SISL

The SINTEF Spline Library

Reference Manual

(version 4.5)

SINTEF Digital, Mathematics and Cybernetics

January 18, 2021

Contents

1 Preface **1**
1.1 The structure of this document 1
1.2 The structure of the software package 2
1.3 Licensing information 2

2 General Introduction **3**
2.1 C Syntax Used in Manual 4
2.2 Dynamic Allocation in SISL 4
2.3 Creating the library 5
2.4 An Example Program 6
2.5 B-spline Curves 9
2.5.1 B-splines 10
2.5.2 The Control Polygon 12
2.5.3 The Knot Vector 12
2.5.4 NURBS Curves 13
2.6 B-spline Surfaces 14
2.6.1 The Basis Functions 15
2.6.2 NURBS Surfaces 16

3 Curve Definition **17**
3.1 Interpolation 17
3.1.1 Compute a curve interpolating a straight line between two points. 17
3.1.2 Compute a curve interpolating a set of points, automatic parameterization. 19
3.1.3 Compute a curve interpolating a set of points, parameterization as input. 21
3.1.4 Compute a curve by Hermite interpolation, automatic parameterization. 24
3.1.5 Compute a curve by Hermite interpolation, parameterization as input. 26
3.1.6 Compute a fillet curve based on parameter value. 28
3.1.7 Compute a fillet curve based on points. 30
3.1.8 Compute a fillet curve based on radius. 32
3.1.9 Compute a circular fillet between a 2D curve and a circle. 35
3.1.10 Compute a circular fillet between two 2D curves. 37
3.1.11 Compute a circular fillet between a 2D curve and a 2D line. 39
3.1.12 Compute a blending curve between two curves. 41

3.2 Approximation 43
3.2.1 Approximate a circular arc with a curve. 43
3.2.2 Approximate a conic arc with a curve. 45
3.2.3 Compute a curve using the input points as controlling vertices, automatic parameterization. 47
3.2.4 Approximate the offset of a curve with a curve. 49
3.2.5 Approximate a curve with a sequence of straight lines. . . 51
3.3 Mirror a Curve 52
3.4 Conversion 53
3.4.1 Convert a curve of order up to four, to a sequence of cubic polynomials. 53
3.4.2 Convert a curve to a sequence of Bezier curves. 54
3.4.3 Pick out the next Bezier curve from a curve. 55
3.4.4 Express a curve using a higher order basis. 57
3.4.5 Express the "i"-th derivative of an open curve as a curve. 58
3.4.6 Express a 2D or 3D ellipse as a curve. 59
3.4.7 Express a conic arc as a curve. 61
3.4.8 Express a truncated helix as a curve. 63

4 Curve Interrogation 65
4.1 Intersections 65
4.1.1 Intersection between a curve and a point. 65
4.1.2 Intersection between a spline curve and a straight line or a plane. 67
4.1.3 Convert a curve/line intersection into a two-dimensional curve/origo intersection 69
4.1.4 Intersection between a spline curve and a 2D circle or a sphere. 70
4.1.5 Intersection between a curve and a quadric curve. 72
4.1.6 Intersection between two curves. 74
4.2 Compute the Length of a Curve 76
4.3 Check if a Curve is Closed 77
4.4 Check if a Curve is Degenerated. 78
4.5 Pick the Parameter Range of a Curve 79
4.6 Closest Points 80
4.6.1 Find the closest point between a curve and a point. . . . 80
4.6.2 Find the closest point between a curve and a point. Simple version. 82
4.6.3 Local iteration to closest point between point and curve. . 84
4.6.4 Find the closest points between two curves. 86
4.6.5 Find a point on a 2D curve along a given direction. 88
4.7 Find the Absolute Extremals of a Curve. 89
4.8 Area between Curve and Point 91
4.8.1 Calculate the area between a 2D curve and a 2D point. . 91
4.8.2 Calculate the weight point and rotational momentum of an area between a 2D curve and a 2D point. 92
4.9 Bounding Box 94
4.9.1 Bounding box object. 94
4.9.2 Create and initialize a curve/surface bounding box instance. 95
4.9.3 Find the bounding box of a curve. 96

4.10 Normal Cone 97
4.10.1 Normal cone object. 97
4.10.2 Create and initialize a curve/surface direction instance. . 98
4.10.3 Find the direction cone of a curve. 99

5 Curve Analysis 100
5.1 Curvature Evaluation 100
5.1.1 Evaluate the curvature of a curve at given parameter values. 100
5.1.2 Evaluate the torsion of a curve at given parameter values. 102
5.1.3 Evaluate the Variation of Curvature (VoC) of a curve at given parameter values. 103
5.1.4 Evaluate the Frenet Frame (t,n,b) of a curve at given parameter values. 104
5.1.5 Evaluate geometric properties at given parameter values. 105

6 Curve Utilities 107
6.1 Curve Object 107
6.1.1 Create new curve object. 109
6.1.2 Make a copy of a curve. 111
6.1.3 Delete a curve object. 112
6.2 Evaluation 113
6.2.1 Compute the position and the left-hand derivatives of a curve at a given parameter value. 113
6.2.2 Compute the position and the right-hand derivatives of a curve at a given parameter value. 115
6.2.3 Evaluate position, first derivative, curvature and radius of curvature of a curve at a given parameter value, from the left hand side. 117
6.2.4 Evaluate position, first derivative, curvature and radius of curvature of a curve at a given parameter value, from the right hand side. 119
6.2.5 Evaluate the curve over a grid of m points. Only positions are evaluated. 121
6.3 Subdivision 121
6.3.1 Subdivide a curve at a given parameter value. 121
6.3.2 Insert a given knot into the description of a curve. 124
6.3.3 Insert a given set of knots into the description of a curve. 125
6.3.4 Split a curve into two new curves. 126
6.3.5 Pick a part of a curve. 127
6.3.6 Pick a part of a closed curve. 128
6.4 Joining 129
6.4.1 Join two curves at specified ends. 129
6.4.2 Join two curves at closest ends. 131
6.5 Reverse the Orientation of a Curve. 132
6.6 Extend a B-spline Curve. 133

7 Surface Definition 135
7.1 Interpolation 135
7.1.1 Compute a surface interpolating a set of points, automatic parameterization. 135

7.1.2 Compute a surface interpolating a set of points, parameterization as input. 138
7.1.3 Compute a surface interpolating a set of points, derivatives as input. 141
7.1.4 Compute a surface interpolating a set of points, derivatives and parameterization as input. 144
7.1.5 Compute a surface by Hermite interpolation, automatic parameterization. 147
7.1.6 Compute a surface by Hermite interpolation, parameterization as input. 149
7.1.7 Create a lofted surface from a set of B-spline input curves. 151
7.1.8 Create a lofted surface from a set of B-spline input curves and parametrization. 153
7.1.9 Create a rational lofted surface from a set of rational input-curves 155
7.1.10 Compute a rectangular blending surface from a set of B-spline input curves. 156
7.1.11 Compute a first derivative continuous blending surface set, over a 3-, 4-, 5- or 6-sided region in space, from a set of B-spline input curves. 158
7.1.12 Compute a surface, representing a Gordon patch, from a set of B-spline input curves. 160
7.2 Approximation 162
7.2.1 Compute a surface using the input points as control vertices, automatic parameterization. 162
7.2.2 Compute a linear swept surface. 164
7.2.3 Compute a rotational swept surface. 165
7.2.4 Compute a surface approximating the offset of a surface. . 167
7.3 Mirror a Surface 169
7.4 Conversion 170
7.4.1 Convert a surface of order up to four to a mesh of Coons patches. 170
7.4.2 Convert a surface to a mesh of Bezier surfaces. 172
7.4.3 Pick the next Bezier surface from a surface. 173
7.4.4 Express a surface using a higher order basis. 175
7.4.5 Express the "i,j"-th derivative of an open surface as a surface. 176
7.4.6 Express the octants of a sphere as a surface. 177
7.4.7 Express a truncated cylinder as a surface. 179
7.4.8 Express the octants of a torus as a surface. 180
7.4.9 Express a truncated cone as a surface. 182

8 Surface Interrogation 184
8.1 Intersection Curves 184
8.1.1 Intersection curve object. 184
8.1.2 Create a new intersection curve object. 186
8.1.3 Delete an intersection curve object. 188
8.1.4 Free a list of intersection curves. 189
8.2 Find the Intersections 190

8.2.1 Intersection between a spline curve and a straight line or a plane. . . . 190
8.2.2 Intersection between a spline curve and a 2D circle or a sphere. . . . 192
8.2.3 Intersection between a spline curve and a cylinder. . . . 194
8.2.4 Intersection between a spline curve and a cone. . . . 196
8.2.5 Intersection between a spline curve and an elliptic cone. . 198
8.2.6 Intersection between a curve and a torus. . . . 200
8.2.7 Intersection between a surface and a point. . . . 202
8.2.8 Intersection between a spline surface and a straight line. . 204
8.2.9 Newton iteration on the intersection between a 3D NURBS surface and a line. . . . 206
8.2.10 Convert a surface/line intersection into a two-dimensional surface/origo intersection 208
8.2.11 Intersection between a spline surface and a circle. . . . 209
8.2.12 Intersection between a surface and a curve. . . . 211

8.3 Find the Topology of the Intersection 213
8.3.1 Find the topology for the intersections between a spline surface and a plane. . . . 213
8.3.2 Find the topology for the intersection between a spline surface and a sphere. . . . 215
8.3.3 Find the topology for the intersections between a spline surface and a cylinder. . . . 217
8.3.4 Find the topology for the intersections between a spline surface and a cone. . . . 219
8.3.5 Find the topology for the intersections between a spline surface and an elliptic cone. . . . 221
8.3.6 Find the topology for the intersections between a spline surface and a torus. . . . 223
8.3.7 Find the topology for the intersection between two spline surfaces. . . . 225

8.4 Find the Topology of a Silhouette 227
8.4.1 Find the topology of the silhouette curves of a spline surface, using parallel projection. . . . 227
8.4.2 Find the topology of the silhouette curves of a spline surface, using perspective projection. . . . 229
8.4.3 Find the topology of the circular silhouette curves of a spline surface. . . . 231

8.5 Marching 233
8.5.1 March an intersection curve between a spline surface and a plane. . . . 233
8.5.2 March an intersection curve between a spline surface and a sphere. . . . 235
8.5.3 March an intersection curve between a spline surface and a cylinder. . . . 237
8.5.4 March an intersection curve between a spline surface and a cone. . . . 239
8.5.5 March an intersection curve between a surface and an elliptic cone. . . . 241

8.5.6 March an intersection curve between a spline surface and a torus. . . . 244
8.5.7 March an intersection curve between two spline surfaces. . 247
8.6 Marching of Silhouettes 249
8.6.1 March a silhouette curve of a surface, using parallel projection. 249
8.6.2 March a silhouette curve of a surface, using perspective projection. . . . 252
8.6.3 March a circular silhouette curve of a surface. . . . 254
8.7 Check if a Surface is Closed or has Degenerate Edges. . . . 256
8.8 Pick the Parameter Ranges of a Surface 258
8.9 Closest Points 259
8.9.1 Find the closest point between a surface and a point. . . . 259
8.9.2 Find the closest point between a surface and a point. Simple version. . . . 261
8.9.3 Local iteration to closest point bewteen point and surface. 263
8.10 Find the Absolute Extremals of a Surface. . . . 265
8.11 Bounding Box 267
8.11.1 Find the bounding box of a surface. . . . 267
8.12 Normal Cone 268
8.12.1 Find the direction cone of a surface. . . . 268

9 Surface Analysis 271
9.1 Curvature Evaluation 271
9.1.1 Gaussian curvature of a spline surface. . . . 271
9.1.2 Mean curvature of a spline surface. . . . 274
9.1.3 Absolute curvature of a spline surface. . . . 276
9.1.4 Total curvature of a spline surface. . . . 278
9.1.5 Second order Mehlum curvature of a spline surface. . . . 280
9.1.6 Third order Mehlum curvature of a spline surface. . . . 282
9.1.7 Gaussian curvature of a B-spline or NURBS surface as a NURBS surface. . . . 284
9.1.8 Mehlum curvature of a B-spline or NURBS surface as a NURBS surface. . . . 286
9.1.9 Curvature on a uniform grid of a NURBS surface. . . . 288
9.1.10 Principal curvatures of a spline surface. . . . 290
9.1.11 Normal curvature of a spline surface. . . . 292
9.1.12 Focal values on a uniform grid of a NURBS surface. . . . 294

10 Surface Utilities 296
10.1 Surface Object 296
10.1.1 Create a new surface object. . . . 298
10.1.2 Make a copy of a surface object. . . . 301
10.1.3 Delete a surface object. . . . 302
10.2 Evaluation 303
10.2.1 Compute the position, the derivatives and the normal of a surface at a given parameter value pair. . . . 303
10.2.2 Compute the position and derivatives of a surface at a given parameter value pair. . . . 305
10.2.3 Compute the position and the left- or right-hand derivatives of a surface at a given parameter value pair. . . . 307

10.2.4 Compute the position and the derivatives of a surface at a given parameter value pair. 310
10.2.5 Evaluate the surface pointed at by ps1 over an m1 * m2 grid of points (x[i],y[j]). Compute ider derivatives and normals if suitable. 314
10.3 Subdivision 316
10.3.1 Subdivide a surface along a given parameter line. 316
10.3.2 Insert a given set of knots, in each parameter direction, into the description of a surface. 317
10.4 Picking Curves from a Surface 319
10.4.1 Pick a curve along a constant parameter line in a surface. 319
10.4.2 Pick the curve lying in a surface, described by a curve in the parameter plane of the surface. 320
10.5 Pick a Part of a Surface. 322
10.6 Turn the Direction of the Surface Normal Vector. 323

11 Data Reduction **324**
11.1 Curves 324
11.1.1 Data reduction: B-spline curve as input. 324
11.1.2 Data reduction: Point data as input. 327
11.1.3 Data reduction: Points and tangents as input. 330
11.1.4 Degree reduction: B-spline curve as input. 332
11.2 Surfaces 334
11.2.1 Data reduction: B-spline surface as input. 334
11.2.2 Data reduction: Point data as input. 337
11.2.3 Data reduction: Points and tangents as input. 340
11.2.4 Degree reduction: B-spline surface as input. 343

12 Tutorial programs **345**
12.1 Compiling the programs 345
12.2 Description and commentaries on the sample programs 345
12.2.1 example01.C 346
12.2.2 example02.C 346
12.2.3 example03.C 346
12.2.4 example04.C 347
12.2.5 example05.C 347
12.2.6 example06.C 348
12.2.7 example07.C 348
12.2.8 example08.C 349
12.2.9 example09.C 349
12.2.10 example10.C 350
12.2.11 example11.C 350
12.2.12 example12.C 350
12.2.13 example13.C 351
12.2.14 example14.C 351
12.2.15 example15.C 352

13 The object viewer program 354
13.1 General . . . 354
13.2 Compiling the viewer . . . 354
13.3 Command line arguments . . . 355
13.4 User controls . . . 355
13.4.1 Mouse commands . . . 355
13.4.2 Keyboard commands . . . 356

14 Appendix: Error Codes 357

A GNU AFFERO GENERAL PUBLIC LICENSE 362

Chapter 1

Preface

Welcome to the SISL 4.5 user's manual. SISL stands for ***S**intef **S**pline **L**ibrary*, and has been gradually developed and enhanced for more than three decades by the geometry group at SINTEF in Oslo. Although it is very comprehensive, its organisation is simple. There are but a few structures, and its nearly four hundred main functions can usually be employed directly and individually. This manual organises and explains the main routines. However, much of this information can also be found directly in the code in the form of commentaries.

The complete software package you have in your hands should contain the following:

- The SISL 4.5 distribution and reference guide (the document you are reading now)
- Supplementary routines for writing SISL objects to streams (including file streams) in a simple ASCII format called `Go`.
- A selection of *sample programs*, designed to demonstrate functionalities and use of SISL.
- Source code for a simple *viewer* that can be used to view geometric objects stored in the `Go`-format. This allows visual inspection of SISL-generated curves and surfaces, as well as points.

1.1 The structure of this document

Chapter 2 is a general introduction to SISL and its programming style. A simple example program including instructions in how to compile and link the program and the expected output is provided. Since it is strongly recommended that the user has some general knowledge of splines, this chapter also contains a couple of sections introducing the subject of spline curves and surfaces.

Chapter 3 to 11 presents the main SISL routines.

Chapter 12 goes through the provided sample programs and explain what these do, and what the user can expect to learn from them. There are a total of 15 sample programs, ranging from very basic to intermediate complexity.

The goal of **Chapter 13** is to explain the use of the *viewer program*, which is a small but handy tool for visually inspecting results from SISL routines.

Chapter 14 is an appendix presenting an explanation of the error codes used in SISL.

Finally there is an **annex**, citing the text of the General Public License.

1.2 The structure of the software package

There are seven directories:

- `include/` - the inlude files related to the 4.5 release of SISL.
- `src/` - the source code of the 4.5 release of SISL.
- `doc/` - the basis for this document.
- `streaming/` - source code for the routines that can read and write SISL objects to a stream.
- `examples/` - sample programs making use of the SISL 4.5 source code.
- `viewer/` - source code for a viewer that can be used to view SISL objects saved in the `Go`-format.
- `app/` - the expected directory for test programs and applications. A couple of applications are provided including the example program described in Chapter 2.

Furthermore is the file CMakeLists.txt provided to facilitate building the library.

1.3 Licensing information

SISL is distibuted under the *GNU Affero General Public License* (aGPL). The license text is given in its entirety as an annex to this document. Commercial licenses are also available from SINTEF. You can contact Tor Dokken (tor.dokken@sintef.no) for more information.

Chapter 2

General Introduction

SISL is a geometric toolkit to model with curves and surfaces. It is a library of C functions to perform operations such as the definition, intersection and evaluation of NURBS (Non-Uniform Rational B-spline) geometries. Since many applications use implicit geometric representation such as planes, cylinders, tori etc., SISL can also handle the interaction between such geometries and NURBS.

Throughout this manual, a distinction is made between NURBS (the default) and B-splines. The term B-splines is used for non-uniform non-rational (or polynomial) B-splines. B-splines are used only where it does not make sense to employ NURBS (such as the approximation of a circle by a B-spline) or in cases where the research community has yet to develop stable technology for treating NURBS. A NURBS require more memory space than a B-spline, even when the extra degrees of freedom in a NURBS are not used. Therefore the routines are specified to give B-spline output whenever the extra degrees of freedom are not required.

Transferring a B-spline into NURBS format is done by constructing a new coefficient vector using the original B-spline coefficients and setting all the rational weights equal to one (1). This new coefficient vector is then given as input to the routine for creating a new curve/surface object while specifying that the object to be created should be of the NURBS (rational B-spline) type.

To approximate a NURBS by a B-spline, use the offset calculation routines with an offset of zero.

The routines in SISL are designed to function on curves and surfaces which are at least continuously differentiable. However many routines will also handle continuous curves and surfaces, including piecewise linear ones.

All arrays in SISL are 1-dimensional. In an array with points or vertices are the points stored consecutively. In a raster are points or vertices stored consecutively while points in the first parameter direction have the shortest stride (stored right after each other). There is a special rule for vertices given as input to a rational curve or surface, see the Sections 6.1.1 and 10.1.1.

The three important data structures used by SISL are SISLCurve, SISLSurf, and SISLIntcurve. These are defined in the Curve Utilities, Surface Utilities, and Surface Interrogation modules respectively. Other structures are SISLBox and SISLCone, which represents a bounding box and a normal cone, respectively. It is important to remember to always free these structures and also to free internally allocated structures used to pass results to the application, otherwise strange errors might result.

The various functions are equipped with a status variable, typically placed as the last entity in the parameter list. It returns information about whether or not the function succeeded in its purpose. A negative value means failure, the result zero means success while a positive number is a warning. Section 14 provides a list over possible error messages where most occurances are explained.

SISL is divided into seven modules, partly in order to provide a logical structure, but also to enable users with a specific application to use subsets of SISL. There are three modules dealing with curves, three with surfaces, and one module to perform data reduction on curves and surfaces. The modules for curves and surfaces focus on functions for creation and definition, intersection and interrogation, and general utilities.

The chapters 3 to 11 in this manual contain information concerning the top level functions of each module. Lower level functions not usually required by an application are not included. Each top level function is documented by describing the purpose, the input and output arguments and an example of use. Input parameters specified in the examples are suggestions, the actual values must be set dependent on context. To get you started, this chapter contains an Example Program.

2.1 C Syntax Used in Manual

This manual uses the K&R style C syntax for historic reasons, but both the ISO/ANSI and the K&R C standards are supported by the library and the include files.

2.2 Dynamic Allocation in SISL

In the description of all the functions in this manual, a convention exists on when to declare or allocate arrays/objects outside a function and when an array is allocated internally. *NB! When memory for output arrays/objects are allocated inside a function you must remember to free the allocated memory when it is not in use any more.*

The convention is the following:

- If [] is used in the synopsis and in the example it means that the array has to be declared or allocated outside the function.
- If $*$ is used it means that the function requires a pointer and that the allocation will be done outside the function if necessary.
- When either an array or an array of pointers or an object is to be allocated in a function, two or three stars are used in the synopsis. To use the function you declare the parameter with one star less and use & in the argument list.
- For all output variables except arrays or objects that are declared or allocated outside the function you have to use & in the argument list.

2.3 Creating the library

In order to access SISL from your program you need one library inclusion, namely the header file sisl.h. The statement

```
#include "sisl.h"
```

must be written at the top of your main program. In this header file all types are defined. It also contains all the SISL top level function declarations. Memory management and input/output require two more includes to avoid compiler warnings, see Section 2.4.

SISL is prepared for makefile generation with CMake and equipped with a CMakeLists.txt file. For information on using CMake, see www.cmake.org. The building procedure depends on whether your platform is Linux or Windows.

LINUX

Start by creating a build directory:

```
$ cd <path_to_source_code>
$ mkdir build
$ cd build
```

Run the cmake program to setup the build process, selecting Debug or Release as build type, optionally selecting a local install folder:

```
$ cmake .. -DCMAKE_BUILD_TYPE=Release (-DCMAKE_INSTALL_PREFIX=$HOME/install)
```

For a gui-like cmake interface use ccmake (from cmake-ncurses-gui) or cmake-gui (from cmake.org).

Build the library:

```
$ make
```

This will install the library in the build folder. Compilation and build of one particular example program is done by a specific make statement:

```
$ make example01
```

This option requires compilation of examples to be set in the Makefile.

Install the library to a local folder (requires the use of -DCMAKE_INSTALL_PREFIX with a local folder in the previous step):

```
$ make install
```

If the -DCMAKE_INSTALL_PREFIX in the cmake step was omitted or was set to a system folder (like /usr/local) the user needs elevated privileges to install the library:

```
$ sudo make install
```

Windows

Add a new build folder somewhere. Start the CMake executable and fill in the paths to the source and build folders. When you run CMake, a Visual Studio project solution file will be generated in the build folder.

2.4 An Example Program

To clarify the previous section here is an example program designed to test the SISL algorithm for intersecting a cone with a B-spline curve. The program calls the SISL routines newCurve() documented in Section 6.1.1, freeCurve() documented in 6.1.3, s1373() found in Section 8.2.4 and freeIntcrvlist() in 8.1.4.

```
#include "sisl.h"
#include <stdlib.h>
#include <stdio.h>

int main()
{
  SISLCurve *pc=0;                      /* Pointer to spline curve */
  double aepsco,aepsge;                 /* Tolerances */
  double top[3],axispt[3],conept[3];  /* Representating the cone */
  double st[100],scoef[100];            /* Knot vector and coefficients of spline curve */
  double *spar;                         /* Parameter values of intersection points */
  int kstat;                            /* Return status from function calls */
  int cone_exists=0;
  int kk,kn,kdim;                       /* Order (polynomial degree+1), number of
                                           coefficients and spatial dimension */
  int ki;                               /* Counter */
  int kpt,kcrv;                         /* Number of intersection points and curves */
  SISLIntcurve **qrcrv;                 /* Array of pointer to intersection curves  */
  char ksvar[100];
  kdim=3;
  aepsge=0.001; /* Geometric tolerance */
  aepsco=0.000001; /* Computational tolerance. This parameter is included from
                      historical reasons and no longer used */

  ksvar[0] = '0';  /* Arbitrary character */
  while (ksvar[0] != 'q')
    {
      printf("\n cu - define a new B-spline curve");
      printf("\n co - define a new cone");
      printf("\n i - intersect the B-spline curve with the cone");
      printf("\n q - quit");
      printf("\n> ");
      scanf("%s",ksvar);

      if (ksvar[0] == 'c' && ksvar[1] == 'u')
        {
          /* Define spline curve */
          printf("\n Give number of vertices, order of curve: ");
          scanf("%d %d", &kn, &kk);
          printf("Give knots values in ascending order: \n");
          for (ki=0; ki<kn+kk; ki++)
            {
               scanf("%lf",&st[ki]);
```

```
        }
      printf("Give vertices \n");
      for (ki=0; ki<kn*kdim; ki++)
        {
          scanf("%lf",&scoef[ki]);
        }
     if(pc) freeCurve(pc);

      /* Create curve */
      pc = newCurve(kn,kk,st,scoef,1,kdim,1);
    }
  else if (ksvar[0] == 'c' && ksvar[1] == 'o')
   {
     printf("\n Give top point: ");
     scanf("%lf %lf %lf",&top[0],&top[1],&top[2]);
     printf("\n Give a point on the axis: ");
     scanf("%lf %lf %lf",&axispt[0],&axispt[1],&axispt[2]);
     printf("\n Give a point on the cone surface: ");
     scanf("%lf %lf %lf",&conept[0],&conept[1],&conept[2]);
     cone_exists=1;
   }
  else if (ksvar[0] == 'i' && cone_exists && pc)
   {
     /* Intersect spline curve with cone */
     s1373(pc,top,axispt,conept,kdim,aepsco,aepsge,
           &kpt,&spar,&kcrv,&qrcrv,&kstat);
     printf("\n kstat %d",kstat);
     printf("\n kpt %d",kpt);
     printf("\n kcrv %d",kcrv);
     for (ki=0;ki<kpt;ki++)
      {
        printf("\nIntersection point %lf",spar[ki]);
      }
    if (spar)
      {
        /* The array containing parameter values of the intersection points between
           the curve and the cone is allocated inside s1373 and must be freed */
        free (spar);
        spar=0;
      }
    if (qrcrv)
     {
       /* The array containing pointers to intersection points curves between
          the curve and the cone is allocated inside s1373 and must be freed.
          This is done in a special function taking care of the intersection
          curves themselves */
      freeIntcrvlist(qrcrv,kcrv);
      qrcrv=0;
    }
  }
```

```
    }
  return 0;
}
```

Note that sisl.h is included. stdlib.h is included to declare free, which releases memory allocated in the function s1373. stdio.h declares printf and scanf.

The program was compiled and built using the command:

```
$ make prog1
```

Note that the program must be placed in the app folder and sisl_COMPILE_APPS must be set to true.

A sample run of prog1 went as follows:

```
$ prog1

     cu - define a new B-spline curve
     co - define a new cone
     i  - intersect the B-spline curve with the cone
     q  - quit
> cu

 Give number of vertices, order of curve: 2 2
Give knots values in ascending order:
0 0 1 1
Give vertices
1 0 0.5
-1 0 0.5

     cu - define a new B-spline curve
     co - define a new cone
     i  - intersect the B-spline curve with the cone
     q  - quit
> co

 Give top point: 0 0 1

 Give a point on the axis: 0 0 0

 Give a point on the cone surface: 1 0 0

     cu - define a new B-spline curve
     co - define a new cone
     i  - intersect the B-spline curve with the cone
     q  - quit
> i

 kstat 0
 kpt   2
```

```
 kcrv  0
Intersection point 0.250000
Intersection point 0.750000
     cu - define a new B-spline curve
     co - define a new cone
     i  - intersect the B-spline curve with the cone
     q  - quit
> q
$
```

SISL found two intersection points given by the parameters 0.25 and 0.75. These parameters correspond to the 3D points $(-0.5, 0, 0.5)$ and $(0.5, 0, 0.5)$ (which could be found by calling the evaluation routine s1221()). They lie on both the B-spline curve and the cone — as expected!

2.5 B-spline Curves

This section is optional reading for those who want to become acquainted with some of the mathematics of B-splines curves. For a description of the data structure for B-spline curves in SISL, see section 6.1.

A B-spline curve is defined by the formula

$$\mathbf{c}(t) = \sum_{i=1}^{n} \mathbf{p}_i B_{i,k,\mathbf{t}}(t).$$

The dimension of the curve $\mathbf{c}$ is equal to that of its *control points* $\mathbf{p}_i$. For example, if the dimension of the control points is one, the curve is a function, if the dimension is two, the curve is planar, and if the dimension is three, the curve is spatial. SISL also allows higher dimensions.

Thus, a B-spline curve is a linear combination of a sequence of B-splines $B_{i,k,\mathbf{t}}$ (called a B-basis) uniquely determined by a knot vector $\mathbf{t}$ and the order k. Order is equivalent to polynomial degree plus one. For example, if the order is two, the degree is one and the B-splines and the curve c they generate are (piecewise) linear. If the order is three, the degree is two and the B-splines and the curve are quadratic. Cubic B-splines and curves have order 4 and degree 3, etc.

The parameter range of a B-spline curve $\mathbf{c}$ is the interval

$$[t_k, t_{n+1}],$$

and so mathematically, the curve is a mapping $\mathbf{c} : [t_k, t_{n+1}] \to \mathbb{R}^d$, where d is the Euclidean space dimension of its control points.

The complete representation of a B-spline curve consists of

dim : The dimension of the underlying Euclidean space, $1, 2, 3, \ldots$.

n : The number of vertices (also the number of B-splines)

k : The order (degree plus one) of the B-splines.

$\mathbf{t}$: The knot vector of the B-splines. $\mathbf{t} = (t_1, t_2, \ldots, t_{n+k})$.

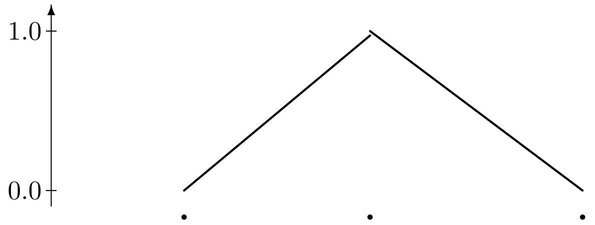


Figure 2.1: A linear B-spline (order 2) defined by three knots.

p : The control points of the B-spline curve. $p_{d,i}$, $d = 1, \ldots, dim$, $i = 1, \ldots, n$. e.g. when $dim = 3$, we have $\mathbf{p} = (x_1, y_1, z_1, x_2, y_2, z_2, \ldots, x_n, y_n, z_n)$.

We note that arrays in c start at index 0 which means, for example, that if the array t holds the knot vector, then $t[0] = t_1, \ldots, t[n + k - 1] = t_{n+k}$ and the parameter interval goes from $t[k - 1]$ to $t[n]$. Similar considerations apply to the other arrays.

The data in the representation must satisfy certain conditions:

- The knot vector must be non-decreasing: $t_i \le t_{i+1}$. Moreover, two knots t_i and t_{i+k} must be distinct: $t_i < t_{i+k}$.
- The number of vertices should be greater than or equal to the order of the curve: $n \ge k$.
- There should be k equal knots at the beginning and at the end of the knot vector; that is the knot vector $\mathbf{t}$ must satisfy the conditions $t_1 = t_2 = \ldots = t_k$ and $t_{n+1} = t_{n+2} = \ldots = t_{n+k}$.

To understand the representation better, we will look at three parts of the representation: the B-splines (the basis functions), the knot vector and the control polygon.

2.5.1 B-splines

A set of B-splines is determined by the order k and the knots. For example, to define a single B-spline of degree one, we need three knots. In figure 2.1 the three knots are marked as dots. Knots can also be equal as shown in figure 2.2. By taking a linear combination of the three types of B-splines shown in figures 2.1 and 2.2 we can generate a linear spline function as shown in figure 2.3.

A quadratic B-spline is a linear combination of two linear B-splines. Shown in figure 2.4 is a quadratic B-spline defined by four knots. A quadratic B-spline is the sum of two products, the first product between the linear B-spline on the left and a corresponding line from 0 to 1, the second product between the linear B-spline on the right and a corresponding line from 1 to 0; see figure 2.4. For higher degree B-splines there is a similar definition. A B-spline of order k is the sum of two B-splines of order $k - 1$, each weighted with weights in the interval [0,1]. In fact we define B-splines of order 1 explicitly as box functions,

$$B_{i,1}(t) = \begin{cases} 1 & \text{if } t_i \le t < t_{i+1}; \\ 0 & \text{otherwise,} \end{cases}$$

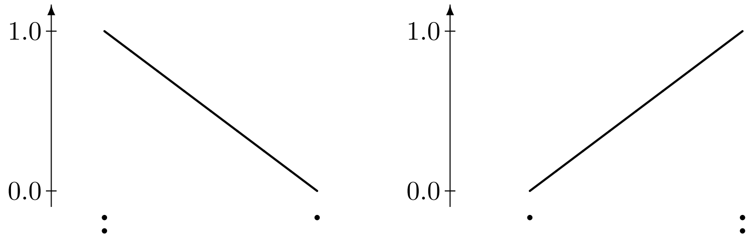


Figure 2.2: Linear B-splines of with multiple knots at one end.

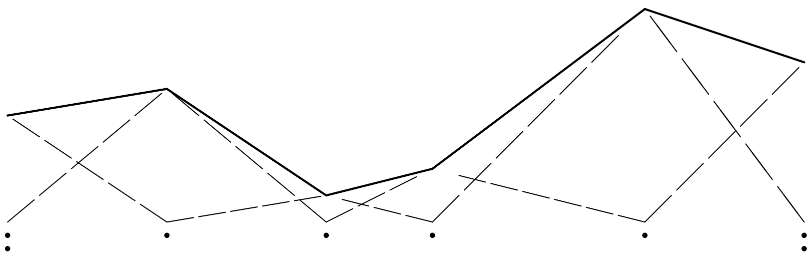


Figure 2.3: A B-spline curve of dimension 1 as a linear combination of a sequence of B-splines. Each B-spline (dashed) is scaled by a coefficient.

and then the complete definition of a k-th order B-spline is

$$B_{i,k}(t) = \frac{t - t_i}{t_{i+k-1} - t_i} B_{i,k-1}(t) + \frac{t_{i+k} - t}{t_{i+k} - t_{i+1}} B_{i-1,k-1}(t).$$

B-splines satisfy some important properties for curve and surface design. Each B-spline is non-negative and it can be shown that they sum to one,

$$\sum_{i=1}^{n} B_{i,k,\mathbf{t}}(t) = 1.$$

These properties combined mean that B-spline curves satisfy the *convex hull property*: the curve lies in the convex hull of its control points. Furthermore, the support of the B-spline $B_{i,k,\mathbf{t}}$ is the interval $[t_i, t_{i+k}]$ which means that B-spline curves has *local control*: moving one control point only alters the curve locally.

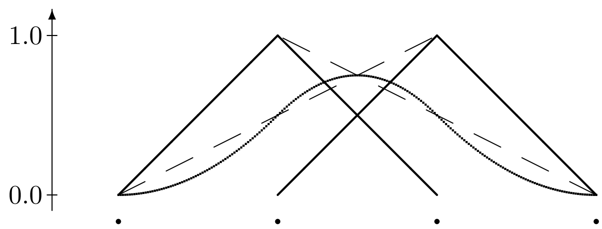


Figure 2.4: A quadratic B-spline, the two linear B-splines and the corresponding lines (dashed) in the quadratic B-spline definition.

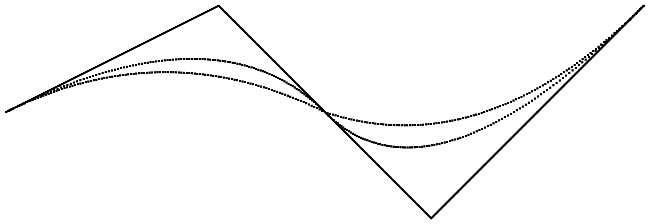


Figure 2.5: Linear, quadratic, and cubic B-spline curves sharing the same control polygon. The control polygon is equal to the linear B-spline curve. The curves are planar, i.e. the space dimension is two.

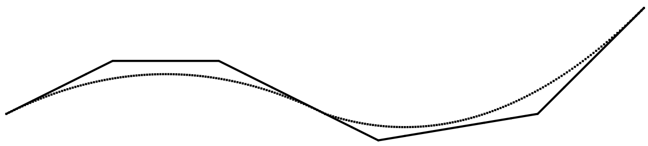


Figure 2.6: The cubic B-spline curve with a redefined knot vector.

Due to the demand of k multiple knots at the ends of the knot vector, B-spline curves in SISL also have the *endpoint property*: the start point of the B-spline curve equals the first control point and the end point equals the last control point, in other words

$$\mathbf{c}(t_k) = \mathbf{p}_1 \qquad \text{and} \qquad \mathbf{c}(t_{n+1}) = \mathbf{p}_n.$$

2.5.2 The Control Polygon

The control points $\mathbf{p}_i$ define the vertices The *control polygon* of a B-spline curve is the polygonal arc formed by its control points, $\mathbf{p}_0, \mathbf{p}_1, \ldots, \mathbf{p}_n$. This means that the control polygon, regarded as a parametric curve, is itself piecewise linear B-spline curve (order two). If we increase the order, the distance between the control polygon and the curve increases (see figure 2.5). A higher order B-spline curve tends to smooth the control polygon and at the same time mimic its shape. For example, if the control polygon is convex, so is the B-spline curve.

Another property of the control polygon is that it will get closer to the curve if it is redefined by inserting knots into the curve and thereby increasing the number of vertices; see figure 2.6. If the refinement is infinite then the control polygon converges to the curve.

2.5.3 The Knot Vector

The knots of a B-spline curve describe the following properties of the curve:

- The parameterization of the B-spline curve

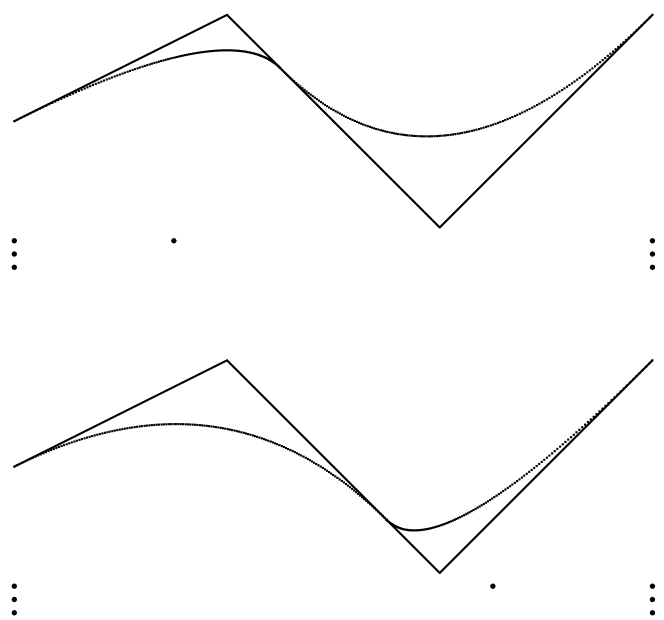


Figure 2.7: Two quadratic B-spline curves with the same control polygon but different knot vectors. The curves and the control polygons are two-dimensional.

- The continuity at the joins between the adjacent polynomial segments of the B-spline curve.

In figure 2.7 we have two curves with the same control polygon and order but with different parameterization.

This example is not meant as an encouragement to use parameterization for modelling, rather to make users aware of the effect of parameterization. Something close to curve length parameterization is in most cases preferable. For interpolation, chord-length parameterization is used in most cases.

The number of equal knots determines the degree of continuity. If k consecutive internal knots are equal, the curve is discontinuous. Similarly if $k-1$ consecutive internal knots are equal, the curve is continuous but not in general differentiable. A continuously differentiable curve with a discontinuity in the second derivative can be modelled using $k-2$ equal knots etc. (see figure 2.8). Normally, B-spline curves in SISL are expected to be continuous. For intersection algorithms, curves are usually expected to be continuously differentiable (C^1).

2.5.4 NURBS Curves

A NURBS (Non-Uniform Rational B-Spline) curve is a generalization of a B-spline curve,

$$\mathbf{c}(t) = \frac{\sum_{i=1}^{n} w_i \mathbf{p}_i B_{i,k,\mathbf{t}}(t)}{\sum_{i=1}^{n} w_i B_{i,k,\mathbf{t}}(t)}.$$

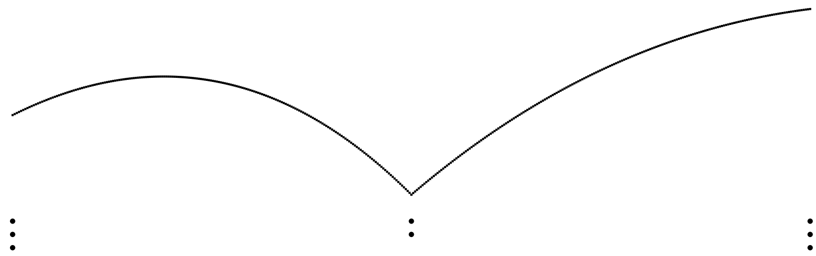


Figure 2.8: A quadratic B-spline curve with two equal internal knots.

In addition to the data of a B-spline curve, the NURBS curve $\mathbf{c}$ has a sequence of weights $w_1, \ldots, w_n$. One of the advantages of NURBS curves over B-spline curves is that they can be used to represent conic sections exactly (taking the order k to be three). A disadvantage is that NURBS curves depend nonlinearly on their weights, making some calculations, like the evaluation of derivatives, more complicated and less efficient than with B-spline curves.

The representation of a NURBS curve is the same as for a B-spline except that it also includes

$\mathbf{w}$: A sequence of weights $\mathbf{w} = (w_1, w_2, \ldots, w_n)$.

In SISL we make the assumption that

- The weights are (strictly) positive: $w_i > 0$.

Under this condition, a NURBS curve, like its B-spline cousin, enjoys the convex hull property. Due to k-fold knots at the ends of the knot vector, NURBS curves in SISL alos have the endpoint

2.6 B-spline Surfaces

This section is optional reading for those who want to become acquainted with some of the mathematics of tensor-product B-splines surfaces. For a description of the data structure for B-spline surfaces in SISL, see section 10.1.

A tensor product B-spline surface is defined as

$$\mathbf{s}(u, v) = \sum_{i=1}^{n_1} \sum_{j=1}^{n_2} \mathbf{p}_{i,j} B_{i,k_1,\mathbf{u}}(u) B_{j,k_2,\mathbf{v}}(v)$$

with control points $\mathbf{p}_{i,j}$ and two variables (or parameters) u and v. The formula shows that a basis function of a B-spline surface is a product of two basis functions of B-spline curves (B-splines). This is why a B-spline surface is called a tensor-product surface. The following is a list of the components of the representation:

dim : The dimension of the underlying Euclidean space.

n_1 : The number of vertices with respect to the first parameter.

n_1 : The number of vertices with respect to the second parameter.

Figure 2.9: A B-spline surface and its control net. The surface is drawn using isocurves. The dimension is 3.

k_1 : The order of the B-splines in the first parameter.

k_2 : The order of the B-splines in the second parameter.

$\mathbf{u}$: The knot vector of the B-splines with respect to the first parameter, $\mathbf{u} = (u_1, u_2, \ldots, u_{n_1+k_1})$.

$\mathbf{v}$: The knot vector of the B-splines with respect to the second parameter, $\mathbf{v} = (v_1, v_2, \ldots, v_{n_2+k_2})$.

$\mathbf{p}$: The control points of the B-spline surface, $c_{d,i,j}$, $d = 1, \ldots, dim$, $i = 1, \ldots, n_1$, $j = 1, \ldots, n_2$. When $dim = 3$, we have $\mathbf{p} = (x_{1,1}, y_{1,1}, z_{1,1}, x_{2,1}, y_{2,1}, z_{2,1}, \ldots, x_{n_1,1}, y_{n_1,1}, z_{n_1,1}, \ldots, x_{n_1,n_2}, y_{n_1,n_2}, z_{n_1,n_2})$.

The data of the B-spline surface must fulfill the following requirements:

- Both knot vectors must be non-decreasing.
- The number of vertices must be greater than or equal to the order with respect to both parameters: $n_1 \geq k_1$ and $n_2 \geq k_2$.
- There should be k_1 equal knots at the beginning and end of knot vector $\mathbf{u}$ and k_2 equal knots at the beginning and end of knot vector $\mathbf{v}$.

The properties of the representation of a B-spline surface are similar to the properties of the representation of a B-spline curve. The control points $\mathbf{p}_{i,j}$ form a *control net* as shown in figure 2.9. The control net has similar properties to the control polygon of a B-spline curve, described in section 2.5.2. A B-spline surface has two knot vectors, one for each parameter. In figure 2.9 we can see *isocurves*, surface curves defined by fixing the value of one of the parameters.

2.6.1 The Basis Functions

A basis function of a B-spline surface is the product of two basis functions of two B-spline curves,

$$B_{i,k_1,\mathbf{u}}(u) B_{j,k_2,\mathbf{v}}(v).$$

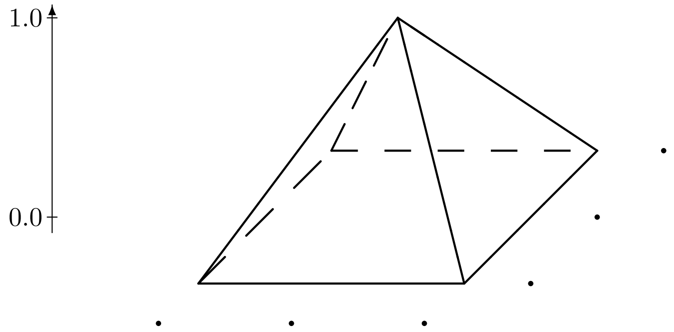


Figure 2.10: A basis function of degree one in both variables.

Its support is the rectangle $[u_i, u_{i+k_1}] \times [v_j, v_{j+k_2}]$. If the basis functions in both directions are of degree one and all knots have multiplicity one, then the surface basis functions are pyramid-shaped (see figure 2.10). For higher degrees, the surface basis functions are bell shaped.

2.6.2 NURBS Surfaces

A NURBS (Non-Uniform Rational B-Spline) surface is a generalization of a B-spline surface,

$$\mathbf{s}(u,v) = \frac{\sum_{i=1}^{n_1} \sum_{j=1}^{n_2} w_{i,j} \mathbf{p}_{i,j} B_{i,k_1,\mathbf{u}}(u) B_{j,k_2,\mathbf{v}}(v)}{\sum_{i=1}^{n_1} \sum_{j=1}^{n_2} w_{i,j} B_{i,k_1,\mathbf{u}}(u) B_{j,k_2,\mathbf{v}}(v)}.$$

In addition to the data of a B-spline surface, the NURBS surface has a weights $w_{i,j}$. NURBS surfaces can be used to exactly represent several common 'analytic' surfaces such as spheres, cylinders, tori, and cones. A disadvantage is that NURBS surfaces depend nonlinearly on their weights, making some calculations, like with NURBS curves, less efficient.

The representation of a NURBS surface is the same as for a B-spline except that it also includes

w : The weights of the NURBS surface, $w_{i,j}$, $i = 1, \ldots, n_1$, $j = 1, \ldots, n_2$, so $\mathbf{w} = (w_{1,1}, w_{2,1}, \ldots, w_{n_1,1}, \ldots, w_{1,2}, \ldots, w_{n_1,n_2})$.

In SISL we make the assumption that

- The weights are (strictly) positive: $w_{i,j} > 0$.

Chapter 3

Curve Definition

This chapter describes all functions in the Curve Definition module.

3.1 Interpolation

In this section we treat different kinds of interpolation of points or points and derivatives (Hermite). In addition to the general functions there are functions to find fillet curves (a curve between two other curves), and blending curves (a curve between the end points of two other curves).

3.1.1 Compute a curve interpolating a straight line between two points.

NAME

s1602 - To make a straight line represented as a B-spline curve between two points.

SYNOPSIS

void s1602(*startpt*, *endpt*, *order*, *dim*, *startpar*, *endpar*, *curve*, *stat*)

double	*startpt*[ ];
double	*endpt*[ ];
int	*order*;
int	*dim*;
double	*startpar*;
double	**endpar*;
SISLCurve	***curve*;
int	**stat*;

ARGUMENTS

Input Arguments:

startpt	-	Start point of the straight line
endpt	-	End point of the straight line
order	-	The order of the curve to be made.
dim	-	The dimension of the geometric space
startpar	-	Start value of the parameterization of the curve

Output Arguments:

endpar	-	Parameter value used at the end of the curve
curve	-	Pointer to the B-spline curve
stat	-	Status messages > 0 : warning $= 0$: ok < 0 : error

EXAMPLE OF USE

```
{
    double      startpt[2];
    double      endpt[2];
    int         order=2; /* If a higher order is requested will a degree
                            one curve be constructed and degree raising
                            performed to reach the requested order */
    int         dim=2; /* Corresponds to the number of parameters
                            in startpt and endpt */
    double      startpar=0.0;
    double      endpar;
    SISLCurve   *curve=NULL;
    int         stat=0;
    ...
    s1602(startpt, endpt, order, dim, startpar, &endpar, &curve, &stat);
    ...
}
```

3.1.2 Compute a curve interpolating a set of points, automatic parameterization.

NAME

s1356 - Compute a curve interpolating a set of points. The points can be assigned a tangent (derivative). The parameterization of the curve will be generated and the curve can be open, closed non-periodic or periodic. If end-conditions are conflicting, the condition closed curve rules out other end conditions. The output will be represented as a B-spline curve.

SYNOPSIS

void s1356(*epoint*, *inbpnt*, *idim*, *nptyp*, *icnsta*, *icnend*, *iopen*, *ik*, *astpar*, *cendpar*, *rc*, *gpar*, *jnbpar*, *jstat*)

double	*epoint*[ ];
int	*inbpnt*;
int	*idim*;
int	*nptyp*[ ];
int	*icnsta*;
int	*icnend*;
int	*iopen*;
int	*ik*;
double	*astpar*;
double	**cendpar*;
SISLCurve	***rc*;
double	***gpar*;
int	**jnbpar*;
int	**jstat*;

ARGUMENTS

Input Arguments:

epoint	-	Array (of length *idim* × *inbpnt*) containing the points/-derivatives to be interpolated.
inbpnt	-	No. of points/derivatives in the *epoint* array.
idim	-	The dimension of the space in which the points lie.
nptyp	-	Array (length *inbpnt*) containing type indicator for points/derivatives/second-derivatives: = 1 : Ordinary point. = 2 : Knuckle point. (Is treated as an ordinary point.) = 3 : Derivative to next point. = 4 : Derivative to prior point. (= 5 : Second-derivative to next point.) (= 6 : Second derivative to prior point.) = 13 : Point of tangent to next point. = 14 : Point of tangent to prior point.

icnsta	-	Additional condition at the start of the curve: $= 0$: No additional condition. $= 1$: Zero curvature at start.
icnend	-	Additional condition at the end of the curve: $= 0$: No additional condition. $= 1$: Zero curvature at end.
iopen	-	Flag telling if the curve should be open or closed: $= 1$: Open curve. $= 0$: Closed, non-periodic curve. $= -1$: Periodic (and closed) curve.
ik	-	The order of the spline curve to be produced.
astpar	-	Parameter value to be used at the start of the curve.

Output Arguments:

cendpar	-	Parameter value used at the end of the curve.
rc	-	Pointer to output B-spline curve.
gpar	-	Pointer to the parameter values of the points in the curve. Represented only once, although derivatives and second-derivatives will have the same parameter value as the points.
jnbpar	-	No. of unique parameter values.
jstat	-	Status message < 0 : Error. $= 0$: Ok. > 0 : Warning.

EXAMPLE OF USE

```
{
    double      epoint[30];
    int         inbpnt = 10;
    int         idim = 3;
    int         nptyp[10];
    int         icnsta = 0;
    int         icnend = 0;
    int         iopen = 1;
    int         ik = 4;
    double      astpar = 0.0;
    double      cendpar = 0.0;
    SISLCurve   *rc = NULL;
    double      *gpar = NULL;
    int         jnbpar = 0;
    int         jstat = 0;
    ...
    s1356(epoint, inbpnt, idim, nptyp, icnsta, icnend, iopen, ik, astpar, &cend-
          par, &rc, &gpar, &jnbpar, &jstat);
    ...
}
```

3.1.3 Compute a curve interpolating a set of points, parameterization as input.

NAME

s1357 - Compute a curve interpolating a set of points. The points can be assigned a tangent (derivative). The curve can be open, closed or periodic. If end-conditions are conflicting, the condition closed curve rules out other end conditions. The parameterization is given by the array *epar*. The output will be represented as a B-spline curve.

SYNOPSIS

void s1357(*epoint*, *inbpnt*, *idim*, *ntype*, *epar*, *icnsta*, *icnend*, *iopen*, *ik*, *astpar*, *cendpar*, *rc*, *gpar*, *jnbpar*, *jstat*)

double	*epoint*[ ];
int	*inbpnt*;
int	*idim*;
int	*ntype*[ ];
double	*epar*[ ];
int	*icnsta*;
int	*icnend*;
int	*iopen*;
int	*ik*;
double	*astpar*;
double	**cendpar*;
SISLCurve	***rc*;
double	***gpar*;
int	**jnbpar*;
int	**jstat*;

ARGUMENTS

Input Arguments:

epoint	-	Array (length *idim* $\times$ *inbpnt*) containing the points/-derivatives to be interpolated.
inbpnt	-	No. of points/derivatives in the *epoint* array.
idim	-	The dimension of the space in which the points lie.
ntype	-	Array (length *inbpnt*) containing type indicator for points/derivatives/second-derivatives:

= 1	: Ordinary point.
= 2	: Knuckle point. (Is treated as an ordinary point.)
= 3	: Derivative to next point.
= 4	: Derivative to prior point.
(= 5	: Second-derivative to next point.)
(= 6	: Second derivative to prior point.)
= 13	: Point of tangent to next point.
= 14	: Point of tangent to prior point.

epar	-	Array containing the wanted parameterization. Only parameter values corresponding to position points are given. For closed curves, one additional parameter value must be specified. The last entry contains the parametrization of the repeated start point. (if the end point is equal to the start point of the interpolation the length of the array should be equal to inpt1 also in the closed case).
icnsta	-	Additional condition at the start of the curve: $= 0$: No additional condition. $= 1$: Zero curvature at start.
icnend	-	Additional condition at the end of the curve: $= 0$: No additional condition. $= 1$: Zero curvature at end.
iopen	-	Flag telling if the curve should be open or closed: $= 1$: The curve should be open. $= 0$: The curve should be closed. $= -1$: The curve should be closed and periodic.
ik	-	The order of the spline curve to be produced.
astpar	-	Parameter value to be used at the start of the curve.

Output Arguments:

cendpar	-	Parameter value used at the end of the curve.
rc	-	Pointer to the output B-spline curve.
gpar	-	Pointer to the parameter values of the points in the curve. Represented only once, although derivatives and second-derivatives will have the same parameter value as the points.
jnbpar	-	No, of unique parameter values.
jstat	-	Status message < 0 : Error. $= 0$: Ok. > 0 : Warning.

EXAMPLE OF USE

```
{
    double      epoint[30];
    int         inbpnt = 10;
    int         idim = 3;
    int         ntype[10];
    double      epar[10];
    int         icnsta = 0;
    int         icnend = 0;
    int         iopen = 0;
    int         ik = 4;
    double      astpar = 0.0;
    double      cendpar;
    SISLCurve   *rc = NULL;
    double      *gpar = NULL;
    int         jnbpar;
    int         jstat = 0;
    ...
    s1357(epoint, inbpnt, idim, ntype, epar, icnsta, icnend, iopen, ik, astpar,
          &cendpar, &rc, &gpar, &jnbpar, &jstat);
    ...
}
```

3.1.4 Compute a curve by Hermite interpolation, automatic parameterization.

NAME

s1380 - To compute the cubic Hermite interpolant to the data given by the points point and the derivatives derivate. The output is represented as a B-spline curve.

SYNOPSIS

void s1380(*point*, *derivate*, *numpt*, *dim*, *typepar*, *curve*, *stat*)

double	*point*[ ];
double	*derivate*[ ];
int	*numpt*;
int	*dim*;
int	*typepar*;
SISLCurve	***curve*;
int	**stat*;

ARGUMENTS

Input Arguments:

point	-	Array (length dim*numpt) containing the points in sequence $(x_0, y_0, x_1, y_1, \ldots)$ to be interpolated.
derivate	-	Array (length dim*numpt) containing the derivate in sequence $(\frac{dx_0}{dt}, \frac{dy_0}{dt}, \frac{dx_1}{dt}, \frac{dy_1}{dt}, \ldots)$ to be interpolated.
numpt	-	No. of points/derivatives in the point and derivative arrays.
dim	-	The dimension of the space in which the points lie.
typepar	-	Type of parameterization: $= 1$: Parameterization using cord length between the points. $\neq 1$: Uniform parameterization.

Output Arguments:

curve	-	Pointer to the output B-spline curve
stat	-	Status messages > 0 : warning $= 0$: ok < 0 : error

EXAMPLE OF USE

```
{
    double      point[10];
    double      derivate[10];
    int         numpt = 5;
    int         dim = 2;
    int         typepar = 1;
    SISLCurve   *curve = NULL;
    int         stat = 0;
    ...
    s1380(point, derivate, numpt, dim, typepar, &curve, &stat);
    ...
}
```

3.1.5 Compute a curve by Hermite interpolation, parameterization as input.

NAME

s1379 - To compute the cubic Hermite interpolant to the data given by the points point and the derivatives derivate and the parameterization par. The output is represented as a B-spline curve.

SYNOPSIS

void s1379(*point*, *derivate*, *par*, *numpt*, *dim*, *curve*, *stat*)

double	*point*[ ];
double	*derivate*[ ];
double	*par*[ ];
int	*numpt*;
int	*dim*;
SISLCurve	***curve*;
int	**stat*;

ARGUMENTS

Input Arguments:

point	-	Array (length dim*numpt) containing the points to be interpolated in the sequence is $(x_0, y_0, x_1, y_1, \ldots)$.
derivate	-	Array (length dim*numpt) containing the derivatives to be interpolated in the sequence is $$(\frac{dx_0}{dt}, \frac{dy_0}{dt}, \frac{dx_1}{dt}, \frac{dy_1}{dt}, \ldots).$$
par	-	Parameterization array, $(t_0, t_1, \ldots)$. The array should be increasing in value.
numpt	-	No. of points/derivatives in the point and derivative arrays.
dim	-	The dimension of the space in which the points lie.

Output Arguments:

curve	-	Pointer to output B-spline curve
stat	-	Status messages > 0 : warning $= 0$: ok < 0 : error

EXAMPLE OF USE

```
{
    double      point[10];
    double      derivate[10];
    double      par[5];
    int         numpt = 5;
    int         dim = 2;
    SISLCurve   *curve = NULL;
    int         stat = 0;
    ...
    s1379(point, derivate, par, numpt, dim, &curve, &stat);
    ...
}
```

3.1.6 Compute a fillet curve based on parameter value.

NAME

s1607 - To calculate a fillet curve between two curves. The start and end point for the fillet is given as one parameter value for each of the curves. The output is represented as a B-spline curve.

SYNOPSIS

void s1607(*curve1*, *curve2*, *epsge*, *end1*, *fillpar1*, *end2*, *fillpar2*, *filltype*, *dim*, *order*, *newcurve*, *stat*)

SISLCurve	**curve1*;
SISLCurve	**curve2*;
double	*epsge*;
double	*end1*;
double	*fillpar1*;
double	*end2*;
double	*fillpar2*;
int	*filltype*;
int	*dim*;
int	*order*;
SISLCurve	***newcurve*;
int	**stat*;

ARGUMENTS

Input Arguments:

curve1	-	The first input curve.
curve2	-	The second input curve.
epsge	-	Geometry resolution.
end1	-	Parameter value on the first curve. The parameter fillpar1 divides the first curve in two pieces. End1 is used to select which of these pieces the fillet should extend.
fillpar1	-	Parameter value of the start point of the fillet on the first curve.
end2	-	Parameter value on the second curve indicating that the part of the curve lying on this side of fillpar2 shall not be replaced by the fillet.
fillpar2	-	Parameter value of the start point of the fillet on the second curve.

filltype - Indicator of the type of fillet.
= 1 : Circle approximation, interpolating tangent on first curve, not on curve 2.
= 2 : Conic approximation if possible,
else : polynomial segment.
dim - Dimension of space.
order - Order of the fillet curve, which is not always used.

Output Arguments:
newcurve - Pointer to the B-spline fillet curve.
stat - Status messages
> 0 : warning
$= 0$: ok
< 0 : error

EXAMPLE OF USE

```
{
    SISLCurve   *curve1; /* Must be defined */
    SISLCurve   *curve2; /* Must be defined */
    double      epsge = 0.0001;
    double      end1;    /* Must be defined */
    double      fillpar1; /* Must be defined */
    double      end2;    /* Must be defined */
    double      fillpar2; /* Must be defined */
    int         filltype = 2;
    int         dim = 3;
    int         order = 4;
    SISLCurve   *newcurve = NULL;
    int         stat = 0;
    ...
    s1607(curve1, curve2, epsge, end1, fillpar1, end2, fillpar2, filltype, dim, order,
          &newcurve, &stat);
    ...
}
```

3.1.7 Compute a fillet curve based on points.

NAME

s1608 - To calculate a fillet curve between two curves. Points indicate between which points on the input curve the fillet is to be produced. The output is represented as a B-spline curve.

SYNOPSIS

void s1608(*curve1*, *curve2*, *epsge*, *point1*, *startpt1*, *point2*, *endpt2*, *filltype*, *dim*, *order*, *newcurve*, *parpt1*, *parspt1*, *parpt2*, *parept2*, *stat*)

SISLCurve	**curve1*;
SISLCurve	**curve2*;
double	*epsge*;
double	*point1*[ ];
double	*startpt1*[ ];
double	*point2*[ ];
double	*endpt2*[ ];
int	*filltype*;
int	*dim*;
int	*order*;
SISLCurve	***newcurve*;
double	**parpt1*;
double	**parspt1*;
double	**parpt2*;
double	**parept2*;
int	**stat*;

ARGUMENTS

Input Arguments:

curve1	-	The first input curve.
curve2	-	The second input curve.
epsge	-	Geometry resolution.
point1	-	Point close to curve 1 indicating that the part of the curve lying on this side of startpt1 is not to be replaced by the fillet.
startpt1	-	Point close to curve 1, indicating where the fillet is to start. The tangent at the start of the fillet will have the same orientation as the curve from point1 to startpt1.
point2	-	Point close to curve 2 indicating that the part of the curve lying on this side of endpt2 is not to be replaced by the fillet.
endpt2	-	Point close to curve two, indicating where the fillet is to end. The tangent at the end of the fillet will have the same orientation as the curve from endpt2 to point2.

filltype	-	Indicator of type of fillet. = 1 : Circle, interpolating tangent on first curve, not on curve 2. = 2 : Conic if possible, else : polynomial segment.
dim	-	Dimension of space.
order	-	Order of fillet curve, which is not always used.

Output Arguments:

newcurve	-	Pointer to the B-spline fillet curve.
parpt1	-	Parameter value of point *point1* on curve 1.
parspt1	-	Parameter value of point *startpt1* on curve 1.
parpt2	-	Parameter value of point *point2* on curve 2.
parept2	-	Parameter value of point *endpt2* on curve 2.
stat	-	Status messages > 0 : warning $= 0$: ok < 0 : error

EXAMPLE OF USE

```
{
    SISLCurve   *curve1; /* Must be defined */
    SISLCurve   *curve2; /* Must be defined */
    double      epsge = 0.0001;
    double      point1[3];   /* Must be defined */
    double      startpt1[3]; /* Must be defined */
    double      point2[3];   /* Must be defined */
    double      endpt2[3];   /* Must be defined */
    int         filltype = 3;
    int         dim = 3;
    int         order = 4;
    SISLCurve   *newcurve = NULL;
    double      parpt1;
    double      parspt1;
    double      parpt2;
    double      parept2;
    int         stat = 0;
    ...
    s1608(curve1, curve2, epsge, point1, startpt1, point2, endpt2,
        filltype, dim, order, &newcurve, &parpt1, &parspt1,
        &parpt2, &parept2, &stat);
    ...
}
```

3.1.8 Compute a fillet curve based on radius.

NAME

s1609 - To calculate a constant radius fillet curve between two curves if possible. The output is represented as a B-spline curve.

SYNOPSIS

void s1609(*curve1*, *curve2*, *epsge*, *point1*, *pointf*, *point2*, *radius*, *normal*, *filltype*, *dim*, *order*, *newcurve*, *parend1*, *parspt1*, *parend2*, *parept2*, *stat*)

SISLCurve	**curve1*;
SISLCurve	**curve2*;
double	*epsge*;
double	*point1*[ ];
double	*pointf*[ ];
double	*point2*[ ];
double	*radius*;
double	*normal*[ ];
int	*filltype*;
int	*dim*;
int	*order*;
SISLCurve	***newcurve*;
double	**parend1*;
double	**parspt1*;
double	**parend2*;
double	**parept2*;
int	**stat*;

ARGUMENTS

Input Arguments:

curve1	-	The first input curve.
curve2	-	The second input curve.
epsge	-	Geometry resolution.
point1	-	Point indicating that the fillet should be put on the side of *curve1* where *point1* is situated.
pointf	-	Point indicating where the fillet curve should go. *point1* together with *pointf* indicates the direction of the start tangent of the curve, while pointf together with *point2* indicates the direction of the end tangent of the curve. If more than one position of the fillet curve is possible, the closest curve to *pointf* is chosen.
point2	-	Point indicating that the fillet should be put on the side of *curve2* where *point2* is situated.
radius	-	The radius to be used on the fillet if a circular fillet is possible, otherwise a conic or a quadratic polynomial curve is used, approximating the circular fillet.
normal	-	Normal to the plane the fillet curve should lie close to. This is only used in 3D fillet calculations, and the fillet centre will be in the direction of the cross product of the curve tangents and the normal.

filltype	-	Indicator of type of fillet. $= 1$: Circle, interpolating tangent on first curve, not on curve 2. $= 2$: Conic if possible, else : polynomial segment.
dim	-	Dimension of space.
order	-	Order of fillet curve, which is not always used.

Output Arguments:

newcurve	-	Pointer to the B-spline fillet curve.
parend1	-	Parameter value of the end of curve 1 not affected by the fillet.
parspt1	-	Parameter value of the point on curve 1 where the fillet starts.
parend2	-	Parameter value of the end of curve 2 not affected by the fillet.
parept2	-	Parameter value of the point on curve 2 where the fillet ends.
stat	-	Status messages > 0 : warning $= 0$: ok < 0 : error

EXAMPLE OF USE

```
{
    SISLCurve   *curve1;  /* Must be defined */
    SISLCurve   *curve2;  /* Must be defined */
    double      epsge = 0.00001;
    double      point1[3];   /* Must be defined */
    double      pointf[3];   /* Must be defined */
    double      point2[3];   /* Must be defined */
    double      radius;      /* Must be defined */
    double      normal[3];  /* Must be defined */
    int         filltype = 2;
    int         dim = 3;
    int         order = 4;  /* If not given by filltype */
    SISLCurve   *newcurve = NULL;
    double      parend1;
    double      parspt1;
    double      parend2;
    double      parept2;
    int         stat = 0;
    ...
    s1609(curve1, curve2, epsge, point1, pointf, point2, radius,
          normal, filltype, dim, order, &newcurve, &parend1, &parspt1,
          &parend2, &parept2, &stat);
    ...
}
```

3.1.9 Compute a circular fillet between a 2D curve and a circle.

NAME

s1014 - Compute the fillet by iterating to the start and end points of a fillet between a 2D curve and a circle. The centre of the circular fillet is also calculated.

SYNOPSIS

void s1014(*pc1*, *circ_cen*, *circ_rad*, *aepsge*, *eps1*, *eps2*, *aradius*, *parpt1*, *parpt2*, *centre*, *jstat*)

```
SISLCurve   *pc1;
double      circ_cen[];
double      circ_rad;
double      aepsge;
double      eps1[];
double      eps2[];
double      aradius;
double      *parpt1;
double      *parpt2;
double      centre[];
int         *jstat;
```

ARGUMENTS

Input Arguments:

pc1	-	The first input curve.
circ_cen	-	2D centre of the circle.
circ_rad	-	Radius of the circle.
aepsge	-	Geometry resolution.
eps1	-	2D point telling that the fillet should be put on the side of curve 1 where *eps1* is situated.
eps2	-	2D point telling that the fillet should be put on the side of the input circle where *eps2* is situated.
aradius	-	The radius to be used on the fillet.

Input/Output Arguments:

parpt1	-	Parameter value of the point on curve 1 where the fillet starts. Input is a guess value for the iteration.
parpt2	-	Parameter value of the point on the input circle where the fillet ends. Input is a guess value for the iteration.

Output Arguments:

centre	-	2D centre of the circular fillet. Space must be allocated outside the function.
jstat	-	Status message $= 1$: Converged, $= 2$: Diverged, < 0 : Error.

EXAMPLE OF USE

```
{
    SISLCurve   *pc1; /* Must be defined */
    double      circ_cen[2]; /* Must be defined */
    double      circ_rad; /* Must be defined */
    double      aepsge = 0.00001;
    double      eps1[2]; /* Must be defined */
    double      eps2[2]; /* Must be defined */
    double      aradius; /* Must be defined */
    double      parpt1;
    double      parpt2;
    double      centre[2];
    int         jstat = 0;
    ...
    s1014(pc1, circ_cen, circ_rad, aepsge, eps1, eps2, aradius, &parpt1, &parpt2,
          centre, &jstat);
    ...
}
```

3.1.10 Compute a circular fillet between two 2D curves.

NAME

s1015 - Compute the fillet by iterating to the start and end points of a fillet between two 2D curves. The centre of the circular fillet is also calculated.

SYNOPSIS

void s1015(*pc1*, *pc2*, *aepsge*, *eps1*, *eps2*, *aradius*, *parpt1*, *parpt2*, *centre*, *jstat*)

SISLCurve	**pc1*;
SISLCurve	**pc2*;
double	*aepsge*;
double	*eps1*[ ];
double	*eps2*[ ];
double	*aradius*;
double	**parpt1*;
double	**parpt2*;
double	*centre*[ ];
int	**jstat*;

ARGUMENTS

Input Arguments:

pc1	-	The first 2D input curve.
pc2	-	The second 2D input curve.
aepsge	-	Geometry resolution.
eps1	-	2D point telling that the fillet should be put on the side of curve 1 where *eps1* is situated.
eps2	-	2D point telling that the fillet should be put on the side of curve 2 where *eps2* is situated.
aradius	-	The radius to be used on the fillet.

Input/Output Arguments:

parpt1	-	Parameter value of the point on curve 1 where the fillet starts. Input is a guess value for the iteration.
parpt2	-	Parameter value of the point on curve 2 where the fillet ends. Input is a guess value for the iteration.

Output Arguments:

centre	-	2D centre of the circular fillet. Space must be allocated outside the function.
jstat	-	Status message $= 1$: Converged, $= 2$: Diverged, < 0 : Error.

EXAMPLE OF USE

```
{
    SISLCurve   *pc1; /* Must be defined */
    SISLCurve   *pc2; /* Must be defined */
    double      aepsge = 0.00001;
    double      eps1[2]; /* Must be defined */
    double      eps2[2]; /* Must be defined */
    double      aradius; /* Must be defined */
    double      parpt1; /* Must be defined */
    double      parpt2; /* Must be defined */
    double      centre[2];
    int         jstat = 0;
    . . .
    s1015(pc1, pc2, aepsge, eps1, eps2, aradius, &parpt1, &parpt2, centre, &js-
        tat);
    . . .
}
```

3.1.11 Compute a circular fillet between a 2D curve and a 2D line.

NAME

s1016 - Compute the fillet by iterating to the start and end points of a fillet between a 2D curve and a 2D line. The centre of the circular fillet is also calculated.

SYNOPSIS

void s1016(*pc1*, *point*, *normal*, *aepsge*, *eps1*, *eps2*, *aradius*, *parpt1*, *parpt2*, *centre*, *jstat*)

```
SISLCurve   *pc1;
double      point[];
double      normal[];
double      aepsge;
double      eps1[];
double      eps2[];
double      aradius;
double      *parpt1;
double      *parpt2;
double      centre[];
int         *jstat;
```

ARGUMENTS

Input Arguments:

pc1	-	The 2D input curve.
point	-	2D point on the line.
normal	-	2D normal to the line.
aepsge	-	Geometry resolution.
eps1	-	2D point telling that the fillet should be put on the side of curve 1 where *eps1* is situated.
eps2	-	2D point telling that the fillet should be put on the side of curve 2 where *eps2* is situated.
aradius	-	The radius to be used on the fillet.

Input/Output Arguments:

parpt1	-	Parameter value of the point on curve 1 where the fillet starts. Input is a guess value for the iteration.
parpt2	-	Parameter value of the point on the line where the fillet ends. Input is a guess value for the iteration.

Output Arguments:

centre	-	2D centre of the (circular) fillet. Space must be allocated outside the function.

jstat - Status message
$= 1$: Converged,
$= 2$: Diverged,
< 0 : Error.

EXAMPLE OF USE

```
{
    SISLCurve   *pc1; /* Must be defined */
    double      point[2]; /* Must be defined */
    double      normal[2]; /* Must be defined */
    double      aepsge = 0.00001;
    double      eps1[2]; /* Must be defined */
    double      eps2[2]; /* Must be defined */
    double      aradius; /* Must be defined */
    double      parpt1;
    double      parpt2;
    double      centre[2];
    int         jstat = 0;
    . . .
    s1016(pc1, point, normal, aepsge, eps1, eps2, aradius, &parpt1, &parpt2,
          centre, &jstat);
    . . .
}
```

3.1.12 Compute a blending curve between two curves.

NAME

s1606 - To compute a blending curve between two curves. Two points indicate between which ends the blend is to be produced. The blending curve is either a circle or an approximated conic section if this is possible, otherwise it is a quadratic polynomial spline curve. The output is represented as a B-spline curve.

SYNOPSIS

void s1606(*curve1*, *curve2*, *epsge*, *point1*, *point2*, *blendtype*, *dim*, *order*, *newcurve*, *stat*)

SISLCurve	**curve1*;
SISLCurve	**curve2*;
double	*epsge*;
double	*point1*[ ];
double	*point2*[ ];
int	*blendtype*;
int	*dim;*
int	*order*;
SISLCurve	***newcurve*;
int	**stat*;

ARGUMENTS

Input Arguments:

curve1	-	The first input curve.
curve2	-	The second input curve.
epsge	-	Geometry resolution.
point1	-	Point near the end of curve 1 where the blend starts.
point2	-	Point near the end of curve 2 where the blend starts.
blendtype	-	Indicator of type of blending. $= 1$: Circle, interpolating tangent on first curve, not on curve 2, if possible. $= 2$: Conic if possible, else : polynomial segment.
dim	-	Dimension of the geometry space.
order	-	Order of the blending curve.

Output Arguments:

newcurve	-	Pointer to the B-spline blending curve.
stat	-	Status messages > 0 : warning $= 0$: ok < 0 : error

EXAMPLE OF USE

```
{
    SISLCurve   *curve1; /* Must be defined */
    SISLCurve   *curve2; /* Must be defined */
    double      epsge = 0.00001;
    double      point1[3]; /* Must be defined */
    double      point2[3]; /* Must be defined */
    int         blendtype = 1;
    int         dim = 3;  /* Must be consistent with curve1 and curve2 /*
    int         order = 3; /* If not given by blendtype */
    SISLCurve   *newcurve;
    int         stat = 0;
    . . .
    s1606(curve1, curve2, epsge, point1, point2, blendtype, dim, order,
          &newcurve, &stat);
    . . .
}
```

3.2 Approximation

Two kinds of curves are treated in this section. The first is approximations of special shapes like circles and conic segments. The second is approximation of a point set, or offsets to curves.

Except for the point set approximation function, all functions require a tolerance for the approximation. Note that there is a close relationship between the size of the tolerance and the amount of data for the curve.

3.2.1 Approximate a circular arc with a curve.

NAME

s1303 - To create a curve approximating a circular arc around the axis defined by the centre point, an axis vector, a start point and a rotational angle. The maximal deviation between the true circular arc and the approximation to the arc is controlled by the geometric tolerance (epsge). The output will be represented as a B-spline curve.

SYNOPSIS

void s1303(*startpt*, *epsge*, *angle*, *centrept*, *axis*, *dim*, *curve*, *stat*)

double	*startpt*[ ];
double	*epsge*;
double	*angle*;
double	*centrept*[ ];
double	*axis*[ ];
int	*dim*;
SISLCurve	***curve*;
int	**stat*;

ARGUMENTS

Input Arguments:

startpt	-	Start point of the circular arc
epsge	-	Maximal deviation allowed between the true circle and the circle approximation.
angle	-	The rotational angle. Counterclockwise around axis. If the rotational angle is outside $< -2\pi, +2\pi >$ then a closed curve is produced.
centrept	-	Point on the axis of the circle.
axis	-	Normal vector to plane in which the circle lies. Used if dim = 3.
dim	-	The dimension of the space in which the circular arc lies (2 or 3).

Output Arguments:

curve	-	Pointer to the B-spline curve.
stat	-	Status messages > 0 : warning $= 0$: ok < 0 : error

EXAMPLE OF USE

```
{
    double      startpt[3];  /* Must be defined */
    double      epsge = 0.001;
    double      angle;    /* Must be defined */
    double      centrept[3]; /* Must be defined */
    double      axis[3];    /* Must be defined */
    int         dim = 3;
    SISLCurve   *curve = NULL;
    int         stat = 0;
    ...
    s1303(startpt, epsge, angle, centrept, axis, dim, &curve, &stat);
    ...
}
```

3.2.2 Approximate a conic arc with a curve.

NAME

s1611 - To approximate a conic arc with a curve in two or three dimensional space. If two points are given, a straight line is produced, if three an approximation of a circular arc, and if four or five a conic arc. The output will be represented as a B-spline curve.

SYNOPSIS

void s1611(*point*, *numpt*, *dim*, *typept*, *open*, *order*, *startpar*, *epsge*, *endpar*, *curve*, *stat*)

double	*point*[ ];
int	*numpt*;
int	*dim*;
double	*typept*[ ];
int	*open*;
int	*order*;
double	*startpar*;
double	*epsge*;
double	**endpar*;
SISLCurve	***curve*;
int	**stat*;

ARGUMENTS

Input Arguments:

point	-	Array of length $dim \times numpt$ containing the points/ derivatives to be interpolated.
numpt	-	No. of points/derivatives in the point array.
dim	-	The dimension of the space in which the points lie.
typept	-	Array (length numpt) containing type indicator for points/derivatives/ second-derivatives: 1 : Ordinary point. 3 : Derivative to next point. 4 : Derivative to prior point.
open	-	Open or closed curve: 0 : Closed curve, not implemented. 1 : Open curve.
order	-	The order of the B-spline curve to be produced.
startpar	-	Parameter-value to be used at the start of the curve.
epsge	-	The geometry resolution.

Output Arguments:

endpar	-	Parameter-value used at the end of the curve.
curve	-	Pointer to the output B-spline curve.
stat	-	Status messages > 0 : warning $= 0$: ok < 0 : error

NOTE

When four points/tangents are given as input, the xy term of the implicit equation is set to zero. Thus the points might end on two branches of a hyperbola and a straight line is produced. When four or five points/tangents are given only three of these should actually be points.

EXAMPLE OF USE

```
{
    double      point[30];  /* Must be defined */
    int         numpt = 10;
    int         dim = 3;
    double      typept[10];  /* Must be defined */
    int         open = 1;
    int         order = 4;
    double      startpar = 0.0;
    double      epsge = 0.0001;
    double      endpar;
    SISLCurve   *curve = NULL;
    int         stat = 0;
    ...
    s1611(point, numpt, dim, typept, open, order, startpar, epsge,
          &endpar, &curve, &stat);
    ...
}
```

3.2.3 Compute a curve using the input points as controlling vertices, automatic parameterization.

NAME

s1630 - To compute a curve using the input points as controlling vertices. The distances between the points are used as parametrization. The output will be represented as a B-spline curve.

SYNOPSIS

void s1630(*epoint*, *inbpnt*, *astpar*, *iopen*, *idim*, *ik*, *rc*, *jstat*)

double	*epoint*[ ];
int	*inbpnt*;
double	*astpar*;
int	*iopen*;
int	*idim*;
int	*ik*;
SISLCurve	***rc*;
int	**jstat*;

ARGUMENTS

Input Arguments:

epoint	-	The array containing the points to be used as controlling vertices of the B-spline curve.
inbpnt	-	No. of points in epoint.
astpar	-	Parameter value to be used at the start of the curve.
iopen	-	Open/closed/periodic condition. $= -1$: Closed and periodic. $= 0$: Closed. $= 1$: Open.
idim	-	The dimension of the space.
ik	-	The order of the spline curve to be produced.

Output Arguments:

rc	-	Pointer to the B-spline curve.
jstat	-	Status message < 0 : Error. $= 0$: Ok. > 0 : Warning.

EXAMPLE OF USE

```
{
    double      epoint[30];  /* Must be defined */
    int         inbpnt = 10;
    double      astpar = 0.0;
    int         iopen = 1;
    int         idim = 3;
    int         ik = 4;
    SISLCurve   *rc = NULL;
    int         jstat = 0;
    ...
    s1630(epoint, inbpnt, astpar, iopen, idim, ik, &rc, &jstat);
    ...
}
```

3.2.4 Approximate the offset of a curve with a curve.

NAME

s1360 - To create a approximation of the offset to a curve within a tolerance. The output will be represented as a B-spline curve.
With an offset of zero, this routine can be used to approximate any NURBS curve, within a tolerance, with a (non-rational) B-spline curve.

SYNOPSIS

void s1360(*oldcurve*, *offset*, *epsge*, *norm*, *max*, *dim*, *newcurve*, *stat*)

```
SISLCurve   *oldcurve;
double      offset;
double      epsge;
double      norm[ ];
double      max;
int         dim;
SISLCurve   **newcurve;
int         *stat;
```

ARGUMENTS

Input Arguments:

oldcurve	-	The input curve.
offset	-	The offset distance. If dim=2, a positive sign on this value put the offset on the side of the positive normal vector, and a negative sign puts the offset on the negative normal vector. If dim=3, the offset direction is determined by the cross product of the tangent vector and the normal vector. The offset distance is multiplied by this cross product.
epsge	-	Maximal deviation allowed between the true offset curve and the approximated offset curve.
norm	-	Vector used in 3D calculations.
max	-	Maximal step length. It is neglected if max≤epsge. If max=0.0, then a maximal step equal to the longest box side of the curve is used.
dim	-	The dimension of the space must be 2 or 3.

NOTE

If the vector norm and the curve tangent are parallel at some point, then the curve produced will not be an offset at this point, and it will probably move from one side of the input curve to the other side.

Output Arguments:

newcurve	-	Pointer to the B-spline curve approximating the offset curve.
stat	-	Status messages. > 0 : Warning. $= 0$: Ok. < 0 : Error.

EXAMPLE OF USE

```
{
    SISLCurve   *oldcurve;   /* Must be defined */
    double      offset;   /* Must be defined */
    double      epsge;   /* Must be defined */
    double      norm[3];   /* Must be defined */
    double      max = 0.0;
    int         dim = 3;
    SISLCurve   *newcurve = NULL;
    int         stat = 0;
    ...
    s1360(oldcurve, offset, epsge, norm, max, dim, &newcurve, &stat);
    ...
}
```

3.2.5 Approximate a curve with a sequence of straight lines.

NAME

s1613 - To calculate a set of points on a curve. The straight lines between the points will not deviate more than *epsge* from the curve at any point. The generated points will have the same spatial dimension as the input curve.

SYNOPSIS

void s1613(*curve*, *epsge*, *points*, *numpoints*, *stat*)

SISLCurve	**curve*;
double	*epsge*;
double	***points*;
int	**numpoints*;
int	**stat*;

ARGUMENTS

Input Arguments:

curve	-	The input curve.
epsge	-	Geometry resolution, maximum distance allowed between the curve and the straight lines that are to be calculated.

Output Arguments:

points	-	Calculated points, (a vector of *numpoints* $\times$ *curve*->*idim* elements).
numpoints	-	Number of calculated points.
stat	-	Status messages > 0 : warning $= 0$: ok < 0 : error

EXAMPLE OF USE

```
{
    SISLCurve   *curve;  /* Must be defined */
    double      epsge;   /* Must be defined */
    double      *points = NULL;
    int         numpoints = 0;
    int         stat = 0;
    ...
    s1613(curve, epsge, &points, &numpoints, &stat);
    ...
}
```

3.3 Mirror a Curve

NAME

s1600 - To mirror a curve around a plane.

SYNOPSIS

void s1600(*oldcurve*, *point*, *normal*, *dim*, *newcurve*, *stat*)

SISLCurve	**oldcurve*;
double	*point*[ ];
double	*normal*[ ];
int	*dim*;
SISLCurve	***newcurve*;
int	**stat*;

ARGUMENTS

Input Arguments:

oldcurve	-	Pointer to original curve.
point	-	A point in the plane.
normal	-	Normal vector to the plane.
dim	-	The dimension of the space.

Output Arguments:

newcurve	-	Pointer to the mirrored curve.
stat	-	Status messages > 0 : warning $= 0$: ok < 0 : error

EXAMPLE OF USE

```
{
    SISLCurve   *oldcurve;  /* Must be defined */
    double      point[3];   /* Must be defined */
    double      normal[3];  /* Must be defined */
    int         dim = 3;
    SISLCurve   *newcurve =NULL;
    int         stat = 0;
    ...
    s1600(oldcurve, point, normal, dim, &newcurve, &stat);
    ...
}
```

3.4 Conversion

3.4.1 Convert a curve of order up to four, to a sequence of cubic polynomials.

NAME

s1389 - Convert a curve of order up to 4 to a sequence of non-rational cubic segments with uniform parameterization.

SYNOPSIS

void s1389(*curve*, *cubic*, *numcubic*, *dim*, *stat*)

SISLCurve	**curve*;
double	***cubic*;
int	**numcubic*;
int	**dim*;
int	**stat*;

ARGUMENTS

Input Arguments:

curve	-	Pointer to the curve that is to be converted

Output Arguments:

cubic	-	Array containing the sequence of cubic segments. Each segment is represented by the start point, followed by the start tangent, end point and end tangent. Each segment needs 4*dim doubles for storage.
numcubic	-	Number of elements of length (4*dim) in the array cubic
dim	-	The dimension of the geometric space.
stat	-	Status messages > 0 : warning $= 0$: ok < 0 : error

EXAMPLE OF USE

```
{
    SISLCurve   *curve;   /* Must be defined */
    double      *cubic = NULL;
    int         numcubic;
    int         dim;
    int         stat = 0;
    ...
    s1389(curve, &cubic, &numcubic, &dim, &stat);
    ...
}
```

3.4.2 Convert a curve to a sequence of Bezier curves.

NAME

s1730 - To convert a curve to a sequence of Bezier curves. The Bezier curves are stored as one curve with all knots of multiplicity newcurve->ik (order of the curve). If the input curve is rational, the generated Bezier curves will be rational too (i.e. there will be rational weights in the representation of the Bezier curves).

SYNOPSIS

void s1730(*curve*, *newcurve*, *stat*)

SISLCurve	**curve*;
SISLCurve	***newcurve*;
int	**stat*;

ARGUMENTS

Input Arguments:

curve	-	The curve to convert.

Output Arguments:

newcurve	-	The new curve containing all the Bezier curves.
stat	-	Status messages > 0 : warning $= 0$: ok < 0 : error

EXAMPLE OF USE

```
{
    SISLCurve   *curve;  /* Must be defined */
    SISLCurve   *newcurve = NULL;
    int         stat = 0;
    ...
    s1730(curve, &newcurve, &stat);
    ...
}
```

3.4.3 Pick out the next Bezier curve from a curve.

NAME

s1732 - To pick out the next Bezier curve from a curve. This function requires a curve represented as the curve that is output from s1730(). If the input curve is rational, the generated Bezier curves will be rational too (i.e. there will be rational weights in the representation of the Bezier curves, note the convention for coefficients in the rational case, see Chapter 6.1.1).

SYNOPSIS

void s1732(*curve*, *number*, *startpar*, *endpar*, *coef*, *stat*)

SISLCurve	**curve*;
int	*number*;
double	**startpar*;
double	**endpar*;
double	*coef*[ ];
int	**stat*;

ARGUMENTS

Input Arguments:

curve	-	curve to pick from.
number	-	The number of the Bezier curve that is to be picked, where $0 \leq number < in/ik$ (i.e. the number of vertices in the curve divided by the order of the curve).

Output Arguments:

startpar	-	The start parameter value of the Bezier curve.
endpar	-	The end parameter value of the Bezier curve.
coef	-	The vertices of the Bezier curve. Space of size $(idim + 1) \times ik$ (i.e. spatial dimension of curve +1 times the order of the curve) must be allocated outside the function.
stat	-	Status messages > 0 : warning $= 0$: ok < 0 : error

EXAMPLE OF USE

```
{
    SISLCurve   *curve;   /* Must be defined */
    int         number;   /* Must be defined */
    double      startpar;
    double      endpar;
    double      coef[12];   /* Assumes dimension=3, order=4, non-rational */
    int         stat = 0;
    ...
    s1732(curve, number, &startpar, &endpar, coef, &stat);
    ...
}
```

3.4.4 Express a curve using a higher order basis.

NAME

s1750 - To describe a curve using a higher order basis.

SYNOPSIS

void s1750(*curve*, *order*, *newcurve*, *stat*)

SISLCurve	**curve*;
int	*order*;
SISLCurve	***newcurve*;
int	**stat*;

ARGUMENTS

Input Arguments:

curve	-	The input curve.
order	-	Order of the new curve.

Output Arguments:

newcurve	-	The new curve of higher order.
stat	-	Status messages > 0 : warning $= 0$: ok < 0 : error

EXAMPLE OF USE

```
{
    SISLCurve   *curve; /* Must be defined */
    double      order;  /* Must be defined */
    SISLCurve   *newcurve = NULL;
    int         stat = 0;
    ...
    s1750(curve, order, &newcurve, &stat);
    ...
}
```

3.4.5 Express the "i"-th derivative of an open curve as a curve.

NAME

s1720 - To express the "i"-th derivative of an open curve as a curve.

SYNOPSIS

void s1720(*curve*, *derive*, *newcurve*, *stat*)

SISLCurve	**curve*;
int	*derive*;
SISLCurve	***newcurve*;
int	**stat*;

ARGUMENTS

Input Arguments:

curve	-	Curve to be differentiated.
derive	-	The order "i" of the derivative, where $0 \leq derive$.

Output Arguments:

newcurve	-	The "i"-th derivative of a curve represented as a curve.
stat	-	Status messages > 0 : warning $= 0$: ok < 0 : error

EXAMPLE OF USE

```
{
    SISLCurve   *curve;  /* Must be defined */
    int         derive = 1;
    SISLCurve   *newcurve = NULL;
    int         stat = 0;
    ...
    s1720(curve, derive, &newcurve, &stat);
    ...
}
```

3.4.6 Express a 2D or 3D ellipse as a curve.

NAME

s1522 - Convert a 2D or 3D analytical ellipse to a curve. The curve will be geometrically exact.

SYNOPSIS

void s1522(*normal*, *centre*, *ellipaxis*, *ratio*, *dim*, *ellipse*, *jstat*)

double	*normal*[ ];
double	*centre*[ ];
double	*ellipaxis*[ ];
double	*ratio*;
int	*dim*;
SISLCurve	***ellipse*;
int	**jstat*;

ARGUMENTS

Input Arguments:

normal	-	3D normal to ellipse plane (not necessarily normalized). Used if $dim = 3$.
centre	-	Centre of ellipse (2D if $dim = 2$ and 3D if $dim = 3$).
ellipaxis	-	This will be used as starting point for the ellipse curve (2D if $dim = 2$ and 3D if $dim = 3$).
ratio	-	The ratio between the length of the given ellipaxis and the length of the other axis, i.e. $\|ellipaxis\|/\|otheraxis\|$ (a compact representation format).
dim	-	Dimension of the space in which the elliptic nurbs curve lies (2 or 3).

Output Arguments:

ellipse	-	Ellipse curve (2D if $dim = 2$ and 3D if $dim = 3$).
stat	-	Status messages > 0 : warning $= 0$: ok < 0 : error

EXAMPLE OF USE

```
{
    double      normal[3];  /* Must be defined */
    double      centre[3];  /* Must be defined */
    double      ellipaxis[3];  /* Must be defined */
    double      ratio;  /* Must be defined */
    int         dim = 3;
    SISLCurve   *ellipse = NULL;
    int         jstat = 0;
    ...
    s1522(normal, centre, ellipaxis, ratio, dim, &ellipse, &jstat);
    ...
}
```

3.4.7 Express a conic arc as a curve.

NAME

s1011 - Convert an analytic conic arc to a curve. The curve will be geometrically exact. The arc is given by position at start, shoulder point and end, and a shape factor.

SYNOPSIS

void s1011(*start_pos*, *top_pos*, *end_pos*, *shape*, *dim*, *arc_seg*, *stat*)

double	*start_pos*[ ];
double	*top_pos*[ ];
double	*end_pos*[ ];
double	*shape*;
int	*dim*;
SISLCurve	***arc_seg*;
int	**stat*;

ARGUMENTS

Input Arguments:

start_pos	-	Start point of segment.
top_pos	-	Shoulder point of segment. This is the intersection point of the tangents in *start_pos* and *end_pos*.
end_pos	-	End point of segment.
shape	-	Shape factor, must be ≥ 0. < 0.5, an ellipse, $= 0.5$, a parabola, > 0.5, a hyperbola, ≥ 1, the start and end points lies on different branches of the hyperbola. We want a single arc segment, therefore if *shape* ≥ 1, shape is set to 0.999999.
dim	-	The spatial dimension of the curve to be produced.

Output Arguments:

jstat	-	Status message < 0 : Error. $= 0$: Ok. > 0 : Warning.
arc_seg	-	Pointer to the curve produced.

EXAMPLE OF USE

```
{
    double      start_pos[3];  /* Must be defined */
    double      top_pos[3];    /* Must be defined */
    double      end_pos[3];    /* Must be defined */
    double      shape = 0.3;
    int         dim = 3;
    SISLCurve   *arc_seg = NULL;
    int         stat = 0;
    ...
    s1011(start_pos, top_pos, end_pos, shape, dim, &arc_seg, &stat);
    ...
}
```

3.4.8 Express a truncated helix as a curve.

NAME

s1012 - Convert an analytical truncated helix to a curve. The curve will be geometrically exact.

SYNOPSIS

void s1012(*start_pos*, *axis_pos*, *axis_dir*, *frequency*, *numb_quad*, *counter_clock*, *helix*, *stat*)

double	*start_pos*[ ];
double	*axis_pos*[ ];
double	*axis_dir*[ ];
double	*frequency*;
int	*numb_quad*;
int	*counter_clock*;
SISLCurve	***helix*;
int	**stat*;

ARGUMENTS

Input Arguments:

start_pos	-	Start position on the helix.
axis_pos	-	Point on the helix axis.
axis_dir	-	Direction of the helix axis.
frequency	-	The length along the helix axis for one period of revolution.
numb_quad	-	Number of quadrants in the helix.
counter_clock	-	Flag for direction of revolution: $= 0$: clockwise, $= 1$: counter_clockwise.

Output Arguments:

jstat	-	Status message < 0 : Error. $= 0$: Ok. > 0 : Warning.
helix	-	Pointer to the helix curve produced.

EXAMPLE OF USE

```
{
    double      start_pos[3];  /* Must be defined */
    double      axis_pos[3];   /* Must be defined */
    double      axis_dir[3];   /* Must be defined */
    double      frequency;     /* Must be defined */
    int         numb_quad = 5;
    int         counter_clock = 1;
    SISLCurve   *helix = NULL;
    int         stat = 0;
    ...
    s1012(start_pos, axis_pos, axis_dir, frequency, numb_quad, counter_clock,
          &helix, &stat)
    ...
}
```

Chapter 4

Curve Interrogation

This chapter describes the functions in the Curve Interrogation module.

4.1 Intersections

4.1.1 Intersection between a curve and a point.

NAME

s1871 - Find all the intersections between a curve and a point.

SYNOPSIS

void s1871(*pc1*, *pt1*, *idim*, *aepsge*, *jpt*, *gpar1*, *jcrv*, *wcurve*, *jstat*)

SISLCurve	**pc1*;
double	**pt1*;
int	*idim*;
double	*aepsge*;
int	**jpt*;
double	***gpar1*;
int	**jcrv*;
SISLIntcurve	****wcurve*;
int	**jstat*;

ARGUMENTS

Input Arguments:

pc1	-	Pointer to the curve.
pt1	-	coordinates of the point.
idim	-	number of coordinates in *pt1*.
aepsge	-	Geometry resolution.

Output Arguments:

jpt	-	Number of single intersection points.
gpar1	-	Array containing the parameter values of the single intersection points in the parameter interval of the curve. The points lie continuous. Intersection curves are stored in *wcurve*.

jcrv	-	Number of intersection curves.
wcurve	-	Array containing descriptions of the intersection curves. The curves are only described by points in the parameter plane. The curve-pointers points to nothing. If the curves given as input are degenerate, an intersection point can be returned as an intersection curve. Use s1327() to decide if an intersection curve is a point on one of the curves.
jstat	-	Status messages > 0 : Warning. $= 0$: Ok. < 0 : Error.

EXAMPLE OF USE

```
{
    SISLCurve    *pc1; /* Must be defined */
    double       *pt1; /* Must be defined */
    int          idim;  /* Equal to the curve dimension */
    double       aepsge = 0.000001 ;
    int          jpt = 0;
    double       *gpar1 = NULL;
    int          jcrv = 0;
    SISLIntcurve **wcurve = NULL;
    int          jstat = 0;
    ...
    s1871(pc1, pt1, idim, aepsge, &jpt, &gpar1, &jcrv, &wcurve, &jstat);
    ...
}
```

4.1.2 Intersection between a spline curve and a straight line or a plane.

NAME

s1850 - Find all the intersections between a spline curve and a plane (if curve dimension and $dim = 3$) or a curve and a line (if curve dimension and $dim = 2$).

SYNOPSIS

void s1850(*curve*, *point*, *normal*, *dim*, *epsco*, *epsge*, *numintpt*, *intpar*, *numintcu*, *intcurve*, *stat*)

SISLCurve	**curve*;
double	*point*[ ];
double	*normal*[ ];
int	*dim*;
double	*epsco*;
double	*epsge*;
int	**numintpt*;
double	***intpar*;
int	**numintcu*;
SISLIntcurve	****intcurve*;
int	**stat*;

ARGUMENTS

Input Arguments:

curve	-	Pointer to the curve.
point	-	Point in the plane/line.
normal	-	Normal to the plane or any normal to the direction of the line.
dim	-	Dimension of the space in which the curve and the plane/line lies, *dim* must be equal to two or three.
epsco	-	Computational resolution (not used).
epsge	-	Geometry resolution.

Output Arguments:

numintpt	-	Number of single intersection points.
intpar	-	Array containing the parameter values of the single intersection points in the parameter interval of the curve. The points lie in sequence. Intersection curves are stored in intcurve.
numintcu	-	Number of intersection curves.

intcurve	-	Array of pointers to SISLIntcurve objects containing description of the intersection curves. The curves are only described by start points and end points in the parameter interval of the curve. The curve pointers point to nothing.
stat	-	Status messages > 0 : warning $= 0$: ok < 0 : error

EXAMPLE OF USE

```
{
    SISLCurve    *curve;  /* Must be defined */
    double       point[3];  /* Must be defined */
    double       normal[3]; /* Must be defined */
    int          dim = 3;
    double       epsco = 1.0e-9; /* Not used */
    double       epsge = 1.0e-6;
    int          numintpt = 0;
    double       *intpar = NULL;
    int          numintcu = 0;
    SISLIntcurve **intcurve = NULL;
    int          stat = 0;
    ...
    s1850(curve, point, normal, dim, epsco, epsge, &numintpt, &intpar, &numintcu, &intcurve, &stat);
    ...
}
```

4.1.3 Convert a curve/line intersection into a two-dimensional curve/origo intersection

NAME

s1327 - Put the equation of the curve pointed at by pcold into two planes given by the point epoint and the normals enorm1 and enorm2. The result is an equation where the new two-dimensional curve rcnew is to be equal to origo.

SYNOPSIS

```
void s1327(pcold, epoint, enorm1, enorm2, idim, rcnew, jstat)
    SISLCurve   *pcold;
    double      epoint[ ];
    double      enorm1[ ];
    double      enorm2[ ];
    int         idim;
    SISLCurve   **rcnew;
    int         *jstat;
```

ARGUMENTS

Input Arguments:

pcold	-	Pointer to input curve.
epoint	-	SISLPoint in the planes.
enorm1	-	Normal to the first plane.
enorm2	-	Normal to the second plane.
idim	-	Dimension of the space in which the planes lie.

Output Arguments:

rcnew	-	2-dimensional curve.
jstat	-	status messages > 0 : warning $= 0$: ok < 0 : error

EXAMPLE OF USE

```
{
    SISLCurve   *pcold;  /* Must be defined */
    double      epoint[3];  /* Must be defined */
    double      enorm1[3];  /* Must be defined */
    double      enorm2[3];  /* Must be defined */
    int         idim = 3; /* Equal to curve dimension */
    SISLCurve   **rcnew = NULL;
    int         *jstat = 0;
    ...
    s1327(pcold, epoint, enorm1, enorm2, idim, rcnew, jstat);
    ...
}
```

4.1.4 Intersection between a spline curve and a 2D circle or a sphere.

NAME

s1371 - Find all the intersections between a curve and a sphere (if curve dimension and $dim = 3$), or a curve and a circle (if curve dimension and $dim = 2$).

SYNOPSIS

void s1371(*curve*, *centre*, *radius*, *dim*, *epsco*, *epsge*, *numintpt*, *intpar*, *numintcu*, *intcurve*, *stat*)

SISLCurve	**curve*;
double	*centre*[ ];
double	*radius*;
int	*dim*;
double	*epsco*;
double	*epsge*;
int	**numintpt*;
double	***intpar*;
int	**numintcu*;
SISLIntcurve	****intcurve*;
int	**stat*;

ARGUMENTS

Input Arguments:

curve	-	Pointer to the curve.
centre	-	Centre of the circle/sphere.
radius	-	Radius of circle or sphere.
dim	-	Dimension of the space in which the curve and the circle/sphere lies, *dim* should be equal to two or three.
epsco	-	Computational resolution (not used).
epsge	-	Geometry resolution.

Output Arguments:

numintpt	-	Number of single intersection points.
intpar	-	Array containing the parameter values of the single intersection points in the parameter interval of the curve. The points lie in sequence. Intersection curves are stored in intcurve.
numintcu	-	Number of intersection curves.
intcurve	-	Array of pointers to SISLIntcurve objects containing descriptions of the intersection curves. The curves are only described by start points and end points in the parameter interval of the curve. The curve pointers point to nothing.

stat	-	Status messages > 0 : warning $= 0$: ok < 0 : error

EXAMPLE OF USE

```
{
    SISLCurve    *curve;   /* Must be defined */
    double       centre[3];  /* Must be defined */
    double       radius;   /* Must be defined */
    int          dim = 3;
    double       epsco = 1.0e-9; /* Not used */
    double       epsge = 1.0e-6;
    int          numintpt = 0;
    double       *intpar = NULL;
    int          numintcu = 0;
    SISLIntcurve **intcurve = NULL;
    int          stat = 0;
    ...
    s1371(curve, centre, radius, dim, epsco, epsge, &numintpt, &intpar, &numintcu, &intcurve, &stat);
    ...
}
```

4.1.5 Intersection between a curve and a quadric curve.

NAME

s1374 - Find all the intersections between a curve and a quadric curve, (if curve dimension and $dim = 2$), or a curve and a quadric surface, (if curve dimension and $dim = 3$).

SYNOPSIS

void s1374(*curve*, *conarray*, *dim*, *epsco*, *epsge*, *numintpt*, *intpar*, *numintcu*, *intcurve*, *stat*)

SISLCurve	**curve*;
double	*conarray*[ ];
int	*dim*;
double	*epsco*;
double	*epsge*;
int	**numintpt*;
double	***intpar*;
int	**numintcu*;
SISLIntcurve	****intcurve*;
int	**stat*;

ARGUMENTS

Input Arguments:

curve	-	Pointer to the curve.
conarray	-	Matrix of dimension $(dim + 1) \times (dim + 1)$ describing the conic curve or surface with homogeneous coordinates. For dim=2 the implicit equation of the curve is that the following is equal to zero:

$$\begin{pmatrix} x & y & 1 \end{pmatrix} \begin{pmatrix} c_0 & c_1 & c_2 \\ c_3 & c_4 & c_5 \\ c_6 & c_7 & c_8 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

dim	-	Dimension of the space in which the cone and the curve lie, *dim* should be equal to two or three.
epsco	-	Computational resolution (not used).
epsge	-	Geometry resolution.

Output Arguments:

numintpt	-	Number of single intersection points.
intpar	-	Array containing the parameter values of the single intersection points in the parameter interval of the curve. The points lie in sequence. Intersection curves are stored in intcurve.
numintcu	-	Number of intersection curves.
intcurve	-	Array of pointers to SISLIntcurve objects containing descriptions of the intersection curves. The curves are only described by start points and end points in the parameter interval of the curve. The curve pointers point to nothing.
stat	-	Status messages > 0 : Warning. $= 0$: Ok. < 0 : Error.

EXAMPLE OF USE

```
{
    SISLCurve     *curve;  /* Must be defined */
    double        conarray[16];  /* Must be defined */
    int           dim = 3;
    double        epsco = 1.0e-9; /* Not used */
    double        epsge = 1.0e-6;
    int           numintpt = 0;
    double        *intpar = NULL;
    int           numintcu = 0;
    SISLIntcurve  **intcurve = NULL;
    int           stat = 0;
    ...
    s1374(curve, conarray, dim, epsco, epsge, &numintpt, &intpar,
          &numintcu, &intcurve, &stat);
    ...
}
```

4.1.6 Intersection between two curves.

NAME

s1857 - Find all the intersections between two curves.

SYNOPSIS

void s1857(*curve1*, *curve2*, *epsco*, *epsge*, *numintpt*, *intpar1*, *intpar2*, *numintcu*, *intcurve*, *stat*)

SISLCurve	**curve1*;
SISLCurve	**curve2*;
double	*epsco*;
double	*epsge*;
int	**numintpt*;
double	***intpar1*;
double	***intpar2*;
int	**numintcu*;
SISLIntcurve	****intcurve*;
int	**stat*;

ARGUMENTS

Input Arguments:

curve1	-	Pointer to the first curve.
curve2	-	Pointer to the second curve.
epsco	-	Computational resolution (not used).
epsge	-	Geometry resolution.

Output Arguments:

numintpt	-	Number of single intersection points.
intpar1	-	Array containing the parameter values of the single intersection points in the parameter interval of the first curve. Intersection curves are stored in intcurve.
intpar2	-	Array containing the parameter values of the single intersection points in the parameter interval of the second curve. Intersection curves are stored in intcurve.
numintcu	-	Number of intersection curves.
intcurve	-	Array of pointers to the SISLIntcurve objects containing descriptions of the intersection curves. The curves are only described by start points and end points in the parameter interval of the curve. The curve pointers point to nothing. If the curves given as input are degenerate, an intersection point can be returned as an intersection curve.
stat	-	Status messages > 0 : warning $= 0$: ok < 0 : error

EXAMPLE OF USE

{

SISLCurve	**curve1*;	/* Must be defined */
SISLCurve	**curve2*;	/* Must be defined */

```
    double        epsco = 1.0e-9; /* Not used */
    double        epsge = 1.0e-6;
    int           numintpt = 0;
    double        *intpar1 = NULL;
    double        *intpar2 = NULL;
    int           numintcu = 0;
    SISLIntcurve **intcurve = NULL;
    int           stat = 0;
    . . .
    s1857(curve1, curve2, epsco, epsge, &numintpt, &intpar1, &intpar2, &nu-
        mintcu, &intcurve, &stat);
    . . .
}
```

4.2 Compute the Length of a Curve

NAME

s1240 - Compute the length of a curve. The length calculated will not deviate more than *epsge* divided by the calculated length, from the real length of the curve.

SYNOPSIS

void s1240(*curve*, *epsge*, *length*, *stat*)

SISLCurve	**curve*;
double	*epsge*;
double	**length*;
int	**stat*;

ARGUMENTS

Input Arguments:

curve	-	The curve.
epsge	-	Geometry resolution.

Output Arguments:

length	-	The length of the curve.
stat	-	Status messages > 0 : Warning. $= 0$: Ok. < 0 : Error.

NOTE

The algorithm is based on recursive subdivision and will thus for small values of *epsge* require long computation time.

EXAMPLE OF USE

```
{
    SISLCurve   *curve;  /* Must be defined */
    double      epsge = 0.001;
    double      length;
    int         stat = 0;
    ...
    s1240(curve, epsge, &length, &stat);
    ...
}
```

4.3 Check if a Curve is Closed

NAME

s1364 - To check if a curve is closed, i.e. test if the distance between the end points of the curve is less than a given tolerance.

SYNOPSIS

```
void s1364(curve, epsge, stat)
     SISLCurve   *curve;
     double      epsge;
     int         *stat;
```

ARGUMENTS

Input Arguments:

curve	-	The curve.
epsge	-	Geometric tolerance.

Output Arguments:

stat	-	Status messages = 2 : Curve is closed and periodic. = 1 : Curve is closed. = 0 : Curve is open. < 0 : Error.

EXAMPLE OF USE

```
{
     SISLCurve   *curve;  /* Must be defined */
     double      epsge = 1.0e-6;
     int         stat = 0;
     ...
     s1364(curve, epsge, &stat);
     ...
}
```

4.4 Check if a Curve is Degenerated.

NAME

s1451 - To check if a curve is degenerated.

SYNOPSIS

void s1451(*pc1*, *aepsge*, *jdgen*, *jstat*)

SISLCurve	**pc1*;
double	*aepsge*;
int	**jdgen*;
int	**jstat*;

ARGUMENTS

Input Arguments:

pc1	-	Pointer to the curve to be tested.
aepsge	-	The curve is degenerate if all vertices lie within the distance aepsge from each other

Output Arguments:

jdgen	-	Degenerate indicator $= 0$: The curve is not degenerate. $= 1$: The curve is degenerate.
jstat	-	Status message < 0 : Error. $= 0$: Ok. > 0 : Warning.

EXAMPLE OF USE

```
{
    SISLCurve   *pc1;  /* Must be defined */
    double      aepsge = 1.0e-5;
    int         *jdgen = 0;
    int         *jstat = 0;
    ...
    s1451(pc1, aepsge, jdgen, jstat);
    ...
}
```

4.5 Pick the Parameter Range of a Curve

NAME

s1363 - To pick the parameter range of a curve.

SYNOPSIS

void s1363(*curve*, *startpar*, *endpar*, *stat*)

```
    SISLCurve   *curve;
    double      *startpar;
    double      *endpar;
    int         *stat;
```

ARGUMENTS

Input Arguments:

curve	-	The curve.

Output Arguments:

startpar	-	Start of the parameter interval of the curve.
endpar	-	End of the parameter interval of the curve.
stat	-	Status messages $= 1$: warning $= 0$: ok < 0 : error

EXAMPLE OF USE

```
{
    SISLCurve   *curve;   /* Must be defined */
    double      startpar;
    double      endpar;
    int         stat = 0;
    ...
    s1363(curve, &startpar, &endpar, &stat);
    ...
}
```

4.6 Closest Points

4.6.1 Find the closest point between a curve and a point.

NAME

s1953 - Find the closest points between a curve and a point.

SYNOPSIS

void s1953(*curve*, *point*, *dim*, *epsco*, *epsge*, *numintpt*, *intpar*, *numintcu*, *intcurve*, *jstat*)

```
SISLCurve     *curve;
double        point[];
int           dim;
double        epsco;
double        epsge;
int           *numintpt;
double        **intpar;
int           *numintcu;
SISLIntcurve  ***intcurve;
int           *jstat;
```

ARGUMENTS

Input Arguments:

curve	-	Pointer to the curve in the closest point problem.
point	-	The point in the closest point problem.
dim	-	Dimension of the space in which the curve and point lie.
epsco	-	Computational resolution (not used).
epsge	-	Geometry resolution.

Output Arguments:

numintpt	-	Number of single closest points.
intpar	-	Array containing the parameter values of the single closest points in the parameter interval of the curve. The points lie in sequence. Closest curves are stored in intcurve.
numintcu	-	Number of closest curves.
intcurve	-	Array of pointers to the SISLIntcurve objects containing descriptions of the closest curves. The curves are only described by start points and end points in the parameter interval of the curve. The curve pointers point to nothing.
jstat	-	Status messages > 0 : warning $= 0$: ok < 0 : error

EXAMPLE OF USE

```
{
    SISLCurve     *curve;   /* Must be defined */
    double        point[3];  /* Must be defined */
    int           dim = 3;
    double        epsco = 1.9e-9; /* Not used */
    double        epsge = 1.0e-6;
    int           numintpt = 0;
    double        *intpar = NULLL;
    int           numintcu = 0;
    SISLIntcurve  **intcurve = NULL;
    int           jstat = 0;
    ...
    s1953(curve, point, dim, epsco, epsge, &numintpt, &intpar,
          &numintcu, &intcurve, &jstat);
    ...
}
```

4.6.2 Find the closest point between a curve and a point. Simple version.

NAME

s1957 - Find the closest point between a curve and a point. The method is fast and should work well in clear cut cases but does not guarantee finding the right solution. As long as it doesn't fail, it will find exactly one point. In other cases, use s1953().

SYNOPSIS

void s1957(*pcurve*, *epoint*, *idim*, *aepsco*, *aepsge*, *gpar*, *dist*, *jstat*)

SISLCurve	**pcurve*;
double	*epoint*[ ];
int	*idim*;
double	*aepsco*;
double	*aepsge*;
double	**gpar*;
double	**dist*;
int	**jstat*;

ARGUMENTS

Input Arguments:

pcurve	-	Pointer to the curve in the closest point problem.
epoint	-	The point in the closest point problem.
idim	-	Dimension of the space in which *epoint* lies.
aepsco	-	Computational resolution (not used).
aepsge	-	Geometry resolution.

Output Arguments:

gpar	-	The parameter value of the closest point in the parameter interval of the curve.
dist	-	The closest distance between curve and point.
jstat	-	Status message < 0 : Error. $= 0$: Point found by iteration. > 0 : Warning. $= 1$: Point lies at an end.

EXAMPLE OF USE

```
{
    SISLCurve   *pcurve;   /* Must be defined */
    double      epoint[3];   /* Must be defined */
    int         idim = 3;
    double      aepsco = 1.0e-9; /* Not used */
    double      aepsge = 1.0e-6;
    double      gpar = 0;
    double      dist = 0;
    int         jstat = 0;
    ...
    s1957(pcurve, epoint, idim, aepsco, aepsge, &gpar, &dist, &jstat);
    ...
}
```

4.6.3 Local iteration to closest point between point and curve.

NAME

s1774 - Newton iteration on the distance function between a curve and a point, to find a closest point or an intersection point. If a bad choice for the guess parameter is given in, the iteration may end at a local, not global closest point.

SYNOPSIS

```
void s1774(crv, point, dim, epsge, start, end, guess, clpar, stat)
    SISLCurve   *crv;
    double      point[];
    int         dim;
    double      epsge;
    double      start;
    double      end;
    double      guess;
    double      *clpar;
    int         *stat;
```

ARGUMENTS

Input Arguments:

crv	-	The curve in the closest point problem.
point	-	The point in the closest point problem.
dim	-	Dimension of the geometry.
epsge	-	Geometrical resolution.
start	-	Curve parameter giving the start of the search interval.
end	-	Curve parameter giving the end of the search interval.
guess	-	Curve guess parameter for the closest point iteration.

Output Arguments:

clpar	-	Resulting curve parameter from the iteration.
stat	-	Status messages > 0 : A minimum distance found. $= 0$: Intersection found. < 0 : Error.

EXAMPLE OF USE

```
{
    SISLCurve   *crv;  /* Must be defined */
    double      point[3];  /* Must be defined */
    int         dim = 3;
    double      epsge = 1.0e-6;
    double      start;  /* Must be defined */
    double      end;  /* Must be defined */
    double      guess;  /* Must be defined */
    double      clpar = 0;
    int         stat = 0;
    ...
```

```
    s1774(crv, point, dim, epsge, start, end, guess, &clpar, &stat);
    ...
}
```

4.6.4 Find the closest points between two curves.

NAME

s1955 - Find the closest points between two curves.

SYNOPSIS

void s1955(*curve1*, *curve2*, *epsco*, *epsge*, *numintpt*, *intpar1*, *intpar2*, *numintcu*, *intcurve*, *stat*)

SISLCurve	**curve1*;
SISLCurve	**curve2*;
double	*epsco*;
double	*epsge*;
int	**numintpt*;
double	***intpar1*;
double	***intpar2*;
int	**numintcu*;
SISLIntcurve	****intcurve*;
int	**stat*;

ARGUMENTS

Input Arguments:

curve1	-	Pointer to the first curve in the closest point problem.
curve2	-	Pointer to the second curve in the closest point problem.
epsco	-	Computational resolution (not used).
epsge	-	Geometry resolution.

Output Arguments:

numintpt	-	Number of single closest points.
intpar1	-	Array containing the parameter values of the single closest points in the parameter interval of the first curve. The points lie in sequence. Closest curves are stored in intcurve.
intpar2	-	Array containing the parameter values of the single closest points in the parameter interval of the second curve. The points lie in sequence. Closest curves are stored in intcurve.
numintcu	-	Number of closest curves.
intcurve	-	Array of pointers to the SISLIntcurve objects containing descriptions of the closest curves. The curves are only described by start points and end points in the parameter interval of the curve. The curve pointers point to nothing. If the curves given as input are degenerate, a closest point may be returned as a closest curve.

stat - Status messages
> 0 : warning
$= 0$: ok
< 0 : error

EXAMPLE OF USE

```
{
    SISLCurve    *curve1;   /* Must be defined */
    SISLCurve    *curve2;   /* Must be defined */
    double       epsco = 1.0e-9; /* Not used */
    double       epsge = 1.0e-6;
    int          numintpt = 0;
    double       *intpar1 = NULL;
    double       *intpar2 = NULL;
    int          numintcu = 0;
    SISLIntcurve **intcurve = NULL;
    int          stat = 0;
    ...
    s1955(curve1, curve2, epsco, epsge, &numintpt, &intpar1, &intpar2, &nu-
        mintcu, &intcurve, &stat);
    ...
}
```

4.6.5 Find a point on a 2D curve along a given direction.

NAME

s1013 - Find a point on a 2D curve along a given direction.

SYNOPSIS

void s1013(*pcurve*, *ang*, *ang_tol*, *guess_par*, *iter_par*, *jstat*)

```
SISLCurve   *pcurve;
double      ang;
double      ang_tol;
double      guess_par;
double      *iter_par;
int         *jstat;
```

ARGUMENTS

Input Arguments:

pcurve	-	Pointer to the curve.
ang	-	The angle (in radians) describing the wanted direction.
ang_tol	-	The angular tolerance (in radians).
guess_par	-	Start parameter value on the curve.

Output Arguments:

iter_par	-	The parameter value found on the curve.
stat	-	Status messages $= 2$: A minimum distance found. $= 1$: Intersection found. < 0 : Error.

EXAMPLE OF USE

```
{
    SISLCurve   *pcurve;  /* Must be defined */
    double      ang;  /* Must be defined */
    double      ang_tol = 0.01;
    double      guess_par;  /* Must be defined */
    double      iter_par;
    int         jstat = 0;
    ...
    s1013(pcurve, ang, ang_tol, guess_par, &iter_par, &jstat);
    ...
}
```

4.7 Find the Absolute Extremals of a Curve.

NAME

s1920 - Find the absolute extremal points/intervals of a curve relative to a given direction.

SYNOPSIS

void s1920(*curve*, *dir*, *dim*, *epsco*, *epsge*, *numintpt*, *intpar*,
numintcu, *intcurve*, *stat*)

SISLCurve	**curve*;
double	*dir*[ ];
int	*dim*;
double	*epsco*;
double	*epsge*;
int	**numintpt*;
double	***intpar*;
int	**numintcu*;
SISLIntcurve	****intcurve*;
int	**stat*;

ARGUMENTS

Input Arguments:

curve	-	Pointer to the curve.
dir	-	The direction in which the extremal point(s) and/or interval(s) are to be calculated. If $dim = 1$, a positive value indicates the maximum of the function and a negative value the minimum. If the dimension is greater than 1, the array contains the coordinates of the direction vector.
dim	-	Dimension of the space in which the curve and *dir* lie.
epsco	-	Computational resolution (not used).
epsge	-	Geometry resolution.

Output Arguments:

numintpt	-	Number of single extremal points.
intpar	-	Array containing the parameter values of the single extremal points in the parameter interval of the curve. The points lie in sequence. Extremal curves are stored in intcurve.
numintcu	-	Number of extremal curves.
intcurve	-	Array of pointers to the SISLIntcurve objects containing descriptions of the extremal curves. The curves are only described by start points and end points in the parameter interval of the curve. The curve pointers point to nothing.

stat	-	Status messages > 0 : Warning. $= 0$: Ok. < 0 : Error.

EXAMPLE OF USE

```
{
    SISLCurve      *curve;  /* Must be defined */
    double         dir[3];  /* Must be defined */
    int            dim = 3;
    double         epsco = 1.0e-9; /* Not used */
    double         epsge = 1.0e-6;
    int            numintpt = 0;
    double         *intpar = NULL;
    int            numintcu = 0;
    SISLIntcurve   **intcurve = NULL;
    int            stat = 0;
    ...
    s1920(curve, dir, dim, epsco, epsge, &numintpt, &intpar, &numintcu,
          &intcurve, &stat);
    ...
}
```

4.8 Area between Curve and Point

4.8.1 Calculate the area between a 2D curve and a 2D point.

NAME

s1241 - To calculate the area between a 2D curve and a 2D point. When the curve is rotating counter-clockwise around the point, the area contribution is positive. When the curve is rotating clockwise around the point, the area contribution is negative. If the curve is closed or periodic, the area calculated is independent of where the point is situated. The area is calculated exactly for B-spline curves, for NURBS the result is an approximation. This routine will only perform if the order of the curve is less than 7 (can easily be extended).

SYNOPSIS

void s1241(*pcurve*, *point*, *dim*, *epsge*, *area*, *stat*)

SISLCurve	**pcurve*;
double	*point*[ ];
int	*dim*;
double	*epsge*;
double	**area*;
int	**stat*;

ARGUMENTS

Input Arguments:

pcurve	-	The 2D curve.
point	-	The reference point.
dim	-	Dimension of geometry (must be 2).
epsge	-	Absolute geometrical tolerance.

Output Arguments:

area	-	Calculated area.
stat	-	Status messages > 0 : Warning. $= 0$: Ok. < 0 : Error.

EXAMPLE OF USE

```
{
    SISLCurve   *pcurve;  /* Must be defined */
    double      point[2];  /* Must be defined */
    int         dim = 2; /* Must be equal to 2 */
    double      epsge = 0.001;
    double      area;
    int         stat = 0;
    ...
    s1241(pcurve, point, dim, epsge, &area, &stat);
    ...
}
```

4.8.2 Calculate the weight point and rotational momentum of an area between a 2D curve and a 2D point.

NAME

s1243 - To calculate the weight point and rotational momentum of an area between a 2D curve and a 2D point. The area is also calculated. When the curve is rotating counter-clockwise around the point, the area contribution is positive. When the curve is rotating clockwise around the point, the area contribution is negative. OBSERVE: FOR CALCULATION OF AREA ONLY, USE s1241().

SYNOPSIS

```
void s1243(pcurve, point, dim, epsge, weight, area, moment, stat)
    SISLCurve   *pcurve;
    double      point[ ];
    int         dim;
    double      epsge;
    double      weight[ ];
    double      *area;
    double      *moment;
    int         *stat;
```

ARGUMENTS

Input Arguments:

pcurve	-	The 2D curve.
point	-	The reference point.
dim	-	Dimension of geometry (must be 2).
epsge	-	Absolute geometrical tolerance.

Output Arguments:

weight	-	Weight point.
area	-	Area.
moment	-	Rotational momentum.
stat	-	Status messages > 0 : warning $= 0$: ok < 0 : error

EXAMPLE OF USE

```
{
    SISLCurve   *pcurve;  /* Must be defined */
    double      point[2];  /* Must be defined */
    int         dim = 2; /* Dimension 2 is required */
    double      epsge = 0.01;
    double      weight[2];
    double      area = 0.0;
    double      moment = 0.0;
    int         stat = 0;
    . . .
    s1243(pcurve, point, dim, epsge, weight, &area, &moment, &stat);
```

```
        ...
}
```

4.9 Bounding Box

Both curves and surfaces have bounding boxes. These are boxes surrounding an object not only parallel to the main axis, but also rotated 45 degrees around each main axis. These bounding boxes are used by the intersection functions to decide if an intersection is possible or not. They might also be used to find the position of objects under other circumstances.

4.9.1 Bounding box object.

In the library a bounding box is stored in a struct SISLbox containing the following:

double	*emax;	Allocated array containing the minimum values of the bounding box
double	*emin;	Allocated array containing the maximum values of the bounding box
int	imin;	The index of the minimum coefficient *ecoef*[*imin*]. Only used in dimension one. *ecoef* is the control polygon of the curve/surface.
int	imax;	The index of the maximum coefficient *ecoef*[*imax*]. Only used in dimension one. *ecoef* is the control polygon of the curve/surface.

4.9.2 Create and initialize a curve/surface bounding box instance.

NAME

newbox - Create and initialize a curve/surface bounding box instance.

SYNOPSIS

SISLbox *newbox(*idim*)
    int *idim*;

ARGUMENTS

Input Arguments:

idim	-	Dimension of geometry space.

Output Arguments:

newbox	-	Pointer to new SISLbox structure. If it is impossible to allocate space for the structure, newbox will return a NULL value.

EXAMPLE OF USE

```
{
    int        idim = 3;
    SISLbox *box = NULL;
    ...
    box = newbox(idim);
    ...
}
```

4.9.3 Find the bounding box of a curve.

NAME

s1988 - Find the bounding box of a SISLCurve. NB. The geometric bounding box is returned also in the rational case, that is the box in homogenous coordinates is NOT computed.

SYNOPSIS

void s1988(*pc*, *emax*, *emin*, *jstat*)

SISLCurve	**pc*;
double	***emax*;
double	***emin*;
int	**jstat*;

ARGUMENTS

Input Arguments:

pc	-	The curve to treat.

Output Arguments:

emin	-	Array of dimension *idim* containing the minimum values of the bounding box, i.e. bottom-left corner of the box.
emax	-	Array of dimension *idim* containing the maximum values of the bounding box, i.e. upper-right corner of the box.
jstat	-	Status message < 0 : Error. $= 0$: Ok. > 0 : Warning.

EXAMPLE OF USE

```
{
    SISLCurve   *pc;   /* Must be defined */
    double      *emax = NULL;
    double      *emin = NULL;
    int         jstat = 0;
    ...
    s1988(pc, &emax, &emin, &jstat);
    ...
}
```

4.10 Normal Cone

Both curves and surfaces have normal cones. These are the cones that are convex hull of all normalized tangents of a curve and all normalized normals of a surface.

These normal cones are used by the intersection functions to decide if only one intersection is possible. They might also be used to find directions of objects for other reasons.

4.10.1 Normal cone object.

In the library a direction cone is stored in a struct SISLdir containing the following:

int	*igtpi*;	To mark if the angle of direction cone is greater than π. $= 0$: The direction of a surface and its boundary curves or a curve is not greater than π in any parameter direction. $= 1$: The direction of a surface or a curve is greater than π in the first parameter direction. $= 2$: The angle of direction cone of a surface is greater than π in the second parameter direction. $= 10$: The angle of direction cone of a boundary curve in first parameter direction of a surface is greater than π. $= 20$: The angle of direction cone of a boundary curve in second parameter direction of a surface is greater than π.
double	**ecoef*;	Allocated array containing the coordinates of the centre of the cone.
double	*aang*;	The angle from the centre which describes the cone.

4.10.2 Create and initialize a curve/surface direction instance.

NAME

newdir - Create and initialize a curve/surface direction instance.

SYNOPSIS

SISLdir *newdir(*idim*)

    int *idim*;

ARGUMENTS

Input Arguments:

idim	-	Dimension of the space in which the object lies.

Output Arguments:

newdir	-	Pointer to new direction structure. If it is impossible to allocate space for the structure, newdir will return a NULL value.

EXAMPLE OF USE

```
{
    int        idim = 3;
    SISLdir    *dir = NULL;
    ...
    dir = newdir(idim);
    ...
}
```

4.10.3 Find the direction cone of a curve.

NAME

s1986 - Find the direction cone of a curve.

SYNOPSIS

void s1986(*pc*, *aepsge*, *jgtpi*, *gaxis*, *cang*, *jstat*)

SISLCurve	**pc*;
double	*aepsge*;
int	**jgtpi*;
double	***gaxis*;
double	**cang*;
int	**jstat*;

ARGUMENTS

Input Arguments:

pc	-	The curve to treat.
aepsge	-	Geometry tolerance.

Output Arguments:

jgtpi	-	To mark if the angle of the direction cone is greater than π. $= 0$ The direction cone of the curve $\leq \pi$. $= 1$ The direction cone of the curve $> \pi$.
gaxis	-	Allocated array containing the coordinates of the centre of the cone. It is only computed if $jgtpi = 0$.
cang	-	The angle from the centre to the boundary of the cone. It is only computed if $jgtpi = 0$.
jstat	-	Status messages > 0 : Warning. $= 0$: Ok. < 0 : Error.

EXAMPLE OF USE

```
{
    SISLCurve   *pc;  /* Must be defined */
    double      aepsge = 1.0e-10;
    int         jgtpi = 0;
    double      *gaxis = NULL;
    double      cang = 0.0;
    int         jstat = 0;
    ...
    s1986(pc, aepsge, &jgtpi, &gaxis, &cang, &jstat);
    ...
}
```

Chapter 5

Curve Analysis

This chapter describes the Curve Analysis part.

5.1 Curvature Evaluation

5.1.1 Evaluate the curvature of a curve at given parameter values.

NAME

s2550 - Evaluate the curvature of a curve at given parameter values ax[0],...,ax[num_ax - 1].

SYNOPSIS

void s2550(*curve*, *ax*, num_ *ax*, *curvature*, jstat)

SISLCurve	**curve*;
double	*ax*[ ];
int	*num*_ax;
double	*curvature*[ ];
int	**jstat*;

ARGUMENTS

Input Arguments:

curve	-	Pointer to the curve.
ax	-	The parameter values
num	-	No. of parameter values

Output Arguments:

	-	
curvature	-	The "num_ax" curvature values computed
jstat	-	Status messages > 0 : Warning. $= 0$: Ok. < 0 : Error.

EXAMPLE OF USE

```
{
    SISLCurve    *curve;   /* Must be defined */
    double       ax[10];   /* Must be defined */
    int          num_ax = 10;
    double       curvature[10]; /* Size equal to num_ax */
    int          jstat = 0;
    ...
    s2550(curve, ax, num_ ax, curvature, &jstat );
    ...
}
```

5.1.2 Evaluate the torsion of a curve at given parameter values.

NAME

s2553 - Evaluate the torsion of a curve at given parameter values ax[0],...,ax[num_ax - 1].

SYNOPSIS

```
void s2553(curve, ax, num_ ax, torsion, jstat )
    SISLCurve   *curve;
    double      ax[ ];
    int         num_ax;
    double      torsion[ ];
    int         *jstat;
```

ARGUMENTS

Input Arguments:

curve	-	Pointer to the curve.
ax	-	The parameter values
num	-	No. of parameter values

Output Arguments:

	-	
torsion	-	The "num_ax" torsion values computed
jstat	-	Status messages > 0 : Warning. $= 0$: Ok. < 0 : Error.

EXAMPLE OF USE

```
{
    SISLCurve   *curve;   /* Must be defined */
    double      ax[10];   /* Must be defined */
    int         num_ax = 10;
    double      torsion[10]; /* Size equal to num_ax */
    int         jstat = 0;
    ...
    s2553(curve, ax, num_ ax, torsion, &jstat );
    ...
}
```

5.1.3 Evaluate the Variation of Curvature (VoC) of a curve at given parameter values.

NAME

s2556 - Evaluate the Variation of Curvature (VoC) of a curve at given parameter values ax[0],...,ax[num_ax - 1].

SYNOPSIS

```
void s2556(curve, ax, num_ ax, VoC, jstat )
    SISLCurve   *curve;
    double      ax[ ];
    int         num_ax;
    double      VoC[ ];
    int         *jstat;
```

ARGUMENTS

Input Arguments:

curve	-	Pointer to the curve.
ax	-	The parameter values
num	-	No. of parameter values

Output Arguments:

	-	
VoC	-	The "num_ax" Variation of Curvature (VoC) values computed
jstat	-	Status messages > 0 : Warning. = 0 : Ok. < 0 : Error.

EXAMPLE OF USE

```
{
    SISLCurve   *curve;  /* Must be defined */
    double      ax[10];  /* Must be defined */
    int         num_ax = 10;
    double      VoC[10]; /* Size equal to num_ax */
    int         jstat = 0;
    ...
    s2556(curve, ax, num_ ax, VoC, &jstat );
    ...
}
```

5.1.4 Evaluate the Frenet Frame (t,n,b) of a curve at given parameter values.

NAME

s2559 - Evaluate the Frenet Frame (t,n,b) of a curve at given parameter values ax[0],...,ax[num_ax - 1].

SYNOPSIS

void s2559(*curve*, *ax*, num_ *ax*, *p*, *t*, *n*, *b*, jstat)

```
SISLCurve   *curve;
double      ax[ ];
int         num_ax;
double      p[ ];
double      t[ ];
double      n[ ];
double      b[ ];
int         *jstat;
```

ARGUMENTS

Input Arguments:

curve	-	Pointer to the curve.
ax	-	The parameter values
num	-	No. of parameter values

Output Arguments:

	-	
t	-	The Frenet Frame (in 3D) computed. Each of the arrays (t,n,b) are of dim. 3*num_ax, and the data are stored like this: tx(ax[0]), ty(ax[0]), tz(ax[0]), ...,tx(ax[num_ax-1]), ty(ax[num_ax-1]), tz(ax[num_ax-1]).
p	-	1]
jstat	-	Status messages > 0 : Warning. = 0 : Ok. < 0 : Error.

EXAMPLE OF USE

```
{
    SISLCurve   *curve;  /* Must be defined */
    double      ax[10];
    int         num_ax = 10;
    double      p[10]; /* Size equal to num_ax */
    double      t[10]; /* Size equal to num_ax */
    double      n[10]; /* Size equal to num_ax */
    double      b[10]; /* Size equal to num_ax */
    int         jstat = 0;
    . . .
    s2559(curve, ax, num_ ax, p, t, n, b, &jstat );
    . . .
}
```

5.1.5 Evaluate geometric properties at given parameter values.

NAME

s2562 - Evaluate the 3D position, the Frenet Frame (t,n,b) and geometric property (curvature, torsion or variation of curvature) of a curve at given parameter values ax[0],...,ax[num_ax-1]. These data are needed to produce spike plots (using the Frenet Frame and the geometric property) and circular tube plots (using circular in the normal plane (t,b), where the radius is equal to the geometric property times a scaling factor for visual effects).

SYNOPSIS

void s2562(*curve*, *ax*, num_ *ax*, val_ *flag*, *p*, *t*, *n*, *b*, *val*, jstat)

SISLCurve	**curve*;
double	*ax*[ ];
int	*num*_ax;
int	*val*_flag;
double	*p*[ ];
double	*t*[ ];
double	*n*[ ];
double	*b*[ ];
double	*val*[ ];
int	**jstat*;

ARGUMENTS

Input Arguments:

curve	-	Pointer to the curve.
ax	-	The parameter values
num	-	No. of parameter values
val	-	Compute geometric property = 1 : curvature = 2 : torsion = 3 : variation of curvature

Output Arguments:

	-	
t	-	The Frenet Frame (in 3D) computed. Each of the arrays (t,n,b) are of dim. 3*num_ax, and the data are stored like this: tx(ax[0]), ty(ax[0]), tz(ax[0]), ...,tx(ax[num_ax-1]), ty(ax[num_ax-1]), tz(ax[num_ax-1]).
p	-	1]
val	-	Geometric property (curvature, torsion or variation of curvature) of a curve at given parameter values ax[0],...,ax[num_ax-1].
jstat	-	Status messages > 0 : Warning. $= 0$: Ok. < 0 : Error.

EXAMPLE OF USE

```
{
    SISLCurve   *curve;   /* Must be defined */
    double      ax[10];   /* Must be defined */
    int         num_ax = 10;
    int         val_flag = 1;
    double      p[30]; /* Size equal to dimension times num_ax */
    double      t[30]; /* Size equal to dimension times num_ax */
    double      n[30]; /* Size equal to dimension times num_ax */
    double      b[30]; /* Size equal to dimension times num_ax */
    double      val[10]; /* Size equal to num_ax */
    int         jstat = 0;
    . . .
    s2562(curve, ax, num_ ax, val_ flag, p, t, n, b, val, &jstat );
    . . .
}
```

Chapter 6

Curve Utilities

This chapter describes the Curve Utilities. These are common to both the Curve Definition and Curve Interrogation modules.

6.1 Curve Object

In the library both B-spline and NURBS curves are stored in a struct SISLCurve containing the following:

int	*ik*;	Order of curve.
int	*in*;	Number of vertices.
double	**et*;	Pointer to the knot vector.
double	**ecoef*;	Pointer to the array containing non-rational vertices, size $in \times idim$.
double	**rcoef*;	Pointer to the array of rational vertices and weights, size $in \times (idim + 1)$.
int	*ikind*;	Type of curve = 1 : Polynomial B-spline curve. = 2 : Rational B-spline (nurbs) curve. = 3 : Polynomial Bezier curve. = 4 : Rational Bezier curve.
int	*idim*;	Dimension of the space in which the curve lies.
int	*icopy*;	Indicates whether the arrays of the curve are allocated and copied or referenced by creation of the curve. = 0 : Pointer set to input arrays. The arrays are not deleted by freeCurve. = 1 : Array allocated and copied. The arrays are deleted by freeCurve. = 2 : Pointer set to input arrays, but are to be treated as copied. The arrays are deleted by freeCurve.
SISLdir	**pdir*;	Pointer to a SISLdir object used for storing curve direction.
SISLbox	**pbox*;	Pointer to a SISLbox object used for storing the surrounding boxes.
int	*cuopen*;	Open/closed/periodic flag.

$= -1$:	Closed curve with periodic (cyclic) parameterization and overlapping end vertices.
$= 0$:	Closed curve with k-tuple end knots and coinciding start/end vertices.
$= 1$:	Open curve (default).

Note that in the rational case are the curve coefficients stored as $w_1\mathbf{p}_1, w_1, w_2\mathbf{p}_2, w_2, \ldots, w_n\mathbf{p}_n, w_n$ where w_i are the weights and $\mathbf{p}_i$, $i = 1, \ldots, n$ are the curve coefficients.

When using a curve, do not declare a SISLCurve but a pointer to a SISLCurve, and initialize it to point on NULL. Then you may use the dynamic allocation functions newCurve and freeCurve described below, to create and delete curves.

There are two ways to pass coefficient and knot arrays to newCurve. By setting *icopy* $= 1$, newCurve allocates new arrays and copies the given ones. But by setting *icopy* $= 0$ or 2, newCurve simply points to the given arrays. Therefore it is IMPORTANT that the given arrays have been allocated in free memory beforehand.

6.1.1 Create new curve object.

NAME

newCurve - Create and initialize a SISLCurve-instance. Note that the vertex input to a rational curve is unstandard. Given the curve

$$\mathbf{c}(t) = \frac{\sum_{i=1}^{n} w_i \mathbf{p}_i B_{i,k,\mathbf{t}}(t)}{\sum_{i=1}^{n} w_i B_{i,k,\mathbf{t}}(t)},$$

must the vertices be given as $w_1\mathbf{p}_1, w_1, w_2\mathbf{p}_2, w_2, \ldots, w_n\mathbf{p}_n, w_n$ when invoking this function. Thus the vertices are multiplied with the associated weight.

SYNOPSIS

SISLCurve *newCurve(*number*, *order*, *knots*, *coef*, *kind*, *dim*, *copy*)

int	*number*;
int	*order*;
double	*knots*[ ];
double	*coef*[ ];
int	*kind*;
int	*dim*;
int	*copy*;

ARGUMENTS

Input Arguments:

number	-	Number of vertices in the new curve.
order	-	Order of curve.
knots	-	Knot vector of curve.
coef	-	Vertices of curve. These can either be the *dim* dimensional non-rational vertices, or the (*dim*+1) dimensional rational vertices.
kind	-	Type of curve. = 1 : Polynomial B-spline curve. = 2 : Rational B-spline (nurbs) curve. = 3 : Polynomial Bezier curve. = 4 : Rational Bezier curve.
dim	-	Dimension of the space in which the curve lies.
copy	-	Flag = 0 : Set pointer to input arrays. = 1 : Copy input arrays. = 2 : Set pointer and remember to free arrays.

Output Arguments:

newCurve	-	Pointer to the new curve. If it is impossible to allocate space for the curve, newCurve returns NULL.

EXAMPLE OF USE

```
{
    SISLCurve   *curve = NULL;
    int         number = 10;
    int         order = 4;
    double      knots[14];  /* Must be defined */
    double      coef[30];   /* Must be defined */
    int         kind = 1; /* Non-rational */
    int         dim = 3;
    int         copy = 1;
    ...
    curve = newCurve(number, order, knots, coef, kind, dim, copy);
    ...
}
```

6.1.2 Make a copy of a curve.

NAME

copyCurve - Make a copy of a curve.

SYNOPSIS

SISLCurve *copyCurve(*pcurve*)

SISLCurve **pcurve*;

ARGUMENTS

Input Arguments:

pcurve	-	Curve to be copied.

Output Arguments:

copyCurve	-	The new curve.

EXAMPLE OF USE

```
{
    SISLCurve   *curvecopy = NULL;
    SISLCurve   *curve = NULL;
    int         number = 10;
    int         order = 4;
    double      knots[14]; /* Must be defined */
    double      coef[30];  /* Must be defined */
    int         kind = 1; /* Non-rational */
    int         dim = 3;
    int         copy = 1;
    ...
    curve = newCurve(number, order, knots, coef, kind, dim, copy);
    ...
    curvecopy = copyCurve(curve);
    ...
}
```

6.1.3 Delete a curve object.

NAME

freeCurve - Free the space occupied by the curve. Before using freeCurve, make sure the curve object exists.

SYNOPSIS

void freeCurve(*curve*)

SISLCurve **curve*;

ARGUMENTS

Input Arguments:

curve - Pointer to the curve to delete.

EXAMPLE OF USE

```
{
    SISLCurve   *curve = NULL;
    int         number = 10;
    int         order = 4;
    double      knots[14];
    double      coef[30];
    int         kind = 1;
    int         dim = 3;
    int         copy = 1;
    ...
    curve = newCurve(number, order, knots, coef, kind, dim, copy);
    ...
    if (curve) freeCurve(curve);
    ...
}
```

6.2 Evaluation

6.2.1 Compute the position and the left-hand derivatives of a curve at a given parameter value.

NAME

s1227 - To compute the position and the first derivatives of the curve at a given parameter value Evaluation from the left hand side.

SYNOPSIS

void s1227(*curve*, *der*, *parvalue*, *leftknot*, *derive*, *stat*)

SISLCurve	**curve*;
int	*der*;
double	*parvalue*;
int	**leftknot*;
double	*derive*[ ];
int	**stat*;

ARGUMENTS

Input Arguments:

curve - Pointer to the curve for which position and derivatives are to be computed.

der - The number of derivatives to compute.
< 0 : Error.
$= 0$: Compute position.
$= 1$: Compute position and derivative.
etc.

parvalue - The parameter value at which to compute position and derivatives.

Input/Output Arguments:

leftknot - Pointer to the interval in the knot vector where *parvalue* is located. If *et*[ ] is the knot vector, the relation:

$$et[\text{leftknot}] < parvalue \leq et[\text{leftknot} + 1]$$

should hold. (If $parvalue \leq et[ik-1]$) then *leftknot* should be “ik-1”. Here “ik” is the order of the curve.) If leftknot does not have the right value when entering the routine, its value will be changed to the value satisfying the above condition.

Output Arguments:

derive - Double array of dimension $(der + 1) \times dim$ containing the position and derivative vectors. (dim is the dimension of the Euclidean space in which the curve lies.) These vectors are stored in the following order: first the components of the position vector, then the dim components of the tangent vector, then the dim components of the second derivative vector, and so on. (The C declaration of derive as a two dimensional array would therefore be $derive[der + 1][dim]$.)

stat - Status messages
> 0 : warning
$= 0$: ok
< 0 : error

EXAMPLE OF USE

```
{
    SISLCurve   *curve;  /* Must be defined */
    int         der = 3;
    double      parvalue;  /* Must be defined */
    int         leftknot = 0; /* Define initially as zero. For consequtive evaluations
                                   leave leftknot as returned from s1227 */
    double      derive[12]; /* Curve dimension times (der+1) */
    int         stat = 0;
    ...
    s1227(curve, der, parvalue, &leftknot, derive, &stat);
    ...
}
```

6.2.2 Compute the position and the right-hand derivatives of a curve at a given parameter value.

NAME

s1221 - To compute the positione and the first derivatives of a curve at a given parameter value. Evaluation from the right hand side.

SYNOPSIS

void s1221(*curve*, *der*, *parvalue*, *leftknot*, *derive*, *stat*)

SISLCurve	**curve*;
int	*der*;
double	*parvalue*;
int	**leftknot*;
double	*derive*[ ];
int	**stat*;

ARGUMENTS

Input Arguments:

curve - Pointer to the curve for which position and derivatives are to be computed.

der - The number (order) of derivatives to compute.
< 0 : Error.
$= 0$: Compute position.
$= 1$: Compute position and derivative.
etc.

parvalue - The parameter value at which to compute position and derivatives.

Input/Output Arguments:

leftknot - Pointer to the interval in the knot vector where *parvalue* is located. If *et*[ ] is the knot vector, the relation:

$$et[leftknot] \leq parvalue < et[leftknot + 1]$$

should hold. (If $parvalue \geq et[in]$) then *leftknot* should be “in-1”. Here “in” is the number of coefficients.) If leftknot does not have the right value when entering the routine, its value will be changed to the value satisfying the above condition.

Output Arguments:

derive	-	Double array of dimension $(der + 1) \times dim$ containing the position and derivative vectors. (dim is the dimension of the Euclidean space in which the curve lies.) These vectors are stored in the following order: first the dim components of the position vector, then the dim components of the tangent vector, then the dim components of the second derivative vector, and so on. (The C declaration of derive as a two dimensional array would therefore be $derive[der+1][dim]$.)
stat	-	Status messages > 0 : warning $= 0$: ok < 0 : error

EXAMPLE OF USE

```
{
    SISLCurve   *curve;   /* Must be defined */
    int         der = 3;
    double      parvalue;   /* Must be defined */
    int         leftknot = 0; /* Define initially as zero. For consequtive evaluations
                                 leave leftknot as returned from s1221 */
    double      derive[12]; /* Curve dimension times (der+1) */
    int         stat = 0;
    ...
    s1221(curve, der, parvalue, &leftknot, derive, &stat);
    ...
}
```

6.2.3 Evaluate position, first derivative, curvature and radius of curvature of a curve at a given parameter value, from the left hand side.

NAME

s1225 - Evaluate position, derivatives, curvature and radius of curvature of a curve at a given parameter value, from the left hand side.

SYNOPSIS

void s1225(*curve*, *der*, *parvalue*, *leftknot*, *derive*, *curvature*, radius_of_*curvature*, *jstat*)

SISLCurve	**curve*;
int	*der*;
double	parvalue;
int	**leftknot*;
double	derive[ ];
double	curvature[ ];
double	**radius*_of_curvature;
int	**jstat*;

ARGUMENTS

Input Arguments:

curve - Pointer to the curve for which position and derivatives are to be computed.

der - The number of derivatives to compute.
- < 0 : Error.
- $= 0$: Compute position.
- $= 1$: Compute position and first derivative.
- etc.

parvalue - The parameter value at which to compute position and derivatives.

Input/Output Arguments:

leftknot - Pointer to the interval in the knot vector where ax is located. If et is the knot vector, the relation

$$et[ileft] < parvalue <= et[ileft + 1]$$

should hold. (If parvalue = et[ik-1] then ileft should be ik-1. Here in is the number of B-spline coefficients.) If ileft does not have the right value upon entry to the routine, its value will be changed to the value satisfying the above condition.

Output Arguments:

derive	-	Double array of dimension $[(ider + 1) * idim]$ containing the position and derivative vectors. (idim is the number of components of each B-spline coefficient, i.e. the dimension of the Euclidean space in which the curve lies.) These vectors are stored in the following order: First the idim components of the position vector, then the idim components of the tangent vector, then the idim components of the second derivative vector, and so on. (The C declaration of eder as a two dimensional array would therefore be eder[ider+1,idim].)
curvature	-	Array of dimension idim
radius	-	1, indicates that the radius of curvature is infinit.
jstat	-	Status messages > 0 : Warning. $= 0$: Ok. < 0 : Error.

EXAMPLE OF USE

```
{
    SISLCurve   *curve;  /* Must be defined */
    int         der = 1;
    double      parvalue;  /* Must be defined */
    int         leftknot = 0; /* Define initially as zero. For consequtive evaluations
                                 leave leftknot as returned from s1225 */
    double      derive[6]; /* Curve dimension times (der + 1) */
    double      curvature[3]; /* Curve dimension */
    double      radius_of_curvature = 0;
    int         jstat = 0;
    ...
    s1225(curve, der, parvalue, leftknot, derive, curvature, &radius_of_curvature,
          &jstat);
    ...
}
```

6.2.4 Evaluate position, first derivative, curvature and radius of curvature of a curve at a given parameter value, from the right hand side.

NAME

s1226 - Evaluate position, derivatives, curvature and radius of curvature of a curve at a given parameter value, from the right hand side.

SYNOPSIS

void s1226(*curve*, *der*, *parvalue*, *leftknot*, *derive*, *curvature*, radius_of_*curvature*, *jstat*)

SISLCurve	**curve*;
int	*der*;
double	parvalue;
int	**leftknot*;
double	derive[ ];
double	curvature[ ];
double	**radius*_of_curvature;
int	**jstat*;

ARGUMENTS

Input Arguments:

curve - Pointer to the curve for which position and derivatives are to be computed.

der - The number of derivatives to compute.
< 0 : Error.
$= 0$: Compute position.
$= 1$: Compute position and first derivative.
etc.

parvalue - The parameter value at which to compute position and derivatives.

Input/Output Arguments:

leftknot - Pointer to the interval in the knot vector where ax is located. If et is the knot vector, the relation

$$et[ileft] < parvalue <= et[ileft + 1]$$

should hold. (If parvalue = et[ik-1] then ileft should be ik-1. Here in is the number of B-spline coefficients.) If ileft does not have the right value upon entry to the routine, its value will be changed to the value satisfying the above condition.

Output Arguments:

derive	-	Double array of dimension [(ider+1)*idim] containing the position and derivative vectors. (idim is the number of components of each B-spline coefficient, i.e. the dimension of the Euclidean space in which the curve lies.) These vectors are stored in the following order: First the idim components of the position vector, then the idim components of the tangent vector, then the idim components of the second derivative vector, and so on. (The C declaration of eder as a two dimensional array would therefore be eder[ider+1,idim].)
curvature	-	Array of dimension idim
radius	-	1, indicates that the radius of curvature is infinit.
jstat	-	Status messages > 0 : Warning. $= 0$: Ok. < 0 : Error.

EXAMPLE OF USE

```
{
    SISLCurve   *curve;   /* Must be defined */
    int         der = 1;
    double      parvalue;   /* Must be defined */
    int         leftknot = 0; /* Define initially as zero. For consequtive evaluations
                                 leave leftknot as returned from s1226 */
    double      derive[6]; /* Geometry space dimension times (der + 1) */
    double      curvature[3]; /* Geometry space dimension */
    double      radius_of_curvature = 0;
    int         jstat = 0;
    ...
    s1226(curve, der, parvalue, leftknot, derive, curvature, &radius_of_curvature,
          &jstat);
    ...
}
```

6.2.5 Evaluate the curve over a grid of m points. Only positions are evaluated.

NAME

s1542 - Evaluate the curve pointed at by pc1 over a m grid of points (x[i]). Only positions are evaluated. Do not apply in the rational case.

SYNOPSIS

```
void s1542(pc1, m, x, eder, jstat)
    SISLCurve   *pc1;
    int         m;
    double      x[ ];
    double      eder[ ];
    int         *jstat;
```

ARGUMENTS

Input Arguments:

pc1	-	Pointer to the curve to evaluate.
m	-	Number of grid points.
x	-	Array of parameter values of the grid.

Output Arguments:

eder	-	Array where the positions of the curve are placed, dimension idim * m. The sequence is position at point x[0], followed by the same information at x[1], etc.
jstat	-	status messages = 0 : Ok. < 0 : Error.

EXAMPLE OF USE

```
{
    SISLCurve   *pc1;   /* Must be defined */
    int         m = 25;
    double      x[25];
    double      eder[75]; /* Geometry space dimension times m */
    int         jstat = 0;
    ...
    s1542(pc1, m, x, eder, &jstat);
    ...
}
```

6.3 Subdivision

6.3.1 Subdivide a curve at a given parameter value.

NAME

s1710 - Subdivide a curve at a given parameter value.
NOTE: When the curve is periodic (i.e. when the *cuopen* flag of the curve has value $= -1$), this function will return only ONE curve through *rcnew1*. This curve is the same geometric curve as *pc1*, but is represented on a closed basis, i.e. with k-tuple start/end knots and coinciding start/end coefficients. The *cuopen* flag of the curve will then be set to closed ($= 0$) and a status value *jstat* equal to 2 will be returned.

SYNOPSIS

void s1710(*pc1*, *apar*, *rcnew1*, *rcnew2*, *jstat*)

SISLCurve	**pc1*;
double	*apar*;
SISLCurve	***rcnew1*;
SISLCurve	***rcnew2*;
int	**jstat*;

ARGUMENTS

Input Arguments:

pc1	-	The curve to subdivide.
apar	-	Parameter value at which to subdivide.

Output Arguments:

rcnew1	-	First part of the subdivided curve.
rcnew2	-	Second part of the subdivided curve. If the parameter value is at the end of a curve NULL pointers might be returned
jstat	-	Status messages

$= 5$:	Parameter value at end of curve, *rcnew1*=NULL or *rcnew2*=NULL.
$= 2$:	*pc1* periodic, *rcnew2*=NULL.
> 0 :	Warning.
$= 0$:	Ok.
< 0 :	Error.

EXAMPLE OF USE

```
{
    SISLCurve   *pc1;   /* Must be defined */
    double      apar;   /* Must be defined */
    SISLCurve   *rcnew1 = NULL;
    SISLCurve   *rcnew2 = NULL;
    int         jstat = 0;
    ...
s1710(pc1, apar, &rcnew1, &rcnew2, &jstat);
    ...
}
```

6.3.2 Insert a given knot into the description of a curve.

NAME

s1017 - Insert a given knot into the description of a curve.
NOTE : When the curve is periodic (i.e. the curve flag *cuopen* $= -1$), the input parameter value must lie in the half-open $[et[kk-1], et[kn])$ interval, the function will automatically update the extra knots and coeffisients. *rcnew*->*in* is still equal to *pc*->*in* $+ 1$!

SYNOPSIS

void s1017(*pc*, *rc*, *apar*, *jstat*)

SISLCurve	**pc*;
SISLCurve	***rc*;
double	*apar*;
int	**jstat*;

ARGUMENTS

Input Arguments:

pc	-	The curve to be refined.
apar	-	Parameter value of the knot to be inserted.

Output Arguments:

rc	-	The new, refined curve.
jstat	-	Status message > 0 : Warning. $= 0$: Ok. < 0 : Error.

EXAMPLE OF USE

```
{
    SISLCurve   *pc;   /* Must be defined */
    double      apar;  /* Must be defined */
    SISLCurve   *rc = NULL;
    int         jstat = 0;
    ...
    s1017(pc, &rc, apar, &jstat);
    ...
}
```

6.3.3 Insert a given set of knots into the description of a curve.

NAME

s1018 - Insert a given set of knots into the description of a curve.
NOTE : When the curve is periodic (i.e. when the curve flag *cuopen* $= -1$), the input parameter values must lie in the half-open $[et[kk-1], et[kn])$, the function will automatically update the extra knots and coeffisients. The *rcnew*->*in* will still be equal to *pc*->*in* + *inpar*.

SYNOPSIS

void s1018(*pc*, *epar*, *inpar*, *rcnew*, *jstat*)

SISLCurve	**pc*;
double	*epar*[ ];
int	*inpar*;
SISLCurve	***rcnew*;
int	**jstat*;

ARGUMENTS

Input Arguments:

pc	-	The curve to be refined.
epar	-	Knots to be inserted. The values are stored in increasing order and may be multiple.
inpar	-	Number of knots in *epar*.

Output Arguments:

rcnew	-	The new, refined curve.
jstat	-	Status message > 0 : Warning. $= 0$: Ok. < 0 : Error.

EXAMPLE OF USE

```
{
    SISLCurve   *pc;  /* Must be defined */
    double      epar[5]; /* Must be defined */
    int         inpar = 5;
    SISLCurve   *rcnew = NULL;
    int         jstat = 0;
    ...
    s1018(pc, epar, inpar, &rcnew, &jstat);
    ...
}
```

6.3.4 Split a curve into two new curves.

NAME

s1714 - Split a curve in two parts at two specified parameter values. The first curve starts at *parval1*. If the curve is open, the last part of the curve is translated so that the end of the curve joins the start.

SYNOPSIS

void s1714(*curve*, *parval1*, *parval2*, *newcurve1*, *newcurve2*, *stat*)

```
SISLCurve   *curve;
double      parval1;
double      parval2;
SISLCurve   **newcurve1;
SISLCurve   **newcurve2;
int         *stat;
```

ARGUMENTS

Input Arguments:

curve	-	The curve to split.
parval1	-	Start parameter value of the first new curve.
parval2	-	Start parameter value of the second new curve.

Output Arguments:

newcurve1	-	The first new curve.
newcurve2	-	The second new curve.
stat	-	Status messages > 0 : warning $= 0$: ok < 0 : error

EXAMPLE OF USE

```
{
    SISLCurve   *curve;  /* Must be defined */
    double      parval1;  /* Must be defined */
    double      parval2;  /* Must be defined */
    SISLCurve   *newcurve1 = NULL;
    SISLCurve   *newcurve2 = NULL;
    int         stat = 0;
    ...
    s1714(curve, parval1, parval2, &newcurve1, &newcurve2, &stat);
    ...
}
```

6.3.5 Pick a part of a curve.

NAME

s1712 - To pick one part of a curve and make a new curve of the part. If *endpar* $<$ *begpar* the direction of the new curve is turned. Use s1713() to pick a curve part crossing the start/end points of a closed (or periodic) curve.

SYNOPSIS

void s1712(*curve*, *begpar*, *endpar*, *newcurve*, *stat*)

```
SISLCurve   *curve;
double      begpar;
double      endpar;
SISLCurve   **newcurve;
int         *stat;
```

ARGUMENTS

Input Arguments:

curve	-	The curve to pick a part from.
begpar	-	Start parameter value of the part curve to be picked.
endpar	-	End parameter value of the part curve to be picked.

Output Arguments:

newcurve	-	The new curve that is a part of the original curve.
stat	-	Status messages > 0 : warning $= 0$: ok < 0 : error

EXAMPLE OF USE

```
{
    SISLCurve   *curve;   /* Must be defined */
    double      begpar;   /* Must be defined */
    double      endpar;   /* Must be defined */
    SISLCurve   *newcurve = NULL;
    int         stat = 0;
    ...
    s1712(curve, begpar, endpar, &newcurve, &stat);
    ...
}
```

6.3.6 Pick a part of a closed curve.

NAME

s1713 - To pick one part of a closed curve and make a new curve of that part. If the routine is used on an open curve and $endpar \leq begpar$, the last part of the curve is translated so that the end of the curve joins the start.

SYNOPSIS

void s1713(*curve*, *begpar*, *endpar*, *newcurve*, *stat*)

SISLCurve	**curve*;
double	*begpar*;
double	*endpar*;
SISLCurve	***newcurve*;
int	**stat*;

ARGUMENTS

Input Arguments:

curve	-	The curve to pick a part from.
begpar	-	Start parameter value of the part of the curve to be picked.
endpar	-	End parameter value of the part of the curve to be picked.

Output Arguments:

newcurve	-	The new curve that is a part of the original curve.
stat	-	Status messages > 0 : warning $= 0$: ok < 0 : error

EXAMPLE OF USE

```
{
    SISLCurve   *curve;   /* Must be defined */
    double      begpar;   /* Must be defined */
    double      endpar;   /* Must be defined */
    SISLCurve   *newcurve = NULL;
    int         stat = 0;
    ...
    s1713(curve, begpar, endpar, &newcurve, &stat);
    ...
}
```

6.4 Joining

6.4.1 Join two curves at specified ends.

NAME

s1715 - To join one end of one curve with one end of another curve by translating the second curve. If *curve1* is to be joined at the start, the direction of the curve is turned. If *curve2* is to be joined at the end, the direction of this curve is turned. This means that *curve1* always makes the first part of the new curve.

SYNOPSIS

void s1715(*curve1*, *curve2*, *end1*, *end2*, *newcurve*, *stat*)

SISLCurve	**curve1*;
SISLCurve	**curve2*;
int	*end1*;
int	*end2*;
SISLCurve	***newcurve*;
int	**stat*;

ARGUMENTS

Input Arguments:

curve1	-	First curve to join.
curve2	-	Second curve to join.
end1	-	True (1) if the first curve is to be joined at the end, else false (0).
end2	-	True (1) if the second curve is to be joined at the end, else false (0).

Output Arguments:

newcurve	-	The new joined curve.
stat	-	Status messages > 0 : warning $= 0$: ok < 0 : error

EXAMPLE OF USE

```
{
    SISLCurve   *curve1;   /* Must be defined */
    SISLCurve   *curve2;   /* Must be defined */
    int         end1 = 1;
    int         end2 = 0;
    SISLCurve   *newcurve = NULL;
    int         stat = 0;
    ...
    s1715(curve1, curve2, end1, end2, &newcurve, &stat);
    ...
}
```

6.4.2 Join two curves at closest ends.

NAME

s1716 - To join two curves at the ends that lie closest to each other, if the distance between the ends is less than the tolerance *epsge*. If *curve1* is to be joined at the start, the direction of the curve is turned. If *curve2* is to be joined at the end, the direction of this curve is turned. This means that *curve1* always makes up the first part of the new curve. If *epsge* is positive, but smaller than the smallest distance between the ends of the two curves, a NULL pointer is returned.

SYNOPSIS

```
void s1716(curve1, curve2, epsge, newcurve, stat)
    SISLCurve   *curve1;
    SISLCurve   *curve2;
    double      epsge;
    SISLCurve   **newcurve;
    int         *stat;
```

ARGUMENTS

Input Arguments:

curve1	-	First curve to join.
curve2	-	Second curve to join.
epsge	-	The curves are to be joined if *epsge* is greater than or equal to the distance between the ends lying closest to each other. If *epsge* is negative, the curves are automatically joined.

Output Arguments:

newcurve	-	The new joined curve.
stat	-	Status messages > 0 : warning $= 0$: ok < 0 : error

EXAMPLE OF USE

```
{
    SISLCurve   *curve1;  /* Must be defined */
    SISLCurve   *curve2;  /* Must be defined */
    double      epsge = 1.0e-6;
    SISLCurve   *newcurve = NULL;
    int         stat = 0;
    ...
    s1716(curve1, curve2, epsge, &newcurve, &stat);
    ...
}
```

6.5 Reverse the Orientation of a Curve.

NAME

s1706 - Turn the direction of a curve by reversing the ordering of the coefficients. The start parameter value of the new curve is the same as the start parameter value of the old curve. This routine turns the direction of the orginal curve. If you want a copy with a turned direction, just make a copy and turn the direction of the copy.

SYNOPSIS

void s1706(*curve*)

SISLCurve **curve*;

ARGUMENTS

Input Arguments:

curve - The curve to turn.

EXAMPLE OF USE

```
{
    SISLCurve    *curve;   /* Must be defined */
    ...
    s1706(curve);
    ...
}
```

6.6 Extend a B-spline Curve.

NAME

s1233 - To extend a B-spline curve (i.e. NOT rationals) at the start and/or the end of the curve by continuing the polynomial behaviour of the curve.

SYNOPSIS

void s1233(*pc*, *afak1*, *afak2*, *rc*, *jstat*)

SISLCurve	**pc*;
double	*afak1*;
double	*afak2*;
SISLCurve	***rc*;
int	**jstat*;

ARGUMENTS

Input Arguments:

pc	-	Pointer to the B-spline curve to be extended.
afak1	-	How much the curve is to be stretched at the start of the curve. The length of the stretched curve will be equal to $(1 + afak1)$ times the input curve. $afak1 \geq 0$ and will be set to 0 if negative.
afak2	-	How much the curve is to be stretched at the end of the curve. The length of the stretched curve will be equal to $(1 + afak2)$ times the input curve. $afak2 \geq 0$ and will be set to 0 if negative.

Output Arguments:

rc	-	Pointer to the extended B-spline curve.
jstat	-	Status message < 0 : Error. $= 0$: Ok. $= 1$: Stretching factors less than 0 – read: adjusted factor(s) have been used. > 0 : Warning.

EXAMPLE OF USE

```
{
    SISLCurve   *pc;   /* Must be defined */
    double      afak1 = 0.1;
    double      afak2 = 0.1;
    SISLCurve   *rc = NULL;
    int         jstat = 0;
    ...
    s1233(pc, afak1, afak2, &rc, &jstat);
    ...
}
```

Chapter 7

Surface Definition

7.1 Interpolation

7.1.1 Compute a surface interpolating a set of points, automatic parameterization.

NAME

s1536 - To compute a tensor surface interpolating a set of points, automatic parameterization. The output is represented as a B-spline surface.

SYNOPSIS

void s1536(*points*, *im1*, *im2*, *idim*, *ipar*, *con1*, *con2*, *con3*, *con4*, *order1*, *order2*, *iopen1*, *iopen2*, *rsurf*, *jstat*)

double	*points*[ ];
int	*im1*;
int	*im2*;
int	*idim*;
int	*ipar*;
int	*con1*;
int	*con2*;
int	*con3*;
int	*con4*;
int	*order1*;
int	*order2*;
int	*iopen1*;
int	*iopen2*;
SISLSurf	***rsurf*;
int	**jstat*;

ARGUMENTS

Input Arguments:

points - Array of dimension $idim \times im1 \times im2$ containing the positions of the nodes (using the same ordering as ecoef in the SISLSurf structure).

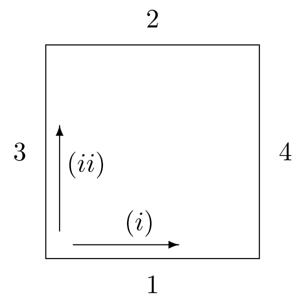
im1 - The number of interpolation points in the first parameter direction.

im2 - The number of interpolation points in the second parameter direction.

idim - Dimension of the space we are working in.

ipar - Flag showing the desired parametrization to be used:
- $= 1$: Mean accumulated cord-length parameterization.
- $= 2$: Uniform parametrization.

Numbering of surface edges:



(i) first parameter direction of surface.
(ii) second parameter direction of surface.

con1 - Additional condition along edge 1:
- $= 0$: No additional condition.
- $= 1$: Zero curvature.

con2 - Additional condition along edge 2:
- $= 0$: No additional condition.
- $= 1$: Zero curvature.

con3 - Additional condition along edge 3:
- $= 0$: No additional condition.
- $= 1$: Zero curvature.

con4 - Additional condition along edge 4:
- $= 0$: No additional condition.
- $= 1$: Zero curvature.

order1 - Order of surface in first parameter direction.

order2 - Order of surface in second

iopen1 - Open/closed/periodic in first parameter direction.
- $= 1$: Open surface.
- $= 0$: Closed surface.
- $= -1$: Closed and periodic surface.

iopen2 - Open/closed/periodic in second parameter direction.
$= 1$: Open surface.
$= 0$: Closed surface.
$= -1$: Closed and periodic surface.

Output Arguments:
rsurf - Pointer to the B-spline surface produced.
jstat - Status message
< 0 : Error.
$= 0$: Ok.
> 0 : Warning.

EXAMPLE OF USE

```
{
    double    points[300];  /* Must be defined */
    int       im1 = 10;
    int       im2 = 10;
    int       idim = 3;
    int       ipar = 1;
    int       con1 = 0;
    int       con2 = 0;
    int       con3 = 0;
    int       con4= 0;
    int       order1 = 4; /* Cubic */
    int       order2 = 4;
    int       iopen1 = 1;
    int       iopen2 = 0;
    SISLSurf  *rsurf = NULL;
    int       jstat = 0;
    ...
    s1536(points, im1, im2, idim, ipar, con1, con2, con3, con4, order1, order2,
          iopen1, iopen2, &rsurf, &jstat);
    ...
}
```

7.1.2 Compute a surface interpolating a set of points, parameterization as input.

NAME

s1537 - Compute a tensor surface interpolating a set of points, parameterization as input. The output is represented as a B-spline surface.

SYNOPSIS

void s1537(*points*, *im1*, *im2*, *idim*, *par1*, *par2*, *con1*, *con2*, *con3*, *con4*, *order1*, *order2*, *iopen1*, *iopen2*, *rsurf*, *jstat*)

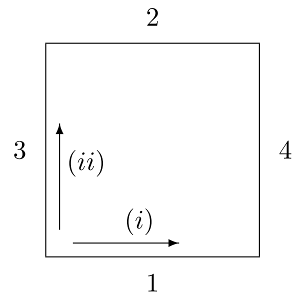
double	*points*[ ];
int	*im1*;
int	*im2*;
int	*idim*;
double	*par1*[ ];
double	*par2*[ ];
int	*con1*;
int	*con2*;
int	*con3*;
int	*con4*;
int	*order1*;
int	*order2*;
int	*iopen1*;
int	*iopen2*;
SISLSurf	***rsurf*;
int	**jstat*;

ARGUMENTS

Input Arguments:

points	-	Array of dimension $idim \times im1 \times im2$ containing the positions of the nodes (using the same ordering as ecoef in the SISLSurf structure).
im1	-	The number of interpolation points in the first parameter direction.
im2	-	The number of interpolation points in the second parameter direction.
idim	-	Dimension of the space we are working in.
par1	-	Parametrization in first parameter direction.
par2	-	Parametrization in second parameter direction.

Numbering of surface edges:



(i) first parameter direction of surface.
(ii) second parameter direction of surface.

con1	-	Additional condition along edge 1:
		$= 0$: No additional condition.
		$= 1$: Zero curvature.
con2	-	Additional condition along edge 2:
		$= 0$: No additional condition.
		$= 1$: Zero curvature.
con3	-	Additional condition along edge 3:
		$= 0$: No additional condition.
		$= 1$: Zero curvature.
con4	-	Additional condition along edge 4:
		$= 0$: No additional condition.
		$= 1$: Zero curvature.
order1	-	Order of surface in first parameter direction.
order2	-	Order of surface in second parameter direction.
iopen1	-	Open/closed/periodic in first parameter direction.
		$= 1$: Open surface.
		$= 0$: Closed surface.
		$= -1$: Closed and periodic surface.
iopen2	-	Open/closed/periodic in second parameter direction.
		$= 1$: Open surface.
		$= 0$: Closed surface.
		$= -1$: Closed and periodic surface.

Output Arguments:

rsurf	-	Pointer to the B-spline surface produced.
jstat	-	Status message
		< 0 : Error.
		$= 0$: Ok.
		> 0 : Warning.

EXAMPLE OF USE

```
{
    double    points[300];  /* Must be defined */
    int       im1 = 10;
    int       im2 = 10;
    int       idim = 3;
    double    par1[10];  /* Must be defined */
    double    par2[10];  /* Must be defined */
    int       con1 = 0;
    int       con2 = 0;
    int       con3 = 0;
    int       con4 = 0;
    int       order1 = 4; /* Cubic */
    int       order2 = 4;
    int       iopen1 = 1;
    int       iopen2 = 0;
    SISLSurf  *rsurf = NULL;
    int       jstat = 0;
    ...
    s1537(points, im1, im2, idim, par1, par2, con1, con2, con3, con4, order1,
          order2, iopen1, iopen2, &rsurf, &jstat);
    ...
}
```

7.1.3 Compute a surface interpolating a set of points, derivatives as input.

NAME

s1534 - To compute a surface interpolating a set of points, derivatives as input. The output is represented as a B-spline surface.

SYNOPSIS

void s1534(*points*, *der10*, *der01*, *der11*, *im1*, *im2*, *idim*, *ipar*, *con1*, *con2*, *con3*, *con4*, *order1*, *order2*, *rsurf*, *jstat*)

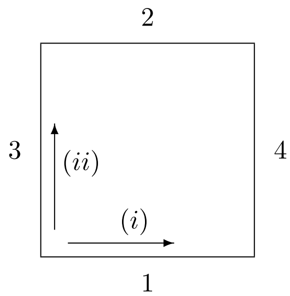
double	*points*[ ];
double	*der10*[ ];
double	*der01*[ ];
double	*der11*[ ];
int	*im1*;
int	*im2*;
int	*idim*;
int	*ipar*;
int	*con1*;
int	*con2*;
int	*con3*;
int	*con4*;
int	*order1*;
int	*order2*;
SISLSurf	***rsurf*;
int	**jstat*;

ARGUMENTS

Input Arguments:

points	-	Array of dimension $idim \times im1 \times im2$ containing the positions of the nodes (using the same ordering as ecoef in the SISLSurf structure).
der10	-	Array of dimension $idim \times im1 \times im2$ containing the first derivatives in the first parameter direction.
der01	-	Array of dimension $idim \times im1 \times im2$ containing the first derivatives in the second parameter direction.
der11	-	Array of dimension $idim \times im1 \times im2$ containing the cross derivatives (the twists).
im1	-	The number of interpolation points in the first parameter direction.
im2	-	The number of interpolation points in the second parameter direction.
idim	-	Dimension of the space we are working in.
ipar	-	Flag showing the desired parametrization to be used: = 1 : Mean accumulated cord-length parameterization. = 2 : Uniform parametrization.

Numbering of surface edges:



(i) first parameter direction of surface.
(ii) second parameter direction of surface.

con1 - Additional condition along edge 1:
$= 0$: No additional condition.
$= 1$: Zero curvature.
con2 - Additional condition along edge 2:
$= 0$: No additional condition.
$= 1$: Zero curvature.
con3 - Additional condition along edge 3:
$= 0$: No additional condition.
$= 1$: Zero curvature.
con4 - Additional condition along edge 4:
$= 0$: No additional condition.
$= 1$: Zero curvature.
order1 - Order of surface in first parameter direction.
order2 - Order of surface in second parameter direction.

Output Arguments:
rsurf - Pointer to the B-spline surface produced.
jstat - Status message
< 0 : Error.
$= 0$: Ok.
> 0 : Warning.

EXAMPLE OF USE

```
{
    double      points[300];  /* Must be defined */
    double      der10[300];   /* Must be defined */
    double      der01[300];   /* Must be defined */
    double      der11[300];   /* Must be defined */
    int         im1 = 10;
    int         im2 = 10;
    int         idim = 3;
    int         ipar = 1;
    int         con1 = 0;
    int         con2 = 0;
    int         con3 = 0;
    int         con4 = 0;
    int         order1 = 4; /* Cubic */
    int         order2 = 4;
    SISLSurf    *rsurf = NULL;
    int         jstat = 0;
    ...
    s1534(points, der10, der01, der11, im1, im2, idim, ipar, con1, con2, con3,
          con4, order1, order2, &rsurf, &jstat);
    ...
}
```

7.1.4 Compute a surface interpolating a set of points, derivatives and parameterization as input.

NAME

s1535 - Compute a surface interpolating a set of points, derivatives and parameterization as input. The output is represented as a B-spline surface.

SYNOPSIS

void s1535(*points*, *der10*, *der01*, *der11*, *im1*, *im2*, *idim*, *par1*, *par2*, *con1*, *con2*, *con3*, *con4*, *order1*, *order2*, *rsurf*, *jstat*)

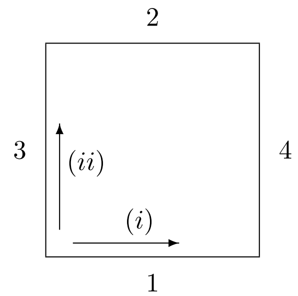
double	*points*[ ];
double	*der10*[ ];
double	*der01*[ ];
double	*der11*[ ];
int	*im1*;
int	*m2*;
int	*idim*;
double	*par1*[ ];
double	*par2*[ ];
int	*con1*;
int	*con2*;
int	*con3*;
int	*con4*;
int	*order1*;
int	*order2*;
SISLSurf	***rsurf*;
int	**jstat*;

ARGUMENTS

Input Arguments:

points	-	Array of dimension $idim \times im1 \times im2$ containing the positions of the nodes (using the same ordering as *ecoef* in the SISLSurf structure).
der10	-	Array of dimension $idim \times im1 \times im2$ containing the first derivatives in the first parameter direction.
der01	-	Array of dimension $idim \times im1 \times im2$ containing the first derivatives in the second parameter direction.
der11	-	Array of dimension $idim \times im1 \times im2$ containing the cross derivatives (the twists).
im1	-	The number of interpolation points in the first parameter direction.
im2	-	The number of interpolation points in the second parameter direction.
idim	-	Dimension of the space we are working in.
par1	-	Parametrization in first parameter direction.
par2	-	Parametrization in second parameter direction.

Numbering of surface edges:



(i) first parameter direction of surface.
(ii) second parameter direction of surface.

con1	-	Additional condition along edge 1: $= 0$: No additional condition. $= 1$: Zero curvature.
con2	-	Additional condition along edge 2: $= 0$: No additional condition. $= 1$: Zero curvature.
con3	-	Additional condition along edge 3: $= 0$: No additional condition. $= 1$: Zero curvature.
con4	-	Additional condition along edge 4: $= 0$: No additional condition. $= 1$: Zero curvature.
order1	-	Order of surface in first parameter direction.
order2	-	Order of surface in second parameter direction.

Output Arguments:

rsurf	-	Pointer to the B-spline surface produced.
jstat	-	Status message < 0 : Error. $= 0$: Ok. > 0 : Warning.

EXAMPLE OF USE

```
{
    double      points[300];  /* Must be defined */
    double      der10[300];   /* Must be defined */
    double      der01[300];   /* Must be defined */
    double      der11[300];   /* Must be defined */
    int         im1 = 10;
    int         im2 = 10;
    int         idim = 3;
    double      par1[10];   /* Must be defined */
    double      par2[10];   /* Must be defined */
    int         con1 = 0;
    int         con2 = 0;
    int         con3 = 0;
    int         con4 = 0;
    int         order1 = 4; /* Cubic */
    int         order2 = 4;
    SISLSurf    *rsurf = NULL;
    int         jstat = 0;
    ...
    s1535(points, der10, der01, der11, im1, im2, idim, par1, par2, con1, con2,
          con3, con4, order1, order2, &rsurf, &jstat);
    ...
}
```

7.1.5 Compute a surface by Hermite interpolation, automatic parameterization.

NAME

s1529 - Compute the cubic Hermite surface interpolant to the data given. More specifically, given positions, (u',v), (u,v'), and (u',v') derivatives at points of a rectangular grid, the routine computes a cubic tensor-product B-spline interpolant to the given data with double knots at each data (the first knot vector will have double knots at all interior points in epar1, quadruple knots at the first and last points, and similarly for the second knot vector). The output is represented as a B-spline surface.

SYNOPSIS

void s1529(*ep*, *eder10*, *eder01*, *eder11*, *im1*, *im2*, *idim*, *ipar*, *rsurf*, *jstat*)

double	*ep*[ ];
double	*eder10*[ ];
double	*eder01*[ ];
double	*eder11*[ ];
int	*im1*;
int	*im2*;
int	*idim*;
int	*ipar*;
SISLSurf	***rsurf*;
int	**jstat*;

ARGUMENTS

Input Arguments:

ep	-	Array of dimension $idim \times im1 \times im2$ containing the positions of the nodes (using the same ordering as ecoef in the SISLSurf structure).
eder10	-	Array of dimension $idim \times im1 \times im2$ containing the first derivative in the first parameter direction.
eder01	-	Array of dimension $idim \times im1 \times im2$ containing the first derivative in the second parameter direction.
eder11	-	Array of dimension $idim \times im1 \times im2$ containing the cross derivative (twist vector).
ipar	-	Flag showing the desired parametrization to be used: $= 1$: Mean accumulated cord-length parameterization. $= 2$: Uniform parametrization.
im1	-	The number of interpolation points in the first parameter direction.
im2	-	The number of interpolation points in the second parameter direction.
idim	-	Spatial dimension.

Output Arguments:

rsurf	-	Pointer to the B-spline surface produced.
jstat	-	Status message

< 0 : Error.
$= 0$: Ok.
> 0 : Warning.

EXAMPLE OF USE

```
{
    double      ep[300];   /* Must be defined */
    double      eder10[300];   /* Must be defined */
    double      eder01[300];   /* Must be defined */
    double      eder11[300];   /* Must be defined */
    int         im1 = 10;
    int         im2 = 10;
    int         idim = 3;
    int         ipar = 1;
    SISLSurf    *rsurf = NULL;
    int         jstat = 0;
    ...
    s1529( ep, eder10, eder01, eder11, im1, im2, idim, ipar, &rsurf, &jstat);
    ...
}
```

7.1.6 Compute a surface by Hermite interpolation, parameterization as input.

NAME

s1530 - To compute the cubic Hermite interpolant to the data given. More specifically, given positions, 10, 01, and 11 derivatives at points of a rectangular grid, the routine computes a cubic tensor-product B-spline interpolant to the given data with double knots at each data point (the first knot vector will have double knots at all interior points in epar1, quadruple knots at the first and last points, and similarly for the second knot vector). The output is represented as a B-spline surface.

SYNOPSIS

void s1530(*ep*, *eder10*, *eder01*, *eder11*, *epar1*, *epar2*, *im1*, *im2*, *idim*, *rsurf*, *jstat*)

double	*ep*[ ];
double	*eder10*[ ];
double	*eder01*[ ];
double	*eder11*[ ];
double	*epar1*[ ];
double	*epar2*[ ];
int	*im1*;
int	*im2*;
int	*idim*;
SISLSurf	***rsurf*;
int	**jstat*;

ARGUMENTS

Input Arguments:

ep	-	Array of dimension $idim \times im1 \times im2$ containing the positions of the nodes (using the same ordering as *ecoef* in the SISLSurf structure).
eder10	-	Array of dimension $idim \times im1 \times im2$ containing the first derivative in the first parameter direction.
eder01	-	Array of dimension $idim \times im1 \times im2$ containing the first derivative in the second parameter direction.
eder11	-	Array of dimension $idim \times im1 \times im2$ containing the cross derivative (twist vector).
epar1	-	Array of size *im1* containing the parametrization in the first direction.
epar2	-	Array of size *im2* containing the parametrization in the first direction.
im1	-	The number of interpolation points in the 1st param. dir.
im2	-	The number of interpolation points in the 2nd param. dir.
idim	-	Dimension of the space we are working in.

Output Arguments:

rsurf	-	Pointer to the B-spline surface produced.
jstat	-	Status message < 0 : Error.

$= 0$: Ok.
> 0 : Warning.

EXAMPLE OF USE

```
{
    double      ep[30];  /* Must be defined */
    double      eder10[30];  /* Must be defined */
    double      eder01[30];  /* Must be defined */
    double      eder11[30];  /* Must be defined */
    double      epar1[2];  /* Must be defined */
    double      epar2[5];  /* Must be defined */
    int         im1 = 2;
    int         im2 = 5;
    int         idim = 3;
    SISLSurf    *rsurf = NULL;
    int         jstat = 0;
    ...
    s1530(ep, eder10, eder01, eder11, epar1, epar2, im1, im2, idim, &rsurf, &jstat);
    ...
}
```

7.1.7 Create a lofted surface from a set of B-spline input curves.

NAME

s1538 - To create a lofted surface from a set of B-spline (i.e. NOT rational) input curves. The output is represented as a B-spline surface.

SYNOPSIS

void s1538(*inbcrv*, *vpcurv*, *nctyp*, *astpar*, *iopen*, *iord2*, *iflag*, *rsurf*, *gpar*, *jstat*)

int	*inbcrv*;
SISLCurve	**vpcurv*[ ];
int	*nctyp*[ ];
double	*astpar*;
int	*iopen*;
int	*iord2*;
int	*iflag*;
SISLSurf	***rsurf*;
double	***gpar*;
int	**jstat*;

ARGUMENTS

Input Arguments:

inbcrv	-	Number of B-spline curves in the curve set.
vpcurv	-	Array (length *inbcrv*) of pointers to the curves in the curve-set.
nctyp	-	Array (length *inbcrv*) containing the types of curves in the curve-set. = 1 : Ordinary curve. = 2 : Knuckle curve. Treated as an ordinary curve. = 3 : Tangent to next curve. = 4 : Tangent to prior curve. (= 5 : Second derivative to prior curve.) (= 6 : Second derivative to next curve.) = 13 : Curve giving start of tangent to next curve. = 14 : Curve giving end of tangent to prior curve.
astpar	-	Start parameter for spline lofting direction.
iopen	-	Flag telling if the resulting surface should be open, closed or periodic in the lofting direction (i.e. not the curve direction). = 1 : Open. = 0 : Closed. = −1 : Closed and periodic.
iord2	-	Maximal order of the surface in the lofting direction.

iflag	-	Flag telling if the size of the tangents in the derivative curves should be adjusted or not. $= 0$: Do not adjust tangent sizes. $= 1$: Adjust tangent sizes.

Output Arguments:

rsurf	-	Pointer to the B-spline surface produced.
gpar	-	The input curves are constant parameter lines in the parameter-plane of the produced surface. The i-th element in this array contains the (constant) value of this parameter of the i-th. input curve.
jstat	-	Status message < 0 : Error. $= 0$: Ok. > 0 : Warning.

EXAMPLE OF USE

```
{
    int         inbcrv = 3;
    SISLCurve   *vpcurv[3]; /* Must be defined */
    int         nctyp[3];  /* Must be defined */
    double      astpar = 0.0;
    int         iopen = 1;
    int         iord2 = 4; /* Cubic */
    int         iflag = 1;
    SISLSurf    *rsurf = NULL;
    double      *gpar = NULL;
    int         jstat = 0;
    ...
    s1538(inbcrv, vpcurv, nctyp, astpar, iopen, iord2, iflag, &rsurf, &gpar, &js-
          tat);
    ...
}
```

7.1.8 Create a lofted surface from a set of B-spline input curves and parametrization.

NAME

s1539 - To create a spline lofted surface from a set of input curves. The parametrization of the position curves is given in epar.

SYNOPSIS

void s1539(*inbcrv*, *vpcurv*, *nctyp*, *epar*, *astpar*, *iopen*, *iord2*, *iflag*, *rsurf*, *gpar*, *jstat*)

int	*inbcrv*;
SISLCurve	**vpcurv*[ ];
int	*nctyp*[ ];
double	*epar*[ ];
double	*astpar*;
int	*iopen*;
int	*iord2*;
int	*iflag*;
SISLSurf	***rsurf*;
double	***gpar*;
int	**jstat*;

ARGUMENTS

Input Arguments:

inbcrv - set.

vpcurv - Array (length inbcrv) of pointers to the curves in the curve-set.

nctyp - Array (length inbcrv) containing the types of curves in the curve-set.
- = 1 : Ordinary curve.
- = 2 : Knuckle curve. Treated as an ordinary curve.
- = 3 : Tangent to next curve.
- = 4 : Tangent to previous curve.
- (= 5 : Second derivative to previous curve.)
- (= 6 : Second derivative to next curve.)
- = 13 : Curve giving start of tangent to next curve.
- = 14 : Curve giving end of tangent to previous curve.

epar - Array containing the wanted parametrization. Only parametervalues corresponding to position curves are given. For closed curves, one additional parameter value must be spesified. The last entry contains the parametrization of the repeated start curve. (if the endpoint is equal to the startpoint of the interpolation the lenght of the array should be equal to inpt1 also in the closed case). The number of entries in the array is thus equal to the number of position curves (number plus one if the curve is closed).

astpar - parameter for spline lofting direction.

iopen - Flag saying whether the resulting surface should be closed or open.
- = 1 : Open.

		$= 0$: Closed.
		$= -1$: Closed and periodic.
iord2	-	spline basis in the lofting direction.
iflag	-	Flag saying whether the size of the tangents in the derivative curves should be adjusted or not. $= 0$: Do not adjust tangent sizes. $= 1$: Adjust tangent sizes.

Output Arguments:

rsurf	-	Pointer to the surface produced.
gpar	-	The input curves are constant parameter lines in the parameter-plane of the produced surface. The i-th element in this array contains the (constant) value of this parameter of the i-th. input curve.
jstat	-	Status message < 0 : Error. $= 0$: Ok. > 0 : Warning.

EXAMPLE OF USE

```
{
    int         inbcrv = 4;
    SISLCurve   *vpcurv[4];  /* Must be defined */
    int         nctyp[4];  /* Must be defined */
    double      epar[5];  /* Must be defined. The length corresponds to only
                           positional curves and no duplication of first curve */
    double      astpar = 0.0;
    int         iopen = 0;
    int         iord2 = 4; /* Cubic */
    int         iflag = 0;
    SISLSurf    *rsurf = NULL;
    double      *gpar = NULL;
    int         jstat = 0;
    ...
    s1539(inbcrv, vpcurv, nctyp, epar, astpar, iopen, iord2, iflag, &rsurf, &gpar,
        &jstat);
    ...
}
```

7.1.9 Create a rational lofted surface from a set of rational input-curves

NAME

s1508 - To create a rational lofted surface from a set of rational input-curves. The surface will be C^1 cubic in the lofting direction.

SYNOPSIS

void s1508(*inbcrv*, *vpcurv*, *par_arr*, *rsurf*, *jstat*)

int	*inbcrv*;
SISLCurve	**vpcurv[ ]*;
double	*par_arr[ ]*;
SISLSurf	***rsurf*;
int	**jstat*;

ARGUMENTS

Input Arguments:

inbcrv	-	Number of NURBS-curves in the curve set.
vpcurv	-	Array (length *inbcrv*) of pointers to the curves in the curve-set.
par_arr	-	The required parametrization, must be strictly increasing, length *inbcrv*.

Output Arguments:

rsurf	-	Pointer to the NURBS surface produced.
jstat	-	status message < 0 : Error. $= 0$: Ok. > 0 : Warning.

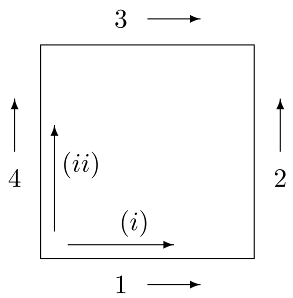
EXAMPLE OF USE

```
{
    int         inbcrv = 3;
    SISLCurve   *vpcurv[3];  /* Must be defined */
    double      par_arr[3];  /* Must be defined */
    SISLSurf    *rsurf = NULL;
    int         jstat = 0;
    ...
    s1508(inbcrv, vpcurv, par_arr, &rsurf, &jstat);
    ...
}
```

7.1.10 Compute a rectangular blending surface from a set of B-spline input curves.

NAME

s1390 - Make a 4-edged blending surface between 4 B-spline (i.e. NOT rational) curves where each curve is associated with a number of cross-derivative B-spline (i.e. NOT rational) curves. The output is represented as a B-spline surface. The input curves are numbered successively around the blending parameter, and the directions of the curves are expected to be as follows when this routine is entered:



(i) first parameter direction of the surface.
(ii) second parameter direction of the surface.

NB! The cross-derivatives are always pointing into the patch, and note the directions in the above diagram.

SYNOPSIS

void s1390(*curves*, *surf*, *numder*, *stat*)

SISLCurve	**curves*[ ];
SISLSurf	***surf*;
int	*numder*[ ];
int	**stat*;

ARGUMENTS

Input Arguments:

curves - Pointers to the boundary B-spline curves:
$curves[i], i = 0, \ldots, numder[0]-1$, are pointers to position and cross-derivatives along the first edge.
$curves[i]$,
$i = numder[0], \ldots, numder[0]+numder[1]-1$, are pointers to position and cross-derivatives along the second edge.
$curves[i], i = numder[0] + numder[1], \ldots,$
$numder[0] + numder[1] + numder[2] - 1$, are pointers to position and cross-derivatives along the third edge.

$curves[i]$,
$i = numder[0] + numder[1] + numder[2], \ldots,$
$numder[0] + numder[1] + numder[2] + numder[3] - 1$, are pointers to position and cross-derivatives along the fourth edge.

numder - Array of length 4, numder[i] gives the number of curves on edge number $i + 1$.

Output Arguments:

surf - Pointer to the blending B-spline surface.

stat - Status messages
> 0 : warning
$= 0$: ok
< 0 : error

EXAMPLE OF USE

```
{
    SISLCurve   *curves[8];   /* Must be defined */
    SISLSurf    *surf = NULL;
    int         numder[4]; /* Each entry is equal to 2 in this case */
    int         stat = 0;
    ...
    s1390(curves, &surf, numder, &stat)
    ...
}
```

7.1.11 Compute a first derivative continuous blending surface set, over a 3-, 4-, 5- or 6-sided region in space, from a set of B-spline input curves.

NAME

s1391 - To create a first derivative continuous blending surface set over a 3-, 4-, 5- and 6-sided region in space. The boundary of the region are B-spline (i.e. NOT rational) curves and the cross boundary derivatives are given as B-spline (i.e. NOT rational) curves. This function automatically preprocesses the input cross tangent curves in order to make them suitable for the blending. Thus, the cross tangent curves should be taken as the cross tangents of the surrounding surface. It is not necessary and not advisable to match tangents etc. in the corners. The output is represented as a set of B-spline surfaces.

SYNOPSIS

```
void s1391(pc, ws, icurv, nder, jstat)
    SISLCurve   **pc;
    SISLSurf    ***ws;
    int         icurv;
    int         nder[ ];
    int         *jstat;
```

ARGUMENTS

Input Arguments:

pc - Pointers to boundary B-spline curves. All curves must have same parameter direction around the patch, either clockwise or counterclockwise. $pc1[i], i = 0, \ldots nder[0] - 1$ are pointers to position and cross-derivatives along first edge. $pc1[i], i = nder[0], \ldots nder[1] - 1$ are pointers to position and cross-derivatives along second edge.

$$\vdots$$

$$pc1[i], i = nder[0]+\ldots+nder[icurv-2], \ldots, nder[icurv-1]-1$$

are pointers to position and cross-derivatives along fourth edge.

icurv - Number of boundary curves (3, 5, 4 or 6).

nder - *nder[i]* gives number of curves on edge number $i+1$. These numbers has to be equal to 2. The vector is of length *icurv*.

Output Arguments:

ws	-	These are pointers to the blending B-spline surfaces. The vector is of length *icurv*.
jstat	-	Status message < 0 : Error. $= 0$: Ok. > 0 : Warning.

EXAMPLE OF USE

```
{
    SISLCurve   *pc[10]; /* Position and derivative curves. Must be defined */
    SISLSurf    **ws = NULL; /* In this case 5 surfaces will be constructed
    int         icurv = 5;
    int         nder[5]; /* Each entry must be equal to 2 */
    int         jstat = 0;
    ...
    s1391(pc, &ws, icurv, nder, &jstat);
    ...
}
```

7.1.12 Compute a surface, representing a Gordon patch, from a set of B-spline input curves.

NAME

s1401 - Compute a Gordon patch, given position and cross tangent conditions as B-spline (i.e. NOT rational) curves at the boundary of a squared region and the twist vector in the corners. The output is represented as a B-spline surface.

SYNOPSIS

void s1401(*vcurve*, *etwist*, *rsurf*, *jstat*)

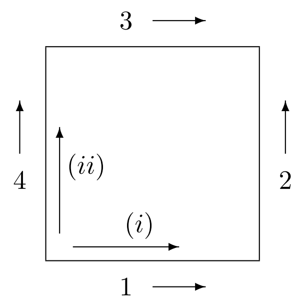
double	*etwist*[ ];
SISLCurve	**vcurve*[ ];
int	**jstat*;
SISLSurf	***rsurf*;

ARGUMENTS

Input Arguments:

vcurve - Position and cross-tangent B-spline curves around the square region. For each edge of the region position and cross-tangent curves are given. The dimension of the array is 8.

The orientation is as follows:



(i) first parameter direction of the surface.
(ii) second parameter direction of the surface.

etwist - Twist-vectors of the corners of the vertex region. The first element of the array is the twist in the corner before the first edge, etc. The dimension of the array is 4 times the spatial dimension of the input curves (currently only 3D).

Output Arguments:

rsurf	-	Gordons-patch represented as a B-spline surface.
jstat	-	Status message < 0 : Error. $= 0$: Ok. > 0 : Warning.

EXAMPLE OF USE

```
{
    int         idim = 3;
    double      etwist[12]; /* 4*idim. Must be defined */
    SISLCurve   *vcurve[8]; /* Position and derivative curves. Must be defined */
    int         jstat = 0;
    SISLSurf    *rsurf = NULL;
    ...
    s1401(vcurve, etwist, &rsurf, &jstat);
    ...
}
```

7.2 Approximation

Two kinds of surfaces are treated in this section. The first is approximation of special shape properties like rotation or sweeping. The second is offsets to surfaces.

All functions require a tolerance for use in the approximation. It is useful to note that there is a close relation between the size of the tolerance and the amount of data for the surface.

7.2.1 Compute a surface using the input points as control vertices, automatic parameterization.

NAME

s1620 - To calculate a surface using the input points as control vertices. The parametrization is calculated according to *ipar*. The output is represented as a B-spline surface.

SYNOPSIS

void s1620(*epoint*, *inbpnt1*, *inbpnt2*, *ipar*, *iopen1*, *iopen2*, *ik1*, *ik2*, *idim*, *rs*, *jstat*)

double	*epoint*[ ];
int	*inbpnt1*;
int	*inbpnt2*;
int	*ipar*;
int	*iopen1*;
int	*iopen2*;
int	*ik1*;
int	*ik2*;
int	*idim*;
SISLSurf	***rs*;
int	**jstat*;

ARGUMENTS

Input Arguments:

epoint	-	The array containing the points to be used as controlling vertices of the B-spline surface.
inbpnt1	-	The number of points in first parameter direction.
inbpnt2	-	The number of points in second parameter direction.
ipar	-	Flag showing the desired parametrization to be used: = 1 : Mean accumulated cord-length parameterization. = 2 : Uniform parametrization.
iopen1	-	Open/close condition in the first parameter direction: = 1 : Open. = 0 : Closed. $= -1$: Closed and periodic.

iopen2	-	Open/close condition in the second parameter direction: $= 1$: Open. $= 0$: Closed. $= -1$: Closed and periodic.
ik1	-	The order of the surface in first direction.
ik2	-	The order of the surface in second direction.
idim	-	The dimension of the space.

Output Arguments:

rs	-	Pointer to the B-spline surface.
jstat	-	Status message < 0 : Error. $= 0$: Ok. > 0 : Warning.

EXAMPLE OF USE

```
{
    double     epoint[300];  /* Must be defined */
    int        inbpnt1 = 10;
    int        inbpnt2 = 10;
    int        ipar = 1;
    int        iopen1 = 1;
    int        iopen2 = 1;
    int        ik1 = 4; /* Cubic */
    int        ik2 = 4;
    int        idim = 3;
    SISLSurf   *rs = NULL;
    int        jstat = 0;
    ...
    s1620(epoint, inbpnt1, inbpnt2, ipar, iopen1, iopen2, ik1, ik2, idim, &rs,
          &jstat);
    ...
}
```

7.2.2 Compute a linear swept surface.

NAME

s1332 - To create a linear swept surface by making the tensor-product of two curves.

SYNOPSIS

void s1332(*curve1*, *curve2*, *epsge*, *point*, *surf*, *stat*)

SISLCurve	**curve1*;
SISLCurve	**curve2*;
double	*epsge*;
double	*point*[ ];
SISLSurf	***surf*;
int	**stat*;

ARGUMENTS

Input Arguments:

curve1	-	Pointer to curve 1.
curve2	-	Pointer to curve 2.
epsge	-	Maximal deviation allowed between the true swept surface and the generated surface.
point	-	Point near the curve to sweep along. The vertices of the new surface are made by adding the vector from point to each of the vertices on the sweep curve, to each of the vertices on the other curve.

Output Arguments:

surf	-	Pointer to the surface produced.
stat	-	Status messages > 0 : warning $= 0$: ok < 0 : error

EXAMPLE OF USE

```
{
    curve       *curve1;   /* Must be defined */
    curve       *curve2;   /* Must be defined */
    double      epsge = 0.001 ;
    double      point[3];  /* Dimension as for curve coefficients. Must be defined */
    SISLSurf    *surf = NULL;
    int         stat = 0;
    . . .
    s1332(curve1, curve2, epsge, point, &surf, &stat);
    . . .
}
```

7.2.3 Compute a rotational swept surface.

NAME

s1302 - To create a rotational swept surface by rotating a curve a given angle around the axis defined by *point*[ ] and *axis*[ ]. The maximal deviation allowed between the true rotational surface and the generated surface, is *epsge*. If *epsge* is set to 0, a NURBS surface is generated and if *epsge* > 0, a B-spline surface is generated.

SYNOPSIS

void s1302(*curve*, *epsge*, *angle*, *point*, *axis*, *surf*, *stat*)

SISLCurve	**curve*;
double	*epsge*;
double	*angle*;
double	*point*[ ];
double	*axis*[ ];
SISLSurf	***surf*;
int	**stat*;

ARGUMENTS

Input Arguments:

curve	-	Pointer to the curve that is to be rotated.
epsge	-	Maximal deviation allowed between the true rotational surface and the generated surface.
angle	-	The rotational angle. The angle is counterclockwise around axis. If the absolute value of the angle is greater than 2π then a rotational surface that is closed in the rotation direction is made.
point	-	Point on the rotational axis.
axis	-	Direction of rotational axis.

Output Arguments:

surf	-	Pointer to the produced surface. This will be a NURBS (i.e. rational) surface if *epsge* $= 0$ and a B-spline (i.e. non-rational) surface if *epsge* > 0.
stat	-	Status messages > 0 : warning $= 0$: ok < 0 : error

EXAMPLE OF USE

```
{
    SISLCurve   *curve;  /* Must be defined */
    double      epsge = 0.0;
    double      angle;  /* Must be defined */
    double      point[3];  /* Must be defined */
    double      axis[3];  /* Must be defined */
    SISLSurf    *surf = NULL;
    int         stat = 0;
    ...
    s1302(curve, epsge, angle, point, axis, &surf, &stat);
    ...
}
```

7.2.4 Compute a surface approximating the offset of a surface.

NAME

s1365 - Create a surface approximating the offset of a surface. The output is represented as a B-spline surface.
With an offset of zero, this routine can be used to approximate any NURBS (rational) surface with a B-spline (non-rational) surface.

SYNOPSIS

void s1365(*ps*, *aoffset*, *aepsge*, *amax*, *idim*, *rs*, *jstat*)

SISLSurf	**ps*;
double	*aoffset*;
double	*aepsge*;
double	*amax*;
int	*idim*;
SISLSurf	***rs*;
int	**jstat*;

ARGUMENTS

Input Arguments:

ps	-	The input surface.
aoffset	-	The offset distance. The offset direction is determined by the normalized cross product of the tangent vector and the anorm vector. The offset distance is multiplied by this vector.
aepsge	-	Maximal deviation allowed between true offset surface and the approximated offset surface.
amax	-	Maximal stepping length. Is negleceted if $amax \leq aepsge$. If $amax = 0$ then a maximal step length of the longest box side is used.
idim	-	The dimension of the space (idim = 3 is required).

Output Arguments:

rs	-	The approximated offset represented as a B-spline surface.
jstat	-	Status message < 0 : Error. $= 0$: Ok. > 0 : Warning.

EXAMPLE OF USE

```
{
    SISLSurf    *ps;   /* Must be defined */
    double      aoffset;   /* Must be defined */
    double      aepsge = 0.001;
    double      amax = 0;
    int         idim = 3;
    SISLSurf    *rs = NULL;
    int         jstat = 0;
    ...
    s1365(ps, aoffset, aepsge, amax, idim, &rs, &jstat);
    ...
}
```

7.3 Mirror a Surface

NAME

s1601 - Mirror a surface about a plane.

SYNOPSIS

void s1601(*psurf*, *epoint*, *enorm*, *idim*, *rsurf*, *jstat*)

```
SISLSurf   *psurf;
double     epoint[ ];
double     enorm[ ];
int        idim;
SISLSurf   **rsurf;
int        *jstat;
```

ARGUMENTS

Input Arguments:

psurf	-	The input surface.
epoint	-	A point in the plane.
enorm	-	The normal vector to the plane.
idim	-	The dimension of the space, must be the same as the surface.

Output Arguments:

rsurf	-	Pointer to the mirrored surface.
jstat	-	Status message < 0 : Error. $= 0$: Ok. > 0 : Warning.

EXAMPLE OF USE

```
{
    SISLSurf   *psurf;  /* Must be defined */
    double     epoint[3];  /* Must be defined */
    double     enorm[3];  /* Must be defined */
    int        idim = 3;
    SISLSurf   *rsurf = NULL;
    int        jstat = 0;
    ...
    s1601(psurf, epoint, enorm, idim, &rsurf, &jstat);
    ...
}
```

7.4 Conversion

7.4.1 Convert a surface of order up to four to a mesh of Coons patches.

NAME

s1388 - To convert a surface of order less than or equal to 4 in both directions to a mesh of Coons patches with uniform parameterization. The function assumes that the surface is C^1 continuous.

SYNOPSIS

void s1388(*surf*, *coons*, *numcoons1*, *numcoons2*, *dim*, *stat*)

SISLSurf	**surf*;
double	***coons*;
int	**numcoons1*;
int	**numcoons2*;
int	**dim*
int	**stat*;

ARGUMENTS

Input Arguments:

surf	-	Pointer to the surface that is to be converted

Output Arguments:

coons	-	Array containing the (sequence of) Coons patches. The total number of patches is $numcoons1 \times numcoons2$. The patches are stored in sequence with $dim \times 16$ values for each patch. For each corner of the patch we store in sequence, positions, derivative in first direction, derivative in second direction, and twists.
numcoons1	-	Number of Coons patches in first parameter direction.
numcoons2	-	Number of Coons patches in second parameter direction.
dim	-	The dimension of the geometric space.
stat	-	Status messages $= 1$: Order too high, surface interpolated. $= 0$: Ok. < 0 : Error.

EXAMPLE OF USE

```
{
    SISLSurf    *surf;  /* Must be defined */
    double      *coons = NULL;
    int         numcoons1 = 0;
    int         numcoons2 = 0;
    int         dim;
    int         stat = 0;
    ...
    s1388(surf, &coons, &numcoons1, &numcoons2, &dim, &stat);
    ...
}
```

7.4.2 Convert a surface to a mesh of Bezier surfaces.

NAME

s1731 - To convert a surface to a mesh of Bezier surfaces. The Bezier surfaces are stored in a surface with all knots having multiplicity equal to the order of the surface in the corresponding parameter direction. If the input surface is rational, the generated Bezier surfaces will be rational too (i.e. there will be rational weights in the representation of the Bezier surfaces).

SYNOPSIS

```
void s1731(surf, newsurf, stat)
    SISLSurf    *surf;
    SISLSurf    **newsurf;
    int         *stat;
```

ARGUMENTS

Input Arguments:

surf	-	Surface to convert.

Output Arguments:

newsurf	-	The new surface storing the Bezier represented surfaces.
stat	-	Status messages > 0 : warning $= 0$: ok < 0 : error

EXAMPLE OF USE

```
{
    SISLSurf    *surf;  /* Must be defined */
    SISLSurf    *newsurf = NULL;
    int         stat = 0;
    ...
    s1731(surf, &newsurf, &stat);
    ...
}
```

7.4.3 Pick the next Bezier surface from a surface.

NAME

s1733 - To pick the next Bezier surface from a surface. This function requires a surface represented as the result of s1731(). See page 172. This routine does not check that the surface is correct. If the input surface is rational, the generated Bezier surfaces will be rational too (i.e. there will be rational weights in the representation of the Bezier surfaces).

SYNOPSIS

void s1733(*surf*, *number1*, *number2*, *startpar1*, *endpar1*, *startpar2*, *endpar2*, *coef*, *stat*)

SISLSurf	**surf*;
int	*number1*;
int	*number2*;
double	**startpar1*;
double	**endpar1*;
double	**startpar2*;
double	**endpar2*;
double	*coef*[ ];
int	**stat*;

ARGUMENTS

Input Arguments:

surf	-	The surface to convert.
number1	-	The number of the Bezier patch to pick in the horizontal direction, where $0 \leq number1 < in1/ik1$ of the surface.
number2	-	The number of the Bezier patch to pick in the vertical direction, , where $0 \leq number2 < in2/ik2$ of the surface.

Output Arguments:

startpar1	-	The start parameter value of the Bezier patch in the horizontal direction.
endpar1	-	The end parameter value of the Bezier patch in the horizontal direction.
startpar2	-	The start parameter value of the Bezier patch in the vertical direction.
endpar2	-	The end parameter value of the Bezier patch in the vertical direction.
coef	-	The vertices of the Bezier patch. Space must be allocated with a size of $idim \times ik1 \times ik2$ or $(idim + 1) \times ik1 \times ik2$ in the rational case. These parameters are given by the surface (this is done for reasons of efficiency).

stat - Status messages
> 0 : warning
$= 0$: ok
< 0 : error

EXAMPLE OF USE

```
{
    SISLSurf    *surf;   /* Must be defined */
    int         number1;  /* Must be defined */
    int         number2;  /* Must be defined */
    double      startpar1;
    double      endpar1;
    double      startpar2;
    double      endpar2;
    double      coef[48]; /* Non-rational, degree 3 in both directions,
                             geometry space dimension equal to 3 */
    int         stat = 0;
    ...
    s1733(surf, number1, number2, &startpar1, &endpar1, &startpar2, &end-
        par2, coef, &stat);
    ...
}
```

7.4.4 Express a surface using a higher order basis.

NAME

s1387 - To express a surface as a surface of higher order.

SYNOPSIS

void s1387(*surf*, *order1*, *order2*, *newsurf*, *stat*)

```
SISLSurf   *surf;
int        order1;
int        order2;
SISLSurf   **newsurf;
int        *stat;
```

ARGUMENTS

Input Arguments:

surf	-	Surface to raise the order of.
order1	-	New order in the first parameter direction.
order2	-	New order in the second parameter direction.

Output Arguments:

newsurf	-	The resulting order elevated surface.
stat	-	Status messages = 1 : Input order equal to order of surface. Pointer set to input. = 0 : Ok. < 0 : Error.

EXAMPLE OF USE

```
{
    SISLSurf   *surf;  /* Must be defined */
    int        order1;  /* Must be defined. Larger than or equal to surf–>ik1 */
    int        order2;  /* Must be defined. Larger than or equal to surf–>ik2 */
    SISLSurf   *newsurf = NULL;
    int        stat = 0;
    ...
    s1387(surf, order1, order2, &newsurf, &stat);
    ...
}
```

7.4.5 Express the “i,j”-th derivative of an open surface as a surface.

NAME

s1386 - To express the $(der1, der2)$-th derivative of an open surface as a surface.

SYNOPSIS

void s1386(*surf*, *der1*, *der2*, *newsurf*, *stat*)

SISLSurf	**surf*;
int	*der1*;
int	*der2*;
SISLSurf	***newsurf*;
int	**stat*;

ARGUMENTS

Input Arguments:

surf	-	Surface to differentiate.
der1	-	The derivative to be produced in the first parameter direction: $0 \leq der1$
der2	-	The derivative to be produced in the second parameter direction: $0 \leq der2$

Output Arguments:

newsurf	-	The result of the (der1, der2) differentiation of surf.
stat	-	Status messages > 0 : warning $= 0$: ok < 0 : error

EXAMPLE OF USE

```
{
    SISLSurf    *surf;  /* Must be defined */
    int         der1 = 1;
    int         der2 = 0;
    SISLSurf    *newsurf = NULL;
    int         stat = 0;
    ...
    s1386(surf, der1, der2, &newsurf, &stat);
    ...
}
```

7.4.6 Express the octants of a sphere as a surface.

NAME

s1023 - To express the octants of a sphere as a surface. This can also be used to describe the complete sphere. The sphere/the octants of the sphere will be geometrically exact.

SYNOPSIS

void s1023(*centre*, *axis*, *equator*, *latitude*, *longitude*, *sphere*, *stat*)

double	*centre*[ ];
double	*axis*[ ];
double	*equator*[ ];
int	*latitude*;
int	*longitude*;
SISLSurf	***sphere*;
int	**stat*;

ARGUMENTS

Input Arguments:

centre	-	Centre point of the sphere.
axis	-	Axis of the sphere (towards the north pole).
equator	-	Vector from centre to start point on the equator.
latitude	-	Flag indicating number of octants in north/south direction: $= 1$: Octants in the northern hemisphere. $= 2$: Octants in both hemispheres.
longitude	-	Flag indicating number of octants along the equator. This is counted counterclockwise from equator. $= 1$: Octants in 1. quadrant. $= 2$: Octants in 1. and 2. quadrant. $= 3$: Octants in 1., 2. and 3. quadrant. $= 4$: Octants in all quadrants.

Output Arguments:

sphere	-	The sphere produced.
stat	-	Status messages > 0 : warning $= 0$: ok < 0 : error

EXAMPLE OF USE

```
{
    double      centre[3];   /* Must be defined */
    double      axis[3];   /* Must be defined */
    double      equator[3];   /* Must be defined */
    int         latitude = 1;
    int         longitude = 2;
    SISLSurf    *sphere = NULL;
    int         stat = 0;
    ...
    s1023(centre, axis, equator, latitude, longitude, &sphere, &stat);
    ...
}
```

7.4.7 Express a truncated cylinder as a surface.

NAME

s1021 - To express a truncated cylinder as a surface. The cylinder can be elliptic. The cylinder will be geometrically exact.

SYNOPSIS

void s1021(*bottom_pos*, *bottom_axis*, *ellipse_ratio*, *axis_dir*, *height*, *cyl*, *stat*)

double	*bottom_pos*[ ];
double	*bottom_axis*[ ];
double	*ellipse_ratio*;
double	*axis_dir*[ ];
double	*height*;
SISLSurf	***cyl*;
int	**stat*;

ARGUMENTS

Input Arguments:

bottom_pos	-	Center point of the bottom.
bottom_axis	-	One of the bottom axis (major or minor).
ellipse_ratio	-	Ratio between the other axis and bottom_axis.
axis_dir	-	Direction of the cylinder axis.
height	-	Height of the cone, can be negative.

Output Arguments:

cyl	-	Pointer to the cylinder produced.
stat	-	Status messages > 0 : Warning. $= 0$: Ok. < 0 : Error.

EXAMPLE OF USE

```
{
    double      bottom_pos[3];  /* Must be defined */
    double      bottom_axis[3];  /* Must be defined */
    double      ellipse_ratio = 1.0; /* Circular cylinder */
    double      axis_dir[3];  /* Must be defined */
    double      height;  /* Must be defined */
    SISLSurf    *cyl = NULL;
    int         stat = 0;
    . . .
    s1021(bottom_pos, bottom_axis, ellipse_ratio, axis_dir, height, &cyl, &stat)
    . . .
}
```

7.4.8 Express the octants of a torus as a surface.

NAME

s1024 - To express the octants of a torus as a surface. This can also be used to describe the complete torus. The torus/the octants of the torus will be geometrically exact.

SYNOPSIS

void s1024(*centre*, *axis*, *equator*, *minor_radius*, *start_minor*, *end_minor*, *numb_major*, *torus*, *stat*)

double	*centre*[ ];
double	*axis*[ ];
double	*equator*[ ];
double	*minor_radius*;
int	*start_minor*;
int	*end_minor*;
int	*numb_major*;
SISLSurf	***torus*;
int	**stat*;

ARGUMENTS

Input Arguments:

centre	-	Centre point of the torus.
axis	-	Normal to the torus plane.
equator	-	Vector from centre to start point on the major circle.
minor_radius	-	Radius of the minor circle.
start_minor	-	Start quadrant on the minor circle (1,2,3 or 4). This is counted clockwise from the extremum in the direction of axis.
end_minor	-	End quadrant on the minor circle (1,2,3 or 4). This is counted clockwise from the extremum in the direction of axis.
numb_major	-	Number of quadrants on the major circle (1,2,3 or 4). This is counted counterclockwise from equator.

Output Arguments:

torus	-	Pointer to the torus produced.
stat	-	Status messages > 0 : Warning. $= 0$: Ok. < 0 : Error.

EXAMPLE OF USE

```
{
    double      centre[3]; /* Must be defined */
    double      axis[3]; /* Must be defined */
    double      equator[3]; /* Must be defined. Length gives major radius */
    double      minor_radius; /* Must be defined */
    int         start_minor = 1;
    int         end_minor = 4; /* start_minor and end_minor defines full circle */
    int         numb_major = 2;
    SISLSurf    *torus = NULL;
    int         stat = 0;
    ...
    s1024(centre, axis, equator, minor_radius, start_minor, end_minor,
          numb_major, &torus, &stat)
    ...
}
```

7.4.9 Express a truncated cone as a surface.

NAME

s1022 - To express a truncated cone as a surface. The cone can be elliptic. The cone will be geometrically exact.

SYNOPSIS

void s1022(*bottom_pos*, *bottom_axis*, *ellipse_ratio*, *axis_dir*, *cone_angle*, *height*, *cone*, *stat*)

double	*bottom_pos*[ ];
double	*bottom_axis*[ ];
double	*ellipse_ratio*;
double	*axis_dir*[ ];
double	*cone_angle*;
double	*height*;
SISLSurf	***cone*;
int	**stat*;

ARGUMENTS

Input Arguments:

bottom_pos	-	Center point of the bottom.
bottom_axis	-	One of the bottom axis (major or minor).
ellipse_ratio	-	Ratio between the other axis and bottom_axis.
axis_dir	-	Direction of the cone axis.
cone_angle	-	Angle between axis_dir and the cone at the end of bottom_axis, positive if the cone is sloping inwards.
height	-	Height of the cone, can be negative.

Output Arguments:

cone	-	Pointer to the cone produced.
stat	-	Status messages > 0 : Warning. $= 0$: Ok. < 0 : Error.

EXAMPLE OF USE

```
{
    double      bottom_pos[3];   /* Must be defined */
    double      bottom_axis[3];   /* Must be defined */
    double      ellipse_ratio =0.5; /* Elliptic cone */
    double      axis_dir[3];   /* Must be defined */
    double      cone_angle;   /* Must be defined */
    double      height;   /* Must be defined */
    SISLSurf    *cone = NULL;
    int         stat = 0;
    ...
    s1022(bottom_pos, bottom_axis, ellipse_ratio, axis_dir, cone_angle, height,
          &cone, &stat)
    ...
}
```

Chapter 8

Surface Interrogation

This chapter describes the functions in the Surface Interrogation module.

8.1 Intersection Curves

Intersection curves are tied to two objects where at least one is a surface or a curve. The representation of the intersection curves in the SISLIntcurve structure has two levels. The first level is guide points which are points in the parametric space and on the intersection curve. In every case there must be at least one guide point, but there is no upper bound. Guide points are computed in the topology detection routines. The second level is curves, one curve in the geometric space and one curve in each parameter plane if each surface is parametric. These curves are the results of the marching routines.

8.1.1 Intersection curve object.

In the library an intersection curve is stored in a struct SISLIntcurve containing the following:

int	*ipoint*;	Number of guide points defining the curve.
double	**epar1*;	Pointer to the parameter values of the points in the first object.
double	**epar2*;	Pointer to the parameter values of the points in the second object.
int	*ipar1*;	Number of parameter directions of first object.
int	*ipar2*;	Number of parameter directions of second object.
SISLCurve	**pgeom*;	Pointer to the intersection curve in the geometry space. If the curve is not computed, pgeom points to NULL.
SISLCurve	**ppar1*;	Pointer to the intersection curve in the parameter plane of the first object. If the curve is not computed, ppar1 points to NULL.
SISLCurve	**ppar2*;	Pointer to the intersection curve in the parameter plane of the second object. If the curve is not computed, ppar2 points to NULL.
int	*itype*;	Type of curve: = 1 : Straight line.

= 2 : Closed loop. No singularities.
= 3 : Closed loop. One singularity. Not used.
= 4 : Open curve. No singularity.
= 5 : Open curve. Singularity at the beginning of the curve.
= 6 : Open curve. Singularity at the end of the curve.
= 7 : Open curve. Singularity at the beginning and end of the curve.
= 8 : An isolated singularity. Not used.

Singularities are points on the intersection curve where, in an intersection between a curve and a surface, the tangent of the curve lies in the tangent plane of the surface, or in an intersection between two surfaces, the tangent plane of the surfaces coincide.

8.1.2 Create a new intersection curve object.

NAME

newIntcurve - Create and initialize a SISLIntcurve-instance. Note that the arrays *guidepar1* and *guidepar2* will be freed by freeIntcurve. In most cases the SISLIntcurve objects will be generated internally in the SISL intersection routines.

SYNOPSIS

SISLIntcurve *newIntcurve(*numgdpt*, *numpar1*, *numpar2*, *guidepar1*, *guidepar2*, type)

int	*numgdpt*;
int	*numpar1*;
int	*numpar2*;
double	*guidepar1*[ ];
double	*guidepar2*[ ];
int	*type*;

ARGUMENTS

Input Arguments:

numgdpt	-	Number of guide points that describe the curve.
numpar1	-	Number of parameter directions of first object involved in the intersection.
numpar2	-	Number of parameter directions of second object involved in the intersection.
guidepar1	-	Parameter values of the guide points in the parameter area of the first object. NB! The epar1 pointer is set to point to this array. The values are not copied.
guidepar2	-	Parameter values of the guide points in the parameter area of the second object. NB! The epar2 pointer is set to point to this array. The values are not copied.
. *type*	-	Kind of curve, see type SISLIntcurve on page 184

Output Arguments:

newIntcurve	Pointer to new SISLIntcurve. If it is impossible to allocate space for the SISLIntcurve, newIntcurve returns NULL.

EXAMPLE OF USE

```
{
    SISLIntcurve *intcurve = NULL;
    int          numgdpt = 2;
    int          numpar1 = 2;
    int          numpar2 = 2;
    double       guidepar1[4];  /* Must be defined */
    double       guidepar2[4];  /* Must be defined */
    int          type = 4;
    ...
    intcurve = newIntcurve(numgdpt, numpar1, numpar2, guidepar1,
                           guidepar2, type);
    ...
}
```

8.1.3 Delete an intersection curve object.

NAME

freeIntcurve - Free the space occupied by a SISLIntcurve.
Note that the arrays *guidepar1* and *guidepar2* will be freed as well.

SYNOPSIS

void freeIntcurve(intcurve)
SISLIntcurve **intcurve*;

ARGUMENTS

Input Arguments:

intcurve - Pointer to the SISLIntcurve to delete.

EXAMPLE OF USE

```
{
    SISLIntcurve *intcurve = NULL;
    int          numgdpt = 2;
    int          numpar1 = 2;
    int          numpar2 = 2;
    double       guidepar1[4];
    double       guidepar2[4];
    int          type = 4;
    ...
    intcurve = newIntcurve(numgdpt, numpar1, numpar2, guidepar1,
                           guidepar2, type);
    ...
    if (intcurve) freeIntcurve(intcurve);
    ...
}
```

8.1.4 Free a list of intersection curves.

NAME

freeIntcrvlist - Free a list of SISLIntcurve.

SYNOPSIS

void freeIntcrvlist(*vilist*, *icrv*)
SISLIntcurve ***vilist*;
int *icrv*;

ARGUMENTS

Input Arguments:

vilist	-	Array of pointers to pointers to instance of Intcurve.
icrv	-	number of SISLIntcurves in the list.

Output Arguments:

None	-	None.

EXAMPLE OF USE

```
{
    SISLIntcurve **vilist = NULL;
    int          icrv = 0;
    ...
    /* SISLIntcurve instances are generated for instance in surface–surface
    intersection */
    ...
    if (vilist) freeIntcrvlist(vilist, icrv);
    ...
}
```

8.2 Find the Intersections

Intersection functionality where at least one of the input geometry entities is or can be a surface.

8.2.1 Intersection between a spline curve and a straight line or a plane.

NAME

s1850 - Find all the intersections between a spline curve and a plane (if curve dimension and $dim = 3$) or a curve and a line (if curve dimension and $dim = 2$).

SYNOPSIS

void s1850(*curve*, *point*, *normal*, *dim*, *epsco*, *epsge*, *numintpt*, *intpar*, *numintcu*, *intcurve*, *stat*)

SISLCurve	**curve*;
double	*point*[ ];
double	*normal*[ ];
int	*dim*;
double	*epsco*;
double	*epsge*;
int	**numintpt*;
double	***intpar*;
int	**numintcu*;
SISLIntcurve	****intcurve*;
int	**stat*;

ARGUMENTS

Input Arguments:

curve	-	Pointer to the curve.
point	-	Point in the plane/line.
normal	-	Normal to the plane or any normal to the direction of the line.
dim	-	Dimension of the space in which the curve and the plane/line lies, *dim* must be equal to two or three.
epsco	-	Computational resolution (not used).
epsge	-	Geometry resolution.

Output Arguments:

numintpt	-	Number of single intersection points.
intpar	-	Array containing the parameter values of the single intersection points in the parameter interval of the curve. The points lie in sequence. Intersection curves are stored in intcurve.
numintcu	-	Number of intersection curves.

intcurve	-	Array of pointers to SISLIntcurve objects containing description of the intersection curves. The curves are only described by start points and end points in the parameter interval of the curve. The curve pointers point to nothing.
stat	-	Status messages > 0 : warning $= 0$: ok < 0 : error

EXAMPLE OF USE

```
{
    SISLCurve      *curve;  /* Must be defined */
    double         point[3];  /* Must be defined */
    double         normal[3]; /* Must be defined */
    int            dim = 3;
    double         epsco = 1.0e-9; /* Not used */
    double         epsge = 1.0e-6;
    int            numintpt = 0;
    double         *intpar = NULL;
    int            numintcu = 0;
    SISLIntcurve   **intcurve = NULL;
    int            stat = 0;
    ...
    s1850(curve, point, normal, dim, epsco, epsge, &numintpt, &intpar, &numintcu, &intcurve, &stat);
    ...
}
```

8.2.2 Intersection between a spline curve and a 2D circle or a sphere.

NAME

s1371 - Find all the intersections between a curve and a sphere (if curve dimension and $dim = 3$), or a curve and a circle (if curve dimension and $dim = 2$).

SYNOPSIS

void s1371(*curve*, *centre*, *radius*, *dim*, *epsco*, *epsge*, *numintpt*, *intpar*, *numintcu*, *intcurve*, *stat*)

SISLCurve	**curve*;
double	*centre*[ ];
double	*radius*;
int	*dim*;
double	*epsco*;
double	*epsge*;
int	**numintpt*;
double	***intpar*;
int	**numintcu*;
SISLIntcurve	****intcurve*;
int	**stat*;

ARGUMENTS

Input Arguments:

curve	-	Pointer to the curve.
centre	-	Centre of the circle/sphere.
radius	-	Radius of circle or sphere.
dim	-	Dimension of the space in which the curve and the circle/sphere lies, *dim* should be equal to two or three.
epsco	-	Computational resolution (not used).
epsge	-	Geometry resolution.

Output Arguments:

numintpt	-	Number of single intersection points.
intpar	-	Array containing the parameter values of the single intersection points in the parameter interval of the curve. The points lie in sequence. Intersection curves are stored in intcurve.
numintcu	-	Number of intersection curves.
intcurve	-	Array of pointers to SISLIntcurve objects containing descriptions of the intersection curves. The curves are only described by start points and end points in the parameter interval of the curve. The curve pointers point to nothing.

stat - Status messages
> 0 : warning
$= 0$: ok
< 0 : error

EXAMPLE OF USE

```
{
    SISLCurve    *curve;   /* Must be defined */
    double       centre[3];  /* Must be defined */
    double       radius;   /* Must be defined */
    int          dim = 3;
    double       epsco = 1.0e-9; /* Not used */
    double       epsge = 1.0e-6;
    int          numintpt = 0;
    double       *intpar = NULL;
    int          numintcu = 0;
    SISLIntcurve **intcurve = NULL;
    int          stat = 0;
    ...
    s1371(curve, centre, radius, dim, epsco, epsge, &numintpt, &intpar, &nu-
        mintcu, &intcurve, &stat);
    ...
}
```

8.2.3 Intersection between a spline curve and a cylinder.

NAME

s1372 - Find all the intersections between a spline curve and a cylinder.

SYNOPSIS

void s1372(*curve*, *point*, *dir*, *radius*, *dim*, *epsco*, *epsge*, *numintpt*, *intpar*, *numintcu*, *intcurve*, *stat*)

SISLCurve	**curve*;
double	*point*[ ];
double	*dir*[ ];
double	*radius*;
int	*dim*;
double	*epsco*;
double	*epsge*;
int	**numintpt*;
double	***intpar*;
int	**numintcu*;
SISLIntcurve	****intcurve*;
int	**stat*;

ARGUMENTS

Input Arguments:

curve	-	Pointer to the curve.
point	-	Point on the cylinder axis.
dir	-	Direction of the cylinder axis.
radius	-	Radius of the cylinder.
dim	-	Dimension of the space in which the cylinder and the curve lie, dim should be equal to three.
epsco	-	Computational resolution (not used).
epsge	-	Geometry resolution.

Output Arguments:

numintpt	-	Number of single intersection points.
intpar	-	Array containing the parameter values of the single intersection points in the parameter interval of the curve. The points lie in sequence. Intersection curves are stored in intcurve.
numintcu	-	Number of intersection curves.
intcurve	-	Array of pointers to the SISLIntcurve objects containing descriptions of the intersection curves. The curves are only described by start points and end points in the parameter interval of the curve. The curve pointers point to nothing.

stat - Status messages
> 0 : warning
$= 0$: ok
< 0 : error

EXAMPLE OF USE

```
{
    SISLCurve    *curve;  /* Must be defined */
    double       point[3];  /* Must be defined */
    double       dir[3];  /* Must be defined */
    double       radius;  /* Must be defined */
    int          dim = 3;
    double       epsco = 1.0e-9 /* Not used */;
    double       epsge = 1.0e-6;
    int          numintpt = 0;
    double       *intpar = NULL;
    int          numintcu = 0;
    SISLIntcurve **intcurve = NULL;
    int          stat = 0;
    ...
    s1372(curve, point, dir, radius, dim, epsco, epsge, &numintpt,
          &intpar, &numintcu, &intcurve, &stat);
    ...
}
```

8.2.4 Intersection between a spline curve and a cone.

NAME

s1373 - Find all the intersections between a curve and a cone.

SYNOPSIS

void s1373(*curve*, *top*, *dir*, *conept*, *dim*, *epsco*, *epsge*, *numintpt*, *intpar*, *numintcu*, *intcurve*, *stat*)

SISLCurve	**curve*;
double	*top*[ ];
double	*axispt*[ ];
double	*conept*[ ];
int	*dim*;
double	*epsco*;
double	*epsge*;
int	**numintpt*;
double	***intpar*;
int	**numintcu*;
SISLIntcurve	****intcurve*;
int	**stat*;

ARGUMENTS

Input Arguments:

curve	-	Pointer to the curve.
top	-	Top point of the cone.
axispt	-	Point on the cone axis.
conept	-	Point on the cone surface, other than the top point.
dim	-	Dimension of the space in which the cone and the curve lie, dim should be equal to three.
epsco	-	Computational resolution (not used).
epsge	-	Geometry resolution.

Output Arguments:

numintpt	-	Number of single intersection points.
intpar	-	Array containing the parameter values of the single intersection points in the parameter interval of the curve. The points lie in sequence. Intersection curves are stored in intcurve.
numintcu	-	Number of intersection curves.
intcurve	-	Array of pointers to the SISLIntcurve object containing descriptions of the intersection curves. The curves are only described by start points and end points in the parameter interval of the curve. The curve pointers point to nothing.

stat - Status messages
> 0 : warning
$= 0$: ok
< 0 : error

EXAMPLE OF USE

```
{
    SISLCurve    *curve;  /*Must be defined */
    double       top[3];  /* Must be defined */
    double       dir[3];  /* Must be defined */
    double       conept[3];  /* Must be defined */
    int          dim = 3;
    double       epsco = 1.0e-9; /* Not used */
    double       epsge = 1.0e-6;
    int          numintpt = 0;
    double       *intpar = NULL;
    int          numintcu = 0;
    SISLIntcurve **intcurve = NULL;
    int          stat = 0;
    ...
    s1373(curve, top, dir, conept, dim, epsco, epsge, &numintpt, &intpar, &numintcu, &intcurve, &stat);
    ...
}
```

8.2.5 Intersection between a spline curve and an elliptic cone.

NAME

s1502 - Find all the intersections between a curve and an elliptic cone.

SYNOPSIS

void s1502(*curve*, *basept*, *normdir*, *ellipaxis*, *alpha*, *ratio*, *dim*, *epsco*, *epsge*, *numintpt*, *intpar*, *numintcu*, *intcurve*, *stat*)

SISLCurve	**curve*;
double	*basept*[ ];
double	*normdir*[ ];
double	*ellipaxis*[ ];
double	*alpha*;
double	*ratio*;
int	*dim*;
double	*epsco*;
double	*epsge*;
int	**numintpt*;
double	***intpar*;
int	**numintcu*;
SISLIntcurve	****intcurve*;
int	**stat*;

ARGUMENTS

Input Arguments:

curve	-	Pointer to the curve.
basept	-	Base point of the cone, centre of elliptic base.
normdir	-	Direction of the cone axis, normal to the elliptic base. The default is pointing from the base point to the top point of the cone.
ellipaxis	-	One of the axes of the ellipse (major or minor).
alpha	-	The opening angle of the cone at the ellipaxis.
ratio	-	The ratio of the major and minor axes = ellipaxis/otheraxis.
dim	-	Dimension of the space in which the cone and the curve lie, dim should be equal to three.
epsco	-	Computational resolution (not used).
epsge	-	Geometry resolution.

Output Arguments:

numintpt	-	Number of single intersection points.
intpar	-	Array containing the parameter values of the single intersection points in the parameter interval of the curve. The points lie in sequence. Intersection curves are stored in intcurve.
numintcu	-	Number of intersection curves.
intcurve	-	Array of pointers to the SISLIntcurve object containing descriptions of the intersection curves. The curves are only described by start points and end points in the parameter interval of the curve. The curve pointers point to nothing.
stat	-	Status messages > 0 : warning $= 0$: ok < 0 : error

EXAMPLE OF USE

```
{
    SISLCurve    *curve;   /* Must be defined */
    double       basept[3];   /* Must be defined */
    double       normdir[3];   /* Must be defined */
    double       ellipaxis[3];   /* Must be defined */
    double       alpha;   /* Must be defined */
    double       ratio = 1.5;
    int          dim = 3;
    double       epsco = 1.0e-9; /* Not used */
    double       epsge = 1.0e-6;
    int          numintpt = 0;
    double       *intpar = NULL;
    int          numintcu = 0;
    SISLIntcurve **intcurve = NULL;
    int          stat = 0;
    . . .
    s1502(curve, basept, normdir, ellipaxis, alpha, ratio, dim, epsco, epsge, &numintpt, &intpar, &numintcu, &intcurve, &stat);
    . . .
}
```

8.2.6 Intersection between a curve and a torus.

NAME

s1375 - Find all the intersections between a spline curve and a torus.

SYNOPSIS

void s1375(*curve*, *centre*, *normal*, *centdist*, *rad*, *dim*, *epsco*, *epsge*, *numintpt*, *intpar*, *numintcu*, *intcurve*, *stat*)

SISLCurve	**curve*;
double	*centre[ ]*;
double	*normal*[ ];
double	*centdist*;
double	*rad*;
int	*dim*;
double	*epsco*;
double	*epsge*;
int	**numintpt*;
double	***intpar*;
int	**numintcu*;
SISLIntcurve	****intcurve*;
int	**stat*;

ARGUMENTS

Input Arguments:

curve	-	Pointer to the curve.
centre	-	The centre of the torus (lying in the symmetry plane)
normal	-	Normal of symmetry plane.
centdist	-	Distance from the centre of the cone to the centre circle of the torus.
rad	-	The radius of the torus surface.
dim	-	Dimension of the space in which the torus and the curve lie, dim should be equal to three.
epsco	-	Computational resolution (not used).
epsge	-	Geometry resolution.

Output Arguments:

numintpt	-	Number of single intersection points.
intpar	-	Array containing the parameter values of the single intersection points in the parameter interval of the curve. The points lie in sequence. Intersection curves are stored in intcurve.
numintcu	-	Number of intersection curves.
intcurve	-	Array of pointers to the SISLIntcurve objects containing descriptions of the intersection curves. The curves are only described by start points and end points in the parameter interval of the curve. The curve pointers point to nothing.
stat	-	Status messages > 0 : warning $= 0$: ok < 0 : error

EXAMPLE OF USE

```
{
    SISLCurve    *curve;  /* Must be defined */
    double       centre[3];  /* Must be defined */
    double       normal[3]; /* Must be defined */
    double       centdist;  /* Must be defined */
    double       rad;  /* Must be defined */
    int          dim = 3;
    double       epsco = 1.0e-9; /* Not used */
    double       epsge = 1.0e-6;
    int          numintpt = 0;
    double       *intpar = NULL;
    int          numintcu = 0;
    SISLIntcurve **intcurve = NULL;
    int          stat = 0;
    ...
    s1375(curve, centre, normal, centdist, rad, dim, epsco, epsge,
          &numintpt, &intpar, &numintcu, &intcurve, &stat);
    ...
}
```

8.2.7 Intersection between a surface and a point.

NAME

s1870 - Find all intersections between a spline surface and a point.

SYNOPSIS

void s1870(*ps1*, *pt1*, *idim*, *aepsge*, *jpt*, *gpar1*, *jcrv*, *wcurve*, *jstat*)

```
SISLSurf     *ps1;
double       pt1[];
int          idim;
double       aepsge;
int          *jpt;
double       **gpar1;
int          *jcrv;
SISLIntcurve ***wcurve;
int          *jstat;
```

ARGUMENTS

Input Arguments:

ps1	-	Pointer to the surface.
pt1	-	Coordinates of the point.
idim	-	Number of coordinates in pt1.
aepsge	-	Geometry resolution.

Output Arguments:

jpt	-	Number of single intersection points.
gpar1	-	Array containing the parameter values of the single intersection points in the parameter interval of the surface. The points lie continuous. Intersection curves are stored in wcurve.
jcrv	-	Number of intersection curves.
wcurve	-	Array containing descriptions of the intersection curves. The curves are only described by points in the parameter plane. The curve-pointers points to nothing. If the curves given as input are degnenerate an intersection point can be returned as an intersection curve. Use s1327 to decide if an intersection curve is a point on one of the curves.
jstat	-	Status messages > 0 : Warning. $= 0$: Ok. < 0 : Error.

EXAMPLE OF USE

```
{
    SISLSurf       *ps1;   /* Must be defined */
    double         pt1[3];   /* Must be defined */
    int            idim = 3;
    double         aepsge = 1.0e-6;
    int            jpt = 0;
    double         *gpar1 = NULL;
    int            jcrv = 0;
    SISLIntcurve **wcurve = NULL;
    int            jstat = 0;
    ...
    s1870(ps1, pt1, idim, aepsge, &jpt, &gpar1, &jcrv, &wcurve, &jstat);
    ...
}
```

8.2.8 Intersection between a spline surface and a straight line.

NAME

s1856 - Find all intersections between a tensor-product surface and an infinite straight line.

SYNOPSIS

void s1856(*surf*, *point*, *linedir*, *dim*, *epsco*, *epsge*, *numintpt*, *pointpar*, *numintcr*, *intcurves*, *stat*)

```
SISLSurf      *surf;
double        point[ ];
double        linedir[ ];
int           dim;
double        epsco;
double        epsge;
int           *numintpt;
double        **pointpar;
int           *numintcr;
SISLIntcurve  ***intcurves;
int           *stat;
```

ARGUMENTS

Input Arguments:

surf	-	Pointer to the surface.
point	-	Point on the line.
linedir	-	Direction vector of the line.
dim	-	Dimension of the space in which the line lies.
epsco	-	Computational resolution (not used).
epsge	-	Geometry resolution.

Output Arguments:

numintpt	-	Number of single intersection points.
pointpar	-	Array containing the parameter values of the single intersection points in the parameter plane of the surface. The points lie in sequence. Intersection curves are stored in intcurves.
numintcr	-	Number of intersection curves.
intcurves	-	Array containing the description of the intersection curves. The curves are only described by start points and end points in the parameter plane. The curve pointers point to nothing.
stat	-	Status messages > 0 : warning $= 0$: ok < 0 : error

EXAMPLE OF USE

```
{
    SISLSurf      *surf;  /* Must be defined */
    double        point[3];  /* Must be defined */
    double        linedir[3]; /* Must be defined */
    int           dim = 3;
    double        epsco = 1.0e-9; /* Not used */
    double        epsge =1.0e-6;
    int           numintpt = 0;
    double        *pointpar = NULL;
    int           numintcr = 0;
    SISLIntcurve **intcurves = NULL;
    int           stat = 0;
    . . .
    s1856(surf, point, linedir, dim, epsco, epsge, &numintpt, &pointpar, &nu-
          mintcr, &intcurves, &stat);
    . . .
}
```

8.2.9 Newton iteration on the intersection between a 3D NURBS surface and a line.

NAME

s1518 - Newton iteration on the intersection between a 3D NURBS surface and a line. If a good initial guess is given, the intersection will be found quickly. However if a bad initial guess is given, the iteration might not converge. We only search in the rectangular subdomain specified by "start" and "end". This can be the whole domain if desired.

SYNOPSIS

```
void s1518(surf, point, dir, epsge, start, end, parin, parout, stat)
    SISLSurf    *surf;
    double      point[];
    double      dir[];
    double      epsge;
    double      start[];
    double      end[];
    double      parin[];
    double      parout[];
    int         *stat;
```

ARGUMENTS

Input Arguments:

surf	-	The NURBS surface.
point	-	A point on the line.
dir	-	The vector direction of the line (not necessarily normalized).
epsge	-	Geometric resolution.
start	-	Lower limits of search rectangle (umin, vmin).
end	-	Upper limits of search rectangle (umax, vmax).
parin	-	Initial guess (u0,v0) for parameter point of intersection (which should be inside the search rectangle).

Output Arguments:

parout	-	Parameter point (u,v) of intersection.
jstat	-	status messages = 1 : Intersection found. ¡ 0 : error.

EXAMPLE OF USE

```
{
    SISLSurf    *surf;  /* Must be defined */
    double      point[3];  /* Must be defined */
    double      dir[3];  /* Must be defined */
    double      epsge = 1.0e-6;
    double      start[2];  /* Must be defined */
    double      end[2];  /* Must be defined */
    double      parin[2];  /* Guess parameter. Must be defined */
    double      parout[2];
    int         stat = 0;
    . . .
```

s1518(*surf*, *point*, *dir*, *epsge*, *start*, *end*, *parin*, *parout*, &*stat*);
...
}

8.2.10 Convert a surface/line intersection into a two-dimensional surface/origo intersection

NAME

s1328 - Put the equation of the surface pointed at by psold into two planes given by the point epoint and the normals enorm1 and enorm2. The result is an equation where the new two-dimensional surface rsnew is to be equal to origo.

SYNOPSIS

```
void s1328(psold, epoint, enorm1, enorm2, idim, rsnew, jstat)
    SISLSurf    *psold;
    double      epoint[ ];
    double      enorm1[ ];
    double      enorm2[ ];
    int         idim;
    SISLSurf    **rsnew;
    int         *jstat;
```

ARGUMENTS

Input Arguments:

psold	-	Pointer to input surface.
epoint	-	SISLPoint in the planes.
enorm1	-	Normal to the first plane.
enorm2	-	Normal to the second plane.
idim	-	Dimension of the space in which the planes lie.

Output Arguments:

rsnew	-	dimensional surface.
jstat	-	status messages > 0 : warning $= 0$: ok < 0 : error

EXAMPLE OF USE

```
{
    SISLSurf    *psold;  /* Must be defined */
    double      epoint[3];  /* Must be defined */
    double      enorm1[3];  /* Must be defined */
    double      enorm2[3];  /* Must be defined */
    int         idim = 3;
    SISLSurf    **rsnew = NULL;
    int         *jstat = 0;
    ...
    s1328(psold, epoint, enorm1, enorm2, idim, &rsnew, &jstat);
    ...
}
```

8.2.11 Intersection between a spline surface and a circle.

NAME

s1855 - Find all intersections between a tensor-product surface and a full circle.

SYNOPSIS

void s1855(*surf*, *centre*, *radius*, *normal*, *dim*, *epsco*, *epsge*, *numintpt*, *pointpar*, *numintcr*, *intcurves*, *stat*)

SISLSurf	**surf*;
double	*centre*[ ];
double	*radius*;
double	*normal*[ ];
int	*dim*;
double	*epsco*;
double	*epsge*;
int	**numintpt*;
double	***pointpar*;
int	**numintcr*;
SISLIntcurve	****intcurves*;
int	**stat*;

ARGUMENTS

Input Arguments:

surf	-	Pointer to the surface.
centre	-	Centre of the circle.
radius	-	Radius of the circle.
normal	-	Normal vector to the plane in which the circle lies.
epsco	-	Computational resolution (not used).
epsge	-	Geometry resolution.

Output Arguments:

numintpt	-	Number of single intersection points.
pointpar	-	Array containing the parameter values of the single intersection points in the parameter plane of the surface. The points lie in sequence. Intersection curves are stored in intcurves.
numintcr	-	Number of intersection curves.
intcurves	-	Array containing the description of the intersection curves. The curves are only described by start points and end points in the parameter plane. The curve pointers point to nothing.
stat	-	Status messages > 0 : warning $= 0$: ok < 0 : error

EXAMPLE OF USE

```
{
    SISLSurf       *surf;    /* Must be defined */
    double         centre[3];  /* Must be defined */
    double         radius;   /* Must be defined */
    double         normal[3];  /* Must be defined */
    int            dim = 3;
    double         epsco = 1.0e-9; /* Not used */
    double         epsge = 1.0e-6;
    int            numintpt = 0;
    double         *pointpar = NULL;
    int            numintcr = 0;
    SISLIntcurve   **intcurves = NULL;
    int            stat = 0;
    . . .
    s1855(surf, centre, radius, normal, dim, epsco, epsge, &numintpt, &pointpar,
          &numintcr, &intcurves, &stat);
    . . .
}
```

8.2.12 Intersection between a surface and a curve.

NAME

s1858 - Find all intersections between a surface and a curve. Intersection curves are described by guide points. To pick the intersection curves use s1712() described on page 127.

SYNOPSIS

void s1858(*surf*, *curve*, *epsco*, *epsge*, *numintpt*, *pointpar1*, *pointpar2*, *numintcr*, *intcurves*, *stat*)

SISLSurf	**surf*;
SISLCurve	**curve*;
double	*epsco*;
double	*epsge*;
int	**numintpt*;
double	***pointpar1*;
double	***pointpar2*;
int	**numintcr*;
SISLIntcurve	****intcurves*;
int	**stat*;

ARGUMENTS

Input Arguments:

surf	-	Pointer to the surface.
curve	-	Pointer to the curve.
epsco	-	Computational resolution (not used).
epsge	-	Geometry resolution.

Output Arguments:

numintpt	-	Number of single intersection points.
pointpar1	-	Array containing the parameter values of the single intersection points in the parameter plane of the surface. The points lie in sequence. Intersection curves are stored in intcurves.
pointpar2	-	Array containing the parameter values of the single intersection points in the parameter interval of the curve.
numintcr	-	Number of intersection curves.
intcurves	-	Array containing the description of the intersection curves. The curves are only described by start points and end points (guide points) in the parameter plane. The curve pointers point to nothing. If the curves given as input are degenerate, an intersection point can be returned as an intersection curve.

stat - Status messages
> 0 : warning
$= 0$: ok
< 0 : error

EXAMPLE OF USE

```
{
    SISLSurf       *surf;  /* Must be defined */
    SISLCurve      *curve; /* Must be defined */
    double         epsco = 1.0e-9; /* Not used */
    double         epsge = 1.0e-6;
    int            numintpt = 0;
    double         *pointpar1 = NULL;
    double         *pointpar2 = NULL;
    int            numintcr = 0;
    SISLIntcurve   **intcurves = NULL;
    int            stat = 0;
    ...
    s1858(surf, curve, epsco, epsge, &numintpt, &pointpar1, &pointpar2, &nu-
        mintcr, &intcurves, &stat);
    ...
}
```

8.3 Find the Topology of the Intersection

8.3.1 Find the topology for the intersections between a spline surface and a plane.

NAME

s1851 - Find all intersections between a tensor-product surface and a plane. Intersection curves are described by guide points. To make the intersection curves use s1314() described on page 233.

SYNOPSIS

void s1851(*surf*, *point*, *normal*, *dim*, *epsco*, *epsge*, *numintpt*, *pointpar*, *numintcr*, *intcurves*, *stat*)

SISLSurf	**surf*;
double	*point*[ ];
double	*normal*[ ];
int	*dim*;
double	*epsco*;
double	*epsge*;
int	**numintpt*;
double	***pointpar*;
int	**numintcr*;
SISLIntcurve	****intcurves*;
int	**stat*;

ARGUMENTS

Input Arguments:

surf	-	Pointer to surface
point	-	Point in the plane.
normal	-	Normal to the plane.
dim	-	Dimension of the space in which the plane lies.
epsco	-	Computational resolution (not used).
epsge	-	Geometry resolution.

Output Arguments:

numintpt	-	Number of single intersection points.
pointpar	-	Array containing the parameter values of the single intersection points in the parameter plane of the surface. The points lie in sequence. Intersection curves are stored in intcurves.
numintcr	-	Number of intersection curves.
intcurves	-	Array containing descriptions of the intersection curves. The curves are only described by start points and end points (guide points) in the parameter plane. The curve pointers point to nothing.

stat - Status messages
> 0 : warning
$= 0$: ok
< 0 : error

EXAMPLE OF USE

```
{
    SISLSurf      *surf;  /* Must be defined */
    double        point[3];  /* Must be defined */
    double        normal[3];  /* Must be defined */
    int           dim = 3;
    double        epsco = 1.0e-9; /* Not used */
    double        epsge = 1.0e-6;
    int           numintpt = 0;
    double        *pointpar = NULL;
    int           numintcr = 0;
    SISLIntcurve **intcurves = NULL;
    int           stat = 0;
    ...
    s1851(surf, point, normal, dim, epsco, epsge, &numintpt, &pointpar, &numintcr, &intcurves, &stat);
    ...
}
```

8.3.2 Find the topology for the intersection between a spline surface and a sphere.

NAME

s1852 - Find all intersections between a tensor-product surface and a sphere. Intersection curves are described by guide points. To produce the intersection curves use s1315() described on page 235.

SYNOPSIS

void s1852(*surf*, *centre*, *radius*, *dim*, *epsco*, *epsge*, *numintpt*, *pointpar*, *numintcr*, *intcurves*, *stat*)

SISLSurf	**surf*;
double	*centre* [];
double	*radius*;
int	*dim*;
double	*epsco*;
double	*epsge*;
int	**numintpt*;
double	***pointpar*;
int	**numintcr*;
SISLIntcurve	****intcurves*;
int	**stat*;

ARGUMENTS

Input Arguments:

surf	-	Pointer to the surface.
centre	-	Center of the sphere.
radius	-	Radius of the sphere.
dim	-	Dimension of the space in which the sphere lies.
epsco	-	Computational resolution (not used).
epsge	-	Geometry resolution.

Output Arguments:

numintpt	-	Number of single intersection points.
pointpar	-	Array containing the parameter values of the single intersection points in the parameter plane of the surface. The points lie in sequence. Intersection curves are stored in intcurves.
numintcr	-	Number of intersection curves.
intcurves	-	Array containing description of the intersection curves. The curves are only described by start points and end points (guide points) in the parameter plane. The curve pointers point to nothing.
stat	-	Status messages > 0 : warning $= 0$: ok < 0 : error

EXAMPLE OF USE

{

SISLSurf **surf*; /* Must be defined */

```
    double        centre[3];  /* Must be defined */
    double        radius;  /* Must be defined */
    int           dim = 3;
    double        epsco = 1.0e-9; /* Not used */
    double        epsge = 1.0e-6;
    int           numintpt = 0;
    double        *pointpar = NULL;
    int           numintcr = 0;
    SISLIntcurve  **intcurves = NULL;
    int           stat = 0;
    ...
    s1852(surf, centre, radius, dim, epsco, epsge, &numintpt, &pointpar, &nu-
          mintcr, &intcurves, &stat);
    ...
}
```

8.3.3 Find the topology for the intersections between a spline surface and a cylinder.

NAME

s1853 - Find all intersections between a tensor-product surface and a cylinder. Intersection curves are described by guide points. To produce the intersection curves use s1316() described on page 237.

SYNOPSIS

void s1853(*surf*, *point*, *cyldir*, *radius*, *dim*, *epsco*, *epsge*, *numintpt*, *pointpar*, *numintcr*, *intcurves*, *stat*)

SISLSurf	**surf*;
double	*point*[ ];
double	*cyldir*[ ];
double	*radius*;
int	*dim*;
double	*epsco*;
double	*epsge*;
int	**numintpt*;
double	***pointpar*;
int	**numintcr*;
SISLIntcurve	****intcurves*;
int	**stat*;

ARGUMENTS

Input Arguments:

surf	-	Pointer to the surface.
point	-	Point on the axis of the cylinder.
cyldir	-	The direction vector of the axis of the cylinder.
radius	-	Radius of the cylinder.
dim	-	Dimension of the space in which the cylinder lies.
epsco	-	Computational resolution (not used).
epsge	-	Geometry resolution.

Output Arguments:

numintpt	-	Number of single intersection points.
pointpar	-	Array containing the parameter values of the single intersection points in the parameter plane of the surface. The points lie in sequence. Intersection curves are stored in intcurves.
numintcr	-	Number of intersection curves.
intcurves	-	Array containing description of the intersection curves. The curves are only described by start points and end points (guide points) in the parameter plane. The curve pointers point to nothing.

stat - Status messages
> 0 : warning
$= 0$: ok
< 0 : error

EXAMPLE OF USE

```
{
    SISLSurf    *surf;   /* Must be defined */
    double      point[3];  /* Must be defined */
    double      cyldir[3];  /* Must be defined */
    double      radius;   /* Must be defined */
    int         dim = 3;
    double      epsco = 1.0e-9; /* Not used */
    double      epsge = 1.0e-6;
    int         numintpt = 0;
    double      *pointpar = NULL;
    int         numintcr = 0;
    intcurve    **intcurves = NULL;
    int         stat = 0;
    ...
    s1853(surf, point, cyldir, radius, dim, epsco, epsge, &numintpt, &pointpar,
          &numintcr, &intcurves, &stat);
    ...
}
```

8.3.4 Find the topology for the intersections between a spline surface and a cone.

NAME

s1854 - Find all intersections between a tensor-product surface and a cone. Intersection curves are described by guide points. To produce the intersection curves use s1317() described on page 239.

SYNOPSIS

void s1854(*surf*, *toppt*, *axispt*, *conept*, *dim*, *epsco*, *epsge*, *numintpt*, *pointpar*, *numintcr*, *intcurves*, *stat*)

SISLSurf	**surf*;
double	*toppt*[ ];
double	*axispt*[ ];
double	*conept*[ ];
int	*dim*;
double	*epsco*;
double	*epsge*;
int	**numintpt*;
double	***pointpar*;
int	**numintcr*;
SISLIntcurve	****intcurves*;
int	**stat*;

ARGUMENTS

Input Arguments:

surf	-	Pointer to the surface
toppt	-	Top point of the cone.
axispt	-	Point on the axis of the cone, axispt must be different from toppt.
conept	-	Point on the cone surface, conept must be different from toppt.
dim	-	Dimension of the space in which the cone lies.
epsco	-	Computational resolution (not used).
epsge	-	Geometry resolution.

Output Arguments:

numintpt	-	Number of single intersection points.
pointpar	-	Array containing the parameter values of the single intersection points in the parameter plane of the surface. The points lie in sequence. Intersection curves are stored in intcurves.
numintcr	-	Number of intersection curves.
intcurves	-	Array containing the description of the intersection curves. The curves are only described by start points and end points (guide points) in the parameter plane. The curve pointers point to nothing.
stat	-	Status messages > 0 : warning $= 0$: ok

< 0 : error

EXAMPLE OF USE

```
{
    SISLSurf      *surf;   /* Must be defined */
    double        toppt[3];   /* Must be defined */
    double        axispt[3];   /* Must be defined */
    double        conept[3];   /* Must be defined */
    int           dim = 3;
    double        epsco = 1.0e-9; /* Not used */
    double        epsge = 1.0e-6;
    int           numintpt = 0;
    double        *pointpar = NULL;
    int           numintcr = 0;
    SISLIntcurve  **intcurves = NULL;
    int           stat = 0;
    ...
    s1854(surf, toppt, axispt, conept, dim, epsco, epsge, &numintpt, &pointpar,
          &numintcr, &intcurves, &stat);
    ...
}
```

8.3.5 Find the topology for the intersections between a spline surface and an elliptic cone.

NAME

s1503 - Find all intersections between a tensor-product surface and an elliptic cone. Intersection curves are described by guide points. To produce the intersection curves use s1501() described on page 241.

SYNOPSIS

void s1503(*surf*, *basept*, *normdir*, *ellipaxis*, *alpha*, *ratio*, *dim*, *epsco*, *epsge*, *numintpt*, *pointpar*, *numintcr*, *intcurves*, *stat*)

SISLSurf	**surf*;
double	*basept*[ ];
double	*normdir*[ ];
double	*ellipaxis*[ ];
double	*alpha*;
double	*ratio*;
int	*dim*;
double	*epsco*;
double	*epsge*;
int	**numintpt*;
double	***pointpar*;
int	**numintcr*;
SISLIntcurve	****intcurves*;
int	**stat*;

ARGUMENTS

Input Arguments:

surf	-	Pointer to the surface
basept	-	Base point of the cone, centre of elliptic base.
normdir	-	Direction of the cone axis, normal to the elliptic base. The default is pointing from the base point to the top point.
ellipaxis	-	One of the axes of the ellipse (major or minor). The other axis will be calculated as $normdir \times ellipaxis$ scaled with *ratio*.
alpha	-	The opening angle in radians of the cone at the ellipaxis.
ratio	-	The ratio of the major and minor axes = ellipaxis/otheraxis.
dim	-	Dimension of the space in which the cone lies.
epsco	-	Computational resolution (not used).
epsge	-	Geometry resolution.

Output Arguments:

numintpt	-	Number of single intersection points.
pointpar	-	Array containing the parameter values of the single intersection points in the parameter plane of the surface. The points lie in sequence. Intersection curves are stored in intcurves.
numintcr	-	Number of intersection curves.
intcurves	-	Array containing the description of the intersection curves. The curves are only described by start points and end points (guide points) in the parameter plane. The curve pointers point to nothing.
stat	-	Status messages > 0 : warning $= 0$: ok < 0 : error

EXAMPLE OF USE

```
{
    SISLSurf      *surf;  /* Must be defined */
    double        basept[3];  /* Must be defined */
    double        normdir[3];  /* Must be defined */
    double        ellipaxis[3];  /* Must be defined */
    double        alpha;  /* Must be defined */
    double        ratio;  /* Must be defined */
    int           dim = 3;
    double        epsco = 1.0e-9; /* Not used */
    double        epsge = 1.0e-6;
    int           numintpt = 0;
    double        *pointpar = NULL;
    int           numintcr = 0;
    SISLIntcurve  **intcurves = NULL;
    int           stat = 0;
    ...
    s1503(surf, basept, normdir, ellipaxis, alpha, ratio, dim, epsco, epsge, &numintpt, &pointpar, &numintcr, &intcurves, &stat);
    ...
}
```

8.3.6 Find the topology for the intersections between a spline surface and a torus.

NAME

s1369 - Find all intersections between a surface and a torus. Intersection curves are described by guide points. To produce the intersection curves use s1318() described on page 244.

SYNOPSIS

void s1369(*surf*, *centre*, *normal*, *cendist*, *radius*, *dim*, *epsco*, *epsge*, *numintpt*, *pointpar*, *numintcr*, *intcurves*, *stat*)

SISLSurf	**surf*;
double	*centre*[ ];
double	*normal*[ ];
double	*cendist*;
double	*radius*;
int	*dim*;
double	*epsco*;
double	*epsge*;
int	**numintpt*;
double	***pointpar*;
int	**numintcr*;
SISLIntcurve	****intcurves*;
int	**stat*;

ARGUMENTS

Input Arguments:

surf	-	Pointer to the surface.
centre	-	The centre of the torus (lying in the symmetry plane)
normal	-	Normal to the symmetry plane.
cendist	-	Distance from centre to centre circle of the torus.
radius	-	The radius of the torus surface.
dim	-	Dimension of the space in which the torus lies. dim should be equal to two or three.
epsco	-	Computational resolution (not used).
epsge	-	Geometry resolution.

Output Arguments:

numintpt	-	Number of single intersection points.
pointpar	-	Array containing the parameter values of the single intersection points in the parameter plane of the surface. The points lie in sequence. Intersection curves are stored in intcurves.
numintcr	-	Number of intersection curves.
intcurves	-	Array containing the description of the intersection curves. The curves are only described by start points and end points (guide points) in the parameter planes. The curve pointers point to nothing.
stat	-	Status messages > 0 : warning $= 0$: ok

< 0 : error

EXAMPLE OF USE

```
{
    SISLSurf       *surf;  /* Must be defined */
    double         centre[3];  /* Must be defined */
    double         normal[3];  /* Must be defined */
    double         cendist;  /* Must be defined */
    double         radius;  /* Must be defined */
    int            dim = 3;
    double         epsco = 1.0e-9; /* Not used */
    double         epsge = 1.0e-6;
    int            numintpt = 0;
    double         *pointpar = NULL;
    int            numintcr = 0;
    SISLIntcurve   **intcurves = NULL;
    int            stat = 0;
    . . .
    s1369(surf, centre, normal, cendist, radius, dim, epsco, epsge,
          &numintpt, &pointpar, &numintcr, &intcurves, &stat);
    . . .
}
```

8.3.7 Find the topology for the intersection between two spline surfaces.

NAME

s1859 - Find all intersections between two surfaces. Intersection curves are described by guide points. To produce the intersection curves use s1310() described on page 247.

SYNOPSIS

void s1859 (*surfl*, *surf2*, *epsco*, *epsge*, *numintpt*, *pointpar1*, *pointpar2*, *numintcr*, *intcurves*, *stat*)

```
SISLSurf      *surf1;
SISLSurf      *surf2;
double        epsco;
double        epsge;
int           *numintpt;
double        **pointpar1;
double        **pointpar2;
int           *numintcr;
SISLIntcurve  ***intcurves;
int           *stat;
```

ARGUMENTS

Input Arguments:

surf1	-	Pointer to the first surface.
surf2	-	Pointer to the second surface.
epsco	-	Computational resolution (not used).
epsge	-	Geometry resolution.

Output Arguments:

numintpt	-	Number of single intersection points.
pointpar1	-	Array containing the parameter values of the single intersection points in the parameter plane of the first surface. The points lie in sequence. Intersection curves are stored in intcurves.
pointpar2	-	Array containing the parameter values of the single intersection points in the parameter plane of the second surface.
numintcr	-	Number of intersection curves.
intcurves	-	Array containing description of the intersection curves. The curves are only described by start points and end points (guide points) in the parameter planes of the surfaces. The curve pointers point to nothing.
stat	-	Status messages > 0 : warning $= 0$: ok < 0 : error

EXAMPLE OF USE

```
{
    SISLSurf      *surf1;   /* Must be defined */
    SISLSurf      *surf2;   /* Must be defined */
    double        epsco = 1.0e-9; /* Not used */
    double        epsge = 1.0e-6;
    int           numintpt = 0;
    double        *pointpar1 = NULL;
    double        *pointpar2 = NULL;
    int           numintcr = 0;
    SISLIntcurve **intcurves = NULL;
    int           stat = 0;
    ...
    s1859(surfl, surf2, epsco, epsge, &numintpt, &pointpar1, &pointpar2, &numintcr, &intcurves, &stat);
    ...
}
```

8.4 Find the Topology of a Silhouette

8.4.1 Find the topology of the silhouette curves of a spline surface, using parallel projection.

NAME

s1860 - Find the silhouette curves and points of a surface when the surface is viewed from a specific direction (i.e. parallel projection). In addition to the points and curves found by this routine, break curves and edge-curves might be silhouette curves. Silhouette curves are described by guide points. To produce the silhouette curves use s1319() described on page 249.

NOTE

The silhouette curves are defined as curves on the surface where the inner product of the surface normal and the direction vector of the viewing is 0. This definition will include surface points where the normal is zero.

SYNOPSIS

void s1860(*surf*, *viewdir*, *dim*, *epsco*, *epsge*, *numsilpt*, *pointpar*, *numsilcr*, *silcurves*, *stat*)

SISLSurf	**surf*;
double	*viewdir*[];
int	*dim*;
double	*epsco*;
double	*epsge*;
int	**numsilpt*;
double	***pointpar*;
int	**numsilcr*;
SISLIntcurve	****silcurves*;
int	**stat*;

ARGUMENTS

Input Arguments:

surf	-	Pointer to the surface.
viewdir	-	The direction vector of the viewing.
dim	-	Dimension of the space in which *viewdir* lies.
epsco	-	Computational resolution (not used).
epsge	-	Geometry resolution.

Output Arguments:

numsilpt	-	Number of single silhouette points.
pointpar	-	Array containing the parameter values of the single silhouette points in the parameter plane of the surface. The points lie in sequence. Silhouette curves are stored in silcurves.
numsilcr	-	Number of silhouette curves.

silcurves	-	Array containing the description of the silhouette curves. The curves are only described by start points and end points (guide points) in the parameter plane. The curve pointers point to nothing.
stat	-	Status messages > 0 : warning $= 0$: ok < 0 : error

EXAMPLE OF USE

```
{
    SISLSurf       *surf;  /* Must be defined */
    double         viewdir[3];  /* Must be defined */
    int            dim = 3;
    double         epsco = 1.0e-9; /* Not used */
    double         epsge = 1.0e-6;
    int            numsilpt = 0;
    double         *pointpar = NULL;
    int            numsilcr = 0;
    SISLIntcurve   **silcurves = NULL;
    int            stat = 0;
    . . .
    s1860(surf, viewdir, dim, epsco, epsge, &numsilpt, &pointpar,
          &numsilcr, &silcurves, &stat);
    . . .
}
```

8.4.2 Find the topology of the silhouette curves of a spline surface, using perspective projection.

NAME

s1510 - Find the silhouette curves and points of a surface when the surface is viewed perspectively from a specific eye point. In addition to the points and curves found by this routine, break curves and edge-curves might be silhouette curves. To march out the silhouette curves, use s1514() on page 252.

SYNOPSIS

void s1510(*ps*, *eyepoint*, *idim*, *aepsco*, *aepsge*, *jpt*, *gpar*, *jcrv*, *wcurve*, *jstat*)

SISLSurf	**ps*;
double	*eyepoint*[ ];
int	*idim*;
double	*aepsco*;
double	*aepsge*;
int	**jpt*;
double	***gpar*;
int	**jcrv*;
SISLIntcurve	****wcurve*;
int	**jstat*;

ARGUMENTS

Input Arguments:

ps	-	Pointer to the surface.
eyepoint	-	The eye point vector.
idim	-	Dimension of the space in which eyepoint lies.
aepsco	-	Computational resolution (not used).
aepsge	-	Geometry resolution.

Output Arguments:

jpt	-	Number of single silhouette points.
gpar	-	Array containing the parameter values of the single silhouette points in the parameter plane of the surface. The points lie continuous. Silhouette curves are stored in wcurve.
jcrv	-	Number of silhouette curves.
wcurve	-	Array containing descriptions of the silhouette curves. The curves are only described by points in the parameter plane. The curve-pointers points to nothing.
jstat	-	Status messages > 0 : warning $= 0$: ok < 0 : error

EXAMPLE OF USE

```
{
    SISLSurf      *ps;   /* Must be defined */
    double        eyepoint[3];   /* Must be defined */
    int           idim = 3;
    double        aepsco = 1.0e-9; /* Not used */
    double        aepsge = 1.0e-6;
    int           jpt = 0;
    double        *gpar = NULL;
    int           jcrv = 0;
    SISLIntcurve **wcurve = NULL;
    int           jstat = 0;
    ...
    s1510(ps, eyepoint, idim, aepsco, aepsge, &jpt, &gpar, &jcrv, &wcurve, &js-
        tat);
    ...
}
```

8.4.3 Find the topology of the circular silhouette curves of a spline surface.

NAME

s1511 - Find the circular silhouette curves and points of a surface. In addition to the points and curves found by this routine, break curves and edge-curves might be silhouette curves. To march out the silhouette curves use s1515() on page 254.

SYNOPSIS

void s1511(*ps*, *qpoint*, *bvec*, *idim*, *aepsco*, *aepsge*, *jpt*, *gpar*, *jcrv*, *wcurve*, *jstat*)

SISLSurf	**ps*;
double	*qpoint*[ ];
double	*bvec*[ ];
int	*idim*;
double	*aepsco*;
double	*aepsge*;
int	**jpt*;
double	***gpar*;
int	**jcrv*;
SISLIntcurve	****wcurve*;
int	**jstat*;

ARGUMENTS

Input Arguments:

ps	-	Pointer to the surface.
qpoint	-	A point on the spin axis.
bvec	-	The circular silhouette axis direction.
idim	-	Dimension of the space in which axis lies.
aepsco	-	Computational resolution (not used).
aepsge	-	Geometry resolution.

Output Arguments:

jpt	-	Number of single silhouette points.
gpar	-	Array containing the parameter values of the single silhouette points in the parameter plane of the surface. The points lie continuous. Silhouette curves are stored in wcurve.
jcrv	-	Number of silhouette curves.
wcurve	-	Array containing descriptions of the silhouette curves. The curves are only described by points in the parameter plane. The curve-pointers points to nothing.
jstat	-	Status messages > 0 : warning $= 0$: ok < 0 : error

EXAMPLE OF USE

{

SISLSurf **ps*; /* Must be defined */

```
        double        qpoint[3];  /* Must be defined */
        double        bvec[3];  /* Must be defined */
        int           idim = 3;
        double        aepsco =1.0e-9; /* Not used */
        double        aepsge = 1.0e-6;
        int           jpt = 0;
        double        *gpar = NULL;
        int           jcrv = 0;
        SISLIntcurve  **wcurve = NULL;
        int           jstat = 0;
        ...
        s1511(ps, qpoint, bvec, idim, aepsco, aepsge, &jpt, &gpar, &jcrv, &wcurve,
              &jstat);
        ...
}
```

8.5 Marching

8.5.1 March an intersection curve between a spline surface and a plane.

NAME

s1314 - To march an intersection curve described by parameter pairs in an intersection curve object, a surface and a plane. The guide points are expected to be found by s1851(), described on page 213. The generated geometric curves are represented as B-spline curves.

SYNOPSIS

void s1314(*surf*, *point*, *normal*, *dim*, *epsco*, *epsge*, *maxstep*, *intcurve*, *makecurv*, *graphic*, *stat*)

SISLSurf	**surf*;
double	*point*[ ];
double	*normal*[ ];
int	*dim*;
double	*epsco*;
double	*epsge*;
double	*maxstep*;
SISLIntcurve	**intcurve*;
int	*makecurv*;
int	*graphic*;
int	**stat*;

ARGUMENTS

Input Arguments:

surf	-	Pointer to the surface.
point	-	Point in the plane.
normal	-	Normal to the plane.
dim	-	Dimension of the space in which the plane lies. Should be 3.
epsco	-	Computational resolution (not used).
epsge	-	Geometry resolution.
maxstep	-	Maximum step length allowed. If maxstep $\leq$ epsge maxstep is neglected. maxstep = 0.0 is recommended.
makecurv	-	Indicator telling if a geometric curve is to be made: 0 - Do not make curves at all. 1 - Make only one geometric curve. 2 - Make geometric curve and curve in the parameter plane.
graphic	-	Indicator telling if the function should draw the curve: 0 - Don't draw the curve. 1 - Draw the geometric curve. This option is outdated, if used see NOTE!

Input/Output Arguments:

intcurve - Pointer to the intersection curve. As input, only guide points (points in parameter space) exist. These guide points are used to guide the marching. The routine adds intersection curve and curve in the parameter plane to the SISLIntcurve object, according to the value of makecurv.

Output Arguments:

stat - Status messages
$= 3$: Iteration stopped due to singular point or degenerate surface. A part of an intersection curve may have been traced out. If no curve is traced out the curve pointers in the SISLIntcurve object point to NULL.
$= 0$: ok
< 0 : error

NOTE

If the draw option is used the empty dummy functions s6move() and s6line() are called. Thus if the draw option is used, make sure you have versions of functions s6move() and s6line() interfaced to your graphic package.

EXAMPLE OF USE

```
{
    SISLSurf      *surf;  /* Must be defined */
    double        point[3];  /* Must be defined */
    double        normal[3]; /* Must be defined */
    int           dim = 3;
    double        epsco = 1.0e-9; /* Not used */
    double        epsge = 1.0e-5;
    double        maxstep = 0.0;
    SISLIntcurve  *intcurve; /* The intersection curve instance is defined in s1851 */
    int           makecurv = 2;
    int           graphic = 0;
    int           stat = 0;
    ...
    s1314(surf, point, normal, dim, epsco, epsge, maxstep, intcurve,
          makecurv, graphic, &stat);
    ...
}
```

8.5.2 March an intersection curve between a spline surface and a sphere.

NAME

s1315 - To march an intersection curve described by parameter pairs in an intersection curve object, a surface and a sphere. The guide points are expected to be found by s1852(), described on page 215. The generated geometric curves are represented as B-spline curves.

SYNOPSIS

void s1315(*surf*, *centre*, *radius*, *dim*, *epsco*, *epsge*, *maxstep*, *intcurve*, makecurv, graphic, stat)

SISLSurf	**surf*;
double	*centre*[ ];
double	*radius*;
int	*dim*;
double	*epsco*;
double	*epsge*;
double	*maxstep*;
SISLIntcurve	**intcurve*;
int	*makecurv*;
int	*graphic*;
int	**stat*;

ARGUMENTS

Input Arguments:

surf	-	Pointer to the surface.
centre	-	Center of the sphere.
radius	-	Radius of sphere
dim	-	Dimension of the space in which the sphere lies. Should be 3.
epsco	-	Computational resolution (not used).
epsge	-	Geometry resolution.
maxstep	-	Maximum step length allowed. If maxstep $\leq$ epsge maxstep is neglected. maxstep = 0.0 is recommended.
makecurv	-	Indicator specifying if a geometric curve is to be made: 0 - Do not make curves at all. 1 - Make only a geometric curve. 2 - Make geometric curve and curve in parameter plane.
graphic	-	Indicator specifying if the function should draw the curve: 0 - Don't draw the curve. 1 - Draw the geometric curve. This option is outdated, if used see NOTE!

Input/Output Arguments:

intcurve - Pointer to the intersection curve. As input only guide points (points in parameter space) exist. These guide points are used to guide the marching. The routine adds intersection curve and curve in the parameter plane to the SISLIntcurve object according to the value of makecurv.

Output Arguments:

stat - Status messages
- $= 3$: Iteration stopped due to singular point or degenerate surface. A part of an intersection curve may have been traced out. If no curve is traced out, the curve pointers in the SISLIntcurve object point to NULL.
- $= 0$: ok
- < 0 : error

NOTE

If the draw option is used the empty dummy functions s6move() and s6line() are called. Thus if the draw option is used, make sure you have versions of functions s6move() and s6line() interfaced to your graphic package.

EXAMPLE OF USE

```
{
    SISLSurf      *surf;  /* Must be defined */
    double        centre[3];  /* Must be defined */
    double        radius;  /* Must be defined */
    int           dim = 3;
    double        epsco = 1.0e-9; /* Not used */
    double        epsge = 1.0e-5;
    double        maxstep = 0;
    SISLIntcurve  *intcurve; /* The intersection curve instance is defined in s1852 */
    int           makecurv = 2;
    int           graphic = 0;
    int           stat = 0;
    ...
    s1315(surf, centre, radius, dim, epsco, epsge, maxstep, intcurve, makecurv,
          graphic, &stat);
    ...
}
```

8.5.3 March an intersection curve between a spline surface and a cylinder.

NAME

s1316 - To march an intersection curve described by parameter pairs in an intersection curve object, a surface and a cylinder. The guide points are expected to be found by s1853() described on page 217. The generated geometric curves are represented as B-spline curves.

SYNOPSIS

void s1316(*surf*, *point*, *cyldir*, *radius*, *dim*, *epsco*, *epsge*, *maxstep*, *intcurve*, *makecurv*, *graphic*, *stat*)

```
SISLSurf      *surf;
double        point[ ];
double        cyldir[ ];
double        radius;
int           dim;
double        epsco;
double        epsge;
double        maxstep;
SISLIntcurve  *intcurve;
int           makecurv;
int           graphic;
int           *stat;
```

ARGUMENTS

Input Arguments:

surf	-	Pointer to the surface.
point	-	Point on the axis of the cylinder.
cyldir	-	The direction vector of the axis of the cylinder.
radius	-	Radius of the cylinder.
dim	-	Dimension of the space in which the cylinder lies. Should be 3.
epsco	-	Computational resolution (not used).
epsge	-	Geometry resolution.
maxstep	-	Maximum step length allowed. If maxstep $\leq$ epsge maxstep is neglected. maxstep = 0.0 is recommended.
makecurv	-	Indicator specifying if a geometric curve is to be made: 0 - Do not make curves at all. 1 - Make only a geometric curve. 2 - Make geometric curve and curve in the parameter plane.
graphic	-	Indicator specifying if the function should draw the curve: 0 - Don't draw the curve. 1 - Draw the geometric curve. This option is outdated, if used see NOTE!

Input/Output Arguments:

intcurve - Pointer to the intersection curve. As input only guide points (points in parameter space) exist. These guide points are used to guide the marching. The routine adds intersection curve and curve in the parameter plane to the SISLIntcurve object according to the value of makecurv.

Output Arguments:

stat - Status messages
- = 3 : Iteration stopped due to singular point or degenerate surface. A part of an intersection curve may have been traced out. If no curve is traced out, the curve pointers in the SISLIntcurve object point to NULL.
- = 0 : ok
- < 0 : error

NOTE

If the draw option is used the empty dummy functions s6move() and s6line() are called. Thus if the draw option is used, make sure you have versions of functions s6move() and s6line() interfaced to your graphic package.

EXAMPLE OF USE

```
{
    SISLSurf      *surf;  /* Must be defined */
    double        point[3];  /* Must be defined */
    double        cyldir[3];  /* Must be defined */
    double        radius;  /* Must be defined */
    int           dim = 3;
    double        epsco = 1.0e-9; /* Not used */
    double        epsge = 1.0e-5;
    double        maxstep = 0.0;
    SISLIntcurve  *intcurve; /* The intersection curve instance is defined in s1853 */
    int           makecurv;
    int           graphic;
    int           stat = 0;
    . . .
    s1316(surf, point, cyldir, radius, dim, epsco, epsge, maxstep, intcurve, make-
        curv, graphic, &stat);
    . . .
}
```

8.5.4 March an intersection curve between a spline surface and a cone.

NAME

s1317 - To march an intersection curve described by parameter pairs in an intersection curve object, a surface and a cone. The guide points are expected to be found by s1854() described on page 219. The generated geometric curves are represented as B-spline curves.

SYNOPSIS

void s1317(*surf*, *toppt*, *axispt*, *conept*, *dim*, *epsco*, *epsge*, *maxstep*, *intcurve*, *makecurv*, *graphic*, *stat*)

SISLSurf	**surf*;
double	*toppt*[ ];
double	*axispt*[ ];
double	*conept*[ ];
int	*dim*;
double	*epsco*;
double	*epsge*;
double	*maxstep*;
SISLIntcurve	**intcurve*;
int	*makecurv*;
int	*graphic*;
int	**stat*;

ARGUMENTS

Input Arguments:

surf	-	Pointer to the surface.
toppt	-	The top point of the cone.
axispt	-	Point on the axis of the cone; axispt must be different from toppt.
conept	-	A point on the cone surface that is not the top point.
dim	-	Dimension of the space in which the cone lies. Should be 3.
epsco	-	Computational resolution (not used).
epsge	-	Geometry resolution.
maxstep	-	Maximum step length allowed. If maxstep $\leq$ epsge, maxstep is neglected. maxstep = 0.0 is recommended.
makecurv	-	Indicator specifying if a geometric curve is to be made: 0 - Do not make curves at all. 1 - Make only a geometric curve. 2 - Make geometric curve and curve in the parameter plane
graphic	-	Indicator specifying if the function should draw the curve: 0 - Don't draw the curve. 1 - Draw the geometric curve. This option is outdated, if used see NOTE!

Input/Output Arguments:

intcurve - Pointer to the intersection curve. As input only guide points (points in parameter space) exist. These guide points are used for guiding the marching. The routine adds the intersection curve and curve in the parameter plane to the SISLIntcurve object according to the value of makecurv.

Output Arguments:

stat - Status messages

= 3 : Iteration stopped due to singular point or degenerate surface. A part of an intersection curve may have been traced out. If no curve is traced out, the curve pointers in the SISLIntcurve object point to NULL.

= 0 : ok

< 0 : error

NOTE

If the draw option is used the empty dummy functions s6move() and s6line() are called. Thus if the draw option is used, make sure you have versions of functions s6move() and s6line() interfaced to your graphic package.

EXAMPLE OF USE

```
{
    SISLSurf      *surf;  /* Must be defined */
    double        toppt[3];  /* Must be defined */
    double        axispt[3];  /* Must be defined */
    double        conept[3];  /* Must be defined */
    int           dim = 3;
    double        epsco = 1.0e-9; /* Not used */
    double        epsge = 1.0e-5;
    double        maxstep = 0.0;
    SISLIntcurve *intcurve; /* The intersection curve instance is defined in s1854 */
    int           makecurv = 2;
    int           graphic = 0;
    int           stat = 0;
    ...
    s1317(surf, toppt, axispt, conept, dim, epsco, epsge, maxstep, intcurve,
          makecurv, graphic, &stat);
    ...
}
```

8.5.5 March an intersection curve between a surface and an elliptic cone.

NAME

s1501 - To march an intersection curve described by parameter pairs in an intersection curve object, a surface and an elliptic cone. The guide points are expected to be found by s1503() described on page 221. The generated geometric curves are represented as B-spline curves.

SYNOPSIS

void s1501(*surf*, *basept*, *normdir*, *ellipaxis*, *alpha*, *ratio*, *dim*, *epsco*, *epsge*, *maxstep*, *intcurve*, *makecurv*, *graphic*, *stat*)

SISLSurf	**surf*;
double	*basept*[ ];
double	*normdir*[ ];
double	*ellipaxis*[ ];
double	*alpha*;
double	*ratio*;
int	*dim*;
double	*epsco*;
double	*epsge*;
double	*maxstep*;
SISLIntcurve	**intcurve*;
int	*makecurv*;
int	*graphic*;
int	**stat*;

ARGUMENTS

Input Arguments:

surf	-	Pointer to the surface.
basept	-	Base point of the cone, centre of elliptic base.
normdir	-	Direction of the cone axis, normal to the elliptic base. The default is pointing from the base point to the top point.
ellipaxis	-	One of the axes of the ellipse (major or minor). The other axis will be calculated as $normdir \times ellipaxis$ scaled with *ratio*.
alpha	-	The opening angle in radians of the cone at the ellipaxis.
ratio	-	The ratio of the major and minor axes = ellipaxis/otheraxis.
dim	-	Dimension of the space in which the cone lies. Should be 3.
epsco	-	Computational resolution (not used).
epsge	-	Geometry resolution.
maxstep	-	Maximum step length allowed. If maxstep $\leq$ epsge, maxstep is neglected. maxstep = 0.0 is recommended.
makecurv	-	Indicator specifying if a geometric curve is to be made: 0 - Do not make curves at all. 1 - Make only a geometric curve. 2 - Make geometric curve and curve in the parameter plane

graphic - Indicator specifying if the function should draw the curve:
- 0 - Don't draw the curve.
- 1 - Draw the geometric curve. This option is outdated, if used see NOTE!

Input/Output Arguments:

intcurve - Pointer to the intersection curve. As input only guide points (points in parameter space) exist. These guide points are used for guiding the marching. The routine adds the intersection curve and curve in the parameter plane to the SISLIntcurve object according to the value of makecurv.

Output Arguments:

stat - Status messages
- $= 3$: Iteration stopped due to singular point or degenerate surface. A part of an intersection curve may have been traced out. If no curve is traced out, the curve pointers in the SISLIntcurve object point to NULL.
- $= 0$: ok
- < 0 : error

NOTE

If the draw option is used the empty dummy functions s6move() and s6line() are called. Thus if the draw option is used, make sure you have versions of functions s6move() and s6line() interfaced to your graphic package.

EXAMPLE OF USE

```
{
    SISLSurf     *surf;  /* Must be defined */
    double       basept[3];  /* Must be defined */
    double       normdir[3];  /* Must be defined */
    double       ellipaxis[3];  /* Must be defined */
    double       alpha;  /* Must be defined */
    double       ratio;  /* Must be defined */
    int          dim = 3;
    double       epsco = 1.0e-9; /* Not used */
    double       epsge = 1.0e-6;
    double       maxstep = 0.0;
    SISLIntcurve *intcurve; /* The intersection curve instance is defined in s1853 */
    int          makecurv = 2;
    int          graphic = 0;
    int          stat = 0;
    ...
    s1501(surf, basept, normdir, ellipaxis, alpha, ratio, dim, epsco, epsge,
          maxstep, intcurve, makecurv, graphic, &stat);
    ...
}
```

8.5.6 March an intersection curve between a spline surface and a torus.

NAME

s1318 - To march an intersection curve described by parameter pairs in an intersection curve object, a surface and a torus. The guide points are expected to be found by s1369(), described on page 223. The generated geometric curves are represented as B-spline curves.

SYNOPSIS

void s1318(*surf*, *centre*, *normal*, *cendist*, *radius*, *dim*, *epsco*, *epsge*, *maxstep*, *intcurve*, *makecurv*, *graphic*, *stat*)

SISLSurf	**surf*;
double	*centre*[ ];
double	*normal*[ ];
double	*cendist*;
double	*radius*;
int	*dim*;
double	*epsco*;
double	*epsge*;
double	*maxstep*;
SISLIntcurve	**intcurve*;
int	*makecurv*;
int	*graphic*;
int	**stat*;

ARGUMENTS

Input Arguments:

surf	-	Pointer to the surface.
centre	-	The centre of the torus (lying in the symmetry plane)
normal	-	Normal to the symmetry plane.
cendist	-	Distance from centre to the centre circle of torus.
radius	-	The radius of the torus surface.
dim	-	Dimension of the space in which the torus lies. Should be 3.
epsco	-	Computational resolution (not used).
epsge	-	Geometry resolution.
maxstep	-	Maximum step length allowed. If maxstep $\leq$ epsge maxstep is neglected. maxstep = 0.0 is recommended.
makecurv	-	Indicator specifying if a geometric curve is to be made: 0 - Do not make curves at all. 1 - Make only a geometric curve. 2 - Make geometric curve and curve in the parameter plane

graphic - Indicator specifying if the function should draw the curve:
  - 0 - Don't draw the curve.
  - 1 - Draw the geometric curve. This option is outdated, if used see NOTE!

Input/Output Arguments:

intcurve - Pointer to the intersection curve. As input only guide points (points in parameter space) exist. These guide points are used for guiding the marching. The routine adds the intersection curve and curve in the parameter plane to the SISLIntcurve object according to the value of makecurv.

Output Arguments:

stat - Status messages
  - = 3 : Iteration stopped due to singular point or degenerate surface. A part of an intersection curve may have been traced out. If no curve is traced out the curve pointers in the SISLIntcurve object point to NULL.
  - = 0 : ok
  - < 0 : error

NOTE

If the draw option is used the empty dummy functions s6move() and s6line() are called. Thus if the draw option is used, make sure you have versions of functions s6move() and s6line() interfaced to your graphic package.

EXAMPLE OF USE

```
{
    SISLSurf      *surf;  /* Must be defined */
    double        centre[3];  /* Must be defined */
    double        normal[3];  /* Must be defined */
    double        cendist;  /* Must be defined */
    double        radius;  /* Must be defined */
    int           dim = 3;
    double        epsco = 1.0e-9; /* Not used */
    double        epsge = 1.0e-5;
    double        maxstep = 0.0;
    SISLIntcurve *intcurve; /* The intersection curve instance is defined in s1369 */
    int           makecurv = 2;
    int           graphic = 0;
    int           stat = 0;
    ...
    s1318(surf, centre, normal, cendist, radius, dim, epsco, epsge, maxstep,
          intcurve, makecurv, graphic, &stat);
    ...
}
```

8.5.7 March an intersection curve between two spline surfaces.

NAME

s1310 - To march an intersection curve between two surfaces. The intersection curve is described by guide parameter pairs stored in an intersection curve object. The guide points are expected to be found by s1859() described on page 225. The generated geometric curves are represented as B-spline curves.

SYNOPSIS

void s1310(*surf1*, *surf2*, *intcurve*, *epsge*, *maxstep*, *makecurv*, *graphic*, *stat*)

SISLSurf	**surf1*;
SISLSurf	**surf2*;
SISLIntcurve	**intcurve*;
double	*epsge*;
double	*maxstep*;
int	*makecurv*;
int	*graphic*;
int	**stat*;

ARGUMENTS

Input Arguments:

surf1 - Pointer to the first surface.

surf2 - Pointer to the second surface.

epsge - Geometry resolution.

maxstep - Maximum step length. If maxstep$\leq$0, maxstep is ignored. maxstep = 0.0 is recommended.

makecurv - Indicator specifying if a geometric curve is to be made:
- 0 - Do not make curves at all
- 1 - Make only a geometric curve.
- 2 - Make geometric curve and curves in the parameter planes

graphic - Indicator specifying if the function should draw the geometric curve:
- 0 - Don't draw the curve
- 1 - Draw the geometric curve. This option is outdated, if used see NOTE!

Input/Output Arguments:

intcurve - Pointer to the intersection curve. As input only guide points (points in parameter space) exist. These guide points are used for guiding the marching. The routine adds intersection curve and curves in the parameter planes to the SISLIntcurve object, according to the value of makecurv.

Output Arguments:

stat	-	Status messages
		= 3 : Iteration stopped due to singular point or degenerate surface. A part of an intersection curve may have been traced out. If no curve is traced out, the curve pointers in the SISLIntcurve object point to NULL.
		= 0 : ok
		< 0 : error

NOTE

If the draw option is used the empty dummy functions s6move() and s6line() are called. Thus if the draw option is used, make sure you have versions of functions s6move() and s6line() interfaced to your graphic package.

EXAMPLE OF USE

```
{
    SISLSurf      *surf1;   /* Must be defined */
    SISLSurf      *surf2;   /* Must be defined */
    SISLIntcurve  *intcurve; /* The intersection curve instance is defined in s1859 */
    double        epsge = 1.0e-5;
    double        maxstep = 0.0;
    int           makecurv = 2;
    int           graphic = 0;
    int           stat = 0;
    ...
    s1310(surf1, surf2, intcurve, epsge, maxstep, makecurv, graphic, &stat);
    ...
}
```

8.6 Marching of Silhouettes

8.6.1 March a silhouette curve of a surface, using parallel projection.

NAME

s1319 - To march the silhouette curve described by an intersection curve object, a surface and a view direction (i.e. parallel projection). The guide points are expected to be found by s1860(), described on page 227. The generated geometric curves are represented as B-spline curves.

NOTE

The silhouette curves are defined as curves on the surface where the inner product of the surface normal and the direction vector of the viewing is 0. This definition will include surface points where the normal is zero.

SYNOPSIS

void s1319(*surf*, *viewdir*, *dim*, *epsco*, *epsge*, *maxstep*, *intcurve*, *makecurv*, *graphic*, *stat*)

SISLSurf	**surf*;
double	*viewdir*[ ];
int	*dim*;
double	*epsco*;
double	*epsge*;
double	*maxstep*;
SISLIntcurve	**intcurve*;
int	*makecurv*;
int	*graphic*;
int	**stat*;

ARGUMENTS

Input Arguments:

surf	-	Pointer to the surface.
viewdir	-	View direction.
dim	-	Dimension of the space in which vector describing the view direction lies. Should be 3.
epsco	-	Computational resolution (not used).
epsge	-	Geometry resolution.
maxstep	-	Maximum step length allowed. If maxstep $\leq$ epsge maxstep is neglected. maxstep = 0.0 is recommended.

makecurv - Indicator specifying if a geometric curve is to be made:
- 0 - Do not make curves at all.
- 1 - Make only a geometric curve.
- 2 - Make geometric curve and curve in the parameter plane.

graphic - Indicator specifying if the function should draw the geometric curve:
- 0 - Don't draw the curve.
- 1 - Draw the geometric curve. This option is outdated, if used see NOTE!

Input/Output Arguments:

intcurve - Pointer to the intersection curve. As input, only guide points (points in parameter space) exist. These guide points are used for guiding the marching. The routine adds intersection curve and curve in the parameter plane to the SISLIntcurve object according to the value of makecurv.

Output Arguments:

stat - Status messages
- $= 3$: Iteration stopped due to singular point or degenerate surface. A part of an intersection curve may have been traced out. If no curve is traced out the curve pointers in the SISLIntcurve object point to NULL.
- $= 0$: ok
- < 0 : error

NOTE

If the draw option is used the empty dummy functions s6move() and s6line() are called. Thus if the draw option is used, make sure you have versions of functions s6move() and s6line() interfaced to your graphic package.

EXAMPLE OF USE

```
{
    SISLSurf      *surf;  /* Must be defined */
    double        viewdir[3];  /* Must be defined */
    int           dim = 3;
    double        epsco = 1.0e-9; /* Not used */
    double        epsge = 1.0e-5;
    double        maxstep = 0.0;
    SISLIntcurve  *intcurve; /* The silhouette curve instance is defined in s1860 */
    int           makecurv = 2;
    int           graphic = 0;
    int           stat = 0;
    . . .
    s1319(surf, viewdir, dim, epsco, epsge, maxstep, intcurve, makecurv,
          graphic, &stat);
    . . .
}
```

8.6.2 March a silhouette curve of a surface, using perspective projection.

NAME

s1514 - To march the perspective silhouette curve described by an intersection curve object, a surface and an eye point. The guide points are expected to be found by s1510() described on page 229. The generated geometric curves are represented as B-spline curves.

SYNOPSIS

void s1514(*ps1*, *eyepoint*, *idim*, *aepsco*, *aepsge*, *amax*, *pintcr*, *icur*, *igraph*, *jstat*)

SISLSurf	**ps1*;
double	*eyepoint*[ ]
int	*idim*;
double	*aepsco*;
double	*aepsge*;
double	*amax*;
SISLIntcurve	**pintcr*;
int	*icur*;
int	*igraph*;
int	**jstat*;

ARGUMENTS

Input Arguments:

ps1	-	Pointer to surface.
eyepoint	-	Eye point for perspective view
idim	-	Dimension of the space in which the *eyepoint* lies.
aepsco	-	Computational resolution (not used).
aepsge	-	Geometry resolution.
amax	-	Maximal allowed step length. If $amax \leq aepsge$ *amax* is neglected.
icur	-	Indicator telling if a 3D curve is to be made.
		$= 0$: Don't make 3D curve.
		$= 1$: Make 3D curve.
		$= 2$: Make 3D curve and curves in the parameter plane.
igraph	-	Indicator telling if the curve is to be output through function calls:
		$= 0$: Don't output curve through function call.
		$= 0$: Output as straight line segments. This option is outdated, if used see NOTE!

Input/Output Arguments:

pintcr - The intersection curve. When coming in as input only parameter values in the parameter plane exist. When coming as output the 3D geometry and possibly the curve in the parameter plane of the surface is added.

Output Arguments:

jstat - Status messages

$= 3$: Iteration stopped due to singular point or degenerate surface. A part of intersection curve may have been traced out. If no curve is traced out the curve pointers in the Intcurve object point to NULL.

> 0 : Warning.

$= 0$: Ok.

< 0 : Error.

$= -185$: No points produced on intersection curve.

NOTE

If the draw option is used the empty dummy functions s6move() and s6line() are called. Thus if the draw option is used, make sure you have versions of functions s6move() and s6line() interfaced to your graphic package.

EXAMPLE OF USE

```
{
    SISLSurf      *ps1;   /* Must be defined */
    double        eyepoint[3];   /* Must be defined */
    int           idim = 3;
    double        aepsco = 1.0e-9; /* Not used */
    double        aepsge = 1.0e-5;
    double        amax = 0.0;
    SISLIntcurve  *pintcr; /* The silhouette curve instance is defined in s1510 */
    int           icur;
    int           igraph;
    int           jstat = 0;
    ...
    s1514(ps1, eyepoint, idim, aepsco, aepsge, amax, pintcr, icur, igraph, &jstat);
    ...
}
```

8.6.3 March a circular silhouette curve of a surface.

NAME

s1515 - To march the circular silhouette curve described by an intersection curve object, a surface, point Q and direction B i.e. solution of $f(u, v) = N(u, v) \times (P(u, v) - Q) \cdot B$.
The guide points are expected to be found by s1511() described on page 231. The generated geometric curves are represented as B-spline curves.

SYNOPSIS

void s1515(*ps1*, *qpoint*, *bvec*, *idim*, *aepsco*, *aepsge*, *amax*, *pintcr*, *icur*, *igraph*, *jstat*)

SISLSurf	**ps1*;
double	*qpoint*[ ];
double	*bvec*[ ];
int	*idim*;
double	*aepsco*;
double	*aepsge*;
double	*amax*;
SISLIntcurve	**pintcr*;
int	*icur*;
int	*igraph*;
int	**jstat*;

ARGUMENTS

Input Arguments:

ps1	-	Pointer to surface.
qpoint	-	Point Q for circular silhouette.
bvec	-	Direction B for circular silhouette.
idim	-	Dimension of the space in which Q lies.
aepsco	-	Computational resolution (not used).
aepsge	-	Geometry resolution.
amax	-	Maximal allowed step length. If $amax \leq aepsge$ *amax* is neglected.
icur	-	Indicator telling if a 3D curve is to be made. = 0 : Don't make 3D curve. = 1 : Make 3D curve. = 2 : Make 3D curve and curves in the parameter plane.
igraph	-	Indicator telling if the curve is to be output through function calls: = 0 : Don't output curve through function call. = 0 : Output as straight line segments . This option is outdated, if used see NOTE!

Input/Output Arguments:

pintcr	-	The intersection curve. When coming in as input only parameter values in the parameter plane exist. When coming as output the 3-D geometry and possibly the curve in the parameter plane of the surface is added.

Output Arguments:

jstat	-	Status messages	
		$= 3$	: Iteration stopped due to singular point or degenerate surface. A part of intersection curve may have been traced out. If no curve is traced out the curve pointers in the Intcurve object point to NULL.
		> 0	: Warning.
		$= 0$	: Ok.
		< 0	: Error.
		$= -185$	: No points produced on intersection curve.

NOTE

If the draw option is used the empty dummy functions s6move() and s6line() are called. Thus if the draw option is used, make sure you have versions of functions s6move() and s6line() interfaced to your graphic package.

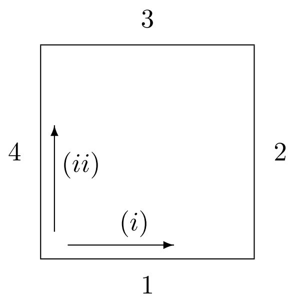
EXAMPLE OF USE

```
{
    SISLSurf      *ps1;  /* Must be defined */
    double        qpoint[3];  /* Must be defined */
    double        bvec[3];  /* Must be defined */
    int           idim = 3;
    double        aepsco = 1.0e-9; /* Not used */
    double        aepsge = 1.0e-6;
    double        amax = 0.0;
    SISLIntcurve *pintcr; /* The silhouette curve instance is defined in s1511 */
    int           icur = 2;
    int           igraph = 0;
    int           jstat = 0;
    ...
s1515(ps1, qpoint, bvec, idim, aepsco, aepsge, amax, pintcr, icur, igraph,
    &jstat);
    ...
}
```

8.7 Check if a Surface is Closed or has Degenerate Edges.

NAME

s1450 - To check if a surface is closed or has degenerate boundaries. The edge numbers correspond to the following:



(i) first parameter direction of surface.
(ii) second parameter direction of surface.

SYNOPSIS

void s1450(*surf*, *epsge*, *close1*, *close2*, *degen1*, *degen2*, *degen3*, *degen4*, *stat*)

SISLSurf	**surf*;
double	*epsge*;
int	**close1*;
int	**close2*;
int	**degen1*;
int	**degen2*;
int	**degen3*;
int	**degen4*;
int	**stat*;

ARGUMENTS

Input Arguments:

surf	-	Pointer to the surface that is to be checked.
epsge	-	Tolerance used during testing.

Output Arguments:

close1	-	Closed indicator in the first parameter direction. = 0 : Surface open in first direction = 1 : Surface closed in first direction
close2	-	Closed indicator in second direction = 0 : Surface open in second direction = 1 : Surface closed in second direction
degen1	-	Degenerate indicator along standard edge 1 = 0 : Edge is not degenerate = 1 : Edge is degenerate
degen2	-	Degenerate indicator along standard edge 2 = 0 : Edge is not degenerate = 1 : Edge is degenerate
degen3	-	Degenerate indicator along standard edge 3 = 0 : Edge is not degenerate = 1 : Edge is degenerate
degen4	-	Degenerate indicator along standard edge 4 = 0 : Edge is not degenerate = 1 : Edge is degenerate
stat	-	Status messages > 0 : warning = 0 : ok < 0 : error

EXAMPLE OF USE

```
{
    SISLSurf    *surf;  /* Must be defined */
    double      epsge = 0.000001;
    int         close1 = 0;
    int         close2 = 0;
    int         degen1 = 0;
    int         degen2 = 0;
    int         degen3 = 0;
    int         degen4 = 0;
    int         stat = 0;
    . . .
    s1450(surf, epsge, &close1, &close2, &degen1, &degen2, &degen3, &degen4,
        &stat);
    . . .
}
```

8.8 Pick the Parameter Ranges of a Surface

NAME

s1603 - To pick the parameter ranges of a surface.

SYNOPSIS

void s1603(*surf*, *min1*, *min2*, *max1*, *max2*, *stat*)

```
SISLSurf    *surf;
double      *min1;
double      *min2;
double      *max1;
double      *max2;
int         *stat;
```

ARGUMENTS

Input Arguments:

surf	-	The surface.

Output Arguments:

min1	-	Start parameter in the first parameter direction.
min2	-	Start parameter in the second parameter direction.
max1	-	End parameter in the first parameter direction.
max2	-	End parameter in the second parameter direction.
stat	-	Status messages > 0 : warning $= 0$: ok < 0 : error

EXAMPLE OF USE

```
{
    SISLSurf    *surf;  /* Must be defined */
    double      min1;
    double      min2;
    double      max1;
    double      max2;
    int         stat = 0;
    ...
    s1603(surf, &min1, &min2, &max1, &max2, &stat);
    ...
}
```

8.9 Closest Points

8.9.1 Find the closest point between a surface and a point.

NAME

s1954 - Find the points on a surface lying closest to a given point.

SYNOPSIS

void s1954(*surf*, *point*, *dim*, *epsco*, *epsge*, *numclopt*, *pointpar*, *numclocr*, *clocurves*, *stat*)

SISLSurf	**surf*;
double	*point*[ ];
int	*dim*;
double	*epsco*;
double	*epsge*;
int	**numclopt*;
double	***pointpar*;
int	**numclocr*;
SISLIntcurve	****clocurves*;
int	**stat*;

ARGUMENTS

Input Arguments:

surf	-	Pointer to the surface in the closest point problem.
point	-	The point in the closest point problem.
dim	-	Dimension of the space in which the point lies.
epsco	-	Computational resolution (not used).
epsge	-	Geometry resolution.

Output Arguments:

numclopt	-	Number of single closest points.
pointpar	-	Array containing the parameter values of the single closest points in the parameter area of the surface. The points lie in sequence. Closest curves are stored in clocurves.
numclocr	-	Number of closest curves.
clocurves	-	Array containing the description of the closest curves. The curves are only described by points in the parameter area. The curve pointers point to nothing.
stat	-	Status messages > 0 : warning $= 0$: ok < 0 : error

EXAMPLE OF USE

```
{
    SISLSurf       *surf  /* Must be defined */;
    double         point[3];  /* Must be defined */
    int            dim = 3;
    double         epsco = 1.0e-9; /* Not used */
    double         epsge = 1.0e-6;
    int            numclopt = 0;
    double         *pointpar = NULL;
    int            numclocr = 0;
    SISLIntcurve   **clocurves = NULL;
    int            stat = 0;
    ...
    s1954(surf, point, dim, epsco, epsge, &numclopt, &pointpar, &numclocr,
          &clocurves, &stat);
    ...
}
```

8.9.2 Find the closest point between a surface and a point. Simple version.

NAME

s1958 - Find the closest point between a surface and a point. The method is fast and should work well in clear cut cases, but there is no guarantee it will find the right solution. As long as it doesn't fail, it will find exactly one point. In other cases, use s1954() on page 259.

SYNOPSIS

void s1958(*psurf*, *epoint*, *idim*, *aepsco*, *aepsge*, *gpar*, *dist*, *jstat*)

SISLSurf	**psurf*;
double	*epoint*[ ];
int	*idim*;
double	*aepsco*;
double	*aepsge*;
double	*gpar*[ ];
double	**dist*;
int	**jstat*;

ARGUMENTS

Input Arguments:

psurf	-	Pointer to the surface in the closest point problem.
epoint	-	The point in the closest point problem.
idim	-	Dimension of the space in which epoint lies.
aepsco	-	Computational resolution (not used).
aepsge	-	Geometry resolution.

Output Arguments:

gpar	-	2D array containing the parameter values of the closest point in the parameter space of the surface.
dist	-	The closest distance between point and the surface.
jstat	-	Status messages > 2 : Warning. $= 2$: Solution at a corner. $= 1$: Solution at an edge. $= 0$: Solution in interior. < 0 : Error.

EXAMPLE OF USE

```
{
    SISLSurf    *psurf;   /* Must be defined */
    double      epoint[3];   /* Must be defined */
    int         idim = 3;
    double      aepsco = 1.0e-9; /* Not used */
    double      aepsge = 1.0e-6;
    double      gpar[2];
    double      dist = 0.0;
    int         jstat = 0;
    . . .
    s1958(psurf, epoint, idim, aepsco, aepsge, gpar, &dist, &jstat);
    . . .
}
```

8.9.3 Local iteration to closest point bewteen point and surface.

NAME

s1775 - Newton iteration on the distance function between a surface and a point, to find a closest point or an intersection point. If a bad choice for the guess parameters is given in, the iteration may end at a local, not global closest point.

SYNOPSIS

void s1775(*surf*, *point*, *dim*, *epsge*, *start*, *end*, *guess*, *clpar*, *stat*)

SISLSurf	**surf*;
double	*point*[ ];
int	*dim*;
double	*epsge*;
double	*start*[ ];
double	*end*[ ];
double	*guess*[ ];
double	*clpar*[ ];
int	**stat*;

ARGUMENTS

Input Arguments:

surf	-	The surface in the closest point problem.
point	-	The point in the closest point problem.
dim	-	Dimension of the geometry.
epsge	-	Geometry resolution.
start	-	Surface parameters giving the start of the search area (umin, vmin).
end	-	Surface parameters giving the end of the search area (umax, vmax).
guess	-	Surface guess parameters for the closest point iteration.

Output Arguments:

clpar	-	Resulting surface parameters from the iteration.
stat	-	Status messages > 0 : A minimum distance found. $= 0$: Intersection found. < 0 : Error.

EXAMPLE OF USE

{

SISLSurf	**surf*; /* Must be defined */
double	*point*[3]; /* Must be defined */
int	*dim* = 3;
double	*epsge* = 1.0e-6;
double	*start*[2]; /* Must be defined */
double	*end*[2]; /* Must be defined */
double	*guess*[2]; /* Must be defined */
double	*clpar*[2];

```
    int         stat = 0;
    ...
    s1775(surf, point, dim, epsge, start, end, guess, clpar, &stat);
    ...
}
```

8.10 Find the Absolute Extremals of a Surface.

NAME

s1921 - Find the absolute extremal points/curves of a surface along a given direction.

SYNOPSIS

void s1921(*ps1*, *edir*, *idim*, *aepsco*, *aepsge*, *jpt*, *gpar*, *jcrv*, *wcurve*, *jstat*)

SISLSurf	**ps1*;
double	*edir*[ ];
int	*idim*;
double	*aepsco*;
double	*aepsge*;
int	**jpt*;
double	***gpar*;
int	**jcrv*;
SISLIntcurve	****wcurve*;
int	**jstat*;

ARGUMENTS

Input Arguments:

ps1	-	Pointer to the surface.
edir	-	The direction in which the extremal point(s) and/or interval(s) are to be calculated. If $idim = 1$ a positive value indicates the maximum of the function and a negative value the minimum. If the dimension is greater that 1 the array contains the coordinates of the direction vector.
idim	-	Dimension of the space in which the vector *edir* lies.
aepsco	-	Computational resolution (not used).
aepsge	-	Geometry resolution.

Output Arguments:

jpt	-	Number of single extremal points.
gpar	-	Array containing the parameter values of the single extremal points in the parameter area of the surface. The points lie continuous. Extremal curves are stored in *wcurve*.
jcrv	-	Number of extremal curves.
wcurve	-	Array containing descriptions of the extremal curves. The curves are only described by points in the parameter area. The curve-pointers point to nothing.

jstat	-	Status messages > 0 : Warning. $= 0$: Ok. < 0 : Error.

EXAMPLE OF USE

```
{
    SISLSurf       *ps1;   /* Must be defined */
    double         edir[3];   /* Must be defined */
    int            idim = 3;
    double         aepsco = 1.0e-9; /* Not used */
    double         aepsge = 1.0e-6;
    int            jpt = 0;
    double         *gpar = NULL;
    int            jcrv = 0;
    SISLIntcurve **wcurve = NULL;
    int            jstat = 0;
    ...
    s1921(ps1, edir, idim, aepsco, aepsge, &jpt, &gpar, &jcrv, &wcurve, &jstat);
    ...
}
```

8.11 Bounding Box

Both curves and surfaces have bounding boxes. These are boxes surrounding an object not only parallel to the main axis, but also rotated 45 degrees around each main axis. These bounding boxes are used by the intersection functions to decide if an intersection is possible or not. They might also be used to find the position of objects under other circumstances. The bounding box object and corresponding initialization functionality ar described in Section 4.9.1 at pages 94 and 95.

8.11.1 Find the bounding box of a surface.

NAME

s1989 - Find the bounding box of a surface.
NOTE: The geometric bounding box is returned also in the rational case, that is the box in homogeneous coordinates is NOT computed.

SYNOPSIS

```
void s1989(ps, emax, emin, jstat)
      SISLSurf    *ps;
      double      **emax;
      double      **emin;
      int         *jstat;
```

ARGUMENTS

Input Arguments:

ps	-	Surface to treat.

Output Arguments:

emin	-	Array of dimension *idim* containing the minimum values of the bounding box, i.e. bottom-left corner of the box.
emax	-	Array of dimension *idim* containing the maximum values of the bounding box, i.e. upper-right corner of the box.
jstat	-	Status messages > 0 : Warning. $= 0$: Ok. < 0 : Error.

EXAMPLE OF USE

```
{
      SISLSurf    *ps;  /* Must be defined */
      double      *emax = NULL;
      double      *emin = NULL;
      int         jstat = 0;
      ...
      s1989(ps, &emax, &emin, &jstat);
      ...
}
```

8.12 Normal Cone

Both curves and surfaces have normal cones. These are the cones that are convex hull of all normalized tangents of a curve and all normalized normals of a surface.

These normal cones are used by the intersection functions to decide if only one intersection is possible. They might also be used to find directions of objects for other reasons. The direction cone object and corresponding initialization functionality ar described in Section 4.10.1 at pages 97 and 98.

8.12.1 Find the direction cone of a surface.

NAME

s1987 - Find the direction cone of a surface.

SYNOPSIS

void s1987(*ps*, *aepsge*, *jgtpi*, *gaxis*, *cang*, *jstat*)

SISLSurf	**ps*;
double	*aepsge*;
int	**jgtpi*;
double	***gaxis*;
double	**cang*;
int	**jstat*;

ARGUMENTS

Input Arguments:

ps	-	Surface to treat.
aepsge	-	Geometry tolerance.

Output Arguments:

jgtpi	-	To mark if the angle of the direction cone is greater than π. $= 0$: The direction cone of the surface is not greater than π in any parameter direction. $= 1$: The direction cone of the surface is greater than π in the first parameter direction. $= 2$: The direction cone of the surface is greater than π in the second parameter direction. $= 10$: The direction cone of a boundary curve of the surface is greater than π in the first parameter direction. $= 20$: The direction cone of a boundary curve of the surface is greater than π in the second parameter direction.
gaxis	-	Allocated array containing the coordinates of the centre of the cone. It is only computed if $jgtpi = 0$.
cang	-	The angle from the centre to the boundary of the cone. It is only computed if $jgtpi = 0$.
jstat	-	Status messages

> 0	:	Warning.
$= 0$	:	Ok.
< 0	:	Error.

EXAMPLE OF USE

```
{
    SISLSurf    *ps;  /* Must be defined */
    double      aepsge = 1.0e-10;
    int         jgtpi = 0;
    double      *gaxis = NULL;
    double      cang = 0.0;
    int         jstat = 0;
    ...
    s1987(ps, aepsge, &jgtpi, &gaxis, &cang, &jstat);
    ...
}
```

Chapter 9

Surface Analysis

This chapter describes the Surface Analysis part.

9.1 Curvature Evaluation

9.1.1 Gaussian curvature of a spline surface.

NAME

s2500 - To compute the Gaussian curvature K(u,v) of a spline surface at given values (u,v) = (parvalue[0],parvalue[1]), where et1[leftknot1] <= parvalue[0] < et1[leftknot1+1] and et2[leftknot2] <= parvalue[1] < et2[leftknot2+1]. See also s2501().

SYNOPSIS

void s2500(*surf*, *ider*, *iside1*, *iside2*, *parvalue*, *leftknot1*, *leftknot2*, *gaussian*, *jstat*)

SISLSurf	**surf*;
int	*ider*;
int	*iside1*;
int	*iside2*;
double	*parvalue*[ ];
int	**leftknot1*;
int	**leftknot2*;
double	*gaussian*[ ];
int	**jstat*;

ARGUMENTS

Input Arguments:

surf	-	Pointer to the surface to evaluate.
ider	-	Number of derivatives to calculate. Only implemented for ider=0. < 0 : No derivative calculated. $= 0$: Position calculated. $= 1$: Position and first derivative calculated, etc.
iside1	-	Flag indicating whether the derivatives in the first parameter direction are to be calculated from the left or from the right:

< 0 : calculate derivative from the left hand side.
>= 0 : calculate derivative from the right hand side.

iside2 - Flag indicating whether the derivatives in the second parameter direction are to be calculated from the left or from the right:
< 0 : calculate derivative from the left hand side.
>= 0 : calculate derivative from the right hand side.

parvalue - Parameter value at which to evaluate. Dimension of parvalue is 2.

Input/Output Arguments:

leftknot1 - Pointer to the interval in the knot vector in the first parameter direction where parvalue[0] is found, that is: et1[leftknot1] <= parvalue[0] < et1[leftknot1+1]. leftknot1 should be set equal to zero at the first call to the routine.

leftknot2 - Pointer to the interval in the knot vector in the second parameter direction where parvalue[1] is found, that is: et2[leftknot2] <= parvalue[1] < et2[leftknot2+1]. leftknot2 should be set equal to zero at the first call to the routine.

Output Arguments:

gaussian - Gaussian of the surface at (u,v) = (parvalue[0],parvalue[1]).

jstat - Status messages
= 2 : Surface is degenerate at the point, that is, the surface is not regular at this point.
= 1 : Surface is close to degenerate at the point. Angle between tangents is less than the angular tolerance.
= 0 : Ok.
< 0 : Error.

EXAMPLE OF USE

```
{
    SISLSurf    *surf;  /* Must be defined */
    int         ider = 0;
    int         iside1 = 1;
    int         iside2 = 1;
    double      parvalue[2]; /* Must be defined */
    int         leftknot1 = 0; /* Define initially as zero. For consequtive evaluations
                                  leave leftknot1 as returned from s1500 */
    int         leftknot2 = 0; /* As for leftknot1 */
    double      gaussian[1]; /* A pre allocated array is expected */
    int         jstat = 0;
    ...
    s2500(surf, ider, iside1, iside2, parvalue, &leftknot1, &leftknot2, gaussian,
          &jstat);
```

```
    ...
}
```

9.1.2 Mean curvature of a spline surface.

NAME

s2502 - To compute the mean curvature H(u,v) of a spline surface at given values (u,v) = (parvalue[0],parvalue[1]), where etl[leftknot1] <= parvalue[0] < etl[leftknot1+1] and et2[leftknot2] <= parvalue[1] < et2[leftknot2+1].

SYNOPSIS

void s2502(*surf*, *ider*, *iside1*, *iside2*, *parvalue*, *leftknot1*, *leftknot2*, *meancurvature*, *jstat*)

SISLSurf	**surf*;
int	*ider*;
int	*iside1*;
int	*iside2*;
double	*parvalue*[ ];
int	**leftknot1*;
int	**leftknot2*;
double	*meancurvature*[ ];
int	**jstat*;

ARGUMENTS

Input Arguments:

surf - Pointer to the surface to evaluate.

ider - Number of derivatives to calculate. Only implemented for ider=0.
< 0 : No derivative calculated.
$= 0$: Position calculated.
$= 1$: Position and first derivative calculated, etc.

iside1 - Flag indicating whether the derivatives in the first parameter direction are to be calculated from the left or from the right:
< 0 : calculate derivative from the left hand side.
$>= 0$: calculate derivative from the right hand side.

iside2 - Flag indicating whether the derivatives in the second parameter direction are to be calculated from the left or from the right:
< 0 : calculate derivative from the left hand side.
$>= 0$: calculate derivative from the right hand side.

parvalue - Parameter value at which to evaluate. Dimension of parvalue is 2.

Input/Output Arguments:

leftknot1 - Pointer to the interval in the knot vector in the first parameter direction where parvalue[0] is found, that is: et1[leftknot1] <= parvalue[0] < et1[leftknot1+1]. leftknot1 should be set equal to zero at the first call to the routine.

leftknot2 - Pointer to the interval in the knot vector in the second parameter direction where parvalue[1] is found, that is: et2[leftknot2] <= parvalue[1] < et2[leftknot2+1]. leftknot2 should be set equal to zero at the first call to the routine.

Output Arguments:

meancurvature - Mean curvature of the surface at (u,v) = (parvalue[0],parvalue[1]).

jstat - Status messages

= 2 :	Surface is degenerate at the point, that is, the surface is not regular at this point.
= 1 :	Surface is close to degenerate at the point. Angle between tangents is less than the angular tolerance.
= 0 :	Ok.
< 0 :	Error.

EXAMPLE OF USE

```
{
    SISLSurf    *surf;  /* Must be defined */
    int         ider = 0;
    int         iside1 = 1;
    int         iside2 = 1;
    double      parvalue[2];  /* Must be defined */
    int         leftknot1 = 0; /* Define initially as zero. For consequtive evaluations
                                  leave leftknot1 as returned from s1502 */
    int         leftknot2 = 0; /* As for leftknot1 */
    double      meancurvature[1]; /* A pre allocated array is expected */
    int         jstat = 0;
    ...
    s2502(surf, ider, iside1, iside2, parvalue, leftknot1, leftknot2, meancurvature,
          &jstat);
    ...
}
```

9.1.3 Absolute curvature of a spline surface.

NAME

s2504 - To compute the absolute curvature A(u,v) of a spline surface at given values (u,v) = (parvalue[0],parvalue[1]), where et1[leftknot1] <= parvalue[0] < et1[leftknot1+1] and et2[leftknot2] <= parvalue[1] < et2[leftknot2+1].

SYNOPSIS

void s2504(*surf*, *ider*, *iside1*, *iside2*, *parvalue*, *leftknot1*, *leftknot2*, *absCurvature*, *jstat*)

SISLSurf	**surf*;
int	*ider*;
int	*iside1*;
int	*iside2*;
double	*parvalue*[ ];
int	**leftknot1*;
int	**leftknot2*;
double	*absCurvature*[ ];
int	**jstat*;

ARGUMENTS

Input Arguments:

surf - Pointer to the surface to evaluate.

ider - Number of derivatives to calculate. Only implemented for ider=0.
- < 0 : No derivative calculated.
- = 0 : Position calculated.
- = 1 : Position and first derivative calculated, etc.

iside1 - Flag indicating whether the derivatives in the first parameter direction are to be calculated from the left or from the right:
- < 0 : calculate derivative from the left hand side.
- >= 0 : calculate derivative from the right hand side.

iside2 - Flag indicating whether the derivatives in the second parameter direction are to be calculated from the left or from the right:
- < 0 : calculate derivative from the left hand side.
- >= 0 : calculate derivative from the right hand side.

parvalue - Parameter value at which to evaluate. Dimension of parvalue is 2.

Input/Output Arguments:

leftknot1 - Pointer to the interval in the knot vector in the first parameter direction where parvalue[0] is found, that is: et1[leftknot1] <= parvalue[0] < et1[leftknot1+1]. leftknot1 should be set equal to zero at the first call to the routine.

leftknot2 - Pointer to the interval in the knot vector in the second parameter direction where parvalue[1] is found, that is: et2[leftknot2] <= parvalue[1] < et2[leftknot2+1]. leftknot2 should be set equal to zero at the first call to the routine.

Output Arguments:

absCurvature - Absolute curvature of the surface at (u,v) = (parvalue[0],parvalue[1]).

jstat - Status messages

= 2 :	Surface is degenerate at the point, that is, the surface is not regular at this point.
= 1 :	Surface is close to degenerate at the point. Angle between tangents is less than the angular tolerance.
= 0 :	Ok.
< 0 :	Error.

EXAMPLE OF USE

```
{
    SISLSurf    *surf;  /* Must be defined */
    int         ider = 0;
    int         iside1 = 1;
    int         iside2 = 1;
    double      parvalue[2];  /* Must be defined */
    int         leftknot1 = 0; /* Define initially as zero. For consequtive evaluations
                                  leave leftknot1 as returned from s1504 */
    int         leftknot2 = 0; /* As for leftknot1 */
    double      absCurvature[1]; /* A pre allocated array is expected */
    int         jstat = 0;
    ...
    s2504(surf, ider, iside1, iside2, parvalue, leftknot1, leftknot2, absCurvature,
          &jstat);
    ...
}
```

9.1.4 Total curvature of a spline surface.

NAME

s2506 - To compute the total curvature T(u,v) of a surface at given values (u,v) = (parvalue[0],parvalue[1]), where et1[leftknot1] <= parvalue[0] < et1[leftknot1+1] and et2[leftknot2] <= parvalue[1] < et2[leftknot2+1].

SYNOPSIS

void s2506(*surf*, *ider*, *iside1*, *iside2*, *parvalue*, *leftknot1*, *leftknot2*, *totalCurvature*, *jstat*)

SISLSurf	**surf*;
int	*ider*;
int	*iside1*;
int	*iside2*;
double	*parvalue*[ ];
int	**leftknot1*;
int	**leftknot2*;
double	*totalCurvature*[ ];
int	**jstat*;

ARGUMENTS

Input Arguments:

surf - Pointer to the surface to evaluate.

ider - Number of derivatives to calculate. Only implemented for ider=0.
< 0 : No derivative calculated.
= 0 : Position calculated.
= 1 : Position and first derivative calculated, etc.

iside1 - Flag indicating whether the derivatives in the first parameter direction are to be calculated from the left or from the right:
< 0 : calculate derivative from the left hand side.
>= 0 : calculate derivative from the right hand side.

iside2 - Flag indicating whether the derivatives in the second parameter direction are to be calculated from the left or from the right:
< 0 : calculate derivative from the left hand side.
>= 0 : calculate derivative from the right hand side.

parvalue - Parameter value at which to evaluate. Dimension of parvalue is 2.

Input/Output Arguments:

leftknot1 - Pointer to the interval in the knot vector in the first parameter direction where parvalue[0] is found, that is: et1[leftknot1] <= parvalue[0] < et1[leftknot1+1]. leftknot1 should be set equal to zero at the first call to the routine.

leftknot2 - Pointer to the interval in the knot vector in the second parameter direction where parvalue[1] is found, that is: et2[leftknot2] <= parvalue[1] < et2[leftknot2+1]. leftknot2 should be set equal to zero at the first call to the routine.

Output Arguments:

totalCurvature - Total curvature of the surface at (u,v) = (parvalue[0],parvalue[1]).

jstat - Status messages

= 2 : Surface is degenerate at the point, that is, the surface is not regular at this point.

= 1 : Surface is close to degenerate at the point. Angle between tangents is less than the angular tolerance.

= 0 : Ok.

< 0 : Error.

EXAMPLE OF USE

```
{
    SISLSurf    *surf;  /* Must be defined */
    int         ider = 0;
    int         iside1 = 1;
    int         iside2 = 1;
    double      parvalue[2]  /* Must be defined */;
    int         leftknot1 = 0; /* Define initially as zero. For consequtive evaluations
                                  leave leftknot1 as returned from s1506 */
    int         leftknot2 = 0; /* As for leftknot1 */
    double      totalCurvature[1]; /* A pre allocated array is expected */
    int         jstat = 0;
    ...
    s2506(surf, ider, iside1, iside2, parvalue, leftknot1, leftknot2, totalCurvature,
          &jstat);
    ...
}
```

9.1.5 Second order Mehlum curvature of a spline surface.

NAME

s2508 - To compute the second order Mehlum curvature M(u,v) of a surface at given values (u,v) = (parvalue[0],parvalue[1]), where et1[leftknot1] <= parvalue[0] < et1[leftknot1+1] and et2[leftknot2] <= parvalue[1] < et2[leftknot2+1]. See also s2509().

SYNOPSIS

void s2508(*surf*, *ider*, *iside1*, *iside2*, *parvalue*, *leftknot1*, *leftknot2*, *mehlum*, *jstat*)

SISLSurf	**surf*;
int	*ider*;
int	*iside1*;
int	*iside2*;
double	*parvalue*[ ];
int	**leftknot1*;
int	**leftknot2*;
double	*mehlum*[ ];
int	**jstat*;

ARGUMENTS

Input Arguments:

surf - Pointer to the surface to evaluate.

ider - Number of derivatives to calculate. Only implemented for ider=0.
- < 0 : No derivative calculated.
- = 0 : Position calculated.
- = 1 : Position and first derivative calculated, etc.

iside1 - Flag indicating whether the derivatives in the first parameter direction are to be calculated from the left or from the right:
- < 0 : calculate derivative from the left hand side.
- >= 0 : calculate derivative from the right hand side.

iside2 - Flag indicating whether the derivatives in the second parameter direction are to be calculated from the left or from the right:
- < 0 : calculate derivative from the left hand side.
- >= 0 : calculate derivative from the right hand side.

parvalue - Parameter value at which to evaluate. Dimension of parvalue is 2.

Input/Output Arguments:

leftknot1 - Pointer to the interval in the knot vector in the first parameter direction where parvalue[0] is found, that is: et1[leftknot1] <= parvalue[0] < et1[leftknot1+1]. leftknot1 should be set equal to zero at the first call to the routine.

leftknot2 - Pointer to the interval in the knot vector in the second parameter direction where parvalue[1] is found, that is: et2[leftknot2] <= parvalue[1] < et2[leftknot2+1]. leftknot2 should be set equal to zero at the first call to the routine.

Output Arguments:

mehlum - The second order Mehlum curvature of the surface at (u,v) = (parvalue[0],parvalue[1]).

jstat - Status messages
- = 2 : Surface is degenerate at the point, that is, the surface is not regular at this point.
- = 1 : Surface is close to degenerate at the point. Angle between tangents is less than the angular tolerance.
- = 0 : Ok.
- < 0 : Error.

EXAMPLE OF USE

```
{
    SISLSurf   *surf;  /* Must be defined */
    int        ider = 0;
    int        iside1 = 1;
    int        iside2 = 1;
    double     parvalue[2];  /* Must be defined */
    int        leftknot1 = 0; /* Define initially as zero. For consequtive evaluations
                                 leave leftknot1 as returned from s1506
    int        leftknot2 = 0; /* As for leftknot1 */
    double     mehlum[1]; /* A pre allocated array is expected
    int        jstat = 0;
    ...
    s2508(surf, ider, iside1, iside2, parvalue, leftknot1, leftknot2, mehlum, &jstat);
    ...
}
```

9.1.6 Third order Mehlum curvature of a spline surface.

NAME

s2510 - To compute the third order Mehlum curvature M(u,v) of a surface at given values (u,v) = (parvalue[0],parvalue[1]), where et1[leftknot1] <= parvalue[0] < et1[leftknot1+1], et2[leftknot2] <= parvalue[1] < et2[leftknot2+1].

SYNOPSIS

void s2510(*surf*, *ider*, *iside1*, *iside2*, *parvalue*, *leftknot1*, *leftknot2*, *mehlum*, *jstat*)

SISLSurf	**surf*;
int	*ider*;
int	*iside1*;
int	*iside2*;
double	*parvalue*[ ];
int	**leftknot1*;
int	**leftknot2*;
double	*mehlum*[ ];
int	**jstat*;

ARGUMENTS

Input Arguments:

surf	-	Pointer to the surface to evaluate.
ider	-	Number of derivatives to calculate. Only implemented for ider=0. < 0 : No derivative calculated. = 0 : Position calculated. = 1 : Position and first derivative calculated, etc.
iside1	-	Flag indicating whether the derivatives in the first parameter direction are to be calculated from the left or from the right: < 0 : calculate derivative from the left hand side. >= 0 : calculate derivative from the right hand side.
iside2	-	Flag indicating whether the derivatives in the second parameter direction are to be calculated from the left or from the right: < 0 : calculate derivative from the left hand side. >= 0 : calculate derivative from the right hand side.
parvalue	-	Parameter value at which to evaluate. Dimension of parvalue is 2.

Input/Output Arguments:

leftknot1	-	Pointer to the interval in the knot vector in the first parameter direction where parvalue[0] is found, that is: et1[leftknot1] <= parvalue[0] < et1[leftknot1+1]. leftknot1 should be set equal to zero at the first call to the routine.

leftknot2	-	Pointer to the interval in the knot vector in the second parameter direction where parvalue[1] is found, that is: et2[leftknot2] <= parvalue[1] < et2[leftknot2+1]. leftknot2 should be set equal to zero at the first call to the routine.

Output Arguments:

mehlum	-	Third order Mehlum curvature of the surface at (u,v) = (parvalue[0],parvalue[1]).
jstat	-	Status messages = 2 : Surface is degenerate at the point, that is, the surface is not regular at this point. = 1 : Surface is close to degenerate at the point. Angle between tangents is less than the angular tolerance. = 0 : Ok. < 0 : Error.

EXAMPLE OF USE

```
{
    SISLSurf    *surf;  /* Must be defined */
    int         ider = 0;
    int         iside1 = 1;
    int         iside2 = 1;
    double      parvalue[2];  /* Must be defined */
    int         leftknot1 = 0; /* Define initially as zero. For consequtive evaluations
                                    leave leftknot1 as returned from s1510 */
    int         leftknot2 = 0; /* As for leftknot1 */
    double      mehlum[1]; /* A pre allocated array is expected */
    int         jstat = 0;
    ...
    s2510(surf, ider, iside1, iside2, parvalue, leftknot1, leftknot2, mehlum, &jstat);
    ...
}
```

9.1.7 Gaussian curvature of a B-spline or NURBS surface as a NURBS surface.

NAME

s2532 - To interpolate or approximate the Gaussian curvature of a B-spline or NURBS surface by a NURBS surface. The desired continuity of the Gaussian curvature surface is input and this may lead to a patchwork of output surfaces. Interpolation results in a high order surface. If the original surface is a B-spline surface of order k, the result is of order $8k-11$, in the NURBS case, order $32k-35$. To avoid instability beacuse of this, a maximum order is applied. This may lead to an approximation rather than an interpolation.

SYNOPSIS

void s2532(*surf*, *u_continuity*, *v_continuity*, *u_surfnumb*, *v_surfnumb*, *gauss_surf*, *stat*)

SISLSurf	**surf*;
int	*u_continuity*;
int	*v_continuity*1;
int	**u_surfnumb*;
int	**v_surfnumb*;
SISLSurf	****gauss_surf*;
int	**stat*;

ARGUMENTS

Input Arguments:

surf - The original surface.

u_continuity - Desired continuity of the Gaussian curvature surfaces in the u direction: 0 implies positional continuity, 1 implies tangential continuity, and so on. SISL only accepts surfaces of continuity 0 or higher. If the surface is to be intersected with another, the continuity must be 1 or higher to find all the intersection curves. If the requested continuity is higher than the minimum continuity of the surface in the first parameter direction minus 2, an approximation will be performed.

v_continuity - Desired continuity of the Gaussian curvature surfaces in the v direction: 0 implies positional continuity, 1 implies tangential continuity, and so on. SISL only accepts surfaces of continuity 0 or higher. If the surface is to be intersected with another, the continuity must be 1 or higher to find all the intersection curves. If the requested continuity is higher than the minimum continuity of the surface in the second parameter direction minus 2, an approximation will be performed.

Output Arguments:

u_surfnumb - Number of Gaussian curvature surface patches in the u direction.

v_surfnumb	-	Number of Gaussian curvature surface patches in the v direction.
gauss_surf	-	The Gaussian curvature interpolation surfaces. This will be a pointer to an array of length *u_surfnum* * *v_surfnumb* of SISLSurf pointers, where the indexing runs fastest in the u direction.
stat	-	Status messages > 0 : Warning. $= 2$: The surface is degenerate. $= 0$: Ok. < 0 : Error.

EXAMPLE OF USE

```
{
    SISLSurf    *surf;  /* Must be defined */
    int         u_continuity = 0; /* Should depend on continuity of input
                                      surface and the use of the result */
    int         v_continuity = 0;
    int         u_surfnumb = 0;
    int         v_surfnumb = 0;
    SISLSurf    **gauss_surf = NULL;
    int         stat = 0;
    ...
    s2532(surf, u_continuity, v_continuity, &u_surfnumb, &v_surfnumb,
          &gauss_surf, &stat);
    ...
}
```

9.1.8 Mehlum curvature of a B-spline or NURBS surface as a NURBS surface.

NAME

s2536 - To interpolate or approximate the Mehlum curvature of a B-spline or NURBS surface by a NURBS surface. The desired continuity of the Mehlum curvature surface is input and this may lead to a patchwork of output surfaces. Interpolation results in a high order surface. If the original surface is a B-spline surface of order k, the result is of order $12k-17$, in the NURBS case, order $48k-53$. To avoid instability beacuse of this, a maximum order is applied. This may lead to an approximation rather than an interpolation.

SYNOPSIS

void s2536(*surf*, *u_continuity*, *v_continuity*, *u_surfnumb*, *v_surfnumb*, *mehlum_surf*, *stat*)

SISLSurf	**surf*;
int	*u_continuity*;
int	*v_continuity*;
int	**u_surfnumb*;
int	**v_surfnumb*;
SISLSurf	****mehlum_surf*;
int	**stat*;

ARGUMENTS

Input Arguments:

surf - The original surface.

u_continuity - Desired continuity of the Mehlum curvature surfaces in the u direction: 0 implies positional continuity, 1 implies tangential continuity, and so on. SISL only accepts surfaces of continuity 0 or higher. If the surface is to be intersected with another, the continuity must be 1 or higher to find all the intersection curves. If the requested continuity is higher than the minimum continuity of the surface in the first parameter direction minus 2, an approximation will be performed.

v_continuity - Desired continuity of the Mehlum curvature surfaces in the v direction: 0 implies positional continuity, 1 implies tangential continuity, and so on. SISL only accepts surfaces of continuity 0 or higher. If the surface is to be intersected with another, the continuity must be 1 or higher to find all the intersection curves. If the requested continuity is higher than the minimum continuity of the surface in the second parameter direction minus 2, an approximation will be performed.

Output Arguments:

u_surfnumb - Number of Mehlum curvature surface patches in the u direction.

v_surfnumb	-	Number of Mehlum curvature surface patches in the v direction.
mehlum_surf	-	The Mehlum curvature interpolation surfaces. This will be a pointer to an array of length *u_surfnum* * *v_surfnumb* of SISLSurf pointers, where the indexing runs fastest in the u direction.
stat	-	Status messages > 0 : Warning. $= 2$: The surface is degenerate. $= 0$: Ok. < 0 : Error.

EXAMPLE OF USE

```
{
    SISLSurf    *surf  /* Must be defined */;
    int         u_continuity = 0; /* Should depend on continuity of input
                                     surface and the use of the result */
    int         v_continuity = 0;
    int         u_surfnumb = 0;
    int         v_surfnumb = 0;
    SISLSurf    **mehlum_surf = NULL;
    int         stat = 0;
    ...
    s2536(surf, u_continuity, v_continuity, &u_surfnumb, &v_surfnumb,
          &mehlum_surf, &stat);
    ...
}
```

9.1.9 Curvature on a uniform grid of a NURBS surface.

NAME

s2540 - To compute a set of curvature values on a uniform grid in a selected subset of the parameter domain of a NURBS surface.

SYNOPSIS

void s2540(*surf*, *curvature_type*, *export_par_val*, *pick_subpart*, boundary[], *n_u*, *n_v*, *garr*, *stat*)

SISLSurf	**surf*;
int	*curvature_type*;
int	*export_par_val*;
int	*pick_subpart*;
double	*boundary*[ ];
int	*n_u*;
int	*n_v*;
double	***garr*;
int	**stat*;

ARGUMENTS

Input Arguments:

surf - The surface to evaluate.

curvature - The type of curvature:

0	: Gaussian curvature.
1	: Mean curvature.
2	: Absolute curvature.
3	: Total curvature.
4	: Second order Mehlum curvature.
5	: Third order Mehlum curvature.

export - Flag indicating whether the parameter values of the grid points are to be exported:

0	: False, do not export parameter values.
1	: True, do export parameter values.

pick - Flag indicating whether the grid is to be calculated on a subpart of the surface:

0	: False, calculate grid on the complete surface.
1	: True, calculate grid on a part of the surface.

boundary - A rectangular subset of the parameter domain.

0	: Minimum value in the first parameter.
1	: Minimum value in the second parameter.
2	: Maximum value in the first parameter.
3	: Maximum value in the second parameter.

ONLY USED WHEN *pick_subpart* = 1. If *pick_subpart* = 0 the parameter area of surf is returned here.

n_u - Number of segments in the first parameter.

n_v - Number of segments in the second parameter.

Output Arguments:

garr	-	Array containing the computed values on the grid. The allocation is done internally and the dimension is 3*(n_u+1)*(n_v+1) if export_par_val is true, and (n_u+1)*(n_v+1) if export_par_val is false. Each grid-point consists of a triple $(u_i, v_j, curvature(u_i, v_j))$ or only $curvature(u, v_j)$. The sequence runs first in the first parameter.
stat	-	Status messages > 0 : Warning. $= 0$: Ok. < 0 : Error.

EXAMPLE OF USE

```
{
    SISLSurf    *surf;  /* Must be defined */
    int         curvature_type = 1;
    int         export_par_val = 1 ;
    int         pick_subpart = 0;
    double      boundary[4];  /* Must be defined if pick_subpart = 1 */
    int         n_u = 10;
    int         n_v = 10;
    double      *garr = NULL;
    int         stat = 0;
    ...
    s2540(surf, curvature_type, export_par_val, pick_subpart, boundary[], n_u,
          n_v, &garr, &stat);
    ...
}
```

9.1.10 Principal curvatures of a spline surface.

NAME

s2542 - To compute principal curvatures (k1,k2) with corresponding principal directions (d1,d2) of a spline surface at given values (u,v) = (parvalue[0],parvalue[1]), where etl[leftknot1] <= parvalue[0] < etl[leftknot1+1] and et2[leftknot2] <= parvalue[1] < et2[leftknot2+1].

SYNOPSIS

void s2542(*surf*, *ider*, *iside1*, *iside2*, *parvalue*, *leftknot1*, *leftknot2*, *k1*, *k2*, *d1*, *d2*, *jstat*)

SISLSurf	**surf*;
int	*ider*;
int	*iside1*;
int	*iside2*;
double	*parvalue*[ ];
int	**leftknot1*;
int	**leftknot2*;
double	**k1*;
double	**k2*;
double	*d1*[ ];
double	*d2*[ ];
int	**jstat*;

ARGUMENTS

Input Arguments:

surf	-	Pointer to the surface to evaluate.
ider	-	Number of derivatives to calculate. Only implemented for ider=0. < 0 : No derivative calculated. = 0 : Position calculated. = 1 : Position and first derivative calculated, etc.
iside1	-	Flag indicating whether the principal curvature in the first parameter is to be calculated from the left or from the right: < 0 : calculate curvature from the left hand side. >= 0 : calculate curvature from the right hand side.
iside2	-	Flag indicating whether the principal curvature in the second parameter is to be calculated from the left or from the right: < 0 : calculate curvature from the left hand side. >= 0 : calculate curvature from the right hand side.
parvalue	-	Parameter value at which to evaluate. Dimension of parvalue is 2.

Input/Output Arguments:

leftknot1	-	Pointer to the interval in the knot vector in the first parameter direction where parvalue[0] is found, that is: et1[leftknot1] <= parvalue[0] < et1[leftknot1+1]. leftknot1 should be set equal to zero at the first call to the routine.
leftknot2	-	Pointer to the interval in the knot vector in the second parameter direction where parvalue[1] is found, that is: et2[leftknot2] <= parvalue[1] < et2[leftknot2+1]. leftknot2 should be set equal to zero at the first call to the routine.

Output Arguments:

k1	-	Max. principal curvature.
k2	-	Min. principal curvature.
d1	-	Max. direction of the principal curvature k1, given in local coordinates (with regard to Xu,Xv). Dim. = 2.
d2	-	Min. direction of the principal curvature k2, given in local coordinates (with regard to Xu,Xv). Dim. = 2.
jstat	-	Status messages = 2 : Surface is degenerate at the point, that is, the surface is not regular at this point. = 1 : Surface is close to degenerate at the point. Angle between tangents is less than the angular tolerance. = 0 : Ok. < 0 : Error.

EXAMPLE OF USE

```
{
    SISLSurf    *surf;  /* Must be defined */
    int         ider = 0;
    int         iside1 = 1;
    int         iside2 = 1;
    double      parvalue[2];  /* Must be defined */
    int         leftknot1 = 0; /* Define initially as zero. For consequtive evaluations
                                  leave leftknot1 as returned from s2542
    int         leftknot2 = 0; /* As for leftknot1 */
    double      k1;
    double      k2;
    double      d1[2];
    double      d2[2];
    int         jstat = 0;
    . . .
    s2542(surf, ider, iside1, iside2, parvalue, &leftknot1, &leftknot2, &k1, &k2,
          d1, d2, &jstat);
    . . .
}
```

9.1.11 Normal curvature of a spline surface.

NAME

s2544 - To compute the Normal curvature of a splne surface at given values (u,v) = (parvalue[0],parvalue[1]) in the direction (parvalue[2],parvalue[3]) where et1[leftknot1] <= parvalue[0] < et1[leftknot1+1] and et2[leftknot2] <= parvalue[1] < et2[leftknot2+1].

SYNOPSIS

void s2544(*surf*, *ider*, *iside1*, *iside2*, *parvalue*, *leftknot1*, *leftknot2*, *norcurv*, *jstat*)

SISLSurf	**surf*;
int	*ider*;
int	*iside1*;
int	*iside2*;
double	*parvalue*[ ];
int	**leftknot1*;
int	**leftknot2*;
double	*norcurv*[ ];
int	**jstat*;

ARGUMENTS

Input Arguments:

surf - Pointer to the surface to evaluate.

ider - Number of derivatives to calculate. Only implemented for ider=0.
- < 0 : No derivative calculated.
- = 0 : Position calculated.
- = 1 : Position and first derivative calculated, etc.

iside1 - Flag indicating whether the derivatives in the first parameter direction are to be calculated from the left or from the right:
- < 0 : calculate derivative from the left hand side.
- >= 0 : calculate derivative from the right hand side.

iside2 - Flag indicating whether the derivatives in the second parameter direction are to be calculated from the left or from the right:
- < 0 : calculate derivative from the left hand side.
- >= 0 : calculate derivative from the right hand side.

parvalue - Parameter value at which to evaluate plus the direction. Dimension of parvalue is 4.

Input/Output Arguments:

leftknot1 - Pointer to the interval in the knot vector in the first parameter direction where parvalue[0] is found, that is: et1[leftknot1] <= parvalue[0] < et1[leftknot1+1]. leftknot1 should be set equal to zero at the first call to the routine.

leftknot2 - Pointer to the interval in the knot vector in the second parameter direction where parvalue[1] is found, that is: et2[leftknot2] <= parvalue[1] < et2[leftknot2+1]. leftknot2 should be set equal to zero at the first call to the routine.

Output Arguments:

gaussian - Normal curvature and derivatives of normal curvature of the surface at (u,v) = (parvalue[0],parvalue[1]) in the direction (parvalue[2],parvalue[3]).

jstat - Status messages

- = 2 : Surface is degenerate at the point, that is, the surface is not regular at this point.
- = 1 : Surface is close to degenerate at the point. Angle between tangents is less than the angular tolerance.
- = 0 : Ok.
- < 0 : Error.

EXAMPLE OF USE

```
{
    SISLSurf    *surf;  /* Must be defined */
    int         ider = 0;
    int         iside1 = 1;
    int         iside2 = 1;
    double      parvalue[2];  /* Must be defined */
    int         leftknot1 = 0;/* Define initially as zero. For consequtive evaluations
                                 leave leftknot1 as returned from s2544 */
    int         leftknot2 = 0; /* As for leftknot1 */
    double      norcurv[1]; /* An allocated array with length ider is expected */
    int         jstat;
    ...
    s2544(surf, ider, iside1, iside2, parvalue, &leftknot1, &leftknot2, norcurv,
          &jstat);
    ...
}
```

9.1.12 Focal values on a uniform grid of a NURBS surface.

NAME

s2545 - To compute a set of focal values on a uniform grid in a selected subset of the parameter domain of a NURBS surface. A focal value is a surface position offset by the surface curvature.

SYNOPSIS

void s2545(*surf*, *curvature_type*, *export_par_val*, *pick_subpart*, boundary, *n_u*, *n_v*, *scale*, *garr*, *stat*)

SISLSurf	**surf*;
int	*curvature_type*;
int	*export_par_val*;
int	*pick_subpart*;
double	*boundary*[ ];
int	*n_u*;
int	*n_v*;
double	*scale*;
double	***garr*;
int	**stat*;

ARGUMENTS

Input Arguments:

surf - The surface to evaluate.

curvature - The type of curvature:

0	: Gaussian curvature.
1	: Mean curvature.
2	: Absolute curvature.
3	: Total curvature.
4	: Second order Mehlum curvature.
5	: Third order Mehlum curvature.

export - Flag indicating whether the parameter values of the grid points are to be exported:

0	: False, do not export parameter values.
1	: True, do export parameter values.

pick - Flag indicating whether the grid is to be calculated on a subpart of the surface:

0	: False, calculate grid on the complete surface.
1	: True, calculate grid on a part of the surface.

boundary - A rectangular subset of the parameter domain.

0	: Minmum value in the first parameter.
1	: Minmum value in the second parameter.
2	: Maximum value in the first parameter.
3	: Maximum value in the second parameter.

ONLY USED WHEN *pick_subpart* = 1. If *pick_subpart* = 0 the parameter area of surf is returned here.

n_u - Number of segments in the first parameter.

n_v - Number of segments in the second parameter.

scale - Scaling factor.

Output Arguments:

garr - Array containing the computed values on the grid. The allocation is done internally and the dimension is (dim+2)*(n_u+1)*(n_v+1) if export_par_val is true, and dim*(n_u+1)*(n_v+1) if export_par_val is false. Each grid-point consists of dim + 2 values $(u_i, v_j, x(u_i, v_j), ...)$ or only the focal points $(x(u_i, v_j), ...)$. The sequence runs first in the first parameter.

stat - Status messages
> 0 : Warning.
$= 0$: Ok.
< 0 : Error.

EXAMPLE OF USE

```
{
    SISLSurf    *surf;  /* Must be defined */
    int         curvature_type = 0;
    int         export_par_val = 1;
    int         pick_subpart 0;
    double      boundary[4];  /* Must be defined if pick_subpart = 1*/
    int         n_u = 10;
    int         n_v = 10;
    double      scale = 1.0;
    double      *garr = NULL;
    int         stat = 0;
    ...
    s2545(surf, curvature_type, export_par_val, pick_subpart, boundary[], n_u,
          n_v, scale, &garr, &stat);
    ...
}
```

Chapter 10

Surface Utilities

This chapter describes the Surface Utilities. These are common to both the Surface Definition and Surface Interrogation modules.

10.1 Surface Object

In the library both B-spline and NURBS surfaces are stored in a struct SISLSurf containing the following:

int	*ik1*;	Order of surface in first parameter direction.
int	*ik2*;	Order of surface in second parameter direction.
int	*in1*;	Number of coefficients in first parameter direction.
int	*in2*;	Number of coefficients in second parameter direction.
double	**et1*;	Pointer to knot vector in first parameter direction.
double	**et2*;	Pointer to knot vector in second parameter direction.
double	**ecoef*;	Pointer to array of non-rational coefficients of the surface, size $in1 \times in2 \times idim$.
double	**rcoef*;	Pointer to the array of rational vertices and weights, size $in1 \times in2 \times (idim + 1)$.
int	*ikind*;	Type of surface $= 1$: Polynomial B-spline tensor-product surface. $= 2$: Rational B-spline (nurbs) tensor-product surface. $= 3$: Polynomial Bezier tensor-product surface. $= 4$: Rational Bezier tensor-product surface.
int	*idim*;	Dimension of the space in which the surface lies.

int *icopy*;	Indicates whether the arrays of the surface are allocated and copied or referenced when the surface was created. $= 0$: Pointer set to input arrays. The arrays are not deleted by freeSurf. $= 1$: Array allocated and copied. The arrays are deleted by freeSurf. $= 2$: Pointer set to input arrays, but the arrays are to be treated as allocated and copied. The arrays are deleted by freeSurf.
SISLdir *_pdir_;	Pointer to a SISLdir object used for storing surface direction.
SISLbox *_pbox_;	Pointer to a SISLbox object used for storing the surrounded boxes.
int *cuopen_1*;	Open/closed/periodic flag for the first parameter direction. $= -1$: Closed curve with periodic (cyclic) parameterization and overlapping end vertices. $= 0$: Closed curve with k-tuple end knots and coinciding start/end vertices. $= 1$: Open curve (default).
int *cuopen_2*;	Open/closed/periodic flag for the second parameter direction. $= -1$: Closed curve with periodic (cyclic) parameterization and overlapping end vertices. $= 0$: Closed curve with k-tuple end knots and coinciding start/end vertices. $= 1$: Open curve (default).

When using a surface, do not declare a Surface but a pointer to a Surface, and initialize it to point to NULL. Then you may use the dynamic allocation functions newSurface and freeSurface, which are described below, to create and delete surfaces.

There are two ways to pass coefficient and knot arrays to newSurf. By setting $icopy = 1$, newSurf allocates new arrays and copies the given ones. But by setting $icopy = 0$ or 2, newSurf simply points to the given arrays. Therefore it is IMPORTANT that the given arrays have been allocated in free memory beforehand.

10.1.1 Create a new surface object.

NAME

newSurf - Create and initialize a surface object instance. Note that the vertex input to a rational surface is unstandard. Given the surface

$$\mathbf{s}(u,v) = \frac{\sum_{i=1}^{n_1}\sum_{j=1}^{n_2} w_{i,j}\mathbf{p}_{i,j}B_{i,k_1,\mathbf{u}}(u)B_{j,k_2,\mathbf{v}}(v)}{\sum_{i=1}^{n_1}\sum_{j=1}^{n_2} w_{i,j}B_{i,k_1,\mathbf{u}}(u)B_{j,k_2,\mathbf{v}}(v)},$$

must the vertices be given as $w_{1,1}\mathbf{p}_{1,1}, w_{1,1}, w_{1,2}\mathbf{p}_{1,2}, w_{1,2}, \ldots, w_{n_1,n_2}\mathbf{p}_{n_1,n_2}, w_{n_1,n_2}$ when invoking this function. Thus the vertices are multiplied with the associated weight.

SYNOPSIS

SISLSurf *newSurf(*number*1, *number*2, *order*1, *order*2, *knot*1, *knot*2, *coef*, *kind*, *dim*, *copy*)

int	*number*1;
int	*number*2;
int	*order*1;
int	*order*2;
double	*knot1*[ ];
double	*knot2*[ ];
double	*coef*[ ];
int	*kind*;
int	*dim*;
int	*copy*;

ARGUMENTS

Input Arguments:

*number*1	-	Number of vertices in the first parameter direction of new surface.
*number*2	-	Number of vertices in the second parameter direction of new surface.
order1	-	Order of surface in first parameter direction.
order2	-	Order of surface in second parameter direction.
knot1	-	Knot vector of surface in first parameter direction.
knot2	-	Knot vector of surface in second parameter direction.
coef	-	Vertices of surface. These may either be the *dim* dimensional non-rational vertices or the *(dim+1)* dimensional rational vertices.
kind	-	Type of surface. = 1 : Polynomial B-spline surface. = 2 : Rational B-spline (nurbs) surface. = 3 : Polynomial Bezier surface. = 4 : Rational Bezier surface.
dim	-	Dimension of the space in which the surface lies.
copy	-	Flag = 0 : Set pointer to input arrays. = 1 : Copy input arrays. = 2 : Set pointer and remember to free arrays.

Output Arguments:

newSurf - Pointer to new surface. If it is impossible to allocate space for the surface, newSurface returns NULL.

EXAMPLE OF USE

```
{
    SISLSurf    *surf = NULL;
    int         number1 = 5;
    int         number2 = 4;
    int         order1 = 4; /* Polynomial degree 3 */
    int         order2 = 3; /* Polynomial degree 2 */
    double      knot1[9];   /* Must be defined */
    double      knot2[7];   /* Must be defined */
    double      coef[60];   /* Must be defined */
    int         kind = 1;
    int         dim = 3;
    int         copy = 1;
    /* Knots and vertices must be defined prior to the function call.
    The vertices are given in a 1-dimensional array */
    ...
    surf = newSurf(number1, number2, order1, order2, knot1, knot2,
                   coef, kind, dim, copy);
    ...
}
```

10.1.2 Make a copy of a surface object.

NAME

copySurface - Make a copy of a SISLSurface object.

SYNOPSIS

SISLSurf *copySurface(*psurf*)
    SISLSurf **psurf*;

ARGUMENTS

Input Arguments:

psurf	-	Surface to be copied.

Output Arguments:

copySurface	-	The new surface.

EXAMPLE OF USE

```
{
    SISLSurf    *surfcopy = NULL;
    SISLSurf    *surf = NULL;
    int         number1 = 5;
    int         number2 = 4;
    int         order1 = 4;
    int         order2 = 3;
    double      knot1[9];
    double      knot2[7];
    double      coef[60];
    int         kind = 1;
    int         dim = 3;
    int         copy = 1;
    . . .
    surf = newSurf(number1, number2, order1, order2, knot1, knot2,
                   coef, kind, dim, copy);
    . . .
    surfcopy = copySurface(surf);
    . . .
}
```

10.1.3 Delete a surface object.

NAME

freeSurf - Free the space occupied by the surface. Before using freeSurf, make sure that the surface object exists.

SYNOPSIS

void freeSurf(*surf*)

SISLSurf **surf*;

ARGUMENTS

Input Arguments:

surf - Pointer to the surface to delete.

EXAMPLE OF USE

```
{
    SISLSurf   *surf = NULL;
    int        number1 = 5;
    int        number2 = 4;
    int        order1 = 4;
    int        order2 = 3;
    double     knot1[9];
    double     knot2[7];
    double     coef[60];
    int        kind = 1;
    int        dim = 3;
    int        copy = 1;
    . . .
    surf=newSurf(number1, number2, order1, order2, knot1, knot2,
                 coef, kind, dim, copy);
    . . .
    if (surf) freeSurf(surf);
    . . .
}
```

10.2 Evaluation

10.2.1 Compute the position, the derivatives and the normal of a surface at a given parameter value pair.

NAME

s1421 - Evaluate the surface at a given parameter value pair. Compute *der* derivatives and the normal if $der \geq 1$. See also s1424() on page 305.

SYNOPSIS

void s1421(*surf*, *der*, *parvalue*, *leftknot1*, *leftknot2*, *derive*, *normal*, *stat*)

SISLSurf	**surf*;
int	*der*;
double	*parvalue*[ ];
int	**leftknot1*;
int	**leftknot2*;
double	*derive*[ ];
double	*normal*[ ];
int	**stat*;

ARGUMENTS

Input Arguments:

surf - Pointer to the surface to evaluate.

der - Number (order) of derivatives to evaluate.
- < 0 : No derivatives evaluated.
- $= 0$: Position evaluated.
- > 0 : Position and derivatives evaluated.

parvalue - Parameter value at which to evaluate. Dimension of parvalue is 2.

Input/Output Arguments:

leftknot1 - Pointer to the interval in the knot vector in first parameter direction where $parvalue[0]$ is found. The relation

$$etl[leftknot1] \leq parvalue[0] < etl[leftknot1 + 1],$$

where *etl* is the knot vector, should hold. *leftknot1* should be set equal to zero at the first call to the routine. Do not change *leftknot* during a section of calls to s1421().

leftknot2 - Corresponding to *leftknot1* in the second parameter direction.

Output Arguments:

derive - Array where the derivatives of the surface in parvalue are placed. The sequence is position, first derivative in first parameter direction, first derivative in second parameter direction, (2,0) derivative, (1,1) derivative, (0,2) derivative, etc. The expresion

$$dim * (1 + 2 + \ldots + (der + 1)) = dim * (der + 1)(der + 2)/2$$

gives the dimension of the *derive* array.

normal - Normal of surface. Is evaluated if $der \geq 1$. Dimension is dim. The normal is not normalised.

stat - Status messages
- $= 2$: Surface is degenerate at the point, normal has zero length.
- $= 1$: Surface is close to degenerate at the point. Angle between tangents is less than the angular tolerance.
- $= 0$: Ok.
- < 0 : Error.

EXAMPLE OF USE

```
{
    SISLSurf    *surf;  /* Must be defined */
    int         der = 2;
    double      parvalue[2];  /* Must be defined */
    int         leftknot1 = 0; /* Define initially as zero. For consequtive evaluations
                                  leave leftknot1 as returned from s1421 */
    int         leftknot2 = 0; /* As for leftknot1 */
    double      derive[18]; /* Length is spatial dimension times total number of entities */
    double      normal[3]; /* Length is spatial dimension */
    int         stat = 0;
    ...
    s1421(surf, der, parvalue, &leftknot1, &leftknot2, derive, normal, &stat);
    ...
}
```

10.2.2 Compute the position and derivatives of a surface at a given parameter value pair.

NAME

s1424 - Evaluate the surface the parameter value (*parvalue*[0], *parvalue*[1]). Compute the $der1 \times der2$ first derivatives. The derivatives that will be computed are $D^{i,j}$, $i = 0, 1, \ldots, der1$, $j = 0, 1, ..., der2$.

SYNOPSIS

void s1424(*surf*, *der1*, *der2*, *parvalue*, *leftknot1*, *leftknot2*, *derive*, *stat*)

SISLSurf	**surf*;
int	*der1*;
int	*der2*;
double	*parvalue*[ ];
int	**leftknot1*;
int	**leftknot2*;
double	*derive*[ ];
int	**stat*;

ARGUMENTS

Input Arguments:

surf	-	Pointer to the surface to evaluate.
der1	-	Number (order) of derivatives to be evaluated in first parameter direction, where $0 \leq der1$.
der2	-	Number (order) of derivatives to be evaluated in second parameter direction, where $0 \leq der2$.
parvalue	-	Parameter-value at which to evaluate. The dimension of *parvalue* is 2.

Input/Output Arguments:

leftknot1 - Pointer to the interval in the knot vector in first parameter direction where *parvalue*[0] is found. The relation

$$etl[leftknot1] \leq parvalue[0] < etl[leftknot1 + 1],$$

where *etl* is the knot vector, should hold. *leftknot1* should be set equal to zero at the first call to the routine. Do not change the value of *leftknot1* between calls to the routine.

leftknot2 - Corresponding to *leftknot1* in the second parameter direction.

Output Arguments:

derive - Array of size $d(der1+1)(der2+1)$ where the position and the derivative vectors of the surface in (*parvalue*[0], *parvalue*[1]) is placed. $d = surf \to dim$ is the number of elements in each vector and is equal to the geometrical dimension. The vectors are stored in the following order: First the d components of the position vector, then the d components of the $D^{1,0}$ vector, and so on up to the d components of the $D^{der1,0}$ vector, then the d components of the $D^{0,1}$ vector etc. If derive is considered to be a three dimensional array, then its declaration in C would be $derive[der2+1][der1+1][d]$.

stat - Status messages
> 0 : Warning.
$= 0$: Ok.
< 0 : Error.

EXAMPLE OF USE

```
{
    SISLSurf    *surf;  /* Must be defined */
    int         der1 = 2;
    int         der2 = 1;
    double      parvalue[2];  /* Must be defined */
    int         leftknot1 = 0; /* Define initially as zero. For consequtive evaluations
                                  leave leftknot1 as returned from s1424 */
    int         leftknot2 = 0;; /* As for leftknot1 */
    double      derive[18]; /* Length is spatial dimension times total number of entities */
    int         stat = 0;
    ...
    s1424(surf, der1, der2, parvalue, &leftknot1, &leftknot2, derive, &stat);
    ...
}
```

10.2.3 Compute the position and the left- or right-hand derivatives of a surface at a given parameter value pair.

NAME

s1422 - Evaluate and compute the left- or right-hand derivatives of a surface at a given parameter position.

SYNOPSIS

void s1422(*ps1*, *ider*, *iside1*, *iside2*, *epar*, *ilfs*, *ilft*, *eder*, *enorm*, *jstat*)

SISLSurf	**ps1*;
int	*ider*;
int	*iside1*;
int	*iside2*;
double	*epar*[ ];
int	**ilfs*;
int	**ilft*;
double	*eder*[ ];
double	*enorm*[ ];
int	**jstat*;

ARGUMENTS

Input Arguments:

ps1 - Pointer to the surface to evaluate.

ider - Number of derivatives to calculate.
- < 0 : No derivative calculated.
- $= 0$: Position calculated.
- $= 1$: Position and first derivative calculated. etc.

iside1 - Indicator telling if the derivatives in the first parameter direction is to be calculated from the left or from the right:
- < 0 : Calculate derivative from the left hand side.
- ≥ 0 : Calculate derivative from the right hand side.

iside2 - Indicator telling if the derivatives in the second parameter direction is to be calculated from the left or from the right:
- < 0 : Calculate derivative from the left hand side.
- ≥ 0 : Calculate derivative from the right hand side.

epar - Parameter value at which to calculate. Dimension of *epar* is 2.

Input/Output Arguments:

ilfs - Pointer to the interval in the knotvector in first parameter direction where *epar*[0] is found. The relation

$$et1[ilfs] \leq epar[0] < et1[ilfs + 1],$$

where *et1* is the knotvektor, should hold. *ilfs* is set equal to zero at the first call to the routine.

ilft - Corresponding to *ilfs* in the second parameter direction.

Output Arguments:

eder - Array where the derivative of the curve in *apar* is placed. The sequence is position, first derivative in first parameter direction, first derivative in second parameter direction, (2,0) derivative, (1,1) derivative, (0,2) derivative, etc. The expression

$$idim * (1 + 2 + ... + (ider + 1))$$

gives the dimension of the *eder* array.

enorm - Normal of surface. Is calculated if $ider \geq 1$. Dimension is *idim*. The normal is not normalized.

jstat - Status messages

$= 2$: Surface is degenerate at the point, normal has zero length.

$= 1$: Surface is close to degenerate at the point. Angle between tangents is less than the angular tolerance.

$= 0$: Ok.

< 0 : Error.

EXAMPLE OF USE

```
{
    SISLSurf    *ps1;   /* Must be defined */
    int         ider = 1;
    int         iside1 = 1;
    int         iside2 = 1;
    double      epar[2];   /* Must be defined */
    int         ilfs = 0; /* Define initially as zero. For consequtive evaluations
                             leave ilfs as returned from s1422 */
    int         ilft = 0; /* As for ilfs */
    double      eder[9]; /* Length is spatial dimension times total number of entities */
    double      enorm[3]; /* Length is spatial dimension */
    int         jstat = 0;
    ...
    s1422(ps1, ider, iside1, iside2, epar, &ilfs, &ilft, eder, enorm, &jstat);
    ...
}
```

10.2.4 Compute the position and the derivatives of a surface at a given parameter value pair.

NAME

s1425 - To compute the value and $ider1 \times ider2$ first derivatives of a tensor product surface at the point with parameter value (*epar*[0], *epar*[1]). The derivatives that will be computed are $D(i,j)$, $i = 0, 1, \ldots, ider1$, $j = 0, 1, \ldots, ider2$. The calculations are from the right hand or left hand side.

SYNOPSIS

void s1425(*ps1*, *ider1*, *ider2*, *iside1*, *iside2*, *epar*, *ileft1*, *ileft2*, *eder*, *jstat*)

```
SISLSurf   *ps1;
int        ider1;
int        ider2;
int        iside1;
int        iside2;
double     epar[];
int        *ileft1;
int        *ileft2;
double     eder[];
int        *jstat;
```

ARGUMENTS

Input Arguments:

ps1 - Pointer to the surface for which position and derivatives are to be computed.

ider1 - The number of derivatives to be computed with respect to the first parameter direction.

< 0 : Error, no derivative calculated.

$= 0$: No derivatives with respect to the first parameter direction will be computed. (Only derivatives of the type $D(0,0), D(0,1), \ldots, D(0, ider2)$).

$= 1$: Derivatives up to first order with respect to the first parameter direction will be computed.

etc.

ider2 - The number of derivatives to be computed with respect to the second parameter direction.

< 0 : Error, no derivative calculated.

$= 0$: No derivatives with respect to the second parameter direction will be computed. (Only derivatives of the type $D(0,0), D(1,0), \ldots, D(ider1,0)$).

$= 1$: Derivatives up to first order with respect to the second parameter direction will be computed.

etc.

iside1 - Indicator telling if the derivatives in the first parameter direction is to be calculated from the left or from the right:

< 0 : Calculate derivative from the left hand side.

≥ 0 : Calculate derivative from the right hand side.

iside2 - Indicator telling if the derivatives in the second parameter direction is to be calculated from the left or from the right:

< 0 : Calculate derivative from the left hand side.

≥ 0 : Calculate derivative from the right hand side.

epar - Array of dimension 2 containing the parameter values of the point at which the position and derivatives are to be computed.

Input/Output Arguments:

ileft1 - Pointer to the interval in the knot vector in the first parameter direction where *epar*[0] is located. If *et1* is the knot vector in the first parameter direction, the relation

$$et1[ileft] \leq epar[0] < et1[ileft+1],$$

should hold. (If $epar[0] = et1[in1]$ then *ileft* should be $in1 - 1$. Here *in1* is the number of B-spline coefficients associated with *et1*.) If *ileft1* does not have the right value upon entry to the routine, its value will be changed to the value satisfying the above condition.

ileft2 - Pointer to the interval in the knot vector in the second parameter direction where *epar*[1] is located. If *et2* is the knot vector in the second parameter direction, the relation

$$et2[ileft] \leq epar[1] < et2[ileft+1],$$

should hold. (If $epar[1] = et2[in2]$ then *ileft* should be $in2 - 1$. Here *in2* is the number of B-spline coefficients associated with *et2*.) If *ileft2* does not have the right value upon entry to the routine, its value will be changed to the value satisfying the above condition.

Output Arguments:

eder	-	Array of dimension $(ider2+1)*(ider1+1)*idim$ containing the position and the derivative vectors of the surface at the point with parameter value (*epar*[0], *epar*[1]). (*idim* is the number of components of each B-spline coefficient, i.e. the dimension of the Euclidean space in which the surface lies.) These vectors are stored in the following order: First the *idim* components of the position vector, then the *idim* components of the $D(1,0)$ vector, and so on up to the *idim* components of the $D(ider1,0)$ vector, then the *idim* components of the $D(1,1)$ vector etc. Equivalently, if *eder* is considered to be a three dimensional array, then its declaration in C would be $eder[ider2+1, ider1+1, idim]$.
jstat	-	Status messages > 0 : Warning. $= 0$: Ok. < 0 : Error.

EXAMPLE OF USE

```
{
    SISLSurf    *ps1;   /* Must be defined */
    int         ider1 = 1;
    int         ider2 = 1;
    int         iside1 = 0;
    int         iside2 = -1;
    double      epar[2];   /* Must be defined */
    int         ileft1 = 0; /* Define initially as zero. For consequtive evaluations
                               leave ileft1 as returned from s1425 */
    int         ileft2 = 0; /* As for ileft1 */
    double      eder[12]; /* Length is spatial dimension times total number of entities */
    int         jstat = 0;
    ...
    s1425(ps1, ider1, ider2, iside1, iside2, epar, &ileft1, &ileft2, eder, &jstat);
    ...
}
```

10.2.5 Evaluate the surface pointed at by ps1 over an m1 * m2 grid of points (x[i],y[j]). Compute ider derivatives and normals if suitable.

NAME

s1506 - Evaluate the surface pointed at by ps1 over an m1 * m2 grid of points (x[i],y[j]). Compute ider derivatives and normals if suitable.

SYNOPSIS

void s1506(*ps1*, *ider*, *m1*, *x*, *m2*, *y*, *eder*, *norm*, *jstat*)

SISLSurf	**ps1*;
int	*ider*;
int	*m1*;
double	**x*;
int	*m2*;
double	**y*;
double	*eder*[ ];
double	*norm*[ ];
int	**jstat*;

ARGUMENTS

Input Arguments:

ps1	-	Pointer to the surface to evaluate.
ider	-	Number of derivatives to calculate. < 0 : No derivative calculated. $= 0$: Position calculated. $= 1$: Position and first derivative calculated. etc.
m1	-	Number of grid points in first direction.
x	-	Array of x values of the grid.
m2	-	Number of grid points in first direction.
y	-	Array of y values of the grid.

Output Arguments:

eder	-	Array where the derivatives of the surface are placed, dimension idim * ((ider+1)(ider+2) / 2) * m1 * m2. The sequence is position, first derivative in first parameter direction, first derivative in second parameter direction, (2,0) derivative, (1,1) derivative, (0,2) derivative, etc. at point (x[0],y[0]), followed by the same information at (x[1],y[0]), etc.
norm	-	Normals of surface. Is calculated if ider ≥ 1. Dimension is idim*m1*m2. The normals are not normalized.
jstat	-	status messages $= 2$: Surface is degenerate at some point, normal has zero length. $= 1$: Surface is close to degenerate at some point. Angle between tangents, less than angular tolerance. $= 0$: Ok. < 0 : Error.

EXAMPLE OF USE

```
{
    SISLSurf    *ps1;   /* Must be defined */
    int         ider = 1;
    int         m1 = 10;
    double      x[10];   /* Must be defined */
    int         m2 = 8;
    double      y[8];   /* Must be defined */
    double      eder[720]; /* Length: spatial dimension times number of
                              entities times number of grid points */
    double      norm[240]; /* Length: spatial dimension times number of grid points */
    int         jstat = 0;
    ...
    s1506(ps1, ider, m1, x, m2, y, eder, norm, &jstat);
    ...
}
```

10.3 Subdivision

10.3.1 Subdivide a surface along a given parameter line.

NAME

s1711 - Subdivide a surface along a given internal parameter line.

SYNOPSIS

void s1711(*surf*, *pardir*, *parval*, *newsurf1*, *newsurf2*, *stat*)

SISLSurf	**surf*;
int	*pardir*;
double	*parval*;
SISLSurf	***newsurf1*;
SISLSurf	***newsurf2*;
int	**stat*;

ARGUMENTS

Input Arguments:

surf	-	Surface to subdivide.
pardir	-	Value used to indicate in which parameter direction the subdivision is to take place. $= 1$: First parameter direction. $= 2$: Second parameter direction.
parval	-	Parameter value at which to subdivide.

Output Arguments:

newsurf1	-	First part of the subdivided surface.
newsurf2	-	Second part of the subdivided surface.
stat	-	Status messages > 0 : warning $= 0$: ok < 0 : error

EXAMPLE OF USE

```
{
    SISLSurf    *surf;  /* Must be defined */
    int         pardir = 2;
    double      parval;  /* Must be defined */
    SISLSurf    *newsurf1 = NULL;
    SISLSurf    *newsurf2 = NULL;
    int         stat = 0;
    ...
    s1711(surf, pardir, parval, &newsurf1, &newsurf2, &stat);
    ...
}
```

10.3.2 Insert a given set of knots, in each parameter direction, into the description of a surface.

NAME

s1025 - Insert a given set of knots in each parameter direction into the description of a surface.
NOTE : When the surface is periodic in one direction, the input parameter values in this direction must lie in the half-open interval $[et[kk-1], et[kn)$, the function will automatically update the extra knots and coeffisients.

SYNOPSIS

void s1025(*ps*, *epar1*, *inpar1*, *epar2*, *inpar2*, *rsnew*, *jstat*)

SISLSurf	**ps*;
double	*epar1*[ ];
int	*inpar1*;
double	*epar2*[ ];
int	*inpar2*;
SISLSurf	***rsnew*;
int	**jstat*;

ARGUMENTS

Input Arguments:

ps	-	Surface to be refined.
epar1	-	Knots to insert in first parameter direction.
inpar1	-	Number of new knots in first parameter direction.
epar2	-	Knots to insert in second parameter direction.
inpar2	-	Number of new knots in second parameter direction.

Output Arguments:

rsnew	-	The new, refined surface.
stat	-	Status messages > 0 : Warning. $= 0$: Ok. < 0 : Error.

EXAMPLE OF USE

```
{
    SISLSurf    *ps;   /* Must be defined */
    double      epar1[3];   /* Must be defined */
    int         inpar1 = 3;
    double      epar2[4];   /* Must be defined */
    int         inpar2 = 4;
    SISLSurf    *rsnew = NULL;
    int         jstat = 0;
    . . .
    s1025(ps, epar1, inpar1, epar2, inpar2, &rsnew, &jstat);
    . . .
}
```

10.4 Picking Curves from a Surface

10.4.1 Pick a curve along a constant parameter line in a surface.

NAME

s1439 - Make a constant parameter curve along a given parameter direction in a surface.

SYNOPSIS

void s1439(*ps1*, *apar*, *idirec*, *rcurve*, *jstat*)

SISLSurf	**ps1*;
double	*apar*;
int	*idirec*;
SISLCurve	***rcurve*;
int	**jstat*;

ARGUMENTS

Input Arguments:

ps1	-	Pointer to the surface.
apar	-	Parameter value to use when picking out constant parameter curve.
idirec	-	Parameter direction in which to pick (must be 1 or 2).

Output Arguments:

rcurve	-	Constant parameter curve.
jstat	-	Status messages > 0 : Warning. $= 0$: Ok. < 0 : Error.

EXAMPLE OF USE

```
{
    SISLSurf     *ps1;  /* Must be defined */
    double       apar;  /* Must be defined */
    int          idirec = 1;
    SISLCurve    *rcurve = NULL;
    int          jstat = 0;
    ...
    s1439(ps1, apar, idirec, &rcurve, &jstat);
    ...
}
```

10.4.2 Pick the curve lying in a surface, described by a curve in the parameter plane of the surface.

NAME

s1383 - To create a 3D approximation to the curve in a surface, traced out by a curve in the parameter plane. The output is represented as a B-spline curve.

SYNOPSIS

void s1383(*surf*, *curve*, *epsge*, *maxstep*, *der*, *newcurve1*, *newcurve2*, *newcurve3*, stat)

```
SISLSurf    *surf;
SISLCurve   *curve;
double      epsge;
double      maxstep;
int         der;
SISLCurve   **newcurve1;
SISLCurve   **newcurve2;
SISLCurve   **newcurve3;
int         *stat;
```

ARGUMENTS

Input Arguments:

surf	-	The surface object
curve	-	The input curve in the parameter plane.
epsge	-	Maximal deviation allowed between true 3D curve lying in the surface, and the approximated 3D curve.
maxstep	-	Maximum step length. Is neglected if $maxstep \leq epsge$ If $maxstep \leq 0.0$ the 3D box of the surface is used to estimate the maximum step length.
der	-	Derivative indicator $= 0$: Calculate only position curve. $= 1$: Calculate position + derivative curves.

Output Arguments:

newcurve1	-	Pointer to the B-spline curve approximating the position curve.
newcurve2	-	Pointer to the B-spline curve approximating the derivative curve along the position curve in the first parameter direction of the surface.
newcurve3	-	Pointer to the B-spline curve approximating derivative curve in the second parameter direction of the surface, along the position curve.
stat	-	Status messages > 0 : warning $= 0$: ok < 0 : error

EXAMPLE OF USE

```
{
    SISLSurf     *surf;  /* Must be defined */
    SISLCurve    *curve;  /* Must be defined */
    double       epsge = 0.0001;
    double       maxstep = 0.0;
    int          der = 1;
    SISLCurve    *newcurve1 = NULL;
    SISLCurve    *newcurve2 = NULL;
    SISLCurve    *newcurve3 = NULL;
    int          stat = 0;
    ...
    s1383(surf, curve, epsge, maxstep, der, &newcurve1, &newcurve2,
          &newcurve3, &stat);
    ...
}
```

10.5 Pick a Part of a Surface.

NAME

s1001 - To pick a part of a surface. The surface produced will always be k-regular, i.e. with k-tupple start/end knots.

SYNOPSIS

void s1001(*ps*, *min1*, *min2*, *max1*, *max2*, *rsnew*, *jstat*)

```
SISLSurf   *ps;
double     min1;
double     min2;
double     max1;
double     max2;
SISLSurf   **rsnew;
int        *jstat;
```

ARGUMENTS

Input Arguments:

ps	-	Surface to pick a part of.
min1	-	Minimum value in first parameter direction.
min2	-	Minimum value in second parameter direction.
max1	-	Maximum value in first parameter direction.
max2	-	Maximum value second parameter direction.

Output Arguments:

rsnew	-	The new, picked surface.
jstat	-	Status messages > 0 : Warning. $= 0$: Ok. < 0 : Error.

EXAMPLE OF USE

```
{
    SISLSurf   *ps;   /* Must be defined */
    double     min1;  /* Must be defined */
    double     min2;  /* Must be defined */
    double     max1;  /* Must be defined */
    double     max2;  /* Must be defined */
    SISLSurf   *rsnew = NULL;
    int        jstat = 0;
    . . .
    s1001(ps, min1, min2, max1, max2, &rsnew, &jstat);
    . . .
}
```

10.6 Turn the Direction of the Surface Normal Vector.

NAME

s1440 - Interchange the two parameter directions used in the mathematical description of a surface and thereby change the direction of the normal vector of the surface.

SYNOPSIS

void s1440(*surf*, *newsurf*, *stat*)

```
SISLSurf    *surf;
SISLSurf    **newsurf;
int         *stat;
```

ARGUMENTS

Input Arguments:

surf	-	Pointer to the original surface.

Output Arguments:

newsurf	-	Pointer to the surface where the parameter directions are interchanged.
stat	-	Status messages > 0 : warning $= 0$: ok < 0 : error

EXAMPLE OF USE

```
{
    SISLSurf    *surf;  /* Must be defined */
    SISLSurf    *newsurf = NULL;
    int         stat = 0;
    . . .
    s1440(surf, &newsurf, &stat);
    . . .
}
```

Chapter 11

Data Reduction

11.1 Curves

11.1.1 Data reduction: B-spline curve as input.

NAME

s1940 - To remove as many knots as possible from a spline curve without perturbing the curve more than a given tolerance.

SYNOPSIS

void s1940(*oldcurve*, *eps*, *startfix*, *endfix*, *iopen*, *itmax*, *newcurve*, *maxerr*, *stat*)

SISLCurve	**oldcurve*;
double	*eps*[ ];
int	*startfix*;
int	*endfix*;
int	*iopen*;
int	*itmax*;
SISLCurve	***newcurve*;
double	*maxerr*[ ];
int	**stat*;

ARGUMENTS

Input Arguments:

oldcurve	-	pointer to the original spline curve.
eps	-	double array giving the desired absolute accuracy of the final approximation as compared to oldcurve. If oldcurve is a spline curve in a space of dimension dim, then eps must have length dim. Note that it is not relative, but absolute accuracy that is being used. This means that the difference in component i at any parameter value, between the given curve and the approximation, is to be less than eps[i]. Note that in such comparisons the same parametrization is used for both curves.

startfix - the number of derivatives to be kept fixed at the beginning of the knot interval. The $0, \ldots, (startfix - 1)$ derivatives will be kept fixed. If startfix < 0, this routine will set it to 0. If startfix < the order of the curve, this routine will set it to the order.

endfix - the number of derivatives to be kept fixed at the end of the knot interval. The $0, \ldots, (endfix - 1)$ derivatives will be kept fixed. If endfix < 0, this routine will set it to 0. If endfix < the order of the curve, this routine will set it to the order.

iopen - Open/closed parameter
$= 1$: Produce open curve.
$= 0$: Produce closed, non-periodic curve if possible.
$= -1$: Produce closed, periodic curve if possible.

itmax - maximum number of iterations. The routine will follow an iterative procedure trying to remove more and more knots. The process will almost always stop after less than 10 iterations and it will often stop after less than 5 iterations. A suitable value for itmax is therefore usually in the region 3-10.

Output Arguments:

-

newcurve - the spline approximation on the reduced knot vector.

maxerr - double array containing an upper bound for the pointwise error in each of the components of the spline approximation. The two curves oldcurve and newcurve are compared at the same parameter value, i.e., if oldcurve is f and newcurve is g, then $|f(t) - g(t)| <= eps$ in each of the components.

stat - Status messages
> 0 : Warning.
$= 0$: Ok.
< 0 : Error.

EXAMPLE OF USE

```
{
    SISLCurve   *oldcurve;  /* Must be defined */
    double      eps[3]; /* Spatial dimension. Must be defined */
    int         startfix = 1;
    int         endfix = 1;
    int         iopen = 1;
    int         itmax = 8;
    SISLCurve   *newcurve = NULL;
    double      maxerr[3]; /* Spatial dimension */
    int         stat = 0;
    ...
    s1940(oldcurve, eps, startfix, endfix, iopen, itmax, &newcurve, maxerr,
          &stat);
    ...
```

}

11.1.2 Data reduction: Point data as input.

NAME

s1961 - To compute a spline-approximation to the data given by the points ep, and represent it as a B-spline curve with parameterization determined by the parameter ipar. The approximation is determined by first forming the piecewise linear interpolant to the data, and then performing knot removal on this initial approximation.

SYNOPSIS

void s1961(*ep*, *im*, *idim*, *ipar*, *epar*, *eeps*, *ilend*, *irend*, *iopen*, *afctol*, *itmax*, *ik*, *rc*, *emxerr*, *jstat*)

double	*ep*[ ];
int	*im*;
int	*idim*;
int	*ipar*;
double	*epar*[ ];
double	*eeps*[ ];
int	*ilend*;
int	*irend*;
int	*iopen*;
double	*afctol*;
int	*itmax*;
int	*ik*;
SISLCurve	***rc*;
double	*emxerr*[ ];
int	**jstat*;

ARGUMENTS

Input Arguments:

ep	-	Array (length $idim * im$) containing the points to be approximated.
im	-	The no. of data points.
idim	-	The dimension of the euclidean space in which the data points lie, i.e. the number of components of each data point.
ipar	-	Flag indicating the type of parameterization to be used: $= 1$: Paramterize by accumulated cord length. (Arc length parametrization for the piecewise linear interpolant.) $= 2$: Uniform parameterization. $= 3$: Parametrization given by epar. If ipar < 1 or ipar > 3, it will be set to 1.
epar	-	Array (length im) containing a parametrization of the given data.
eeps	-	Array (length idim) containing the tolerance to be used during the data reduction stage. The final approximation to the data will deviate less than eeps from the piecewise linear interpolant in each of the idim components.

ilend	-	The no. of derivatives that are not allowed to change at the left end of the curve. The $0, \ldots, (ilend - 1)$ derivatives will be kept fixed. If ilend < 0, this routine will set it to 0. If ilend $< ik$, this routine will set it to ik.
irend	-	The no. of derivatives that are not allowed to change at the right end of the curve. The $0, \ldots, (irend - 1)$ derivatives will be kept fixed. If irend < 0, this routine will set it to 0. If irend $< ik$, this routine will set it to ik.
iopen	-	Open/closed parameter $= 1$: Produce open curve. $= 0$: Produce closed, non-periodic curve if possible. $= -1$: Produce closed, periodic curve if possible. If a closed or periodic curve is to be produced and the start- and endpoint is more distant than the length of the tolerance, a new point is added. Note that if the parametrization is given as input, the parametrization if the last point will be arbitrary.
afctol	-	Number indicating how the tolerance is to be shared between the two data reduction stages. For the linear reduction, a tolerance of $afctol * eeps$ will be used, while a tolerance of $(1 - afctol) * eeps$ will be used during the final data reduction. (Similarly for edgeps.)
itmax	-	Max. no. of iterations in the data-reduction routine.
ik	-	The polynomial order of the approximation.

Output Arguments:

rc	-	Pointer to curve.
emxerr	-	Array (length idim) (allocated outside this routine.) containing for each component an upper bound on the max. deviation of the final approximation from the initial piecewise linear interpolant.
jstat	-	Status messages > 0 : Warning. $= 0$: Ok. < 0 : Error.

EXAMPLE OF USE

```
{
    double  ep[300];   /* Must be defined */
    int     im = 100;
    int     idim = 3;
    int     ipar = 1;
    double  epar[100]; /* Used if ipar = 3 */
    double  eeps[3];   /* Spatial dimension. Must be defined */
    int     ilend = 0;
    int     irend = 0;
    int     iopen = 1;
    double  afctol = 0.5;
    int     itmax = 6;
    int     ik = 4;
```

```
    SISLCurve   *rc = NULL;
    double      emxerr[3]; /* Spatial dimension */
    int         jstat = 0;
    ...
    s1961(ep, im, idim, ipar, epar, eeps, ilend, irend, iopen, afctol, itmax, ik,
          &rc, emxerr, &jstat);
    ...
}
```

11.1.3 Data reduction: Points and tangents as input.

NAME

s1962 - To compute the approximation to the data given by the points ep and the derivatives (tangents) ev, and represent it as a B-spline curve with parametrization determined by the parameter ipar. The approximation is determined by first forming the cubic hermite interpolant to the data, and then performing knot removal on this initial approximation.

SYNOPSIS

void s1962(*ep*, *ev*, *im*, *idim*, *ipar*, *epar*, *eeps*, *ilend*, *irend*, *iopen*, *itmax*, *rc*, *emxerr*, *jstat*)

double	*ep*[ ];
double	*ev*[ ];
int	*im*;
int	*idim*;
int	*ipar*;
double	*epar*[ ];
double	*eeps*[ ];
int	*ilend*;
int	*irend*;
int	*iopen*;
int	*itmax*;
SISLCurve	***rc*;
double	*emxerr*[ ];
int	**jstat*;

ARGUMENTS

Input Arguments:

ep	-	Array (length idim*im) comtaining the points to be approximated.
ev	-	Array (length idim*im) containing the derivatives of the points to be approximated.
im	-	The no. of data points.
idim	-	The dimension of the euclidean space in which the curve lies.
ipar	-	Flag indicating the type of parameterization to be used: = 1 : Paramterize by accumulated cord length. (Arc length parametrization for the piecewise linear interpolant.) = 2 : Uniform parameterization. = 3 : Parametrization given by epar. If ipar $<$ 1 or ipar $>$ 3, it will be set to 1.
epar	-	Array (length im) containing a parameterization of the given data.
eeps	-	Array (length idim) giving the desired accuracy of the spline-approximation in each component.

ilend	-	The no. of derivatives that are not allowed to change at the left end of the curve. The $0, \ldots, (ilend-1)$ derivatives will be kept fixed. If ilend < 0, this routine will set it to 0. If ilend $< ik$, this routine will set it to ik.
irend	-	The no. of derivatives that are not allowed to change at the right end of the curve. The $0, \ldots, (irend-1)$ derivatives will be kept fixed. If irend < 0, this routine will set it to 0. If irend $< ik$, this routine will set it to ik.
iopen	-	Open/closed parameter $= 1$: Produce open curve. $= 0$: Produce closed, non-periodic curve if possible. $= -1$: Produce closed, periodic curve if possible. If a closed or periodic curve is to be produced and the start- and endpoint is more distant than the length of the tolerance, a new point is added. Note that if the parametrization is given as input, the parametrization if the last point will be arbitrary.
itmax	-	Max. no. of iteration.

Output Arguments:

rc	-	Pointer to curve.
emxerr	-	Array (length idim) (allocated outside this routine.) containing an upper bound for the pointwise error in each of the components of the spline-approximation.
jstat	-	Status messages > 0 : Warning. $= 0$: Ok. < 0 : Error.

EXAMPLE OF USE

```
{
    double      ep[120];  /* Must be defined */
    double      ev[120];  /* Must be defined */
    int         im = 40;
    int         idim = 3;
    int         ipar = 3;
    double      epar[40];  /* Must be defined. Used only if ipar = 3 */
    double      eeps[3];  /* Spatial dimension. Must be defined */
    int         ilend = 1;
    int         irend = 1;
    int         iopen = 1;
    int         itmax = 8;
    SISLCurve   *rc = NULL;
    double      emxerr[3];  /* Spatial dimension */
    int         jstat = 0;
    ...
    s1962(ep, ev, im, idim, ipar, epar, eeps, ilend, irend, iopen, itmax, &rc,
          emxerr, &jstat);
    ...
}
```

11.1.4 Degree reduction: B-spline curve as input.

NAME

s1963 - To approximate the input spline curve by a cubic spline curve with error less than eeps in each of the kdim components.

SYNOPSIS

void s1963(*pc*, *eeps*, *ilend*, *irend*, *iopen*, *itmax*, *rc*, *jstat*)

SISLCurve	**pc*;
double	*eeps*[ ];
int	*ilend*;
int	*irend*;
int	*iopen*;
int	*itmax*;
SISLCurve	***rc*;
int	**jstat*;

ARGUMENTS

Input Arguments:

pc	-	Pointer to curve.
eeps	-	Array (length kdim) giving the desired accuracy of the spline-approximation in each component.
ilend	-	The no. of derivatives that are not allowed to change at the left end of the curve. The $0, \ldots, (ilend-1)$ derivatives will be kept fixed. If ilend < 0, this routine will set it to 0. If ilend $< ik$, this routine will set it to ik.
irend	-	The no. of derivatives that are not allowed to change at the right end of the curve. The $0, \ldots, (irend-1)$ derivatives will be kept fixed. If irend < 0, this routine will set it to 0. If irend $< ik$, this routine will set it to ik.
iopen	-	Open/closed parameter $= 1$: Produce open curve. $= 0$: Produce closed, non-periodic curve if possible. $= -1$: Produce closed, periodic curve if possible.
itmax	-	Max. no. of iterations.

Output Arguments:

rc	-	Pointer to curve.
jstat	-	Status messages > 0 : Warning. $= 0$: Ok. < 0 : Error.

EXAMPLE OF USE

```
{
    SISLCurve   *pc;  /* Must be defined */
    double      eeps[3];  /* Spatial dimension. Must be defined */
    int         ilend = 1;
    int         irend = 1;
```

```
    int         iopen = 1;
    int         itmax = 8;
    SISLCurve   *rc = NULL;
    int         jstat = 0;
    ...
    s1963(pc, eeps, ilend, irend, iopen, itmax, &rc, &jstat);
    ...
}
```

11.2 Surfaces

11.2.1 Data reduction: B-spline surface as input.

NAME

s1965 - To remove as many knots as possible from a spline surface without perturbing the surface more than the given tolerance. The error in continuity over the start and end of a closed or periodic surface is only guaranteed to be within edgeps.

SYNOPSIS

void s1965(*oldsurf*, *eps*, *edgefix*, *iopen1*, *iopen2*, *edgeps*, *opt*, *itmax*, *newsurf*, *maxerr*, *stat*)

SISLSurf	**oldsurf*;
double	*eps*[ ];
int	*edgefix*[4];
int	*iopen1*;
int	*iopen2*;
double	*edgeps*[ ];
int	*opt*;
int	*itmax*;
SISLSurf	***newsurf*;
double	*maxerr*[ ];
int	**stat*;

ARGUMENTS

Input Arguments:

oldsurf - pointer to the original spline surface. Note if the polynomial orders of the surface are k1 and k2, then the two knot vectors are assumed to have knots of multiplicity k1 and k2 at the ends.

eps - double array of length dim (the number of components of the surface, typically three) giving the desired accuracy of the final approximation compared to oldcurve. Note that in such comparisons the two surfaces are not reparametrized in any way.

edgefix - integer array of dimension (4) giving the number of derivatives to be kept fixed along each edge of the surface. The numbering of the edges is the same as for edgeps below. All the derivatives of order $< nend(i) - 1$ will be kept fixed along edge i. Hence $nend(i) = 0$ indicates that nothing is to be kept fixed along edge i. NB! TO BE KEPT FIXED HERE MEANS TO HAVE ERROR LESS THAN EDGEPS. IN GENERAL, IT IS IMPOSSIBLE TO REMOVE KNOTS AND KEEP AN EDGE COMPLETELY FIXED.

iopen1 - Open/closed parameter in first direction.
$= 1$: Produce open surface.
$= 0$: Produce closed, non-periodic surface if possible.
$= -1$: Produce closed, periodic surface

iopen2 - Open/closed parameter in second direction.
= 1 : Produce open surface.
= 0 : Produce closed, non-periodic surface if possible.
= −1 : Produce closed, periodic surface

edgeps - double array of length 4*dim ([4,dim]) (dim is the number of components of each coefficient) containing the maximum deviation which is acceptable along the edges of the surface. $edgeps[0] - edgeps[dim-1]$ gives the tolerance along the edge corresponding to x1 (the first parameter) having it's minimum value. $edgeps[dim] - edgeps[2*dim-1]$ gives the tolerance along the edge corresponding to x1 (the first parameter) having it's maximum value. $edgeps[2*dim] - edgeps[3*dim-1]$ gives the tolerance along the edge corresponding to x2 (the second parameter) having it's minimum value. $edgeps[3*dim] - edgeps[4*dim-1]$ gives the tolerance along the edge corresponding to x2 (the second parameter) having its maximum value. NB! EDGEPS WILL ONLY HAVE ANY SIGNIFICANCE IF THE CORRESPONDING ELEMENT OF EDGEFIX IS POSITIVE.

itmax - maximum number of iterations. The routine will follow an iterative procedure trying to remove more and more knots, one direction at a time. The process will almost always stop after less than 10 iterations and it will often stop after less than 5 iterations. A suitable value for itmax is therefore usually in the region 3-10.

opt - integer indicating the order in which the knot removal is to be performed.

1 : remove knots in parameter 1 only.
2 : remove knots in parameter 2 only.
3 : remove knots first in parameter 1 and then 2.
4 : remove knots first in parameter 2 and then 1.

Output Arguments:

newsurf - the approximating surface on the reduced knot vectors.

maxerr - double array of length dim containing an upper bound for the pointwise error in each of the components of the spline approximation. The two surfaces oldsurf and newsurf are compared at the same parameter vaues, i.e., if oldsurf is f and newsurf is g then $|f(u,v) - g(u,v)| <= eps$ in each of the components.

stat - Status messages
> 0 : Warning.
$= 0$: Ok.
< 0 : Error.

EXAMPLE OF USE

```
{
    SISLSurf    *oldsurf;   /* Must be defined */
```

```
    double      eps[3];   /* Spatial dimension. Must be defined */
    int         edgefix[4];   /* Must be defined */
    int         iopen1 = 1;
    int         iopen2 = 1;
    double      edgeps[12];   /* Spatial dimension times number of edges.
                                   Must be defined */
    int         opt = 3;
    int         itmax = 8;
    SISLSurf    *newsurf = NULL;
    double      maxerr[3];   /* Spatial dimension */
    int         stat = 0;
    ...
    s1965(oldsurf, eps, edgefix, iopen1, iopen2, edgeps, opt, itmax, &newsurf,
          maxerr, &stat);
    ...
}
```

11.2.2 Data reduction: Point data as input.

NAME

s1966 - To compute a tensor-product spline-approximation of order (ik1,ik2) to the rectangular array of idim-dimensional points given by ep.

SYNOPSIS

void s1966(*ep*, *im1*, *im2*, *idim*, *ipar*, *epar1*, *epar2*, *eeps*, *nend*, *iopen1*, *iopen2*, *edgeps*, *afctol*, *iopt*, *itmax*, *ik1*, *ik2*, *rs*, *emxerr*, *jstat*)

double	*ep*[ ];
int	*im1*;
int	*im2*;
int	*idim*;
int	*ipar*;
double	*epar1*[ ];
double	*epar2*[ ];
double	*eeps*[ ];
int	*nend*[ ];
int	*iopen1*;
int	*iopen2*;
double	*edgeps*[ ];
double	*afctol*;
int	*iopt*;
int	*itmax*;
int	*ik1*;
int	*ik2*;
SISLSurf	***rs*;
double	*emxerr*[ ];
int	**jstat*;

ARGUMENTS

Input Arguments:

ep	-	Array (length idim*im1*im2) containing the points to be approximated.
im1	-	The no. of points in the first parameter.
im2	-	The no. of points in the second parameter.
idim	-	The no. of components of each input point. The approximation will be a parametric surface situated in idim-dimensional Euclidean space (usually 3).
ipar	-	Flag determining the parametrization of the data points: = 1 : Mean accumulated cord-length parameterization. = 2 : Uniform parametrization. = 3 : Parametrization given by epar1 and epar2.
epar1	-	Array (length im1) containing a parametrization in the first parameter. (Will only be used if $ipar = 3$).
epar2	-	Array (length im2) containing a parametrization in the second parameter. (Will only be used if $ipar = 3$).

eeps	-	Array (length idim) containing the max. permissible deviation of the approximation from the given data points, in each of the components. More specifically, the approximation will not deviate more than eeps(kdim) in component no. kdim, from the bilinear approximation to the data.
nend	-	Array (length 4) giving the no. of derivatives to be kept fixed along each edge of the bilinear interpolant. The numbering of the edges is the same as for edgeps below. All the derivatives of order $< (nend(i)-1)$ will be kept fixed along the edge i. Hence $nend(i) = 0$ indicates that nothing is to be kept fixed along edge i. To be kept fixed here means to have error less than edgeps. In general, it is impossible to remove any knots and keep an edge completely fixed.
iopen1	-	Open/closed parameter in first direction. $= 1$: Produce open surface. $= 0$: Produce closed, non-periodic surface if possible. $= -1$: Produce closed, periodic surface NB! The surface will be closed/periodic only if the first and last column of data points are (approximately) equal.
iopen2	-	Open/closed parameter in second direction. $= 1$: Produce open surface. $= 0$: Produce closed, non-periodic surface if possible. $= -1$: Produce closed, periodic surface NB! The surface will be closed/periodic only if the first and last row of data points are (approximately) equal.
edgeps	-	Array (length idim*4) containing the max. deviation from the bilinear interpolant which is acceptable along the edges of the surface. edgeps(1,i):edgeps(idim,i) gives the tolerance along the edge corresponding to the i-th parameter having one of it's extremal-values. $i = 1$: min value of first parameter. $i = 2$: max value of first parameter. $i = 3$: min value of second parameter. $i = 4$: max value of second parameter. edgeps(kp,i) will only have significance if $nend(i) > 0$.
afctol	-	$0.0 >= afctol <= 1.0$. Afctol indicates how the tolerance is to be shared between the two data-reduction stages. For the linear reduction, a tolerance of $afctol * eeps$ will be used, while a tolerance of $(1.0 - afctol) * eeps$ will be used during the final data reduction (similarly for edgeps.) Default is 0.
iopt	-	Flag indicating the order in which the data-reduction is to be performed: $= 1$: Remove knots in parameter 1 only. $= 2$: Remove knots in parameter 2 only. $= 3$: Remove knots first in parameter 1 and then in 2. $= 4$: Remove knots first in parameter 2 and then in 1.
itmax	-	Max. no. of iterations in the data-reduction..

ik1	-	The order of the approximation in the first parameter.
ik2	-	The order of the approximation in the second parameter.

Output Arguments:

rs	-	Pointer to surface.
emxerr	-	Array (length idim) (allocated outside this routine.) containing the error in the approximation to the data. This is a guaranteed upper bound on the max. deviation in each component, between the final approximation and the bilinear spline- pproximation to the original data.
jstat	-	Status messages > 0 : Warning. $= 0$: Ok. < 0 : Error.

EXAMPLE OF USE

```
{
    double      ep[750];  /* Spatial dimension times number of points. Must be defined */
    int         im1 = 50;
    int         im2 = 50;
    int         idim = 3;
    int         ipar = 1;
    double      epar1[50]; /* Used if ipar = 3 */
    double      epar2[50]; /* Used if ipar = 3 */
    double      eeps[3];  /* Must be defined */
    int         nend[4];  /* Must be defined */
    int         iopen1 = 1;
    int         iopen2 = 1;
    double      edgeps[12]; /* Spatial dimension times number of edges.
                                    Must be defined */
    double      afctol = 0.5;
    int         iopt = 4;
    int         itmax = 8;
    int         ik1 = 4;
    int         ik2 = 4;
    SISLSurf    *rs = NULL;
    double      emxerr[3]; /* Spatial dimension */
    int         jstat = 0;
    ...
    s1966(ep, im1, im2, idim, ipar, epar1, epar2, eeps, nend, iopen1, iopen2,
          edgeps, afctol, iopt, itmax, ik1, ik2, &rs, emxerr, &jstat);
    ...
}
```

11.2.3 Data reduction: Points and tangents as input.

NAME

s1967 - To compute a bicubic hermite spline-approximation to the position and derivative data given by ep,etang1,etang2 and eder11.

SYNOPSIS

void s1967(*ep*, *etang1*, *etang2*, *eder11*, *im1*, *im2*, *idim*, *ipar*, *epar1*, *epar2*, *eeps*, *nend*, *iopen1*, *iopen2*, *edgeps*, *iopt*, *itmax*, *rs*, *emxerr*, *jstat*)

double	*ep*[ ];
double	*etang1*[ ];
double	*etang2*[ ];
double	*eder11*[ ];
int	*im1*;
int	*im2*;
int	*idim*;
int	*ipar*;
double	*epar1*[ ];
double	*epar2*[ ];
double	*eeps*[ ];
int	*nend*[ ];
int	*iopen1*;
int	*iopen2*;
double	*edgeps*[ ];
int	*iopt*;
int	*itmax*;
SISLSurf	***rs*;
double	*emxerr*[ ];
int	**jstat*;

ARGUMENTS

Input Arguments:

ep	-	Array (length idim*im1*im2) containing the points to be approximated.
etang1	-	Array (length idim*im1*im2) containing the derivatives (tangents) in the first parameter-direction at the data-points.
etang2	-	Array (length idim*im1*im2) containing the derivatives (tangents) in the second parameter-direction at the data-points.
eder11	-	Array (length idim*im1*im2) containing the cross (twist) derivatives at the data-points.
im1	-	The no. of points in the first parameter.
im2	-	The no. of points in the second parameter.
idim	-	The no. of components of each input point. The approximation will be a parametric surface situated in idim-dimensional Euclidean space (usually 3).
ipar	-	Flag determining the parametrization of the data points: = 1 : Mean accumulated cord-length parameterization.

		$= 2$: Uniform parametrization. $= 3$: Parametrization given by epar1 and epar2.
epar1	-	Array (length im1) containing a parametrization in the first parameter. (Will only be used if $ipar = 3$).
epar2	-	Array (length im2) containing a parametrization in the second parameter. (Will only be used if $ipar = 3$).
eeps	-	Array (length idim) containing the maximum deviation which is acceptable in each of the idim components of the surface (except possibly along the edges).
nend	-	Array (length 4) giving the no. of derivatives to be kept fixed along each edge of the bilinear interpolant. The numbering of the edges is the same as for edgeps below. All the derivatives of order $< (nend(i)-1)$ will be kept fixed along the edge i. Hence $nend(i) = 0$ indicates that nothing is to be kept fixed along edge i. To be kept fixed here means to have error less than edgeps. In general, it is impossible to remove any knots and keep an edge completely fixed.
iopen1	-	Open/closed parameter in first direction. $= 1$: Produce open surface. $= 0$: Produce closed, non-periodic surface if possible. $= -1$: Produce closed, periodic surface NB! The surface will be closed/periodic only if the first and last column of data points are (approximately) equal.
iopen2	-	Open/closed parameter in second direction. $= 1$: Produce open surface. $= 0$: Produce closed, non-periodic surface if possible. $= -1$: Produce closed, periodic surface NB! The surface will be closed/periodic only if the first and last row of data points are (approximately) equal.
edgeps	-	Array (length idim*4) containing the max. deviation from the bilinear interpolant which is acceptable along the edges of the surface. edgeps(1,i):edgeps(idim,i) gives the tolerance along the edge corresponding to the i-th parameter having one of it's extremal-values. $i = 1$: min value of first parameter. $i = 2$: max value of first parameter. $i = 3$: min value of second parameter. $i = 4$: max value of second parameter. edgeps(kp,i) will only have significance if $nend(i) > 0$.
iopt	-	Flag indicating the order in which the data reduction is to be performed: $= 1$: Remove knots in parameter 1 only. $= 2$: Remove knots in parameter 2 only. $= 3$: Remove knots first in parameter 1 and then in 2. $= 4$: Remove knots first in parameter 2 and then in 1.
itmax	-	Max. no. of iterations in the data reduction.

Output Arguments:

rs - Pointer to surface.

emxerr - Array (length idim) (allocated outside this routine.) containing an upper bound for the error comitted in each component during the data reduction.

jstat - Status messages
> 0 : Warning.
$= 0$: Ok.
< 0 : Error.

EXAMPLE OF USE

```
{
    double      ep[6000];  /* Spatial dimension times number of points.
                                    Must be defined */
    double      etang1[6000];  /* Spatial dimension times number of points.
                                    Must be defined */
    double      etang2[6000];  /* Spatial dimension times number of points.
                                    Must be defined */
    double      eder11[6000];  /* Spatial dimension times number of points.
                                    Must be defined */
    int         im1 = 100;
    int         im2 = 20;
    int         idim = 3;
    int         ipar = 3;
    double      epar1[100];   /* Must be defined, used when ipar = 3 */
    double      epar2[20];   /* Must be defined, used when ipar = 3
    double      eeps[3];   /* Must be defined */
    int         nend[4];   /* Must be defined */
    int         iopen1 = 1;
    int         iopen2 = 1;
    double      edgeps[12];/* Spatial dimension times number of edges.
                                    Must be defined */
    int         iopt = 1;
    int         itmax = 7;
    SISLSurf    *rs = NULL;
    double      emxerr[3]; /* Spatial dimension */
    int         jstat = 0;
    ...
    s1967(ep, etang1, etang2, eder11, im1, im2, idim, ipar, epar1, epar2, eeps,
          nend, iopen1, iopen2, edgeps, iopt, itmax, &rs, emxerr, &jstat);
    ...
}
```

11.2.4 Degree reduction: B-spline surface as input.

NAME

s1968 - To compute a cubic tensor-product spline approximation to a given tensor product spline surface of arbitrary order, with error less than eeps in each of the idim components. The error in continuity over the start and end of a closed or periodic surface is only guaranteed to be within edgeps.

SYNOPSIS

void s1968(*ps*, *eeps*, *nend*, *iopen1*, *iopen2*, *edgeps*, *iopt*, *itmax*, *rs*, *jstat*)

SISLSurf	**ps*;
double	*eeps*[ ];
int	*nend*[ ];
int	*iopen1*;
int	*iopen2*;
double	*edgeps*[ ];
int	*iopt*;
int	*itmax*;
SISLSurf	***rs*;
int	**jstat*;

ARGUMENTS

Input Arguments:

ps - Pointer to surface.

eeps - Array (length idim) containing the max. permissible deviation of the approximation from the given data points, in each of the components. More specifically, the approximation will not deviate more than eeps(kdim) in component no. kdim, from the bilinear approximation to the data.

nend - Array (length 4) giving the no. of derivatives to be kept fixed along each edge of the bilinear interpolant. The numbering of the edges is the same as for edgeps below. All the derivatives of order $< (nend(i)-1)$ will be kept fixed along the edge i. Hence $nend(i) = 0$ indicates that nothing is to be kept fixed along edge i. To be kept fixed here means to have error less than edgeps. In general, it is impossible to remove any knots and keep an edge completely fixed.

iopen1 - Open/closed parameter in first direction.
$= 1$: Produce open surface.
$= 0$: Produce closed, non-periodic surface if possible.
$= -1$: Produce closed, periodic surface
NB! The surface will be closed/periodic only if the first and last column of data points are (approximately) equal.

iopen2 - Open/closed parameter in second direction.
$= 1$: Produce open surface.
$= 0$: Produce closed, non-periodic surface if possible.
$= -1$: Produce closed, periodic surface
NB! The surface will be closed/periodic only if the first and last row of data points are (approximately) equal.

edgeps - Array (length idim*4) containing the max. deviation from the bilinear interpolant which is acceptable along the edges of the surface. edgeps(1,i):edgeps(idim,i) gives the tolerance along the edge corresponding to the i-th parameter having one of it's extremal-values.

$i = 1$: min value of first parameter.
$i = 2$: max value of first parameter.
$i = 3$: min value of second parameter.
$i = 4$: max value of second parameter.
edgeps(kp,i) will only have significance if $nend(i) > 0$.

iopt - Flag indicating the order in which the data-reduction is to be performed:

= 1: Remove knots in parameter 1 only.
= 2: Remove knots in parameter 2 only.
= 3: Remove knots first in parameter 1 and then in 2.
= 4: Remove knots first in parameter 2 and then in 1.

itmax - Max. no. of iterations in the data-reduction..

Output Arguments:

rs - Pointer to surface.
jstat - Status messages
> 0 : Warning.
$= 0$: Ok.
< 0 : Error.

EXAMPLE OF USE

```
{
    SISLSurf    *ps;  /*Must be defined */
    double      eeps[3];  /*Must be defined */
    int         nend[4]; /*Must be defined */
    int         iopen1 = 1;
    int         iopen2 = -1;
    double      edgeps[12];/* Spatial dimension times number of edges.
                                  Must be defined */
    int         iopt = 4;
    int         itmax = 7;
    SISLSurf    *rs = NULL;
    int         **jstat = 0;
    ...
    s1968(ps, eeps, nend, iopen1, iopen2, edgeps, iopt, itmax, &rs, &jstat);
    ...
}
```

Chapter 12

Tutorial programs

This release of SISL is bundled with a number of sample programs which are intended to make the user more familiar with the use of the API, as well as demonstrating some of its capabilities.

12.1 Compiling the programs

The default cmake setup is not to compile example programs, the stream library and the viewer. To enable compilation of the example programs the cmake call must be extended with -Dsisl_COMPILE_EXAMPLES=ON. This option also enables compilation of the streaming library. With ccmake compile options are changed pressing enter. In cmake-gui compilation of the examples is invoked by ticking the appropriate box. Compilation and linking is performed with the call

```
$ make example01
```

The example programs and the code for the streaming library is written in C++.

12.2 Description and commentaries on the sample programs

The example programs are named `example01` through `example15`. Each of the program demonstrates the use of a single or a couple of SISL functions. The programs produces output files that contain geometric objects in the `Go`-format, which can then be visualised by the provided viewer. These objects can also be visualized in the viewer belonging to the GoTools library.

To keep things as simple as possible, the example programs (with the exception of `example15`) take no command line arguments. Instead, upon execution they inform the user about what they are about to do, and which files will be read from and written to. The names of the input and output files are hard-coded in each example, but the user can experiment by changing the name of these files if she wants to. Several of the sample programs rely upon files generated by earlier examples, so the user should make sure she runs through them in chronological order.

12.2.1 example01.C

What it does

This program demonstrates how to directly specify a spline curve by providing the position of control points and a knotvector (parametrization). It generates such a curve by using hard-coded values as input to the SISL `newCurve` routine.

What it demonstrates

1. How control points and knotvectors are specified in memory.
2. How to use the `newCurve` routine.
3. How to clean up memory using `freeCurve`.

Input/output

The program takes no input files.
The program generates the files `example1_curve.g2` and `example1_points.g2`. The former contains the curve object and the latter contains the control points, expressed in the `Go`-format.

12.2.2 example02.C

What it does

This program demonstrates one of the simplest *interpolation* cases for spline curves in SISL. A sequence of 6 3D-points are provided (hardcoded), and the routine generates a spline curve that fits exactly through these points. Note that this is a simple example of a more general routine, which can also take into consideration tangents, end point conditions, etc.

What it demonstrates

1. The use of the SISL routine `s1356` for interpolating points with a curve.

Input/output

The program takes no input files.
The program generates the file `example2_points.g2` and `example2_curve.g2`. The first file contains the points to be interpolated, and the second file contains the generated curve.

12.2.3 example03.C

What it does

This program creates a so-called *blend-curve* between two other curves, creating a smooth connection between these. In this program, the blend curve connects the *end points* of the two other curves, but in its generality, the routine can be used to create blend curves connecting to any point on the other curves.

What it demonstrates

1. What a blend curve is and how it can be specified.
2. The use of the SISL routine `s1606`, which computes the blend curve.
3. The use of the SISL routine `s1227`, which evaluates points (and derivatives) on a spline curve.
4. How to directly access data members of the `SISLCurve` struct.

Input/output

The program takes as input the files `example1_curve.g2` and `example2_points.g2`, which are respectively generated by the programs `example01` and `example02`. The generated blend curve will be saved to the file `example3_curve.g2`.

12.2.4 example04.C

What it does

This program generates an *offset curve* from another curve. An offset curve is specified as having a fixed distance in a specified direction from the original curve. The generated offset curve will not be exact, as this would in general be impossible using a spline-function. We can however obtain an approximation within a user-specified tolerance.

What it demonstrates

1. What an offset curve is and how it can be specified.
2. The way in which many SISL routines deal with geometric tolerances.
3. The use of the SISL routine `s1360`, which computes the offset curve within a specified, geometric tolerance.

Input/output

The original curve is read from the file `example1_curve.g2`, which is generated by the program `example01`. The resulting approximation of the offset curve will be written to the file `example4_curve.g2`.

12.2.5 example05.C

What it does

This program generates a family of conic section curves, which are represented as rational splines. Conic sections can be *exactly* represented with such splines, so no geometric tolerance specification is needed. The program will generate three ellipse segments, one parabola segment and three hyperbola segments, based on internal, hard-coded data.

What it demonstrates

1. The use of the SISL routine `s1011` to generate all kinds of conic sections.
2. The important fact that conic sections can be exactly represented by rational splines.
3. How a single *shape* parameter can specify whether the generated curve will be an ellipse, a parabola or a hyperbola.

Input/output

The program takes no input files.
The program generates the file `example5_curve.g2` which contains all the generated curves.

12.2.6 example06.C

What it does

This program generates two curves (from internal, hardcoded data), and computes their intersections. Computation of intersections is an extremely important part of SISL, although the intersection of two curves is a minor problem in this respect.

What it demonstrates

1. The use of the SISL routine `s1857` for computing the intersection points between two given spline curves.
2. Underlines the fact that the detected intersection points are returned as parameter values, and have to be evaluated in order to find their 3D positions.
3. How to clean up an array of intersection curves (SISLIntcurve), although, in this example, this array will already be empty.

Input/output

The program takes no input files (the data for the curves is hard-coded).
The generated curves will be written to the files `example6_curve_1.g2` and `example6_curve_2.g2`. The intersection point positions will be written to the file `example6_isectpoints.g2`.

12.2.7 example07.C

What it does

This is a very short and simple program that calculates the arc length of a curve.

What it demonstrates

1. The use of the SISL routine `s1240` for computing the length of a spline curve.

Input/output

The curve whose length is calculated is read from the file `example6_curve_1`, which has been generated by the sample program `example06`. The calculated length will be written to standard output.

12.2.8 example08.C

What it does

This program generates two non-intersecting spline curves (from internal, hard-coded cata), and computes their mutual closest point. The call is very similar to the one in `example06`, where we wanted to compute curve intersections.

What it demonstrates

1. The use of the SISL routine `s1955` for locating the closest points of two curves.

Input/output

As the curves are specified directly by internal data, no input files are needed. The two generated curves will be saved to the two files `example8_curve_1.g2` and `example8_curve_2.g2`. The closest points will be written to the file `example8_closestpoints.g2`.

12.2.9 example09.C

What it does

This program generates four different surfaces interpolating an array of spatial points. The surfaces have different spline order, so that even though they interpolate the same points, they have different shapes.

What it demonstrates

1. The use of the SISL routine `s1537` for generating an interpolating surface to a grid of points.
2. The effect of the spline order on the interpolating surface.

Input/output

The program takes no input files (the points to be interpolated are hard-coded). The program creates two data files: `example9_points.g2`, which contains all the interpolated points, and `example9_surf.g2`, which contains the four generated surfaces.

12.2.10 example10.C

What it does

This program generates a sequence of spline curves. Moreover, it generates a *lofted surface* interpolating these curves. The lofted surface has the original sequence of curves as isoparametric curves in one of its parameters.

What it demonstrates

1. The use of the SISL routine `s1538` for generating lofted spline surfaces.
2. Gives a good example of what a lofted surface looks like.

Input/output

The program takes no input files (the curves to be interpolated are hard-coded).
The program creates two data files: `example10_curves.g2`, containing the generated sequence of curves, and `example10_surf.g2`, containing the lofted surface.

12.2.11 example11.C

What it does

This program generates a cylindrical surface with an oval base.

What it demonstrates

1. The use of the SISL routine `s1021` for generating cylindrical surfaces.
2. The fact that cylindrical surfaces are exactly representable as rational spline surfaces.

Input/output

The program takes no input files.
The program creates one data file: `example11_surf.g2`, containing the generated surface.

12.2.12 example12.C

What it does

This program finds the intersection points between a curve and a surface. The curve and the surface in question have been defined by previous example programs.

What it demonstrates

1. The use of the SISL routine `s1858` for computing intersection points between a curve and a surface.

Input/output

The curve and the surface in question are read from the files `example4_curve.g2` and `example10_surf.g2`, respectively generated by the sample programs `example04` and `example10`. The found intersections are written to the file `example12_isectpoints.g2`.

12.2.13 example13.C

What it does

This program computes all intersection curves between two surfaces. This is a nontrivial task in geometrical modeling. The problem is twofold. The first problem is to determine the number of intersections, and their topology. The region of an intersection can be either a point, a curve and a surface. In the two latter cases, the shape of the region can usually only be approximated. We do not know a priori how many separate intersections there exists between two surfaces, so we have to look systematically for them. Intersection curves can take the form of closed loops on the interior of the surfaces, of curves running from the surface edges, or of curves meeting in a singularity. When we have successfully determined the topology of the intersections, the second problem is to determine their acutal shape. This is usually done by *marching techniques*. However, we may run into problems with 'degenerated' surfaces, or surfaces being close to coplanar in the intersection.

What it demonstrates

1. The use of the SISL routine `s1859` for determining the topology of the intersections between two spline surfaces.
2. The use of the SISL routine `s1310` for marching out the detected curves after their topologies have been determined.

Input/output

The two surfaces have been generated by the previous sample programs `example10` and `example11`, and can be found in the files `example10_surf.g2` and `example11_surf.g2`. The resulting intersection curves will be written to the file `example13_isectcurves.g2`.

12.2.14 example14.C

What it does

This program demonstrates one of the data reduction techniques of SISL. As input data, it first generates a dense point set by sampling from a (predefined) spline curve. Then, using this data, it attempts to generate a new spline curve that fits closely to these samples, while using as few control points as possible. Since we know that in this case the data points come from a simple spline curve, it should be no surprise that the generated curve will have approximately the same expression as the sampled curve (and thus reduce the quantity of data substantially compared to what is needed to store the points). However, data reduction can be obtained on any sufficiently smooth point set, even if it originates from other processes.

What it demonstrates

1. The use of the SISL routine `s1961` for generating approximating spline curves through a set of data, using as few control points as possible.

2. The power of this data reduction technique on smooth point data.

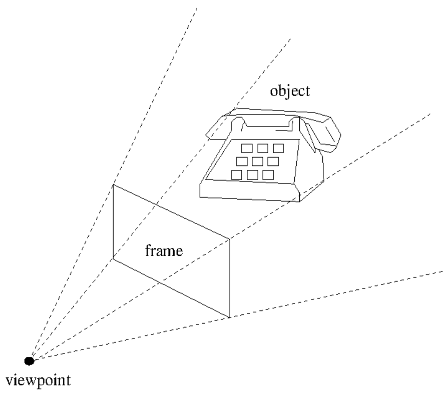
Input/output

The program takes no input files, as the curve to be sampled from is hard-coded. The sampled points will be written to the file `example14_points.g2`, and the obtained curve will be stored in `example14_curve.g2`.

12.2.15 example15.C

What it does

This is the last of the sample programs, and by far the most complicated. It aims not only to demonstrate a certain feature of SISL, but to show how this feature can be used for a purpose (raytracing). Moreover, it demonstrates two ways of achieveing this, one slow and robust method and one rapid but fragile method.
Raytracing can be seen as the process of determining what an object 'looks like' from a certain viewpoint, through a certain 'window', as illustrated below. Lines ('rays') are extended from the viewpoint through a dense grid of points



on the window, and checked for intersection with the object. If such an intersection exists, it should be registered as a point on the object 'visible' from the viewpoint. In computer graphics, these points are projected back on the window, which becomes a 2D image that can be displayed on the computer screen. For our purposes, we refrain from doing this projection, and store the full 3D coordinates of the detected point.
Note that a ray may intersect the object more than once. In these cases, the intersection point closest to the viewpoint is chosen, as the other points are

'hidden' by it. As mentioned above, there are two raytracing routines in this example program. The robust routine calculates all possible intersection points for each ray, and then choses the nearest one. This should always work, but can be slow since no information is re-used. When we have found an intersection point for a given ray, we can usually expect that the next, neighouring ray will intersect in a point close to the one already found. If this is the case, it would be speedier to use a local algorithm that converges on the intersection point quickly given a good initial guess. This is the basis for our 'quick' routine. This routine uses the robust raytracing algorithm to find the first point on a surface, and then it switches over to the fast method as long as it is possible to do so. However, since the quick method never finds more than one intersection point, and since a ray may generally intersect an object more than once, we have no guarantee that the point found is the one truly visible from the viewpoint. There are some checking procedures that make things better, but we still have no guarantee. If the user inspects the results obtained, he will notice this problem even on the simple example given here. In general, it can be said that the rapid algorithm should only be used in some special cases, where we know for a fact that any ray from the viewpoint will not intersect the surface more than once.

This is the only of the example programs that can be run with a command line argument. If the first argument is `q`, then the quick raytracing routine will be invoked. Else, the robust and slow routine is used.

What it demonstrates

1. The basic setting and principe of a raytracer, with a defined viewpoint, window and intersection with rays.
2. The use of the SISL routine `s1856`, which calculates all intersections between a spline surface and a line.
3. The use of the SISL routine `s1518`, which converges to an intersection between a spline surface and a line, given a good initial guess.

Input/output

The surface to be raytraced is read from the file `example10_surf.g2`, generated by the `example10` program. The other parameters necessary for the raytracing are hard- coded (viewpoint, view window, resolution, etc.). The resulting points are written to the file `example15_points.g2`.

Chapter 13

The object viewer program

13.1 General

The object viewer program bundled with this distribution of SISL is intended to be a simple but handy tool for visualising curves and surfaces generated by SISL. The supported file format is the `Go` format, which is a simple, ASCII-based format defined by SINTEF. The viewer is based on OpenGL. An alternative viewer with a more evolved user interface, but also more dependencies can be found in the library GoTools also provided by SINTEF Mathematics and Cybernetics. The object(s) to be viewed are for this viewer specified on the command line when starting the program. Once the program is started, the user cannot open other files containing SISL objects. The viewer allows the user to zoom, pan and rotate the objects with the mouse, and some other useful commands can be accessed through the keyboard.

In the viewer window, several curves and surfaces can be displayed simultaneously. At all times, exactly *one* surface and *one* curve are defined as being *active* (the other ones being *passive*). With keyboard commands, the user can change the currently active surface/curve. An object just becoming active will flash for a few seconds. With other keyboard commands, the user can *enable/disable* surfaces and curves. This refers to turning the display of these objects on or off. For details, refer to the section on keyboard commands.

13.2 Compiling the viewer

The default cmake setup is not to compile example programs, the stream library and the viewer. To enable compilation of the example programs the cmake call must be extended with -Dsisl_COMPILE_VIEWER=ON. This option also enables compilation of the streaming library. With ccmake compile options are changed pressing enter. In cmake-gui compilation of the viewer is invoked by ticking the appropriate box. Compilation and linking is performed with the call

```
$ make sisl_view_demo
```

The viewer is written in C++.

13.3 Command line arguments

When starting up the viewer, the options listed below can be used. If no option is specified, a short text listing the available options is printed on screen.

- `s` *filename* - view the surface(s) contained in the file *filename*. Note: this command line option can be used repetitively if the user wants to inspect several surfaces at once.
- `c` *filename* - view the curve(s) contained in the file *filename*. Note: this command can be used repetitively if the user wants to inspect several curves at once.
- `p` *filename* - view the point(s) contained in the file *filename*. Note: this command line option can be used repetitively if the user wants to inspect several surfaces at once.
- `r` *integer* set surface refinement factor (number of facets in each direction on the surface). Default value is 100. Higher values gives smoother drawing of the surface. NB: this option has to *precede* the 's' option!
- `e` *string* the string contains keypresses to execute directly upon start (see the section on keyboard control keys for details).
- `hotkeys` does not start the viewer, but displays a list of keyboard commands that can be used when viewing.

A file can contain one or several curves, or one or several surfaces. Files containing both curves and surfaces are not supported. The viewer can read several files to be viewed at once. On the command line, each "curve" file should be preceded with the letter 'c', and each "surface" file should be preceded with the letter 's'. After launch, all the objects contained in the given files are shown simultaneously. The user can disable the view of certain curves and surfaces if he or she wants to.

13.4 User controls

After program launch, the viewing of curves and surfaces can be controlled with the mouse and keyboard. The mouse is used to define viewing angle, direction and zoom factor, while keyboard keys are used to turn on/off objects and to change certain view parameters.

13.4.1 Mouse commands

It is assumed that a 3-button mouse is used. By dragging the mouse while holding down the *left button*, the user can rotate the current view in an intuitive way. By dragging with a certain speed, the view will continue to rotate even after the left button is released. The *middle button* is used for zooming. Hold down this button and move the mouse forwards and backwards in order to zoom in and out. Holding down the *right button* while dragging the mouse moves the view up and down.

13.4.2 Keyboard commands

The available keyboard commands are:

- q - quit the viewer program
- <space> - change the currently active curve (cycles through each of them)
- <tab - change the currently active surface (cycles through each of them)
- w - turn on/off the wireframe display for surfaces
- B - toggle between black and white color for backgrounds
- A - toggle drawing of coordinate axes on/off
- S - toggle drawing of surfaces
- e - toggle visibility of currently active surface
- a - make all loaded surfaces visible
- d - hide all surfaces except the currently active one
- <ctrl>-e - toggle visibility of currently active curve
- <ctrl>-a - make all loaded curves visible
- <ctrl>-d - hide all curves except the currently active one
- O - center all objects around origo, and rescale objects so that they fit inside the unit volume (does not preserve aspect ratio)
- o - center all objects around origo, no rescaling
- + - increase thickness of axes
- - - decrease thickness of axes
- > - increase size of points
- < - decrease size of points
- / - decrease length of axes
- <esc>-w-[n] - store viewpoint in slot [n], where [n] is a number from 0 to 9. The viewpoint will be saved to file, and can such be preserved from one session to another.
- <esc>-r-[n] - load a previously saved viewpoint from slot [n], where [n] is a number from 0 to 9.

Chapter 14

Appendix: Error Codes

For reference, here is a list of the error codes used in SISL. They can be useful for diagnosing problems encountered when calling SISL routines. However please note that a small number of SISL routines use their own convention.

```
Label Value  Description
--------------------------------------------------------------------------------
err101 -101  Error in memory allocation.

err102 -102  Error in input. Dimension less than 1.

err103 -103  Error in input. Dimension less than 2.

err104 -104  Error in input. Dimension not equal 3.

err105 -105  Error in input. Dimension not equal 2 or 3.

err106 -106  Error in input. Conflicting dimensions.

err107 -107

err108 -108  Error in input. Dimension not equal 2.

err109 -109  Error in input. Order less than 2.

err110 -110  Error in Curve description. Order less than 1.

err111 -111  Error in Curve description. Number of vertices less than order.

err112 -112  Error in Curve description. Error in knot vector.

err113 -113  Error in Curve description. Unknown kind of Curve.

err114 -114  Error in Curve description. Open Curve when expecting closed.

err115 -115  Error in Surf description. Order less than 1.
```

err116	-116	Error in Surf description. Number of vertices less than order.
err117	-117	Error in Surf description. Error in knot vector.
err118	-118	Error in Surf description. Unknown kind of Surf.
err119	-119	
err120	-120	Error in input. Negative relative tolerance.
err121	-121	Error in input. Unknown kind of Object.
err122	-122	Error in input. Unexpected kind of Object found.
err123	-123	Error in input. Parameter direction does not exist.
err124	-124	Error in input. Zero length parameter interval.
err125	-125	
err126	-126	
err127	-127	Error in input. The whole curve lies on axis.
err128	-128	
err129	-129	
err130	-130	Error in input. Parameter value is outside parameter area.
err131	-131	
err132	-132	
err133	-133	
err134	-134	
err135	-135	Error in data structure. Intersection point exists when it should not.
err136	-136	Error in data structure. Intersection list exists when it should not.
err137	-137	Error in data structure. Expected intersection point not found.
err138	-138	Error in data structure. Wrong number of intersections on edges/endpoints.

err139 -139	Error in data structure. Edge intersection does not lie on edge/endpoint.
err140 -140	Error in data structure. Intersection interval crosses subdivision line when not expected to.
err141 -141	Error in input. Illegal edge point requested.
err142 -142	
err143 -143	
err144 -144	Unknown kind of intersection curve.
err145 -145	Unknown kind of intersection list (internal format).
err146 -146	Unknown kind of intersection type.
err147 -147	
err148 -147	
err149 -149	
err150 -150	Error in input. NULL pointer was given.
err151 -151	Error in input. One or more illegal input values.
err152 -152	Too many knots to insert.
err153 -153	Lower level routine reported error. SHOULD use label "error".
err154 -154	
err155 -155	
err156 -156	Illegal derivative requested. Change this label to err178.
err157 -157	
err158 -158	Intersection point outside Curve.
err159 -159	No of vertices less than 1. SHOULD USE err111 or err116.
err160 -160	Error in dimension of interpolation problem.
err161 -161	Error in interpolation problem.
err162 -162	Matrix may be noninvertible.

err163 -163 Matrix part contains diagonal elements.

err164 -164 No point conditions specified in interpolation problem.

err165 -165 Error in interpolation problem.

err166 -166

err167 -167

err168 -168

err169 -169

err170 -170 Internal error: Error in moving knot values.

err171 -171 Memory allocation failure: Could not create curve or surface.

err172 -172 Input error, inarr < 1 || inarr > 3.

err173 -173 Direction vector zero length.

err174 -174 Degenerate condition.

err175 -175 Unknown degree/type of implicit surface.

err176 -176 Unexpected iteration situation.

err177 -177 Error in input. Negative step length requested.

err178 -178 Illegal derivative requested.

err179 -179 No. of Curves < 2.

err180 -180 Error in torus description.

err181 -181 Too few points as input.

err182 -182

err183 -183 Order(s) specified to low.

err184 -184 Negative tolerance given.

err185 -185 Only degenerate or singular guide points.

err186 -186 Special error in traversal of curves.

err187 -187 Error in description of input curves.

```
err188 -188

err189 -189

err190 -190  Too small array for storing Curve segments.

err191 -191  Error in inserted parameter number.

err192 -192

err193 -193

err194 -194

err195 -195

err196 -196

err197 -197

err198 -198

err199 -199  Error in vectors?
```

Appendix A

GNU AFFERO GENERAL PUBLIC LICENSE

Preamble

The GNU Affero General Public License is a free, copyleft license for software and other kinds of works, specifically designed to ensure cooperation with the community in the case of network server software.

The licenses for most software and other practical works are designed to take away your freedom to share and change the works. By contrast, our General Public Licenses are intended to guarantee your freedom to share and change all versions of a program–to make sure it remains free software for all its users.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for them if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs, and that you know you can do these things.

Developers that use our General Public Licenses protect your rights with two steps: (1) assert copyright on the software, and (2) offer you this License which gives you legal permission to copy, distribute and/or modify the software.

A secondary benefit of defending all users' freedom is that improvements made in alternate versions of the program, if they receive widespread use, become available for other developers to incorporate. Many developers of free software are heartened and encouraged by the resulting cooperation. However, in the case of software used on network servers, this result may fail to come about. The GNU General Public License permits making a modified version and letting the public access it on a server without ever releasing its source code to the public.

The GNU Affero General Public License is designed specifically to ensure that, in such cases, the modified source code becomes available to the community. It requires the operator of a network server to provide the source code of the modified version running there to the users of that server. Therefore, public use of a modified version, on a publicly accessible server, gives the public access to the source code of the modified version.

An older license, called the Affero General Public License and published by Affero, was designed to accomplish similar goals. This is a different license, not a version of the Affero GPL, but Affero has released a new version of the Affero GPL which permits relicensing under this license.

The precise terms and conditions for copying, distribution and modification follow.

Terms and Conditions

0. Definitions.

   "This License" refers to version 3 of the GNU Affero General Public License.

   "Copyright" also means copyright-like laws that apply to other kinds of works, such as semiconductor masks.

   "The Program" refers to any copyrightable work licensed under this License. Each licensee is addressed as "you". "Licensees" and "recipients" may be individuals or organizations.

   To "modify" a work means to copy from or adapt all or part of the work in a fashion requiring copyright permission, other than the making of an exact copy. The resulting work is called a "modified version" of the earlier work or a work "based on" the earlier work.

   A "covered work" means either the unmodified Program or a work based on the Program.

   To "propagate" a work means to do anything with it that, without permission, would make you directly or secondarily liable for infringement under applicable copyright law, except executing it on a computer or modifying a private copy. Propagation includes copying, distribution (with or without modification), making available to the public, and in some countries other activities as well.

   To "convey" a work means any kind of propagation that enables other parties to make or receive copies. Mere interaction with a user through a computer network, with no transfer of a copy, is not conveying.

   An interactive user interface displays "Appropriate Legal Notices" to the extent that it includes a convenient and prominently visible feature that (1) displays an appropriate copyright notice, and (2) tells the user that there is no warranty for the work (except to the extent that warranties are provided), that licensees may convey the work under this License, and how to view a copy of this License. If the interface presents a list of user commands or options, such as a menu, a prominent item in the list meets this criterion.

1. Source Code.

   The "source code" for a work means the preferred form of the work for making modifications to it. "Object code" means any non-source form of a work.

   A "Standard Interface" means an interface that either is an official standard defined by a recognized standards body, or, in the case of interfaces specified for a particular programming language, one that is widely used among developers working in that language.

   The "System Libraries" of an executable work include anything, other than the work as a whole, that (a) is included in the normal form of packaging a Major Component, but which is not part of that Major Component, and (b) serves only to enable use of the work with that Major Component, or to implement a Standard Interface for which an implementation is available to the public in source code form. A "Major Component", in this context, means a major essential component (kernel, window system, and so on) of the specific operating system (if any) on which the executable work runs, or a compiler used to produce the work, or an object code interpreter used to run it.

The "Corresponding Source" for a work in object code form means all the source code needed to generate, install, and (for an executable work) run the object code and to modify the work, including scripts to control those activities. However, it does not include the work's System Libraries, or general-purpose tools or generally available free programs which are used unmodified in performing those activities but which are not part of the work. For example, Corresponding Source includes interface definition files associated with source files for the work, and the source code for shared libraries and dynamically linked subprograms that the work is specifically designed to require, such as by intimate data communication or control flow between those subprograms and other parts of the work.

The Corresponding Source need not include anything that users can regenerate automatically from other parts of the Corresponding Source.

The Corresponding Source for a work in source code form is that same work.

2. Basic Permissions.

   All rights granted under this License are granted for the term of copyright on the Program, and are irrevocable provided the stated conditions are met. This License explicitly affirms your unlimited permission to run the unmodified Program. The output from running a covered work is covered by this License only if the output, given its content, constitutes a covered work. This License acknowledges your rights of fair use or other equivalent, as provided by copyright law.

   You may make, run and propagate covered works that you do not convey, without conditions so long as your license otherwise remains in force. You may convey covered works to others for the sole purpose of having them make modifications exclusively for you, or provide you with facilities for running those works, provided that you comply with the terms of this License in conveying all material for which you do not control copyright. Those thus making or running the covered works for you must do so exclusively on your behalf, under your direction and control, on terms that prohibit them from making any copies of your copyrighted material outside their relationship with you.

   Conveying under any other circumstances is permitted solely under the conditions stated below. Sublicensing is not allowed; section 10 makes it unnecessary.

3. Protecting Users' Legal Rights From Anti-Circumvention Law.

   No covered work shall be deemed part of an effective technological measure under any applicable law fulfilling obligations under article 11 of the WIPO copyright treaty adopted on 20 December 1996, or similar laws prohibiting or restricting circumvention of such measures.

When you convey a covered work, you waive any legal power to forbid circumvention of technological measures to the extent such circumvention is effected by exercising rights under this License with respect to the covered work, and you disclaim any intention to limit operation or modification of the work as a means of enforcing, against the work's users, your or third parties' legal rights to forbid circumvention of technological measures.

4. Conveying Verbatim Copies.

   You may convey verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice; keep intact all notices stating that this License and any non-permissive terms added in accord with section 7 apply to the code; keep intact all notices of the absence of any warranty; and give all recipients a copy of this License along with the Program.

   You may charge any price or no price for each copy that you convey, and you may offer support or warranty protection for a fee.

5. Conveying Modified Source Versions.

   You may convey a work based on the Program, or the modifications to produce it from the Program, in the form of source code under the terms of section 4, provided that you also meet all of these conditions:

   (a) The work must carry prominent notices stating that you modified it, and giving a relevant date.

   (b) The work must carry prominent notices stating that it is released under this License and any conditions added under section 7. This requirement modifies the requirement in section 4 to "keep intact all notices".

   (c) You must license the entire work, as a whole, under this License to anyone who comes into possession of a copy. This License will therefore apply, along with any applicable section 7 additional terms, to the whole of the work, and all its parts, regardless of how they are packaged. This License gives no permission to license the work in any other way, but it does not invalidate such permission if you have separately received it.

   (d) If the work has interactive user interfaces, each must display Appropriate Legal Notices; however, if the Program has interactive interfaces that do not display Appropriate Legal Notices, your work need not make them do so.

   A compilation of a covered work with other separate and independent works, which are not by their nature extensions of the covered work, and which are not combined with it such as to form a larger program, in or on a volume of a storage or distribution medium, is called an "aggregate" if the compilation and its resulting copyright are not used to limit the access or legal rights of the compilation's users beyond what the individual works permit. Inclusion of a covered work in an aggregate does not cause this License to apply to the other parts of the aggregate.

6. Conveying Non-Source Forms.

   You may convey a covered work in object code form under the terms of sections 4 and 5, provided that you also convey the machine-readable Corresponding Source under the terms of this License, in one of these ways:

   (a) Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by the Corresponding Source fixed on a durable physical medium customarily used for software interchange.

   (b) Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by a written offer, valid for at least three years and valid for as long as you offer spare parts or customer support for that product model, to give anyone who possesses the object code either (1) a copy of the Corresponding Source for all the software in the product that is covered by this License, on a durable physical medium customarily used for software interchange, for a price no more than your reasonable cost of physically performing this conveying of source, or (2) access to copy the Corresponding Source from a network server at no charge.

   (c) Convey individual copies of the object code with a copy of the written offer to provide the Corresponding Source. This alternative is allowed only occasionally and noncommercially, and only if you received the object code with such an offer, in accord with subsection 6b.

   (d) Convey the object code by offering access from a designated place (gratis or for a charge), and offer equivalent access to the Corresponding Source in the same way through the same place at no further charge. You need not require recipients to copy the Corresponding Source along with the object code. If the place to copy the object code is a network server, the Corresponding Source may be on a different server (operated by you or a third party) that supports equivalent copying facilities, provided you maintain clear directions next to the object code saying where to find the Corresponding Source. Regardless of what server hosts the Corresponding Source, you remain obligated to ensure that it is available for as long as needed to satisfy these requirements.

   (e) Convey the object code using peer-to-peer transmission, provided you inform other peers where the object code and Corresponding Source of the work are being offered to the general public at no charge under subsection 6d.

   A separable portion of the object code, whose source code is excluded from the Corresponding Source as a System Library, need not be included in conveying the object code work.

   A "User Product" is either (1) a "consumer product", which means any tangible personal property which is normally used for personal, family, or household purposes, or (2) anything designed or sold for incorporation into a dwelling. In determining whether a product is a consumer product, doubtful cases shall be resolved in favor of coverage. For a particular product received by a particular user, “normally used” refers to a typical or common use of that class of product, regardless of the status of the particular user or of the way in which the particular user actually uses, or expects or is expected to use, the product. A product is a consumer product regardless of whether the product has substantial commercial, industrial or non-consumer uses, unless such uses represent the only significant mode of use of the product.

“Installation Information” for a User Product means any methods, procedures, authorization keys, or other information required to install and execute modified versions of a covered work in that User Product from a modified version of its Corresponding Source. The information must suffice to ensure that the continued functioning of the modified object code is in no case prevented or interfered with solely because modification has been made.

If you convey an object code work under this section in, or with, or specifically for use in, a User Product, and the conveying occurs as part of a transaction in which the right of possession and use of the User Product is transferred to the recipient in perpetuity or for a fixed term (regardless of how the transaction is characterized), the Corresponding Source conveyed under this section must be accompanied by the Installation Information. But this requirement does not apply if neither you nor any third party retains the ability to install modified object code on the User Product (for example, the work has been installed in ROM).

The requirement to provide Installation Information does not include a requirement to continue to provide support service, warranty, or updates for a work that has been modified or installed by the recipient, or for the User Product in which it has been modified or installed. Access to a network may be denied when the modification itself materially and adversely affects the operation of the network or violates the rules and protocols for communication across the network.

Corresponding Source conveyed, and Installation Information provided, in accord with this section must be in a format that is publicly documented (and with an implementation available to the public in source code form), and must require no special password or key for unpacking, reading or copying.

7. Additional Terms.

   “Additional permissions” are terms that supplement the terms of this License by making exceptions from one or more of its conditions. Additional permissions that are applicable to the entire Program shall be treated as though they were included in this License, to the extent that they are valid under applicable law. If additional permissions apply only to part of the Program, that part may be used separately under those permissions, but the entire Program remains governed by this License without regard to the additional permissions.

   When you convey a copy of a covered work, you may at your option remove any additional permissions from that copy, or from any part of it. (Additional permissions may be written to require their own removal in certain cases when you modify the work.) You may place additional permissions on material, added by you to a covered work, for which you have or can give appropriate copyright permission.

Notwithstanding any other provision of this License, for material you add to a covered work, you may (if authorized by the copyright holders of that material) supplement the terms of this License with terms:

(a) Disclaiming warranty or limiting liability differently from the terms of sections 15 and 16 of this License; or

(b) Requiring preservation of specified reasonable legal notices or author attributions in that material or in the Appropriate Legal Notices displayed by works containing it; or

(c) Prohibiting misrepresentation of the origin of that material, or requiring that modified versions of such material be marked in reasonable ways as different from the original version; or

(d) Limiting the use for publicity purposes of names of licensors or authors of the material; or

(e) Declining to grant rights under trademark law for use of some trade names, trademarks, or service marks; or

(f) Requiring indemnification of licensors and authors of that material by anyone who conveys the material (or modified versions of it) with contractual assumptions of liability to the recipient, for any liability that these contractual assumptions directly impose on those licensors and authors.

All other non-permissive additional terms are considered "further restrictions" within the meaning of section 10. If the Program as you received it, or any part of it, contains a notice stating that it is governed by this License along with a term that is a further restriction, you may remove that term. If a license document contains a further restriction but permits relicensing or conveying under this License, you may add to a covered work material governed by the terms of that license document, provided that the further restriction does not survive such relicensing or conveying.

If you add terms to a covered work in accord with this section, you must place, in the relevant source files, a statement of the additional terms that apply to those files, or a notice indicating where to find the applicable terms.

Additional terms, permissive or non-permissive, may be stated in the form of a separately written license, or stated as exceptions; the above requirements apply either way.

8. Termination.

You may not propagate or modify a covered work except as expressly provided under this License. Any attempt otherwise to propagate or modify it is void, and will automatically terminate your rights under this License (including any patent licenses granted under the third paragraph of section 11).

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, you do not qualify to receive new licenses for the same material under section 10.

9. Acceptance Not Required for Having Copies.

   You are not required to accept this License in order to receive or run a copy of the Program. Ancillary propagation of a covered work occurring solely as a consequence of using peer-to-peer transmission to receive a copy likewise does not require acceptance. However, nothing other than this License grants you permission to propagate or modify any covered work. These actions infringe copyright if you do not accept this License. Therefore, by modifying or propagating a covered work, you indicate your acceptance of this License to do so.

10. Automatic Licensing of Downstream Recipients.

    Each time you convey a covered work, the recipient automatically receives a license from the original licensors, to run, modify and propagate that work, subject to this License. You are not responsible for enforcing compliance by third parties with this License.

    An "entity transaction" is a transaction transferring control of an organization, or substantially all assets of one, or subdividing an organization, or merging organizations. If propagation of a covered work results from an entity transaction, each party to that transaction who receives a copy of the work also receives whatever licenses to the work the party's predecessor in interest had or could give under the previous paragraph, plus a right to possession of the Corresponding Source of the work from the predecessor in interest, if the predecessor has it or can get it with reasonable efforts.

    You may not impose any further restrictions on the exercise of the rights granted or affirmed under this License. For example, you may not impose a license fee, royalty, or other charge for exercise of rights granted under this License, and you may not initiate litigation (including a cross-claim or counterclaim in a lawsuit) alleging that any patent claim is infringed by making, using, selling, offering for sale, or importing the Program or any portion of it.

11. Patents.

A "contributor" is a copyright holder who authorizes use under this License of the Program or a work on which the Program is based. The work thus licensed is called the contributor's "contributor version".

A contributor's "essential patent claims" are all patent claims owned or controlled by the contributor, whether already acquired or hereafter acquired, that would be infringed by some manner, permitted by this License, of making, using, or selling its contributor version, but do not include claims that would be infringed only as a consequence of further modification of the contributor version. For purposes of this definition, "control" includes the right to grant patent sublicenses in a manner consistent with the requirements of this License.

Each contributor grants you a non-exclusive, worldwide, royalty-free patent license under the contributor's essential patent claims, to make, use, sell, offer for sale, import and otherwise run, modify and propagate the contents of its contributor version.

In the following three paragraphs, a "patent license" is any express agreement or commitment, however denominated, not to enforce a patent (such as an express permission to practice a patent or covenant not to sue for patent infringement). To "grant" such a patent license to a party means to make such an agreement or commitment not to enforce a patent against the party.

If you convey a covered work, knowingly relying on a patent license, and the Corresponding Source of the work is not available for anyone to copy, free of charge and under the terms of this License, through a publicly available network server or other readily accessible means, then you must either (1) cause the Corresponding Source to be so available, or (2) arrange to deprive yourself of the benefit of the patent license for this particular work, or (3) arrange, in a manner consistent with the requirements of this License, to extend the patent license to downstream recipients. "Knowingly relying" means you have actual knowledge that, but for the patent license, your conveying the covered work in a country, or your recipient's use of the covered work in a country, would infringe one or more identifiable patents in that country that you have reason to believe are valid.

If, pursuant to or in connection with a single transaction or arrangement, you convey, or propagate by procuring conveyance of, a covered work, and grant a patent license to some of the parties receiving the covered work authorizing them to use, propagate, modify or convey a specific copy of the covered work, then the patent license you grant is automatically extended to all recipients of the covered work and works based on it.

A patent license is "discriminatory" if it does not include within the scope of its coverage, prohibits the exercise of, or is conditioned on the non-exercise of one or more of the rights that are specifically granted under this License. You may not convey a covered work if you are a party to an arrangement with a third party that is in the business of distributing software, under which you make payment to the third party based on the extent of your activity of conveying the work, and under which the third party grants, to any of the parties who would receive the covered work from you, a discriminatory patent license (a) in connection with copies of the covered work conveyed by you (or copies made from those copies), or (b) primarily for and in connection with specific products or compilations that contain the covered work, unless you entered into that arrangement, or that patent license was granted, prior to 28 March 2007.

Nothing in this License shall be construed as excluding or limiting any implied license or other defenses to infringement that may otherwise be available to you under applicable patent law.

12. No Surrender of Others' Freedom.

    If conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot convey a covered work so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not convey it at all. For example, if you agree to terms that obligate you to collect a royalty for further conveying from those to whom you convey the Program, the only way you could satisfy both those terms and this License would be to refrain entirely from conveying the Program.

13. Remote Network Interaction; Use with the GNU General Public License.

    Notwithstanding any other provision of this License, if you modify the Program, your modified version must prominently offer all users interacting with it remotely through a computer network (if your version supports such interaction) an opportunity to receive the Corresponding Source of your version by providing access to the Corresponding Source from a network server at no charge, through some standard or customary means of facilitating copying of software. This Corresponding Source shall include the Corresponding Source for any work covered by version 3 of the GNU General Public License that is incorporated pursuant to the following paragraph.

    Notwithstanding any other provision of this License, you have permission to link or combine any covered work with a work licensed under version 3 of the GNU General Public License into a single combined work, and to convey the resulting work. The terms of this License will continue to apply to the part which is the covered work, but the work with which it is combined will remain governed by version 3 of the GNU General Public License.

14. Revised Versions of this License.

    The Free Software Foundation may publish revised and/or new versions of the GNU Affero General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

    Each version is given a distinguishing version number. If the Program specifies that a certain numbered version of the GNU Affero General Public License "or any later version" applies to it, you have the option of following the terms and conditions either of that numbered version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of the GNU Affero General Public

License, you may choose any version ever published by the Free Software Foundation.

If the Program specifies that a proxy can decide which future versions of the GNU Affero General Public License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Program.

Later license versions may give you additional or different permissions. However, no additional obligations are imposed on any author or copyright holder as a result of your choosing to follow a later version.

15. Disclaimer of Warranty.

    THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

16. Limitation of Liability.

    IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MODIFIES AND/OR CONVEYS THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

17. Interpretation of Sections 15 and 16.

    If the disclaimer of warranty and limitation of liability provided above cannot be given local legal effect according to their terms, reviewing courts shall apply local law that most closely approximates an absolute waiver of all civil liability in connection with the Program, unless a warranty or assumption of liability accompanies a copy of the Program in return for a fee.

End of Terms and Conditions

Appendix: How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively state the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

```
<one line to give the program's name and a brief idea of what it does.>

Copyright (C) <textyear>  <name of author>

This program is free software: you can redistribute it and/or modify
it under the terms of the GNU Affero General Public License as published by
the Free Software Foundation, either version 3 of the License, or
(at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU Affero General Public License for more details.

You should have received a copy of the GNU Affero General Public License
along with this program.  If not, see <https://www.gnu.org/licenses/>.
```

Also add information on how to contact you by electronic and paper mail.

If your software can interact with users remotely through a computer network, you should also make sure that it provides a way for users to get its source. For example, if your program is a web application, its interface could display a "Source" link that leads users to an archive of the code. There are many ways you could offer source, and different solutions will be better for different programs; see section 13 for the specific requirements.

You should also get your employer (if you work as a programmer) or school, if any, to sign a "copyright disclaimer" for the program, if necessary. For more information on this, and how to apply and follow the GNU AGPL, see `https://www.gnu.org/licenses/`.