + 1$ ) zawierające węzły i tablicę `ctlp` zawierającą  $(N - n)(M - m)d$  liczb zmiennopozycyjnych, które są współrzędnymi

punktów kontrolnych płata. Podziałka tablicy, czyli długość każdej kolumny, jest równa  $(M - m)d$ .

Aby podwyższyć stopień ze względu na parametr „u” możemy potraktować ten płat jak krzywą B-sklejaną w przestrzeni o wymiarze  $(M - m)d$ . Zatem, obliczamy długości potrzebnych tablic, rezerwujemy pamięć i dokonujemy podwyższenia stopnia (w tym przykładzie o 1):

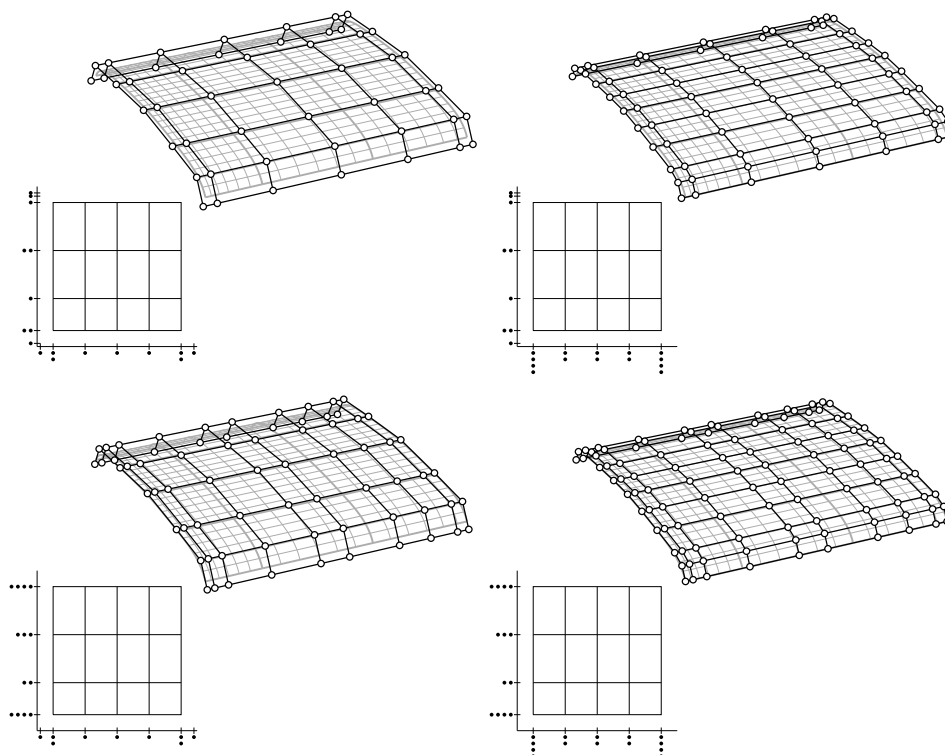
```
ku = mbs_NumKnotIntervalsf ( n, N, uknots );
for ( d0 = 0; uknots[n + d0 + 1] == uknots[n]; d0++ )
    ;
for ( d1 = 0; uknots[N - n - d1 - 1] == uknots[N - n]; d1++ )
    ;
ua = pkv_GetScratchMemf ( N + 2 + ku - d0 - d1 );
cpa = pkv_GetScratchMemf ( (N - n + ku - d0 - d1)(M - m)d );
mbs_multiBSDegElevf ( 1, (M - m)d, n, N, uknots, 0, ctlp, 1,
                      &na, &Na, ua, 0, cpa, false );
```

Podziałki tablic `cp` i `cpa` są nieistotne (odpowiednie parametry są równe 0), bo mamy tu podwyższenie stopnia tylko jednej krzywej. Do zmiennych `na` i `Na` procedura przypisuje stopień (równy  $n + 1$ ) i indeks ostatniego węzła (równy  $N + ku - d0 - d1 + 1$ ) otrzymanej reprezentacji płata. Stopień ze względu na parametr „v” oraz ciąg węzłów związanych z tym parametrem są identyczne jak w reprezentacji początkowej płata.

Podwyższenie stopnia płata ze względu na parametr „v” jest równoważne podwyższeniu stopnia krzywych B-sklejanych reprezentowanych przez kolumny siatki kontrolnej. Odpowiedni kod, dokonujący podwyższenia stopnia o 1, wygląda tak:

```
kv = mbs_NumKnotIntervalsf ( m, M, vknots );
for ( d0 = 0; vknots[m + d0 + 1] == vknots[m]; d0++ )
    ;
for ( d1 = 0; vknots[M - m - d1 - 1] == vknots[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 );
```

W razie potrzeby podwyższenia stopnia o więcej niż 1, można wykonać przedstawione wyżej procedury kilkakrotnie, ale znacznie szybciej i z mniejszymi błędami zaokrągleń można otrzymać wynik podając odpowiedni parametr `deltadeg`. Wymaga to właściwego obliczenia długości i podziałek tablic potrzebnych do pomieszczenia poszukiwanych reprezentacji płata. Odpowiednie wskazówki są podane w opisie procedury `mbs_multiBSDegElevf`



Rys. 7.9. Przykład podwyższania stopnia płata B-sklejanego

## 7.11 Obniżanie stopnia

Obniżanie stopnia krzywej B-sklejanej jest zadaniem aproksymacyjnym (podobnie jak usuwanie węzła). Jego celem jest otrzymanie krzywej B-sklejanej  $\tilde{s}$  stopnia  $\tilde{n} = n - d$  (dla  $d \in \{1, \dots, n\}$ ), która przybliży daną krzywą  $s$  stopnia  $n$ . W tej konstrukcji należy arbitralnie przyjąć ciąg węzłów krzywej wynikowej, i może to mieć to duży wpływ na kształt tej krzywej. Następujące założenia są chyba oczywiste:

- Krzywa wynikowa musi mieć tę samą dziedzinę, co krzywa dana.
- Jeśli krzywa dana  $s$  powstała przez podwyższenie o  $d$  stopnia krzywej  $\tilde{s}$  stopnia  $n'$ , to wynikiem obniżania stopnia powinna być krzywa  $\tilde{s}$ .

W konstrukcjach zaimplementowanych w procedurach opisanych w tym punkcie zbiór węzłów krzywej wynikowej jest podzbiorem zbioru węzłów krzywej danej, przy czym reguła określania krotności tych węzłów jest taka: niech pewien węzeł  $u_i$  w reprezentacji krzywej danej ma krotność  $r$ . Jeśli  $r \leq d$ , to krotność  $\tilde{r}$  tego

węzła w reprezentacji wynikowej jest równa 1. Jeśli  $d < r \leq n + 1$ , to  $\tilde{r} = r - d$ , a jeśli  $r > n + 1$ , to  $\tilde{r} = n - d + 1$ .

Dla krzywej niezamkniętej ciąg węzłów otrzymany na podstawie opisanej wyżej reguły jest modyfikowany w taki sposób, aby otrzymać ciąg  $\tilde{u}_0, \dots, \tilde{u}_{\tilde{N}}$ , taki że  $\tilde{u}_{n'} < \tilde{u}_{n'+1}$  oraz  $\tilde{u}_{\tilde{N}-n'-1} < \tilde{u}_{\tilde{N}-n'-1}$ . W tym celu na początku i na końcu pewne węzły mogą zostać odrzucone lub dopisane (dopisywany jest węzeł początkowy lub końcowy ciągu).

Następnie wyznaczany jest pomocniczy ciąg węzłów  $\hat{u}_0, \dots, \hat{u}_{\hat{N}}$ , w którym występują wszystkie węzły ciągu wynikowego, przy czym krotności tych węzłów są większe o  $d$  od krotności węzłów w ciągu wynikowym  $\tilde{u}_0, \dots, \tilde{u}_{\tilde{N}}$ . Przez wstawienie węzłów (algorytmem Oslo, za pomocą procedury `mbs_multiOsloInsertKnotsf`) i usunięcie nadmiarowych węzłów (o krotności większej niż  $n + 1$ , za pomocą procedury `mbs_multiRemoveSuperfluousKnotsf`) otrzymywana jest reprezentacja danej krzywej  $s$  oparta na ciągu węzłów pomocniczych:

$$s(t) = \sum_{i=0}^{N-n-1} d_i N_i^n(t) = \sum_{i=0}^{\hat{N}-n-1} \hat{d}_i \hat{N}_i^n(t).$$

Następnie konstruowana jest macierz  $A$ , która opisuje podwyższenie o  $d$  stopnia krzywej B-sklejanej stopnia  $n'$  opartej na ciągu węzłów  $\tilde{u}_0, \dots, \tilde{u}_{\tilde{N}}$ . Punkty kontrolne  $\tilde{d}_0, \dots, \tilde{d}_{\tilde{N}-n'-1}$  krzywej wynikowej  $\tilde{s}$  są obliczane przez rozwiązanie liniowego zadania najmniejszych kwadratów dla układu równań

$$Ax = b,$$

w którym  $x = [\tilde{d}_0, \dots, \tilde{d}_{\tilde{N}-n'-1}]^T$  oraz  $b = [\hat{d}_0, \dots, \hat{d}_{\hat{N}-n-1}]^T$ .

```
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 );
```

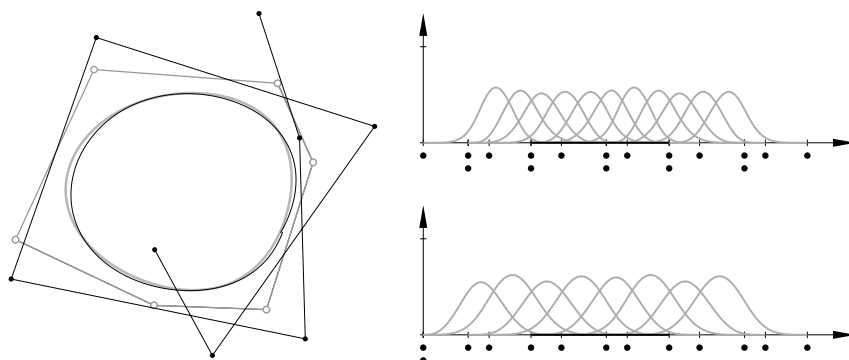
Procedura `mbs_multiBSDegRedf` dokonuje obniżenia stopnia B-sklejanych krzywych niezamkniętych, zgodnie z opisem podanym wyżej. Parametry wejściowe opisują: `ncurves` — liczbę krzywych, `spdimen` — wymiar przestrzeni, w której leżą krzywe, `indegree` — stopień  $n$ , `inlastknot` — indeks  $N$  ostatniego węzła reprezentacji danej, `inknots` — ciąg węzłów reprezentacji danej (w tablicy o długości  $N+1$ ), `inpitch` — podziałkę tablicy z punktami kontrolnymi krzywych danych, `deltadeg` — liczbę  $d$ , o którą należy obniżyć stopień.

Parametry wyjściowe: `*outdegree` — zmienna, której zostanie przypisany stopień  $n'$  krzywych wynikowych, `*outlastknot` — zmienna, której zostanie przypisany indeks  $\hat{N}$  ostatniego węzła ciągu wynikowego, `outknots` — tablica, do której zostaną wpisane węzły wynikowe  $\tilde{u}_0, \dots, \tilde{u}_{\tilde{N}}$ , `outpitch` — podziałka tablicy

outctlpoints, do której procedura wstawi punkty kontrolne krzywych wynikowych.

**Uwaga:** Nie ma obecnie osobnej procedury, która oblicza długość wynikowego ciągu węzłów i której można by użyć w celu zaalokowania tablic na te węzły i na punkty kontrolne o odpowiednich długościach. Zanim to zostanie zrobione, trzeba podać tablice z miejscem na zapas. Także i podziałkę tablicy outctlpoints na razie wysysa się z palca.

Wartością procedury jest true jeśli obniżanie stopnia zakończyło się sukcesem, a false w przeciwnym razie. W razie błędu procedura wywołuje jednak procedurę pkv\_SignalError, której domyślne działanie powoduje zatrzymanie programu.



Rys. 7.10. Obniżanie stopnia krzywej B-sklejanej z 5 do 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) ...
```

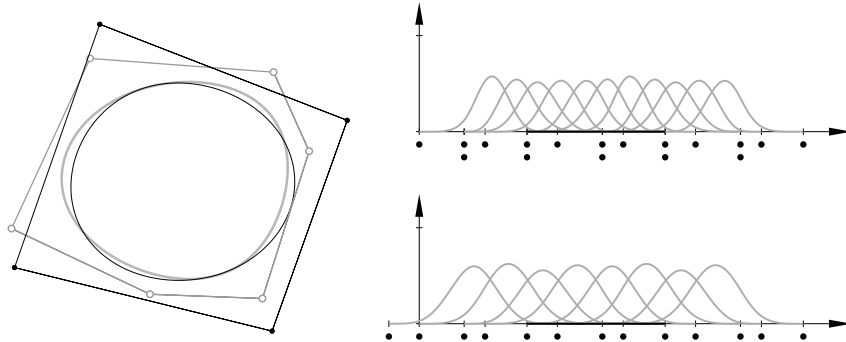
Powyższe makra służą do wywołania procedury mbs\_multiBSDegRedf w celu obniżenia stopnia jednej krzywej w przestrzeni o wymiarze 1,...,4.

```
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 );
```

Procedura `mbs_multiBSDegRedClosedf` dokonuje obniżenia stopnia B-sklejanej krzywej zamkniętej. Jej parametry mają opisy identyczne jak parametry procedury `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) ...
```

Powyższe makra służą do wywołania procedury `mbs_multiBSDegRedClosedf` w celu obniżenia stopnia jednej krzywej zamkniętej położonej w przestrzeni o wymiarze  $1, \dots, 4$ .



Rys. 7.11. Obniżanie stopnia zamkniętej krzywej B-sklejanej z 5 do 4



## 7.12 Działania algebraiczne na funkcjach i krzywych sklepanych

Zadaniem opisanych w tym punkcie procedur jest wyznaczanie B-sklejanej reprezentacji sumy (wektorowych) krzywych B-sklejanych oraz iloczynu (skalarnych) funkcji i krzywych (wektorowych). Działania te mogą być potrzebne w różnych konstrukcjach, na przykład powierzchni wykazujących ciągłość geometryczną.

### 7.12.1 Dodawanie funkcji i krzywych sklepanych

Dodanie, tj. wyznaczenie reprezentacji sumy, krzywych B-sklejanych wymaga określenia stopnia i ciągu węzłów tej reprezentacji. Stopień reprezentacji sumy jest największym stopniem składników. Ciąg węzłów jest określony przez ciągi węzłów składników, które muszą wyznaczać tę samą dziedzinę. Aby dodać krzywe (które muszą leżeć w przestrzeni o tym samym wymiarze), trzeba znaleźć reprezentacje tych krzywych, o stopniu i ciągu węzłów, które będą używane do reprezentowania sumy (i to zadanie pomocnicze jest najbardziej skomplikowanym i kosztownym elementem algorytmu dodawania). Ostatnim i najprostszym elementem procedury jest sumowanie współczynników (tj. punktów kontrolnych) składników.

```
boolean mbs_FindBSCommonKnotSequencef ( int *degree, int *lastknot,
                                         float **knots, int nsequences, ... );
```

Procedura `mbs_FindBSCommonKnotSequencef` otrzymuje  $k$  ciągów węzłów, będących podstawą reprezentacji krzywych B-sklejanych podanych stopni. Zadaniem procedury jest znalezienie minimalnego stopnia i ciągu węzłów, wystarczającego do reprezentowania sumy tych krzywych. Znaleziony stopień  $n$  jest największym spośród podanych stopni krzywych, lub początkową wartością zmiennej `*degree`, jeśli jest większa (można zatem wymusić poszukiwanie reprezentacji wyższego stopnia). Znaleziony ciąg węzłów ma następujące własności:

- Węzły brzegowe mają krotność  $n+1$  (zatem węzły zewnętrzne, w tym skrajne, pokrywają się z węzłami brzegowymi).
- Występują w nim wszystkie węzły wewnętrzne ciągów węzłów danych.
- Krotności węzłów wewnętrznych są dobrane tak, aby po podwyższeniu stopnia każdej krzywej do  $n$  można ją było reprezentować z tym ciągiem węzłów.

Parametry `degree`, `lastknot` i `knots` służą do wyprowadzenia wyników (z uwagi na składnię C występują one na początku listy parametrów, co jest odstępstwem od konwencji przyjętej w całym pakiecie `BSTools`). Zmienne wskazywane przez te parametry otrzymują wartości, którymi są odpowiednio stopień reprezentacji, numer ostatniego węzła w znalezionym ciągu i wskaźnik do tablicy z tymi węzłami.

**Uwaga:** Procedura rezerwuje tę tablicę na stosie pamięci pomocniczej, a zatem wywołujący ją podprogram jest odpowiedzialny za zwolnienie tej rezerwacji (za pomocą procedury `pkv_FreeScratchMem` albo `pkv_SetScratchMemTop`).

Parametr `nsequences` określa liczbę  $k$  danych ciągów węzłów (musi być  $k \geq 1$ ). W wywołaniu procedury należy po nim podać  $3k$  parametrów. Kolejne trójki parametrów opisują kolejne ciągi. Pierwszym elementem trójki jest stopień  $n_i$  krzywej (typu `int`), drugim elementem trójki jest indeks  $N_i$  ostatniego węzła (typu `int`), a trzeci element trójki jest wskaźnikiem tablicy z  $N_i + 1$  liczbami zmiennopozycyjnymi reprezentującymi węzły (parametr ten ma typ `float*`).

We wszystkich ciągach danych węzeł o numerze  $n_i$  musi być identyczny; to samo dotyczy węzła o numerze  $N_i - n_i$ .

Wartością procedury jest `true`, jeśli obliczenie zakończyło się sukcesem, albo `false` w przeciwnym razie. Przyczyną niepowodzenia mogą być błędne dane lub brak miejsca na stosie pamięci pomocniczej.

```
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 );
```

Procedura `mbs_multiAdjustBSCRepf` „uzgadnia” reprezentację krzywych B-sklejanych, tj. wyznacza reprezentację potrzebnego stopnia, opartą na wskazanym ciągu węzłów. Polega to na podwyższeniu stopnia, jeśli stopień początkowy jest za mały, a następnie wstawieniu „brakujących” węzłów (za pomocą algorytmu Oslo). Aby dodać k krzywych B-sklejanych o różnych reprezentacjach (ale o tej samej dziedzinie), należy wyznaczyć stopień i ciąg węzłów ich sumy (za pomocą `mbs_FindBSCCommonKnotSequencef`), a następnie wyznaczyć odpowiednią reprezentację każdego składnika, wywołując procedurę `mbs_multiAdjustBSCRepf`.

Parametry: `ncurves` — liczba krzywych, `spdimen` — wymiar przestrzeni, w której one leżą, `indegree`, `inlastknot`, `inknots` — stopień, indeks ostatniego węzła i tablica węzłów reprezentacji danej, `inpitch` — podziałka tablicy `inctlpoints`, w której są podane punkty kontrolne reprezentacji danej.

Parametry `outdegree`, `outlastknot` i `outknots` opisują stopień i ciąg węzłów reprezentacji, którą procedura ma wyznaczyć. Punkty kontrolne tej reprezentacji są wpisywane do tablicy `outctlpoints` o podziałce `outpitch`.

Wartością procedury jest `true` w razie sukcesu oraz `false` w razie porażki (spowodowanej błędnymi danymi lub brakiem miejsca na stosie pamięci pomocniczej).

```

#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 );

```

Procedura `mbs_multiAddBSCurvesf` oblicza sumy `ncurves` par krzywych B-sklejanych w przestrzeni o wymiarze `spdimen`.

Pierwsza krzywa w parze jest określona za pomocą parametrów `degree1` (stopień), `lastknot1` (indeks ostatniego węzła), `knots1` (tablica węzłów) i `ctlpoints1` (tablica punktów kontrolnych o podziałce `pitch1`).

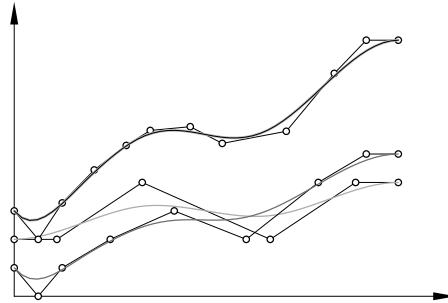
Druga krzywa w parze jest podobnie określona za pomocą parametrów `degree2`, `lastknot2`, `knots2`, `pitch2` i `ctlpoints2`.

Parametry wyjściowe to `*sumdeg` (otrzymuje wartość stopnia reprezentacji sumy), `*sumlastknot` (indeks ostatniego węzła reprezentacji sumy), `sumknots` (tablica, do której procedura wstawia węzły reprezentacji sumy), `sumctlpoints` (tablica o podziałce `sumpitch`, do której procedura wstawia punkty kontrolne sumy).

```

#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) ...

```



Rys. 7.12. Funkcje sklejane stopni 3 i 4 i ich suma

```

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 Przejście między bazami Bernsteina i skalowanymi

W tym punkcie są opisane procedury pomocnicze używane do mnożenia funkcji i krzywych sklejanych.

Aby pomnożyć wielomiany dane za pomocą współczynników w bazach Bernsteina, wygodnie jest przejść do tzw. *baz skalowanych*. Baza skalowana stopnia  $n$  składa się z wielomianów

$$b_i^n(t) \stackrel{\text{def}}{=} \frac{1}{\binom{n}{i}} B_i^n(t) = t^i (1-t)^{n-i}. \quad (7.23)$$

Zatem współczynniki wielomianu w bazie skalowanej otrzymamy mnożąc współczynniki tego wielomianu w bazie Bernsteina przez liczby  $\binom{n}{i}$ .

Wynik mnożenia wielomianów reprezentowanych w bazach skalowanych stopnia  $n$  i  $m$  otrzymujemy w postaci układu współczynników w bazie skalowanej stopnia  $n+m$ , po czym możemy przejść do bazy Bernsteina stopnia  $n+m$  wykonując odpowiednie dzielenia.

```
void mbs_multiBezScalef ( int degree, int narcs,
                          int ncurves, int spdimen,
                          int pitch, float *ctlpoints );
```

Procedura `mbs_multiBezScalef` otrzymuje tablicę krzywych Béziera stopnia `degree` w przestrzeni o wymiarze `spdimen` i oblicza współczynniki tych krzywych w bazie skalowanej. Przyjęte jest założenie, że krzywe te to kolejne łuki krzywych B-sklejanych, które zostały otrzymane przez odpowiednie wstawienie węzłów (np. za pomocą procedury `mbs_multiMaxKnotInsf`).

Parametry: `degree` — stopień krzywych, `narcs` — liczba łuków Béziera wchodzących w skład każdej krzywej B-sklejanej, `ncurves` — liczba krzywych B-sklejanych, `spdimen` — wymiar  $d$  przestrzeni, w której leżą krzywe.

Parametr `pitch` określa podziałkę tablicy `ctlpoints`, w której przed wywołaniem procedury są podane punkty kontrolne krzywych (tj. ich współczynniki w bazach Bernsteina stopnia  $n = \text{degree}$ ), a po jej zakończeniu współczynniki w bazach skalowanych. Wartość parametru `pitch` określa odległość początków pierwszych punktów kontrolnych kolejnych krzywych B-sklejanych. Reprezentacje kolejnych krzywych Béziera zajmują zawsze  $(n+1)d$  miejsc i nie ma między nimi wolnych miejsc. Podziałka tablicy nie może być mniejsza niż  $(n+1)d \cdot \text{narcs}$ .

```
void mbs_multiBezUnscalef ( int degree, int narcs,
                            int ncurves, int spdimen,
                            int pitch, float *ctlpoints );
```

Procedura `mbs_multiBezUnscalef` otrzymuje tablicę reprezentacji krzywych wielomianowych w bazie skalowanej i dokonuje przejścia do bazy Bernsteina, tj. do reprezentacji Béziera. Parametry tej procedury (z wyjątkiem opisu początkowej i końcowej zawartości tablicy `ctlpoints`) są identyczne jak parametry procedury `mbs_multiBezScalef`.

### 7.12.3 Mnożenie funkcji i krzywych sklejaných

W tym punkcie są opisane procedury mnożenia wielomianowych i sklejaných krzywych (tj. funkcji wektorowych) przez skalarne funkcje wielomianowe i sklejanę. Dane dla procedur składają się z reprezentacji jednej lub większej liczby funkcji skalarnych (wielomianów lub funkcji sklejaných jednej zmiennej)  $s_i$  oraz jednej lub większej liczby krzywych (wielomianowych lub sklejaných)  $v_i$ . Obie te liczby muszą być równe, lub jedna z nich musi być równa 1. Procedury obliczają reprezentację Béziera lub B-sklejaną funkcji wektorowych

$$\mathbf{w}_i(t) = s_i(t)\mathbf{v}_i(t),$$

przy czym jeśli jest tylko jedna funkcja skalarna  $s_0$  i więcej funkcji wektorowych, to przyjmuje się, że każda z tych funkcji wektorowych będzie pomnożona przez tę jedną funkcję  $s_0$  i podobnie, jeśli jest wiele funkcji skalarnych  $s_i$  i jedna funkcja wektorowa  $v_0$ , to obliczane są iloczyny tej jednej funkcji wektorowej i wszystkich funkcji skalarnych.

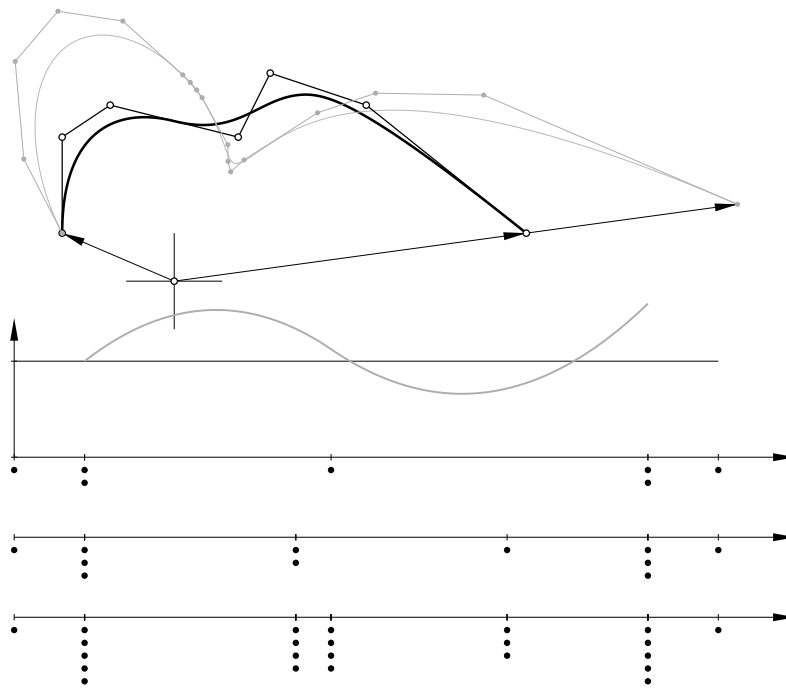
Opisane tu procedury mogą znaleźć zastosowanie w różnych zaawansowanych konstrukcjach. Najprostsza z nich to podwyższenie stopnia krzywej, przez pomnożenie jej przez odpowiednio reprezentowaną funkcję stałą  $s_0(t) = 1$  (stopień reprezentacji funkcji  $s_0$  jest różnicą stopni reprezentacji wynikowej i danej przetwarzanej funkcji wektorowej). W tym przypadku lepiej jest jednak użyć procedury podwyższania stopnia (np. `mbs_multiBSDegElevf`), która realizuje zręczniejszy algorytm.

```
int mbs_BSPProdRepSizef ( int degree1, int lastknot1,
                          const float *knots1,
                          int degree2, int lastknot2,
                          const float *knots2 );
```

Procedura `mbs_BSPProdRepSizef` otrzymuje *dwa* ciągi węzłów, `knots1` o długości `lastknot1+1` oraz `knots2` o długości `lastknot2+1`. Pierwszy z tych ciągów służy do określenia funkcji sklejanę stopnia `degree1`, a drugi — funkcji sklejanę stopnia `degree2`. Ciągi te powinny określać taką samą dziedzinę krzywych sklejaných. Wartością procedury jest indeks ostatniego elementu najkrótszego ciągu węzłów wystarczającego do reprezentowania iloczynu dowolnych funkcji sklejaných określonych dla podanych ciągów węzłów.

```
void mbs_SetBSPProdKnotsf ( int degree1, int lastknot1,
                            const float *knots1,
                            int degree2, int lastknot2,
                            const float *knots2,
                            int *degree, int *lastknot,
                            float *knots );
```

Procedura `mbs_SetBSPProdKnotsf` otrzymuje dwa ciągi węzłów i tworzy nowy ciąg, który jest minimalnym ciągiem wystarczającym do reprezentowania iloczynu



Rys. 7.13. Mnożenie płaskiej krzywej wektorowej przez funkcję sklejającą

dowolnych funkcji sklejanych określonych dla ciągów danych.

```
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 *prodcf );
```

Procedura `mbs_multiMultBezCf` wykonuje mnożenie wielomianów reprezentowanych w bazie wielomianów Bernsteina stopnia `degscf` i wielomianowych funkcji wektorowych (krzywych Béziera) stopnia `degvecf`. Parametr `spdimen` określa wymiar przestrzeni, w której leżą te krzywe. Liczba funkcji skalarnych jest określona przez parametr `nscf`, a liczba krzywych jest wartością parametru `nvecf`. Liczba obliczonych iloczynów jest równa większej z tych liczb, zobacz uwagi na początku tego punktu.

Tablica `scfcoeff` zawiera współczynniki wielomianów w bazie Bernsteina, przy czym współczynniki każdego wielomianu zajmują kolejne miejsca w tablicy, a jej podziałka (czyli różnica indeksów pierwszych współczynników kolejnych dwóch wie-



lomianów) jest podana jako wartość parametru `scfpitch`. Podobnie, parametr `vecfpitch` określa podziałkę tablicy `vecfcp`, która zawiera wektorowe współczynniki krzywych (każdy współczynnik składa się z kolejnych `spdimen` liczb).

Iloczyny są reprezentowane w bazie Bernsteina stopnia równego sumie stopni reprezentacji argumentów (tj. `degscf + degvecf`), który to stopień jest zwracany poprzez parametr `degprod`. Reprezentacje poszczególnych iloczynów składają się z ciągów `spdimen * (stopień + 1)` liczb, które procedura wpisuje do tablicy `prodcp` z podziałką `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 );
```

Procedura `mbs_multiMultBSCf` oblicza reprezentacje iloczynów `nscf` skalarnych funkcji sklejanych  $s_i$  i `nvecf` wektorowych funkcji sklejanych  $v_i$ , przy czym liczby te mogą być różne i wtedy jedna z nich musi być równa 1 (zobacz uwagi na początku tego punktu).

Funkcje skalarne są określone za pomocą parametrów `degscf` (stopień reprezentacji), `scflastknot` i `scfknots` (indeks ostatniego węzła i tablica z tymi węzłami), `scfcoeff` i `scfpitch` (tablica współczynników w bazie B-sklejanej poszczególnych funkcji i podziałka tej tablicy).

Funkcje wektorowe, w przestrzeni o wymiarze `spdimen`, są podobnie reprezentowane przez parametry `degvecf`, `vecflastknot`, `vecfknots`, `vecfpitch` i `vecfcp`.

Wynik jest wpisywany do tablic `prodknots` (węzły) i `prodcp` (wektorowe współczynniki w bazie B-sklejanej stopnia równego sumie stopni reprezentacji argumentów, jest on zwracany poprzez parametr `degprod`). Podziałka tej ostatniej tablicy jest określona przez parametr `prodpitch`. Początkowa wartość parametru `*prodlastknot` określa ilość miejsca w tablicy `prodknots` (ma być o 1 większa od wartości tego parametru). Należy ją obliczyć i utworzyć odpowiednią tablicę *przed* wywołaniem procedury `mbs_multiMultBSCf`, najlepiej jest użyć do tego procedury `mbs_BSProdRepSizef`, która bada ciągi węzłów argumentów mnożenia, podane jej jako parametry.

### 7.12.4 Wyznaczanie płata opisującego wektory normalne

```
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 );
```

Procedura `mbs_BezP3Normalf` oblicza punkty kontrolne płata  $\mathbf{n} = \mathbf{p}_u \wedge \mathbf{p}_v$ , opisującego wektor normalny danego wielomianowego płata Béziera  $\mathbf{p}$  stopnia  $(n, m)$  w  $\mathbb{R}^3$ . Parametry `degreeu = n` i `degreev = m` określają stopień płata danego. Jego punkty kontrolne są podane w tablicy `ctlpoints`, która zawiera kolejne kolumny tego płata bez przerw.

Obliczony płat wyjściowy ma stopień `*ndegu = 2n - 1` i `*ndegv = 2m - 1`, a jego punkty kontrolne są wpisywane do tablicy `ncp` (bez przerw między kolumnami).

Wartość procedury `mbs_BezP3Normalf` jest równa 0 w razie niepowodzenia (błędne parametry lub brak pamięci), albo 1, jeśli obliczenie zakończyło się sukcesem.

Procedura `mbs_BezP3NormalDeg` oblicza stopień płata opisującego wektor normalny. Może ona być użyteczna w celu zarezerwowania bloku pamięci o wielkości odpowiedniej do przechowania punktów kontrolnych płata  $\mathbf{n}$ .

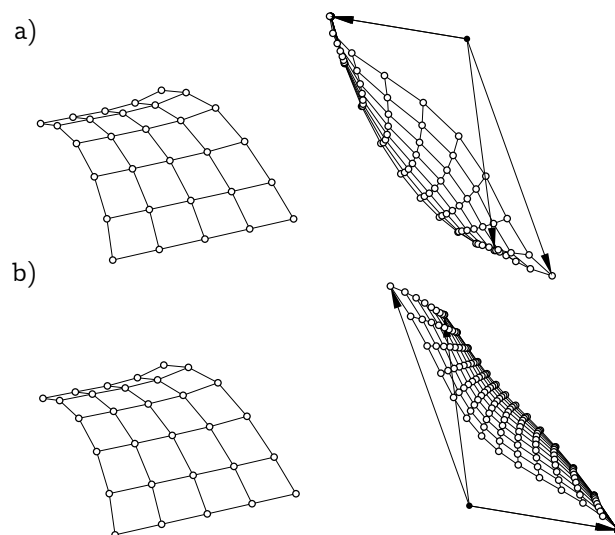
```
void mbs_BezP3RNormalDeg ( int degreeu, int degreev,
                           int *ndegu, int *ndegv );
char mbs_BezP3RNormalf ( int degreeu, int degreev,
                         const point4f *ctlpoints,
                         int *ndegu, int *ndegv, vector3f *ncp );
```

Procedura `mbs_BezP3RNormalf` oblicza punkty kontrolne płata  $\mathbf{n}$  opisującego wektor normalny danego wymiernego płata Béziera  $\mathbf{p}$  stopnia  $(n, m)$  w  $\mathbb{R}^3$ . Punkty te powstają przez odrzucenie ostatniej współrzędnej punktów kontrolnych płata  $\mathbf{N} = \mathbf{P} \wedge \mathbf{P}_u \wedge \mathbf{P}_v$  w  $\mathbb{R}^4$ . Parametry `degreeu = n` i `degreev = m` określają stopień płata danego. Punkty kontrolne jego jednorodnej reprezentacji są podane w tablicy `ctlpoints`, która zawiera kolejne kolumny tego płata bez przerw.

Obliczony płat wyjściowy ma stopień `*ndegu = 3n - 2` i `*ndegv = 3m - 2`, a jego punkty kontrolne są wpisywane do tablicy `ncp` (bez przerw między kolumnami).

Wartość procedury `mbs_BezP3RNormalf` jest równa 0 w razie niepowodzenia (błędne parametry lub brak pamięci), albo 1, jeśli obliczenie zakończyło się sukcesem.

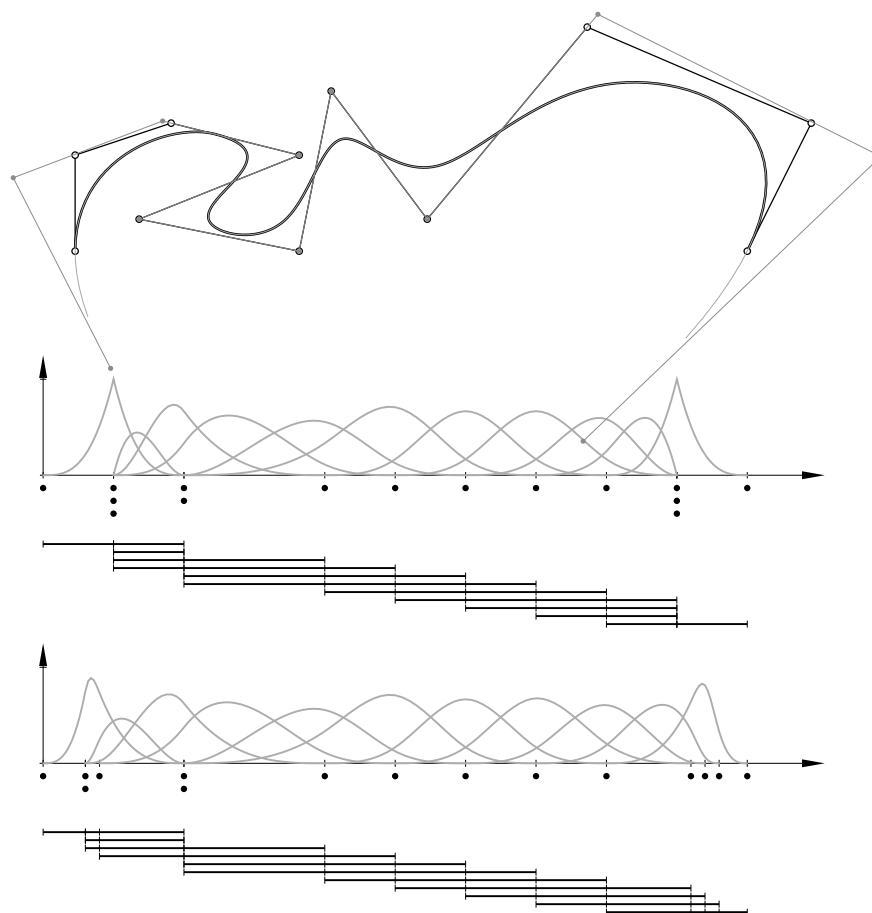
Procedura `mbs_BezP3RNormalDeg` oblicza stopień płata opisującego wektor normalny. Może ona być użyteczna w celu zarezerwowania bloku pamięci o wielkości odpowiedniej do przechowania punktów kontrolnych płata  $\mathbf{n}$ .



Rys. 7.14. Siatki kontrolne płatów Béziera i ich płatów opisujących wektory normalne: a) płat wielomianowy, b) wymierny

### 7.13 Zmiana węzłów krzywych B-sklejanych na końcach

Makra służą do wywołania procedury `mbs_multiBSChangeLeftKnotsf` albo `mbs_multiBSChangeRightKnotsf` w razie potrzeby zastosowania jej do jednej krzywej B-sklejanej w przestrzeni o wymiarze  $1, \dots, 4$ .



Rys. 7.15. Krzywa B-sklejana przed i po zmianie reprezentacji na końcach

## 7.14 Konstrukcje krzywych interpolacyjnych

Konstrukcja krzywych interpolacyjnych jest czasem głównym zadaniem, ale też przydaje się do rozwiązania zadań takich jak wyznaczanie powierzchni rozpinanych i wypełniających.

### 7.14.1 Konstrukcja krzywych interpolacyjnych trzeciego stopnia

W tym punkcie jest opisana procedura wyznaczania B-sklejanej reprezentacji krzywych interpolacyjnych *trzeciego stopnia*. Węzły interpolacyjne (podane jako parametr wejściowy) będą węzłami krzywych, przy czym skrajne węzły interpolacyjne w wyznaczonej reprezentacji B-sklejanej będą miały krotność 3, a do tego dojdą jeszcze jednokrotne węzły dodatkowe, które są potrzebne w definicji reprezentacji B-sklejanej.

Oprócz węzłów i warunków interpolacyjnych należy podać *warunki brzegowe*. Rodzaje warunków brzegowych obsługiwanych przez procedurę w obecnej wersji są opisane dalej.

```
void mbs_multiBSCubicInterpf ( int lastinterpknott,
                               float *interpknotts,
                               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 );
```

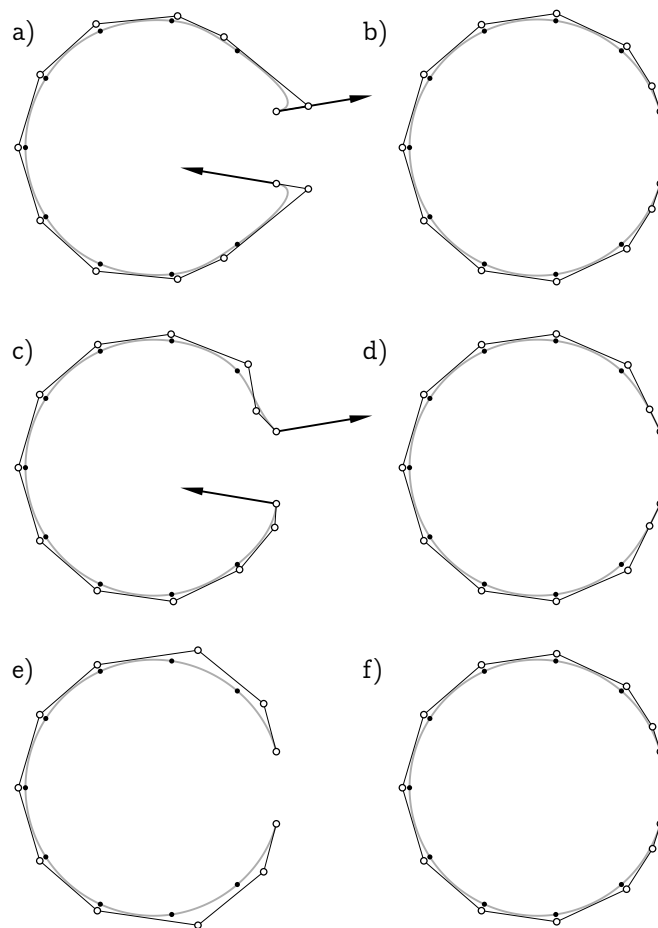
Procedura `mbs_multiBSCubicInterpf` służy do konstruowania B-sklejanych krzywych interpolacyjnych trzeciego stopnia, klasy  $C^2$ .

Parametry: `lastinterpknott` określa indeks ostatniego węzła interpolacyjnego, oznaczmy go literą  $N$ . Węzły interpolacyjne  $u_0, \dots, u_N$ , które tworzą ciąg ściśle rosnący, podaje się w tablicy `interpknotts`.

Parametry `ncurves` i `spdimen` określają liczbę krzywych i wymiar przestrzeni, w której leżą te krzywe. Tablica `x` zawiera współrzędne punktów przez które mają przechodzić skonstruowane krzywe; dla każdej krzywej trzeba podać w niej `spdimen(lastinterpknott+1)` liczb rzeczywistych. Podziałka tej tablicy (tj. różnica indeksów pierwszej liczby w danych dla kolejnych krzywych) jest równa `xpitch`.

Parametr `ypitch` określa podziałkę tablic `ybcl` i `ybcr`, w których należy podać dane liczbowe określające warunki brzegowe.

Parametry `bcl` i `bcr` służą do wybrania warunków brzegowych odpowiednio na lewym i prawym końcu dziedziny krzywych; wszystkie krzywe mają warunki



Rys. 7.16. B-sklejane krzywe interpolacyjne trzeciego stopnia. Węzłami są liczby  $0, 1, \dots, 10$ . Warunki brzegowe krzywej na rysunku a) były określone przez podanie wektorów pochodnych na końcach, na rysunku b) krzywa jest określona za pomocą warunków Bessela c) dane wektory pochodnych drugiego rzędu, d) naturalna krzywa sklejana, e) warunek nie-węzeł, f) pochodna trzeciego rzędu na końcach równa 0

tego samego rodzaju, ale na każdym końcu warunek może być inny. Dopuszczalne wartości tych parametrów są zdefiniowane w pliku `multibs.h` jako makra i obecnie są następujące:

`BS3_BC_FIRST_DER` — warunek brzegowy jest określony przez podanie wektora pochodnej każdej krzywej w pierwszym albo ostatnim węźle interpolacyjnym (tj.  $u_0$  albo  $u_N$ ). Współrzędne tych wektorów dla wszystkich krzywych należy podać w tablicy `ybc1` (jeśli w pierwszym) albo `ybcr` (jeśli w ostatnim węźle). Zatem,

tablice `ybc1` i `ybc2` zawierają dla każdej krzywej `spdimen` liczb rzeczywistych, które są współrzędnymi tych wektorów.

`BS3_BC_FIRST_DER0` — warunek brzegowy określony jak wyżej, z wektorem zerowym pochodnej w odpowiednim węźle interpolacyjnym. Parametr `ybc1` lub `ybc2` jest ignorowany, może więc mieć wartość `NULL`.

`BS3_BC_SECOND_DER` — warunek brzegowy jest określony przez podanie wektora pochodnej drugiego krzywej sklejanej rzędu w węźle  $u_0$  albo  $u_N$ . Współrzędne tego wektora (albo wektorów, jeśli liczba krzywych jest większa niż 1) są podane w tablicy odpowiednio `ybc1` lub `ybc2`.

`BS3_BC_SECOND_DER0` — warunek brzegowy jest określony jak wyżej, z wektorem zerowym pochodnej drugiego rzędu w odpowiednim węźle. Parametr `ybc1` lub `ybc2` jest ignorowany, może więc mieć wartość `NULL`.

Krzywa określona z tym warunkiem na obu końcach to tzw. **naturalna krzywa sklejana**.

`BS3_BC_THIRD_DER` — warunek brzegowy jest określony przez podanie wektorów pochodnej trzeciego rzędu krzywych. Współrzędne tych wektorów są podane w tablicy `ybc1` albo `ybc2`.

`BS3_BC_THIRD_DER0` — warunek brzegowy jest określony przez żądanie, aby pochodna trzeciego rzędu krzywych na jednym lub drugim końcu była wektorem zerowym. Ponieważ pochodna trzeciego rzędu krzywej wielomianowej trzeciego stopnia jest stała, więc oznacza to, że pierwszy lub ostatni łuk wielomianowy krzywej jest łukiem paraboli. Parametr `ybc1` lub `ybc2` w przypadku zadania tego warunku jest ignorowany, może mieć zatem wartość `NULL`.

`BS3_BC_BESSEL` — warunek brzegowy jest tzw. warunkiem Bessela. Wektor pochodnej krzywej w pierwszym lub ostatnim węźle jest wektorem pochodnej wielomianowej krzywej interpolacyjnej drugiego stopnia, opartej na pierwszych trzech albo ostatnich trzech węzłach interpolacyjnych konstruowanej krzywej.

Parametr `ybc1` albo `ybc2` w przypadku określenia warunku Bessela jest ignorowany, a zatem może mieć wartość `NULL`.

`BS3_BC_NOT_A_KNOT` — warunek nie-węzeł; węzeł interpolacyjny  $u_1$  albo  $u_{N-1}$  nie jest węzłem krzywej sklejanej, tj. łuki wielomianowe krzywej łączą się w tym węźle z ciągłością  $C^\infty$ . Parametr `ybc1` albo `ybc2` jest ignorowany, może mieć więc wartość `NULL`.

Reprezentacja krzywej interpolacyjnej skonstruowanej przez procedurę jest opisana za pomocą następujących parametrów: `*lastbsknot` — indeks ostatniego węzła krzywej sklejanej, `bsknots` — tablica węzłów (to są węzły interpolacyjne, ale węzły  $u_0$  i  $u_N$  mają w tej tablicy krotność 3 i są dołączone dwa dodatkowe węzły wymagane w definicji krzywych B-sklejanych). Parametr wejściowy `bspitch` oznacza podziałkę tablicy `ctlpoints`, w której procedura umieszcza punkty kontrolne kolejnych krzywych interpolacyjnych.



### 7.14.2 Konstrukcja krzywych interpolacyjnych Hermite'a

W tym punkcie są opisane procedury dokonująca bardzo szczególnej konstrukcji: znajdują krzywe Béziera i B-sklejane stopnia  $n$ , spełniające warunki interpolacyjne Hermite'a zadane w dwóch węzłach, 0 i 1 albo  $u_n$  i  $u_{N-n}$ . Istnieje zastosowanie, w którym takie właśnie procedury były mi potrzebne (algorytm konstrukcji w tym przypadku jest sprawniejszy niż ogólny algorytm rozwiązania zadania interpolacyjnego Hermite'a z krzywą B-sklejaną).

```
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 );
```

Procedura `mbs_multiInterp2knHermiteBezF` konstruuje `ncurves` krzywych Béziera stopnia  $n$  (stopień jest wartością parametru `degree`) w przestrzeni o wymiarze  $d$  (jest on wartością parametru `spdimen`).

Liczba warunków interpolacyjnych dla każdej krzywej w węźle 0 jest równa `nlbc`, a w węźle 1 `nrbc`, przy oba te parametry muszą być nieujemne i ich suma musi być równa  $n + 1$  (to zapewnia, że warunki interpolacyjne określają krzywe jednoznacznie).

Warunki interpolacyjne podaje się w tablicach `lbc` (dla węzła 0) i `rbc` (dla węzła 1). Początkowe  $d$  liczb w każdej z tych tablic to odpowiedni punkt pierwszej krzywej, następne  $d$  liczb opisuje wektor pochodnej, potem pochodnej drugiego rzędu itd. Dane opisujące warunki interpolacyjne następnej krzywej zaczynają się w pozycji odpowiednio `lbcpitch` i `rbcpitch`.

Punkty kontrolne skonstruowanych krzywych (czyli wynik działania procedury) są wstawiane do tablicy `ctlpoints`, której podziałka (odległość początków danych opisujących kolejne krzywe) jest wartością parametru `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 );
```

Procedura `mbs_multiInterp2knHermiteBSF` konstruuje `ncurves` B-sklejanych krzywych stopnia  $n$  (stopień jest wartością parametru `degree`) w przestrzeni o wymiarze  $d$  (jest on wartością parametru `spdimen`). Krzywa jest oparta o ciąg węzłów o długości  $N + 1$  podany w tablicy `knots` (liczba  $N$  jest wartością parametru `lastknot`).

Liczba warunków interpolacyjnych dla każdej krzywej w węźle  $u_n$  jest równa `nlbc`, a w węźle  $u_{N-n}$  `nrbc`, przy czym żaden z tych parametrów nie może być

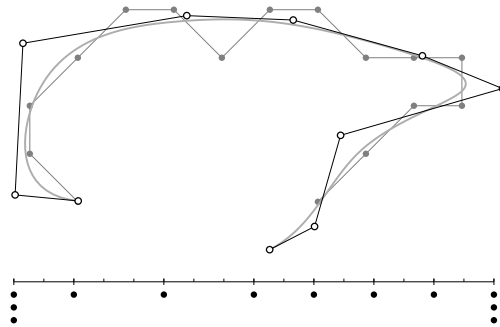
większy niż  $n$ , zaś suma ich wartości musi być równa  $N - n$  (to zapewnia, że warunki interpolacyjne określają krzywe jednoznacznie).

Ciąg węzłów w tablicy `knots` musi spełniać warunki  $u_1 = \dots = u_n < u_{n+1}$  oraz  $u_{N-n-1} < u_{N-n} = \dots = u_{N-1}$ , których procedura nie sprawdza. Warunki interpolacyjne podaje się w tablicach `lbc` (dla węzła  $u_n$ ) i `rbc` (dla węzła  $u_{N-n}$ ). Początkowe  $d$  liczb w każdej z tych tablic to odpowiedni punkt pierwszej krzywej, następne  $d$  liczb opisuje wektor pochodnej, potem pochodnej drugiego rzędu itd. Dane opisujące warunki interpolacyjne następnej krzywej zaczynają się w pozycji odpowiednio `lbcpitch` i `rbcpitch`.

Punkty kontrolne skonstruowanych krzywych (czyli wynik działania procedury) są wstawiane do tablicy `ctlpoints`, której podziałka (odległość początków danych opisujących kolejne krzywe) jest wartością parametru `pitch`.

## 7.15 Konstrukcja krzywych aproksymacyjnych

Można nałożyć liczbę warunków interpolacyjnych na funkcję lub krzywą sklejaną większą niż wymiar odpowiedniej przestrzeni. Wtedy powstaje układ równań w ogólności sprzeczny. Układ ten można rozwiązać jako liniowe zadanie najmniejszych kwadratów i w ten sposób otrzymać funkcję lub krzywą sklejaną spełniającą nałożone warunki interpolacyjne z pewnym błędem. W tym punkcie są opisane procedury wykonujące tę konstrukcję.



Rys. 7.17. Przykład płaskiej aproksymacyjnej krzywej B-sklejanej

Konstrukcję krzywej aproksymacyjnej można przeprowadzić wywołując opisaną dalej procedurę `mbs_multiConstructApproxBSCf`. Wcześniej opisane procedury są pomocnicze i w typowych aplikacjach nie będą bezpośrednio wywoływane.

```
boolean mbs_ApproxBSKnotsValidf ( int degree, int lastknot,
                                const float *knots,
                                int lastiknot, const float *iknots );
```

Procedura `mbs_ApproxBSKnotsValidf` dokonuje sprawdzenia, czy ciągi węzłów interpolacyjnych i węzłów krzywej sklejanej spełniają założenia twierdzenia Schoenberga-Whitney. Jeśli tak, to konstrukcja krzywej aproksymacyjnej jest wykonalna.

```
int mbs_ApproxBSBandmSizef ( int degree, const float *knots,
                             int lastiknot, const float *iknots );
```

Procedura `mbs_ApproxBSBandmSizef` oblicza długość tablicy potrzebnej do reprezentowania macierzy wstępowej układu równań rozwiązywanego (jako liniowe zadanie najmniejszych kwadratów) w konstrukcji B-sklejanej krzywej aproksymacyjnej.

Parametry `degree`, i `knots` opisują przestrzeń funkcji sklejanych, której elementy opisują krzywą (odpowiednio stopień i ciąg węzłów; długość tego ciągu jest określana na podstawie ciągu węzłów interpolacyjnych). Parametry `lastiknot` oraz

iknots opisują ciąg węzłów *interpolacyjnych* krzywej — warunki interpolacyjne określone w tych węzłach będą przez krzywą spełnione w przybliżeniu.

Wartością procedury jest długość tablicy, w której mają być przechowywane niezerowe współczynniki macierzy układu.

```
boolean mbs_ConstructApproxBSProfilef ( int degree, int lastknot,
                                         const float *knots,
                                         int lastiknot, const float *iknots,
                                         bandm_profile *prof );
```

Procedura `mbs_ConstructApproxBSProfilef` konstruuje profil macierzy wstępnej (zobacz p. 3.2) układu równań występującego w konstrukcji aproksymacyjnej krzywej sklejanej. Parametry `degree`, `lastknot`, `knots`, `lastiknot`, `iknots` opisują węzły krzywej sklejanej i węzły interpolacyjne (zobacz opis parametrów procedury `mbs_ApproxBSBandmSizef`).

Parametr `prof` wskazuje tablicę o długości `lastknot-degree+1`. Procedura umieszcza w niej profil macierzy.

```
boolean mbs_ConstructApproxBSMatrixf ( int degree, int lastknot,
                                         const float *knots,
                                         int lastiknot, const float *iknots,
                                         int *nrows, int *ncols,
                                         bandm_profile *prof,
                                         float *a );
```

Procedura `mbs_ConstructApproxBSMatrixf` oblicza współczynniki macierzy układu, którego rozwiązanie średniokwadratowe określa sklejaną funkcję lub krzywą aproksymacyjną.

```
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 );
```

Procedura `mbs_multiConstructApproxBSCf` konstruuje sklejane funkcje lub krzywe aproksymacyjne przez utworzenie odpowiedniego układu równań liniowych i rozwiązanie go jako liniowego zadania najmniejszych kwadratów.

Parametry: `degree` — stopień krzywej, `lastknot` — indeks ostatniego węzła, `knots` — tablica węzłów, `lastpknot` — indeks ostatniego węzła interpolacyjnego, `pknots` — tablica węzłów interpolacyjnych, `ncurves` — liczba konstruowanych krzywych, `spdimen` — wymiar przestrzeni, w której leżą krzywe.

Parametry `ppitch` i `ppoints` opisują warunki interpolacyjne; `ppitch` jest podziałką tablicy `ppoints`, która zawiera punkty, przez które mają przechodzić (w przybliżeniu) krzywe.

Parametr `bcpitch` jest podziałką tablicy `ctlpoints`, w której procedura umieszcza punkty kontrolne skonstruowanych krzywych aproksymacyjnych.

Wartością procedury jest `true` jeśli obliczenie zakończyło się sukcesem, albo `false` w przeciwnym razie. Przyczyną niepowodzenia może być niespełnienie warunku regularności zadania, wynikającego z twierdzenia Schoenberga-Whitney, albo brak pamięci.

```
#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) ...
```

Powyższe makra wywołują procedurę `mbs_multiConstructApproxBSCf` w celu otrzymania krzywej aproksymacyjnej w przestrzeni jedno-, dwu-, trój- lub cztero-wymiarowej.

## 7.16 Obcinanie krzywych Béziera

```
boolean mbs_FindPolynomialZerosf ( int degree, const float *coeff,
                                   int *nzeros, float *zeros, float eps );
```

Procedura `mbs_FindPolynomialZerosf` oblicza rzeczywiste miejsca zerowe wielomianu stopnia  $n$  w przedziale  $[0, 1]$ .

Parametry wejściowe: `degree` — stopień  $n$  wielomianu, `coeff` — współczynniki wielomianu w bazie Bernsteina stopnia  $n$ , `eps` — żądana dokładność rozwiązań (musi to być liczba dodatnia).

Parametry wyjściowe: `*nzeros` — zmienna, której procedura przypisze liczbę znalezionych zer, `*zeros` — tablica, do której miejsca te mają być wpisane. Tablica ta ma mieć długość co najmniej  $n$ .

Wartością procedury jest `true`, jeśli obliczenie zakończyło się sukcesem, `false` w przeciwnym razie, np. gdy zabrakło pamięci pomocniczej.

```
void mbs_ClipBC2f ( int ncplanes, const vector3f *cplanes,
                   int degree, const point2f *cpoints,
                   void (*output) (int degree, const point2f *cpoints) );
```

Procedura `mbs_ClipBC2f` obcina płaską wielomianową krzywą Béziera do wielokąta wypukłego, tj. oblicza i wyprowadza łuki krzywej położone wewnątrz tego wielokąta.

Parametry: `ncplanes` — liczba półpłaszczyzn, których wielokąt jest przecięciem, `cplanes` — tablica reprezentacji tych półpłaszczyzn. Dla półpłaszczyzny  $ax + by + c > 0$  liczby  $a$ ,  $b$ ,  $c$  są współrzędnymi odpowiedniego wektora w tablicy `cplanes`.

Parametry `degree` i `cpoints` opisują reprezentację płaskiej krzywej Béziera, tj. odpowiednio stopień i punkty kontrolne. Parametr `output` jest wskaźnikiem procedury, która będzie wywołana w celu wyprowadzenia (np. narysowania) poszczególnych łuków krzywej położonych wewnątrz wielokąta.

```
void mbs_ClipBC2Rf ( int ncplanes, const vector3f *cplanes,
                    int degree, const point3f *cpoints,
                    void (*output) (int degree, const point3f *cpoints) );
```

Procedura `mbs_ClipBC2Rf` obcina płaską wymierną krzywą Béziera do wielokąta wypukłego, tj. oblicza i wyprowadza łuki krzywej położone wewnątrz tego wielokąta.

Parametry: `ncplanes` — liczba półpłaszczyzn, których wielokąt jest przecięciem, `cplanes` — tablica reprezentacji tych półpłaszczyzn. Dla półpłaszczyzny  $ax + by + c > 0$  liczby  $a$ ,  $b$ ,  $c$  są współrzędnymi odpowiedniego wektora w tablicy `cplanes`.

Parametry `degree` i `cpoints` opisują reprezentację płaskiej wymiernej krzywej Béziera, tj. odpowiednio stopień i punkty kontrolne krzywej jednorodnej. Parametr

output jest wskaźnikiem procedury, która będzie wywołana w celu wyprowadzenia (np. narysowania) poszczególnych łuków krzywej położonych wewnątrz wielokąta.

```
void mbs_ClipBC3f ( int ncplanes, const vector4f *cplanes,
                  int degree, const point3f *cpoints,
                  void (*output) (int degree, const point3f *cpoints) );
```

Procedura `mbs_ClipBC3f` obcina wielomianową krzywą Béziera w przestrzeni trójwymiarowej do wielościanu wypukłego, tj. oblicza i wyprowadza łuki krzywej położone wewnątrz tego wielościanu.

Parametry: `ncplanes` — liczba półprzestrzeni, których wielościan jest przecięciem, `cplanes` — tablica reprezentacji tych półprzestrzeni. Dla półprzestrzeni  $ax + by + cz + d > 0$  liczby  $a$ ,  $b$ ,  $c$ ,  $d$  są współrzędnymi odpowiedniego wektora w tablicy `cplanes`.

Parametry `degree` i `cpoints` opisują reprezentację krzywej Béziera, tj. odpowiednio stopień i punkty kontrolne. Parametr `output` jest wskaźnikiem procedury, która będzie wywołana w celu wyprowadzenia (np. narysowania) poszczególnych łuków krzywej położonych wewnątrz wielokąta.

```
void mbs_ClipBC3Rf ( int ncplanes, const vector4f *cplanes,
                   int degree, const point4f *cpoints,
                   void (*output) (int degree, const point4f *cpoints) );
```

Procedura `mbs_ClipBC3Rf` obcina wymierną krzywą Béziera w przestrzeni trójwymiarowej do wielościanu wypukłego, tj. oblicza i wyprowadza łuki krzywej położone wewnątrz tego wielościanu.

Parametry: `ncplanes` — liczba półprzestrzeni, których wielościan jest przecięciem, `cplanes` — tablica reprezentacji tych półprzestrzeni. Dla półprzestrzeni  $ax + by + cz + d > 0$  liczby  $a$ ,  $b$ ,  $c$ ,  $d$  są współrzędnymi odpowiedniego wektora w tablicy `cplanes`.

Parametry `degree` i `cpoints` opisują reprezentację wymiernej krzywej Béziera, tj. odpowiednio stopień i punkty kontrolne krzywej jednorodnej. Parametr `output` jest wskaźnikiem procedury, która będzie wywołana w celu wyprowadzenia (np. narysowania) poszczególnych łuków krzywej położonych wewnątrz wielokąta.

## 7.17 Badanie kształtu łamanych

```
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 );
```

Procedury `mbs_MonotonicPolylinef` i `mbs_MonotonicPolylineRf` dokonują sprawdzenia monotoniczności łamanych ze względu na wektor  $v$ .

Łamane leżą w przestrzeni  $\mathbb{R}^d$ , o wymiarze określonym przez parametr `spdimen`. Dla procedury `mbs_MonotonicPolylinef` musi on mieć wartość  $d$ , a dla procedury `mbs_MonotonicPolylineRf`  $d + 1$ .

Parametr `npoints` określa liczbę punktów. Współrzędne kartezjańskie (dla `mbs_MonotonicPolylinef`) albo jednorodne (dla `mbs_MonotonicPolylineRf`) tych punktów są podane w tablicy `points`. Parametr `pitch` określa odległości w tablicy początków reprezentacji kolejnych punktów (która może być inna niż `spdimen`).

Parametr  $v$  jest wskaźnikiem tablicy zawierającej  $d$  liczb będących współrzędnymi wektora  $v$ .

Wartością każdej z tych procedur jest `true`, jeśli rzuty kolejnych punktów na prostą o kierunku wektora  $v$  są uporządkowane wzdłuż prostej (i współrzędne wagowe wszystkich punktów w przypadku procedury `mbs_MonotonicPolylineRf` mają ten sam znak) oraz `false` w przeciwnym razie.

Procedury mogą być użyte do sprawdzania, czy łamane kontrolne krzywych są monotoniczne ze względu na wektor  $v$ , co jest warunkiem dostatecznym monotoniczności krzywych Béziera i B-sklejanych (przy założeniu, dla krzywych wymiernych, że wszystkie wagi mają ten sam znak).



## 7.18 Rasteryzacja krzywych

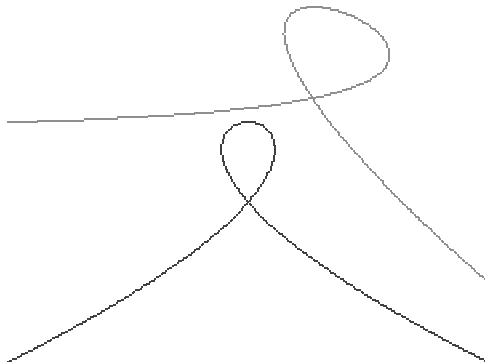
Procedury rasteryzacji krzywych korzystają z reprezentacji pikseli w postaci struktur `xpoint` oraz z bufora i makr jego obsługi zdefiniowanych w pliku `pkvaria.h`. W szczególności krzywe stopnia 1 są rasteryzowane jako odcinki, za pomocą procedury `_pkv_DrawLine` z biblioteki `libpkvaria`.

```
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 );
```

Procedury `mbs_RasterizeBC2f` i `mbs_RasterizeBD2Rf` dokonują rasteryzacji krzywych Béziera, tj. wyznaczają piksele tworzące ośmiospójne obrazy krzywych.

Parametry procedur: `degree` — stopień krzywej, `cpoints` — punkty kontrolne (dla krzywej wymiernej są to punkty kontrolne krzywej jednorodnej), `output` — procedura wyjściowa (wyprowadzająca piksele, tj. na przykład rysująca je). Parametr `outlast` określa, czy ma być wyprowadzony ostatni piksel obrazu krzywej. Rysując krzywą sklejaną (z wielu łuków) albo zamkniętą nie należy wyprowadzać ostatniego piksela każdego łuku.

Liczba wywołań procedury `output` w trakcie działania procedur rasteryzacji krzywej zależy od liczby pikseli do narysowania i od pojemności wewnętrznego buforu na piksele. Parametr `n` procedury określa liczbę pikseli do wyświetlenia.



Rys. 7.18. Obrazy rastrowe krzywych Béziera (wielomianowej i wymiernej) stopnia 3

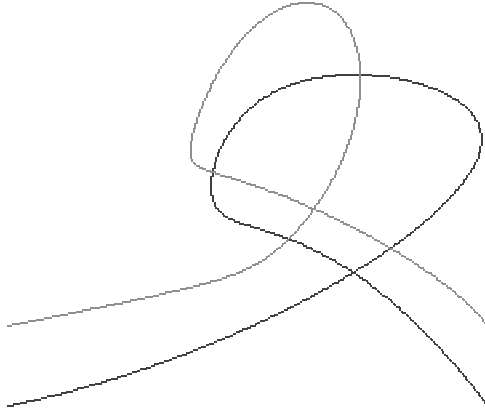
```

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 );

```

Procedury `mbs_RasterizeBS2f` i `mbs_RasterizeBS2Rf` dokonują rasteryzacji płaskich krzywych B-sklejanych. Parametry `degree` (stopień), `lastknot` (numer ostatniego węzła), `knots` (tablica węzłów) i `cpoints` (tablica punktów kontrolnych) opisują krzywą. Parametr `output` jest wskaźnikiem procedury, która będzie wywoływana w celu wyprowadzenia (np. wyświetlenia) pikseli. Parametr `outlast` określa, czy ostatni piksel ma być wyprowadzany, czy nie.

**Do zrobienia:** Obcinanie krzywych przed wyświetlaniem. Sprawdzanie, czy krzywa nie jest tak krótka, że jej obraz jest jednym pikselem. „Wygładzanie” obliczonego ciągu pikseli.



Rys. 7.19. Obrazy rastrowe krzywych B-sklejanych (wielomianowej i wymiernej) stopnia 3

## 7.19 Przetwarzanie płatów Coonsa

### 7.19.1 Płaty wielomianowe

Wielomianowe płaty Coonsa są reprezentowane za pomocą krzywych Béziera, których stopnie mogą być różne. Dziedziną płata jest kwadrat  $[0, 1]^2$ , zatem liczby  $a, b, c, d$  opisane w p. 7.1.6 są równe odpowiednio 0, 1, 0, 1.

```
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 );
```

Procedura `mbs_BezC1CoonsFindCornersf` wyznacza macierz  $P$  o wymiarach  $4 \times 4$ , której elementami są odpowiednie punkty krzywych  $c_{00}, c_{10}, c_{01}, c_{11}$  i wektory ich pochodnych.

Parametry: `spdimen` — wymiar  $d$  przestrzeni, w której leżą krzywe i reprezentowany przez nie wielomianowy bikubiczny (klasy  $C^1$ ) płat Coonsa. Każda para parametrów `degc??` i `c??` opisuje jedną z krzywych, stopień i tablicę punktów kontrolnych.

Parametr `pcorners` jest wskaźnikiem tablicy, w której ma być umieszczony wynik; tablica ta musi mieć długość 16d.

```
boolean mbs_BezC1CoonsToBez ( 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 );
```

Procedura `mbs_BezC1CoonsToBez` wyznacza reprezentację Béziera bikubicznego płata Coonsa (klasy  $C^1$ ) określonego przez dane krzywe wielomianowe. Wartością procedury jest `true`, jeśli obliczenie zakończyło się sukcesem, a `false` w przeciwnym razie (przyczyną niepowodzenia może być brak pamięci na stosie pamięci pomocniczej).

Wartość parametru `spdimen` jest wymiarem  $d$  przestrzeni, w której leżą krzywe i określony przez nie płat. Każda para parametrów `degc??` i `c??` opisuje odpowiednią krzywą z rodziny  $c_{00}, c_{01}, c_{10}, c_{11}$ , przez podanie stopnia i tablicy punktów kontrolnych. Kolejne pary parametrów `degd??` i `d??` opisują w ten sam sposób

krzywe z rodziny  $\mathbf{d}_{00}, \mathbf{d}_{01}, \mathbf{d}_{10}, \mathbf{d}_{11}$ . Krzywe te muszą spełniać (z dokładnością do błędów zaokrągleń) warunki zgodności (7.15), co *nie jest* sprawdzane.

Zmienne  $*n$  i  $*m$  otrzymują wartości opisujące stopień reprezentacji Béziera płata. Wartością  $n$  zmiennej  $*n$  jest największa z wartości parametrów  $\text{degc}??$  lub 3 (jeśli liczba 3 jest większa). Podobnie, wartość  $m$  zmiennej  $*m$  jest największą z wartości parametrów  $\text{degd}??$  lub 3. W tablicy wskazywanej przez parametr  $p$  procedura umieszcza punkty kontrolne Béziera płata; tablica ta musi być dostatecznie pojemna (musi mieć długość co najmniej  $(n+1)(m+1)d$ ).

```
void mbs_TabCubicHFuncDer2f ( float a, float b,
                             int nkn, const float *kn,
                             float *hfunc, float *dhfunc, float *ddhfunc );
```

Procedura `mbs_TabCubicHFuncDer2f` tablicuje wielomiany  $\tilde{H}_{00}, \tilde{H}_{10}, \tilde{H}_{01}$  i  $\tilde{H}_{11}$ , będące podstawą definicji bikubicznego płata Coonsa, oraz ich pochodne rzędu 1 i 2. Wyniki tego obliczenia mogą być użyte do stabilizowania bikubicznego płata Coonsa na siatce prostokątnej, za pomocą procedury `mbs_TabBezC1CoonsDer2f` (płat wielomianowy) lub `mbs_TabBSC1CoonsDer2f` (płat sklepany).

Parametry  $a$  i  $b$  opisują końce odcinka przyjętego za dziedzinę krzywych  $c_{ij}$  lub  $\mathbf{d}_{ij}$ ; dla wielomianowych płatów Coonsa ich wartościami powinny być liczby 0 i 1.

Parametr  $nkn$  określa liczbę  $k$  punktów  $u_m \in [a, b]$ , w których należy obliczyć wartości wielomianów; punkty te (tj. liczby zmiennopozycyjne) są podane w tablicy  $kn$ .

Wartości wielomianów oraz ich pochodnych rzędu 1 i 2 są wpisywane odpowiednio do tablic  $hfunc$ ,  $dhfunc$  i  $ddhfunc$ . Tablice te muszą mieć długość co najmniej  $4k$ ; na każdych kolejnych czterech pozycjach w tablicy są wpisywane wartości czterech wielomianów lub ich pochodnych w kolejnym punkcie  $u_m$ .

```
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 );

```

Procedura `mbs_TabBezC1CoonsDer2f` służy do szybkiego stablicowania bikubicznego wielomianowego płata Coonsa, razem z pochodnymi cząstkowymi rzędu 1 i 2, dla punktów  $(u_i, v_j)$ , gdzie  $i \in \{0, \dots, k-1\}$ ,  $j \in \{0, \dots, l-1\}$ .

Parametr `spdimen` określa wymiar  $d$  przestrzeni, w której jest płat. Parametr `nknu` określa liczbę  $k$ , tablica `knu` zawiera liczby  $t_0, \dots, t_{k-1}$ . Tablice `hfuncu`, `dhfuncu`, `ddhfuncu` zawierają odpowiednio wartości wielomianów  $H_{00}, H_{10}, H_{01}, H_{11}$  i ich pochodnych rzędu 1 i 2 w punktach  $u_0, \dots, u_{k-1}$ ; wartości te najprościej jest obliczyć wywołując zawczasu procedurę `mbs_TabCubicHFuncDer2f` (z parametrami  $a = 0$ ,  $b = 1$ ).

Ciąg liczb  $v_j$  określających drugie współrzędne punktów tablicowania płata jest w analogiczny sposób opisany za pomocą parametrów `nknv` i `knv`, tablice `hfuncv`, `dhfuncv` i `ddhfuncv` zawierają wartości funkcji  $H_{ij}$  i ich pochodnych dla tych liczb.

Pary parametrów `degc??` i `c??` oraz `degd??` i `d??` opisują krzywe Béziera określające płat. Krzywe te muszą spełniać (z dokładnością do błędów zaokrągleń) warunki zgodności (7.15).

Do tablic wskazywanych przez parametry `p`, `pu`, `pv`, `puu`, `puv`, `pvv` procedura wpisuje obliczone punkty płata i wektory pochodnych cząstkowych rzędu 1 i 2; jeśli któryś z tych parametrów ma wartość `NULL`, to odpowiednia pochodna nie jest tablicowana. W przeciwnym razie wskazywana tablica musi mieć długość co najmniej  $k^2d$ .

Wartością procedury jest `true` w razie sukcesu i `false` w razie niepowodzenia obliczeń (z powodu braku miejsca na stosie pamięci pomocniczej).

```

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 );

```

Procedura `mbs_TabBezC1Coons0Der2f` jest nieco uproszczoną wersją procedury `mbs_TabBezC1CoonsDer2f` dla przypadku, gdy krzywe  $c_{10}$ ,  $c_{11}$ ,  $d_{10}$  i  $d_{11}$  są zerowe (tj. gdy wszystkie ich punkty kontrolne mają zerowe wszystkie współrzędne). Stablicowanie płata określonego przez takie krzywe może być wykonane w krótszym czasie; z procedury tej korzysta biblioteka `libg1hole`.

Parametry procedury `mbs_TabBezC1Coons0Der2f` są takie same, jak parametry procedury `mbs_TabBezC1CoonsDer2f` o tych samych nazwach.

```

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 );

```

Procedura `mbs_BezC2CoonsFindCornersf` wyznacza macierz **P** o wymiarach  $6 \times 6$ , której elementami są odpowiednie punkty krzywych  $c_{00}$ ,  $c_{10}$ ,  $c_{01}$ ,  $c_{11}$ ,  $c_{02}$ ,  $c_{12}$  i wektory ich pochodnych.

Parametry: `spdimen` — wymiar  $d$  przestrzeni, w której leżą krzywe i reprezentowany przez nie wielomianowy dwupiętny (klasy  $C^2$ ) płat Coonsa. Każda para parametrów `degc??` i `c??` opisuje jedną z krzywych, stopień i tablicę punktów kontrolnych.

Parametr `pcorners` jest wskaźnikiem tablicy, w której ma być umieszczony wynik; tablica ta musi mieć długość  $36d$ .

```

boolean mbs_BezC2CoonsToBez ( 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 );

```

Procedura `mbs_BezC2CoonsToBez` dokonuje konwersji dwupiętnego płata Coonsa do postaci Béziera. Płat jest dany za pomocą 12 krzywych wielomianowych, opisujących brzeg (krzywe  $c_{00}$ ,  $c_{10}$ ,  $d_{00}$ ,  $d_{10}$ ) i pochodne poprzeczne rzędu 1 (krzywe  $c_{01}$ ,  $c_{11}$ ,  $d_{01}$ ,  $d_{11}$ ) i 2 (krzywe  $c_{02}$ ,  $c_{12}$ ,  $d_{02}$ ,  $d_{12}$ ). Wszystkie te krzywe są dane w postaci Béziera, ich stopnie są określone za pomocą parametrów  $\text{degc00}, \dots, \text{degd12}$ , punkty kontrolne (w przestrzeni o wymiarze  $\text{spdimen}$ ) są dane w tablicach  $c_{00}, \dots, d_{12}$ .

Parametry wyjściowe to  $*n$  i  $*m$ , które otrzymują wartości określające stopień, oraz tablica  $p$ , do której procedura wpisuje współrzędne punktów kontrolnych Béziera płata.

```

void mbs_TabQuintichFuncDer3f ( float a, float b,
                                int nkn, const float *kn,
                                float *hfunc, float *dhfunc,
                                float *ddhfunc, float *dddfunc );

```

Procedura `mbs_TabQuintichFuncDer3f` tablicuje wielomiany  $\tilde{H}_{00}, \tilde{H}_{10}, \tilde{H}_{01}, \tilde{H}_{11}, \tilde{H}_{02}$  i  $\tilde{H}_{12}$ , będące podstawą definicji dwupiętnego płata Coonsa, oraz ich pochodne rzędu 1, 2 i 3. Wyniki tego obliczenia mogą być użyte do stabilizowania dwupiętnego płata Coonsa na siatce prostokątnej, za pomocą procedury `mbs_TabBezC2CoonsDer3f` (płat wielomianowy) lub `mbs_TabBSC2CoonsDer3f` (płat sklepany).

Parametry  $a$  i  $b$  opisują końce odcinka przyjętego za dziedzinę krzywych  $c_{ij}$  lub  $d_{ij}$ ; dla wielomianowych płatów Coonsa ich wartościami powinny być liczby 0 i 1.

Parametr  $nkn$  określa liczbę  $k$  punktów  $u_m \in [a, b]$ , w których należy obliczyć wartości wielomianów; punkty te (tj. liczby zmiennopozycyjne) są podane w tablicy  $kn$ .

Wartości wielomianów oraz ich pochodnych rzędu 1, 2 i 3 są wpisywane odpowiednio do tablic  $hfunc$ ,  $dhfunc$ ,  $ddhfunc$  i  $dddfunc$ . Tablice te muszą mieć



długość co najmniej 6k; na każdych kolejnych sześciu pozycjach w tablicy są wpisywane wartości sześciu wielomianów lub ich pochodnych w kolejnym punkcie  $u_m$ .

```
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 );
```

Procedura `mbs_TabBezC2CoonsDer3f` służy do szybkiego stabilizowania dwupiętowego wielomianowego płata Coonsa, razem z pochodnymi cząstkowymi rzędu 1, 2 i 3, dla punktów  $(u_i, v_j)$ , gdzie  $i \in \{0, \dots, k-1\}$ ,  $j \in \{0, \dots, l-1\}$ .

Parametr `spdimen` określa wymiar  $d$  przestrzeni, w której jest płat. Parametr `nknu` określa liczbę  $k$ , tablica `knu` zawiera liczby  $u_0, \dots, u_{k-1}$ . Tablice `hfuncu`, `dhfuncu`, `ddhfuncu` i `dddhfuncu` zawierają odpowiednio wartości wielomianów  $H_{00}, H_{10}, H_{01}, H_{11}, H_{02}, H_{12}$  i ich pochodnych rzędu 1, 2 i 3 w punktach  $t_0, \dots, t_{k-1}$ ; wartości te najprościej jest obliczyć wywołując zawczasu procedurę `mbs_TabQuinticHFuncDer3f` (z parametrami  $a = 0$ ,  $b = 1$ ).

Parametry `nknv`, `knv`, `hfuncv`, `dhfuncv`, `ddhfuncv` i `dddhfuncv` w analogiczny sposób reprezentują ciąg  $v_0, \dots, v_{l-1}$  i wartości i pochodne funkcji  $H_{ij}$  w punktach z tego ciągu.

Pary parametrów `degc??` i `c??` oraz `degd??` i `d??` opisują krzywe Béziera określające płat. Krzywe te muszą spełniać (z dokładnością do błędów zaokrągleń) warunki zgodności (7.15).

Do tablic wskazywanych przez parametry `p`, `pu`, `pv`, `puu`, `puv`, `pvv`, `puuu`, `puuv`, `puvv` i `pvvv` procedura wpisuje obliczone punkty płata i wektory pochodnych cząst-

kowych rzędu 1, 2 i 3; jeśli któryś z tych parametrów ma wartość NULL, to odpowiednia pochodna nie jest tablicowana. W przeciwnym razie wskazywana tablica musi mieć długość co najmniej  $k^2d$ .

Wartością procedury jest true w razie sukcesu i false w razie niepowodzenia obliczeń (z powodu braku miejsca na stosie pamięci pomocniczej).

```
boolean mbs_TabBezC2Coons0Der3f ( int spdimen,
    int nknu, const float *knu, const float *hfuncu,
    const float *dhfuncu, const float *ddhfuncu,
    const float *dddhfuncu,
    int knv, 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 );
```

Procedura mbs\_TabBezC2Coons0Der3f jest nieco uproszczoną wersją procedury mbs\_TabBezC2CoonsDer3f dla przypadku, gdy krzywe  $c_{10}$ ,  $c_{11}$ ,  $c_{12}$ ,  $d_{10}$ ,  $d_{11}$ , i  $d_{12}$  są zerowe (tj. gdy wszystkie ich punkty kontrolne mają zerowe wszystkie współrzędne). Stablicowanie płata określonego przez takie krzywe może być wykonane w krótszym czasie; z procedury tej korzysta biblioteka libg1hole.

Parametry procedury mbs\_TabBezC2Coons0Der3f są takie same, jak parametry procedury mbs\_TabBezC2CoonsDer3f o tych samych nazwach.

### 7.19.2 Płaty sklejjane

Sklejane płaty Coonsa są określone za pomocą krzywych B-sklejanych; zarówno stopnie poszczególnych krzywych, jak i ciągi węzłów użyte do ich reprezentowania, mogą być różne; krzywe  $c_{ij}$  muszą mieć tylko wspólną dziedzinę (wyznaczoną przez węzły brzegowe w ich ciągach węzłów) i to samo dotyczy krzywych  $d_{ij}$ .

```
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 );
```

Procedura `mbs_BSC1CoonsFindCornersf` wyznacza macierz  $\mathbf{P}$  o wymiarach  $4 \times 4$ , której elementami są odpowiednie punkty krzywych  $c_{00}$ ,  $c_{10}$ ,  $c_{01}$ ,  $c_{11}$  i wektory ich pochodnych.

Parametry: `spdimen` — wymiar  $d$  przestrzeni, w której leżą krzywe i reprezentowany przez nie sklejjany bikubiczny (klasy  $C^1$ ) płat Coonsa. Każda czwórka parametrów `degc??`, `lastknotc??`, `knotsc??` i `c??` opisuje jedną z krzywych, stopień, numer ostatniego węzła, tablicę węzłów i tablicę punktów kontrolnych.

Parametr `pcorners` jest wskaźnikiem tablicy, w której ma być umieszczony wynik; tablica ta musi mieć długość  $16d$ .

```

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 );

```

Procedura `mbs_BSC1CoonsToBSf` wyznacza reprezentację B-sklejaną bikubicznego płata Coonsa (klasy  $C^1$ ) określonego przez dane krzywe sklejane. Wartością procedury jest `true`, jeśli obliczenie zakończyło się sukcesem, a `false` w przeciwnym razie (przyczyną niepowodzenia może być brak pamięci na stosie pamięci pomocniczej lub niepoprawne ciągi węzłów w reprezentacjach krzywych).

Wartość parametru `spdimen` jest wymiarem  $d$  przestrzeni, w której leżą krzywe i określony przez nie płat. Każda czwórka parametrów `degc??`, `lastknotc??`, `knotsc??` i `c??` opisuje odpowiednią krzywą z rodziny  $c_{00}, c_{01}, c_{10}, c_{11}$ , przez podanie stopnia numeru ostatniego węzła, ciągu węzłów i tablicy punktów kontrolnych. Kolejne pary parametrów `degd??`, `lastknotd??`, `knotsd??` i `d??` opisują w ten sam sposób krzywe z rodziny  $d_{00}, d_{01}, d_{10}, d_{11}$ . Krzywe te muszą spełniać (z dokładnością do błędów zaokrągleń) warunki zgodności (7.15), co *nie jest* sprawdzane.

Zmienne `*n` i `*m` otrzymują wartości opisujące stopień reprezentacji B-sklejanej płata. Wartość `n` zmiennej `*n` jest największą z wartości parametrów `degc??` lub 3 (jeśli liczba 3 jest większa). Podobnie, wartość `m` zmiennej `*m` jest największą z wartości parametrów `degd??` lub 3. Parametry `lastuknot`, `uknots`, `lastvknot` i `vknots` służą do wyprowadzenia ciągów węzłów skonstruowanej reprezentacji B-sklejanej płata Coonsa. W tablicy wskazywanej przez parametr `p` procedura umieszcza punkty kontrolne.

Tablice `unknots`, `vknots` i `p` muszą być dostatecznie pojemne; aby zarezerwować te tablice, można posłużyć się procedurą `mbs_FindBSCommonKnotSequencef` dla rodzin krzywych  $c_{ij}$  oraz  $d_{ij}$ ; zmienne wskazywane przez parametr `lastknot` tej

procedury powinny mieć wartości początkowe 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 );
```

Procedura `mbs_TabBSC1CoonsDer2f` służy do szybkiego stabilizowania bikubicznego sklejonego płata Coonsa, razem z pochodnymi cząstkowymi rzędu 1 i 2, dla punktów  $(u_i, v_j)$ , gdzie  $i \in \{0, \dots, k_u - 1\}$ ,  $j \in \{0, \dots, k_v - 1\}$ .

Parametr `spdimen` określa wymiar  $d$  przestrzeni, w której jest płat. Parametry `nknu` i `nknv` określają liczby  $k_u$  i  $k_v$ , tablice `knu` i `knv` zawierają odpowiednio liczby  $u_0, \dots, u_{k_u-1}$  i  $v_0, \dots, v_{k_v-1}$ . Tablice `hfuncu`, `dhfuncu`, `ddhfuncu` zawierają odpowiednio wartości wielomianów  $\tilde{H}_{00}, \tilde{H}_{10}, \tilde{H}_{01}, \tilde{H}_{11}$  i ich pochodnych rzędu 1 i 2 w punktach  $u_0, \dots, u_{k_u-1}$ ; wartości te najprościej jest obliczyć wywołując zawczasu procedurę `mbs_TabCubicHFuncDer2f` z parametrami  $a, b$  o wartościach będących końcami przedziału zmienności parametru  $u$  płata. Podobnie, tablice `hfuncv`, `dhfuncv`, `ddhfuncv` zawierają wartości wielomianów  $\hat{H}_{00}, \hat{H}_{10}, \hat{H}_{01}, \hat{H}_{11}$  i ich pochodnych rzędu 1 i 2 w punktach  $v_0, \dots, v_{k_v-1}$ .

Czwórki parametrów `degc??`, `lastknotc??`, `knotsc??` i `c??` oraz `degd??`, `lastknotd??`, `knotsd??` i `d??` opisują krzywe B-sklejane określające płat. Krzywe te muszą spełniać (z dokładnością do błędów zaokrągleń) warunki zgodności (7.15).

Do tablic wskazywanych przez parametry `p`, `pu`, `pv`, `puu`, `puv`, `pvv` procedura wpisuje obliczone punkty płata i wektory pochodnych cząstkowych rzędu 1 i 2;

jeśli któryś z tych parametrów ma wartość NULL, to odpowiednia pochodna nie jest tablicowana. W przeciwnym razie wskazywana tablica musi mieć długość co najmniej  $k_u k_v d$ .

Wartością procedury jest true w razie sukcesu i false w razie niepowodzenia obliczeń (z powodu braku miejsca na stosie pamięci pomocniczej).

```
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 );
```

Procedura mbs\_TabBSC1Coons0Der2f jest nieco uproszczoną wersją procedury mbs\_TabBSC1CoonsDer2f dla przypadku, gdy krzywe  $c_{10}$ ,  $c_{11}$ ,  $d_{10}$  i  $d_{11}$  są zerowe (tj. gdy wszystkie ich punkty kontrolne mają zerowe wszystkie współrzędne). Stablicowanie płata określonego przez takie krzywe może być wykonane w krótszym czasie.

Parametry procedury mbs\_TabBSC1Coons0Der2f są takie same, jak parametry procedury mbs\_TabBSC1CoonsDer2f o tych samych nazwach.

```
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 );
```

Procedura `mbs_BSC2CoonsFindCornersf` wyznacza macierz **P** o wymiarach  $6 \times 6$ , której elementami są odpowiednie punkty krzywych  $c_{00}, c_{10}, c_{01}, c_{11}, c_{02}, c_{12}$  i wektory ich pochodnych.

Parametry: `spdimen` — wymiar  $d$  przestrzeni, w której leżą krzywe i reprezentowany przez nie sklejany dwupiętny (klasy  $C^2$ ) płat Coonsa. Każda czwórka parametrów `degc??`, `lastknotc??`, `knotsc??` i `c??` opisuje jedną z krzywych, stopień, numer ostatniego węzła, tablicę węzłów i tablicę punktów kontrolnych.

Parametr `pcorners` jest wskaźnikiem tablicy, w której ma być umieszczony wynik; tablica ta musi mieć długość 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 );
```

Procedura `mbs_BSC2CoonsToBSf` wyznacza reprezentację B-sklejaną dwupiętnego płata Coonsa (klasy  $C^2$ ) określonego przez dane krzywe sklejane. Wartością procedury jest `true`, jeśli obliczenie zakończyło się sukcesem, a `false` w przeciwnym razie (przyczyną niepowodzenia może być brak pamięci na stosie pamięci pomocniczej lub niepoprawne ciągi węzłów w reprezentacjach krzywych).

Wartość parametru `spdimen` jest wymiarem  $d$  przestrzeni, w której leżą krzywe i określony przez nie płat. Każda czwórka parametrów `degc??`, `lastknotc??`, `knotsc??` i `c??` opisuje odpowiednią krzywą z rodziny  $c_{00}, c_{01}, c_{02}, c_{10}, c_{11}, c_{12}$ , przez podanie stopnia numeru ostatniego węzła, ciągu węzłów i tablicy punktów kontrolnych. Kolejne pary parametrów `degd??`, `lastknotd??`, `knotsd??` i `d??` opisują w ten sam sposób krzywe z rodziny  $d_{00}, d_{01}, d_{02}, d_{10}, d_{11}, d_{12}$ . Krzywe te muszą spełniać (z dokładnością do błędów zaokrągleń) warunki zgodności (7.15), co *nie jest* sprawdzane.

Zmienne `*n` i `*m` otrzymują wartości opisujące stopień reprezentacji B-sklejanej płata. Wartość `n` zmiennej `*n` jest największą z wartości parametrów `degc??` lub 5 (jeśli liczba 5 jest większa). Podobnie, wartość `m` zmiennej `*m` jest największą z wartości parametrów `degd??` lub 5. Parametry `lastuknot`, `uknots`, `lastvknot` i `vknots` służą do wyprowadzenia ciągów węzłów skonstruowanej reprezentacji B-sklejanej płata Coonsa. W tablicy wskazywanej przez parametr `p` procedura umieszcza punkty kontrolne.

Tablice `unkots`, `vknots` i `p` muszą być dostatecznie pojemne; aby zarezerwować te tablice, można posłużyć się procedurą `mbs_FindBSCommonKnotSequencef` dla rodzin krzywych  $c_{ij}$  oraz  $d_{ij}$ ; zmienne wskazywane przez parametr `lastknot` tej procedury powinny mieć wartości początkowe 5.



```

boolean mbs_TabBSC2CoonsDer3f ( int spdimen,
    int nknu, const float *knu, const float *hfuncu,
    const float *dhfuncu, const float *ddhfuncu,
    const float *dddhfuncu,
    int knv, 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 );

```

Procedura `mbs_TabBSC2CoonsDer3f` służy do szybkiego stabcowania dwupiętowego sklejanego płata Coonsa, razem z pochodnymi cząstkowymi rzędu 1, 2 i 3, dla punktów  $(u_i, v_j)$ , gdzie  $i \in \{0, \dots, k_u - 1\}$ ,  $j \in \{0, \dots, k_v - 1\}$ .

Parametr `spdimen` określa wymiar  $d$  przestrzeni, w której jest płat. Parametry `nknu` i `knv` określają liczby  $k_u$  i  $k_v$ , tablice `knu` i `knv` zawierają odpowiednio liczby  $u_0, \dots, u_{k_u-1}$  i  $v_0, \dots, v_{k_v-1}$ . Tablice `hfuncu`, `dhfuncu`, `ddhfuncu`, `dddhfuncu` zawierają odpowiednio wartości wielomianów  $\tilde{H}_{00}, \tilde{H}_{10}, \tilde{H}_{01}, \tilde{H}_{11}, \tilde{H}_{02}, \tilde{H}_{12}$  i ich pochodnych rzędu 1, 2 i 3 w punktach  $u_0, \dots, u_{k_u-1}$ ; wartości te najprościej jest ob-

liczyć wywołując zawczasu procedurę `mbs_TabQuinticHFuncDer3f` z parametrami `a`, `b` o wartościach będących końcami przedziału zmienności parametru `u` płata. Podobnie, tablice `hfuncv`, `dhfuncv`, `ddhfuncv`, `dddhfuncv` zawierają wartości wielomianów  $\hat{H}_{00}, \hat{H}_{10}, \hat{H}_{01}, \hat{H}_{11}, \hat{H}_{02}, \hat{H}_{12}$  i ich pochodnych rzędu 1, 2 i 3 w punktach  $v_0, \dots, v_{k_v-1}$

Czwórki parametrów `degc??`, `lastknotc??`, `knotsc??` i `c??` oraz `degd??`, `lastknotd??`, `knotsd??` i `d??` opisują krzywe B-sklejane określające płat. Krzywe te muszą spełniać (z dokładnością do błędów zaokrągleń) warunki zgodności (7.15).

Do tablic wskazywanych przez parametry `p`, `pu`, `pv`, `puu`, `puv`, `pvv`, `puuu`, `puuv`, `puvv`, `pvuv` procedura wpisuje obliczone punkty płata i wektory pochodnych cząstkowych rzędu 1, 2 i 3; jeśli któryś z tych parametrów ma wartość `NULL`, to odpowiednia pochodna nie jest tablicowana. W przeciwnym razie wskazywana tablica musi mieć długość co najmniej  $k_u k_v d$ .

Wartością procedury jest `true` w razie sukcesu i `false` w razie niepowodzenia obliczeń (z powodu braku miejsca na stosie pamięci pomocniczej).

```
boolean mbs_TabBSC2Coons0Der3f ( int spdimen,
    int nknu, const float *knu, const float *hfuncu,
    const float *dhfuncu, const float *ddhfuncu,
    const float *dddhfuncu,
    int knv, 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 *pvuv );
```

Procedura `mbs_TabBSC2Coons0Der3f` jest nieco uproszczoną wersją procedury `mbs_TabBSC2CoonsDer3f` dla przypadku, gdy krzywe  $c_{10}, c_{11}, c_{12}, d_{10}, d_{11}$  i  $d_{12}$  są zerowe (tj. gdy wszystkie ich punkty kontrolne mają zerowe wszystkie współrzędne). Stablicowanie płata określonego przez takie krzywe może być wykonane w krótszym

czasie.

Parametry procedury `mbs_TabBSC2Coons0Der3f` są takie same, jak parametry procedury `mbs_TabBSC2CoonsDer3f` o tych samych nazwach.

## 7.20 Produkt sferyczny

Produkt sferyczny płaskich krzywych parametrycznych,  $\mathbf{p}(t) = [x_{\mathbf{p}}(t), y_{\mathbf{p}}(t)]^T$  i  $\mathbf{q}(t) = [x_{\mathbf{q}}(t), y_{\mathbf{q}}(t)]^T$ , jest powierzchnią parametryczną w  $\mathbb{R}^3$ , daną wzorem

$$\mathbf{s}(u, v) = \begin{bmatrix} x_{\mathbf{p}}(u)x_{\mathbf{q}}(v) \\ y_{\mathbf{p}}(u)x_{\mathbf{q}}(v) \\ y_{\mathbf{q}}(v) \end{bmatrix}.$$

Krzywe  $\mathbf{p}$  i  $\mathbf{q}$  są zwane odpowiednio równikiem (*equator*) i południkiem (*meridian*). Procedury opisane niżej obliczają punkty kontrolne reprezentacji B-sklejanej produktu sferycznego płaskich krzywych B-sklejanych, odpowiednio kawałkami wielomianowej i wymiernej.

Ciąg węzłów równika jest ciągiem „u” produktu sferycznego, a ciąg węzłów południka jest ciągiem „v”.

```
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 Rysowanie płatów obciętych

### 7.21.1 Reprezentacja dziedziny

Dziedzina obciętego płata B-sklejanego stopnia  $(n, m)$ , o węzłach  $u_0, \dots, u_N$  oraz  $v_0, \dots, v_M$  jest podzbiorem prostokąta  $[u_n, u_{N-n}] \times [v_m, v_{M-m}]$ . W szczególności jest to zawsze zbiór ograniczony. Brzeg dziedziny powierzchni obciętej jest sumą płaskich krzywoliniowych łamanych zamkniętych. Każda taka łamana składa się z

- łamanych (ciągów odcinków),
- krzywych Béziera,
- krzywych B-sklejanych,

zwanych dalej elementami brzegu, przy czym punkty (wierzchołki łamanej i punkty kontrolne) mogą być dane za pomocą współrzędnych kartezjańskich (wtedy są typu `point2f`) albo jednorodnych (wtedy są typu `vector3f`).

Dane opisujące każdą taką łamaną muszą spełniać następujący warunek: krzywe B-sklejane muszą być ciągłe, a ponadto punkt końcowy każdego elementu (łamanej lub krzywej) jest punktem początkowym elementu następnego (przy czym za element następnym elementu ostatniego uważa się element pierwszy). Jeśli ten warunek nie jest spełniony, to procedury rysowania płatów obciętych wstawią odpowiednie odcinki.

Drugi warunek to brak punktów niewłaściwych. Wystarczy, aby wszystkie współrzędne wagowe były dodatnie, choć nie jest to konieczne. Natomiast niespełnienie tego warunku, czyli podanie brzegu, który jest nieograniczony, może spowodować błąd wykonania programu. Co więcej, wszystkie łamane i krzywe muszą leżeć w prostokącie, który jest dziedziną płata nieobciętego.

Brzeg dziedziny będzie reprezentowany za pomocą tablicy zawierającej struktury typu `polycurvef`.

```
typedef struct{
    boolean closing;
    byte    spdimen;
    short   degree;
    int     lastknot;
    float   *knots;
    float   *points;
} polycurvef;
```

Pole `closing` określa, z czym łączy się koniec elementu. Jeśli wartością tego pola jest `false`, to z następnym elementem (tj. z elementem opisanym przez następny element tablicy). Jeśli `true`, to z początkiem elementu pierwszego w tablicy lub ostatniego z elementów poprzedzających dany, którego poprzednik ma pole `closing`

o wartości true (czyli z początkiem ostatniego elementu wyznaczającego początek nowego zamkniętego fragmentu brzegu).

Pole `spdimen` może mieć wartość 2 lub 3. W pierwszym przypadku pole `points` wskazuje tablicę struktur `point2f` — zawierają one współrzędne kartezjańskie punktów na płaszczyźnie. W drugim przypadku tablica wskazywana przez pole `points` zawiera struktury `vector3f`, które zawierają współrzędne jednorodnie punktów (co oznacza, że krzywa, której to są punkty kontrolne, jest wymierna, w reprezentacji jednorodnej).

Pole `degree` określa stopień  $n$  krzywej, który musi być większy lub równy 1.

Pole `lastknot` określa indeks  $N$  ostatniego wierzchołka łamanej albo ostatniego węzła krzywej B-sklejanej.

Pole `knots` jest wskaźnikiem do tablicy węzłów krzywej B-sklejanej, o długości  $N + 1$ .

Pole `points` jest wskaźnikiem do tablicy zawierającej wierzchołki łamanej, tablica ta zawiera pary lub trójki liczb typu `float`, zależnie od wartości pola `spdimen`.

Aby określić łamaną złożoną z  $N$  odcinków, należy polom struktury nadać wartości `degree=1`, `lastknot=N`, `knots=NULL`. W tablicy wskazywanej przez pole `points` ma być `spdimen*(N + 1)` liczb typu `float` ( $N + 1$  struktur `point2f` lub `vector3f`).

Aby określić krzywą Béziera stopnia  $n > 0$ , należy polom struktury nadać wartości `degree=n`, `lastknot=-1`, `knots=NULL`. W tablicy wskazywanej przez pole `points` ma być `spdimen*(n + 1)` liczb typu `float` ( $n + 1$  struktur `point2f` lub `vector3f`).

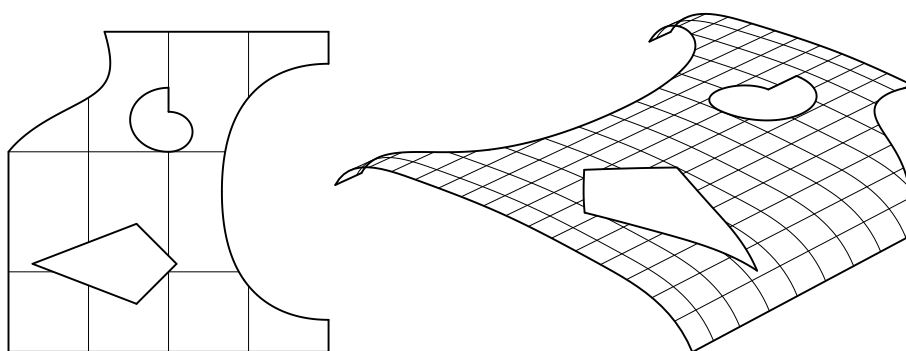
Aby określić krzywą B-sklejaną stopnia  $n > 0$ , należy polom struktury nadać wartości `degree=n`, `lastknot=N`. Pole `knots` ma wskazywać tablicę  $N + 1$  liczb typu `float`, zawierającą ciąg węzłów, a pole `points` tablicę `spdimen*(N - n)` liczb typu `float`, zawierającą współrzędne punktów kontrolnych.

Nie ma wymagania, aby reprezentacja krzywej B-sklejanej wchodzącej w skład brzegu była o końcach zaczepionych, ale krzywa taka musi łączyć się z sąsiednimi elementami brzegu. W szczególności można określić spójny fragment brzegu dziedziny płata jako jedną zamkniętą krzywą B-sklejaną.

Brzeg płata obciętego może (ale nie musi) być zorientowany. Można przyjąć konwencję, że podczas poruszania się wzdłuż wszystkich łamanych i krzywych opisujących brzeg zgodnie z naturalną parametryzacją, mamy wewnątrz dziedziny po lewej stronie (albo po prawej). Procedury rysowania płata powinny być tak zaimplementowane, aby odpowiednia informacja była wyprowadzana. Ponadto brzeg może mieć samoprzecięcia, co nie powinno powodować błędów wykonania programu.

**Przykład.** Brzeg dziedziny płata na rys. 7.20 ma następujący opis:

```
#define n 3
#define NNt1a 10
float ut1a[NNt1a+1] =
```



Rys. 7.20. Obraz dziedziny i B-sklejanego płata obciętego

```

{-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-sklejana */
   {false,2, 1, 2, NULL,(float*)&cpd1a[0]}, /* łamana */
   {false,3, 3, -1, NULL,(float*)&cpt1b[0]}, /* krzywa Béziera */
   {true, 2, 1, 3, NULL,(float*)&cpd1b[0]}, /* łamana */
   {true, 2, 1, 4, NULL,(float*)&cpd1c[0]}, /* łamana */
   {false,3, 3, -1, NULL,(float*)&cpt1c[0]}, /* krzywa Béziera */
   {false,3, 3, -1, NULL,(float*)&cpt1d[0]}, /* krzywa Béziera */
   {true, 2, 1, 1, NULL,(float*)&cpd1e[0]}}; /* odcinek */

```

Brzeg w przykładzie składa się z trzech zamkniętych krzywych. Pierwsza z nich jest „obrysem zewnętrznym” i składa się z czterech elementów: krzywej B-sklejanej, łamanej złożonej z dwóch odcinków, wymiernej krzywej Béziera i łamanej złożonej z trzech odcinków. Druga krzywa to jedna łamana zamknięta złożona z czterech odcinków, a trzecia krzywa składa się z dwóch półokręgów (reprezentowanych jako

wymierne krzywe Béziera trzeciego stopnia) i łamanej składającej się z jednego odcinka.

**Indeksem** dowolnego punktu na płaszczyźnie nazwiemy liczbę okrążeń (w kierunku przeciwnym do zegara) wykonanych wokół tego punktu podczas obchodzenia brzegu zgodnie z jego orientacją. Pierwsze dwie z trzech krzywych zamkniętych opisujących brzeg płata w przykładzie są zorientowane tak, że poruszając się wzdłuż nich ma się wewnątrz dziedziny po lewej stronie. Trzecia krzywa jest zorientowana odwrotnie. Dlatego indeks punktów na zewnątrz „obrysu” (pierwszej krzywej) i wewnątrz wielokąta, którego brzegiem jest druga krzywa jest równy 0, zaś punkty w dwóch półkółach ograniczonych przez trzecią krzywą mają indeks równy 2. Indeks punktów z wnętrza dziedziny jest równy 1.

### 7.21.2 Kompilacja brzegu dziedziny

Rysowanie płata obciętego wymaga wielokrotnego obliczania punktów przecięcia prostych z brzegiem dziedziny płata. Dla oszczędności czasu reprezentacja opisana w poprzednim punkcie jest tłumaczona na kod zawierający opis łamanych i krzywych Béziera, z których składa się brzeg.

```
int mbs_TrimCVBoundSizef ( int nelem, const polycurvef *bound );
```

Wartością procedury `mbs_TrimCVBoundSizef` jest długość (w bajtach) kodu opisującego brzeg dziedziny płata obciętego. Brzeg jest reprezentowany zgodnie z opisem w poprzednim punkcie; poszczególne części brzegu są łamanymi, krzywymi Béziera lub krzywymi B-sklejanymi opisanymi przez kolejne elementy tablicy `bound` o długości `nelem`.

Procedury tej można użyć w celu zarezerwowania odpowiedniej tablicy do przechowania kodu.

```
void *mbs_CompileTrimPatchBoundf ( int nelem,
                                   const polycurvef *bound,
                                   void *buffer );
```

Procedura `mbs_CompileTrimPatchBoundf` dokonuje „kompilacji” opisu brzegu dziedziny obciętego płata B-sklejanego, tj. generuje kod reprezentujący łamane i krzywe Béziera, z których składa się brzeg (krzywe B-sklejane są zastępowane ciągami odpowiednich krzywych Béziera).

Parametr `nelem` określa długość tablicy `bound`, której elementy opisują brzeg dziedziny, zaś parametr `buffer` jest tablicą, w której ma być umieszczony kod. Zakłada się, że tablica ta jest dostatecznie długa (do obliczenia potrzebnej jej długości służy procedura `mbs_TrimCVBoundSizef`). Jeśli wartość parametru `buffer` jest wskaźnikiem pustym (NULL), to procedura rezerwuje w pamięci pomocniczej (za pomocą `pkv_GetScratchMem`) tablicę o długości obliczonej za pomocą procedury `mbs_TrimCVBoundSizef`.



Wartością procedury jest wskaźnik tablicy zawierającej utworzony kod (czyli początkowa wartość parametru `buffer` lub adres tablicy zarezerwowanej przez procedurę), albo `NULL`, jeśli wystąpił błąd (np. brak pamięci).

### 7.21.3 Wykonywanie obrazków kreskowych

Obrazek kreskowy płata lub jego dziedziny składa się z krzywych będących obrazami brzegu oraz linii stałego pierwszego i drugiego parametru. Aby narysować taki obrazek należy wyznaczyć te linie, czyli wyznaczyć części wspólne odpowiednich prostych z dziedziną płata. To zadanie wykonuje opisana niżej procedura `mbs_FindBoundLineIntersections`. Bardziej „wysokopoziomowa” procedura `mbs_DrawTrimBSPatchDomf` generuje zbiór prostych i wyznacza ich części wspólne z dziedziną. Dla każdej takiej części (odcinka) procedura wywołuje podaną jako parametr procedurę wyjściową, która w odpowiedni sposób wyświetla ten odcinek w dziedzinie albo jego obraz (fragment krzywej stałego parametru) na płacie.

```
typedef struct {
    float t;
    char  sign1, sign2;
} signpoint1f;
```

Struktura typu `signpoint1f` służy do opisanego punktu przecięcia prostej z brzegiem płata obciętego. Prosta jest dana w postaci parametrycznej i dzieli płaszczyznę (w której leży dziedzina) na dwie półpłaszczyzny. Pole `t` struktury służy do przechowania wartości parametru prostej odpowiadającego punktowi przecięcia z brzegiem, zaś pola `sign1` i `sign2` opisują sposób, w jaki brzeg przecina się z prostą. Możliwe wartości tych pól to 0,  $-1$  i  $+1$ , które odpowiadają przypadkom, gdy punkt początkowy (`sign1`) albo końcowy (`sign2`) przecinającego się z prostą małego fragmentu brzegu leży na tej prostej albo we wnętrzu jednej z dwóch półpłaszczyzn.

```
void mbs_FindBoundLineIntersections ( const void *bound,
                                     const point2f *p0, float t0,
                                     const point2f* p1, float t1,
                                     signpoint1f *inters,
                                     int *ninters );
```

Procedura `mbs_FindBoundLineIntersections` oblicza punkty przecięcia prostej przechodzącej przez punkty `p0` i `p1` z brzegiem dziedziny płata obciętego, reprezentowanego przez kod w tablicy `bound` (otrzymany za pomocą procedury `mbs_CompileTrimPatchBoundf`). Znalezione punkty przecięcia są wstawiane do tablicy `inters`. Jeśli brzeg ma z prostą wspólny odcinek, to w tablicy `inters` jest on reprezentowany przez dwa elementy, odpowiadające początkowi i końcowi tego odcinka, przy czym w takim przypadku pola `sign1` i `sign2` tych elementów mają wartość 0.

Liczby  $t_0$  i  $t_1$  są parametrami prostej przyporządkowanymi odpowiednio punktom  $p_0$  i  $p_1$ , przy czym zarówno punkty te, jak i odpowiadające im parametry muszą być różne.

Początkowa wartość parametru `*ninters` określa długość (pojemność) tablicy `inters`, czyli maksymalną liczbę punktów przecięcia, jaką program spodziewa się znaleźć. Na wyjściu parametr ten otrzymuje wartość równą liczbie znalezionych przecięć. Jeśli wystąpił błąd (np. w kodzie), albo przepełnienie tablicy `inters`, to parametr `inters` na wyjściu z procedury ma wartość ujemną.

Tablica `inters` po znalezieniu wszystkich przecięć jest sortowana w kolejności rosnących wartości pól  $t$ .

```
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*) );
```

Procedura `mbs_DrawTrimBSPatchDomf` może być użyta do utworzenia obrazu kreskowego dziedziny obciętego płata B-sklejanego, albo samego płata. Celem procedury jest wyznaczenie odcinków leżących w dziedzinie takiego płata i wywołanie dla każdego takiego odcinka procedury wyjściowej, która wykonuje rysowanie. Całość wiedzy o sposobie dalszego przetwarzania, w tym rysowania odcinka (np. na ekranie lub w pliku postscriptowym) jest odizolowana od procedury `mbs_DrawTrimBSPatchDomf`.

Pierwsze 8 parametrów procedury składa się na opis brzegu dziedziny obciętego płata B-sklejanego. Są to kolejno: stopień  $n$  płata ze względu na parametr  $u$  (`degu`), indeks  $N$  ostatniego węzła w ciągu  $u_0, \dots, u_N$  (`lastuknot`), tablica z tymi węzłami (`uknots`), stopień  $m$  płata ze względu na parametr  $v$  (`degv`), indeks  $M$  ostatniego elementu ciągu węzłów  $v_0, \dots, v_M$ , tablica z tymi węzłami (`vknots`), liczba elementów brzegu (`nelem`) i tablica `bound`, której elementy reprezentują brzeg dziedziny płata zgodnie z opisem w p. 7.21.1.

Następne 6 parametrów procedury określa siatkę prostych, których przecięcia z dziedziną płata mają być wyznaczone. Siatka składa się z prostych „pionowych” (linii stałego parametru  $u$ ) i „poziomych” (linii stałego parametru  $v$ ).

Linie „pionowe” odpowiadają węzłom  $u_n, \dots, u_{N-n}$  (a zatem mają niepuste przecięcia z dziedziną płata nieobciętego) i dodatkowo liczbom dzielącym każdy z przedziałów  $[u_i, u_{i+1}]$ ,  $i = n, \dots, N - n - 1$ , na podprzedziały o równych długościach. Domyślna liczba tych podprzedziałów jest równa wartości parametru `nu`, ale

jest ona dobierana tak, aby długość podprzedziałów była nie mniejsza niż wartość parametru  $au$  i nie większa niż wartość parametru  $bu$ .

W podobny sposób parametry  $nv$ ,  $av$  i  $bv$  określają zbiór prostych „poziomych” (tj. linii stałego parametru  $v$ ) generowany przez procedurę.

Parametr `maxinters` określa maksymalną spodziewaną liczbę przecięć prostej z brzegiem dziedziny płata obciętego. Stosownie do wartości tego parametru procedura rezerwuje tablicę na przecięcia i w razie jej przepełnienia może zawieść.

Ostatnie trzy parametry to procedury wyjściowe. Każdy z nich może mieć wartość `NULL`, co oznacza, że odpowiednie wyniki nie będą przez procedurę wywoływane.

Pierwsza z procedur, `NotifyLine`, jest wywoływana dla każdej nowej prostej „pionowej” lub „poziomej” z wygenerowanej przez procedurę siatki. Pierwszy parametr (typu `char`) tej procedury ma wartość 1 jeśli prosta jest pionowa, albo 2 jeśli pozioma. Drugi parametr określa numer odpowiedniego przedziału między węzłami, a kolejne dwa parametry to punkty końcowe odcinka będącego przecięciem prostej z dziedziną płata nieobciętego. Na przykład jeśli pierwszy parametr ma wartość 1, a drugi  $k$ , to prosta jest „pionowa”, tj. jest linią stałego parametru  $u$ , który jest liczbą z przedziału  $[u_k, u_{k+1})$ . Liczba ta jest też wartością współrzędnej  $x$  punktów przekazanych jako trzeci i czwarty parametr.

Procedura wyjściowa `DrawLine` jest wywoływana po znalezieniu przecięć brzegu dziedziny płata obciętego z prostą, dla *każdej* pary kolejnych punktów przecięcia. Punkty te są przekazywane jako pierwsze dwa parametry. Trzeci parametr ma wartość, która jest indeksem punktów wewnątrz odcinka (zobacz p. 7.21.1). Jeśli brzeg jest zorientowany w ten sposób, że podczas jego obchodzenia mamy wewnątrz dziedziny po lewej stronie, to indeks ten zawsze będzie miał wartość 1 (co oznacza, że odcinek leży w dziedzinie) albo 0 (co oznacza, że odcinek leży poza dziedziną). W ogólności orientacja poszczególnych krzywych zamkniętych, z których składa się brzeg, może być inna (tak jest np. w przykładzie w p. 7.21.1). Określenie które odcinki leżą w dziedzinie zależy od procedury `DrawLine` (może ona np. być oparta o regułę parzystości: w dziedzinie leżą te odcinki, których punkty mają indeks nieparzysty).

Procedura `DrawCurve` jest wywoływana w celu narysowania elementów brzegu dziedziny płata obciętego. Pierwszy jej parametr ma wartość  $d = 2$  albo  $3$ , co oznacza odpowiednio, że płaska krzywa Béziera jest wielomianowa albo wymierna (w reprezentacji jednorodnej). Drugi parametr określa stopień  $n$  krzywej (jeśli ma wartość 1, to krzywa jest odcinkiem, co można wykorzystać). Trzeci parametr jest tablicą punktów kontrolnych, czyli  $(n + 1)d$  liczb zmiennopozycyjnych, które są współrzędnymi tych punktów.

## Przykład — procedury wyjściowe dla obrazków kreskowych

Poniżej są opisane procedury przykładowe, których zadaniem jest wykonanie obrazów na rys. 7.20 w języku PostScript.

Obrazek z lewej strony przedstawia dziedzinę płyta B-sklejanego, tj. przecięcia linii stałego parametru odpowiadających węzłom płyta z dziedziną oraz jej brzeg. Odcinki linii stałego parametru są rysowane przez podaną niżej procedurę DrawLine1, która używa procedury MapPoint do odpowiedniego odwzorowania (przeskalowania i przesunięcia) końców odcinków.

```
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*/
```

Brzeg dziedziny został narysowany przez procedurę DrawCurve1, której skrócona wersja jest taka (pełna wersja jest w pliku 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 ) {
        /* Krzywa Béziera stopnia 1 jest odcinkiem, więc ten */
        /* przypadek jest traktowany osobno. Tablica cp zawiera 4 */
        /* lub 6 liczb, tj. współrzędne kartezjanskie lub jednorodne */
        /* (zależnie od wartości parametru dim) końców odcinka. */
        ...
    }
    else /* degree > 1, rysujemy łamaną */ {
        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*/

```

Wywołanie procedury `mbs_DrawTrimBSPatchDomf`, które spowodowało powstanie tego rysunku, ma postać

```

mbs_DrawTrimBSPatchDomf ( n1, NN1, u1, m1, MM1, v1, 8, boundary1,
                        1, 2.0, 2.0, 1, 2.0, 2.0,
                        20, NULL, DrawLine1, DrawCurve1 );

```

Pierwsze 6 parametrów opisuje stopień i węzły, czyli w szczególności dziedzinę płata nieobciętego, zgodnie z wcześniejszym opisem. Dziedzina ta jest prostokątem  $[0,4] \times [0,4]$ , a długości przedziałów między węzłami są między 1 i 1.5. Dlatego wartości parametrów `nu`, `au`, `bu`, `nv`, `av`, `bv` zapewniają rysowanie tylko linii stałego parametru odpowiadające węzłom płata.

Wykonanie takiego obrazka płata obciętego jak na rys. 7.20 z prawej strony wymaga odwzorowania odpowiednich linii w dziedzinie na płat, a następnie ich zrzutowanie. Procedura `DrawLine2`, która została użyta w tym przypadku, ma postać

```

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*/
```

Kilka słów wyjaśnienia: zamiast krzywej procedura rysuje łamaną. Liczba jej odcinków jest zależna od długości odcinka w dziedzinie płata, którego obraz na płacie jest rysowany (można by też wziąć pod uwagę kształt płata, ale tak jest najprościej). Procedura DrawLine2 ma dostęp do reprezentacji płata (tj. węzłów i punktów kontrolnych) poprzez zmienne globalne. Punkty płata są obliczane za pomocą algorytmu de Boora (przez wywołanie mbs\_deBoorP3f). Można zmniejszyć koszt obliczania tych punktów, podając jako parametr NotifyLine procedurę, której zadaniem byłoby wyznaczenie reprezentacji B-sklejanej (ewentualnie kawałkami Béziera) krzywej stałego parametru u albo v. Wywołania procedury podanej jako parametr DrawLine po wywołaniu NotifyLine mają na celu narysowanie łuków tej krzywej stałego parametru.

## 8. Biblioteka libraybez

Biblioteka libraybez zawiera procedury, których podstawowym (ale nie jedynym) zadaniem jest wspomaganie śledzenia promieni, a dokładniej wyznaczanie przecięć promieni z płaszczyznami Béziera. W tym celu tworzone jest drzewo rekurencyjnego binarnego podziału płaszczyzny, które ma na celu przyspieszenie (przez wyeliminowanie wielokrotnego wykonywania tej samej pracy) rozwiązywania równań opisujących przecięcia.

Ewentualne rozszerzenia tej biblioteki powinny obejmować konstrukcję drzew dla płaszczyzn B-sklejanych, także obciętych, oraz obsługę drzew z dodatkowymi atrybutami, w celu umożliwienia rozwiązywania zadań takich jak wyznaczanie przecięć powierzchni.

### 8.1 Deklaracje i procedury wspólne

```
typedef struct {  
    float xmin, xmax, ymin, ymax, zmin, zmax;  
} Box3f;
```

Struktura typu Box3f reprezentuje prostopadłościan. W reprezentacji fragmentu płaszczyzny w drzewie podziału jest ona stosowana do lokalizacji tego fragmentu w przestrzeni (fragment leży wewnątrz odpowiedniego prostopadłościanu).

```
typedef struct {  
    point3f p;  
    vector3f nv;  
    float u, v, t;  
} RayObjectIntersf, *RayObjectIntersfp;
```

Struktura typu RayObjectIntersf reprezentuje punkt przecięcia promienia z płaszczyzną. Struktura ta składa się z następujących pól: p — punkt wspólny płaszczyzny i promienia, nv — wektor normalny płaszczyzny w tym punkcie, u, v, t — parametry płaszczyzny i promienia odpowiadające punktowi przecięcia.

## 8.2 Drzewa binarne dla wielomianowych płatów Béziera

```
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;
```

Struktura typu `_BezPatchTreeVertexf` reprezentuje wierzchołek drzewa binarnego podziału wielomianowego płata Béziera  $p$ .

Pola tej struktury służą do przechowania następujących informacji: `left`, `right`, `up` — wskaźniki odpowiednio lewego i prawego poddrzewa oraz wskaźnik „do góry”, tj. do wierzchołka, którego lewe lub prawe poddrzewo reprezentuje dany wierzchołek, `ctlpoints` — wskaźnik tablicy punktów kontrolnych fragmentu płata odpowiadającego danemu wierzchołkowi, `u0`, `u1`, `v0`, `v1` — liczby określające dziedzinę  $[u_0, u_1] \times [v_0, v_1]$  fragmentu płata, `bbox` — prostopadłościan zawierający fragment płata, `pcent` — punkt  $p((u_0 + u_1)/2, (v_0 + v_1)/2)$ , `maxder` — górne oszacowanie długości wektora pochodnych cząstkowych fragmentu płata ze względu na lokalne parametry, `level` — poziom wierzchołka w drzewie, `divdir` — wskaźnik kierunku podziału fragmentu płata, `pad` — pole nieużywane (wyrównujące wielkość struktury do liczby parzystej).

```
typedef struct {
    unsigned char      n, m;
    unsigned int       cpsize;
    BezPatchTreeVertexfp root;
} BezPatchTreef, *BezPatchTreefp;
```

Struktura typu `BezPatchTreef` reprezentuje drzewo binarnego podziału wielomianowego płata Béziera. Pola tej struktury są następujące: `n`, `m` — stopień płata ze względu na zmienne  $u$  i  $v$ , `cpsize` — ilość miejsca potrzebnego do przechowywania punktów kontrolnych, `root` — wskaźnik korzenia drzewa.



```
BezPatchTreefp
```

```
  rbez_NewBezPatchTreef ( unsigned char n, unsigned char m,
                          float u0, float u1, float v0, float v1,
                          point3f *ctlpoints );
```

Procedura `rbez_NewBezPatchTreef` tworzy drzewo binarnego podziału wielomianowego płata Béziera i zwraca wskaźnik struktury, która reprezentuje to drzewo. Drzewo początkowo składa się tylko z korzenia, który reprezentuje cały płat.

Parametry `n` i `m` określają stopień płata odpowiednio ze względu na zmienne `u` i `v`. Parametry `u0`, `u1`, `v0` i `v1` określają dziedzinę płata, tj. prostokąt  $[u_0, u_1] \times [v_0, v_1]$  (jeśli płat powstał z podziału płata B-sklejanego, to liczby te powinny być odpowiednimi węzłami).

Parametr `ctlpoints` jest tablicą punktów kontrolnych płata.

Wartością procedury jest wskaźnik do struktury opisującej drzewo. Obszar pamięci na tę strukturę i na struktury opisujące wszystkie wierzchołki drzewa jest rezerwowany za pomocą funkcji `malloc`.

```
void rbez_DestroyBezPatchTreef ( BezPatchTreefp tree );
```

Procedura `rbez_DestroyBezPatchTreef` zwalnia pamięć zajmowaną przez drzewo binarnego podziału płata. Parametr `tree` jest wskaźnikiem struktury reprezentującej drzewo.

```
BezPatchTreeVertexp
```

```
  rbez_GetBezLeftVertexf ( BezPatchTreefp tree,
                          BezPatchTreeVertexfp vertex );
```

```
BezPatchTreeVertexfp
```

```
  rbez_GetBezRightVertexf ( BezPatchTreefp tree,
                           BezPatchTreeVertexfp vertex );
```

Procedury `rbez_GetBezLeftVertexf` i `rbez_GetBezRightVertexf` zwracają wskaźniki odpowiednio lewego lub prawego poddrzewa danego wierzchołka drzewa binarnego podziału płata.

Parametry: `tree` — wskaźnik struktury opisującej drzewo, `vertex` — wskaźnik jednego z wierzchołków tego drzewa.

Wartością procedury jest wskaźnik do korzenia odpowiedniego (lewego albo prawego) poddrzewa. Jeśli wierzchołek ten nie istnieje, to procedura dokonuje podziału fragmentu płata reprezentowanego przez wierzchołek `*vertex` i tworzy lewy i prawy wierzchołek (dla każdego wierzchołka oba poddrzewa istnieją albo oba nie istnieją), a następnie zwraca odpowiedni wskaźnik.

```
int rbez_FindRayBezPatchIntersf ( BezPatchTreef *tree,
                                ray3f *ray,
                                int maxlevel, int maxinters,
                                int *ninters, RayObjectIntersf *inters );
```

Procedura FindRayBezPatchIntersf oblicza punkty przecięcia promienia (półprostej) z wielomianowym płatem Béziera w  $\mathbb{R}^3$ .

Parametry: tree — wskaźnik drzewa binarnego podziału płata; ray — wskaźnik promienia (struktura ray3f jest zdefiniowana w pliku geomf.h; maxlevel — ograniczenie wysokości drzewa binarnego podziału (procedura nie będzie tworzyć wierzchołków drzewa na wyższym poziomie); maxinters — długość tablicy inters, w której procedura ma umieścić wyniki. Tablica ta musi mieć co najmniej taką długość, procedura zakończy działanie po znalezieniu najwyższej tylu przecięć. Wartość parametru \*ninters na wyjściu jest liczbą znalezionych przecięć.

Wartością procedury jest liczba znalezionych punktów przecięcia.

### 8.3 Drzewa binarne dla wymiernych płatów Béziera

Drzewa binarnego podziału wymiernych płatów Béziera są oprogramowane w prawie identyczny sposób jak drzewa binarnego podziału płatów wielomianowych. Wszystkie struktury danych i procedury opisane w poprzednim punkcie mają tu swoje odpowiedniki.

```
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;
```

Struktura typu \_RBezPatchTreeVertexf reprezentuje wierzchołek drzewa binarnego podziału wymiernego płata Béziera p.

Pola tej struktury służą do przechowania następujących informacji: left, right, up — wskaźniki odpowiednio lewego i prawego poddrzewa oraz wskaźnik „do góry”, tj. do wierzchołka, którego lewe lub prawe poddrzewo reprezentuje dany wierzchołek, ctlpoints — wskaźnik tablicy punktów kontrolnych jednorodnego płata Béziera reprezentującego fragment płata odpowiadającego danemu wierzchołkowi, u0, u1, v0, v1 — liczby określające dziedzinę  $[u_0, u_1] \times [v_0, v_1]$  frag-

mentu płata, bbox — prostopadłościan zawierający fragment płata, pcent — punkt  $p((u_0 + u_1)/2, (v_0 + v_1)/2)$ , maxder — górne oszacowanie długości wektora pochodnych cząstkowych fragmentu płata ze względu na lokalne parametry, level — poziom wierzchołka w drzewie, divdir — wskaźnik kierunku podziału fragmentu płata, pad — pole nieużywane (wyrównujące wielkość struktury do liczby parzystej).

```
typedef struct {
    unsigned char      n, m;
    unsigned int        csize;
    RBezPatchTreeVertexfp root;
} RBezPatchTreefp, *RBezPatchTreefp;
```

Struktura typu RBezPatchTreefp reprezentuje drzewo binarnego podziału płata Béziera. Pola tej struktury są następujące: n, m — stopień płata ze względu na zmienne u i v, csize — ilość miejsca potrzebnego do przechowywania punktów kontrolnych, root — wskaźnik korzenia drzewa.

```
RBezPatchTreefp
rbez_NewRBezPatchTreef ( unsigned char n, unsigned char m,
                          float u0, float u1, float v0, float v1,
                          point4f *ctlpoints );
```

Procedura rbez\_NewRBezPatchTreef tworzy drzewo binarnego podziału wymiennego płata Béziera i zwraca wskaźnik struktury, która reprezentuje to drzewo. Drzewo początkowo składa się tylko z korzenia, który reprezentuje cały płat.

Parametry n i m określają stopień płata odpowiednio ze względu na zmienne u i v. Parametry u0, u1, v0 i v1 określają dziedzinę płata, tj. prostokąt  $[u_0, u_1] \times [v_0, v_1]$  (jeśli płat powstał z podziału płata B-sklejanego, to liczby te powinny być odpowiednimi węzłami).

Parametr ctlpoints jest tablicą punktów kontrolnych płata jednorodnego.

Wartością procedury jest wskaźnik do struktury opisującej drzewo. Obszar pamięci na tę strukturę i na struktury opisujące wszystkie wierzchołki drzewa jest rezerwowany za pomocą funkcji malloc.

```
void rbez_DestroyRBezPatchTreef ( RBezPatchTreefp tree );
```

Procedura rbez\_DestroyRBezPatchTreef zwalnia pamięć zajmowaną przez drzewo binarnego podziału płata. Parametr tree jest wskaźnikiem struktury reprezentującej drzewo.

```

RBezPatchTreeVertexp
  rbez_GetRBezLeftVertexf ( RBezPatchTreefp tree,
                           RBezPatchTreeVertexfp vertex );
RBezPatchTreeVertexfp
  rbez_GetRBezRightVertexf ( RBezPatchTreefp tree,
                             RBezPatchTreeVertexfp vertex );

```

Procedury `rbez_GetRBezLeftVertexf` i `rbez_GetRBezRightVertexf` zwracają wskaźniki odpowiednio lewego lub prawego poddrzewa danego wierzchołka drzewa binarnego podziału płata.

Parametry: `tree` — wskaźnik struktury opisującej drzewo, `vertex` — wskaźnik jednego z wierzchołków tego drzewa.

Wartością procedury jest wskaźnik do korzenia odpowiedniego (lewego albo prawego) poddrzewa. Jeśli wierzchołek ten nie istnieje, to procedura dokonuje podziału fragmentu płata reprezentowanego przez wierzchołek `*vertex` i tworzy lewy i prawy wierzchołek (dla każdego wierzchołka oba poddrzewa istnieją albo oba nie istnieją), a następnie zwraca odpowiedni wskaźnik.

```

int rbez_FindRayRBezPatchIntersf ( RBezPatchTreef *tree,
                                   ray3f *ray,
                                   int maxlevel, int maxinters,
                                   int *ninters, RayObjectIntersf *inters );

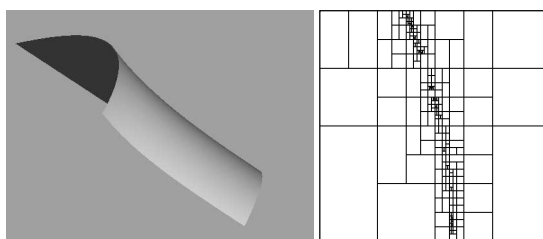
```

Procedura `FindRayRBezPatchIntersf` oblicza punkty przecięcia promienia (półprostej) z wymiernym płatem Béziera w  $\mathbb{R}^3$ .

Parametry: `tree` — wskaźnik drzewa binarnego podziału płata; `ray` — wskaźnik promienia (struktura `ray3f` jest zdefiniowana w pliku `geomf.h`; `maxlevel` — ograniczenie wysokości drzewa binarnego podziału (procedura nie będzie tworzyć wierzchołków drzewa na wyższym poziomie); `maxinters` — długość tablicy `inters`, w której procedura ma umieścić wyniki. Tablica ta musi mieć co najmniej taką długość, procedura zakończy działanie po znalezieniu najwyżej tylu przecięć. Wartość parametru `*ninters` na wyjściu jest liczbą znalezionych przecięć.

Wartością procedury jest liczba znalezionych punktów przecięcia.

Rysunek 8.1 przedstawia obrazek wymiernego płata Béziera stopnia (5,5) wykonany przy użyciu tej procedury. Pełny kod programu, który utworzył ten obrazek jest w pliku `../cpict/raybez.c`.



Rys. 8.1. Obraz płata wykonany metodą śledzenia promieni za pomocą procedury `rbez_FindRayRBezPatchIntersf`. Obok podział dziedziny płata dokonany przez procedury obsługi drzewa podziału



# 9. Biblioteka libeghole

Biblioteka libeghole zawiera procedury wypełniania wielokątnego otworu w powierzchni sklejaną z płatów stopnia  $(3, 3)$ . Podstawy teoretyczne i opis konstrukcji są opisane w pracy *Konstrukcje powierzchni gładko wypełniających wielokątne otwory*.

## 9.1 Przygotowanie danych

Dane dla procedur wypełniania otworu składają się z czterech części:

- Liczby wierzchołków otworu,  $k$ ,
- $k$  jedenastoelementowych ciągów węzłów,
- $12k + 1$  punktów kontrolnych dziediny,
- $12k + 1$  punktów kontrolnych powierzchni.

Oprócz wymienionych wyżej danych można określić **więzy**, czyli równania liniowe, które mają być spełnione przez powierzchnię wypełniającą otwór. Sposób ich przygotowania jest opisany w p. 9.3.4.

Liczba całkowita  $k$  musi być nie mniejsza niż 3 i nie większa niż 16. Powierzchnia z otworem składa się z  $3k$  płatów wielomianowych stopnia  $(3, 3)$ , otaczających  $k$ -kątny otwór.

Ciągi węzłów  $u_0^{(n)}, \dots, u_{10}^{(n)}$ , dla  $n = 0, \dots, k - 1$ , muszą spełniać warunki

$$u_0^{(n)} \leq u_1^{(n)} < \dots < u_9^{(n)} \leq u_{10}^{(n)},$$

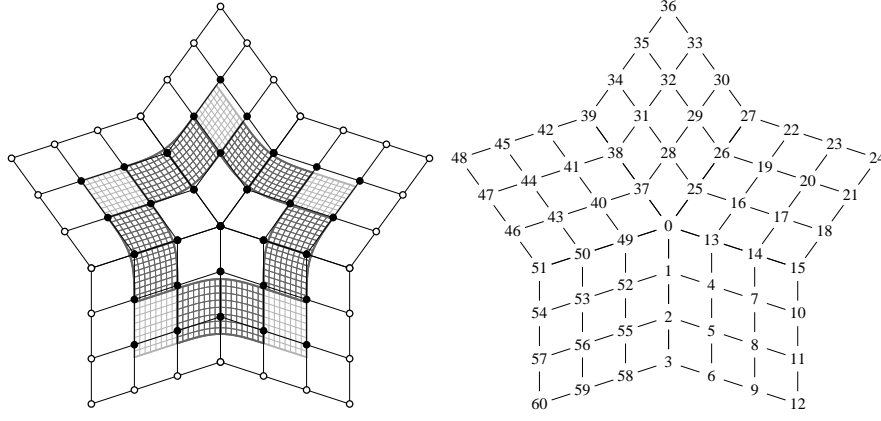
oraz

$$u_i^{(n)} - u_1^{(n)} = u_0^{(m)} - u_{10-i}^{(m)},$$

dla  $m = (n+2) \bmod k$  i  $i = 4, \dots, 9$ . Ciągi te podaje się w jednowymiarowej tablicy o długości  $11k$ ; musi ona zawierać wyrazy tych ciągów kolejno po sobie, bez przerw.

**Punkty kontrolne dziediny**,  $c_0, \dots, c_{12k}$  leżą w płaszczyźnie i stanowią wierzchołki siatki kontrolnej. Schemat siatki i sposób numeracji tych punktów jest pokazany na rysunku 9.1. Siatka ta zawiera  $k$  siatek kontrolnych płaskich bikubicznych płatów B-sklejanych, które mają wspólne kawałki wielomianowe.

Dla  $n = 0, \dots, k-1$   $n$ -ty płat B-sklejany jest reprezentowany przez ciągi węzłów  $u_0^{(n)}, \dots, u_{10}^{(n)}$  i  $u_0^{(m)}, \dots, u_7^{(m)}$ , gdzie  $m = (n+1) \bmod k$ , oraz punkty kontrolne  $c_{ij}^{(n)}$ ,  $i = 0, \dots, 6$ ,  $j = 0, \dots, 3$ , takie że:



Rys. 9.1. Reprezentacja dziedziny powierzchni wypełniającej otwór

- $c_{ij}^{(n)} = c_{12(n+1)-3j-i}$  dla  $i = 0, \dots, 2, j = 0, \dots, 3$
- $c_{ij}^{(n)} = c_{12m-3i-j}$  dla  $i = 3, \dots, 6, j = 0, \dots, 2$ , gdzie  $m = (n+1) \bmod k$ ,
- $c_{3,3}^{(n)} = c_0$ ,
- $c_{ij}^{(n)} = c_{12m+i-3}$  dla  $i = 4, \dots, 6, j = 3$ , gdzie  $m = (n+2) \bmod k$ .

Płaty B-sklejane reprezentowane przez te ciągi węzłów i punkty kontrolne muszą być regularne i z wyjątkiem wspólnych fragmentów, których istnienie zapewnia reprezentacja, rozłączne.

Zbiór punktów płatów B-sklejanych reprezentowanych przez ciągi węzłów i punkty kontrolne opisane wyżej jest dziedziną pewnej parametryzacji powierzchni z otworem, zaś otoczony tymi płatami obszar  $\Omega$ , który jest krzywoliniowym  $k$ -kątem, jest dziedziną pewnej parametryzacji powierzchni wypełniającej, która ma być skonstruowana. Dla takiej parametryzacji będzie określony funkcjonal  $F$ , którego wartość jest miarą jakości powierzchni (ma on być minimalizowany). Zmieniając punkty kontrolne  $c_i$  zmienia się ten funkcjonal, co wpływa na wynik konstrukcji.

**Punkty kontrolne powierzchni**,  $b_0, \dots, b_{12k}$ , leżą w przestrzeni o wymiarze  $d$  (w praktyce zwykle będzie  $d = 3$  dla powierzchni wielomianowej, albo  $d = 4$ , jeśli ma być wypełniony otwór w powierzchni będącej jednorodną reprezentacją powierzchni kawałkami wymiernej). Siatka kontrolna powierzchni jest zbudowana analogicznie jak siatka kontrolna dziedziny, tj. można w niej wyróżnić  $k$  siatek kontrolnych płatów B-sklejanych stopnia  $(3,3)$ . Dla  $n = 0, \dots, k-1$  płat  $n$ -ty jest reprezentowany przez ciągi węzłów  $u_0^{(n)}, \dots, u_{10}^{(n)}$  i  $u_0^{(m)}, \dots, u_7^{(m)}$ , gdzie  $m = (n+1) \bmod k$ , oraz punkty kontrolne  $b_{ij}^{(n)}$ ,  $i = 0, \dots, 6, j = 0, \dots, 3$ , które są punktami  $b_l$  o indeksach  $l$  określonych tak samo jak indeksy punktów kontrolnych  $c_{ij}^{(n)}$  płatów B-sklejanych otaczających dziedzinę.



Tablica punktów kontrolnych powierzchni, która ma być parametrem procedur konstruujących wypełnienie otworu, składa się z  $(12k + 1)d$  liczb zmiennopozycyjnych — każde kolejne  $d$  z nich to współrzędne kolejnego punktu  $\mathbf{b}_l$ . Punkty  $\mathbf{c}_{ij}^{(n)}$  i  $\mathbf{b}_{ij}^{(n)}$  dla  $i \in \{0, 6\}$  oraz dla  $j = 0$  nie mają wpływu na wynik konstrukcji (tj. na powierzchnię wypełniającą otwór), podobnie jak węzły  $u_0^{(n)}$  i  $u_{10}^{(n)}$ , ale trzeba je podać. Na rysunku 9.1 punkty, które mają wpływ na wynik konstrukcji, są zaznaczone czarnymi kropkami.

## 9.2 Minimum teorii

Dokładny opis podstaw teoretycznych konstrukcji realizowanych przez procedury z biblioteki `libeghole` znajduje się w pracy *Konstrukcje powierzchni gładko wypełniających wielokątne otwory*. Opis poniżej zawiera tylko wiadomości teoretyczne niezbędne do poprawnego przygotowania danych dla procedur.

### 9.2.1 Bazy używane w konstrukcjach

Aby skonstruować powierzchnię wypełniającą otwór, procedury biblioteczne konstruują bazę  $\phi_0, \dots, \phi_{n+m}$  pewnej przestrzeni liniowej  $V$ , do której należą funkcje skalarne klasy  $C^1$  albo  $C^2$  opisujące współrzędne powierzchni wypełniającej. Powierzchnia ta określona jest wzorem

$$\mathbf{p} = \sum_{i=0}^{n-1} \mathbf{a}_i \phi_i + \sum_{i=0}^{m-1} \mathbf{b}_i \phi_{n+i}. \quad (9.1)$$

Wektory  $\mathbf{b}_i \in \mathbb{R}^d$  są danymi punktami kontrolnymi powierzchni z otworem. Siatka kontrolna powierzchni, której to są wierzchołki, jest grafem izomorficznym z siatką kontrolną dziedziny pokazaną na rysunku 9.1 i jej wierzchołki są ponumerowane analogicznie. Wierzchołki te podaje się (w takiej kolejności) w tablicy przekazywanej procedurom konstrukcji powierzchni jako parametr. Tablica zawiera  $12k + 1$  punktów kontrolnych, z których  $m = 6k + 1$  ma wpływ na powierzchnię wypełniającą otwór.

Zadaniem procedur konstrukcji jest obliczenie wektorów  $\mathbf{a}_0, \dots, \mathbf{a}_{n-1} \in \mathbb{R}^d$ , które minimalizują pewne funkcjonały przyjęte za miarę „brzydoty” powierzchni. Funkcje bazowe  $\phi_0, \dots, \phi_{n+m-1}$  są określone w obszarze  $\Omega \in \mathbb{R}^2$ , który jest otworem w płaskiej powierzchni reprezentowanej przez węzły i siatkę kontrolną dziedziny, opisaną w poprzednim punkcie. Obszar  $\Omega$  jest podzielony na  $k$  czworokątów krzywoliniowych  $\Omega_0, \dots, \Omega_{k-1}$ , które są obrazami kwadratu jednostkowego w przekształceniach  $\mathbf{d}_0, \dots, \mathbf{d}_{k-1}$ , zwanych **płatami dziedziny**. Funkcja  $\phi_i$  jest zdefiniowana wzorem

$$\phi_i(\mathbf{x}) = p_{li}(\mathbf{d}_l^{-1}(\mathbf{x})) \quad \text{dla } \mathbf{x} \in \Omega_i,$$

za pomocą płatów dziedziny i funkcji  $p_{i0}, \dots, p_{i,k-1}$ , zwanych **płatami funkcji bazowych**. Powierzchnia wypełniająca otwór składa się z  $k$  płatów wielomianowych lub sklejanych  $p_0, \dots, p_{k-1}$ , określonych wzorem

$$p_l = \sum_{i=0}^{n-1} a_i p_{li} + \sum_{i=0}^{m-1} b_i p_{l,i+n}.$$

Reprezentacja Béziera lub B-sklejana tych płatów jest ostatecznym wynikiem konstrukcji.

Funkcje bazowe  $\phi_0, \dots, \phi_{n+m-1}$  można podzielić na dwa podzbiory. Funkcje  $\phi_0, \dots, \phi_n$  spełniają jednorodny warunek brzegowy, tj. ich wartości i pochodne cząstkowe rzędu 1 (lub 1 i 2) na brzegu obszaru  $\Omega$  są równe 0. Funkcje  $\phi_n, \dots, \phi_{n+m-1}$  spełniają warunki brzegowe dobrane tak, aby dla dowolnych wektorów  $a_0, \dots, a_{n-1}$  powierzchnia opisana wzorem (9.1) łączyła się z powierzchnią daną z ciągłością płaszczyzny stycznej lub krzywizny. Funkcje  $\phi_n, \dots, \phi_{n+m-1}$  i ich pochodne rzędu 1 i 2 (albo 1,  $\dots$ , 4) w punkcie środkowym, tj. wspólnym punkcie wszystkich obszarów  $\Omega_l$  mają wartość 0.

Półproste styczne do wspólnych krzywych obszarów  $\Omega_0, \dots, \Omega_{k-1}$  są nachylone pod kątami  $\alpha_0, \dots, \alpha_{k-1}$ , przy czym  $\alpha_0 < \alpha_1 < \dots < \alpha_{k-1} < \alpha_0 + 2\pi$ . Zbiór  $\Delta = \{\alpha_0, \dots, \alpha_{k-1}\}$  nazywa się **podziałem kąta pełnego**. Niech  $h$  oznacza liczbę par  $\{\alpha_i, \alpha_i + \pi\} \subset \Delta$ . W przypadku powierzchni wypełniającej klasy  $G^1$  lub  $G^1 Q^2$ , oznaczmy

$$n' = 3 + \max\{k, h + 3\}.$$

Liczba  $n'$  jest liczbą elementów tzw. **bazy podstawowej**, w której wszystkie płaty funkcji bazowych są bikubicznymi płatami Coonsa, określonymi przez wielomiany stopnia 5 (są to zatem płaty wielomianowe stopnia (5,5)).

Można przyjąć  $n = n'$  lub  $n = n' + 4k$ ; w tym drugim przypadku mamy **bazę rozszerzoną** odpowiednio powiększonej przestrzeni  $V_0 = \text{lin}\{\phi_0, \dots, \phi_{n-1}\} \subset V$ . Płaty funkcji bazowych dołączonych  $4k$  funkcji bazowych są iloczynami tensorowymi wielomianów Bernsteina  $B_2^5$  i  $B_3^5$ . Podobnie jak w przypadku użycia bazy podstawowej, wynik konstrukcji składa się z  $k$  płatów Béziera stopnia (5,5).

Jeszcze jedna możliwość to wypełnianie otworu płatami B-sklejanymi stopnia (5,5). Baza odpowiedniej przestrzeni  $V_0$  oprócz elementów bazy podstawowej zawiera dwie rodziny funkcji: w pierwszej z nich płaty funkcji bazowych są iloczynami tensorowymi funkcji B-sklejanych  $N_i^5$  i  $N_j^5$  dla  $i, j \in \{2, \dots, 3 + n_k m_2\}$ . Musi być  $1 \leq m_2 \leq 4$ . Płaty funkcji bazowych drugiej rodziny funkcji są bikubicznymi płatami Coonsa, określonymi przez krzywe sklepane stopnia 5, mające  $n_k m_1$  węzłów, gdzie  $1 \leq m_1 \leq 2$ . Wymiar przestrzeni  $V_0$  jest wtedy równy  $n' + k((2 + n_k m_2)^2 + 2n_k m_1)$ . Wynik konstrukcji ma postać  $k$  płatów B-sklejanych stopnia (5,5).

Dla konstrukcji powierzchni klasy  $G^2$  niech

$$n' = 6 + \max\{k, h + 4\} + \max\{2k, 2h + 5\}.$$

Jeśli  $n = n'$ , to mamy bazę podstawową, w której płaty funkcji bazowych są dwupiętnymi płatami Coonsa stopnia  $(9, 9)$ .

Baza rozszerzona zawiera dodatkowo 16k funkcji, których płaty funkcji bazowych są iloczynami tensorowymi wielomianów Bernsteina  $B_3^9, \dots, B_{y_6}$ . W obu przypadkach wynik konstrukcji składa się z  $k$  wielomianowych płatów stopnia  $(9, 9)$ , reprezentowanych w postaci Béziera.

Można też użyć bazy sklejaney, określonej za pomocą trzech parametrów,  $n_k$ ,  $m_1$  i  $m_2$ ; w tym przypadku przestrzeń  $V_0$  ma wymiar

$$n' + k((4 + n_k m_2)^2 + 3n_k m_1),$$

a wynik konstrukcji składa się z  $k$  płatów B-sklejanych stopnia  $(9, 9)$ . Musi być  $1 \leq m_1 \leq 3$ ,  $1 \leq m_2 \leq 7$ .

### 9.2.2 Kryteria optymalizacji powierzchni klasy $G^1$

Powierzchnie wypełniające klasy  $G^1$  mają stopień  $(5, 5)$ . Powierzchnie te są konstruowane przez minimalizację funkcjonałów

$$F_a(\mathbf{p}) \stackrel{\text{def}}{=} \int_{\Omega} \|\Delta \mathbf{p}\|_2^2 d\Omega,$$

$$F_b(\mathbf{p}) \stackrel{\text{def}}{=} \int_{\Omega} H^2 \sqrt{\det G} d\Omega,$$

gdzie  $G$  oznacza macierz pierwszej formy podstawowej, a  $H$  oznacza krzywiznę średnią powierzchni. Funkcjonał  $F_a$  jest formą kwadratową, która ma jednoznacznie określone minimum (także dla dowolnych niesprzecznych węzłów). Funkcjonał  $F_b$ , podobnie jak  $F_d$ , jest istotnie nieliniowy, a jego wartość zależy tylko od kształtu powierzchni. Minimalizacja funkcjonału  $F_b$  jest bardziej kłopotliwa i czasochłonna niż minimalizacja  $F_a$  i nie dla każdej powierzchni danej jest wykonalna.

### 9.2.3 Kryteria optymalizacji powierzchni klasy $G^2$

Wektory  $\mathbf{a}_0, \dots, \mathbf{a}_{n-1}$  są dobierane tak, aby zminimalizować wartość jednego z następujących funkcjonałów:

$$F_c(\mathbf{p}) \stackrel{\text{def}}{=} \int_{\Omega} \|\nabla \Delta \mathbf{p}\|_F^2 d\Omega,$$

$$F_d(\mathbf{p}) \stackrel{\text{def}}{=} \int_{\Omega} \|\nabla_{\mathcal{M}} H\|_2^2 \sqrt{\det G} d\Omega.$$

Kolejne wiersze macierzy  $\nabla \Delta \mathbf{p}$  są gradientami laplasjanów  $d$  funkcji skalarnych opisujących parametryzację  $\mathbf{p}$ ; symbol  $\|\cdot\|_F$  oznacza normę Frobeniusa, tj. pierwiastek sumy kwadratów wszystkich współczynników macierzy.

Funkcjonał  $F_c$  jest określony dla powierzchni w przestrzeni o dowolnym wymiarze  $d$ , natomiast w przypadku funkcjonału  $F_d$  musi być  $d = 3$ . Symbol  $H$  oznacza

krzywiznę średnią powierzchni,  $\nabla_{\mathcal{M}} H$  oznacza gradient krzywizny średniej na powierzchni, zaś  $G$  oznacza macierz pierwszej formy podstawowej.

Funkcjonał  $F_c$  jest formą kwadratową, której minimalizacja polega na rozwiązaniu układu równań liniowych

$$A\mathbf{a} = -B\mathbf{b}, \quad (9.2)$$

w którym występują macierze  $A = [a_{ij}]_{i,j}$  i  $B = [b_{ij}]_{i,j}$  o wymiarach  $n \times n$  i  $n \times m$ , których współczynniki

$$a_{ij} = a(\phi_i, \phi_j), \quad b_{ij} = a(\phi_i, \phi_{j+n}),$$

są wartościami formy dwuliniowej

$$a(f, g) = \int_{\Omega} \langle \nabla \Delta f, \nabla \Delta g \rangle d\Omega.$$

Macierz  $\mathbf{b}$  o wymiarach  $m \times d$  składa się z punktów kontrolnych powierzchni danej, macierz  $\mathbf{a}$ , o wymiarach  $n \times d$  składa się z niewiadomych wektorów  $\mathbf{a}_0, \dots, \mathbf{a}_{n-1}$ . Liczba  $d$  jest wymiarem przestrzeni, w której jest powierzchnia, na przykład 3 (ale może też być 4 w konstrukcji wielomianowej powierzchni jednorodnej, reprezentującej powierzchnię wymierną).

Wartość funkcjonału  $F_d$  nie zależy od parametryzacji powierzchni (która musi leżeć w  $\mathbb{R}^3$ ), tylko od jej kształtu. Znalezienie jego minimum jest trudniejsze, bardziej czasochłonne i nie zawsze wykonalne (wykonalność konstrukcji zależy od danej powierzchni z otworem). Polega ono na rozwiązaniu układu równań nieliniowych

$$\nabla F(\mathbf{a}_0, \dots, \mathbf{a}_{n-1}) = \mathbf{0}, \quad (9.3)$$

gdzie funkcja  $F$  jest określona wzorem

$$F(\mathbf{a}_0, \dots, \mathbf{a}_{n-1}) = F_d(\mathbf{p}),$$

dla parametryzacji  $\mathbf{p}$  danej wzorem

$$\mathbf{p}(u, v) = \begin{bmatrix} u \\ v \\ p(u, v) \end{bmatrix}, \quad p(u, v) = \sum_{i=0}^{n-1} a_i \phi_i + \sum_{i=0}^{m-1} b_i \phi_{n+i}.$$

Powierzchnia z otworem jest przedstawiana w takim układzie współrzędnych  $uvw$ , aby była w nim wykresem funkcji skalarnej,  $w = q(u, v)$ . Dziedzinę  $\Omega$  konstruuje się przez zrzutowanie powierzchni na płaszczyznę  $uv$ . Liczby  $b_0, \dots, b_{m-1}$  są współrzędnymi w punktów kontrolnych powierzchni danej.

### 9.2.4 Kryteria optymalizacji dla powierzchni klasy $G^1Q^2$

### 9.2.5 Równania więzów

Konstrukcje umożliwiają nakładanie więzów opisanych przez równania liniowe, np. więzów interpolacyjnych. Minimum funkcjonału  $F_c$  lub  $F_d$  można poszukiwać w zbiorze powierzchni, których współczynniki spełniają układ równań

$$C\mathbf{a} = \mathbf{d}. \quad (9.4)$$

Macierz  $C$  o wymiarach  $w \times n$ , musi być wierszowo regularna. Macierz  $\mathbf{d}$ , o wymiarach  $w \times d$  opisuje prawą stronę układu równań więzów, przy czym  $w$  jest liczbą więzów, a  $d$  jest wymiarem przestrzeni, w której jest powierzchnia; dla konstrukcji z minimalizacją funkcjonału  $F_d$  musi być  $d = 3$ .

Kolejne wiersze niewiadomej macierzy  $\mathbf{a}$  są wektorami  $\mathbf{a}_0, \dots, \mathbf{a}_{n-1}$ , występującymi we wzorze (9.1). Jeśli  $i$ -ty warunek nałożony na powierzchnię ma postać  $\mathbf{p}(\mathbf{x}) = \mathbf{p}_0$  (to jest warunek interpolacyjny, który narzuca punkt powierzchni odpowiadający punktowi  $\mathbf{x} \in \Omega$ ), to współczynniki w  $i$ -tym wierszu macierzy  $C$  mają być równe  $\phi_0(\mathbf{x}), \dots, \phi_{n-1}(\mathbf{x})$ , a  $i$ -ty wiersz macierzy  $\mathbf{d}$  ma być równy  $\mathbf{p}_0 - \sum_{i=0}^{m-1} \mathbf{b}_i \phi_{n+i}(\mathbf{x})$ . Podobnie, narzucenie wartości  $v$  pochodnej cząstkowej na przykład ze względu na  $u$  w punkcie  $\mathbf{x}$  następuje za pomocą równania więzu, dla którego odpowiedni wiersz macierzy  $C$  składa się z liczb  $\frac{\partial}{\partial u} \phi_0(\mathbf{x}), \dots, \frac{\partial}{\partial u} \phi_{n-1}(\mathbf{x})$ , a po prawej stronie (tj. w macierzy  $\mathbf{d}$ ) jest  $v - \sum_{i=0}^{m-1} \mathbf{b}_i \frac{\partial}{\partial u} \phi_{n+i}(\mathbf{x})$ .

W przypadku funkcjonału  $F_d$ , w razie zastosowania bazy rozszerzonej, dopuszczalne jest tylko narzucanie więzów interpolacyjnych w punkcie środkowym dziedziny (tj. we wspólnym punkcie obszarów  $\Omega_i$ ). Biblioteka zawiera procedury obliczające wartości funkcji bazowych i ich pochodnych cząstkowych w tym punkcie. Ponadto są też procedury udostępniające pełną informację o dowolnej funkcji bazowej. Na podstawie tej informacji program może obliczyć wartości dowolnego funkcjonału liniowego na wszystkich funkcjach bazowych. Wartości tych można następnie użyć jako współczynniki w równaniu więzu.

Opisany wyżej sposób narzuca jednocześnie i niezależnie na wszystkie współrzędne powierzchni wypełniającej więzy tej samej natury. Alternatywna postać równań więzów jest następująca:

$$C_0 \mathbf{a}_0 + \dots + C_{d-1} \mathbf{a}_{d-1} = \mathbf{d}. \quad (9.5)$$

Macierze  $C_0, \dots, C_{d-1}$  mają wymiary  $w \times n$ , przy czym macierz  $C = [C_0, \dots, C_{d-1}]$  (o wymiarach  $w \times nd$ ) musi być wierszowo regularna. Ta postać więzów jest ogólniejsza i dopuszcza określenie wartości dowolnego funkcjonału liniowego dla parametryzacji  $\mathbf{p}$ . Można na przykład narzucić tylko wartość pierwszej współrzędnej punktu  $\mathbf{p}(\mathbf{x})$ , przyjmując  $i$ -ty wiersz macierzy  $C_0$  złożony ze współczynników  $\phi_0(\mathbf{x}), \dots, \phi_{n-1}(\mathbf{x})$ , a w macierzach  $C_1, \dots, C_{d-1}$  umieszczając w  $i$ -tym wierszu zera.

### 9.2.6 Tabela procedur konstrukcji powierzchni

Orientację wśród dostępnych procedurach konstrukcji powierzchni wypełniających powinna ułatwić następująca tabela:

		Coons	Bézier	B-spline	Coons	Bézier	B-spline	Coons	Bézier	B-spline
L	$G^1$	1.	2.	3.	4.	5.		7.	8.	
	$G^2$	10.	11.	12.	13.	14.		16.	17.	
	$G^1Q^2$	19.	20.	21.	22.	23.		25.	26.	
NL	$G^1$	28.	29.	30.	31.	32.		34.	35.	
	$G^2$	37.	38.	39.	40.	41.		43.	44.	
	$G^1Q^2$	46.	47.	48.						
bez więzów				więzy (9.4)			więzy (9.5)			

Procedury zaznaczone w pierwszych trzech wierszach realizują konstrukcje z minimalizacją form kwadratowych (przez rozwiązanie układu równań liniowych).

Procedury w kolejnych trzech wierszach rozwiązują układy równań nieliniowych, w celu dokonania minimalizacji odpowiednich funkcjonałów niezależnych od parametryzacji.

Procedury w pierwszych trzech kolumnach dokonują konstrukcji bez nakładania więzów. W kolejnych trzech kolumnach są wyliczone procedury konstrukcji z więzami o postaci (9.4), a w następnych trzech z więzami (9.5).

Na górze każdej kolumny jest podana postać bazy używanej w konstrukcji. Nazwy procedur są następujące:

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 Sposób użycia procedur

### 9.3.1 Konstrukcja podstawowa

Konstrukcja powierzchni wypełniającej składa się z dwóch głównych etapów. Pierwszy etap polega na skonstruowaniu bazy przestrzeni liniowej  $V$ , której elementami są funkcje opisujące parametryzację powierzchni (tj. każdą ze współrzędnych), oraz obliczenie zależnych od tej bazy współczynników macierzy, które występują w rozwiązywanych w drugim etapie układów równań. To obliczenie jest dość czasochłonne, ale przetwarzana jest w nim tylko reprezentacja dziedziny.

W drugim etapie na podstawie punktów kontrolnych powierzchni i ewentualnie więzów (jeśli zostały one określone) jest obliczana prawa strona układu równań, który jest następnie rozwiązywany. Rozwiązanie układu jest używane do wyznaczenia  $k$  płatów Béziera stopnia  $(9, 9)$ , z których składa się powierzchnia wypełniająca. Drugi etap zabiera znacznie mniej czasu i w praktyce może być powtarzany wielokrotnie, gdy użytkownik programu interakcyjnego manipuluje punktami kontrolnymi powierzchni lub więzami (ale zmiana węzłów wymaga powtórzenia pierwszego etapu konstrukcji).

Pierwszy etap konstrukcji zostanie zrealizowany przez wykonanie następujących instrukcji:

```
GHoleDomainf *domain;
...
if ( !(domain = gh_CreateDomainf ( k, knots, domain_cp )) )
    exit ( 1 );
if ( !g2h_ComputeBasisf ( domain ) )
    exit ( 1 );
if ( !g2h_DecomposeMatrixf ( domain ) )
    exit ( 1 );
```

Parametr  $k$  określa liczbę wierzchołków otworu, zaś tablice  $knots$  i  $domain\_cp$  zawierają odpowiednio ciągi węzłów i punkty kontrolne dziedziny. Procedura `gh_CreateDomainf` tworzy strukturę danych, która zawiera reprezentację dziedziny  $\Omega$  parametryzacji powierzchni wypełniającej, a także sposobu jej podziału na fragmenty ( $k$  krzywoliniowych czworokątów) i bazy przestrzeni  $V$ . Struktura ta dalej będzie nazywana **rekordem dziedziny**.

Po utworzeniu rekordu dziedziny (przed wywołaniem `g2h_ComputeBasisf`), można wywołać procedurę `g2h_SetOptionProcf` w celu użycia innych niż domyślne opcji w konstrukcji. Procedura `g2h_ComputeBasisf` oblicza reprezentację funkcji, z których składa się baza przestrzeni  $V$ , co zajmuje raczej mało czasu.

Procedura `g2h_DecomposeMatrixf` oblicza współczynniki macierzy  $A$  i  $B$ , które występują w układzie równań (9.2). Współczynniki te są wartościami formy dwuliniowej w przestrzeni  $V$  dla par funkcji bazowych. Wektor  $b$  składa się z punktów kontrolnych powierzchni (będą one określone w drugim etapie konstrukcji), a nie-



wiadomy wektor  $\mathbf{a}$  składa się z pozostałych współczynników używanej w konstrukcji reprezentacji powierzchni wypełniającej. Macierz  $A$ , która jest symetryczna i dodatnio określona, jest następnie rozkładana metodą Choleskiego na czynniki trójkątne:  $A = LL^T$ , które będą potrzebne podczas rozwiązywania układu. Obliczenie współczynników macierzy  $A$  i  $B$  jest najbardziej czasochłonnym krokiem konstrukcji — dla  $k = 8$  procesor Pentium IV z zegarem 1.8GHz może na to zużyć ok. 0.15s.

Wykonanie drugiego etapu konstrukcji polega na wykonaniu instrukcji

```
if ( !g2h_FillHolef ( domain, d, surf_cp, acoeff, output ) )
    exit ( 1 );
```

Parametr `domain` jest wskaźnikiem rekordu dziedziny, dla którego pierwszy etap konstrukcji został (z sukcesem) zakończony. Parametr `d` określa wymiar przestrzeni, w której leży powierzchnia, tablica `surf_cp` zawiera punkty kontrolne powierzchni, a parametr `output` jest wskaźnikiem procedury, która zostanie wywołana  $k$ -krotnie. Za każdym razem jej parametry będą opisywać reprezentację Béziera kolejnego płata będącego częścią powierzchni wypełniającej otwór.

Parametr `acoeff` jest tablicą, do której ma być wstawione rozwiązanie układu równań (9.2). Może ono być potrzebne, jeśli kogoś interesuje wartość funkcjonału  $F$  dla skonstruowanej powierzchni (ściślej, suma wartości dla funkcji opisujących współrzędne powierzchni). Parametr `teh` może mieć też wartość `NULL` i wtedy jest ignorowany.

### 9.3.2 Konstrukcja nieliniowa

Aby otrzymać powierzchnię wypełniającą, dla której funkcjonał  $F_d$  przyjmuje minimalną wartość, należy utworzyć reprezentację dziedziny, skonstruować bazę i obliczyć współczynniki macierzy  $A$  i  $B$ , a następnie zamiast procedury `g2h_FillHolef` wywołać procedurę `g2h_NLFillHolef`.

Obliczenia w tej konstrukcji są znacznie bardziej pracochłonne (do paru sekund), a ponadto wykonalność tej konstrukcji zależy od powierzchni z otworem. Jeśli powierzchnia ta nie jest dostatecznie płaska lub jest zdegenerowana, to konstrukcja może zakończyć się niepowodzeniem.

### 9.3.3 Konstrukcje z przestrzenią rozszerzoną

Płaty wypełniające otwór skonstruowane w sposób opisany wyżej są określone jako dwupiętne płaty Coonsa, których krzywe opisujące brzegi i pochodne w kierunku poprzecznym do brzegu mają stopień 9 lub mniejszy. Reprezentacja Coonsa używana w konstrukcji jest poddawana konwersji do postaci Béziera. Ponieważ przestrzeń wielomianów dwóch zmiennych stopnia  $(9,9)$  ma wymiar 100, a wielomiany

stopnia  $(9, 9)$  mające dwupiętną reprezentację Coonsa tworzą podprzestrzeń o wymiarze 84, więc istnieje możliwość rozszerzenia przestrzeni funkcji, za pomocą których są reprezentowane powierzchnie wypełniające, do przestrzeni, której wymiar jest o 16k większy. Powierzchnie, wyznaczone za pomocą minimalizacji funkcjonału  $F$  w rozszerzonej przestrzeni, mogą mieć lepszy kształt (i często mają).

Aby skorzystać z tej możliwości, po utworzeniu rekordu dziedziny za pomocą procedury `gh_CreateDomainf` i ewentualnym zarejestrowaniu procedury wprowadzania opcji, należy utworzyć bazę przestrzeni podstawowej, wywołując jak poprzednio `g2h_ComputeBasisf` (konstrukcja dodatkowych funkcji, które wchodzą w skład bazy rozszerzonej przestrzeni nie wymaga żadnych dodatkowych obliczeń). Następnie *zamiast* procedury `g2h_DecomposeMatrixf` należy wywołać procedurę `g2h_DecomposeExtMatrixf`, która obliczy macierze  $A$  i  $B$  odpowiednio powiększonego układu równań (9.2) i rozłoży macierz  $A$  na czynniki trójkątne.

Drugi etap konstrukcji przy użyciu przestrzeni rozszerzonej wykonuje procedura `g2h_ExtFillHolef`, którą należy wywołać *zamiast* procedury `g2h_FillHolef`. Parametry tych procedur są identyczne.

Dane wykorzystywane w obu konstrukcjach są tworzone i przechowywane w rekordzie dziedziny niezależnie, w związku z czym można najpierw utworzyć i rozłożyć macierze układów dla obu tych konstrukcji, a potem wykonać je w dowolnej kolejności i porównać wyniki. Czas obliczeń dla przestrzeni rozszerzonej jest dłuższy, ale różnica jest praktycznie niezauważalna. Obliczenia dla przestrzeni rozszerzonej wymagają za to więcej pamięci — zarówno na wyniki (współczynniki macierzy, które przechowuje się w obszarach rezerwowanych przez `malloc`), jak i na tablice pomocnicze (w puli pamięci „podręcznej”, obsługiwanej przez procedury opisane w p. 2.3). W wersji podwójnej precyzji dla  $k = 8$  tablice pomocnicze mogą zajmować około 2MB.

Aby skonstruować powierzchnię minimalną funkcjonału  $F_d$  przy użyciu przestrzeni rozszerzonej, należy wywołać procedurę `g2h_NLExtFillHolef`. Pamięć „podręczna” potrzebna w tej konstrukcji może mieć wielkość do ok. 8MB.

#### 9.3.4 Konstrukcje z więzami

W celu skonstruowania powierzchni z więzami należy skonstruować reprezentację dziedziny (przy użyciu `gh_CreateDomainf`), skonstruować bazę, a następnie wprowadzić macierz  $C$  układu równań więzów i wywołać procedurę konstrukcji powierzchni z więzami.

Dla **przestrzeni podstawowej**, do wprowadzania macierzy układu równań więzów (9.4) służy procedura `g2h_SetConstraintMatrixf`. Powierzchnię minimalną funkcjonału  $F_c$  z więzami konstruuje procedura `g2h_FillHoleConstrf`. Powierzchnię minimalną funkcjonału  $F_d$  z więzami o tej postaci konstruuje procedura `g2h_NLFillHoleConstrf`.

Do wprowadzenia macierzy układu równań więzów o postaci (9.5) służy pro-

cedura `g2h_SetAltConstraintMatrixf`. Powierzchnię minimalną funkcjonału  $F_c$  z więzami o tej postaci konstruuje procedura `g2h_FillHoleAltConstrf`, a powierzchnię minimalną funkcjonału  $F_d$  procedura `g2h_NLFillHoleAltConstrf`.

Dla przestrzeni rozszerzonej do wprowadzenia macierzy układu równań (9.4) służy procedura `g2h_SetExtConstraintMatrixf`. Powierzchnia minimalna funkcjonału  $F_c$  jest konstruowana przez procedurę `g2h_ExtFillHoleConstrf`, a powierzchnia minimalna  $F_d$  przez `g2h_NLExtFillHoleConstrf`.

Macierz układu (9.5) dla przestrzeni rozszerzonej wprowadza się za pomocą procedury `g2h_SetExtAltConstraintMatrixf`. Powierzchnie minimalne funkcjonałów  $F_c$  i  $F_d$  otrzymuje się za pomocą procedur `g2h_ExtFillHoleAltConstrf` i `g2h_NLExtFillHoleAltConstrf`.

Macierz każdego z czterech rodzajów więzów (tj. o postaci (9.4) i (9.5) dla przestrzeni podstawowej i rozszerzonej) może być określona niezależnie od pozostałych. Aby zmienić macierz układu więzów, wystarczy ponownie wywołać odpowiednią procedurę spośród wymienionych wyżej.

## 9.4 Procedury podstawowe

```
#define G2H_FINALDEG 9
#define GH_MAX_K 16
```

Powyższe dwie stałe symboliczne określają stopień płatów wypełniających otwór i największą dopuszczalną liczbę wierzchołków wielokątnego otworu.

Stałych tych nie można po prostu zmienić — stopień 9 jest rezultatem zastosowania odpowiedniego schematu interpolacyjnego. Procedury biblioteczne mogą zrealizować też schemat interpolacyjny, którego wynikiem są płyty stopnia 10. W tym celu trzeba usunąć definicję symbolu `G2H_FINALDEG9` w pliku nagłówkowym i skompilować procedury.

Dziedzina parametryzacji powierzchni wypełniającej  $k$ -kątny otwór jest dzielona na  $k$  części. Do reprezentowania zbiorów tych części są używane liczby całkowite krótkie, czyli zmienne o długości 16 bitów. Aby wypełniać otwory więcej niż szesnastokątne, trzeba przerobić odpowiedni fragment procedur, tak, aby używały one np. słów 32-bitowych, co umożliwi wypełnianie otworów trzydziestodwukątnych.

```
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;
```

Typ struktury GHoleDomainf opisuje rekord dziedziny, tj. obiekt reprezentujący dane potrzebne do skonstruowania powierzchni wypełniającej otwór. W programie deklaruje się zmienną wskaźnikową do rekordu tego typu, ponieważ za ich tworzenie i poprawność danych odpowiedzialne są procedury biblioteczne.

Pole hole\_k opisuje liczbę wierzchołków wielokątnego otworu do wypełnienia (od 3 do 16).

Pole hole\_knots jest wskaźnikiem do tablicy 11k liczb, które są węzłami w reprezentacji powierzchni z otworem.

Pole domain\_cp jest wskaźnikiem do tablicy  $12k + 1$  punktów statki kontrolnej reprezentującej dziedzinę parametryzacji powierzchni wypełniającej.

Pola privateG, privateG1, SprivateG1, privateG2 i SprivateG2 są wskaźnikami rekordów (ich budowa i zawartość są niewidoczne dla aplikacji) z wszystkimi innymi danymi potrzebnymi w poszczególnych konstrukcjach powierzchni wypełniających.

Pole error\_code służy do przechowywania informacji o sukcesie, albo o przyczynie niepowodzenia obliczeń.

```
GHoleDomainf* gh_CreateDomainf ( int hole_k,
                                float *hole_knots,
                                point2f *domain_cp );
void gh_DestroyDomainf ( GHoleDomainf *domain );
```

Procedura gh\_CreateDomainf tworzy obiekt typu GHoleDomainf, który reprezentuje dziedzinę powierzchni wypełniającej i zwraca wskaźnik do tego obiektu. Bloki pamięci dla tego obiektu i wszystkich danych wskazywanych przez zawarte w nim wskaźniki są rezerwowane za pomocą procedury malloc.

Parameter hole\_k określa liczbę k wierzchołków dziedziny otworu (musi być od 3 do 16).

Parameter hole\_knots jest tablicą z 11k liczbami — węzłami reprezentacji powierzchni z otworem i dziedziny.

Parameter domain\_cp jest tablicą z  $12k + 1$  punktami kontrolnymi reprezentacji

dziedziny. Zawartość tych dwóch tablic jest kopiowana do pamięci rezerwowanej przez procedurę `gh_CreateDomainf`.

Jeśli nie można zarezerwować pamięci lub w danych został wykryty błąd, to wartością procedury jest `NULL`.

Obiekt natychmiast po utworzeniu nie jest gotów do wypełniania powierzchni — ta procedura nie konstruuje bazy. Obliczenia przygotowawcze są nieco czasochłonne, dlatego w aplikacji może być wygodne ich oddzielenie od utworzenia obiektu.

Procedura `gh_DestroyDomainf` zwalnia (za pomocą procedury `free`) pamięć zajmowaną przez reprezentację dziedziny (w szczególności wszystkie bloki pamięci zarezerwowane podczas obliczeń wykonanych podczas przetwarzania tej reprezentacji).

```
void g2h_SetOptionProc ( GHoleDomainf *domain,
    int (*OptionProc)( GHoleDomainf *domain, int query, int qn,
        int *ndata, int **idata, float **fdata ) );
```

Procedura `g2h_SetOptionProc` rejestruje dostarczoną przez aplikację procedurę, której zadaniem jest określenie opcji dla konstrukcji i przekazanie odpowiednich danych. Jeśli po utworzeniu reprezentacji dziedziny nie zarejestrujemy takiej procedury, to używana jest procedura domyślna (która na każde pytanie o opcję daje odpowiedź domyślną).

Ten sposób określania opcji został zrealizowany w celu ustalenia listy parametrów procedur bibliotecznych podczas opracowywania konstrukcji. Korzyść z niego jest taka, że aplikacja, która nie używa opcji innych niż standardowe, nie musi wywoływać procedur bibliotecznych z parametrami pozbawionymi dla tej aplikacji znaczenia.

Zasady określania opcji są opisane w p. 9.5.

```
boolean g2h_ComputeBasisf ( GHoleDomainf *domain );
```

Procedura `g2h_ComputeBasisf` konstruuje funkcje bazowe, przy użyciu których można będzie wypełniać otwory w powierzchniach. Parametr procedury jest wskaźnikiem obiektu utworzonego przez procedurę `gh_CreateDomainf`. Wartość procedury `true` oznacza sukces, a `false` oznacza niepowodzenie obliczenia.

Jeśli dla dziedziny przekazanej jako parametr została zarejestrowana procedura określania opcji, to będzie ona wywołana pewną liczbę razy. Procedura ta ma wpływ na wynik obliczeń (tj. postać funkcji bazowych), czyli także na powierzchnie wypełniające otrzymane przy użyciu tej bazy.

Procedura `g2h_ComputeBasisf` powinna być wywołana tylko raz dla reprezentacji dziedziny utworzonej przez procedurę `gh_CreateDomainf`. Jeśli trzeba dla tej samej dziedziny skonstruować więcej niż jedną bazę, np. przy użyciu różnych opcji, to trzeba za każdym razem zlikwidować reprezentację dziedziny (wywołując `gh_DestroyDomainf`) i utworzyć ją od nowa.

Obliczenie realizowane przez tę procedurę zajmuje umiarkowaną ilość czasu, dlatego można je wykonać „na poczekaniu” podczas obsługi komunikatu (związane z tym opóźnienie nie powinno być zauważalne dla użytkownika programu interakcyjnego).

```
boolean g2h_ComputeFormMatrixf ( GHoleDomainf *domain );
boolean g2h_DecomposeMatrixf ( GHoleDomainf *domain );
```

Procedura `g2h_ComputeFormMatrixf` oblicza współczynniki macierzy układu równań rozwiązywanego podczas konstruowania powierzchni wypełniającej otwór, **przy użyciu przestrzeni podstawowej**. Parametrem procedury jest reprezentacja dziedziny utworzona przez procedurę `gh_CreateDomainf`, dla której procedura `g2h_ComputeBasisf` skonstruowała z sukcesem reprezentację funkcji bazowych.

Procedura `g2h_DecomposeMatrixf` rozkłada (metodą Choleskiego) macierz układu, skonstruowaną przez procedurę `g2h_ComputeFormMatrixf`. Jeśli współczynniki nie zostały obliczone, to procedura `g2h_DecomposeMatrixf` najpierw oblicza je, wywołując `g2h_ComputeFormMatrixf`.

Wartością obu procedur jest `true` jeśli obliczenie przebiegło poprawnie, albo `false` w przeciwnym razie.

```
boolean g2h_FillHolef ( GHoleDomainf *domain,
                      int spdimen, const float *hole_cp, float *acoeff,
                      void (*outpatch) ( int n, int m, const float *cp ) );
```

Procedura `g2h_FillHolef` konstruuje powierzchnię wypełniającą wielokątny otwór, **w oparciu o przestrzeń podstawową**.

Parametr `domain` wskazuje reprezentację dziedziny utworzoną przez procedurę `gh_CreateDomainf`, dla której procedura `g2h_ComputeBasisf` skonstruowała (z sukcesem) reprezentację funkcji bazowych. Liczba  $k$  wierzchołków otworu i ciągi węzłów należące do reprezentacji powierzchni zostały określone przy wywołaniu procedury `gh_CreateDomainf`.

Parametr `spdimen` określa wymiar  $d$  przestrzeni, w której leży powierzchnia. Dla powierzchni wielomianowej w  $\mathbb{R}^3$  parametr ten będzie miał wartość 3. Dla powierzchni wielomianowej w  $\mathbb{R}^4$ , która jest jednorodną reprezentacją powierzchni wymiernej w  $\mathbb{R}^3$  trzeba podać 4.

Parametr `hole_cp` jest tablicą, która zawiera  $(12k + 1)d$  liczb zmiennopozycyjnych, będących współrzędnymi  $12k + 1$  punktów kontrolnych powierzchni.

Parametr `acoeff` może mieć wartość `NULL` (i wtedy jest ignorowany), lub wskazywać tablicę, do której procedura wstawi rozwiązanie układu równań (9.2). Tablica ta musi mieć długość  $nd$ , gdzie  $d$  jest wymiarem przestrzeni, w której leży powierzchnia (tj. wartością parametru `spdimen`), a  $n$  jest wymiarem przestrzeni podstawowej — można go otrzymać za pomocą procedury `g2h_V0SpaceDimf`.

Parametr `outpatch` jest wskaźnikiem procedury (dostarczonej przez aplikację), która będzie wywołana  $k$  razy, w celu wyprowadzenia punktów kontrolnych  $k$  płatów Béziera stopnia  $(9, 9)$  wypełniających otwór.

```
boolean g2h_ComputeExtFormMatrixf ( GHoleDomainf *domain );
boolean g2h_DecomposeExtMatrixf ( GHoleDomainf *domain );
```

Procedura `g2h_ComputeExtFormMatrixf` oblicza współczynniki macierzy układu równań rozwiązywanego podczas konstruowania powierzchni wypełniającej otwór, przy użyciu przestrzeni rozszerzonej. Parametrem procedury jest reprezentacja dziedziny utworzona przez procedurę `gh_CreateDomainf`, dla której procedura `g2h_ComputeBasisf` skonstruowała z sukcesem reprezentację funkcji bazowych.

Procedura `g2h_DecomposeExtMatrixf` rozkłada (metodą Choleskiego) macierz układu, skonstruowaną przez procedurę `g2h_ComputeExtFormMatrixf`. Jeśli współczynniki nie zostały obliczone, to procedura `g2h_DecomposeExtMatrixf` najpierw oblicza je, wywołując `g2h_ComputeExtFormMatrixf`.

Wartością obu procedur jest `true` jeśli obliczenie przebiegło poprawnie, albo `false` w przeciwnym razie.

```
boolean g2h_ExtFillHolef ( GHoleDomainf *domain,
                          int spdimen, const float *hole_cp, float *acoeff,
                          void (*outpatch) ( int n, int m, const float *cp ) );
```

Procedura `g2h_ExtFillHolef` konstruuje powierzchnię wypełniającą wielokątny otwór, w oparciu o przestrzeń rozszerzoną.

Parametr `domain` wskazuje reprezentację dziedziny utworzoną przez procedurę `gh_CreateDomainf`, dla której procedura `g2h_ComputeBasisf` skonstruowała (z sukcesem) reprezentację funkcji bazowych. Liczba  $k$  wierzchołków otworu i ciągi węzłów należące do reprezentacji powierzchni zostały określone przy wywołaniu procedury `gh_CreateDomainf`.

Parametr `spdimen` określa wymiar  $d$  przestrzeni, w której leży powierzchnia. Dla powierzchni wielomianowej w  $\mathbb{R}^3$  parametr ten będzie miał wartość 3. Dla powierzchni wielomianowej w  $\mathbb{R}^4$ , która jest jednorodną reprezentacją powierzchni wymiernej w  $\mathbb{R}^3$  trzeba podać 4.

Parametr `hole_cp` jest tablicą, która zawiera  $(12k + 1)d$  liczb zmiennopozycyjnych, będących współrzędnymi  $12k + 1$  punktów kontrolnych powierzchni.

Parametr `acoeff` może mieć wartość `NULL` (i wtedy jest ignorowany), lub wskazywać tablicę, do której procedura wstawi rozwiązanie układu równań (9.2). Tablica ta musi mieć długość  $nd$ , gdzie  $d$  jest wymiarem przestrzeni, w której leży powierzchnia (tj. wartością parametru `spdimen`), a  $n$  jest wymiarem przestrzeni rozszerzonej — można go otrzymać za pomocą procedury `g2h_ExtV0SpaceDimf`.

Parametr `outpatch` jest wskaźnikiem procedury (dostarczonej przez aplikację), która będzie wywołana  $k$  razy, w celu wyprowadzenia punktów kontrolnych  $k$  płatów Béziera stopnia  $(9, 9)$  wypełniających otwór.

```
int g2h_GetErrorCodef ( GHoleDomainf *domain,
                        char **ErrorString );
```

Procedura `g2h_GetErrorCodef` może być wywołana w razie niepowodzenia któregoś etapu konstrukcji, w celu ustalenia przyczyny. Jej wartością jest numer (kod) błędu. Jeśli parametr `ErrorString` jest różny od `NULL`, to zmienna `*ErrorString` po powrocie z procedury wskazuje napis, który jest opisem błędu (po angielsku).

## 9.5 Wprowadzanie opcji

Procedura `g2h_SetOptionProc` opisana w poprzednim punkcie służy do zarejestrowania procedury (należącej do aplikacji), która ma „odpowiadać na pytania” na temat opcji. Procedura ta ma mieć następujący nagłówek (nazwy procedury i parametrów mogą oczywiście być inne):

```
int SetOptionf ( GHoleDomainf *domain, int query, int qn,
                 int *ndata, int **idata, float **fdata );
```

Podczas konstruowania bazy procedura ta będzie wywołana pewną liczbę razy. Pierwszy jej parametr jest wskaźnikiem reprezentacji dziedziny wypełnianego otworu. Drugi parametr (`query`) jest numerem opcji, którą procedura ma określić. Parametr `qn` jest dodatkowym numerem, który może być potrzebny w opcjach dodanych do przyszłych wersji biblioteki, a na razie można go zignorować.

Wartość procedury jest interpretowana jako odpowiedź na pytanie o opcję, którą należy zastosować. Możliwe numery opcji (tj. wartości parametru `query`) i odpowiedzi są stałymi symbolicznymi (o nazwach zaczynających się odpowiednio od `G2HQUERY_` i `G2H_`) wymienionymi niżej. Lista ta może się zmienić w przyszłych wersjach biblioteki `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

```

Po każdym wywołaniu procedura wprowadzania opcji może zwrócić wartość G2H\_DEFAULT. W szczególności taka powinna być wartość zwracana dla każdej opcji (określonej przez parametr query), której procedura „nie rozumie”. Dzięki temu aplikacja ma szansę działać poprawnie po skompilowaniu z nowszą wersją biblioteki.

Jeśli parametr query ma wartość G2HQUERY\_CENTRAL\_POINT, to odpowiedź (tj. wartość procedury) G2H\_DEFAULT spowoduje przyjęcie za punkt środkowy dziedziny (tj. wspólny narożnik obszarów, na które dziedzina zostanie podzielona) środka ciężkości środków boków otworu (to jest konstrukcja opisana w artykułach). Jeśli procedura zwróci wartość G2H\_CENTRAL\_POINT\_GIVEN, to zmienna \*ndata musi otrzymać wartość 2, a zmiennej \*fdata należy przypisać wartość wskazującą tablicę z dwiema liczbami zmiennopozycyjnymi, które są współrzędnymi punktu środkowego podanego przez aplikację.

Jeśli wartością parametru query jest G2HQUERY\_CENTRAL\_DERIVATIVES1 i procedura zwróci wartość G2H\_DEFAULT, to wektory pochodnych pierwszego rzędu krzywych podziału dziedziny w punkcie środkowym i wektory pochodnych poprzecznych płatów pomocniczych dziedziny zostaną przyjęte zgodnie z opisem w artykułach. Wartość G2H\_CENTRAL\_DERIVATIVES\_ALT spowoduje przyjęcie wektorów pochodnych krzywych tak samo, zaś wektory pochodnych poprzecznych będą do nich prostopadłe. Wartość G2H\_CENTRAL\_DERIVATIVES\_GIVEN oznacza, że aplikacja podaje wektory pochodnych krzywych. Zmienna \*ndata ma mieć wartość 2k (dla k-kątnego otworu), a tablica wskazywana przez zmienną \*fdata ma zawierać 2k liczb zmiennopozycyjnych. Każde kolejne dwie z tych liczb są współrzędnymi wektora pochodnej kolejnej krzywej.

Parametr query o wartości G2HQUERY\_DOMAIN\_CURVES oznacza pytanie o pochodne krzywych podziału dziedziny rzędu wyższego niż 1 w punkcie środkowym. W odpowiedzi na to pytanie należy zwrócić wartość G2H\_DEFAULT, co spowoduje przyjęcie zerowych pochodnych rzędu 2, 3 i 4. Inne opcje w tym przypadku są na razie niedopracowane i mogą dać niepoprawne skutki.

Jeśli parametr query ma wartość G2HQUERY\_BASIS, to odpowiedź G2H\_DEFAULT oznacza użycie wszystkich stopni swobody wyboru pochodnych cząstkowych płatów wypełniających otwór w ich punkcie wspólnym (liczba ta jest wymiarem przestrzeni podstawowej). Zależnie od liczby wierzchołków otworu i podziału dziedziny, liczba tych stopni jest od 16 do 30 (dla otworów trój- do ośmiokątnych). Odpowiedź G2H\_USE\_RESTRICTED\_BASIS powoduje ograniczenie liczby stopni swobody do 15 (jest to wymiar przestrzeni wielomianów dwóch zmiennych stopnia co najwyżej 4).

## 9.6 Nakładanie więzów

```
int g2h_VOSpaceDimf ( GHoleDomainf *domain );
int g2h_ExtVOSpaceDimf ( GHoleDomainf *domain );
```

Procedury g2h\_VOSpaceDimf i g1h\_ExtVOSpaceDimf obliczają odpowiednio wymiary przestrzeni podstawowej i rozszerzonej, używanych w konstrukcjach powierzchni wypełniających otwór. Parametr domain jest wskaźnikiem rekordu dziedziny, dla której baza przestrzeni podstawowej została pomyślnie skonstruowana.

```
boolean g2h_GetBPDerivativesf ( GHoleDomainf *domain,
                                int cno, float *val );
```

Procedura g1h\_GetBPDerivativesf oblicza wartości funkcji bazowych (bazy przestrzeni podstawowej) w punkcie środkowym i wartości pochodnych rzędu 1, ..., 4 krzywych brzegowych płatów funkcji bazowych. Numer krzywej jest określony przez wartość parametru cno (musi być od 0 do  $k-1$ ). Obliczone wartości są wpisywane do tablicy val, o długości  $5n$ , gdzie  $n$  jest wymiarem przestrzeni podstawowej. Kolejne piątki liczb wpisywanych do tej tablicy odpowiadają kolejnym funkcjom bazowym.

```
boolean g2h_GetBFuncPatchf ( GHoleDomainf *domain,
                              int fn, int pn, float *bp );
```

Procedura g1h\_GetBFuncPatchf oblicza i umieszcza w tablicy bp współczynniki  $j$ -tego płata  $i$ -tej funkcji bazowej. Wartość parametru fn określa numer  $i \in \{0, \dots, n-1\}$  funkcji bazowej, a pn określa numer  $j \in \{0, \dots, k-1\}$  płata tej funkcji. Procedura ta może się przydać, jeśli więzy nakładane na powierzchnię wypełniającą otwór nie są warunkami interpolacyjnymi w punkcie środkowym powierzchni.

Płaty funkcji bazowych są (skalarnymi) wielomianami dwóch zmiennych, stopnia G2H\_FINALDEG ze względu na każdą zmienną. Współczynniki reprezentują te płaty w tensorowej bazie Bernsteina.

```
boolean g2h_SetConstraintMatrixf ( GHoleDomainf *domain,
                                   int nconstr, const float *cmat );
```

Procedura g2h\_SetConstraintMatrixf łączy z dziedziną powierzchni wypełniającej macierz układu równań liniowych opisujących więzy, które mają być

nałożone na powierzchnię. Parametr `nconstr` jest liczbą więzów (tj. równań), czyli liczbą wierszy macierzy. Liczba kolumn jest równa wymiarowi przestrzeni podstawowej. Kolejne wiersze są podane w tablicy `cmat`. Wiersze te muszą być liniowo niezależne.

Wartość `true` procedury oznacza powodzenie, zaś `false` oznacza, że podana macierz nie jest wierszowo-regularna.

```
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 ) );
```

Procedura `g1h_FillHoleConstrf` konstruuje powierzchnię wypełniającą otwór, z nałożonymi więzami, przy użyciu przestrzeni podstawowej. Przed wywołaniem tej procedury należy z dziedziną powierzchni związać macierz układu równań opisującego więzy (to określa m.in. liczbę nałożonych więzów). Parametry `domain`, `spdimen`, `hole_cp`, `acoeff` i `outpatch` mają takie samo znaczenie jak w procedurze `g2h_FillHolef`. Parametr `nconstrf` określa liczbę więzów (musi ona zgadzać się z liczbą podaną w wywołaniu `g2h_SetConstraintMatrixf`. Tablica `constr` zawiera macierz prawej strony równań więzów — `nconstr` wierszy po `spdimen` liczb.

Zmiana więzów (zarówno macierzy jak i prawej strony) nie wymaga tworzenia rekordu dziedziny od początku. Aby zmienić więzy, wystarczy ponownie wywołać `g2h_SetConstraintMatrixf` i `g1h_FillHoleConstrf`.

```
boolean g2h_SetAltConstraintMatrixf ( GHoleDomainf *domain,
                                      int spdimen,
                                      int nconstr, const float *cmat );
```

Procedura `g2h_SetAltConstraintMatrixf` służy do wprowadzenia macierzy  $C$  układu równań więzów o postaci (9.5) dla konstrukcji z przestrzenią podstawową. Macierz ta ma wymiary  $nd \times w$ , gdzie  $n$  jest wymiarem przestrzeni (można go otrzymać, wywołując procedurę `g2h_V0SpaceDimf`),  $d$  jest wymiarem przestrzeni, w której znajduje się powierzchnia (czyli np. 3),  $w$  jest liczbą więzów.

Parametr `spdimen` określa wymiar  $d$ , parametr `nconstr` określa liczbę więzów. Współczynniki macierzy  $C$  należy podać w tablicy `cmat`. Podziałka tej tablicy jest równa długości wiersza, tj.  $nd$ . Macierz  $C$  musi być wierszowo regularna.

Wartość `true` procedury `g2h_SetAltConstraintMatrixf` oznacza, że macierz jest wierszowo regularna. Wartość `false` oznacza, że nie (tj. procedura numeryczna uznała, że wiersze są liniowo zależne) i z taką macierzą nie można wykonać konstrukcji.

```
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 ) );
```

Procedura `g2h_FillHoleAltConstrf` konstruuje powierzchnię wypełniającą, która jest punktem minimalnym funkcjonału  $F_c$  w zbiorze powierzchni reprezentowalnych przy użyciu przestrzeni podstawowej i spełniających równania więzów (9.5). Macierz  $C$  tego układu musi być wprowadzona wcześniej, przez wywołanie procedury `g2h_SetAltConstraintMatrixf`.

Parametr `spdimen` określa wymiar  $d$  przestrzeni, w której znajduje się powierzchnia. Tablica `hole_cp` zawiera punkty kontrolne tej powierzchni. Parametr `naconstr` określa liczbę więzów  $w$ . W tablicy `constr` należy podać współczynniki wektora prawej strony układu równań więzów ( $w$  liczb). Liczby  $d$  i  $w$  muszą się zgadzać z wartościami odpowiednich parametrów poprzedzającego wywołania procedury `g2h_SetAltConstraintMatrixf`.

Jeśli parametr `acoeff` jest różny od `NULL`, to ma wskazywać tablicę, do której procedura wpisze obliczone jako wynik optymalizacji wektory  $\mathbf{a}_0, \dots, \mathbf{a}_{n-1}$ . Procedura wskazywana przez parametr `output` zostanie wywołana  $k$  razy w celu wyprowadzenia wyniku konstrukcji, tj. płatów Béziera wypełniających otwór.

```
boolean g2h_SetExtConstraintMatrixf ( GHoleDomainf *domain,
                                      int nconstr, const float *cmat );
```

Procedura `g2h_SetExtConstraintMatrixf` wiązuje z dziedziną powierzchni wypełniającej macierz  $C$  układu równań liniowych (9.4) opisujących więzy, które mają być nałożone na powierzchnię reprezentowaną przy użyciu przestrzeni rozszerzonej. Parametr `nconstr` jest liczbą więzów (tj. równań), czyli liczbą wierszy macierzy. Liczba kolumn jest równa wymiarowi przestrzeni rozszerzonej. Kolejne wiersze są podane w tablicy `cmat`. Wiersze te muszą być liniowo niezależne. Długość wiersza  $n$  jest równa wymiarowi przestrzeni  $V_0$ , który można poznać, wywołując procedurę `g2h_ExtV0SpaceDimf`.

Wartość `true` procedury oznacza powodzenie, zaś `false` oznacza, że podana macierz nie jest wierszowo-regularna.

Pierwsze  $16k$  współczynników odpowiada funkcjom bazowym, które w punkcie środkowym dziedziny mają wartości i pochodne do czwartego rzędu włącznie równe 0. Ostatnie  $n'$  współczynników w wierszu to wartości odpowiedniego funkcjonału liniowego dla funkcji bazowych przestrzeni podstawowej. Jeśli więzy są określone w celu skonstruowania powierzchni minimalnej funkcjonału  $F_d$  (za pomocą procedury `g2h_NLExtFillHoleConstrf`), to pierwsze  $16k$  współczynników w każdym wierszu muszą być zerami. Ograniczenie to wynika z zastosowanej w tej konstrukcji metody numerycznej.

```
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 ) );
```

Procedura `g1h_ExtFillHoleConstrf` konstruuje powierzchnię wypełniającą otwór, z nałożonymi więzami, przy użyciu przestrzeni rozszerzonej. Przed wywołaniem tej procedury należy z dziedziną powierzchni związać macierz układu równań opisującego więzy (to określa m.in. liczbę nałożonych więzów). Parametry `domain`, `spdimen`, `hole_cp`, `acoeff` i `outpatch` mają takie samo znaczenie jak w procedurze `g2h_ExtFillHolef`. Parametr `nconstrf` określa liczbę więzów (musi ona zgadzać się z liczbą podaną w wywołaniu `g2h_SetExtConstraintMatrixf`). Tablica `constr` zawiera macierz prawej strony równań więzów — `nconstr` wierszy po `spdimen` liczb.

Zmiana więzów (zarówno macierzy jak i prawej strony) nie wymaga tworzenia rekordu dziedziny od początku. Aby zmienić więzy, wystarczy ponownie wywołać `g2h_SetExtConstraintMatrixf` i `g1h_ExtFillHoleConstrf`.

```
boolean g2h_SetExtAltConstraintMatrixf ( GHoleDomainf *domain,
                                         int spdimen,
                                         int naconstr, const float *acmat );
```

Zadaniem procedury `g2h_SetExtAltConstraintMatrixf` jest wprowadzenie macierzy  $C$  układu równań więzów (9.5) dla konstrukcji powierzchni wypełniającej reprezentowanej za pomocą przestrzeni rozszerzonej. Wymiary tej macierzy to  $nd \times w$ , gdzie  $n$  jest wymiarem przestrzeni rozszerzonej,  $d$  jest wymiarem przestrzeni, w której jest powierzchnia (np. 3), a liczba wierszy  $w$  jest liczbą więzów.

Parametry: `spdimen` — wymiar  $d$ , `naconstr` — liczba więzów  $w$ , `acmat` — tablica współczynników macierzy  $C$ . Tablica ta ma podziałkę równą długości wiersza, tj.  $nd$ .

Tablicę  $C$  można podzielić na bloki  $C_0, \dots, C_{d-1}$ , o wymiarach  $w \times d$ . Jeśli więzy są określane na potrzeby konstrukcji opartej na minimalizacji funkcjonału  $F_d$ , to musi być  $d = 3$ . W każdym wierszu każdego bloku pierwsze 16k współczynników muszą być zerami, a ponadto macierz  $C$  musi być wierszowo-regularna. Ograniczenie to wynika z zastosowanej w konstrukcji metody numerycznej.

Wartość `true` procedury oznacza akceptację macierzy, zaś wartość `false` oznacza, że na podstawie rachunku numerycznego macierz została przez procedurę uznana za wierszowo-nieregularną.

```
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 ) );
```



```
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 Procedury konstrukcji nieliniowych

Opisane w tym punkcie procedury dokonują konstrukcji powierzchni wypełniających przez minimalizację funkcjonału  $F_d$ . Wykonalność tych konstrukcji zależy w znacznie większym stopniu od danej powierzchni z otworem. Ponadto są one bardziej czasochłonne. Powierzchnia z otworem do wypełnienia przy użyciu tych procedur musi leżeć w przestrzeni  $\mathbb{R}^3$  (a więc nie może to być np. jednorodna reprezentacja powierzchni wymiernej).

```
boolean g2h_ComputeNLNormalf ( GHoleDomainf *domain,
    const point3f *hole_cp,
    vector3f *anv );
```

Procedura `g2h_ComputeNLNormalf` konstruuje wektor jednej z osi układu współrzędnych, w którym powierzchnia z otworem i konstruowana powierzchnia wypełniająca będą przedstawione jako wykresy funkcji skalarnych. Parametry wejściowe tej procedury to `domain` — wskaźnik reprezentacji dziedziny i `hole_cp` — tablica punktów kontrolnych powierzchni. Parametr `anv` wskazuje zmienną, do której procedura wpisuje wynik.

Wartość `true` procedury oznacza sukces, a `false` — jego brak (jeśli powierzchnia określona przez podane punkty kontrolne nie jest dostatecznie płaska). Procedura `g2h_ComputeNLNormalf` w zasadzie nie jest przeznaczona do wywoływania przez aplikację.

```
boolean g2h_NLFillHolef ( GHoleDomainf *domain,
    const point3f *hole_cp, float *acoeff,
    void (*outpatch) ( int n, int m, const point3f *cp ) );
```

Procedura `g2h_NLFillHolef` dokonuje konstrukcji powierzchni wypełniającej, będącej punktem minimalnym funkcjonału  $F_d$  w przestrzeni podstawowej, bez nałożonych więzów. Procedura ta jest odpowiednikiem procedury `g2h_FillHolef` i ma takie same parametry z wyjątkiem `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 ) );
```

Procedura `g2h_NLFillHoleConstrf` konstruuje powierzchnię wypełniającą, która minimalizuje funkcjonal  $F_d$  w przestrzeni podstawowej, z nałożonymi więzami opisanymi przez układ równań (9.4). Procedura ta jest odpowiednikiem procedury `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 ) );
```

Procedura `g2h_NLFillHoleAltConstrf` konstruuje powierzchnię wypełniającą, która minimalizuje funkcjonal  $F_d$  w przestrzeni podstawowej, z nałożonymi więzami opisanymi przez układ równań (9.5). Jest ona odpowiednikiem procedury `g2h_FillHoleAltConstrf`.

```
boolean g2h_NLExtFillHolef ( GHoleDomainf *domain,
                             const point3f *hole_cp,
                             float *acoeff,
                             void (*outpatch) ( int n, int m, const point3f *cp ) );
```

Procedura `g2h_NLExtFillHolef` dokonuje konstrukcji powierzchni minimalnej funkcjonału  $F_d$  w przestrzeni rozszerzonej, bez nałożonych więzów. Procedura ta jest odpowiednikiem procedury `g2h_ExtFillHolef` i ma takie same parametry (z wyjątkiem `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 ) );
```

Procedura `g2h_NLExtFillHoleConstrf` konstruuje powierzchnię wypełniającą, która minimalizuje funkcjonal  $F_d$  w przestrzeni rozszerzonej, z nałożonymi więzami opisanymi przez układ równań (9.4). Procedura ta jest odpowiednikiem procedury `g2h_ExtFillHoleConstrf`.

Dopuszczalna w konstrukcji wykonywanej przez tę procedurę macierz  $C$  układu równań opisujących więzy, wprowadzona przez wcześniejsze wywołanie procedury `g2h_SetExtConstraintMatrixf`, musi mieć w każdym wierszu pierwsze 16k współczynników zerowych.



Procedura `g2h_NLExtFillHoleAltConstrf` konstruuje powierzchnię wypełniającą, która minimalizuje funkcjonal  $F_d$  w przestrzeni rozszerzonej, z nałożonymi więzami opisanymi przez układ równań (9.5). Jest ona odpowiednikiem procedury `g2h_ExtFillHoleAltConstrf`.

```
boolean g2h_NLFunctionalValuef ( GHoleDomaininf *domain,
                                const point3f *hole_cp,
                                const vector3f *acoeff,
                                float *funcval );
```

```
boolean g2h_NLExtFunctionalValuef ( GHoleDomainf *domain,
                                   const point3f *hole_cp,
                                   const vector3f *acoeff,
                                   float *funcval );
```

```
boolean g2h_NLSplFillHolef ( GHoleDomaininf *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 Procedury wizualizacji

Nazwa „procedury wizualizacji” dotyczy procedur, które wyciągają z rekordu dziedziny rozmaite dane, na podstawie których można uzyskać wgląd w działanie konstrukcji, na przykład rysując rozmaite obrazki.

```
void g2h_DrawDomSurrndPatchesf ( GHoleDomainf *domain,
                                void (*drawpatch) ( int n, int m, const point2f *cp ) );
```

Procedura `g2h_DrawDomSurrndPatchesf` wyciąga reprezentacje Béziera stopnia (3,3) płatów otaczających dziedzinę, tj. wielomianowych kawałków płatów B-sklejanych reprezentowanych przez węzły i punkty kontrolne dziedziny podane podczas tworzenia rekordu dziedziny.

Parametr `domain` wskazuje rekord dziedziny, parametr `drawpatch` jest wskaźnikiem procedury, która dla otworu  $k$ -kątnego będzie wywołana  $3k$  razy, z parametrami opisującymi kolejne płaty otaczające dziedzinę.

```
void g2h_DrawDomAuxPatchesf ( GHoleDomainf *domain,
                              void (*drawpatch) ( int n, int m, const point2f *cp ) );
```

Procedura `g2h_DrawDomAuxPatchesf` wyciąga reprezentacje Béziera płatów pomocniczych dziedziny. Parametr `domain` wskazuje rekord dziedziny, a parametr `drawpatch` jest wskaźnikiem procedury, która będzie wywołana  $k$  razy, w celu przekazania aplikacji kolejnych płatów.

```
void g2h_DrawBasAuxPatchesf ( GHoleDomainf *domain, int fn,
                              void (*drawpatch) ( int n, int m, const float *cp ) );
```

Procedura `g2h_DrawBasAuxPatchesf` wyciąga reprezentacje Béziera płatów pomocniczych funkcji bazowych. Są to wielomiany dwóch zmiennych; jest ich  $k$  dla każdej funkcji będącej elementem bazy przestrzeni podstawowej. Wymiar  $n$  tej przestrzeni można poznać wywołując procedurę `g2h_V0SpaceDimf`.

Parametr `domain` wskazuje rekord dziedziny, parametr `fn` jest numerem funkcji bazowej (musi mieć wartość od 0 do  $n - 1$ ), parametr `drawpatch` jest wskaźnikiem procedury, która zostanie wywołana  $k$  razy w celu przekazania aplikacji kolejnych płatów pomocniczych funkcji bazowej o numerze `fn`.

```
void g2h_DrawJFunctionf ( GHoleDomainf *domain, int i, int l,
                          void (*drawpoly) ( int deg, const float *f ) );
```

Procedura `g2h_DrawJFunctionf` wyciąga współczynniki wielomianu (w bazie Bernsteina odpowiedniego stopnia), który jest funkcją połączenia używaną w konstrukcji funkcji bazowych. Na każdy podobszar  $\Omega_i$  dziedziny takich funkcji jest 16.

Parametr `domain` wskazuje rekord dziedziny. Parametr `i` jest numerem obszaru (musi być od 0 do  $k - 1$ ), parametr `l` wybiera funkcję połączenia, która ma być wprowadzona za pomocą procedury wskazywanej przez parametr `drawpoly`.

Wartość parametru  $l$  od 0 do 15 określa jedną z szesnastu funkcji połączenia, a od 16 do 27 iloczyn odpowiednich funkcji, używany w konstrukcji. Po informację, która funkcja ma który numer, odsyłam do tekstu źródłowego procedury.

```
void g2h_DrawDiPatchesf ( GHoleDomainf *domain,
                        void (*drawpatch) ( int n, int m, const point2f *cp ) );
```

Procedura `g2h_DrawDiPatchesf` wyciąga reprezentację Béziera stopnia  $(9,9)$  płatów dziedziny. Parametr `domain` wskazuje rekord dziedziny, a parametr `drawpatch` wskazuje procedurę, która ma być wywołana  $k$  razy w celu wyprowadzenia kolejnych płatów.

```
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 );
```

Procedura `g2h_ExtractPartitionf` wyciąga informację na temat podziału kąta pełnego w punkcie środkowym dziedziny  $\Omega$  podzielonej na podobszary  $\Omega_i$ .

```
void g2h_ExtractCentralPointf ( GHoleDomainf *domain,
                               point2f *centp, vector2f *centder );
```

Procedura `g2h_ExtractCentralPointf` wyciąga punkt środkowy dziedziny i pochodne pierwszego rzędu krzywych podziału dziedziny w punkcie środkowym.

Parametr `domain` wskazuje rekord dziedziny, parametr `centp` wskazuje zmienną, której ma być przypisany punkt środkowy, a parametr `centder` wskazuje tablicę o długości  $k$ , do której zostaną wpisane wektory pochodnych kolejnych krzywych.

```
void g2h_DrawBasAFunctionf ( GHoleDomainf *domain, int fn,
                             void (*drawpatch) ( int n, int m, const point3f *cp ) );
```

Procedura `g2h_DrawBasAFunctionf` służy do uzyskania informacji na temat funkcji bazowej o numerze  $fn \in \{0, \dots, n' - 1\}$ . Procedura `*drawpatch` jest wywoływana  $k$  razy, za każdym wywołaniem jej parametry opisują jeden płat dziedziny (współrzędne  $x, y$  punktów w tablicy `cp`) i odpowiedni płat funkcji bazowej (współrzędne  $z$ ). Parametry tej procedury opisują stopień, a w tablicy `cp` są punkty kontrolne Béziera.

```
void g2h_DrawBasBFunctionf ( GHoleDomainf *domain, int fn,
                             void (*drawpatch) ( int n, int m, const point3f *cp ) );
```

Procedura `g2h_DrawBasBFunctionf` służy do uzyskania informacji na temat funkcji bazowej o numerze  $fn \in \{n, \dots, n + m - 1\}$ . Procedura `*drawpatch`, która służy do przekazania tej informacji, jest wywoływana tak samo jak dla procedury `g2h_DrawBasAFunctionf` (i może to być ta sama procedura w programie).

```
void g2h_DrawBasCNetf ( GHoleDomainf *domain, int fn,
                        void (*drawnet) ( int n, int m, const point3f *cp ) );
```

Procedura `g2h_DrawBasCNetf` służy do uzyskania B-sklejanych siatek kontrolnych reprezentujących funkcję  $\varphi_i$  dla  $i = fn$ .

```
void g2h_DrawMatricesf ( GHoleDomainf *domain,
                        void (*drawmatrix)(int nfa, int nfb,
                                           float *amat, float *bmat) );
```

Procedura `g2h_DrawMatricesf` służy do uzyskania macierzy A i B, występujących w układzie równań (9.2), napisanym dla bazy podstawowej. Ponieważ macierz A jest symetryczna, jej reprezentacja jest „spakowana”, zgodnie z opisem w p. 3.3.

Parametr `drawmatrix` jest wskaźnikiem procedury, która zostanie wywołana z parametrami opisującymi macierze; `nfa` jest liczbą wierszy obu macierzy i liczbą kolumn macierzy A. Parametr `nfb` jest liczbą kolumn macierzy B. Parametry `amat` i `bmat` są tablicami współczynników.

```
void g2h_DrawExtMatricesf ( GHoleDomainf *domain,
                           void (*drawmatrix)(int k, int r, int s,
                                                float *Aii, float *Aki, float *Akk,
                                                float *Bi, float *Bk) );
```

Procedura `g2h_DrawMatricesf` służy do uzyskania macierzy A i B, występujących w układzie równań (9.2), napisanym dla bazy podstawowej. Macierz A jest symetryczna, o strukturze blokowej i jest reprezentowana w sposób opisany w p. 3.5. Macierz B jest pełna, jest ona reprezentowana w postaci blokowej.

Parametr `drawmatrix` jest wskaźnikiem procedury, która zostanie wywołana z parametrami opisującymi macierze; jej parametry `k`, `r` i `s` opisują liczbę i wielkość bloków macierzy A. W tablicach `Aii`, `Aki` i `Akk` są podane współczynniki macierzy A, a w tablicach `Bi` i `Bk` współczynniki macierzy B.

```
int g2h_DrawBFcpnf ( int hole_k, unsigned char *bfcpn );
```

Procedura `g2h_DrawBFcpnf` umieszcza w tablicy `bfcpn` indeksy tych punktów kontrolnych powierzchni i dziedziny, które są istotne dla kształtu brzegu otworu w powierzchni i dziedziny, a także płaszczyzny stycznej i krzywizny powierzchni na brzegu. Wszystkich punktów kontrolnych dla powierzchni z otworem k-kątnym jest  $12k + 1$ , tych istotnych jest  $6k + 1$ ; na rysunku 9.1 są one zaznaczone czarnymi kropkami. Kolejność indeksów punktów w tablicy odpowiada kolejności funkcji bazowych  $\phi_n, \dots, \phi_{n+m-1}$ .

Wartością procedury jest  $6k + 1$ .

```
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. Biblioteka libbsmesh

## 10.1 Reprezentacja siatki

Siatka jest obiektem składającym się z **wierzchołków**, **krawędzi** i **ścian**. Może ona służyć m.in. do reprezentowania np. bryły wielościennej lub powierzchni sklejanej. Krawędź jest odcinkiem łączącym wierzchołki. Ściana jest zamkniętą łamaną zbudowaną z krawędzi. Krawędź może należeć do jednej lub do dwóch ścian; w pierwszym przypadku jest nazywana **krawędzią brzegową**, a w drugim — **krawędzią wewnętrzną**.

W reprezentacji siatki przetwarzanej przez procedury z biblioteki libbsmesh krawędzie brzegowe i wewnętrzne są reprezentowane odpowiednio za pomocą jednej lub dwóch **półkrawędzi**. Półkrawędź ma orientację, tj. jeden z jej wierzchołków jest początkiem, a drugi końcem. Orientacja drugiej półkrawędzi w parze reprezentującej krawędź wewnętrzną jest przeciwna. Każda półkrawędź jest związana tylko z jedną ścianą.

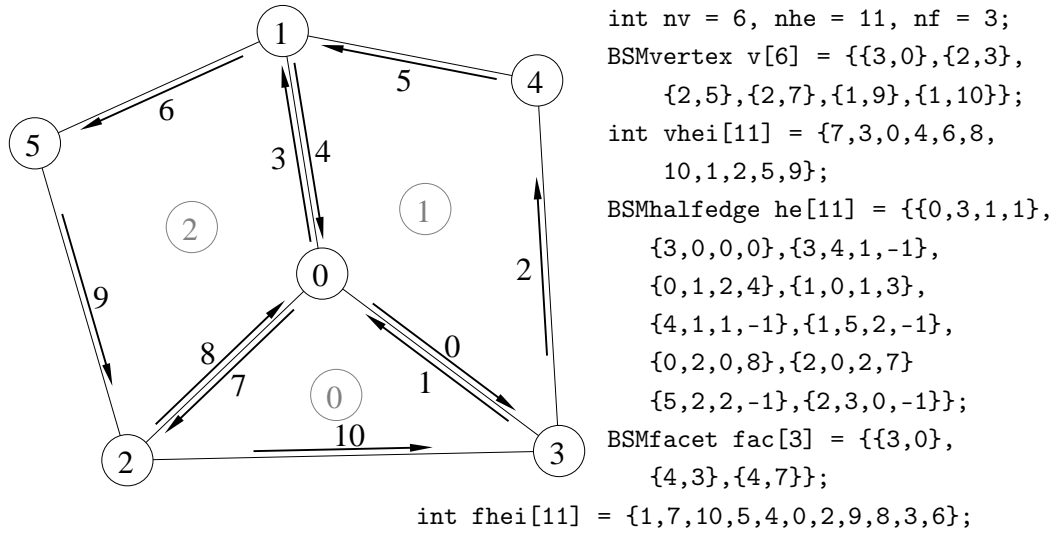
Wierzchołki, półkrawędzie i ściany są przechowywane w tablicach (indeksowanych od 0), przy czym identyfikatorem wierzchołka, półkrawędzi i ściany jest jego indeks w tablicy. Kompletna reprezentacja siatki składa się z trzech liczb: liczby wierzchołków  $n_v$ , liczby półkrawędzi  $n_h$  i liczby ścian  $n_f$ , oraz sześciu tablic: tablicy wierzchołków  $v$ , tablicy indeksów półkrawędzi wychodzących z wierzchołków  $vhei$ , tablicy pozycji wierzchołków  $pos$ , tablicy półkrawędzi  $he$ , tablicy ścian  $fac$  i tablicy indeksów półkrawędzi ścian  $fhei$ . Wierzchołki, ściany i półkrawędzie są opisane przez następujące struktury:

```
typedef struct {
    char degree;
    int firsthalfedge;
} BSMfacet, BSMvertex;

typedef struct {
    int v0, v1;
    int facetnum;
    int otherhalf;
} BSMhalfedge;
```

Przykład reprezentacji siatki jest pokazany na rysunku 10.1; tablica ze współrzednymi wierzchołków jest pominięta.

```
boolean bsm_CheckMeshIntegrity (
    int nv, const BSMvertex *mv, const int *mvhei,
```



Rys. 10.1. Przykład siatki

```

int nhe, const BSMhalfedge *mhe,
int nfac, const BSMfacet *mfac, const int *mfhei );

```

```

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 );

```



## 10.2 Procedury zagęszczania siatki

```
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 );
```

```
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 );
```

```
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 );
```

```
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 );
```

```
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 );
```

```
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 );
```

### 10.3 Operacje Eulerowskie i nie-Eulerowskie

```
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 );
```

```
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 );
```

```
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 );
```

```

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 );

```

```

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 );

```

```

int bsm_HalfedgeLoopLength ( int nv, BSMvertex *mv, int *mvhei,
                             int nhe, BSMhalfedge *mhe,
                             int he );

```

```

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 );

```



```

        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 );

```

```

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 );

```

## 10.5 Inne procedury

```

void bsm_TagBoundaryZoneVertices ( int nv, BSMvertex *mv, int *mvhei,
        int nhe, BSMhalfedge *mhe,
        char d, char *vtag );

```

```

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 );

```

## 11. Biblioteka libg1blending

Ten rozdział czeka na napisanie, gdy będę miał na to czas. Znaczna część tej biblioteki została napisana przez Mateusza Markowskiego.

```

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 );
```

[illegible]

```

        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 d0, 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 d0, 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 d0, 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 d0, 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 ncurv, 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 ncurv, double *ucknots,
        int cpitch, double *uccp,
        int *nconstr, double *cmat, double
        *crhs );

```

```

boolean g1bl_SetupClosedUNLConstraintsd ( int lastknotu, int lastknotv,
        int ppitch, point3d *cp,
        int ncurv, 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 );

```





```

        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_InitBlSurfaceOptLMTd ( int lastknotu, int lastknotv, int pitch,
        point3d *cp,
        double C, double d0, 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 d0, double dM,
        int maxit, int nkn1, int nkn2 );

```

```

boolean g2bl_InitBlSurfaceConstrOptLMTd ( int lastknotu, int lastknotv, int
pitch,
        point3d *cp,
        int nconstr, double *constrmat,
        double *constrrrhs,
        double C, double d0, 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 d0, double dM,
                                int maxit, int nkn1, int nkn2 );

```

```

boolean g2bl_ClosedInitBlSurfaceOptLMTd ( int lastknotu, int lastknotv, int
pitch,
                                point3d *cp,
                                double C, double d0, 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 d0, 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 d0, 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 d0, double dM,
    int maxit, int nkn1, int nkn2 );

```

```

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 Optimalizacja powierzchni reprezentowanych przez siatki nieregularne

```

extern GHoleDomaind *g2mbl_domaind[GH_MAX_K-3];
extern double *g2mbl_patchmatrixd[GH_MAX_K-3];

```

```
extern void (*g2mbl_outputnzdistr)( int nbl, int blnum, boolean final,
                                     int nvcp, int n, byte *nzdistr );
```

```
void g2mbl_CleanupHoleDomainsd ( void );
boolean g2mbl_SetupHolePatchMatrixd ( int k );
```

```
void g2mbl_OptLMTDeallocated ( void **data );
```

```
int g2mbl_GetNvcp ( int nv, BSMvertex *mv, int *mvhei,
                   int nhe, BSMhalfedge *mhe,
                   int nfac, BSMfacet *mfac, int *mfhei,
                   byte *mkcp );
```

```
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 );
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 );
```

```
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 ( /* fine mesh */
                                     int fnv, BSMvertex *fmv, int *fmvhei, point3d *fmvcp,
                                     int fnhe, BSMhalfedge *fmhe,
                                     int fnfac, BSMfacet *fmfac, int *fmfhei,
                                     byte *fmkcp,
```

```

        /* number of vertices of the coarse mesh */
int cnv,
        /* refinement matrix */
int rmnz, index2 *rmnzi, double *rmnzc,
        /* optimization parameters */
double C, double d0, double dM,
int nkn1, int nkn2, int nbl,
        /* created data structure */
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 );

```

```

boolean g2mbl_TimeBlSurfaceOptBLMTd ( void *data, int bnum, double *tt,
int *n0, int *n, int **prof,
double ***rows, double ***lrows,
double *_func, double *_grad,
int **iHbl, int **tHbl, double **Hbl );

```

```

int g2mbl_GetBLMBlockNumd ( void *data, int *lastblock );
void g2mbl_GetBLMTBlockInfod ( void *data,
int bln, int *nv, int *nvcp, int **nncpi,
int *c0, int *bnvcp,
int **vncpi, int **bvncpi, int **vpermut );

```

```

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 ( /* fine mesh */
int fnv, BSMvertex *fmv, int *fmvhei, point3d *fmvcp,
int fnhe, BSMhalfedge *fmhe,
int fnfac, BSMfacet *fmfac, int *fmfhei,

```



```

        byte *fmkcp,
            /* number of vertices of the coarse mesh */
        int cnv,
            /* refinement matrix */
        int rmnz, index2 *rmnzi, double *rmnzc,
            /* optimization parameters */
        double C, double d0, double dM,
        int nkn1, int nkn2, short nlevels,
            /* created data structure */
        void **data );
void g2mbl_MLOptDeallocated ( void **data );
boolean g2mbl_MLOptIterd ( void *data, boolean *finished );

void g2mbl_MLSetLogLeveld ( void *data, short level );
short g2mbl_MLGetInfod ( void *data );

int g2mbl_MLGetLastBlockd ( void *data );
boolean g2mbl_MLGetBlockVCPNumbersd ( void *data, int bl,
                                     int *nvcp, int **vncpi, int *seed );

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_MLCPSSuggestNLevels ( int nv, BSMvertex *mv, int *mvhei,
                                    int nhe, BSMhalfedge *mhe,
                                    int nfac, BSMfacet *mfac, int *mfhei,
                                    byte *mkcp,
                                    int *minlev, int *maxlev );

void g2mbl_MLSetNextBlock ( void *data, int nbl );

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 ( /* fine mesh */
                                int fnv, BSMvertex *fmv, int *fmvhei, point3d *fmvcp,

```

```

        int fnhe, BSMhalfedge *fmhe,
        int fnfac, BSMfacet *mfac, int *mfhei,
        byte *fmkcp,
            /* coarse mesh */
        int cnv,
            /* refinement matrix */
        int rmnnz, index2 *rmnzi, double *rmnzc,
            /* optimization parameters */
        int nkn1, int nkn2, short nlevels,
            /* created data structure */
        void **data );
boolean g2mbl_MLSOptIterd ( void *data, boolean *finished );

```

```

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 );

```

```

void g2mbl_MLGetTimes ( void *data,
                        float *time_prep, float *time_h, float *time_cg );

```

## 13. Biblioteka libbsfile

```
#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. Biblioteka libxgedit

Procedury w bibliotece libxgedit obsługują dialog aplikacji z użytkownikiem, poprzez system XWindow. Biblioteka umożliwia tworzenie wielu okien i wihajstrów i zastępuje w tej roli wszelkie bardziej wyrafinowane nakładki na system XWindow. Głównym jej celem jest zastosowanie w programach demonstracyjnych pakietu BSTools (i być może do niczego innego się ona nie nadaje).

```
#if __WORDSIZE == 64
typedef unsigned int xgecolour_int;
#else
#if __WORDSIZE == 32
typedef unsigned long xgecolour_int;
#else
#error Either 32-bit or 64-bit integers are assumed
#endif
#endif
```

```
#define xge_MAX_WINDOWS 8
#define xge_MAX_CURSORS 16

#define xge_MAX_WIDTH 1024
#define xge_MAX_HEIGHT 768
#define xge_WIDTH 480
#define xge_HEIGHT 360
```

```
#define xge_CHAR_WIDTH 6
#define xge_CHAR_HEIGHT 13
```

```
#define xge_RECT_NONE -1
#define xge_MINDIST 8
```

```
/* mouse button states */
#define xgemouse_LBUTTONDOWN (1 < 0)
#define xgemouse_LBUTTONDOWN_CHANGE (1 < 1)
#define xgemouse_RBUTTONDOWN (1 < 2)
#define xgemouse_RBUTTONDOWN_CHANGE (1 < 3)
#define xgemouse_MBUTTONDOWN (1 < 4)
#define xgemouse_MBUTTONDOWN_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)
```

```
/* message codes */
#define xgemsg_NULL 0
#define xgemsg_INIT 0x100
```

```
/* user action messages */
#define xgemsg_KEY 0x101
#define xgemsg_SPECIAL_KEY 0x102
#define xgemsg_MMOVE 0x103
#define xgemsg_MCLICK 0x104
#define xgemsg_STATE_CHANGE 0x105
#define xgemsg_OTHHEREVENT 0x106
```

```
/* commands sent by libxgedit procedures */
#define xgemsg_ENTERING 0x107
#define xgemsg_EXITING 0x108
#define xgemsg_RESIZE 0x109
#define xgemsg_MOVE 0x10A
#define xgemsg_BUTTON_COMMAND 0x10B
#define xgemsg_SWITCH_COMMAND 0x10C
#define xgemsg_SLIDEBAR_COMMAND 0x10D
#define xgemsg_KNOB_COMMAND 0x10E
#define xgemsg_TEXT_EDIT_COMMAND 0x10F
#define xgemsg_INT_WIDGET_COMMAND 0x110
#define xgemsg_LISTBOX_ITEM_SET 0x111
#define xgemsg_LISTBOX_ITEM_PICK 0x112
```



```
#define xgemsg_2DWIN_RESIZE 0x113
#define xgemsg_2DWIN_PROJCHANGE 0x114
#define xgemsg_2DWIN_PICK_POINT 0x115
#define xgemsg_2DWIN_MOVE_POINT 0x116
#define xgemsg_2DWIN_SELECT_POINTS 0x117
#define xgemsg_2DWIN_UNSELECT_POINTS 0x118
#define xgemsg_2DWIN_CHANGE_TRANS 0x119
#define xgemsg_2DWIN_SAVE_POINTS 0x11A
#define xgemsg_2DWIN_TRANSFORM_POINTS 0x11B
#define xgemsg_2DWIN_FIND_REFBBOX 0x11C
#define xgemsg_2DWIN_ERROR 0x11D
```

```
#define xgemsg_3DWIN_RESIZE 0x11E
#define xgemsg_3DWIN_PROJCHANGE 0x11F
#define xgemsg_3DWIN_PICK_POINT 0x120
#define xgemsg_3DWIN_MOVE_POINT 0x121
#define xgemsg_3DWIN_SELECT_POINTS 0x122
#define xgemsg_3DWIN_UNSELECT_POINTS 0x123
#define xgemsg_3DWIN_CHANGE_TRANS 0x124
#define xgemsg_3DWIN_SAVE_POINTS 0x125
#define xgemsg_3DWIN_TRANSFORM_POINTS 0x126
#define xgemsg_3DWIN_FIND_REFBBOX 0x127
#define xgemsg_3DWIN_ERROR 0x128
```

```
#define xgemsg_KNOTWIN_CHANGE_KNOT 0x129
#define xgemsg_KNOTWIN_INSERT_KNOT 0x12A
#define xgemsg_KNOTWIN_REMOVE_KNOT 0x12B
#define xgemsg_KNOTWIN_CHANGE_MAPPING 0x12C
#define xgemsg_KNOTWIN_ERROR 0x12D
```

```
#define xgemsg_T2KNOTWIN_RESIZE 0x12E
#define xgemsg_T2KNOTWIN_PROJCHANGE 0x12F
#define xgemsg_T2KNOTWIN_CHANGE_KNOT_U 0x130
#define xgemsg_T2KNOTWIN_CHANGE_KNOT_V 0x131
#define xgemsg_T2KNOTWIN_INSERT_KNOT_U 0x132
#define xgemsg_T2KNOTWIN_INSERT_KNOT_V 0x133
#define xgemsg_T2KNOTWIN_REMOVE_KNOT_U 0x134
#define xgemsg_T2KNOTWIN_REMOVE_KNOT_V 0x135
#define xgemsg_T2KNOTWIN_SELECT_POINTS 0x136
#define xgemsg_T2KNOTWIN_UNSELECT_POINTS 0x137
#define xgemsg_T2KNOTWIN_CHANGE_MAPPING 0x138
#define xgemsg_T2KNOTWIN_ERROR 0x139
```

```
#define xgemsg_IDLE_COMMAND 0x13A
```

```
/* additional application messages must be greater than  
xgemsg_LAST_MESSAGE */  
#define xgemsg_LAST_MESSAGE xgemsg_IDLE_COMMAND
```

```
/* States of the program. Others may be defined in applications. */
#define xgestate_NOTHING 0
#define xgestate_MOVINGSLIDE 1
#define xgestate_TURNINGKNOB 2
#define xgestate_MESSAGE 3
#define xgestate_RESIZING_X 4
#define xgestate_RESIZING_Y 5
#define xgestate_RESIZING_XY 6
#define xgestate_TEXT_EDITING 7
#define xgestate_2DWIN_MOVINGPOINT 8
#define xgestate_2DWIN_PANNING 9
#define xgestate_2DWIN_ZOOMING 10
#define xgestate_2DWIN_SELECTING 11
#define xgestate_2DWIN_UNSELECTING 12
#define xgestate_2DWIN_MOVING_GEOM_WIDGET 13
#define xgestate_2DWIN_USING_GEOM_WIDGET 14
#define xgestate_2DWIN_ALTUSING_GEOM_WIDGET 15
#define xgestate_3DWIN_MOVINGPOINT 16
#define xgestate_3DWIN_PARPANNING 17
#define xgestate_3DWIN_PARZOOMING 18
#define xgestate_3DWIN_TURNING_VIEWER 19
#define xgestate_3DWIN_PANNING 20 #define xgestate_3DWIN_ZOOMING 21
#define xgestate_3DWIN_SELECTING 22
#define xgestate_3DWIN_UNSELECTING 23
#define xgestate_3DWIN_MOVING_GEOM_WIDGET 24
#define xgestate_3DWIN_USING_GEOM_WIDGET 25
#define xgestate_3DWIN_ALTUSING_GEOM_WIDGET 26
#define xgestate_KNOTWIN_MOVINGKNOT 27
#define xgestate_KNOTWIN_PANNING 28
#define xgestate_KNOTWIN_ZOOMING 29
#define xgestate_T2KNOTWIN_MOVINGKNOT_U 30
#define xgestate_T2KNOTWIN_MOVINGKNOT_V 31
#define xgestate_T2KNOTWIN_MOVING_POINT 32
#define xgestate_T2KNOTWIN_PANNING 33
#define xgestate_T2KNOTWIN_ZOOMING 34
#define xgestate_T2KNOTWIN_SELECTING 35
#define xgestate_T2KNOTWIN_UNSELECTING 36
/* additional application states must be greater than
xge_LAST_STATE */
#define xgestate_LAST xgestate_T2KNOTWIN_UNSELECTING
```

```
#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
```

```
/* wrappers around XWindow drawing and some other procedures */
#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,xgепixmap,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)
#define xgeDefineCursor(cursor) \
XDefineCursor(xgedisplay,xgewindow,cursor)

```

```

typedef struct xge_widget {
int id;
short w, h, x, y;
void *data0, *data1;
short xofs, yofs;
char rpos;
char window_num;
boolean (*msgproc) ( struct xge_widget*, int, int, short, short );
void (*redraw) ( struct xge_widget*, boolean );
struct xge_widget *next, *prev, *up;
struct xge_widget *prevfocus;
} 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 );

```

```

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 Window xgewindow;
extern Pixmap xgepixmap;
extern int xgescreen;
extern GC xgegc;
extern Visual *xgevisual;
extern XSizeHints xgehints;
extern Cursor xgecursor[];
extern KeySym xgekeysym;
extern XEvent xgeevent;
extern float xge_aspect;
```

```
extern unsigned short xge_current_width, xge_current_height;
```

```
extern char *xge_p_name;
extern char xge_done;
extern short xge_prevx, xge_prevy;
extern int xge_slidebarid;
```

```
extern xgecolour_int xge_foreground, xge_background;
extern int xge_nplanes;
extern xgecolour_int *xge_palette;
extern const char *xge_colour_name[];
```

```
extern int xge_state, xge_prev_state;
extern xge_widget *xge_focus;
extern boolean xge_notinfocus;
```

```
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 );
```

```
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 );
```

```
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 );
```

```
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 );
```

```
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 );
```

```
void xge_DrawSliderbarf ( xge_widget *er, boolean onscreen );
boolean xge_SliderbarfMsg ( xge_widget *er, int msg, int key, short
x, short y );
xge_widget *xge_NewSliderbarf ( char window_num, xge_widget *prev,
int id,
short w, short h, short x, short y,
float *data );
```

```
void xge_DrawVSliderbarf ( xge_widget *er, boolean onscreen );
boolean xge_VSliderbarfMsg ( xge_widget *er, int msg, int key, short
x, short y );
xge_widget *xge_NewVSliderbarf ( char window_num, xge_widget *prev,
int id,
short w, short h, short x, short y,
float *data );
```

```
float xge_LinSliderbarValuef ( float xmin, float xmax, float t );
float xge_LogSliderbarValuef ( float xmin, float xmax, float t );
```

```
void xge_DrawKnobf ( xge_widget *er, boolean onscreen );
boolean xge_KnobfMsg ( xge_widget *er, int msg, int key, short x,
short y );
xge_widget *xge_NewKnobf ( char window_num, xge_widget *prev, int
id,
short w, short h, short x, short y,
char *title, float *data );
```

```
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 );
```

```
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 */
boolean active;
} 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 );
```

```
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 );
```

```
#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 */
} 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_SetupFileList ( xge_listbox *lbox, const char *dir,
const char *filter );
boolean xge_SetupDirList ( xge_listbox *lbox, const char *dir,
const char *filter );
boolean xge_FilterMatches ( const char *name, const char *filter );
```

```
#define xge_2DWIN_MIN_ZOOM 0.1
#define xge_2DWIN_MAX_ZOOM 10.0
```

```
#define xge_2DWIN_NO_TOOL 0
#define xge_2DWIN_MOVING_TOOL 1
#define xge_2DWIN_SCALING_TOOL 2
#define xge_2DWIN_ROTATING_TOOL 3
```

```
typedef struct xge_2Dwinf {
xge_widget *er;
CameraRecf CPos;
Box2f DefBBox, RefBBox;
boolean panning, selecting_mode;
boolean display_coord, inside;
boolean moving_tool, scaling_tool, rotating_tool;
char current_tool;
int current_point;
short xx, yy;
float zoom;
Box2s selection_rect;
point2f saved_centre;
point2f scaling_centre;
vector2f scaling_factors;
short scaling_size;
char scaling_mode;
point2f rotating_centre;
short rotating_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, 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 );

```

```

typedef struct xge_fourww {
xge_widget *er, *last;
xge_widget *win[4];
float xsfr, ysfr;
short splitx, splity;
boolean resized;
} xge_fourww;

```

```

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 );

```

```

#define xge_3DWIN_MIN_PARZOOM 0.1
#define xge_3DWIN_MAX_PARZOOM 10.0
#define xge_3DWIN_MIN_ZOOM 1.0
#define xge_3DWIN_MAX_ZOOM 100.0

```

```
#define xge_3DWIN_NO_TOOL 0
#define xge_3DWIN_MOVING_TOOL 1
#define xge_3DWIN_SCALING_TOOL 2
#define xge_3DWIN_ROTATING_TOOL 3
```

```
typedef struct xge_3Dwinf {
xge_fourww fww; /* this must be the first component */
xge_widget *cwin[4];
CameraRecf CPos[4];
Box3f DefBBox, RefBBox, PerspBBox;
boolean panning, selecting_mode;
boolean display_coord;
boolean moving_tool, scaling_tool, rotating_tool;
char current_tool;
signed char CoordWin;
int current_point;
short xx, yy;
float perspzoom;
Box2s selection_rect;
point3f saved_centre;
point3f scaling_centre;
vector3f scaling_factors;
short scaling_size;
char scaling_mode;
point3f rotating_centre;
short rotating_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_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, 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 );
```

```
#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;
boolean closed;
float akm, bkm, umin, umax, knotscf, knotshf;
int clcK;
float clcT;
int degree;
unsigned char current_mult;
short xx;
int maxknots, lastknot;
float *knots;
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_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, int x );
boolean xge_KnotWinfInsertKnot ( xge_KnotWinf *knw, int x );
boolean xge_KnotWinfRemoveKnot ( xge_KnotWinf *knw );

```

```
typedef struct {
xge_widget *er;
CameraRecf CPos;
Box2f DefBBox, RefBBox;
point3f centre;
boolean panning, selecting_mode, moving_many;
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;
} 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 );

```

```

int xge_T2KnotWinfFindDomWinRegion ( xge_T2KnotWinf *T2win, int x,
int y );
int xge_T2KnotWinfFindNearestKnot ( xge_T2KnotWinf *T2win, int x,
int y );
int xge_T2KnotWinfMapKnotU ( xge_T2KnotWinf *T2win, float u );
float xge_T2KnotWinfUnmapKnotU ( xge_T2KnotWinf *T2win, int xi );
int xge_T2KnotWinfMapKnotV ( xge_T2KnotWinf *T2win, float v );
float xge_T2KnotWinfUnmapKnotV ( xge_T2KnotWinf *T2win, int eta );
boolean xge_T2KnotWinfSetKnotU ( xge_T2KnotWinf *T2win, int x );
boolean xge_T2KnotWinfInsertKnotU ( xge_T2KnotWinf *T2win, int x );
boolean xge_T2KnotWinfRemoveKnotU ( xge_T2KnotWinf *T2win );
boolean xge_T2KnotWinfSetKnotV ( xge_T2KnotWinf *T2win, int y );
boolean xge_T2KnotWinfInsertKnotV ( xge_T2KnotWinf *T2win, int 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 );

```

```

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 );
```

```
void xge_AddPopup ( xge_widget *er );
void xge_RemovePopup ( void );
void xge_RemovePopups ( void );
```

```
/* error and info message procedures */
void xge_DisplayErrorMessage ( char *message );
void xge_DisplayWarningMessage ( char *message );
void xge_DisplayInfoMessage ( char **msglines );
```

```
void xge_OutPixels ( const xpoint *buf, int n );
void xge_DrawBC2f ( int n, const point2f *cp );
void xge_DrawBC2Rf ( int n, const point3f *cp );
```

```
int xge_NewWindow ( char *p_name );
boolean xge_SetWindow ( int win );
int xge_CurrentWindow ( void );
void xge_SetWinEdRect ( xge_widget *edr );
int xge_NewCursor ( unsigned int shape );
```

```
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
);
```

```
void xge_SetClipping ( xge_widget *edr );  
void xge_ResetClipping ( void );
```

```
void xge_RepositionWidgets ( short w, short h, short x, short y,  
xge_widget *edr );
```

```
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_OrderSelectionRect ( Box2s *sel_rect );  
void xge_DrawGeomWinBackground ( xge_widget *er );  
void xge_DrawGeomWinFrame ( xge_widget *er, boolean onscreen );  
void xge_Draw2DGeomWinfCursorPos ( xge_widget *er, CameraRecf  
*CPos,  
short x, short y );  
void xge_DrawGeomWinSelectionRect ( xge_widget *er, Box2s *sel_rect  
);
```

```
void xge_GetWindowSize ( void );  
void xge_Init ( int argc, char *argv[],  
int (*callback)(xge_widget*,int,int,short,short) );  
void xge_Cleanup ( void );  
void xge_MessageLoop ( 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. Programy demonstracyjne

Programy demonstracyjne dołączone do bibliotek (w poddrzewie katalogów demo) mają trzy cele. Po pierwsze wyświetlają ruchome obrazki, które mogą pomóc w zaznajamianiu się z krzywymi i powierzchniami sklejanymi. Po drugie, umożliwiają testowanie procedur z bibliotek (wiele błędów zostało wykryte dzięki tym programom). Po trzecie, mogą stanowić źródło informacji o sposobie wykorzystania bibliotek w innych zastosowaniach.

Programy te powstawały w sposób niezbyt systematyczny, na zasadzie dorabiania kolejnych funkcji w miarę potrzeb zgłaszanych przez moje widzimsię. Dlatego **nie są one przykładem** szczególnie eleganckiego programowania. Są one oparte o założenie, że poza działającym środowiskiem XWindow nie ma w systemie żadnych specyficznych bibliotek graficznych ani innych (np. Gnome, KDE, Athena, Motif, OpenGL). Programy te korzystają więc z interfejsu z systemem realizowanego tylko przez bibliotekę Xlib. Dzięki temu powinno dać się je skompilować i uruchomić na dowolnym komputerze wyposażonym w system XWindow. Osoby, których wygląd, możliwości lub sposób obsługi tych programów nie zadowala, zachęcam do zaprojektowania i napisania własnych programów; nie wątpię, że będą znacznie lepsze.

## 15.1 Program pokrzyw

Program pokrzyw (katalog demo/pokrzyw) wyświetla i umożliwia kształtowanie (pokrzywienie) płaskiej („zwykłej” lub wymiernej) krzywej B-sklejanej stopnia od 1 do 8. Dwa okna z prawej strony ekranu (okna programu utworzonego przez system XWindow) zawierają obraz krzywej (wraz z lamana kontrolną i innymi obiektami) oraz ciągu węzłów.

Większość poleceń wydaje się za pomocą lewego guzika myszy; służy on do „chwytania i trzymania” punktów kontrolnych i węzłów, a także do pstrykania<sup>1</sup> wihajstrów<sup>2</sup> (guzików itp.).

Prawy guzik służy tylko do wstawiania węzłów. Aby przesunąć węzeł lub punkt kontrolny należy ustawić na nim kursor i nacisnąć *lewy* guzik, a następnie przesunąć kursor dokądkolwiek. Aby usunąć węzeł należy wskazać go kursorem, nacisnąć *lewy* guzik i trzymając go nacisnąć klawisz **R**.

Lewa strona ekranu zawiera menu, tj. kolekcję wihajstrów, które służą do wydawania poleceń. Aby zakończyć działanie programu należy pstryknąć (lewym guzikiem myszy) guzik **Quit**, albo nacisnąć klawisz **Q**.

---

<sup>1</sup>to jest jedyny uznawany przeze mnie polski odpowiednik angielskiego czasownika „to click”.

<sup>2</sup>ang. „widgets”.

Naciśnięcie klawisza **[M]** nadaje oknu programu wielkość maksymalną, zaś **[m]** minimalną. Pozostałe polecenia interpretowane przez program (poprzez wihajstry) można odgadnąć po jego uruchomieniu.

## 15.2 Program pognij

Program pognij (katalog demo/pognij) wyświetla i umożliwia kształtowanie (pogięcie) krzywej („zwykłej” lub wymiernej) B-sklejanej w przestrzeni trójwymiarowej. Od programu pokrzyw różni się on głównie tym, że ma cztery okna z obrazem krzywej zamiast jednego.

Trzy z tych okien przedstawiają obraz krzywej w rzutach prostopadłych na płaszczyzny  $xy$ ,  $yz$ ,  $zx$  układu współrzędnych, a czwarte (na dole z prawej strony) obraz krzywej w rzucie perspektywicznym. Pierwsze trzy okna umożliwiają zmienianie punktów kontrolnych krzywej; można w nich „przesuwać” punkty kontrolne (trzymając naciśnięty lewy guzik myszy). W oknie czwartym lewy guzik służy do zmieniania położenia obserwatora, a prawy guzik umożliwia „zmienianie ogniskowej” kamery użytej do otrzymania obrazu krzywej w tym oknie (należy go nacisnąć i przesunąć kursor w górę lub w dół).

Naciśnięcie klawisza **[R]** w chwili, gdy kursor jest w oknie rzutu perspektywicznego powoduje przywrócenie początkowego położenia obserwatora.

Wielkość czterech okien z obrazami krzywej można zmieniać. W tym celu należy ustawić kursor między tymi oknami (wyświetlany kształt kursora zmieni się), a następnie nacisnąć lewy guzik i przesunąć kursor. Naciśnięcie klawisza **[R]** w chwili gdy kursor znajduje się między tymi oknami powoduje nadanie im jednakowych wielkości.

Pozostałe elementy obsługi tego programu są takie same jak programu pokrzyw.

## 15.3 Program pomnij

## 15.4 Program polep

Program polep służy do demonstrowania procedury wypełniania wielokątnego otworu w powierzchni skleianej (złożonej z płatów wielomianowych stopnia  $(3,3)$ ). Procedura ta jest opisana szczegółowo w książce *Podstawy modelowania krzywych i powierzchni*. Cztery okna, w których program wyświetla obrazy powierzchni, są obsługiwane w sposób identyczny jak cztery okna z obrazami krzywej w programie pognij.

Punkty kontrolne powierzchni można zmieniać za pomocą myszy, w oknach z rzutami równoległymi. Oprócz tego program zawiera generator „gotowych” danych, obsługiwany przez trzy suwaki w górnej części menu.

# 16. Projekty

Ten rozdział dokumentacji jest poświęcony procedurom „wysokopoziomowym” rozwiązującym różne, raczej skomplikowane zadania, zrealizowanym przy użyciu bibliotek opisanych wcześniej. Obecnie w pakiecie jest jedna taka procedura, konstruująca powierzchnie gładko wypełniające wielokątne otwory w powierzchniach złożonych z płatów prostokątnych.

## 16.1 Wypełnianie wielokątnych otworów

W programie demonstracyjnym polep jest procedura wypełniania wielokątnego otworu w uogólnionej powierzchni B-sklejanej trzeciego stopnia, płatami Béziera stopnia (5, 5), z zachowaniem ciągłości  $G^1$  połączeń tych płatów między sobą i z płatami otaczającymi otwór. Dokładny opis teoretyczny tej procedury znajduje się w książce *Podstawy modelowania krzywych i powierzchni*. Konstrukcja realizowana przez tę procedurę jest znacznie prostsza i mniej ogólna niż konstrukcje realizowane za pomocą procedur znajdujących się w bibliotekach `libg1hole` i `libg2hole`; była ona opracowana znacznie wcześniej i zebrane przy tej okazji doświadczenia pomogły przy opracowaniu konstrukcji zaimplementowanych w tych bibliotekach.

W tym miejscu jest opis sposobu reprezentowania danych dla tej procedury i jej parametrów.

Kod źródłowy procedury w wersji pojedynczej precyzji jest w pliku `g1holef.c`, a odpowiedni plik nagłówkowy to `g1holef.h`. Odpowiednie pliki dla podwójnej precyzji mają nazwy odpowiednio `g1holed.c` i `g1holed.h`.

```
boolean FillG1Holef ( int hole_k, point3f>(*GetBezp)(int i, int j),
                    float beta1, float beta2,
                    point3f *hpcp );
```

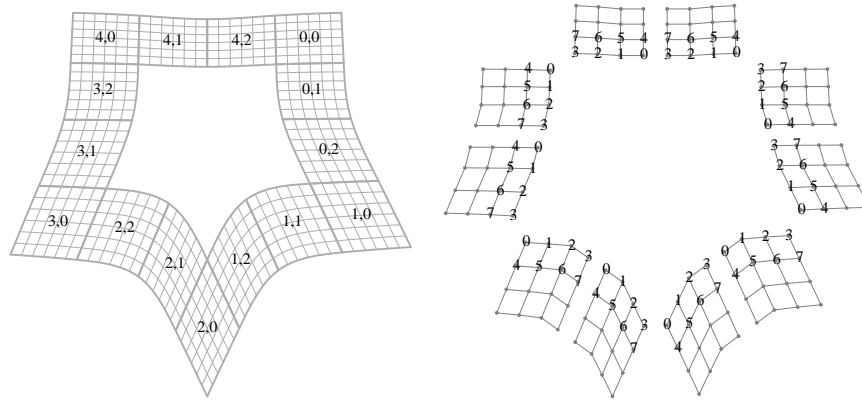
Procedura `FillG1Holef` konstruuje powierzchnię wypełniającą k-kątny otwór w powierzchni. Parametry tej procedury są następujące:

Parametr `hole_k` określa liczbę k wierzchołków otworu. Powinna to być liczba 3, 5, 6, 7 lub 8.

Parametr `GetBezp` jest wskaźnikiem procedury którą `FillG1Holef` wywołuje w celu otrzymania punktów kontrolnych płatów Béziera stopnia (3, 3) otaczających otwór. Procedura ta ma zwrócić (jako wartość) wskaźnik tablicy, w której znajdują się punkty kontrolne tych płatów.

Płaty otaczające otwór są określane przez pary liczb (i, j) zgodnie ze schematem na rysunku 16.1; zmienna i (parametr i) przyjmuje wartości od 0 do k−1, zmienna j (parametr j) wartości 1 lub 2. Na rysunku tym jest pokazana również kolejność,

w jakiej należy umieścić punkty kontrolne tych pól w tablicy. Dla każdego pola wystarczy podać w tablicy tylko 8 punktów, których numery są widoczne na rysunku.



Rys. 16.1. Schemat numeracji pól otaczających otwór i kolejność punktów kontrolnych w tablicach

Płaty otaczające otwór muszą spełniać podane niżej warunki zgodności narożników, pochodnych cząstkowych i pochodnych mieszanych.  $m$ -ty punkt kontrolny pła  $(i, j)$  jest oznaczony symbolem  $\mathbf{p}_m^{(i,j)}$ . Dodatkowo  $l = i + 1 \bmod k$ .

- Warunki zgodności narożników:

$$\mathbf{p}_0^{(i,1)} = \mathbf{p}_3^{(i,2)} \quad \text{oraz} \quad \mathbf{p}_0^{(i,2)} = \mathbf{p}_3^{(l,1)}.$$

- Warunki zgodności pochodnych cząstkowych:

$$\begin{aligned} \mathbf{p}_0^{(i,1)} - \mathbf{p}_1^{(i,1)} &= \mathbf{p}_2^{(i,2)} - \mathbf{p}_3^{(i,2)}, \\ \mathbf{p}_0^{(i,2)} - \mathbf{p}_1^{(i,2)} &= \mathbf{p}_7^{(l,1)} - \mathbf{p}_3^{(l,1)}, \\ \mathbf{p}_0^{(i,2)} - \mathbf{p}_4^{(i,2)} &= \mathbf{p}_2^{(l,1)} - \mathbf{p}_3^{(l,1)}. \end{aligned}$$

- Warunki zgodności pochodnych mieszanych:

$$\begin{aligned} \mathbf{p}_4^{(i,1)} - \mathbf{p}_5^{(i,1)} &= \mathbf{p}_6^{(i,2)} - \mathbf{p}_7^{(i,2)}, \\ \mathbf{p}_0^{(i,2)} - \mathbf{p}_1^{(i,2)} - \mathbf{p}_4^{(i,2)} + \mathbf{p}_5^{(i,2)} &= \mathbf{p}_6^{(l,1)} - \mathbf{p}_7^{(l,1)} - \mathbf{p}_2^{(l,1)} + \mathbf{p}_3^{(l,1)}. \end{aligned}$$

Parametry  $\beta_1$  i  $\beta_2$  są czynnikami, przez które procedura FillG1Holef mnoży pewne wektory konstruowane w trakcie obliczeń. W zasadzie ich wartość powinna być równa 1, ale można podać inną w celu skorygowania kształtu (jeśli wartość 1 daje nieodpowiedni efekt).



Parametr `hpcp` jest wskaźnikiem tablicy, do której procedura wstawia punkty kontrolne płatów Béziera stopnia (5,5), z których składa się powierzchnia wypełniająca otwór.

Wartością procedury jest `true` jeśli konstrukcja powierzchni wypełniającej zakończyła się sukcesem, albo `false` w przeciwnym razie.

```
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 ndegu, int ndegv,
                                const point3f *apcp );
```

Powyższe zmienne umożliwiają „podczepienie” procedur wyprowadzających częściowe wyniki konstrukcji powierzchni wypełniającej. Ich wartość początkowa jest równa `NULL`. Przypisanie dowolnej z tych zmiennych odpowiedniej procedurze przed wywołaniem `FillG1Hole` powoduje wywołanie tej procedury; może ona wyprowadzić dane lub narysować na ich podstawie obrazek.

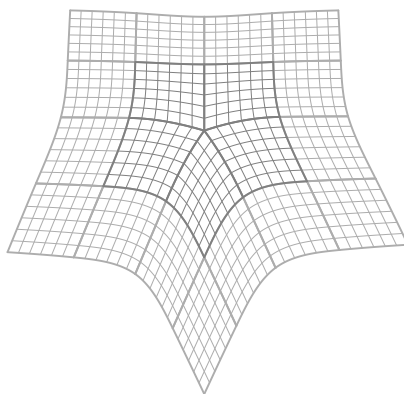
Procedura wskazywana przez zmienną `G1OutCentralPointf` otrzymuje jako parametr wskaźnik punktu „środkowego” (tj. wspólnego narożnika k płatów wypełniających otwór). Procedura ta ma prawo zmienić ten punkt (tj. przypisać mu nowe współrzędne), ponieważ taka ingerencja w konstrukcję jest dopuszczalna i w pewnych sytuacjach potrzebna.

Pozostałe procedury nie mogą zmieniać wartości zmiennych wskazywanych przez parametry, z którymi zostały wywołane. Procedura wskazywana przez zmienną `G1OutAuxCurvesf` jest wywoływana z parametrami opisującymi tzw. krzywe pomocnicze w konstrukcji — krzywe Béziera stopnia `degree` (w tej implementacji krzywe te są stopnia 3). W każdym wywołaniu jest podawana jedna krzywa.

Procedura wskazywana przez zmienną `G1OutStarCurvesf` jest wywoływana z parametrami opisującymi krzywe brzegowe płatów wypełniających otwór (konstrukcja tych krzywych jest jednym z pierwszych etapów konstrukcji). Parametry procedury opisują reprezentację Béziera stopnia 3 tych krzywych. W każdym wywołaniu jest podawana jedna krzywa.

Procedura wskazywana przez zmienną `G1OutAuxPatchesf` jest wywoływana z parametrami opisującymi tzw. płyty pomocnicze, które określają płaszczyzny styczne do konstruowanej powierzchni we wszystkich punktach krzywej brzegowej. Płyty te mają stopień (3, 1) (parametry `ndegu` i `ndegv` mają takie wartości). Parametr `apcp` wskazuje tablicę punktów kontrolnych. W każdym wywołaniu przekazywana jest reprezentacja jednego płyta pomocniczego.

**Uwaga:** Obecna wersja procedury nie zawiera obsługi sytuacji wyjątkowych, które umożliwiają „bezpieczny powrót” w razie niepowodzenia konstrukcji. Opracowanie takiej obsługi jest konieczne przed wbudowaniem procedury do systemu „produk-



Rys. 16.2. Powierzchnia z otworem wypełnionym przez procedurę `FillG1Holef`

cyjnego”, tj. nadającego się do użycia w celu projektowania przemysłowego. Co więcej, większość procedur w bibliotekach opisanych wcześniej nie ma obsługi błędów (w razie wystąpienia błędu jest wywoływana procedura `exit`), a zatem obecnie cały pakiet nadaje się tylko do celów eksperymentalnych (i zresztą po to go zacząłem pisać).