SISL

The SINTEF Spline Library

Reference Manual

(version 4.5)

SINTEF Digital, Mathematics and Cybernetics

January 18, 2021

Contents

Chapter 1

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 5.1.1 and 9.1.1.

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 11 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 in this manual contains 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. To get you started, this chapter contains an Example Program.

SISL is a mature library that is no longer subject to the introduction of new functionality. This version of the library differ from the previous one by bug fixing and an update of the documentation. SINTEF Mathematics and Cybernetics provide, in addition to SISL, GoTools. This is a collection of geometry libraries written in C++ that complements SISL. New geometry software development is integrated into GoTools. Corresponding geometry entities like NURBS curves and surfaces have different representations in SISL and GoTools, but conversion functionality exist and is placed in GoTools as well as a viewer of a set of geometry entities including NURBS curves and surfaces. An overview of the interplay between SISL and GoTools will be provided in the end of this document.

1.1 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.

1.2 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.

1.3 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 1.4.

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.

1.4 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 5.1.1, freeCurve() documented in 5.1.3, s1373() found in Section 7.2.4 and freeIntcrvlist() in 7.1.4.

```
#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!

1.5 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 5.1.

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 of the B-splines.

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

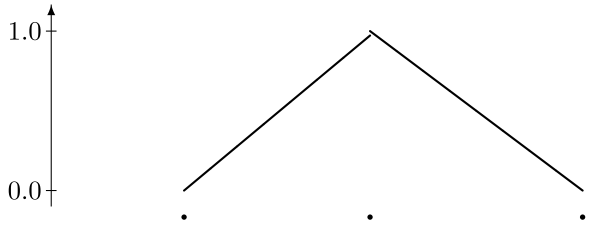


Figure 1.1: A linear B-spline (order 2) defined by three knots.

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: $t_i \leq t_{i+1}$. 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: $n \geq k$.
- There should be k equal knots at the beginning and at the end of the knot vector; that is the knot vector **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.

1.5.1 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 1.1 the three knots are marked as dots. Knots can also be equal as shown in figure 1.2. By taking a linear combination of the three types of B-splines shown in figures 1.1 and 1.2 we can generate a linear spline function as shown in figure 1.3.

A quadratic B-spline is a linear combination of two linear B-splines. Shown in figure 1.4 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 1.4. 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,

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

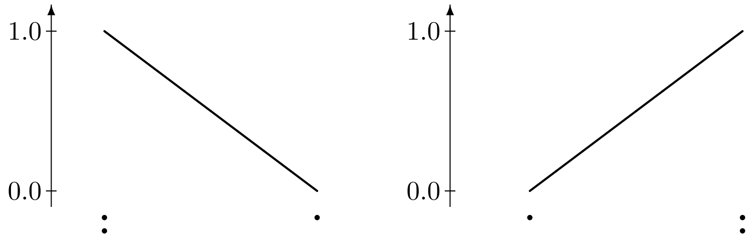


Figure 1.2: Linear B-splines of with multiple knots at one end.

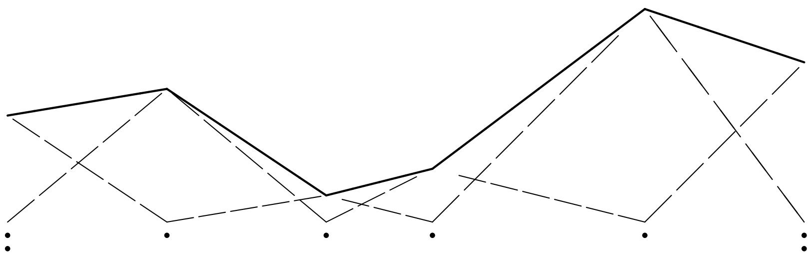


Figure 1.3: A B-spline curve of dimension 1 as a linear combination of a sequence of B-splines. Each B-spline (dashed) is scaled by a coefficient.

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.

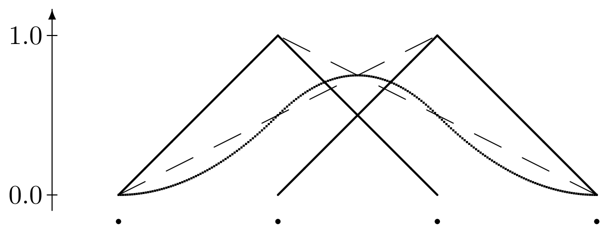


Figure 1.4: A quadratic B-spline, the two linear B-splines and the corresponding lines (dashed) in the quadratic B-spline definition.

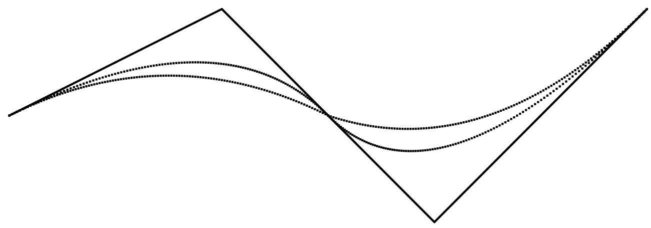


Figure 1.5: Linear, quadratic, and cubic B-spline curves sharing the same control polygon. The control polygon is equal to the linear B-spline curve. The curves are planar, i.e. the space dimension is two.

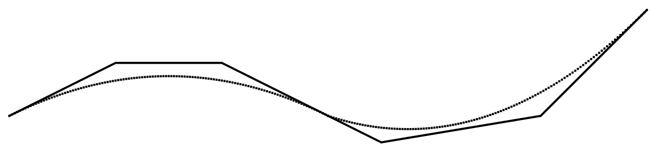


Figure 1.6: The cubic B-spline curve with a redefined knot vector.

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

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

1.5.2 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 1.5). 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 1.6. If the refinement is infinite then the control polygon converges to the curve.

1.5.3 The Knot Vector

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

- The parameterization of the B-spline curve

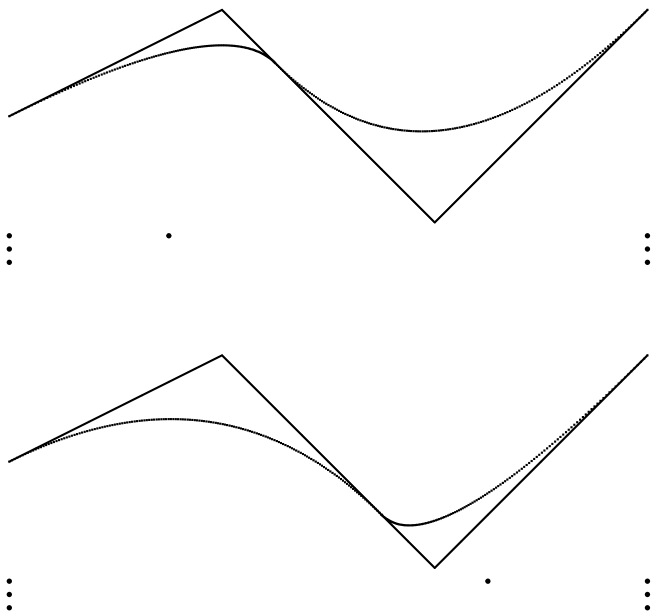


Figure 1.7: Two quadratic B-spline curves with the same control polygon but different knot vectors. The curves and the control polygons are two-dimensional.

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

In figure 1.7 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 1.8). Normally, B-spline curves in SISL are expected to be continuous. For intersection algorithms, curves are usually expected to be continuously differentiable (C^1).

1.5.4 NURBS Curves

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

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

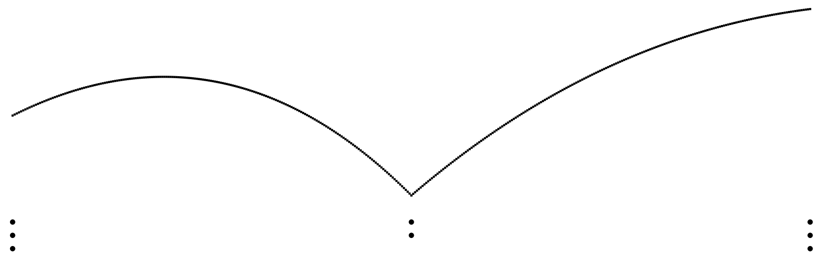


Figure 1.8: A quadratic B-spline curve with two equal internal knots.

In addition to the data of a B-spline curve, the NURBS curve **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

1.6 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 9.1.

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 1.9: 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 1.9. The control net has similar properties to the control polygon of a B-spline curve, described in section 1.5.2. A B-spline surface has two knot vectors, one for each parameter. In figure 1.9 we can see *isocurves*, surface curves defined by fixing the value of one of the parameters.

1.6.1 The Basis Functions

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

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

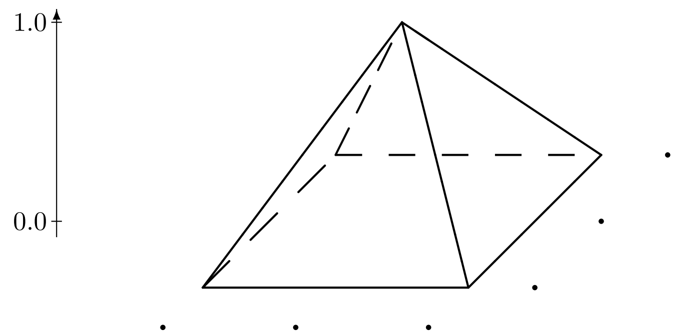


Figure 1.10: A basis function of degree one in both variables.

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 1.10). For higher degrees, the surface basis functions are bell shaped.

1.6.2 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 2

Curve Definition

This chapter describes all functions in the Curve Definition module.

2.1 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).

2.1.1 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;
    int         dim;
    double      startpar;
    double      endpar;
    SISLCurve   *curve;
    int         stat;
    ...
    s1602(startpt, endpt, order, dim, startpar, &endpar, &curve, &stat);
    ...
}
```

2.1.2 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;
    . . .
    s1356(epoint, inbpnt, idim, nptyp, icnsta, icnend, iopen, ik, astpar, &cend-
          par, &rc, &gpar, &jnbpar, &jstat);
    . . .
}
```

2.1.3 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* × *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;
    double      *gpar;
    int         jnbpar;
    int         jstat;
    . . .
    s1357(epoint, inbpnt, idim, ntype, epar, icnsta, icnend, iopen, ik, astpar,
          &cendpar, &rc, &gpar, &jnbpar, &jstat);
    . . .
}
```

2.1.4 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;
    SISLCurve   *curve;
    int         stat;
    . . .
    s1380(point, derivate, numpt, dim, typepar, &curve, &stat);
    . . .
}
```

2.1.5 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;
    int         stat;
    ...
    s1379(point, derivate, par, numpt, dim, &curve, &stat);
    ...
}
```

2.1.6 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;
    SISLCurve   *curve2;
    double      epsge;
    double      end1;
    double      fillpar1;
    double      end2;
    double      fillpar2;
    int         filltype;
    int         dim;
    int         order;
    SISLCurve   *newcurve;
    int         stat;
    ...
    s1607(curve1, curve2, epsge, end1, fillpar1, end2, fillpar2, filltype, dim, order,
          &newcurve, &stat);
    ...
}
```

2.1.7 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;
    SISLCurve   *curve2;
    double      epsge;
    double      point1[3];
    double      startpt1[3];
    double      point2[3];
    double      endpt2[3];
    int         filltype;
    int         dim = 3;
    int         order;
    SISLCurve   *newcurve;
    double      parpt1;
    double      parspt1;
    double      parpt2;
    double      parept2;
    int         stat;
    . . .
    s1608(curve1, curve2, epsge, point1, startpt1, point2, endpt2,
        filltype, dim, order, &newcurve, &parpt1, &parspt1,
        &parpt2, &parept2, &stat);
    . . .
}
```

2.1.8 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;
    SISLCurve   *curve2;
    double      epsge;
    double      point1[3];
    double      pointf[3];
    double      point2[3];
    double      radius;
    double      normal[3];
    int         filltype;
    int         dim = 3;
    int         order;
    SISLCurve   *newcurve;
    double      parend1;
    double      parspt1;
    double      parend2;
    double      parept2;
    int         stat;
    . . .
    s1609(curve1, curve2, epsge, point1, pointf, point2, radius,
          normal, filltype, dim, order, &newcurve, &parend1, &parspt1,
          &parend2, &parept2, &stat);
    . . .
}
```

2.1.9 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;
    double      circ_cen[2];
    double      circ_rad;
    double      aepsge;
    double      eps1[2];
    double      eps2[2];
    double      aradius;
    double      parpt1;
    double      parpt2;
    double      centre[2];
    int         jstat;
    ...
    s1014(pc1, circ_cen, circ_rad, aepsge, eps1, eps2, aradius, &parpt1, &parpt2,
          centre, &jstat);
    ...
}
```

2.1.10 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;
    SISLCurve   *pc2;
    double      aepsge;
    double      eps1[2];
    double      eps2[2];
    double      aradius;
    double      parpt1;
    double      parpt2;
    double      centre[2];
    int         jstat;
    ...
    s1015(pc1, pc2, aepsge, eps1, eps2, aradius, &parpt1, &parpt2, centre, &jstat);
    ...
}
```

2.1.11 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;
    double      point[2];
    double      normal[2];
    double      aepsge;
    double      eps1[2];
    double      eps2[2];
    double      aradius;
    double      parpt1;
    double      parpt2;
    double      centre[2];
    int         jstat;
    ...
    s1016(pc1, point, normal, aepsge, eps1, eps2, aradius, &parpt1, &parpt2,
          centre, &jstat);
    ...
}
```

2.1.12 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;
    SISLCurve   *curve2;
    double      epsge;
    double      point1[3];
    double      point2[3];
    int         blendtype;
    int         dim = 3;
    int         order;
    SISLCurve   *newcurve;
    int         stat;
    ...
    s1606(curve1, curve2, epsge, point1, point2, blendtype, dim, order,
          &newcurve, &stat);
    ...
}
```

2.2 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.

2.2.1 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];
    double      epsge;
    double      angle;
    double      centrept[3];
    double      axis[3];
    int         dim = 3;
    SISLCurve   *curve;
    int         stat;
    ...
    s1303(startpt, epsge, angle, centrept, axis, dim, &curve, &stat);
    ...
}
```

2.2.2 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];
    int         numpt = 10;
    int         dim = 3;
    double      typept[10];
    int         open;
    int         order;
    double      startpar;
    double      epsge;
    double      endpar;
    SISLCurve   *curve;
    int         stat;
    ...
    s1611(point, numpt, dim, typept, open, order, startpar, epsge,
          &endpar, &curve, &stat);
    ...
}
```

2.2.3 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];
    int         inbpnt = 10;
    double      astpar = 0.0;
    int         iopen = 1;
    int         idim = 3;
    int         ik = 4;
    SISLCurve   *rc = NULL;
    int         jstat;
    ...
    s1630(epoint, inbpnt, astpar, iopen, idim, ik, &rc, &jstat);
    ...
}
```

2.2.4 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;
    double      offset;
    double      epsge;
    double      norm[3];
    double      max;
    int         dim = 3;
    SISLCurve   *newcurve;
    int         stat;
    ...
    s1360(oldcurve, offset, epsge, norm, max, dim, &newcurve, &stat);
    ...
}
```

2.2.5 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;
    double      epsge;
    double      *points;
    int         numpoints;
    int         stat;
    ...
    s1613(curve, epsge, &points, &numpoints, &stat);
    ...
}
```

2.3 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;
    double      point[3];
    double      normal[3];
    int         dim = 3;
    SISLCurve   *newcurve;
    int         stat;
    ...
    s1600(oldcurve, point, normal, dim, &newcurve, &stat);
    ...
}
```

2.4 Conversion

2.4.1 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;
    double      *cubic;
    int         numcubic;
    int         dim;
    int         stat;
    ...
    s1389(curve, &cubic, &numcubic, &dim, &stat);
    ...
}
```

2.4.2 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;
    SISLCurve   *newcurve;
    int         stat;
    ...
    s1730(curve, &newcurve, &stat);
    ...
}
```

2.4.3 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).

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;
    int         number;
    double      startpar;
    double      endpar;
    double      coef[12];
    int         stat;
    ...
    s1732(curve, number, &startpar, &endpar, coef, &stat);
    ...
}
```

2.4.4 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;
    double      order;
    SISLCurve   *newcurve;
    int         stat;
    ...
    s1750(curve, order, &newcurve, &stat);
    ...
}
```

2.4.5 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;
    int         derive;
    SISLCurve   *newcurve;
    int         stat;
    ...
    s1720(curve, derive, &newcurve, &stat);
    ...
}
```

2.4.6 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];
    double      centre[3];
    double      ellipaxis[3];
    double      ratio;
    int         dim = 3;
    SISLCurve   *ellipse;
    int         jstat;
    ...
    s1522(normal, centre, ellipaxis, ratio, dim, &ellipse, &jstat);
    ...
}
```

2.4.7 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];
    double      top_pos[3];
    double      end_pos[3];
    double      shape;
    int         dim = 3;
    SISLCurve   *arc_seg;
    int         stat;
    ...
    s1011(start_pos, top_pos, end_pos, shape, dim, &arc_seg, &stat);
    ...
}
```

2.4.8 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];
    double      axis_pos[3];
    double      axis_dir[3];
    double      frequency;
    int         numb_quad;
    int         counter_clock;
    SISLCurve   *helix;
    int         stat;
    ...
    s1012(start_pos, axis_pos, axis_dir, frequency, numb_quad, counter_clock,
          &helix, &stat)
    ...
}
```

Chapter 3

Curve Interrogation

This chapter describes the functions in the Curve Interrogation module.

3.1 Intersections

3.1.1 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;
    double        *pt1;
    int           idim;
    double        aepsge;
    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);
    ...
}
```

3.1.2 Intersection between a curve and a straight line or a plane.

NAME

s1850 - Find all the intersections between a 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;
    double         point[3];
    double         normal[3];
    int            dim = 3;
    double         epsco;
    double         epsge;
    int            numintpt;
    double         *intpar;
    int            numintcu;
    SISLIntcurve   **intcurve;
    int            stat;
    ...
    s1850(curve, point, normal, dim, epsco, epsge, &numintpt, &intpar, &numintcu, &intcurve, &stat);
    ...
}
```

3.1.3 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;
    double      epoint[ ];
    double      enorm1[ ];
    double      enorm2[ ];
    int         idim;
    SISLCurve   **rcnew;
    int         *jstat;
    ...
    s1327(pcold, epoint, enorm1, enorm2, idim, rcnew, jstat);
    ...
}
```

3.1.4 Intersection between a 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;
    double       centre[3];
    double       radius;
    int          dim = 3;
    double       epsco;
    double       epsge;
    int          numintpt;
    double       *intpar;
    int          numintcu;
    SISLIntcurve **intcurve;
    int          stat;
    ...
    s1371(curve, centre, radius, dim, epsco, epsge, &numintpt, &intpar, &numintcu, &intcurve, &stat);
    ...
}
```

3.1.5 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;
    double        conarray[16];
    int           dim = 3;
    double        epsco;
    double        epsge;
    int           numintpt;
    double        *intpar;
    int           numintcu;
    SISLIntcurve  **intcurve;
    int           stat;
    . . .
    s1374(curve, conarray, dim, epsco, epsge, &numintpt, &intpar,
          &numintcu, &intcurve, &stat);
    . . .
}
```

3.1.6 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*;
SISLCurve	**curve2*;

double *epsco*;
double *epsge*;
int *numintpt*;
double **intpar1*;
double **intpar2*;
int *numintcu*;
SISLIntcurve ***intcurve*;
int *stat*;
. . .
s1857(*curve1*, *curve2*, *epsco*, *epsge*, &*numintpt*, &*intpar1*, &*intpar2*, &*numintcu*, &*intcurve*, &*stat*);
. . .
}

3.2 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;
    double      epsge;
    double      length;
    int         stat;
    ...
    s1240(curve, epsge, &length, &stat);
    ...
}
```

3.3 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;
    double      epsge;
    int         stat;
    ...
    s1364(curve, epsge, &stat);
    ...
}
```

3.4 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;
    double      aepsge;
    int         *jdgen;
    int         *jstat;
    ...
    s1451(pc1, aepsge, jdgen, jstat);
    ...
}
```

3.5 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;
    double      startpar;
    double      endpar;
    int         stat;
    ...
    s1363(curve, &startpar, &endpar, &stat);
    ...
}
```

3.6 Closest Points

3.6.1 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;
    double       point[3];
    int          dim = 3;
    double       epsco;
    double       epsge;
    int          numintpt;
    double       *intpar;
    int          numintcu;
    SISLIntcurve **intcurve;
    int          jstat;
    ...
    s1953(curve, point, dim, epsco, epsge, &numintpt, &intpar,
          &numintcu, &intcurve, &jstat);
    ...
}
```

3.6.2 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;
    double      epoint[3];
    int         idim = 3;
    double      aepsco;
    double      aepsge;
    double      gpar = 0;
    double      dist = 0;
    int         jstat = 0;
    ...
    s1957(pcurve, epoint, idim, aepsco, aepsge, &gpar, &dist, &jstat);
    ...
}
```

3.6.3 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*;
double	*point*[ ];
int	*dim*;
double	*epsge*;
double	*start*;
double	*end*;
double	*guess*;
double	**clpar*;
int	**stat*;

. . .

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

3.6.4 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;
    SISLCurve    *curve2;
    double       epsco;
    double       epsge;
    int          numintpt;
    double       *intpar1;
    double       *intpar2;
    int          numintcu;
    SISLIntcurve **intcurve;
    int          stat;
    ...
    s1955(curve1, curve2, epsco, epsge, &numintpt, &intpar1, &intpar2, &numintcu, &intcurve, &stat);
    ...
}
```

3.6.5 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;
    double      ang;
    double      ang_tol;
    double      guess_par;
    double      iter_par;
    int         jstat;
    ...
    s1013(pcurve, ang, ang_tol, guess_par, &iter_par, &jstat);
    ...
}
```

3.7 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;
    double       dir[3];
    int          dim = 3;
    double       epsco;
    double       epsge;
    int          numintpt;
    double       *intpar;
    int          numintcu;
    SISLIntcurve **intcurve;
    int          stat;
    ...
    s1920(curve, dir, dim, epsco, epsge, &numintpt, &intpar, &numintcu,
        &intcurve, &stat);
    ...
}
```

3.8 Area between Curve and Point

3.8.1 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;
    double      point[ ];
    int         dim;
    double      epsge;
    double      *area;
    int         *stat;
    ...
    s1241(pcurve, point, dim, epsge, area, stat);
    ...
}
```

3.8.2 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;
    double      point[ ];
    int         dim;
    double      epsge;
    double      weight[ ];
    double      *area;
    double      *moment;
    int         *stat;
    ...
    s1243(pcurve, point, dim, epsge, weight, area, moment, stat);
```

```
        ...
}
```

3.9 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.

3.9.1 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.

3.9.2 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;
    SISLbox *box;
    ...
    box = newbox(idim);
    ...
}
```

3.9.3 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;
    double      *emax = NULL;
    double      *emin = NULL;
    int         jstat = 0;
    ...
    s1988(pc, &emax, &emin, &jstat);
    ...
}
```

3.10 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.

3.10.1 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.

3.10.2 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;
    SISLdir      *dir;
    ...
    dir = newdir(idim);
    ...
}
```

3.10.3 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;
    double      aepsge;
    int         jgtpi = 0;
    double      *gaxis = NULL;
    double      cang = 0.0;
    int         jstat = 0;
    ...
    s1986(pc, aepsge, &jgtpi, &gaxis, &cang, &jstat);
    ...
}
```

Chapter 4

Curve Analysis

This chapter describes the Curve Analysis part.

4.1 Curvature Evaluation

4.1.1 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;
    double      ax[ ];
    int         num_ax;
    double      curvature[ ];
    int         *jstat;
    . . .
    s2550(curve, ax, num_ ax, curvature, jstat );
    . . .
}
```

4.1.2 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;
    double      ax[ ];
    int         num_ax;
    double      torsion[ ];
    int         *jstat;
    . . .
    s2553(curve, ax, num_ ax, torsion, jstat );
    . . .
}
```

4.1.3 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;
    double      ax[ ];
    int         num_ax;
    double      VoC[ ];
    int         *jstat;
    . . .
    s2556(curve, ax, num_ ax, VoC, jstat );
    . . .
}
```

4.1.4 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;
    double      ax[ ];
    int         num_ax;
    double      p[ ];
    double      t[ ];
    double      n[ ];
    double      b[ ];
    int         *jstat;
    ...
    s2559(curve, ax, num_ ax, p, t, n, b, jstat );
    ...
}
```

4.1.5 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;
    double      ax[ ];
    int         num_ax;
    int         val_flag;
    double      p[ ];
    double      t[ ];
    double      n[ ];
    double      b[ ];
    double      val[ ];
    int         *jstat;
    . . .
    s2562(curve, ax, num_ ax, val_ flag, p, t, n, b, val, jstat );
    . . .
}
```

Chapter 5

Curve Utilities

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

5.1 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).

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.

5.1.1 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 $w_1\mathbf{p}_1, w_1, w_1\mathbf{p}_2, w_2, \ldots, w_n\mathbf{p}_n, w_n$ 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];
    double      coef[30];
    int         kind = 1;
    int         dim = 3;
    int         copy = 1;
    ...
    curve = newCurve(number, order, knots, coef, kind, dim, copy);
    ...
}
```

5.1.2 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];
    double      coef[30];
    int         kind = 1;
    int         dim = 3;
    int         copy = 1;
    . . .
    curve = newCurve(number, order, knots, coef, kind, dim, copy);
    . . .
    curvecopy = copyCurve(curve);
    . . .
}
```

5.1.3 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);
    ...
    freeCurve(curve);
    ...
}
```

5.2 Evaluation

5.2.1 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;
    int        der = 3;
    double     parvalue;
    int        leftknot;
    double     derive[12];
    int        stat;
    ...
    s1227(curve, der, parvalue, &leftknot, derive, &stat);
    ...
}
```

5.2.2 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;
    int         der = 3;
    double      parvalue;
    int         leftknot = 0;
    double      derive[12];
    int         stat;
    ...
    s1221(curve, der, parvalue, &leftknot, derive, &stat);
    ...
}
```

5.2.3 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, first derivative, 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;
    int         der;
    double      parvalue;
    int         *leftknot;
    double      derive[ ];
    double      curvature[ ];
    double      *radius_of_curvature;
    int         *jstat;
    . . .
    s1225(curve, der, parvalue, leftknot, derive, curvature, radius_of_ curvature,
        jstat);
    . . .
}
```

5.2.4 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, first derivative, 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;
    int         der;
    double      parvalue;
    int         *leftknot;
    double      derive[ ];
    double      curvature[ ];
    double      *radius_of_curvature;
    int         *jstat;
    ...
    s1226(curve, der, parvalue, leftknot, derive, curvature, radius_of_ curvature,
        jstat);
    ...
}
```

5.2.5 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. This does not work for 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 derivatives of the curve are placed, dimension idim * (ider+1) * 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;
    int         m;
    double      x[ ];
    double      eder[ ];
    int         *jstat;
    . . .
    s1542(pc1, m, x, eder, jstat);
    . . .
}
```

5.3 Subdivision

5.3.1 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, *rcnew*1=NULL or *rcnew2*=NULL.
$= 2$:	*pc1* periodic, *rcnew2*=NULL.
> 0 :	Warning.
$= 0$:	Ok.
< 0 :	Error.

EXAMPLE OF USE

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

5.3.2 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*;
int	**jstat*;
double	*apar*;
SISLCurve	***rc*;

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

5.3.3 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;
    double      epar[5];
    int         inpar = 5;
    SISLCurve   *rcnew = NULL;
    int         jstat = 0;
    ...
    s1018(pc, epar, inpar, &rcnew, &jstat);
    ...
}
```

5.3.4 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;
    double      parval1;
    double      parval2;
    SISLCurve   *newcurve1;
    SISLCurve   *newcurve2;
    int         stat;
    ...
    s1714(curve, parval1, parval2, &newcurve1, &newcurve2, &stat);
    ...
}
```

5.3.5 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;
    double      begpar;
    double      endpar;
    SISLCurve   *newcurve;
    int         stat;
    ...
    s1712(curve, begpar, endpar, &newcurve, &stat);
    ...
}
```

5.3.6 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;
    double      begpar;
    double      endpar;
    SISLCurve   *newcurve;
    int         stat;
    ...
    s1713(curve, begpar, endpar, &newcurve, &stat);
    ...
}
```

5.4 Joining

5.4.1 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;
    SISLCurve   *curve2;
    int         end1;
    int         end2;
    SISLCurve   *newcurve;
    int         stat;
    ...
    s1715(curve1, curve2, end1, end2, &newcurve, &stat);
    ...
}
```

5.4.2 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;
    SISLCurve   *curve2;
    double      epsge;
    SISLCurve   *newcurve;
    int         stat;
    ...
    s1716(curve1, curve2, epsge, &newcurve, &stat);
    ...
}
```

5.5 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;
    ...
    s1706(curve);
    ...
}
```

5.6 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 – readjusted factor(s) have been used.
> 0 : Warning.

EXAMPLE OF USE

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

5.7 Drawing

5.7.1 Draw a sequence of straight lines.

NAME

s6drawseq - Draw a broken line as a sequence of straight lines described by the array points. For dimension 3.

SYNOPSIS

void s6drawseq(*points*, *numpoints*)

double	*points*[ ];
int	*numpoints*;

ARGUMENTS

Input Arguments:

points	-	Points stored in sequence. i.e. $(x_0, y_0, z_0, x_1, y_1, z_1, \ldots)$.
numpoints	-	Number of points in the sequence.

NOTE

s6drawseq() is device dependent, it calls the empty dummy functions s6move() and s6line(). Before using it, make sure you have a version of these two functions interfaced to your graphic package.
More about s6move() and s6line() on pages 330 and 331.

EXAMPLE OF USE

```
{
    double      points[30];
    int         numpoints = 10;
    ...
    s6drawseq(points, numpoints)
    ...
}
```

5.7.2 Basic graphics routine template - move plotting position.

NAME

s6move - Move the graphics plotting position to a 3D point.

SYNOPSIS

void s6move(*point*)

double *point*[ ];

ARGUMENTS

Input Arguments:

point	-	A 3D point, i.e. (x, y, z), to move the graphics plotting position to.

NOTE

The functionality of s6move() is device dependent, so it is only an empty (`printf()` call) dummy routine. Before using it, make sure you have a version of s6move() interfaced to your graphic package.

EXAMPLE OF USE

```
{
    double      point[3];
    ...
    s6move(point)
    ...
}
```

5.7.3 Basic graphics routine template - plot line.

NAME

s6line - Plot a line between the current 3D graphics plotting position and a given 3D point.

SYNOPSIS

void s6line(*point*)

double *point*[ ];

ARGUMENTS

Input Arguments:

point - A 3D point, i.e. (x, y, z), to draw a line to, from the current graphics plotting position.

NOTE

The functionality of s6line() is device dependent, so it is only an empty (`printf()` call) dummy routine. Before using it, make sure you have a version of s6line() interfaced to your graphic package.

EXAMPLE OF USE

```
{
    double      point[3];
    . . .
    s6line(point)
    . . .
}
```

Chapter 6

Surface Definition

6.1 Interpolation

6.1.1 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).

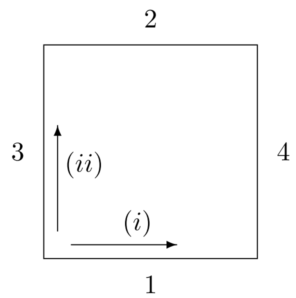
im1 - The number of interpolation points in the first parameter direction.

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];
    int         im1 = 10;
    int         im2 = 10;
    int         idim = 3;
    int         ipar;
    int         con1;
    int         con2;
    int         con3;
    int         con4;
    int         order1;
    int         order2;
    int         iopen1;
    int         iopen2;
    SISLSurf    *rsurf;
    int         jstat;
    ...
    s1536(points, im1, im2, idim, ipar, con1, con2, con3, con4, order1, order2,
          iopen1, iopen2, &rsurf, &jstat);
    ...
}
```

6.1.2 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*)

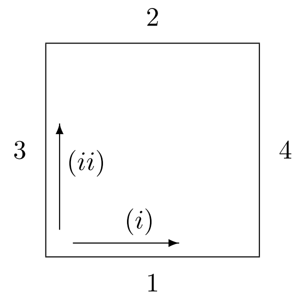
double	*points*[ ];
int	*im1*;
int	*im2*;
int	*idim*;
double	*par1*[ ];
double	*par2*[ ];
int	*con1*;
int	*con2*;
int	*con3*;
int	*con4*;
int	*order1*;
int	*order2*;
int	*iopen1*;
int	*iopen2*;
SISLSurf	***rsurf*;
int	**jstat*;

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];
    int         im1 = 10;
    int         im2 = 10;
    int         idim = 3;
    double      par1[10];
    double      par2[10];
    int         con1;
    int         con2;
    int         con3;
    int         con4;
    int         order1;
    int         order2;
    int         iopen1;
    int         iopen2;
    SISLSurf    *rsurf;
    int         jstat;
    ...
    s1537(points, im1, im2, idim, par1, par2, con1, con2, con3, con4, order1,
        order2, iopen1, iopen2, &rsurf, &jstat);
    ...
}
```

6.1.3 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*)

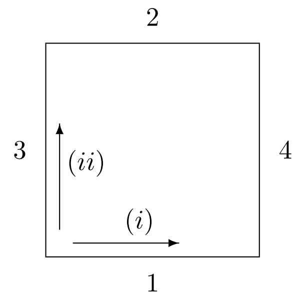
double	*points*[ ];
double	*der10*[ ];
double	*der01*[ ];
double	*der11*[ ];
int	*im1*;
int	*im2*;
int	*idim*;
int	*ipar*;
int	*con1*;
int	*con2*;
int	*con3*;
int	*con4*;
int	*order1*;
int	*order2*;
SISLSurf	***rsurf*;
int	**jstat*;

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];
    double      der10[300];
    double      der01[300];
    double      der11[300];
    int         im1 = 10;
    int         im2 = 10;
    int         idim = 3;
    int         ipar;
    int         con1;
    int         con2;
    int         con3;
    int         con4;
    int         order1;
    int         order2;
    SISLSurf    *rsurf;
    int         jstat;
    ...
    s1534(points, der10, der01, der11, im1, im2, idim, ipar, con1, con2, con3,
          con4, order1, order2, &rsurf, &jstat);
    ...
}
```

6.1.4 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*)

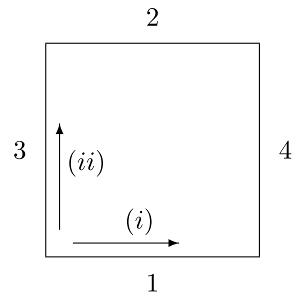
double	*points*[ ];
double	*der10*[ ];
double	*der01*[ ];
double	*der11*[ ];
int	*im1*;
int	*m2*;
int	*idim*;
double	*par1*[ ];
double	*par2*[ ];
int	*con1*;
int	*con2*;
int	*con3*;
int	*con4*;
int	*order1*;
int	*order2*;
SISLSurf	***rsurf*;
int	**jstat*;

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

6.1.5 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];
    double      eder10[300];
    double      eder01[300];
    double      eder11[300];
    int         im1 = 10;
    int         im2 = 10;
    int         idim = 3;
    int         ipar;
    SISLSurf    *rsurf = NULL;
    int         jstat = 0;
    ...
    s1529( ep, eder10, eder01, eder11, im1, im2, idim, ipar, &rsurf, &jstat);
    ...
}
```

6.1.6 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];
    double      eder10[30];
    double      eder01[30];
    double      eder11[30];
    double      epar1[2];
    double      epar2[5];
    int         im1 = 2;
    int         im2 = 5;
    int         idim = 3;
    SISLSurf    *rsurf;
    int         jstat;
    ...
    s1530(ep, eder10, eder01, eder11, epar1, epar2, im1, im2, idim, &rsurf, &js-
        tat);
    ...
}
```

6.1.7 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;
    SISLCurve   *vpcurv[3];
    int         nctyp[3];
    double      astpar;
    int         iopen;
    int         iord2;
    int         iflag;
    SISLSurf    *rsurf = NULL;
    double      *gpar = NULL;
    int         jstat = 0;
    ...
    s1538(inbcrv, vpcurv, nctyp, astpar, iopen, iord2, iflag, &rsurf, &gpar, &jstat);
    ...
}
```

6.1.8 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 repeted 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;
    SISLCurve    *vpcurv[ ];
    int          nctyp[ ];
    double       epar[ ];
    double       astpar;
    int          iopen;
    int          iord2;
    int          iflag;
    SISLSurf     **rsurf;
    double       **gpar;
    int          *jstat;
    . . .
    s1539(inbcrv, vpcurv, nctyp, epar, astpar, iopen, iord2, iflag, rsurf, gpar,
          jstat);
    . . .
}
```

6.1.9 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.

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.

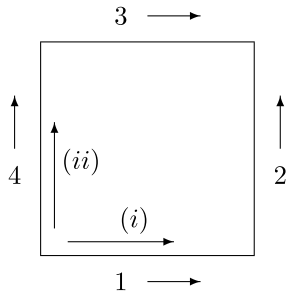
EXAMPLE OF USE

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

6.1.10 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];
    SISLSurf    *surf;
    int         numder[4];
    int         stat;
    ...
    s1390(curves, &surf, numder, &stat)
    ...
}
```

6.1.11 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;
    SISLSurf    **ws = NULL;
    int         icurv = 5;
    int         nder[5];
    int         jstat = 0;
    ...
    s1391(pc, &ws, icurv, nder, &jstat);
    ...
}
```

6.1.12 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*)

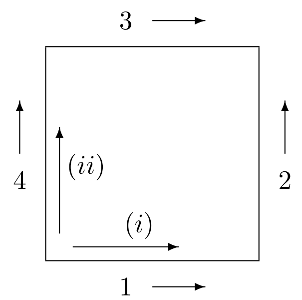
double	*etwist*[ ];
SISLCurve	**vcurve*[ ];
int	**jstat*;
SISLSurf	***rsurf*;

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[4*idim];
    SISLCurve    *vcurve[8];
    int          jstat = 0;
    SISLSurf     *rsurf = NULL;
    ...
    s1401(vcurve, etwist, &rsurf, &jstat);
    ...
}
```

6.2 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.

6.2.1 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];
    int         inbpnt1 = 10;
    int         inbpnt2 = 10;
    int         ipar;
    int         iopen1;
    int         iopen2;
    int         ik1;
    int         ik2;
    int         idim = 3;
    SISLSurf    *rs = NULL;
    int         jstat = 0;
    . . .
    s1620(epoint, inbpnt1, inbpnt2, ipar, iopen1, iopen2, ik1, ik2, idim, &rs,
          &jstat);
    . . .
}
```

6.2.2 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;
    curve       *curve2;
    double      epsge;
    double      point[3];
    SISLSurf    *surf;
    int         stat;
    ...
    s1332(curve1, curve2, epsge, point, &surf, &stat);
    ...
}
```

6.2.3 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;
    double      epsge;
    double      angle;
    double      point[3];
    double      axis[3];
    SISLSurf    *surf;
    int         stat;
    ...
    s1302(curve, epsge, angle, point, axis, &surf, &stat);
    ...
}
```

6.2.4 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. If $idim = 2$ a positive signe on this value put the offset on the side of the positive normal vector, and a negative sign puts the offset on the sign of the negative normal vector. If $idim = 3$ the offset is determined by the 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 neglected 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 (2 or 3).

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;
    double      aoffset;
    double      aepsge;
    double      amax;
    int         idim;
    SISLSurf    *rs;
    int         jstat;
    ...
    s1365(ps, aoffset, aepsge, amax, idim, &rs, &jstat);
    ...
}
```

6.3 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;
    double      epoint[3];
    double      enorm[3];
    int         idim = 3;
    SISLSurf    *rsurf = NULL;
    int         jstat = 0;
    ...
    s1601(psurf, epoint, enorm, idim, &rsurf, &jstat);
    ...
}
```

6.4 Conversion

6.4.1 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;
    double      *coons;
    int         numcoons1;
    int         numcoons2;
    int         dim
    int         stat;
    ...
    s1388(surf, &coons, &numcoons1, &numcoons2, &dim, &stat);
    ...
}
```

6.4.2 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;
    SISLSurf    *newsurf;
    int         stat;
    . . .
    s1731(surf, &newsurf, &stat);
    . . .
}
```

6.4.3 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 175. 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 $(idim+1) \times ik1 \times ik2$ as 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;
    int         number1;
    int         number2;
    double      startpar1;
    double      endpar1;
    double      startpar2;
    double      endpar2;
    double      coef[48];
    int         stat;
    ...
    s1733(surf, number1, number2, &startpar1, &endpar1, &startpar2, &endpar2, coef, &stat);
    ...
}
```

6.4.4 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;
    int         order1;
    int         order2;
    SISLSurf    *newsurf;
    int         stat;
    . . .
    s1387(surf, order1, order2, &newsurf, &stat);
    . . .
}
```

6.4.5 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;
    int         der1;
    int         der2;
    SISLSurf    *newsurf;
    int         stat;
    ...
    s1386(surf, der1, der2, &newsurf, &stat);
    ...
}
```

6.4.6 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];
    double      axis[3];
    double      equator[3];
    int         latitude;
    int         longitude;
    SISLSurf    *sphere = NULL;
    int         stat = 0;
    ...
    s1023(centre, axis, equator, latitude, longitude, &sphere, &stat);
    ...
}
```

6.4.7 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];
    double      bottom_axis[3];
    double      ellipse_ratio;
    double      axis_dir[3];
    double      height;
    SISLSurf    *cyl = NULL;
    int         stat = 0;
    ...
    s1021(bottom_pos, bottom_axis, ellipse_ratio, axis_dir, height, &cyl, &stat)
    ...
}
```

6.4.8 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];
    double      axis[3];
    double      equator[3];
    double      minor_radius;
    int         start_minor;
    int         end_minor;
    int         numb_major;
    SISLSurf    *torus = NULL;
    int         stat = 0;
    ...
    s1024(centre, axis, equator, minor_radius, start_minor, end_minor,
          numb_major, &torus, &stat)
    ...
}
```

6.4.9 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];
    double      bottom_axis[3];
    double      ellipse_ratio;
    double      axis_dir[3];
    double      cone_angle;
    double      height;
    SISLSurf    *cone = NULL;
    int         stat = 0;
    ...
    s1022(bottom_pos, bottom_axis, ellipse_ratio, axis_dir, cone_angle, height,
          &cone, &stat)
    ...
}
```

Chapter 7

Surface Interrogation

This chapter describes the functions in the Surface Interrogation module.

7.1 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. This will be the result from the topology routines. The second level is curves, one curve in the geometric space and one curve in each parameter plane if each surface is parametric. This will be the result from the marching routines.

7.1.1 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.

7.1.2 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 187

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];
    double       guidepar2[4];
    int          type = 4;
    ...
    intcurve = newIntcurve(numgdpt, numpar1, numpar2, guidepar1,
                           guidepar2, type);
    ...
}
```

7.1.3 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);
    ...
    freeIntcurve(intcurve);
    ...
}
```

7.1.4 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;
    int          icrv;
    . . .
    freeIntcrvlist(vilist, icrv);
    . . .
}
```

7.2 Find the Intersections

7.2.1 Intersection between a curve and a straight line or a plane.

NAME

s1850 - Find all the intersections between a 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;
    double         point[3];
    double         normal[3];
    int            dim = 3;
    double         epsco;
    double         epsge;
    int            numintpt;
    double         *intpar;
    int            numintcu;
    SISLIntcurve   **intcurve;
    int            stat;
    ...
    s1850(curve, point, normal, dim, epsco, epsge, &numintpt, &intpar, &numintcu, &intcurve, &stat);
    ...
}
```

7.2.2 Intersection between a 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;
    double        centre[3];
    double        radius;
    int           dim = 3;
    double        epsco;
    double        epsge;
    int           numintpt;
    double        *intpar;
    int           numintcu;
    SISLIntcurve  **intcurve;
    int           stat;
    ...
    s1371(curve, centre, radius, dim, epsco, epsge, &numintpt, &intpar, &numintcu, &intcurve, &stat);
    ...
}
```

7.2.3 Intersection between a curve and a cylinder.

NAME

s1372 - Find all the intersections between a 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;
    double        point[3];
    double        dir[3];
    double        radius;
    int           dim = 3;
    double        epsco;
    double        epsge;
    int           numintpt;
    double        *intpar;
    int           numintcu;
    SISLIntcurve  **intcurve;
    int           stat;
    ...
    s1372(curve, point, dir, radius, dim, epsco, epsge, &numintpt,
          &intpar, &numintcu, &intcurve, &stat);
    ...
}
```

7.2.4 Intersection between a 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;
  double top[3];
  double dir[3];
  double conept[3];
  int dim = 3;
  double epsco;
  double epsge;
  int numintpt;
  double *intpar;
  int numintcu;
  SISLIntcurve **intcurve;
  int stat;
  ...
  s1373(curve, top, dir, conept, dim, epsco, epsge, &numintpt, &intpar, &numintcu, &intcurve, &stat);
  ...
}

7.2.5 Intersection between a 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;
    double       basept[3];
    double       normdir[3];
    double       ellipaxis[3];
    double       alpha;
    double       ratio;
    int          dim = 3;
    double       epsco;
    double       epsge;
    int          numintpt;
    double       *intpar;
    int          numintcu;
    SISLIntcurve **intcurve;
    int          stat;
    ...
    s1502(curve, basept, normdir, ellipaxis, alpha, ratio, dim, epsco, epsge, &numintpt, &intpar, &numintcu, &intcurve, &stat);
    ...
}
```

7.2.6 Intersection between a curve and a torus.

NAME

s1375 - Find all the intersections between a 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;
    double       centre[3];
    double       normal[3];
    double       centdist;
    double       rad;
    int          dim = 3;
    double       epsco;
    double       epsge;
    int          numintpt;
    double       *intpar;
    int          numintcu;
    SISLIntcurve **intcurve;
    int          stat;
    ...
    s1375(curve, centre, normal, centdist, rad, dim, epsco, epsge,
          &numintpt, &intpar, &numintcu, &intcurve, &stat);
    ...
}
```

7.2.7 Intersection between a surface and a point.

NAME

s1870 - Find all intersections between a 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;
    double         *pt1;
    int            idim;
    double         aepsge;
    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);
    ...
}
```

7.2.8 Intersection between a 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;
    double        point[3];
    double        linedir[3];
    int           dim = 3;
    double        epsco;
    double        epsge;
    int           numintpt;
    double        *pointpar;
    int           numintcr;
    SISLIntcurve  **intcurves;
    int           stat;
    ...
    s1856(surf, point, linedir, dim, epsco, epsge, &numintpt, &pointpar, &numintcr, &intcurves, &stat);
    ...
}
```

7.2.9 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;
    double      point[];
    double      dir[];
    double      epsge;
    double      start[];
    double      end[];
    double      parin[];
    double      parout[];
    int         *stat;
    ...
```

s1518(*surf*, *point*, *dir*, *epsge*, *start*, *end*, *parin*, *parout*, *stat*);
. . .
}

7.2.10 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;
    double      epoint[];
    double      enorm1[];
    double      enorm2[];
    int         idim;
    SISLSurf    **rsnew;
    int         *jstat;
    . . .
    s1328(psold, epoint, enorm1, enorm2, idim, rsnew, jstat);
    . . .
}
```

7.2.11 Intersection between a 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;
    double        centre[3];
    double        radius;
    double        normal[3];
    int           dim = 3;
    double        epsco;
    double        epsge;
    int           numintpt;
    double        *pointpar;
    int           numintcr;
    SISLIntcurve  **intcurves;
    int           stat;
    . . .
    s1855(surf, centre, radius, normal, dim, epsco, epsge, &numintpt, &pointpar,
        &numintcr, &intcurves, &stat);
    . . .
}
```

7.2.12 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;
    SISLCurve     *curve;
    double        epsco;
    double        epsge;
    int           numintpt;
    double        *pointpar1;
    double        *pointpar2;
    int           numintcr;
    SISLIntcurve  **intcurves;
    int           stat;
    ...
    s1858(surf, curve, epsco, epsge, &numintpt, &pointpar1, &pointpar2, &nu-
        mintcr, &intcurves, &stat);
    ...
}
```

7.3 Find the Topology of the Intersection

7.3.1 Find the topology for the intersection of a 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 236.

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;
    double        point[3];
    double        normal[3];
    int           dim = 3;
    double        epsco;
    double        epsge;
    int           numintpt;
    double        *pointpar;
    int           numintcr;
    SISLIntcurve  **intcurves;
    int           stat;
    ...
    s1851(surf, point, normal, dim, epsco, epsge, &numintpt, &pointpar, &numintcr, &intcurves, &stat);
    ...
}
```

7.3.2 Find the topology for the intersection of a 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 238.

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*;

```
    double        centre[3];
    double        radius;
    int           dim = 3;
    double        epsco;
    double        epsge;
    int           numintpt;
    double        *pointpar;
    int           numintcr;
    SISLIntcurve  **intcurves;
    int           stat;
    ...
    s1852(surf, centre, radius, dim, epsco, epsge, &numintpt, &pointpar, &numintcr, &intcurves, &stat);
    ...
}
```

7.3.3 Find the topology for the intersection of a 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 240.

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;
    double      point[3];
    double      cyldir[3];
    double      radius;
    int         dim = 3;
    double      epsco;
    double      epsge;
    int         numintpt;
    double      *pointpar;
    int         numintcr;
    intcurve    **intcurves;
    int         stat;
    ...
    s1853(surf, point, cyldir, radius, dim, epsco, epsge, &numintpt, &pointpar,
        &numintcr, &intcurves, &stat);
    ...
}
```

7.3.4 Find the topology for the intersection of a 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 242.

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;
    double        toppt[3];
    double        axispt[3];
    double        conept[3];
    int           dim = 3;
    double        epsco;
    double        epsge;
    int           numintpt;
    double        *pointpar;
    int           numintcr;
    SISLIntcurve  **intcurves;
    int           stat;
    ...
    s1854(surf, toppt, axispt, conept, dim, epsco, epsge, &numintpt, &pointpar,
          &numintcr, &intcurves, &stat);
    ...
}
```

7.3.5 Find the topology for the intersection of a 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 244.

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;
    double          basept[3];
    double          normdir[3];
    double          ellipaxis[3];
    double          alpha;
    double          ratio;
    double          alpha;
    int             dim = 3;
    double          epsco;
    double          epsge;
    int             numintpt;
    double          *pointpar;
    int             numintcr;
    SISLIntcurve    **intcurves;
    int             stat;
    . . .
    s1503(surf, basept, normdir, ellipaxis, alpha, ratio, dim, epsco, epsge, &numintpt, &pointpar, &numintcr, &intcurves, &stat);
    . . .
}
```

7.3.6 Find the topology for the intersection of a 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 247.

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;
    double         centre[3];
    double         normal[3];
    double         cendist;
    double         radius;
    int            dim = 3;
    double         epsco;
    double         epsge;
    int            numintpt;
    double         *pointpar;
    int            numintcr;
    SISLIntcurve   **intcurves;
    int            stat;
    ...
    s1369(surf, centre, normal, cendist, radius, dim, epsco, epsge,
          &numintpt, &pointpar, &numintcr, &intcurves, &stat);
    ...
}
```

7.3.7 Find the topology for the intersection between two 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 250.

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;
    SISLSurf      *surf2;
    double        epsco;
    double        epsge;
    int           numintpt;
    double        *pointpar1;
    double        *pointpar2;
    int           numintcr;
    SISLIntcurve  **intcurves;
    int           stat;
    ...
    s1859(surfl, surf2, epsco, epsge, &numintpt, &pointpar1, &pointpar2, &nu-
          mintcr, &intcurves, &stat);
    ...
}
```

7.4 Find the Topology of a Silhouette

7.4.1 Find the topology of the silhouette curves of a 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 252.

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;
    double        viewdir[3];
    int           dim;
    double        epsco;
    double        epsge;
    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);
    ...
}
```

7.4.2 Find the topology of the silhouette curves of a 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 255.

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;
    double        eyepoint[3];
    int           idim = 3;
    double        aepsco;
    double        aepsge;
    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, &jstat);
    ...
}
```

7.4.3 Find the topology of the circular silhouette curves of a 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 257.

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*;

```
    double        qpoint[3];
    double        bvec[3];
    int           idim = 3;
    double        aepsco;
    double        aepsge;
    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);
    ...
}
```

7.5 Marching

7.5.1 March an intersection curve between a 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 216. 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. If this option is 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 s6move() and s6line() interfaced to your graphic package. More about s6move() and s6line() on pages 330 and 331.

EXAMPLE OF USE

```
{
    SISLSurf     *surf;
    double       point[3];
    double       normal[3];
    int          dim = 3;
    double       epsco;
    double       epsge;
    double       maxstep = 0.0;
    SISLIntcurve *intcurve;
    int          makecurv;
    int          graphic;
    int          stat;
    ...
    s1314(surf, point, normal, dim, epsco, epsge, maxstep, intcurve,
          makecurv, graphic, &stat);
    ...
}
```

7.5.2 March an intersection curve between a 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 218. 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. If this option is 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 s6move() and s6line() interfaced to your graphic package. More about s6move() and s6line() on pages 330 and 331.

EXAMPLE OF USE

```
{
    SISLSurf      *surf;
    double        centre[3];
    double        radius;
    int           dim = 3;
    double        epsco;
    double        epsge;
    double        maxstep = 0;
    SISLIntcurve  *intcurve;
    int           makecurv;
    int           graphic;
    int           stat;
    ...
    s1315(surf, centre, radius, dim, epsco, epsge, maxstep, intcurve, makecurv,
          graphic, &stat);
    ...
}
```

7.5.3 March an intersection curve between a 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 220. 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. If this option is 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 s6move() and s6line() interfaced to your graphic package. More about s6move() and s6line() on pages 330 and 331.

EXAMPLE OF USE

```
{
    SISLSurf      *surf;
    double        point[3];
    double        cyldir[3];
    double        radius;
    int           dim = 3;
    double        epsco;
    double        epsge;
    double        maxstep = 0.0;
    SISLIntcurve  *intcurve;
    int           makecurv;
    int           graphic;
    int           stat = 0;
    ...
    s1316(surf, point, cyldir, radius, dim, epsco, epsge, maxstep, intcurve, make-
          curv, graphic, &stat);
    ...
}
```

7.5.4 March an intersection curve between a 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 222. 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. If this option is 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 s6move() and s6line() interfaced to your graphic package. More about s6move() and s6line() on pages 330 and 331.

EXAMPLE OF USE

```
{
    SISLSurf      *surf;
    double        toppt[3];
    double        axispt[3];
    double        conept[3];
    int           dim = 3;
    double        epsco;
    double        epsge;
    double        maxstep = 0.0;
    SISLIntcurve  *intcurve;
    int           makecurv;
    int           graphic;
    int           stat = 0;
    ...
    s1317(surf, toppt, axispt, conept, dim, epsco, epsge, maxstep, intcurve,
          makecurv, graphic, &stat);
    ...
}
```

7.5.5 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 224. 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. If this option is 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 s6move() and s6line() interfaced to your graphic package. More about s6move() and s6line() on pages 330 and 331.

EXAMPLE OF USE

```
{
    SISLSurf      *surf;
    double        basept[3];
    double        normdir[3];
    double        ellipaxis[3];
    double        alpha;
    double        ratio;
    int           dim = 3;
    double        epsco;
    double        epsge;
    double        maxstep = 0.0;
    SISLIntcurve  *intcurve;
    int           makecurv;
    int           graphic;
    int           stat = 0;
    . . .
    s1501(surf, basept, normdir, ellipaxis, alpha, ratio, dim, epsco, epsge,
          maxstep, intcurve, makecurv, graphic, &stat);
    . . .
}
```

7.5.6 March an intersection curve between a 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 226. 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. If this option is 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 s6move() and s6line() interfaced to your graphic package. More about s6move() and s6line() on pages 330 and 331.

EXAMPLE OF USE

```
{
    SISLSurf      *surf;
    double        centre[3];
    double        normal[3];
    double        cendist;
    double        radius;
    int           dim = 3;
    double        epsco;
    double        epsge;
    double        maxstep = 0.0;
    SISLIntcurve  *intcurve;
    int           makecurv;
    int           graphic;
    int           stat = 0;
    ...
    s1318(surf, centre, normal, cendist, radius, dim, epsco, epsge, maxstep,
          intcurve, makecurv, graphic, &stat);
    ...
}
```

7.5.7 March an intersection curve between two 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 228. 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. If this option is 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 s6move() and s6line() interfaced to your graphic package. More about s6move() and s6line() on pages 330 and 331.

EXAMPLE OF USE

```
{
    SISLSurf      *surf1;
    SISLSurf      *surf2;
    SISLIntcurve  *intcurve;
    double        epsge;
    double        maxstep;
    int           makecurv;
    int           graphic;
    int           stat = 0;
    . . .
    s1310(surf1, surf2, intcurve, epsge, maxstep, makecurv, graphic, &stat);
    . . .
}
```

7.6 Marching of Silhouettes

7.6.1 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 230. 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. If this option is 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 s6move() and s6line() interfaced to your graphic package. More about s6move() and s6line() on pages 330 and 331.

EXAMPLE OF USE

```
{
    SISLSurf     *surf;
    double       viewdir[3];
    int          dim = 3;
    double       epsco;
    double       epsge;
    double       maxstep = 0.0;
    SISLIntcurve *intcurve;
    int          makecurv;
    int          graphic;
    int          stat = 0;
    ...
    s1319(surf, viewdir, dim, epsco, epsge, maxstep, intcurve, makecurv,
          graphic, &stat);
    ...
}
```

7.6.2 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 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 through s6move() and s6line().

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 s6move() and s6line() interfaced to your graphic package. More about s6move() and s6line() on pages 330 and 331.

EXAMPLE OF USE

```
{
    SISLSurf      *ps1;
    double        eyepoint[3];
    int           idim = 3;
    double        aepsco;
    double        aepsge;
    double        amax;
    SISLIntcurve  *pintcr;
    int           icur;
    int           igraph;
    int           jstat = 0;
    ...
    s1514(ps1, eyepoint, idim, aepsco, aepsge, amax, pintcr, icur, igraph, &jstat);
    ...
}
```

7.6.3 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 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 through s6move() and s6line().

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 s6move() and s6line() interfaced to your graphic package. More about s6move() and s6line() on pages 330 and 331.

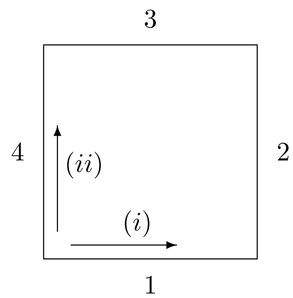
EXAMPLE OF USE

```
{
    SISLSurf      *ps1;
    double        qpoint[3];
    double        bvec[3];
    int           idim;
    double        aepsco;
    double        aepsge;
    double        amax;
    SISLIntcurve  *pintcr;
    int           icur;
    int           igraph;
    int           jstat = 0;
    ...
s1515(ps1, qpoint, bvec, idim, aepsco, aepsge, amax, pintcr, icur, igraph,
    &jstat);
    ...
}
```

7.7 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;
    double      epsge;
    int         close1;
    int         close2;
    int         degen1;
    int         degen2;
    int         degen3;
    int         degen4;
    int         stat;
    ...
    s1450(surf, epsge, &close1, &close2, &degen1, &degen2, &degen3, &degen4,
          &stat);
    ...
}
```

7.8 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;
    double     min1;
    double     min2;
    double     max1;
    double     max2;
    int        stat;
    ...
    s1603(surf, &min1, &min2, &max1, &max2, &stat);
    ...
}
```

7.9 Closest Points

7.9.1 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;
    double         point[3];
    int            dim = 3;
    double         epsco;
    double         epsge;
    int            numclopt;
    double         *pointpar;
    int            numclocr;
    SISLIntcurve   **clocurves;
    int            stat;
    ...
    s1954(surf, point, dim, epsco, epsge, &numclopt, &pointpar, &numclocr,
          &clocurves, &stat);
    ...
}
```

7.9.2 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 262.

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;
    double      epoint[3];
    int         idim = 3;
    double      aepsco;
    double      aepsge;
    double      gpar[2];
    double      dist = 0;
    int         jstat = 0;
    . . .
    s1958(psurf, epoint, idim, aepsco, aepsge, gpar, &dist, &jstat);
    . . .
}
```

7.9.3 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*;
double	*point*[ ];
int	*dim*;
double	*epsge*;
double	*start*[ ];
double	*end*[ ];
double	*guess*[ ];
double	*clpar*[ ];

```
        int         *stat;
        ...
        s1775(surf, point, dim, epsge, start, end, guess, clpar, stat);
        ...
}
```

7.10 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;
    double          edir[3];
    int             idim = 3;
    double          aepsco;
    double          aepsge;
    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);
    ...
}
```

7.11 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.

7.11.1 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.

7.11.2 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;
    SISLbox *box;
    ...
    box = newbox(idim);
    ...
}
```

7.11.3 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;
    double      *emax = NULL;
    double      *emin = NULL;
    int         jstat = 0;
    ...
    s1989(ps, &emax, &emin, &jstat);
    ...
}
```

7.12 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.

7.12.1 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.

7.12.2 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;
    SISLdir     *dir;
    ...
    dir = newdir(idim);
    ...
}
```

7.12.3 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;
    double      aepsge;
    int         jgtpi = 0;
    double      *gaxis = NULL;
    double      cang = 0.0;
    int         jstat = 0;
    ...
    s1987(ps, aepsge, &jgtpi, &gaxis, &cang, &jstat);
    ...
}
```

Chapter 8

Surface Analysis

This chapter describes the Surface Analysis part.

8.1 Curvature Evaluation

8.1.1 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;
    int         ider;
    int         iside1;
    int         iside2;
    double      parvalue[ ];
    int         *leftknot1;
    int         *leftknot2;
    double      *gaussian;
    int         *jstat;
    ...
    s2500(surf, ider, iside1, iside2, parvalue, leftknot1, leftknot2, gaussian, jstat);
    ...
}
```

8.1.2 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;
    int        ider;
    int        iside1;
    int        iside2;
    double     parvalue[ ];
    int        *leftknot1;
    int        *leftknot2;
    double     *meancurvature;
    int        *jstat;
    ...
    s2502(surf, ider, iside1, iside2, parvalue, leftknot1, leftknot2, meancurvature,
          jstat);
    ...
}
```

8.1.3 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;
    int         ider;
    int         iside1;
    int         iside2;
    double      parvalue[ ];
    int         *leftknot1;
    int         *leftknot2;
    double      *absCurvature;
    int         *jstat;
    ...
    s2504(surf, ider, iside1, iside2, parvalue, leftknot1, leftknot2, absCurvature,
        jstat);
    ...
}
```

8.1.4 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;
    int         ider;
    int         iside1;
    int         iside2;
    double      parvalue[ ];
    int         *leftknot1;
    int         *leftknot2;
    double      *totalCurvature;
    int         *jstat;
    ...
    s2506(surf, ider, iside1, iside2, parvalue, leftknot1, leftknot2, totalCurvature,
        jstat);
    ...
}
```

8.1.5 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;
    int        ider;
    int        iside1;
    int        iside2;
    double     parvalue[ ];
    int        *leftknot1;
    int        *leftknot2;
    double     *mehlum;
    int        *jstat;
    ...
    s2508(surf, ider, iside1, iside2, parvalue, leftknot1, leftknot2, mehlum, jstat);
    ...
}
```

8.1.6 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;
    int         ider;
    int         iside1;
    int         iside2;
    double      parvalue[];
    int         *leftknot1;
    int         *leftknot2;
    double      *mehlum;
    int         *jstat;
    ...
    s2510(surf, ider, iside1, iside2, parvalue, leftknot1, leftknot2, mehlum, jstat);
    ...
}
```

8.1.7 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*;
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.
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.

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;
    int         u_continuity;
    int         v_continuity;
    int         *u_surfnumb;
    int         *v_surfnumb;
    SISLSurf    ***gauss_surf;
    int         *stat;
    ...
    s2532(surf, u_continuity, v_continuity, u_surfnumb, v_surfnumb, gauss_surf,
        stat);
    ...
}
```

8.1.8 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.
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.

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;
    int         u_continuity;
    int         v_continuity;
    int         *u_surfnumb;
    int         *v_surfnumb;
    SISLSurf    ***mehlum_surf;
    int         *stat;
    ...
    s2536(surf, u_continuity, v_continuity, u_surfnumb, v_surfnumb,
          mehlum_surf, stat);
    ...
}
```

8.1.9 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 : 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.

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;
    int         curvature_type;
    int         export_par_val;
    int         pick_subpart;
    double      boundary[ ];
    int         n_u;
    int         n_v;
    double      **garr;
    int         *stat;
    ...
    s2540(surf, curvature_type, export_par_val, pick_subpart, boundary[], n_u,
          n_v, garr, stat);
    ...
}
```

8.1.10 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;
    int         ider;
    int         iside1;
    int         iside2;
    double      parvalue[ ];
    int         *leftknot1;
    int         *leftknot2;
    double      *k1;
    double      *k2;
    double      d1[ ];
    double      d2[ ];
    int         *jstat;
    . . .
    s2542(surf, ider, iside1, iside2, parvalue, leftknot1, leftknot2, k1, k2, d1, d2,
          jstat);
    . . .
}
```

8.1.11 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;
    int         ider;
    int         iside1;
    int         iside2;
    double      parvalue[ ];
    int         *leftknot1;
    int         *leftknot2;
    double      norcurv[ ];
    int         *jstat;
    . . .
    s2544(surf, ider, iside1, iside2, parvalue, leftknot1, leftknot2, norcurv, jstat);
    . . .
}
```

8.1.12 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;
    int         curvature_type;
    int         export_par_val;
    int         pick_subpart;
    double      boundary[ ];
    int         n_u;
    int         n_v;
    double      scale;
    double      **garr;
    int         *stat;
    ...
    s2545(surf, curvature_type, export_par_val, pick_subpart, boundary[], n_u,
          n_v, scale, garr, stat);
    ...
}
```

Chapter 9

Surface Utilities

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

9.1 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.

9.1.1 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 $w_1\mathbf{p}_{1,1}, w_{1,1}, w_{1,2}\mathbf{p}_{1,2}, w_{1,2}, \ldots, w_{n_1,n_2}\mathbf{p}_{n_1,n_2}, w_{n_1,n_2}$ 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];
    double      knot2[7];
    double      coef[60];
    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);
    ...
}
```

9.1.2 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);
    . . .
}
```

9.1.3 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);
    ...
    freeSurf(surf);
    ...
}
```

9.2 Evaluation

9.2.1 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 310.

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;
    int         der = 2;
    double      parvalue[2];
    int         leftknot1 = 0;
    int         leftknot2 = 0;
    double      derive[18];
    double      normal[3];
    int         stat;
    ...
    s1421(surf, der, parvalue, &leftknot1, &leftknot2, derive, normal, &stat);
    ...
}
```

9.2.2 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. $d = surf \rightarrow dim$ 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;
    int         der1 = 2;
    int         der2 = 1;
    double      parvalue[2];
    int         leftknot1 = 0;
    int         leftknot2 = 0;
    double      derive[18];
    int         stat;
    ...
    s1424(surf, der1, der2, parvalue, &leftknot1, &leftknot2, derive, &stat);
    ...
}
```

9.2.3 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;
    int         ider = 1;
    int         iside1;
    int         iside2;
    double      epar[2];
    int         ilfs = 0;
    int         ilft = 0;
    double      eder[9];
    double      enorm[3];
    int         jstat = 0;
    ...
    s1422(ps1, ider, iside1, iside2, epar, &ilfs, &ilft, eder, enorm, &jstat);
    ...
}
```

9.2.4 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;
    int         ider1 = 1;
    int         ider2 = 1;
    int         iside1;
    int         iside2;
    double      epar[2];
    int         ileft1 = 0;
    int         ileft2 = 0;
    double      eder[12];
    int         jstat = 0;
    ...
    s1425(ps1, ider1, ider2, iside1, iside2, epar, &ileft1, &ileft2, eder, &jstat);
    ...
}
```

9.2.5 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;
    int         ider;
    int         m1;
    double      *x;
    int         m2;
    double      *y;
    double      eder[ ];
    double      norm[ ];
    int         *jstat;
    . . .
    s1506(ps1, ider, m1, x, m2, y, eder, norm, jstat);
    . . .
}
```

9.3 Subdivision

9.3.1 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;
    int         pardir;
    double      parval;
    SISLSurf    *newsurf1;
    SISLSurf    *newsurf2;
    int         stat;
    ...
    s1711(surf, pardir, parval, &newsurf1, &newsurf2, &stat);
    ...
}
```

9.3.2 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;
    double      epar1[3];
    int         inpar1 = 3;
    double      epar2[4];
    int         inpar2 = 4;
    SISLSurf    *rsnew = NULL;
    int         jstat = 0;
    ...
    s1025(ps, epar1, inpar1, epar2, inpar2, &rsnew, &jstat);
    ...
}
```

9.4 Picking Curves from a Surface

9.4.1 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;
    double      apar;
    int         idirec;
    SISLCurve   *rcurve = NULL;
    int         jstat = 0;
    . . .
    s1439(ps1, apar, idirec, &rcurve, &jstat);
    . . .
}
```

9.4.2 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;
    SISLCurve   *curve;
    double      epsge;
    double      maxstep;
    int         der;
    SISLCurve   *newcurve1;
    SISLCurve   *newcurve2;
    SISLCurve   *newcurve3;
    int         stat;
    ...
    s1383(surf, curve, epsge, maxstep, der, &newcurve1, &newcurve2,
          &newcurve3, &stat);
    ...
}
```

9.5 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;
    double      min1;
    double      min2;
    double      max1;
    double      max2;
    SISLSurf    *rsnew = NULL;
    int         jstat = 0;
    ...
    s1001(ps, min1, min2, max1, max2, &rsnew, &jstat);
    ...
}
```

9.6 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;
    SISLSurf    *newsurf;
    int         stat;
    ...
    s1440(surf, &newsurf, &stat);
    ...
}
```

9.7 Drawing

9.7.1 Draw a sequence of straight lines.

NAME

s6drawseq - Draw a broken line as a sequence of straight lines described by the array points. For dimension 3.

SYNOPSIS

void s6drawseq(*points*, *numpoints*)

double	*points*[ ];
int	*numpoints*;

ARGUMENTS

Input Arguments:

points	-	Points stored in sequence. i.e. $(x_0, y_0, z_0, x_1, y_1, z_1, \ldots)$.
numpoints	-	Number of points in the sequence.

NOTE

s6drawseq() is device dependent, it calls the empty dummy functions s6move() and s6line(). Before using it, make sure you have a version of these two functions interfaced to your graphic package.
More about s6move() and s6line() on pages 330 and 331.

EXAMPLE OF USE

```
{
    double      points[30];
    int         numpoints = 10;
    ...
    s6drawseq(points, numpoints)
    ...
}
```

9.7.2 Basic graphics routine template - move plotting position.

NAME

s6move - Move the graphics plotting position to a 3D point.

SYNOPSIS

void s6move(*point*)

double *point*[ ];

ARGUMENTS

Input Arguments:

point	-	A 3D point, i.e. (x, y, z), to move the graphics plotting position to.

NOTE

The functionality of s6move() is device dependent, so it is only an empty (`printf()` call) dummy routine. Before using it, make sure you have a version of s6move() interfaced to your graphic package.

EXAMPLE OF USE

```
{
    double     point[3];
    ...
    s6move(point)
    ...
}
```

9.7.3 Basic graphics routine template - plot line.

NAME

s6line - Plot a line between the current 3D graphics plotting position and a given 3D point.

SYNOPSIS

void s6line(*point*)
double *point*[ ];

ARGUMENTS

Input Arguments:

point - A 3D point, i.e. (x, y, z), to draw a line to, from the current graphics plotting position.

NOTE

The functionality of s6line() is device dependent, so it is only an empty (`printf()` call) dummy routine. Before using it, make sure you have a version of s6line() interfaced to your graphic package.

EXAMPLE OF USE

```
{
    double     point[3];
    . . .
    s6line(point)
    . . .
}
```

9.7.4 Draw constant parameter lines in a surface using piecewise straight lines.

NAME

s1237 - Draw constant parameter lines in a surface. The distance between the surface and the straight lines is less than a tolerance epsge. Also see NOTE!

SYNOPSIS

void s1237(*surf*, *numline1*, *numline2*, *epsge*, *stat*)

SISLSurf	**surf*;
int	*numline1*;
int	*numline2*;
double	*epsge*;
int	**stat*;

ARGUMENTS

Input Arguments:

surf	-	Pointer to the surface.
numline1	-	Number of constant parameter lines to be drawn in the first parameter direction.
numline2	-	Number of constant parameter lines to be drawn in the second parameter direction.
epsge	-	The maximal distance allowed between the drawn curves and the surface.

Output Arguments:

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

NOTE

This function calls s6drawseq() which is device dependent. Before using the function make sure you have a version of s6drawseq() interfaced to your graphic package. More about s6drawseq() on page 329.

EXAMPLE OF USE

```
{
    SISLSurf    *surf;
    int         numline1;
    int         numline2;
    double      epsge;
    int         stat;
    ...
    s1237(surf, numline1, numline2, epsge, &stat);
    ...
}
```

9.7.5 Draw constant parameter lines in a surface bounded by a closed curve in the parameter plane of the surface.

NAME

s1238 - Draw constant parameter lines in a surface. The lines are limited by a closed curve lying in the parameter plane of the surface, i.e. a 2D curve. All lines are drawn as piecewise straight lines. Also see NOTE!

SYNOPSIS

void s1238(*surf*, *curve*, *numline1*, *numline2*, *epsco*, *epsge*, *stat*)

SISLSurf	**surf*;
SISLCurve	**curve*;
int	*numline1*;
int	*numline2*;
double	*epsco*;
double	*epsge*;
int	**stat*;

ARGUMENTS

Input Arguments:

surf	-	Pointer to the surface.
curve	-	The 2D curve, in the parameter plane of the surface, bounding the part of the surface that is to be drawn.
numline1	-	Number of constant parameter lines to be drawn in the first parameter direction.
numline2	-	Number of constant parameter lines to be drawn in the second parameter direction.
epsco	-	Not in use!
epsge	-	The maximal distance allowed between the drawn curves and the surface.

Output Arguments:

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

NOTE

This function calls s6drawseq() which is device dependent. Before using the function make sure you have a version of s6drawseq() interfaced to your graphic package. More about s6drawseq() on page 329.

EXAMPLE OF USE

```
{
    SISLSurf     *surf;
    SISLCurve    *curve;
    int          numline1;
    int          numline2;
    double       epsco;
    double       epsge;
    int          stat;
    ...
    s1238(surf, curve, numline1, numline2, epsco, epsge, &stat);
    ...
}
```

Chapter 10

Data Reduction

10.1 Curves

10.1.1 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;
    double      eps[ ];
    int         startfix;
    int         endfix;
    int         iopen;
    int         itmax;
    SISLCurve   **newcurve;
    double      maxerr[ ];
    int         *stat;
    ...
    s1940(oldcurve, eps, startfix, endfix, iopen, itmax, newcurve, maxerr, stat);
    ...
}
```

10.1.2 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[];
    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;
    ...
    s1961(ep, im, idim, ipar, epar, eeps, ilend, irend, iopen, afctol, itmax, ik, rc,
          emxerr, jstat);
    ...
}
```

10.1.3 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[];
    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;
    ...
    s1962(ep, ev, im, idim, ipar, epar, eeps, ilend, irend, iopen, itmax, rc, emxerr,
        jstat);
    ...
}
```

10.1.4 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*;
double	*eeps*[ ];
int	*ilend*;
int	*irend*;

```
    int          iopen;
    int          itmax;
    SISLCurve    **rc;
    int          *jstat;
    ...
    s1963(pc, eeps, ilend, irend, iopen, itmax, rc, jstat);
    ...
}
```

10.2 Surfaces

10.2.1 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;
```

```
    double      eps[];
    int         edgefix[4];
    int         iopen1;
    int         iopen2;
    double      edgeps[];
    int         opt;
    int         itmax;
    SISLSurf    **newsurf;
    double      maxerr[];
    int         *stat;
    ...
    s1965(oldsurf, eps, edgefix, iopen1, iopen2, edgeps, opt, itmax, newsurf,
          maxerr, stat);
    ...
}
```

10.2.2 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[ ];
    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;
    ...
    s1966(ep, im1, im2, idim, ipar, epar1, epar2, eeps, nend, iopen1, iopen2,
          edgeps, afctol, iopt, itmax, ik1, ik2, rs, emxerr, jstat);
    ...
}
```

10.2.3 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[ ];
    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;
    . . .
    s1967(ep, etang1, etang2, eder11, im1, im2, idim, ipar, epar1, epar2, eeps,
          nend, iopen1, iopen2, edgeps, iopt, itmax, rs, emxerr, jstat);
    . . .
}
```

10.2.4 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;
    double     eeps[ ];
    int        nend[ ];
    int        iopen1;
    int        iopen2;
    double     edgeps[ ];
    int        iopt;
    int        itmax;
    SISLSurf   **rs;
    int        *jstat;
    ...
    s1968(ps, eeps, nend, iopen1, iopen2, edgeps, iopt, itmax, rs, jstat);
    ...
}
```

Chapter 11

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?

Index