Copy No. _____

Defence Research and Development Canada
Recherche et développement pour la défense Canada

DEFENCE R&D DÉFENSE

C++ classes for representing curves and surfaces

Part II: Splines

David Hally

Defence R&D Canada – Atlantic

Technical Memorandum
DRDC Atlantic TM 2006-255
January 2007

Canada

This page intentionally left blank.

C++ classes for representing curves and surfaces

Part II: Splines

David Hally

Defence R&D Canada – Atlantic

Technical Memorandum

DRDC Atlantic TM-2006-255

January 2007

Principal Author

Original signed by David Hally

David Hally

Approved by

Original signed by R. Kuwahara

R. Kuwahara
Head/Signatures

Approved for release by

Original signed by K. Foster

K. Foster
Chair/Document Review Panel

Abstract

Splines are in widespread use as a means of interpolating or approximating discrete data. They provide an efficient means of representing fully differentiable curves. This document describes a library of C++ classes which represent splines of different types; they include one and two parameter splines with a piecewise-polynomial representation, one and two parameter B-spline curves, one and two parameter non-uniform rational B-spline (NURB) curves, one and two parameter Hermite splines, Akima splines and standard cubic splines.

The spline classes are based on the CurveLib library of C++ classes which represent fully differentiable curves of a more general nature. The spline classes inherit many convenient features of the CurveLib classes: they can be added to an arbitrary curve, scaled, composed with an arbitrary curve and inverted.

Résumé

Les splines sont très utilisées comme méthode d'interpolation ou d'approximation de données discrètes. Elles sont un moyen efficace de représenter des courbes continûment différentiables. Le présent document décrit une bibliothèque de classes C++ de différents types de splines : les splines à un ou deux paramètres avec représentation par morceaux de polynômes, les B-splines à un ou deux paramètres, les B-splines rationnelles non uniformes (NURB) à un ou deux paramètres, les splines de Hermite à un ou deux paramètres, les splines d'Akima, et les splines cubiques traditionnelles.

Les classes de splines ont été produites à partir de la bibliothèque de classes C++ CurveLib qui contient des courbes continûment différentiables plus générales. Les classes de splines héritent de plusieurs caractéristiques utiles des classes CurveLib : on peut les ajouter à une courbe arbitraire, les réduire, les composer avec une courbe arbitraire et les inverser.

This page intentionally left blank.

Executive summary

C++ classes for representing curves and surfaces: Part II: Splines

David Hally; DRDC Atlantic TM-2006-255; Defence R&D Canada – Atlantic; January 2007.

Background: The flow around ships and propellers affects their performance in many ways. Defence R&D Canada – Atlantic uses Computational Fluid Dynamics (CFD) to calculate these flows so that the performance of the hull and propellers can be evaluated and improved. Before the flow can be calculated, the geometry of the ship or propeller must be represented in a fashion that can be used by the CFD applications.

Splines are in widespread use as a means of interpolating or approximating discrete data. They provide an efficient means of representing fully differentiable curves.

The current document describes a library of C++ classes which represent splines; they can be used to represent the geometry of ship hulls and propellers in CFD applications.

Results: A library of C++ classes has been written; it represent splines of different types; they include one and two parameter splines with a piecewise-polynomial representation, one and two parameter B-spline curves, one and two parameter non-uniform rational B-spline (NURB) curves, one and two parameter Hermite splines, Akima splines and standard cubic splines.

The spline classes are based on the CurveLib library of C++ classes which represent fully differentiable curves of a more general nature. The spline classes inherit many convenient features of the CurveLib classes: they can be added to an arbitrary curve, scaled, composed with an arbitrary curve and inverted.

Significance: The library of C++ spline classes provides a useful tool for representing complex geometry for use in CFD programs. However, because splines are so useful as a means of interpolation, they can also be used in a wide variety of applications.

Sommaire

C++ classes for representing curves and surfaces: Part II: Splines

David Hally ; DRDC Atlantic TM-2006-255 ; R & D pour la défense Canada – Atlantique ; janvier 2007.

Contexte : L'écoulement de l'eau autour des navires et de leurs hélices influence leur comportement de différentes manières. R & D pour la défense Canada – Atlantique utilise la dynamique numérique des fluides pour calculer ces écoulements et ainsi évaluer et améliorer le comportement des carènes et des hélices. Pour calculer l'écoulement, on doit toutefois pouvoir représenter la géométrie du navire ou de l'hélice d'une manière compatible avec les logiciels de dynamique numérique des fluides.

Les splines sont très utilisées comme méthode d'interpolation ou d'approximation de données discrètes. Elles sont un moyen efficace de représenter des courbes continûment différentiables. Le présent document décrit une bibliothèque de classes C++ représentant des splines que l'on peut utiliser pour représenter la géométrie des carènes et des hélices dans les logiciels de dynamique numérique des fluides.

Résultats : Nous avons constitué une bibliothèque de classes C++ qui contient différents types de splines : les splines à un ou deux paramètres avec représentation par morceaux de polynômes, les B-splines à un ou deux paramètres, les B-splines rationnelles non uniformes (NURB) à un ou deux paramètres, les splines de Hermite à un ou deux paramètres, les splines d'Akima, et les splines cubiques traditionnelles.

Les classes de splines ont été produites à partir de la bibliothèque de classes C++ CurveLib, qui contient des courbes continûment différentiables plus générales. Les classes de splines héritent de plusieurs caractéristiques utiles des classes CurveLib : on peut les ajouter à une courbe arbitraire, les réduire, les composer avec une courbe arbitraire et les inverser.

Importance : La bibliothèque de classes C++ de splines constitue un outil précieux pour la représentation d'objets à la géométrie complexe dans les logiciels de dynamique numérique des fluides. Qui plus est, parce que les splines sont très utiles comme méthode d'interpolation, elles peuvent être appliquées à différents domaines.

Table of contents

Abstract i

Résumé i

Executive summary iii

Sommaire iv

Table of contents v

List of figures vii

1 Introduction 1

2 Overview of the spline classes 1

3 Knot sequences 6

4 The base spline classes 8

5 General splines 11

5.1 Two-parameter general splines 12

6 PP-splines 12

6.1 Two-parameter PP-splines 15

7 Linear splines 17

7.1 Two-dimensional linear splines 17

8 Hermite splines 18

8.1 Slope generators 19

8.1.1 Parabolic slope generator 19

8.1.2 AkimaSlopeGenerator 20

8.1.3 Cubic slope generator 21

8.1.4 Monotonic slope generator 23

8.1.5 Two-parameter slope generators 24

8.2 One-parameter Hermite splines . . . 25
8.3 Cubic splines . . . 27
8.4 Akima splines . . . 28
8.5 Two-parameter Hermite splines . . . 29
8.6 Hermite interpolation of several curves . . . 33
9 B-splines . . . 34
9.1 One-parameter B-spline curves . . . 35
9.2 Two-parameter B-spline curves . . . 36
10 Non-uniform rational B-splines (NURBs) . . . 37
10.1 One-parameter NURBs . . . 38
10.2 Two-parameter NURBs . . . 39
11 Approximating generic curves with splines . . . 40
11.1 Approximation using Hermite splines . . . 40
11.2 Approximation using B-spline curves . . . 42
12 Converting between spline representations . . . 45
12.1 Converting from PP-splines to B-spline curves . . . 45
12.2 Converting from B-spline curves to PP-splines . . . 46
13 Concluding remarks . . . 47
References . . . 48
Index . . . 49

List of figures

Figure 1: The inheritance relations between the spline classes. 2

Figure 2: Region in the $\alpha\beta$ plane for which monotonicity is preserved. . . . 24

Figure 3: The approximation of e^{-x^2} over $[-5, 5]$ using a Hermite spline with seven equally spaced knots. 41

Figure 4: The approximation of e^{-x^2} over $[-5, 5]$ using a Hermite spline to an accuracy of 10^{-3}. 42

Figure 5: The approximation of e^{-x^2} over $[-5, 5]$ using a B-spline curve of order four with 13 knots (six polynomial segments). 44

This page intentionally left blank.

1 Introduction

Splines are in widespread use as a means of interpolating or approximating discrete data. They provide an efficient means of representing fully differentiable curves. This document describes a library of C++ classes which represent splines of different types. It is based on the CurveLib library of C++ classes[1] which represent fully differentiable curves of a more general nature. The spline classes inherit many convenient features of the CurveLib classes: they can be added to an arbitrary curve, scaled, composed with an arbitrary curve, etc.

The spline classes were originally designed for representing the geometry of complex shapes (e.g. ship hulls and propellers) for use in Computation Fluid Dynamics applications. However, they may be put to many other uses as well.

There are three other documents which are companions to this one. The first[1] describes C++ classes for implementing differentiable multi-parameter functions. The classes described there are used by the spline classes described in this document. The second[2] describes classes for saving curves, including splines, in files in Initial Graphic Exchange Standard (IGES) format[3] or defining curves from the data in an IGES file. The third[4] describes classes for defining distributions, differentiable functions which map $[0, 1]$ to $[0, 1]$, which are themselves important in a wide variety of applications.

2 Overview of the spline classes

A spline is defined on an increasing set of points $\{t_i, 0 \leq i \leq N-1\}$ called the knot sequence. Between each pair of neighbouring knots $[t_i, t_{i+1})$, the spline is represented by a curve, $f_i(x)$. The spline usually has imposed requirements on continuity at the knots which provides constraints for the functions $f_i(x)$.

Most commonly the functions $f_i(x)$ are polynomials: cubic polynomials are by far the most popular. However, the $f_i(x)$ can be of any form whatever: the class `GeneralSpline<V,F>` represents a spline whose constituent curves are completely arbitrary.

When the constituent curves are polynomials, there are two common methods for representing them:

1. by specifying the coefficients of the polynomials; following de Boor[5] we will call this the PP-representation and call a spline using the PP-representation a PP-spline;
2. as a sum of B-spline basis functions.

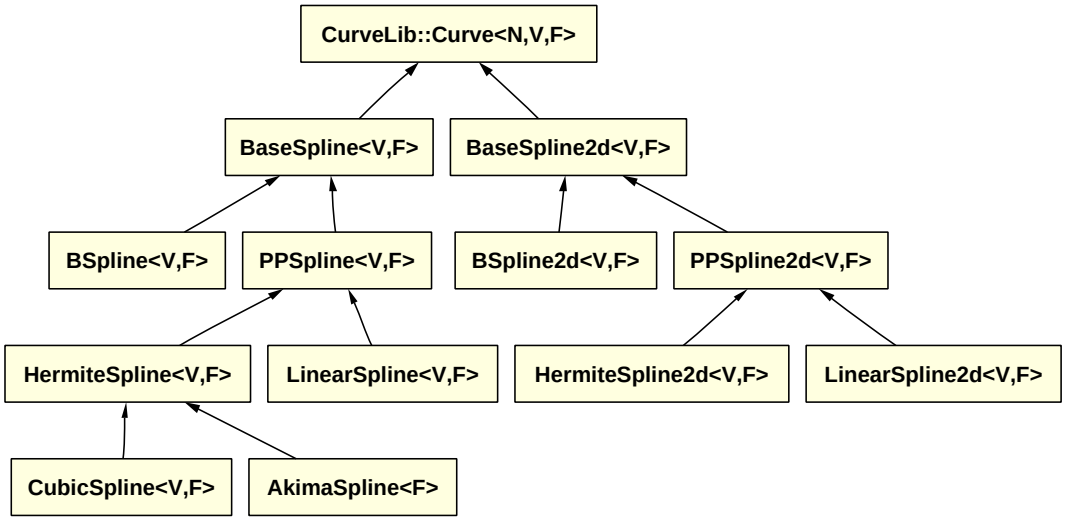

Figure 1: *The inheritance relations between the spline classes. The arrows point from a derived class to its base class.*

The classes `PPSpline<V,F>` and `PPSpline2d<V,F>` represent one-parameter and two-parameter PP-splines. The order of the polynomials comprising the splines is completely arbitrary. The classes `BSpline<V,F>` and `BSpline2d<V,F>` represent one-parameter and two-parameter splines using B-spline basis functions: we will call these spline B-spline curves. Again the order of the constituent polynomials is arbitrary.

The class `LinearSpline<V,F>` connects a collection of data points with straight line segments. It is a specialization of a `PPSpline<V,F>` whose polynomial segments are of order two. Similarly, `LinearSpline2d<V,F>` connects a two-dimensional array of data points with bilinear patches.

The classes `HermiteSpline<V,F>` and `HermiteSpline2d<V,F>` represent one and two-parameter Hermite splines. These are specializations of fourth order PP-splines in which the spline is uniquely determined by specifying the values and slopes at the knots. A standard cubic spline can be represented as a Hermite spline with the slopes generated so that the spline has continuous second derivatives: see Sections 8.1.3 and 8.3.

The inheritance relations between the spline classes is shown in Figure 1.

All the classes described in this document are templates. To define the attributes required of template arguments we will use the Standard Template Library[6] notion of a concept. Each class using a template argument imposes certain requirements on objects of the type of the template argument. When different classes impose the same set of requirements on a template argument, it is convenient to give that set

of requirements a name; a *concept* is the named set of requirements for a template argument. The spline library uses several different concepts to define the attributes of the template arguments of its classes. The details of these concepts are defined in Annexes of Reference 1.

All the C++ classes described in this document are defined in the header files `Spline.h` and `HermiteSpline.h`. They all belong to the namespace `Spline`.

All the classes representing one-parameter spline curves are derived from the base class `BaseSpline<V,F>`. This class does little except provide a common means of retrieving the knot sequence of the spline. It can usually be assumed that a spline is C^∞ in the region between two knots (this is certainly true of PP-splines and B-spline curves). Therefore the knot sequence provides a list of the locations where the spline or one of its derivatives might be discontinuous.

Similarly, the classes representing two-parameter spline curves are derived from the base class `BaseSpline2d<V,F>`. It provides member functions for retrieving the knot sequences associated with each parameter.

`BaseSpline<V,F>` and `BaseSpline2d<V,F>` are derived from `Curve<1U,V,F>` and `Curve<2U,V,F>`, respectively, in namespace `CurveLib` and described in Reference 1, Section 2. `Curve<N,V,F>` represents a differentiable function having `N` arguments: $f(x_1, \ldots, x_N)$. The value which the function returns is of type `V` and the type of each of the arguments, x_i, is `F`. The splines inherit arithmetic and composition operators from `Curve<N,V,F>` so that they can easily be combined to make new curves: see Reference 1, Sections 4 and 5.

The spline classes are all templates inheriting the template arguments `V` and `F` from `Curve<N,V,F>`. The number of parameters is always fixed at either one or two, so `N` is not needed. The template parameter `F` is the type of a parameter used when evaluating a spline. It is also the type of a knot. For the latter purpose it is necessary that two `F`s be comparable. Therefore the spline classes require that `F` is a model of a Comparable Scalar Object: see Reference 1, Annex A.3. Loosely speaking, this requires `F` to model a floating point number for which relational operators (e.g. less than, greater than) are defined. Typically `F` will be `float` or `double`. The template argument `F` is always optional; if it is omitted it defaults to `double`.

The spline classes also require that an `F` can be written to a `std::ostream`: i.e. that `operator<<(std::ostream&,const F&)` is defined. This is so that error messages can be constructed if there are problems with the arguments of the splines.

The template argument `V` is the type of the value returned by the spline. The base class `Curve<N,V,F>` requires that it be a model of an Arithmetic Object (see Reference 1, Annex A.1). This requires that `V` has the arithmetic operators one normally

expects from scalars, vectors and matrices: e.g. addition, subtraction, multiplication, division by a scalar, etc. Some of the spline classes (`BSpline<V,F>`, `BSpline2d<V,F>`, `HermiteSpline<V,F>`, and `HermiteSpline2d<V,F>`) impose the additional requirement that `V` is also a model of an Absolute Object: see Reference 1, Annex A.5. An Absolute Object includes a notion of size via the function `abs`. In the spline classes this is used when approximating a curve by a spline; the notion of size is used to determine how close the approximation is by evaluating the size of the difference in values returned by the curve and the spline.

Each spline class defines the type `ParamType` to represent its list of parameters. This is an array of parameter values for which a full set of arithmetic functions is defined (it is a model of Vector Object with respect to `F`: see Reference 1, Annex A.4). In the current implementation, when the spline has only only parameter, `ParamType` will be the same as `VecN<1U,F>` from the namespace `VecMtx`; when there are two parameters `ParamType` will be the same as `VecN<2U,F>`. A list of function prototypes for `VecN<N,F>` is given in Reference 1, Annex B.

Each spline class also defines the type `DerivType`. It is an array of unsigned integers which give the number of derivatives to be taken with respect to each parameter. In the current implementation, when the spline has only only parameter, `DerivType` will be the same as `Derivs<1U>` from the namespace `CurveLib`; when there are two parameters `DerivType` will be the same as `Derivs<2U>`. A list of function prototypes for `Derivs<N>` is given in Reference 1, Annex D.

All exceptions thrown by the spline classes are derived from the base class `Error` described in Reference 1, Annex F. Each exception stores a message describing the error; the message can be retrieved using the member function `get_msg()`.

All the spline classes have a default constructor. When this constructor is used the spline is initially undefined. Before it is evaluated it must be defined either by assignment to another spline or by calling member functions to define its representation. In the lists of member functions in the following sections the default constructor will usually be omitted. Similarly destructors and copy constructors are omitted.

Since a spline may remain undefined if a default constructor is used, the following member function (inherited from `Curve<N,V,F>`) is provided for determining whether the spline is defined or not.

`bool is_defined() const`
: Returns `true` if the curve has been defined; `false` if it has not.

An attempt to evaluate an undefined function will cause an `Error` exception to be thrown.

Any spline may be evaluated using the following two member functions (inherited from `Curve<N,V,F>`):

`V operator()(const ParamType &p) const`
: Returns the value of the curve for parameters `p`.

`V operator()(const ParamType &p, const DerivType &d) const`
: Returns the value of the differentiated curve for parameters `p`. The number of derivatives to be taken with respect to each parameter is specified by `d`. If all values of `d` are zero, then this function is equivalent to `operator()(const ParamType &p) const`.

For example, to evaluate a `PPSpline2d<double>` (a two-parameter PP-spline whose knots are of type `double` and which returns a value of type `double`) at the point (0.5,1.8) we could use:

```
using namespace Spline;
PPSpline2d<double> spline;
// ... code to define the spline ...
PPSpline2d<double>::ParamType p(0.5,1.8);
double value = spline(p);
```

To evaluate its second derivative with respect to y (the second parameter) at (0.5,1.8) use

```
PPSpline2d<double>::ParamType p(0.5,0.8);
PPSpline2d<double>::DerivType d(0,2);
double dvalue = spline(p,d);
```

The two parameter splines also define the the following two member functions for evaluating the spline; they are usually more convenient.

`V operator()(F x, F y) const`
: Returns the value of the spline at `(x,y)`.

`V operator()(F x, F y, unsigned int dx, unsigned int dy) const`
: Returns the value of the spline at `(x,y)` differentiated `dx` times with respect to `x` and `dy` times with respect to `y`. If both `dx` and `dy` are zero, then this function is equivalent to `operator()(F x, F y) const`.

Using these operators, the code above for evaluating the `PPSpline2d<double>` could be replaced by:

```
using namespace Spline;
PPSpline2d<double> spline;
```

```
// ... code to define the spline ...
double value = spline(0.5,1.8);
double dvalue = spline(0.5,1.8,0,2);
```

Similarly, the one parameter splines define the the following two member functions for evaluating the spline.

`V operator()(F x) const`
: Returns the value of the curve for parameter `x`.

`V operator()(F x, unsigned int d) const`
: Returns the value of the curve at `x` differentiated `d` times. If `d` is zero, then this function is equivalent to `operator()(F x) const`.

For example, to evaluate a `PPSpline<double>` and its second derivative at 0.5 we could use.

```
using namespace Spline;
PPSpline<double> spline;
// ... code to define the spline ...
double value = spline(0.5);
double dvalue = spline(0.5,2);
```

3 Knot sequences

The class `KnotSeq<F>` represents a knot sequence. Its template parameter `F` is the type of each knot. It must be a model of a Comparable Scalar Object (see Reference 1, Annex A.3): usually `F` is `float` or `double`. The knot sequence is a non-decreasing sequence of floating point values. The non-decreasing nature of the knots is *not* guaranteed but can be checked using the member function `check()`.

PP-splines require that their knot sequences are strictly increasing: two or knots with the same value are not allowed. Several of the `KnotSeq<F>` member functions have arguments that assist in ensuring that this is the case.

`KnotSeq<F>` defines the type `SizeType` to represent the size of the knot sequence and indices into it. It is an unsigned integer type currently defined to be a `size_t`; the actual type may vary from system to system.

`KnotSeq<F>` has the following public member functions:

`KnotSeq()`
: A default constructor: the knot sequence remains empty.

`explicit KnotSeq(SizeType n)`
: Makes a knot sequence of length `n`. The values of the knots remain undefined.

`KnotSeq(const KnotSeq<F> &knts, bool no_multiples = false)`
: Copies `knts`. If `no_multiples` is true, duplicates of knots will be discarded. This is convenient for converting a knot sequence for a B-spline curve (which allows multiple knots) to the equivalent knot sequence for a PP-spline (which forbids multiple knots).

`SizeType size() const`
: Returns the length of the knot sequence.

`void resize(SizeType n)`
: Modifies the knot sequence so that it has exactly `n` knots, inserting new knots at the end or erasing elements from the end if necessary. Any knots that are inserted will have undefined values.

`F& operator[](SizeType i);`
`F operator[](SizeType i) const`
: Returns the $\texttt{i}^{\text{th}}$ knot.

`F front() const`
: Returns the first knot.

`F back() const`
: Returns the last knot.

`bool check(bool strictly_increasing = false) const`
: Returns true if the elements are non-decreasing. If `strictly_increasing` is true, no two knots can be equal.

`void reorder()`
: Makes the elements non-decreasing.

`SizeType insert(F x, bool no_multiples = false)`
: Inserts a new knot with value `x` into the sequence. The insertion maintains the monotonicity of the knot sequence: i.e. if the knot sequence was non-decreasing before the insertion, it will also be non-decreasing after the insertion. Returns the location of `x` in the new knot sequence.

  If `no_multiples` is true, then `x` is not inserted if it is already present. If `x` is not inserted, the value returned will be `size()`.

`SizeType find(const F &x) const`

Finds `i` such that $t[i] \leq x < t[i+1]$. Throws an `OutOfRange` exception if `x` is not in the range of the knot sequence. `OutOfRange` is derived from the base class `Error` (see Reference 1, Annex F).

`std::vector<std::pair<unsigned,unsigned> > multiple_knots() const`

Returns a list of the multiple knots and their multiplicities. The first value in each pair is the location in the knot sequence of the first knot of a group of multiple knots. The second value is the number of knots in the group. Only groups with two or more knots are returned.

4 The base spline classes

All the one parameter spline classes are derived from the base class `BaseSpline<V,F>`, where the template parameter `V` is the type of the value of the spline (a model of an Arithmetic Object) and the template parameter `F` is the type of each knot (a model of a Comparable Scalar Object). `BaseSpline<V,F>` is derived from `Curve<1U,V,F>` in namespace `CurveLib` and has the following member functions in addition to the default and copy constructors and the assignment operator:

`typedef CurveLib::Curve<1U,V,F>::ParamType ParamType`

The type of the parameter list represented as a one element vector. This is defined by the base class `Curve<1U,V,F>` but is rarely used in `BaseSpline<V,F>` as it is usually easier to use parameters of type `F`: see `operator()(F x)` below.

`typedef CurveLib::Curve<1U,V,F>::DerivType DerivType`

The type of the derivative specifier represented as a one element vector of unsigned integers. This is defined by the base class `Curve<1U,V,F>` but is rarely used in `BaseSpline<V,F>` as it is usually easier to use an unsigned integer to specify the number of derivatives: see `operator()(F x, unsigned d)` below.

`unsigned number_of_knots() const`

Returns the number of knots.

`const KnotSeq<F>& knots() const`

Returns the knot sequence.

`bool is_defined() const`

Returns true if the spline is defined; false otherwise.

`F operator()(F x) const`

Returns the value of the spline at `x`.

`F operator()(F x, unsigned d) const`

Returns the value of the `d`$^{\text{th}}$ derivative of the spline at `x`.

`F operator()(const ParamType &p) const`

Returns the value of the spline at `p`: equivalent to `operator()(p[0])`. It is defined because it is inherited from the base class, but is rarely used because it is usually more convenient to use the version whose argument is an `F`.

`F operator()(const ParamType &p, const DerivType &d) const`

Returns the value of the differentiated curve at `p`. This function returns the same value as `operator()(p[0],d[0])`. It is defined because it is inherited from the base class, but is rarely used because it is usually more convenient to use the version whose arguments are of type `F` and `unsigned`.

```
template<unsigned M>
CurveLib::Curve<M,V,F> operator()(
  const CurveLib::Curve<M,ParamType,F> &c) const
```

Composition operator. If the spline is $s(x)$, this function returns a curve equivalent to $s \circ c$: i.e. a curve whose value at $(x_1, \ldots, x_M)$ is $s\big(c(x_1, \ldots, x_M)\big)$.

```
template<unsigned M>
Curve<M,V,F> operator()(const Curve<M,F,F> &c) const
```

Composition operator. Similar to the previous function but in this case the value of `c` is of type `F`, not `ParamType`.

If template member functions are not allowed by the compiler, then the two composition operators will only be defined for `M` equal to 1, 2 or 3.

All the two-parameter spline classes are derived from `BaseSpline2d<V,F>`, where the template parameters `V` and `F` have similar restrictions to those for `BaseSpline<V,F>`. `BaseSpline2d<V,F>` is derived from `Curve<2U,V,F>` in namespace `CurveLib` and has the following member functions in addition to the default and copy constructors and the assignment operator:

`typedef CurveLib::Curve<2U,V,F>::ParamType ParamType`

The type of the parameter list represented as a two element vector. This is defined by the base class `Curve<2U,V,F>` but is rarely used in `BaseSpline2d<V,F>` as it is usually easier to use parameters of type `F`: see `operator()(F x, F y)` below.

`typedef CurveLib::Curve<2U,V,F>::DerivType DerivType`

The type of the derivative specifier represented as a two element vector of unsigned integers. This is defined by the base class `Curve<2U,V,F>` but is rarely used in `BaseSpline<V,F>` as it is usually easier to use unsigned integers

to specify the number of derivatives to be taken: see `operator()(F x, F y, unsigned dx, unsigned dy)` below.

`unsigned number_of_x_knots() const`
Returns the number of knots associated with the first parameter.

`unsigned number_of_y_knots() const`
Returns the number of knots associated with the second parameter.

`const KnotSeq<F>& x_knots() const`
Returns the knot sequence associated with the first parameter.

`const KnotSeq<F>& y_knots() const`
Returns the knot sequence associated with the second parameter.

`bool is_defined() const`
Returns true if the spline is defined; false otherwise.

`V operator()(F x, F y) const`
Returns the value of the curve at `(x,y)`.

`V operator()(F x, F y, unsigned dx, unsigned dy) const`
Returns the value of the differentiated curve at `(x,y)`. The number of derivatives to be taken with respect to each parameter is specified by `dx` and `dy`.

`F operator()(const ParamType &p) const`
Returns the value of the spline at `p`: equivalent to `operator()(p[0],p[1])`. It is defined because it is inherited from the base class, but is rarely used because it is usually more convenient to use the version whose arguments are of type `F`.

`F operator()(const ParamType &p, const DerivType &d) const`
Returns the value of the differentiated curve at `p`. This function returns the same value as `operator()(p[0],p[1],d[0],d[1])`. It is defined because it is inherited from the base class, but is rarely used because it is usually more convenient to use the version whose arguments are of type `F` and `unsigned`.

```
template<unsigned M>
CurveLib::Curve<M,V,F> operator()(
  const CurveLib::Curve<M,ParamType,F> &c) const
```
Composition operator. If the spline is $s(x)$, this function returns a curve equivalent to $s \circ c$: i.e. a curve whose value at $(x_1, \ldots, x_M)$ is $s\big(c(x_1, \ldots, x_M)\big)$. If template member functions are not allowed by the compiler, then this function will only be defined for `M` equal to 1, 2 or 3.

5 General splines

A general spline is a curve constructed from several segments; each segment can be any one-parameter curve. General splines are represented by the class `GeneralSpline<V,F>`, where the template parameter `V` is the type of the value of the spline (a model of an Arithmetic Object) and the template parameter `F` is the type of each knot (a model of a Comparable Scalar Object). `GeneralSpline<V,F>` makes no guarantees on continuity where the segments meet. It is a specialization of the base class `Curve<1U,V,F>` in namespace `CurveLib`.

`GeneralSpline<V,F>` contains a knot sequence, t_i and an array of curves, c_i. If $t_i \leq x < t_{i+1}$, then the value of the curve is $c_i(x)$. The knot sequence of a general spline should be strictly increasing. Although it is not a strict requirement, it is usually advisable to ensure that each curve $c_i(x)$ is C^∞ on $t_i \leq x < t_{i+1}$. Functions using the general spline may then assume that the only discontinuities occur at the knots.

`GeneralSpline<V,F>` has the following public members in addition to the default and copy constructors, the assignment operator and the member functions inherited from `BaseSpline<V,F>`:

`typedef CurveLib::Curve<1U,V,F> SegCurveType`
: The type of curve used for each segment.

`typedef std::vector<SegCurveType> SegCurveArray`
: The type of the list of curves used for each segment.

`GeneralSpline(const KnotSeq<F> &kts, const SegCurveArray &segs)`
: Makes a general spline with knot sequence `kts` and segment curves `segs`. The length of `segs` must be one less than the length of `knts`. The curve `segs[i]` is used to evaluate the spline whenever the parameter is between `knts[i]` and `knts[i+1]`.

`void define(const KnotSeq<F> &kts, const SegCurveArray &segs)`
: Defines the spline curve by copying the knots in `kts` and the curves in `segs`.

`unsigned number_of_segments() const`
: Returns the number of segments.

`const SegCurveArray& get_segments() const`
: Returns the segment curves.

5.1 Two-parameter general splines

The class `GeneralSpline2d<V,F>` is a natural extension of a general spline to a two-parameter curve. We will denote the first parameter by x and the second by y. Their corresponding knot sequences are $\{s_i, 0 \le i \le N_x - 1\}$ and $\{t_j, 0 \le j \le N_y - 1\}$. A curve must be specified for each coordinate patch $\{(x, y) : x \in [s_i, s_{i+1}); y \in [t_i, t_{i+1})\}$. `GeneralSpline2d<V,F>` has the following public members as well as those inherited from its base classes:

```
typedef CurveLib::Curve<2U,V,F> PatchCurveType
```
The type of curve used for each patch.

```
typedef std::vector<std::vector<PatchCurveType> > PatchCurveArray
```
The type of an array of patch curves.

```
GeneralSpline2d()
```
The default constructor: the spline curve remains undefined.

```
GeneralSpline2d(const KnotSeq<F> &xkts, const KnotSeq<F> &ykts,
                const PatchCurveArray &pcrvs)
```
Makes a two-dimensional general spline with knot sequences `xkts` and `ykts` and with patch curves `pcrvs`. The length of `pcrvs` must be one less than the length of `ykts`. For each `j`, the length of `pcrvs[j]` must be one less than the length of `xkts`. The curve `pcrvs[j][i]` is used on the patch between `xkts[i]` and `xkts[i+1]`, and between `ykts[j]` and `ykts[j+1]`.

```
void define(const KnotSeq<F> &xkts, const KnotSeq<F> &ykts,
            const PatchCurveArray &pcrvs)
```
Defines the spline in a similar manner to the constructor with the same arguments.

```
const PatchCurveArray& get_patch_curves() const
```
Returns the curves for each patch.

6 PP-splines

In the PP-representation, the curve between each pair of knots is a polynomial of order k:

$$f_i(x) = \sum_{n=0}^{k-1} \frac{c_{n,i}(x - t_i)^n}{n!} \tag{1}$$

The normalization by $n!$ makes the coefficient $c_{n,i}$ equal to the n^{th} derivative of $f_i(x)$ at t_i.

The knot sequence of a PP-spline should be strictly increasing.

PP-splines are represented by the class `PPSpline<V,F>` where `V` is the type of the spline value (a model of an Arithmetic Object) and `F` is the type of each knot (a model of a Comparable Scalar Object).

Each PP-spline stores the knot sequence, t_i, and an array of spline coefficients, $c_{n,i}$. We first define the class `PPCoefArray<V>` used to represent the coefficients. The template argument `V` is the type of each coefficient; it is the same as the return type of the spline. The coefficients are organized as a $N \times k$ array where N is the number of knots and `k` is the order of the spline. The public members of `PPCoefArray<V>` are:

`PPCoefArray()`
: Default constructor: the coefficients are undefined.

`PPCoefArray(unsigned k, unsigned nkt)`
: Creates a `CoefArray<V>` for a spline of order `k` with `nkt` knots. The coefficients remain undefined.

`void resize(unsigned k, unsigned nkt)`
: Resize the array of coefficients for a spline of order `k` with `nkt` knots. Any new coefficients added because the order or number of knots has increased will be undefined.

`unsigned order() const`
: Returns the order of the spline.

`unsigned number_of_knots() const`
: Returns the number of knots.

`V& operator()(unsigned i, unsigned j);`
`const V& operator()(unsigned i, unsigned j) const`
: Returns the $\texttt{i}^{\text{th}}$ coefficient at knot `j`: i.e. the $\texttt{i}^{\text{th}}$ derivative of the spline at knot `j`. The index `i` must be in the range $[0, k-1]$ and `j` must be in the range $[0, N-1]$; there is no checking to ensure that this is the case.

Notice that `PPCoefArray<V>` stores coefficients for the last knot even though these values are not strictly required, since the range of the spline is restricted to $[t_0, t_{N-1}]$ and the value at t_{N-1} could be obtained from the coefficients at t_{N-2}. The reason for including the coefficients at t_{N-1} is to ensure that the spline will return the correct

value at all knots with no round-off error. This can be important, especially when the value of the spline is used as the argument to another curve with a restricted range (e.g. `Sqrt<F>` or `ArcSin<F>` in namespace `CurveLib`).

`PPSpline<V,F>` is a specialization of `BaseSpline<V,F>` and has the following public members in addition to the default and copy constructors, the assignment operator and the member functions inherited from `BaseSpline<V,F>`:

`typedef PPCoefArray<V> CoefArray`
: The type of the array storing the spline coefficients.

`PPSpline(const KnotSeq<F> &knt, const CoefArray &cf)`
: Constructs the spline with given knots and coefficients. The knot sequence must be strictly increasing. The order, `k`, is obtained from the coefficient array. The length of the knot sequence must agree with the number of knots used by `cf`; if not a `PPInconsistentData` exception is thrown.

`void define(const KnotSeq<F> &knt, const CoefArray &cf)`
: Define the spline with given knots and coefficients. The arguments are the same as those in the constructor above.

`void redefine(const CoefArray &cf)`
: Redefine the spline to have the coefficients in `cf`. The current knot sequence is retained. The length of the knot sequence must agree with the number of knots used by `cf`; if not a `PPInconsistentData` exception is thrown.

`void set_knots(const KnotSeq<F> &knots)`
: Sets the knot sequence to `knots`. Its length must be at least two and it must be strictly increasing; if not a `PPInconsistentData` exception is thrown. If the number of knots differs from the number of knots used by the current set of coefficients, the spline must not be evaluated until a new set of coefficients has been specified using `redefine()`.

`unsigned order() const`
: Returns the order of the spline.

`const CoefArray& coefficients() const`
: Returns the coefficient array.

6.1 Two-parameter PP-splines

Two-parameter PP-splines can be defined using two knot sequences: one for each parameter. We will denote the first parameter by x and the second by y. Their corresponding knot sequences are $\{s_i, 0 \le i \le N_x - 1\}$ and $\{t_j, 0 \le j \le N_y - 1\}$. The spline curve is defined by:

$$f_{ij}(x,y) = \sum_{m=0}^{k_x-1} \sum_{n=0}^{k_y-1} \frac{c_{mnij}(x-s_i)^m (y-t_j)^n}{m!n!} \quad \text{for } x \in [s_i, s_{i+1}) \text{ and } y \in [t_j, t_{j+1}) \quad (2)$$

where k_x and k_y are the spline orders in each direction and c_{mnij} are spline coefficients. The coefficient c_{mnij} is the m^{th} derivative with respect to x and n^{th} derivative with respect to y of the spline at (s_i, t_j).

The class `PPSpline2d<V,F>` represents a two-parameter PP-spline. It is a specialization of `BaseSpline2d<V,F>`.

Each `PPSpline2d<V,F>` stores two knot sequences and an array of spline coefficients, c_{mnij}. We first define the class `PPCoefArray2d<V>` to represent the spline coefficients. Its template argument `V` is the type of the coefficients which is also the type of the spline value. `PPCoefArray2d<V>` has the following members:

`PPCoefArray2d()`
: Default constructor: the coefficients are undefined.

`PPCoefArray2d(unsigned kx, unsigned ky, unsigned nktx, unsigned nkty)`
: Creates a `PPCoefArray2d<V>` for a spline with order `kx` and `nktx` knots in the x direction and order `ky` and `nkty` knots in the y direction.

`void resize(unsigned kx, unsigned ky, unsigned nktx, unsigned nkty)`
: Resize the array of coefficients for a spline with order `kx` and `nktx` knots in the x direction and order `ky` and `nkty` knots in the y direction.

`unsigned x_order() const`
: Returns the order of the spline in the x direction.

`unsigned y_order() const`
: Returns the order of the spline in the y direction.

`unsigned number_of_x_knots() const`
: Returns the number of knots in the x direction.

`unsigned number_of_y_knots() const`
: Returns the number of knots in the y direction.

```
V& operator()(unsigned m, unsigned n, unsigned i, unsigned j);
const V& operator()(unsigned m, unsigned n,
                    unsigned i, unsigned j) const
```

Returns the coefficient c_{mnij}: see Section 6.1. The index `m` must be in the range $[0, k_x - 1]$, `n` must be in the range $[0, k_y - 1]$, `i` must be in the range $[0, N_x - 1]$ and `j` must be in the range $[0, N_y - 1]$; there is no checking to ensure that this is the case.

The class `PPSpline2d<V,F>` has the following public members as well as the default and copy constructors, the assignment operator, and the member functions inherited from `BaseSpline2d<V,F>`:

```
typedef PPCoefArray2d<V> CoefArray
```

The type of the array of coefficients.

```
PPSpline2d(const KnotSeq<F> &xknt, const KnotSeq<F> &yknt,
           const CoefArray &cf)
```

Constructs the spline with given knots and coefficients. The knot sequences must have length of at least two and must be strictly increasing; otherwise a `PPInconsistentData` exception is thrown. The order of the spline in each direction is obtained from the coefficient array. The lengths of the knot sequences must agree with the number of knots used by `cf`; if not a `PPInconsistentData` exception is thrown.

```
void define(const KnotSeq<F> &xknt, const KnotSeq<F> &yknt,
            const CoefArray &cf)
```

Defines the spline with given knots and coefficients. The arguments are the same as those in the constructor above.

```
void redefine(const CoefArray &cf)
```

Defines the spline with given coefficients. The order of the spline in each direction is obtained from the coefficient array. The current knot sequences are used. The lengths of the knot sequences must agree with the number of knots used by `cf`; if not a `PPInconsistentData` exception is thrown.

```
void set_knots(const KnotSeq<F> &xknt, const KnotSeq<F> &yknt)
```

Sets the knot sequences. They must have length of at least two and must be strictly increasing; otherwise a `PPInconsistentData` exception is thrown. If the lengths of the knot sequences differ from those used by the current set of coefficients, the spline must not be evaluated until a new set of coefficients has been specified using `redefine()`.

`unsigned x_order() const`
: Returns the order of the spline in the x direction.

`unsigned y_order() const`
: Returns the order of the spline in the y direction.

`const CoefArray& coefficients() const`
: Returns the coefficient array.

7 Linear splines

The class `LinearSpline<V,F>` is a piecewise linear curve connecting a given set of points. It is a specialization of `PPSpline<V,F>` having a fixed order of two. It defines the following public members as well as the default and copy constructors, the assignment operator and the member functions inherited from `PPSpline<V,F>`:

`typedef std::vector<V> ValArray`
: The type of the array of values defining the spline.

`LinearSpline(const KnotSeq<F> &knt, const ValArray &vals)`
: Constructs the spline with given knots and values. The knot sequence must contain at least two knots and it must be strictly increasing; otherwise a `PPInconsistentData` exception is thrown.

`void define(const KnotSeq<F> &knt, const ValArray &vals)`
: Define the spline with given knots and values. The arguments are the same as for the constructor above.

`void redefine(const ValArray &vals)`
: Define the spline with given values. The current knot sequence is used. The length of `vals` must be the same as the number of knots; otherwise a `PPInconsistentData` exception is thrown.

7.1 Two-dimensional linear splines

The class `LinearSpline2d<V,F>` represents a two-dimensional spline which is piecewise linear in each direction. The spline is defined by rows of points in the x direction. A linear spline is calculated on each row. The values are then splined in the y direction to give the value of the spline. `LinearSpline2d<V,F>` defines the following public members as well as the default and copy constructors, the assignment operator and the member functions inherited from `PPSpline2d<V,F>`:

`typedef std::vector<std::vector<V> > ValArray`

The type of a two-dimensional array of values used to define the spline. If `v` is a `ValArray`, then `v[j][i]` gives the value of the spline at (x_i, y_j).

`LinearSpline2d(const KnotSeq<F> &xknt, const KnotSeq<F> &yknt, const ValArray &vals)`

Defines the spline using `xknt` and `yknt` for knot sequences and `vals` for the values at the points. The knot sequences must have length of at least two and must be strictly increasing; otherwise a `PPInconsistentData` exception is thrown. The length of `vals` must be the same as the length of `yknt`, and for each `i`, the length of `vals[i]` must be the same as the length of `xknt`; otherwise a `PPInconsistentData` exception is thrown.

`void define(const KnotSeq<F> &xknt, const KnotSeq<F> &yknt, const ValArray &vals)`

Defines the spline using knot sequences `xknt` and `yknt` and values `vals`. The restrictions on the arguments are the same as for the constructor above.

`void define(const ValArray &vals)`

Defines the spline using the values in `vals`. The current knots are used. The dimensions of the array `vals` must agree with the lengths of the knot sequences; if not a `PPInconsistentData` exception is thrown.

8 Hermite splines

If a value, y_i, and a slope, m_i, is specified at each knot, then there is a unique cubic spline which will interpolate the given values and have the given slopes. A spline specified in this way is called a Hermite spline. A Hermite spline can be represented as a PP-spline of order four as follows:

$$f_i(x) = y_i + m_i(x - t_i) + \frac{3M_i - m_{i+1} - 2m_i}{t_{i+1} - t_i}(x - t_i)^2 + \frac{m_{i+1} + m_i - 2M_i}{(t_{i+1} - t_i)^2}(x - t_i)^3 \qquad \text{for } t_i \le x \le t_{i+1} \tag{3}$$

where

$$M_i \equiv \frac{y_{i+1} - y_i}{t_{i+1} - t_i} \tag{4}$$

It can easily be verified that this curve passes through (t_i, y_i) and (t_{i+1}, y_{i+1}) and that its slopes at t_i and t_{i+1} are m_i and m_{i+1} respectively.

The C++ classes representing Hermite splines are defined in the header file `HermiteSpline.h`. All the classes defined in this file belong to the namespace `Spline`.

8.1 Slope generators

For a given knot sequence t_i and values y_i, Hermite splines differ only by the method with which they assign the slopes m_i. `SlopeGenerator<V,F>` is an abstract base class for classes which calculate the slopes. Its template parameter `V` is the type of the value of the spline (a model of Arithmetic Object) and its template parameter `F` is the type of each knot (a model of Comparable Scalar Object).

`SlopeGenerator<V,F>` has the following public members:

`typedef std::vector<V> ValArray`
: The type of the arrays of values and slopes.

`SlopeGenerator(bool p = false)`
: Constructor. If `p` is true, the slopes and values must be periodic: i.e. the values at the first and last knots must be the same and the slopes at the first and last knots must be the same.

`bool is_periodic() const`
: True if the spline is periodic.

```
virtual void calc_slopes(const KnotSeq<F> &knots,
                         const ValArray &v, ValArray &slopes) const = 0
```
: Given the knots and the values of the spline at the knots (in `v`), calculates the slopes at each of the knots and returns them in `slopes`.

Four specializations of `SlopeGenerator<V,F>` are defined. They each overload the pure virtual function `calc_slopes` to calculate the slopes in a different way.

8.1.1 Parabolic slope generator

A parabolic slope generator calculates the slopes at each point by fitting a parabola through the point and its two nearest neighbours. This results in the formula:

$$m_i = \frac{t_{i+1} - t_i}{t_{i+1} - t_{i-1}} M_{i-1} + \frac{t_i - t_{i-1}}{t_{i+1} - t_{i-1}} M_i \tag{5}$$

At the end points this formula will not work. Instead m_0 is set to the slope of the quadratic through the first three points:

$$m_0 = -\frac{(t_1 - t_0)y_2}{(t_2 - t_0)(t_2 - t_1)} + \frac{(t_2 - t_0)y_1}{(t_1 - t_0)(t_2 - t_1)} - \frac{(t_1 + t_2 - 2t_0)y_0}{(t_1 - t_0)(t_2 - t_0)} \tag{6}$$

A similar formula is used for m_N.

If $t_{i+1} - t_i = h$ for all i, then

$$m_i = \frac{M_{i-1} + M_i}{2}; \qquad m_0 = -\frac{3y_0 - 4y_1 + y_2}{2h} = \frac{3M_0 - M_1}{2} \tag{7}$$

The class `ParSlopeGenerator<V,F>` represents a parabolic slope generator. It is derived from the base class `SlopeGenerator<V,F>` and has the following constructor:

`ParSlopeGenerator(bool p = false)`
: If `p` is true, the slopes and values must be periodic.

8.1.2 AkimaSlopeGenerator

An Akima slope generator calculates the slopes at each point using the algorithm of Akima[7]. In the Akima algorithm the slope at t_i is a weighted average of M_{i-1} and M_i:

$$m_i = \frac{w_0 M_{i-1} + w_1 M_i}{w_0 + w_1} \tag{8}$$

The weight w_0 is calculated as the difference between M_i and M_{i+1}; similarly w_1 is the difference between M_{i-1} and M_{i-2}:

$$w_0 = |M_i - M_{i+1}|; \qquad w_1 = |M_{i-1} - M_{i-2}| \tag{9}$$

This choice tends to suppress wiggles in the interpolant since if the points at $i-2$, $i-1$ and i are nearly collinear, then m_i will be nearly equal to M_{i-1}. Similarly, if the points at i, $i+1$ and $i+2$ are nearly collinear, then m_i will be nearly equal to M_i.

If the spline is periodic, there is no difficulty implementing this algorithm at the end points. However, if it is not, then the evaluation of the slopes at t_0, t_1, t_{N-2} and t_{N-1} must be treated as special cases. If the the slope at t_0 is known, then the slope at t_1 can be calculated by setting M_{-1} to m_0. A similar procedure is used to calculate m_{N-2} when m_{N-1} is known. If the slope at t_0 is not known, then the slopes m_0 and m_1 are calculated using a parabolic slope generator. A similar procedure is used if m_{N-1} is not known.

An `AkimaSlopeGenerator<F>` calculates the slopes for a Hermite spline using the Akima algorithm. Because of the nature of the algorithm, it has only been implemented for scalar-valued splines, not for vector valued splines: i.e. it is derived from the base class `SlopeGenerator<F,F>`. `AkimaSlopeGenerator<F>` has the following constructor:

`AkimaSlopeGenerator(bool p = false)`
: If `p` is true, the slopes and values must be periodic.

8.1.3 Cubic slope generator

A standard cubic spline can be represented as a Hermite spline. It has continuity of both first and second derivatives at all interior knots. This condition can be written as

$$m_i \Delta t_{i+1} + 2m_{i+1}(\Delta t_i + \Delta t_{i+1}) + m_{i+2}\Delta t_i = 3M_i \Delta t_{i+1} + 3M_{i+1}\Delta t_i \tag{10}$$

with $\Delta t_i = t_{i+1} - t_i$.

Equation (10) generates a tri-diagonal linear system for the slopes, m_i provided that suitable conditions are placed on the slopes at the end points. There are three different end conditions:

1. The slope at an end point is set explicitly.

2. The second derivative at an end point is set; most commonly it is set to zero. If the second derivative at t_0 is to be set to d, the condition is:

$$2m_0 + m_1 = 3M_0 - \frac{d(t_1 - t_0)}{2} \tag{11}$$

   (see Equation (3)). Similarly, the condition if the second derivative at t_{N-1} is d is:

$$2m_{N-1} + m_{N-2} = 3M_{N-2} + \frac{d(t_{N-1} - t_{N-2})}{2} \tag{12}$$

3. The 'not-a-knot' condition is used. This condition specifies that the third derivative of the spline is continuous at the second or next to last knot. When the condition is applied at the second knot we find from Equation (3) that

$$m_0(\Delta t_1)^2 + m_1\left((\Delta t_1)^2 - (\Delta t_0)^2\right) - m_2(\Delta t_0)^2 = 2M_0(\Delta t_1)^2 - 2M_1(\Delta t_0)^2 \tag{13}$$

   The slope m_2 can be eliminated from this equation using Equation (10) with $i = 1$ to obtain.

$$m_0\Delta t_1 + m_1(\Delta t_0 + \Delta t_1) = \frac{M_0\Delta t_1(2\Delta t_1 + 3\Delta t_0) + M_1(\Delta t_0)^2}{\Delta t_0 + \Delta t_1} \tag{14}$$

   A similar condition is obtained if the not-a-knot condition is applied at the next to last knot:

$$m_{n-2}(\Delta t_{n-2} + \Delta t_{n-3}) + m_{n-1}\Delta t_{n-3} = \tag{15}$$

$$\frac{M_{n-3}(\Delta t_{n-2})^2 + M_{n-2}\Delta t_{n-3}(2\Delta t_{n-3} + 3\Delta t_{n-2})}{\Delta t_{n-3} + \Delta t_{n-2}} \tag{16}$$

If the spline is periodic, then the slopes and second derivatives at the end points must be equal

$$m_{N-2}\Delta t_0 + 2m_0(\Delta t_{N-2} + \Delta t_0) + m_1\Delta t_{N-2} = 3M_{N-2}\Delta t_0 + 3M_0\Delta t_{N-2} \quad (17)$$

$$m_{N-1} = m_0 \quad (18)$$

The resulting system of equations for the m_i is tri-diagonal except for non-zero elements in the upper right and lower left corners. Call the matrix for this linear system $\mathbf{A}$. Then we can write

$$A_{mn} = T_{mn} + a_m b_n \quad (19)$$

where

$$T_{mn} = \begin{cases} A_{00} - A_{0,N-1} & \text{for } m = n = 0 \\ A_{mn} & \text{for } n = m-1,\ m = 1, \ldots, N-1 \\ A_{mn} & \text{for } n = m,\ m = 1, \ldots, N-2 \\ A_{mn} & \text{for } n = m+1,\ m = 0, \ldots, N-2 \\ A_{N-1,N-1} - A_{N-1,0} & \text{for } m = n = N-1 \\ 0 & \text{otherwise} \end{cases} \quad (20)$$

$$a_m = \begin{cases} A_{0,N-1} & \text{for } m = 0 \\ A_{N-1,0} & \text{for } m = N-1 \\ 0 & \text{otherwise} \end{cases} \quad (21)$$

$$b_n = \begin{cases} 1 & \text{for } n = 0 \text{ or } N-1 \\ 0 & \text{otherwise} \end{cases} \quad (22)$$

(23)

The Sherman-Morrison-Woodbury formula then states that the solution to $\mathbf{Ax} = \mathbf{r}$ is

$$\mathbf{x} = \mathbf{p} - \frac{\mathbf{q}(\mathbf{b}\cdot\mathbf{p})}{1 + \mathbf{b}\cdot\mathbf{q}} \quad (24)$$

with $\mathbf{p}$ and $\mathbf{q}$ being the solutions to the linear systems:

$$\mathbf{Tp} = \mathbf{r}; \qquad \mathbf{Tq} = \mathbf{a} \quad (25)$$

Since $\mathbf{T}$ is tri-diagonal, $\mathbf{p}$ and $\mathbf{q}$ are easily obtained.

A cubic slope generator calculates the slopes for a standard cubic spline; it is represented by the class `CubicSlopeGenerator<V,F>` with the following public members:

`enum EndCondType { not_a_knot, slope, second_derivative }`
: The types of end-point conditions.

`CubicSlopeGenerator(bool p = false)`
: Makes a slope generator which calculates the slopes for a standard cubic spline.

If `p` is true, the spline will be periodic: i.e. the slopes and second derivatives at the end points will be the same. If `p` is false, the not-a-knot condition will be used at both ends of the spline.

`CubicSlopeGenerator(EndCondType low_cond, F low_val,`
`EndCondType high_cond, F high_val)`

Makes a slope generator which calculates the slopes for a standard cubic spline. The condition at the start of the spline is `lo_cond` with value (if it is needed), `lo_val`. The condition at the end of the spline is `hi_cond` with value `hi_val`. For example, to set the slope at the start to 1.0 and the second derivative at the end to 2.0, use `lo_cond = slope`, `lo_val = 1.0`, `hi_cond = second_derivative` and `hi_val = 2.0`. The value of the condition is ignored for the `not_a_knot` end condition.

8.1.4 Monotonic slope generator

A monotonic slope generator calculates slopes at each point by first applying another slope generator, then adjusting the slopes so that monotonicity of the data points is preserved.

On the knot interval $[t_i, t_{i+1}]$, the spline is monotonic if

$$\{0 \le \alpha \le 3 \quad \textbf{and} \quad 0 \le \beta \le 3\} \quad \textbf{or} \quad 3(\alpha+\beta-4)^2 + (\alpha-\beta)^2 - 12 \le 0 \tag{26}$$

where

$$\alpha \equiv \frac{m_i}{M_i}; \qquad \beta \equiv \frac{m_{i+1}}{M_i} \tag{27}$$

The region in the $\alpha\beta$ plane for which monotonicity is preserved is shown in Figure 2.

By scaling m_i and m_{i+1} so that $\sqrt{\alpha^2+\beta^2} = 3$, the algorithm of Fritsch and Carlson[8] guarantees monotonicity whenever the data points are increasing.

A `MonotonicSlopeGenerator<V,F>` implements the Fritsch-Carlson algorithm on every sub-interval $[t_i, t_{i+1}]$ for which $y_{i-1}, \ldots, y_{i+2}$ are strictly increasing. Since monotonicity has no meaning for a vector quantity, it is assumed that when `V` is a vector type, it can be adjusted component by component. This requires that `V` is a model of a Vector Object (see Reference 1, Annex A.4). A specialization of the class is defined for the case when `V = F`. Therefore, either `V` is a model of a Vector Object or `V = F`.

`MonotonicSlopeGenerator<V,F>` defines the following public members:

`MonotonicSlopeGenerator(const SlopeGenerator<V,F> &gen)`

Constructs the slope generator using `gen` to initialize the slopes. The periodicity will match the periodicity of `gen`. `gen` must persist longer than `*this`.

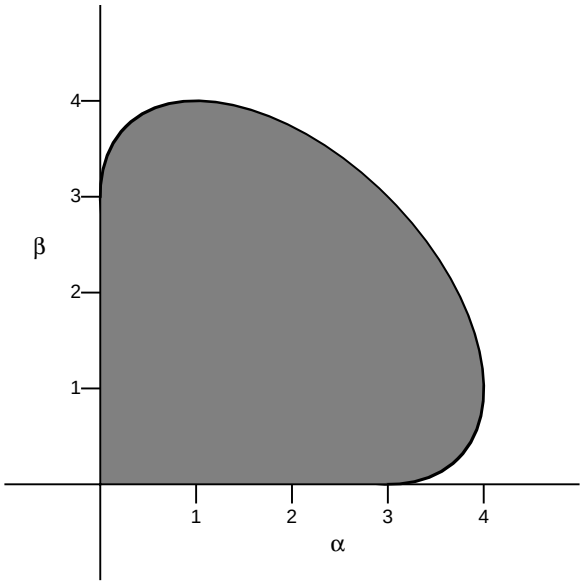

Figure 2: *Region in the $\alpha\beta$ plane for which monotonicity is preserved.*

```
virtual void adjust_slopes(const KnotSeq<F> &knots,
                           const ValArray &v,
                           ValArray &slopes) const
```

Adjusts the slopes in `slopes` to ensure that the spline will be monotonic.

8.1.5 Two-parameter slope generators

A two-parameter Hermite spline is uniquely defined by specifying values, the derivative with respect to x, the derivative with respect to y, and the derivative with respect to both x and y at each point, where x and y refer to the parametric directions. Different types of Hermite splines typically use different methods to define the derivatives from the values. A `SlopeGenerator2d<V,F>` implements this calculation using `SlopeGenerator<V,F>` for each parametric direction. It defines the following public members:

`typedef std::vector<typename SlopeGenerator<V,F>::ValArray> ValArray`

The type of a two-dimensional array of values used to define the values and derivatives.

`SlopeGenerator2d()`

Default constructor.

```
virtual void calc_slopes(const SlopeGenerator<V,F> &xsg,
                         const SlopeGenerator<V,F> &ysg,
                         const KnotSeq<F> &xknt,
                         const KnotSeq<F> &yknt,
                         const ValArray &v, ValArray &xds,
                         ValArray &yds, ValArray &xyds) const
```

Calculates the slopes using slope generators `xsg` and `ysg` to generate the slopes in each direction. The knot sequences are `xknt` and `yknt` and `v` contains the values at the points. On return, `xds` contains the x derivatives, `yds` the y derivatives and `xyds` the xy derivatives at each point.

8.2 One-parameter Hermite splines

The class `HermiteSpline<V,F>` represents a Hermite spline with a single parameter. It is a specialization of `PPSpline<V,F>` with the additional constraint that its template argument `V` must be a model of an Absolute Object as well as an Arithmetic Object. As explained in Section 2, this is required in order to implement the functions that define the spline by approximating an existing curve.

`HermiteSpline<V,F>` defines the following public members in addition to the default and copy constructors, the assignment operator and the member functions inherited from `PPSpline<V,F>`:

```
typedef typename SlopeGenerator<V,F>::ValArray ValArray
```

The type of the array of values used to generate the spline. It is the same as `SlopeGenerator<V,F>::ValArray` to facilitate the generation of slopes for the splines.

```
HermiteSpline(const KnotSeq<F> knots, const ValArray &values,
              const ValArray &slopes)
```

Construct the spline from a given knot sequence, set of values, and function slopes.

```
HermiteSpline(const KnotSeq<F> &knots, const ValArray &values)
```

Construct the spline from a given knot sequence and a set of values. A parabolic slope generator is used to make the slopes.

```
HermiteSpline(const KnotSeq<F> &knots, const ValArray &values,
              const SlopeGenerator<V,F> &sgen)
```

Construct the spline from a given knot sequence, set of values and slope generator.

`HermiteSpline(const KnotSeq<F> &knots,`
`const CurveLib::Curve<1U,V,F> &crv)`

Construct the spline from a given knot sequence and a curve. The values of the curve and its derivatives at the knots are used to define the spline. The resulting spline is an approximation of the original curve over the range of the knots.

`HermiteSpline(const CurveLib::Curve<1U,V,F> &crv, F x0, F x1,`
`F prec, unsigned max_knots = 1000)`

Construct the spline so that it approximates `crv` over the range `[x0,x1]` to precision `prec`. The knot sequence is refined until the difference between the spline and `crv`, sampled at three points in each knot interval, is less than `prec`. The knot sequence will not be allowed to increase to more than `max_knots` knots. If the required precision has not be attained with fewer than `max_knots` knots, a `TooManyKnots` exception is thrown.

`void define(const KnotSeq<F> &knots, const ValArray &values,`
`const ValArray &slopes)`

Define the spline from a given knot sequence, set of values and slopes.

`void define(const KnotSeq<F> &knots, const ValArray &values)`

Define the spline from a given knot sequence and set of values. A parabolic slope generator is used to make the slopes.

`void define(const KnotSeq<F> &knots, const ValArray &values,`
`const SlopeGenerator<V,F> &sgen)`

Define the spline from a given knot sequence, set of values and slope generator.

`void define(const KnotSeq<F> &knots,`
`const CurveLib::Curve<1U,V,F> &crv)`

Define the spline from a given knot sequence and a curve. Similar to the constructor with the same arguments.

`void define(const CurveLib::Curve<1U,V,F> &crv, F x0, F x1,`
`F prec, unsigned max_knots = 1000)`

Define the spline so that it approximates `crv` over the range `[x0,x1]` to precision `prec`. Similar to the constructor with the same arguments.

`void redefine(const ValArray &values, const ValArray &slopes)`

Define the spline from a given set of values and slopes. The current knot sequence is used. If the lengths of `values` and `slopes` are not the same as the number of knots, a `PPInconsistentData` exception is thrown.

`void redefine(const ValArray &values)`

Define the spline from a given set of values. The current set of knots is used. The slopes are generated with a parabolic slope generator. If the length of `values` is not the same as the number of knots, a `PPInconsistentData` exception is thrown.

```
void redefine(const ValArray &values,
              const SlopeGenerator<V,F> &sgen)
```

Define the spline from a given set of values and slope generator. The current set of knots is used. If the length of `values` is not the same as the number of knots, a `PPInconsistentData` exception is thrown.

`void set_slope_at_start(const V& slope)`

Sets the slope at the left end of the spline.

`void set_slope_at_end(const V& slope)`

Sets the slope at the right end of the spline.

8.3 Cubic splines

The class `CubicSpline<V,F>` represents a standard cubic spline through a set of data points: at each knot the spline will be continuous and will have continuous first and second derivatives. `CubicSpline<V,F>` is a specialization of `HermiteSpline<V,F>` which uses a `CubicSlopeGenerator<V,F>` to generate the slopes at the knots. It defines the following public members in addition to the default and copy constructors, the assignment operator and the member functions inherited from the base class `HermiteSpline<V,F>`:

`typedef typename CubicSlopeGenerator<V,F>::EndCondType EndCondType`

The type of end-point conditions.

```
CubicSpline(const KnotSeq<F> &knt, const ValArray &vals,
            bool p = false)
```

Constructs the spline with given knots and values. If `p` is true the spline is periodic; otherwise the not-a-knot condition is used at the end-points. The knot sequence must be strictly increasing.

```
CubicSpline(const KnotSeq<F> &knt, const ValArray &vals,
            EndCondType lo_cond, F lo_val,
            EndCondType hi_cond, F hi_val)
```

Constructs the spline with given knots and values. The condition at the start of the spline is `lo_cond` with value (if it is needed), `lo_val`. The condition at the

end of the spline is `hi_cond` with value `hi_val`. For example, to set the slope at the start to 1.0 and the second derivative at the end to 2.0, use `lo_cond = slope`, `lo_val = 1.0`, `hi_cond = second_derivative` and `hi_val = 2.0`. The value of the condition is ignored for the not-a-knot end condition. The knot sequence must be strictly increasing.

```
void define(const KnotSeq<F> &knt, const ValArray &vals,
            bool p = false)
```

Define the spline with given knots and values. Similar to the constructor with the same arguments.

```
void define(const KnotSeq<F> &knt, const ValArray &vals,
            EndCondType lo_cond, F lo_val,
            EndCondType hi_cond, F hi_val)
```

Defines the spline with given knots and values. Similar to the constructor with the same arguments.

```
void redefine(const ValArray &vals, bool p = false)
```

Define the spline with given values. If `p` is true the spline is periodic; otherwise the not-a-knot condition is used at the end-points. The current knot sequence is used. If the length of `values` is not the same as the number of knots, a `PPInconsistentData` exception is thrown.

```
void redefine(const ValArray &vals,
              EndCondType lo_cond, F lo_val,
              EndCondType hi_cond, F hi_val)
```

Defines the spline with given values and end-point conditions. The current knot sequence is used. If the length of `values` is not the same as the number of knots, a `PPInconsistentData` exception is thrown.

8.4 Akima splines

The class `AkimaSpline<F>` is a Hermite spline which uses Akima's algorithm[7] to calculate the slopes at the data points. Because of the nature of the algorithm, it has only been implemented for scalar-valued splines, not for vector valued splines. It is a specialization of `HermiteSpline<F,F>` which uses an `AkimaSlopeGenerator<F>` to calculate the slopes.

If the spline is not periodic, the slope at each end-point may be set or left free. In the latter case the slope will be set to the slope of the parabola through the nearest three points.

`AkimaSpline<F>` defines the following public members in addition to the default and

copy constructors, the assignment operator and the member functions inherited from `HermiteSpline<V,F>`:

```
AkimaSpline(const KnotSeq<F> &knt, const ValArray &vals,
            bool p = false)
```
Constructs the spline with given knots and values. If `p` is true the spline is periodic. The knot sequence must be strictly increasing.

```
AkimaSpline(const KnotSeq<F> &knt, const ValArray &vals,
            F lo_slope, F hi_slope)
```
Constructs the spline with given knots and values. The slopes at the first and last knots are given in `lo_slope` and `hi_slope`. Their values can be set to `AkimaSlopeGenerator<F>::value_not_set` to cause the slope to be ignored (for example, when you only want to set the slope at one end). The knot sequence must be strictly increasing.

```
void define(const KnotSeq<F> &knt, const ValArray &vals,
            bool p = false)
```
Define the spline with given knots and values. Similar to the constructor with the same arguments.

```
void define(const KnotSeq<F> &knt, const ValArray &vals,
            F lo_slope, F hi_slope)
```
Defines the spline with given knots and values. Similar to the constructor with the same arguments.

```
void redefine(const ValArray &vals, bool p = false)
```
Define the spline with given values. If `p` is true the spline is periodic. The current knot sequence is used. If the length of `vals` is not the same as the number of knots, a `PPInconsistentData` exception is thrown.

```
void redefine(const ValArray &vals, F lo_slope, F hi_slope)
```
Defines the spline to have the values in `vals`. The slopes at the first and last knots are given in `lo_slope` and `hi_slope`. Their values can be set to `AkimaSlopeGenerator<F>::value_not_set` to cause the slope to be ignored and end-slopes. The current knot sequence is used. If the length of `vals` is not the same as the number of knots, a `PPInconsistentData` exception is thrown.

8.5 Two-parameter Hermite splines

The class `HermiteSpline2d<V,F>` represents a two-parameter spline which is a Hermite spline in each direction. The type `V` representing the return value of the spline

must be a model of an Absolute Object (see Reference 1, Annex A.5) as well as a model of an Arithmetic Object (see Reference 1, Annex A.1). The type `F`, representing the type of a knot, must be a model of a Comparable Scalar Object: see Reference 1, Annex A.3.

`HermiteSpline2d<V,F>` is defined using two knot sequences, s_i and t_j. At each point (s_i, t_j) the value, x derivative, y derivative and xy derivative are given; they uniquely define the spline. `HermiteSpline2d<V,F>` has base class `PPSpline2d<V,F>` and defines the following public members in addition to the default and copy constructors, the assignment operator and the member functions inherited from `PPSpline2d<V,F>`:

```
typedef typename SlopeGenerator2d<V,F>::ValArray ValArray
```
The type of a two-dimensional array of values used to define the spline values and derivatives.

```
HermiteSpline2d(const KnotSeq<F> &xknt, const KnotSeq<F> &yknt,
                const ValArray &vals, const ValArray &xds,
                const ValArray &yds, const ValArray &xyds)
```
Constructs the spline using `xknt` and `yknt` for knot sequences, `vals` for the values at the points, `xds` for the derivatives with respect to x, `yds` for the derivatives with respect to y and `xyds` for the derivatives with respect to both x and y.

```
HermiteSpline2d(const KnotSeq<F> &xknt, const KnotSeq<F> &yknt,
                const ValArray &vals)
```
Constructs the spline using `xknt` and `yknt` for knot sequences and `vals` for the values at the points. A parabolic slope generator is used to generate the slopes in each direction.

```
HermiteSpline2d(const KnotSeq<F> &xknt, const KnotSeq<F> &yknt,
                const ValArray &vals,
                const SlopeGenerator<V,F> &xsg,
                const SlopeGenerator<V,F> &ysg)
```
Constructs the spline using `xknt` and `yknt` for knot sequences `vals` for the values at the points, and `xsg` and `ysg` to generate the slopes in each direction.

```
HermiteSpline2d(const KnotSeq<F> &xknt, const KnotSeq<F> &yknt,
                const CurveLib::Curve<2U,V,F> &crv)
```
Constructs the spline using `xknt` and `yknt` for knot sequences and with the values and slopes at the knots determined from `crv`.

```
HermiteSpline2d(const CurveLib::Curve<2U,V,F> &crv,
                F x0, F x1, F y0, F y1,
                F prec, unsigned max_knots = 100)
```

Construct the spline so that it approximates `crv` over the ranges x in `[x0,x1]` and y in `[y0,y1]` to precision `prec`. The knot sequences are refined until the difference between the spline and `crv`, sampled at five points in each knot interval, are less than `prec`.

If the curve derivatives cannot be evaluated, finite differences will be used instead.

The knot sequences will not be allowed to increase to more than `max_knots` knots each. If the required precision has not be attained with fewer than `max_knots` knots, a `TooManyKnots` exception is thrown.

```
void define(const KnotSeq<F> &xknt, const KnotSeq<F> &yknt,
            const ValArray &vals, const ValArray &xds,
            const ValArray &yds, const ValArray &xyds)
```

Defines the spline using `xknt` and `yknt` for knot sequences `vals` for the values at the points, `xds` for the x derivatives, `yds` for the y derivatives and `xyds` for the xy derivatives.

```
void define(const KnotSeq<F> &xknt, const KnotSeq<F> &yknt,
            const ValArray &vals)
```

Defines the spline using `xknt` and `yknt` for knot sequences and `vals` for the values at the points. A parabolic slope generator is used to generate the slopes in each direction.

```
void define(const KnotSeq<F> &xknt, const KnotSeq<F> &yknt,
            const ValArray &vals,
            const SlopeGenerator<V,F> &xsg,
            const SlopeGenerator<V,F> &ysg)
```

Defines the spline using `xknt` and `yknt` for knot sequences `vals` for the values at the points, and `xsg` and `ysg` to generate the slopes in each direction.

```
void define(const KnotSeq<F> &xknt, const KnotSeq<F> &yknt,
            const CurveLib::Curve<2U,V,F> &crv)
```

Defines the spline using `xknt` and `yknt` for knot sequences and with the values and slopes at the knots determined from `crv`.

```
void define(const CurveLib::Curve<2U,V,F> &crv, F prec,
            F x0, F x1, F y0, F y1, unsigned max_knots = 100)
```

Defines the spline so that it approximates `crv` over the ranges x in `[x0,x1]` and y in `[y0,y1]` to precision `prec`. Similar to the constructor with the same arguments.

```
void redefine(const ValArray &vals, const ValArray &xds,
              const ValArray &yds, const ValArray &xyds)
```

Defines the spline using `vals` for the values at the points, `xds` for the x derivatives, `yds` for the y derivatives and `xyds` for the xy derivatives. The current knot sequences are used. If the dimensions of `vals`, `xds`, `yds` and `xyds` are not the same as the number of knots, a `PPInconsistentData` exception is thrown.

```
void redefine(const ValArray &vals)
```

Defines the spline using `vals` for the values at the points, and `xsg` and `ysg` to generate the slopes in each direction. The current knots are used. If the dimensions of `vals` are not the same as the number of knots, a `PPInconsistentData` exception is thrown.

```
void redefine(const ValArray &vals,
              const SlopeGenerator<V,F> &xsg,
              const SlopeGenerator<V,F> &ysg)
```

Defines the spline using `vals` for the values at the points, and `xsg` and `ysg` to generate the slopes in each direction. The current knots are used. If the dimensions of `vals` are not the same as the number of knots, a `PPInconsistentData` exception is thrown.

```
void set_x_slopes_at_start(const V &v)
```

Sets the slope at the start of each x spline to `v`. Used primarily to set the slopes to zero.

```
void set_x_slopes_at_end(const V &v)
```

Sets the slope at the end of each x spline to `v`. Used primarily to set the slopes to zero.

```
void set_y_slopes_at_start(const V &v)
```

Sets the slope at the start of each y spline to `v`. Used primarily to set the slopes to zero.

```
void set_y_slopes_at_end(const V &v)
```

Sets the slope at the end of each y spline to `v`. Used primarily to set the slopes to zero.

```
static void adjust_periodic_values(ValArray &v, F tol,
                                   bool xdir)
```

The periodic slope generators require that the values at the ends of the spline rows are exactly equal. This function adjusts values which differ by no more than `tol` so that they are exactly equal. If two values differ by more than `tol`, an `Error` exception is thrown.

The argument `xdir` is true if the values are to be adjusted for periodicity in the x direction. Otherwise they are adjusted for periodicity in the y direction.

8.6 Hermite interpolation of several curves

The class `HermiteExtendedCurve<N,V,F>` is a curve created by using a Hermite spline to interpolate between a collection of curves of one fewer parameters. The parameter list of the new curve is obtained from the parameter list of the collection curves by adding a new parameter to the end. Thus the parameter of the spline is the last parameter in the list.

`HermiteExtendedCurve<N,V,F>` is a specialization of `Curve<N,V,F>` in namespace `CurveLib`, where the template parameter `N` is the number of parameters of the curve, `V` is the type of the value of the spline (a model of an Arithmetic Object) and `F` is the type of each knot (a model of a Comparable Scalar Object). Each of the curves that are splined are of type `Curve<N-1,V,F>`. If template arithmetic is not allowed by the compiler, then an additional template argument is necessary; the class is then `HermiteExtendedCurve<N,NM1,V,F>` where the template argument `N-1` must be equivalent to `N-1`.

In order to calculate the derivatives of the new curve, it is necessary that the Hermite spline slopes be linear with respect to the values at the knots. This is true for the parabolic slope generator, but not true in general. Therefore, only parabolic slope generators are used.

`HermiteExtendedCurve<N,V,F>` defines the following public members in addition to the default and copy constructors, the assignment operator and the member functions inherited from `Curve<N,V,F>`:

```
typedef std::vector<CurveLib::Curve<N-1,V,F> > CurveListType
```
The type of the collection of curves to be splined.

```
HermiteExtendedCurve(const KnotSeq<F> knots,
                     const CurveListType &curves)
```
Constructs the spline using the knot sequence, `knots`, and the list of curves, `curves`.

```
void define(const KnotSeq<F> &knots,
            const CurveListType &curves)
```
Define the spline from a given knot sequence and a list of curves.

```
const KnotSeq<F>& knots() const
```
Returns the knot sequence.

9 B-splines

In the B-spline representation, the spline is represented by:

$$f(x) = \sum_{i=0}^{N-k-1} \beta_i B_{i,k}(x) \tag{28}$$

The basis functions $B_{i,k}(x)$ are piecewise polynomials of order k with breakpoints at the knots. The basis function $B_{i,k}$ is non-zero only on the range $[t_i, t_{i+k}]$; therefore, in the sum of Equation (28), there are only k non-zero terms for any x. The B-spline basis functions also satisfy the normalization property:

$$\sum_{i=0}^{N-k-1} B_{i,k}(x) = 1 \qquad \text{for any } x \text{ in } [t_{k-1}, t_{N-k}] \tag{29}$$

When calculating B-splines the knot sequence may contain multiple knots: i.e. one or more knots having the same value. The continuity of $f(x)$ is related to the multiplicity of the knots: if m is the multiplicity of a knot, then $f(x)$ has at least $k - m - 1$ continuous derivatives at that knot. Thus if k is four (the polynomial segments are cubic), then $f(x)$ has continuous second derivatives at each single knot, continuous first derivative at each double knot, is simply continuous at each triple knot, and may be discontinuous at each quadruple knot. It is normal to assign k knots at each end of the range of the spline; outside the range the spline will evaluate to zero; the k knots at each end allow the discontinuity from the value of the spline at the end point and the value of zero beyond it.

The basis functions can be calculated using the following recursion relation:

$$B_{i,1}(x) = \begin{cases} 1 \text{ if } t_i \le x < t_{i+1} \\ 0 \text{ otherwise} \end{cases} \tag{30}$$

$$B_{i,k}(x) = \frac{x - t_i}{t_{i+k-1} - t_i} B_{i,k-1}(x) + \frac{t_{i+k} - x}{t_{i+k} - t_{i+1}} B_{i+1,k-1}(x) \tag{31}$$

For more information on B-splines and how to evaluate them efficiently, see de Boor[5].

Two-parameter B-spline curves can be defined using two knot-sequences, s_i and t_j, and a two-dimensional array of coefficients:

$$f(x,y) = \sum_{i=0}^{N_x-k_x-1} \sum_{j=0}^{N_y-k_y-1} \beta_{ij} B_{i,k_x}(x) B_{j,k_y}(y) \tag{32}$$

9.1 One-parameter B-spline curves

The class `BSpline<V,F>` is a spline represented as a sum of B-spline basis functions. It is a specialization of `BaseSpline<V,F>`. The type `V` representing the return value of the spline must be a model of an Absolute Object as well as a model of an Arithmetic Object for reasons explained in Section 2. `BSpline<V,F>` has the following public members in addition to the default and copy constructors, the assignment operator and the member functions inherited from `BaseSpline<V,F>`:

`typedef std::vector<V> CoefArray`
: The type of the array of coefficients.

`BSpline(unsigned k, const KnotSeq<F> &kts, const CoefArray &c)`
: Makes a B-spline curve with order `k`, knot sequence `kts`, and coefficient array `c`. The arguments are copied.

```
BSpline(unsigned k, const KnotSeq<F> &kts,
        const CurveLib::Curve<1U,V,F> &c)
```
: Makes a B-spline curve with order `k` and knot sequence `kts` which is an approximation of the curve `c`. The approximation is exact if `c` belongs to the vector space of functions spanned by the B-splines on `kts`. This constructor can be used to convert splines with different representations to B-splines.

```
BSpline(unsigned k, const CurveLib::Curve<1U,V,F> &crv, F x0, F x1,
        F prec, unsigned max_knots = 1000)
```
: Makes a B-spline curve of order `k` so that it approximates `crv` over the range `[x0,x1]` to precision `prec`. The knot sequence is refined until the difference between the spline and `crv`, sampled at three points in each knot interval, is less than `prec`. The knot sequence will not be allowed to increase to more than `max_knots` knots. If the required precision has not be attained with fewer than `max_knots` knots, a `TooManyKnots` exception is thrown.

`void define(unsigned k, const KnotSeq<F> &kts, const CoefArray &c)`
: Defines the B-spline curve by setting its order, knots and coefficients. Similar to the constructor with the same arguments.

```
void define(unsigned k, const KnotSeq<F> &kts,
            const CurveLib::Curve<1U,V,F> &c)
```
: Defines the B-spline curve by giving it order `k` and knot sequence `kts` and then approximating the curve `c`. Similar to the constructor with the same arguments.

```
void define(unsigned k, const CurveLib::Curve<1U,V,F> &crv,
            F x0, F x1, F prec, unsigned max_knots = 1000)
```

Redefines the spline by giving it order `k` and then approximating `crv` over the range `[x0,x1]` to precision `prec`. Similar to the constructor with the same arguments.

`unsigned order() const`
Returns the order of the spline.

`unsigned number_of_Bsplines() const`
Returns the number of B-splines.

`const CoefArray& Bspline_coefs() const`
Returns the spline coefficients.

`void adjust_parameter(F plo, F phi)`
Adjusts the knot sequence so that it spans the range `[plo,phi]` instead of the range of the knot sequence.

9.2 Two-parameter B-spline curves

The class `BSpline2d<V,F>` represents a two-dimensional B-spline curve. It is a specialization of `BaseSpline2d<V,F>`. It defines the following public members in addition to the default and copy constructors, the assignment operator and the member functions inherited from `BaseSpline2d<V,F>`:

`typedef std::vector<V> CoefArray`
The type of the array of coefficients.

```
BSpline2d(unsigned kx, unsigned ky,
          const KnotSeq<F> &ktx, const KnotSeq<F> &kty,
          const CoefArray &c)
```
Makes a spline with order `kx` in the x direction, `ky` in the y direction, knot sequences `ktx` and `kty`, and coefficients `c`. The coefficients are stored as a linear sequence with values with increasing y appearing first, then the values with increasing x.

`unsigned x_order() const`
Returns the order of the splines in the x direction.

`unsigned y_order() const`
Returns the order of the splines in the y direction.

`unsigned number_of_x_Bsplines() const`
Returns the number of B-splines in the x direction

`unsigned number_of_y_Bsplines() const`
Returns the number of B-splines in the y direction

`const std::vector<V>& Bspline_coefs() const`
Returns the spline coefficients. They are stored as a packed array with y values increasing first.

```
void define(unsigned kx, unsigned ky, const KnotSeq<F> &ktx,
            const KnotSeq<F> &kty, const CoefArray &c)
```
Defines the spline by setting its order, knot sequences and coefficients. Similar to the constructor with the same arguments.

```
void adjust_parameters(const ParamType &plo,
                       const ParamType &phi)
```
Adjusts the parameters so that they span the range `[plo,phi]` rather than the range of the knot sequence. The adjustment is made by transforming the knot sequences.

`void adjust_parameter(unsigned i, F plo, F phi)`
Adjusts parameter `i` so that it spans the range `[plo,phi]`. The adjustment is made by transforming the knot sequences.

10 Non-uniform rational B-splines (NURBs)

Non-uniform rational B-splines, usually simply known as NURBs, are closely related to B-splines. A one-parameter NURB curve is defined as follows:

$$f(x) = \frac{\sum_{i=0}^{N-k-1} \beta_i w_i B_{i,k}(x)}{\sum_{i=0}^{N-k-1} w_i B_{i,k}(x)} \tag{33}$$

where the weights, w_i, must be non-negative. Because the B-splines satisfy the normalization property (see Equation (29)), a B-spline curve can be represented as a NURB with constant weights.

Two-parameter NURBs can be defined using two knot-sequences, s_i and t_j, and two-dimensional arrays of weights and coefficients:

$$f(x,y) = \frac{\sum_{i=0}^{N_x-k_x-1} \sum_{j=0}^{N_y-k_y-1} \beta_{ij} w_{ij} B_{i,k_x}(x) B_{j,k_y}(y)}{\sum_{i=0}^{N_x-k_x-1} \sum_{j=0}^{N_y-k_y-1} w_{ij} B_{i,k_x}(x) B_{j,k_y}(y)} \tag{34}$$

10.1 One-parameter NURBs

NURBs are represented by the C++ class `NURBSpline<V,F>`, a specialization of `BaseSpline<V,F>`. The type `V` representing the return value of the spline must be a model of an Absolute Object as well as an Arithmetic Object and the type `F`, representing the type of a knot, must be a model of a Comparable Scalar Object.

`NURBSpline<V,F>` has the following public members in addition to the default and copy constructors, the assignment operator and the member functions inherited from `BaseSpline<V,F>`:

`typedef std::vector<V> CoefArray`
: The type of the array of spline coefficients.

`typedef std::vector<F> WeightArray`
: The type of the array of weights.

```
NURBSpline(unsigned k, const KnotSeq<F> &kts,
           const WeightArray &w, const CoefArray &c)
```
: Makes a NURB spline with order `k`, knot sequence `kts`, weights `w` and coefficients `c`.

`unsigned order() const`
: Returns the order of the NURB spline.

`unsigned number_of_Bsplines() const`
: Returns the number of B-splines.

`CoefArray Bspline_coefs() const`
: Returns the spline coefficients.

`const WeightArray& weights() const`
: Returns the spline weights.

`void define(unsigned order, const KnotSeq<F> &kts,`
`const WeightArray &w, const CoefArray &c)`

Defines the NURB spline by setting its order, knots, weights and coefficients. The arguments are copied.

`void adjust_parameter(F plo, F phi)`

Adjusts the knot sequence so that it spans the range `[plo,phi]` instead of the range of the knot sequence.

10.2 Two-parameter NURBs

The class `NURBSpline2d<V,F>` represents a two-dimensional tensor product NURB curve. It is a specialization of `BaseSpline2d<2U,V,F>`. It defines the following public members in addition to the default and copy constructors, the assignment operator and the member functions inherited from `BaseSpline2d<V,F>`:

`typedef std::vector<V> CoefArray`

The type of the array of spline coefficients.

`typedef std::vector<F> WeightArray`

The type of the array of weights.

`NURBSpline2d(unsigned kx, unsigned ky, const KnotSeq<F> &ktx,`
`const KnotSeq<F> &kty, const WeightArray &w,`
`const CoefArray &c)`

Makes a spline with order `kx` in the x direction, `ky` in the y direction, knot sequences `ktx` and `kty`, weights `w` and coefficients `c`.

`unsigned x_order() const`

Returns the order of the splines in the x direction.

`unsigned y_order() const`

Returns the order of the splines in the y direction.

`unsigned number_of_x_Bsplines() const`

Returns the number of B-splines in the x direction

`unsigned number_of_y_Bsplines() const`

Returns the number of B-splines in the y direction.

`CoefArray Bspline_coefs() const`

Returns the spline coefficients. They are stored as a packed array with y values increasing first.

`const WeightArray& weights() const`

Returns the spline weights. They are stored as a packed array with y values increasing first.

```
void define(unsigned kx, unsigned ky, const KnotSeq<F> &ktx,
            const KnotSeq<F> &kty, const WeightArray &w,
            const CoefArray &c)
```

Defines the spline by setting its order, knot sequences, weights and coefficients.

```
void adjust_parameters(const ParamType &plo,
                       const ParamType &phi)
```

Adjusts the parameters so that they span the range `[plo,phi]` rather than the range of the knot sequence. The adjustment is made by transforming the knot sequences.

`void adjust_parameter(unsigned i, F plo, F phi)`

Adjusts parameter `i` so that it spans the range `[plo,phi]`. The adjustment is made by transforming the knot sequences.

11 Approximating generic curves with splines

One of the important properties of the classes `BSpline<V,F>`, `HermiteSpline<V,F>` and `HermiteSpline2d<V,F>` is the ability to approximate any curve represented by a `Curve<1U,V,F>` or `Curve<2U,V,F>`. This is especially important if the curves are to be used in a different application. There are many applications that can use splines but which cannot deal with the full variety of curves that can be defined using the CurveLib library. In these cases a curve can often be converted to a spline, exported to an IGES file (Reference 2 describes how to do this) which is imported into the application.

11.1 Approximation using Hermite splines

The Hermite splines approximate a curve by sampling its values and first derivatives at a series of points, then constructing the spline with the points as the knot sequence. The accuracy is increased by increasing the number of sampling points. For example, in the following code a `HermiteSpline<V,F>` is used to approximate e^{-x^2} over the range $[-5, 5]$.

```
using namespace Spline;
using namespace CurveLib;
```

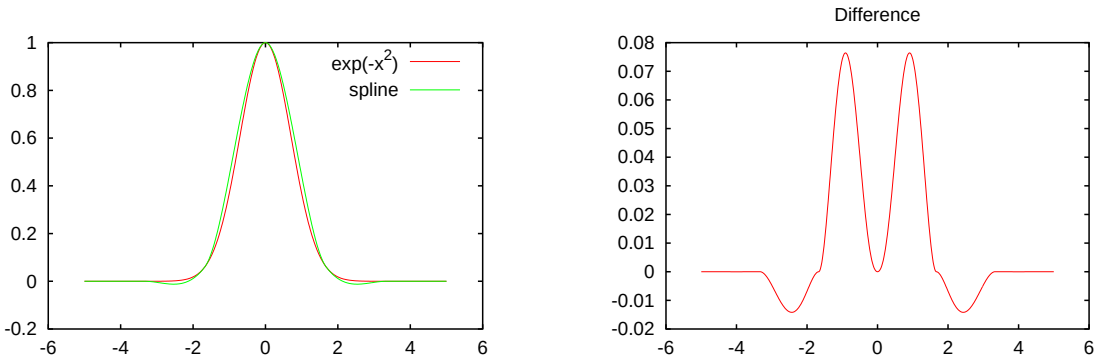

Figure 3: *The approximation of* e^{-x^2} *over* $[-5, 5]$ *using a Hermite spline with seven equally spaced knots.*

```
unsigned nkt = 7;
double xlo = -5.0, xhi = 5.0;
KnotSeq<double> knots(nkt);
for (unsigned i = 0; i < nkt; ++i) {
  knots[i] = xlo + i*(xhi-xlo)/double(nkt-1);
}
Curve<1U,double> crv = Exp<>()(-PowInt<>(2));
HermiteSpline<double> spline(knots,crv);
```

Figure 3 plots e^{-x^2} and the curve generated by `spline`. On the right of the figure the difference between the two curves is plotted. With only seven knots the accuracy of the approximation is approximately 7%.

An approximation to a specified accuracy can also be made. In this case the Hermite spline first allocates a sparse knot sequence then generates the spline by sampling the values and slopes at those knots. The spline is then sampled between the knots. Where it does not meet the specified accuracy, additional knots are inserted and a new spline is generated. This procedure is continued until the accuracy is achieved everywhere or the maximum number of knots is exceeded. In the latter case a `TooManyKnots` exception is thrown. The following code takes the spline generated above and increases the accuracy to 10^{-3}.

```
try {
  spline.define(crv,0.001);
}
catch(TooManyKnots &tmk) {
  // Handle the exception
}
```

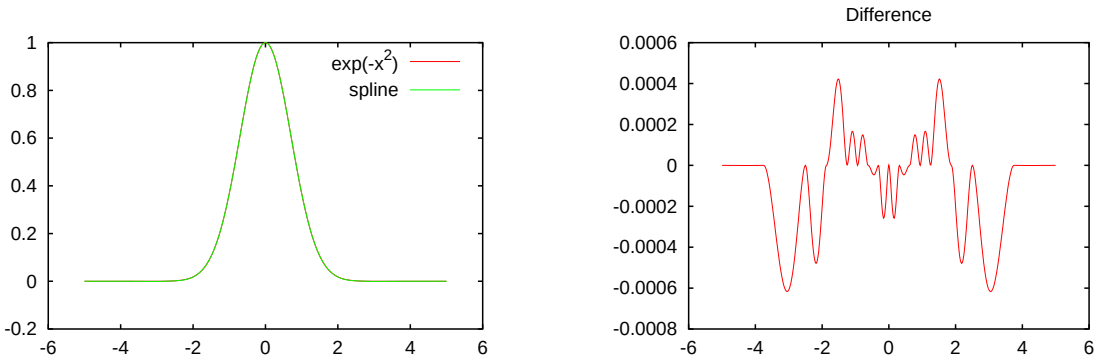

Figure 4: *The approximation of* e^{-x^2} *over* $[-5, 5]$ *using a Hermite spline to an accuracy of* 10^{-3}.

The new spline approximation is shown in Figure 4. It is sufficiently accurate that it is difficult to distinguish the spline curve from the original curve. The locations of the knots used can be determined from the places where the difference between the two curves is zero. From the right side of Figure 4 we can see that there are 17 knots and that they are not evenly spaced; there are more knots near zero, where the curve varies most rapidly.

11.2 Approximation using B-spline curves

The class `BSpline<V,F>` has similar member functions for approximating curves. They use the same technique of sampling the curve between the knots and adding knots where required. For a given knot sequence the following expression, based on de Boor's Lemma IX.1[5], is used to calculate each B-spline coefficient:

$$\beta_i = \sum_{r=0}^{k-1} (-1)^{k-r-1} \frac{d^{k-r-1}\psi}{dx^{k-r-1}}(\tau) \frac{d^r c}{dx^r}(\tau) \tag{35}$$

$$\psi(x) = \frac{(t_i - x) \cdots (t_{i+k} - x)}{(k-1)!} \tag{36}$$

$$\tau = \frac{t_{i+1} + \cdots + t_{i+k-1}}{k-1} \tag{37}$$

where $c(x)$ is the curve to be approximated. The algorithm used to evaluate it is discussed in Reference 9. The approximation is exact if the curve to be approximated belongs to the vector space of functions spanned by the B-spline basis functions: i.e. if $c(x)$ is a piecewise polynomial of order k or less with breakpoints at the t_i and whose continuity is reflected in the multiplicity of the knots.

Equation (35) requires that all derivatives of $c(x)$ up to the $(k-1)^{\text{th}}$ be evaluated when calculating each B-spline coefficient: e.g. up to third derivatives for a fourth order

(cubic) spline. In comparison, when approximating a curve using a Hermite spline, only the value of the curve and its first derivative need be evaluated. When $c(x)$ is complex, the evaluation of the higher order derivatives is often very costly. Therefore, in general, approximation using B-splines is much slower than approximation using Hermite splines.

When a `BSpline<V,F>` is used to approximate a curve, the order of the spline, k, must be specified. In addition, if a knot sequence is specified, it must include k knots at or before the lowest value at which the curve is to be evaluated, and k knots at or after the highest value of the curve to be evaluated. For example, the following code generates a B-spline approximation of order four of e^{-x^2}:

```
using namespace Spline;
using namespace CurveLib;
unsigned k = 4, n = 7;
double xlo = -5.0, xhi = 5.0;

// Generate the knot sequence as before
KnotSeq<double> knots(nkt);
for (unsigned i = 0; i < n; ++i) {
  knots[i] = xlo i*(xhi-xlo)/double(nkt-1);
}

// Add k-1 additional knots at the ends
for (unsigned i = 0; i < k-1; ++i) {
  knots.insert(xlo);
  knots.insert(xhi);
}

// Approximate the curve
Curve<1U,double> crv = Exp<>()(-PowInt<>(2));
HermiteSpline<double> spline(k,knots,crv);
```

In this example there are 6 polynomial intervals over the range $[-5, 5]$, exactly as in the first example above; however, for the B-spline this requires 13 knots since four must be placed at -5 and four at 5.

Figure 5 compares the B-spline curve with e^{-x^2}. The B-spline approximation is less accurate that the Hermite spline approximation although they used similar knot sequences and both use cubic polynomials. Loosely speaking, this is because the requirement that the B-spline have continuity of second derivative at the knots makes it somewhat less flexible. We could relax this requirement by doubling each of the interior knots so that the B-spline curve has the same order, the same number of polynomial segments, *and the same continuity of derivatives* as the Hermite spline. The result is a curve very similar to the Hermite spline approximation with a maximum discrepancy of about 7%.

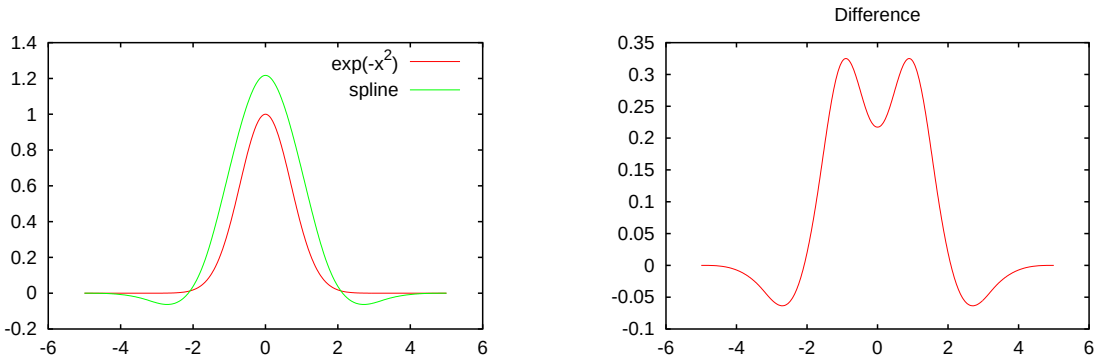

Figure 5: *The approximation of e^{-x^2} over $[-5, 5]$ using a B-spline curve of order four with 13 knots (six polynomial segments).*

Notice that, unlike the Hermite spline approximation, the B-spline approximation is not exact when evaluated at the knots.

We could also generate an approximation of accuracy 10^{-3} using:

```
try {
  bspline.define(k,crv,0.001);
}
catch(TooManyKnots &tmk) {
  // Handle the exception
}
```

Table 1 lists the number of polynomial segments required to achieve accuracies between 10^{-3} and 10^{-8} using a Hermite spline and B-splines of different orders (the B-spline curves each have continuity of derivative $k-2$ at the knots: i.e. the interior knots are not duplicated). For the low accuracies the number of polynomial segments needed is not strongly dependent on the order. However, as greater accuracy is required, fewer polynomial segments are required for the splines of higher order. However, this advantage must be weighed against the disadvantage that the higher order B-splines are more costly to compute.

The two-parameter spline class `HermiteSpline2d<V,F>` has similar member functions for approximating any `Curve<2U,V,F>`. For `BSpline2d<V,F>`, this functionality has only been included for specified knot sequences, not for approximation to arbitrary accuracy. This is because the latter is usually too inefficient too calculate.

Table 1: *The number of knot intervals required to approximate e^{-x^2} with different accuracies. The B-spline curves have orders from 3 to 6.*

Accuracy		10^{-1}	10^{-2}	10^{-3}	10^{-4}	10^{-5}	10^{-6}	10^{-7}	10^{-8}
Number of polynomial segments	Hermite	6	8	16	28	50	80	144	250
	B-spline 3	10	16	28	52	102	204	412	838
	B-spline 4	8	16	28	44	74	128	238	440
	B-spline 5	12	12	22	34	46	76	102	154
	B-spline 6	10	14	22	34	46	72	94	152
	B-spline 7	12	14	22	26	42	48	72	94
	B-spline 8	12	14	24	26	42	48	76	94

12 Converting between spline representations

B-spline curves and PP-splines are different representations of the same thing: piecewise polynomials. Therefore it is possible to convert a curve from one representation to the other. This section describes functions for doing this.

12.1 Converting from PP-splines to B-spline curves

A PP-spline provides no guarantees of continuity at the knots, whereas the continuity of a B-spline curve reflects the multiplicity of its knots. Therefore, to convert from a completely arbitrary PP-spline to a B-spline curve requires a B-spline knot sequence each of whose knots has multiplicity k. It is hard to imagine why one would want to do this, but a conversion function has been provided just in case.

An arbitrary Hermite spline is of fourth order and has continuity of first derivatives. It can be represented exactly by a B-spline curve in which all the knots are duplicated and two additional knots are added at each end-point.

A standard cubic spline `CubicSpline<V,F>` has continuous second derivatives, so it can be converted to a B-spline curve whose internal knots are single.

The following functions are provide for converting from a PP-spline to a B-spline. They each construct an appropriate knot sequence for the B-spline curve, then use the `BSpline<V,F>` constructor which approximates `spline` using that knot sequence. In these cases the approximation is exact.

```
template<class V, class F>
BSpline<V,F> convert_to_BSpline(PPSpline<V,F> spline)
```

Converts an arbitrary PP-spline to a B-spline curve. Since the continuity of the PP-spline is not known, this requires a B-spline knot sequence each of whose knots has multiplicity k, where k is the order.

```
template<class V, class F>
BSpline<V,F> convert_to_BSpline(HermiteSpline<V,F> spline)
```

Converts an arbitrary Hermite spline to a B-spline curve. Since the Hermite spline has continuity of only first derivatives, this requires a B-spline knot sequence each of whose knots has multiplicity two.

```
template<class V, class F>
BSpline<V,F> convert_to_BSpline(CubicSpline<V,F> spline)
```

Converts a standard cubic spline to a B-spline curve.

12.2 Converting from B-spline curves to PP-splines

Since PP-splines are inherently capable of representing any of the continuity requirements of a B-spline, only one function is needed to convert efficiently from an arbitrary B-spline curve to a PP-spline. It first constructs a knot sequence for the PP-spline by copying the B-spline knots but reducing the multiplicity of each knot to one. A coefficient array is then calculated using the derivatives of the B-spline at each knot. The PP-spline is constructed using the knots and coefficients. Two-parameters B-spline curves are converted similarly.

```
template<class V, class F>
PPSpline<V,F> convert_to_PPSpline(BSpline<V,F> spline)
```

Converts an arbitrary B-spline curve to a PP-spline.

```
template<class V, class F>
PPSpline2d<V,F> convert_to_PPSpline2d(BSpline2d<V,F> spline)
```

Converts an arbitrary two-parameter B-spline curve to a two-parameter PP-spline.

13 Concluding remarks

This document has described a library of C++ classes which represent splines of different types. The spline classes are based on the more general CurveLib library for representing multi-parameter differentiable functions. From the CurveLib classes they inherit arithmetic and composition operators that allow the splines to be combined with other functions to generate complex curves and surfaces.

Although originally developed for representing complex geometric shapes, the spline classes have much wider applicability.

References

[1] Hally, D. (2006), C++ classes for representing curves and surfaces: Part I: Multi-parameter differentiable functions, (DRDC Atlantic TM 2006-254) Defence R&D Canada – Atlantic.

[2] Hally, D. (2006), C++ classes for representing curves and surfaces: Part III: Reading and writing in IGES format, (DRDC Atlantic TM 2006-256) Defence R&D Canada – Atlantic.

[3] (1988), Initial Graphics Exchange Specification (IGES) Version 4.0, US Dept. of Commerce, National Bureau of Standards. Document No. NBSIR 88-3813.

[4] Hally, D. (2006), C++ classes for representing curves and surfaces: Part IV: Distribution functions, (DRDC Atlantic TM 2006-257) Defence R&D Canada – Atlantic.

[5] de Boor, C. (1978), A Practical Guide to Splines, New York: Springer Verlag.

[6] Standard Template Library Programmer's Guide (online), Silicon Graphics, Inc., http://www.sgi.com/tech/stl (Access Date: November 2006).

[7] Akima, H. (1970), A new Method of Interpolation and Smooth Curve Fitting Based on Local Procedures, *Journal of the Association for Computing Machinery*, 17, 589.

[8] Fritsch, F. N. and Carlson, R. E. (1980), Monotone Piecewise Cubic Interpolation, *SIAM Journal of Numerical Analysis*, 17, 238.

[9] Hally, D. (1991), Representation of Knuckles in Tensor Product B-spline Hull Representations, (DREA TM 91-216) Defence R&D Canada – Atlantic.

Index

Absolute Object, 4, 25, 30, 35, 38
Akima spline, i, iii, 20, 28–29, 54
`AkimaSlopeGenerator<F>`, **20**, 28
`AkimaSpline<F>`, **28–29**
approximating curves, 40–44
Arithmetic Object, 3, 8, 11, 13, 25, 30, 33, 35, 38

B-spline basis functions, 1, 2, 34–35, 42
B-spline curves, i, iii, 2, 3, 7, 34–37, 45, 54
  approximation using, 42–44
  conversion to PP-splines, 46
  creation from Hermite splines, 45
  creation from PP-splines, 45–46
  creation from standard cubic spline, 45
  two parameter, 34
`BaseSpline2d<2U,V,F>`, 39
`BaseSpline2d<V,F>`, 3, **9–11**, 15, 16, 36, 39
`BaseSpline<V,F>`, 3, **8–9**, 11, 14, 35, 38
`BSpline2d<V,F>`, 2, 4, **36–37**, 44
`BSpline<V,F>`, 2, 4, **35–36**, 40, 42, 43, 45

`CoefArray<V>`, 13
Comparable Scalar Object, 3, 8, 11, 13, 33, 38
cubic spline, 21–23, 27–28
`CubicSlopeGenerator<V,F>`, **22–23**, 27
`CubicSpline<V,F>`, **27–28**, 45
`CurveLib` classes
  `ArcSin<F>`, 14
  `Curve<1U,V,F>`, 3, 8, 11, 26, 35, 36, 40
  `Curve<2U,V,F>`, 3, 9, 30, 31, 40, 44
  `Curve<N,V,F>`, 3–5, 33
  `Derivs<1U>`, 4
  `Derivs<2U>`, 4
  `Sqrt<F>`, 14
CurveLib library, i, iii, 1, 40, 54

`DerivType`, 4, 5

exceptions, 4
  `Error`, 4, 8, 32
  `OutOfRange`, 8
  `PPInconsistentData`, 14, 16–18, 26–29, 32
  `TooManyKnots`, 26, 31, 35, 41

general splines, **11–12**
  two-parameter, 12
`GeneralSpline2d<V,F>`, **12**
`GeneralSpline<V,F>`, 1, **11**

header files
  `HermiteSpline.h`, 18
  `Spline.h`, 3
Hermite spline
  conversion to B-spline curve, 45
Hermite splines, i, iii, 18–33, 54
`HermiteExtendedCurve<N,V,F>`, **33**
`HermiteSpline2d<V,F>`, 2, 4, **29–33**, 40, 44
`HermiteSpline<F,F>`, 28
`HermiteSpline<V,F>`, 2, 4, **25–27**, 27, 29, 40

IGES file, 40

`KnotSeq<F>`, **6–8**, 11, 14, 16–19, 24–31, 33, 35–40
`KnotSeq<F>::SizeType`, 6

`LinearSpline2d<V,F>`, 2, **17–18**
`LinearSpline<V,F>`, 2, **17**

`MonotonicSlopeGenerator<V,F>`, **23–24**

namespaces
    CurveLib, 3–5, 8, 9, 11, 14, 26, 30, 31, 33, 35, 36, 40, 44
    Spline, 3, 18
    std, 3
    VecMtx, 4
non-uniform rational B-splines, i, iii, 37–40, 54
NURBs, i, iii, 37–40, 54
    two-parameter, 37–38
NURBSpline2d<V,F>, **39–40**
NURBSpline<V,F>, **38–39**

ParamType, 4, 5, 37, 40
ParSlopeGenerator<V,F>, **19–20**
PP-representation, 1, 12
PP-splines, i, iii, 6, 7, 12–17, 45, 54
    conversion to B-spline curves, 45–46
    creation from B-spline curves, 46
    two parameter, 15–17
PPCoefArray2d<V>, 15
PPCoefArray<V>, **13–14**
PPSpline2d<V,F>, 2, 5, **15–17**, 17, 30
PPSpline<V,F>, 2, **12–14**, 17, 25

slope generators, 19–25
SlopeGenerator2d<V,F>, **24–25**, 30
SlopeGenerator<F,F>, 20
SlopeGenerator<V,F>, **19**, 20, 24, 25, 27, 30, 32
SlopeGenerator<V,F>::ValArray, 25
standard cubic spline, i, iii, 54
    conversion to B-spline curve, 45
Standard Template Library, 2
std classes
    ostream, 3

VecMtx classes
    VecN<1U,F>, 4
    VecN<2U,F>, 4
Vector Object, 4, 23

Distribution list

DRDC Atlantic TM-2006-255

Internal distribution

1 Author

5 Library

Total internal copies: 6

External distribution

Department of National Defence

1 DRDKIM

2 DMSS 2

Others

2 Canadian Acquisitions Division
National Library of Canada
395 Wellington Street
Ottawa, Ontario
K1A ON4
Attn: Government Documents

1 Director-General
Institute for Marine Dynamics
National Research Council of Canada
P.O. Box 12093, Station A
St. John's, Newfoundland
A1B 3T5

1 Director-General
Institute for Aerospace Research
National Research Council of Canada
Building M-13A
Ottawa, Ontario
K1A OR6

1 Transport Development Centre
Transport Canada
6th Floor
800 Rene Levesque Blvd, West
Montreal, Que.
H3B 1X9
Attn: Marine R&D Coordinator

1 Canadian Coast Guard
Ship Safety Branch
Canada Building, 11th Floor
344 Slater Street
Ottawa, Ontario
K1A 0N7
Att: Chief, Design and Construction

MOUs

6 Canadian Project Officer ABCA-02-01 (C/SCI, DRDC Atlantic – 3 paper copies, 3 PDF files on CDROM)

Total external copies: 15

Total copies: 21

DOCUMENT CONTROL DATA

(Security classification of title, body of abstract and indexing annotation must be entered when document is classified)

1. ORIGINATOR (The name and address of the organization preparing the document. Organizations for whom the document was prepared, e.g. Centre sponsoring a contractor's report, or tasking agency, are entered in section 8.)

   Defence R&D Canada – Atlantic
   PO Box 1012, Dartmouth NS B2Y 3Z7, Canada

2. SECURITY CLASSIFICATION (Overall security classification of the document including special warning terms if applicable.)

   UNCLASSIFIED

3. TITLE (The complete document title as indicated on the title page. Its classification should be indicated by the appropriate abbreviation (S, C or U) in parentheses after the title.)

   C++ classes for representing curves and surfaces: Part II: Splines

4. AUTHORS (Last name, followed by initials – ranks, titles, etc. not to be used.)

   Hally, D.

5. DATE OF PUBLICATION (Month and year of publication of document.)

   January 2007

6a. NO. OF PAGES (Total containing information. Include Annexes, Appendices, etc.)

   64

6b. NO. OF REFS (Total cited in document.)

   9

7. DESCRIPTIVE NOTES (The category of the document, e.g. technical report, technical note or memorandum. If appropriate, enter the type of report, e.g. interim, progress, summary, annual or final. Give the inclusive dates when a specific reporting period is covered.)

   Technical Memorandum

8. SPONSORING ACTIVITY (The name of the department project office or laboratory sponsoring the research and development – include address.)

   Defence R&D Canada – Atlantic
   PO Box 1012, Dartmouth NS B2Y 3Z7, Canada

9a. PROJECT NO. (The applicable research and development project number under which the document was written. Please specify whether project or grant.)

   11cj18

9b. GRANT OR CONTRACT NO. (If appropriate, the applicable number under which the document was written.)

10a. ORIGINATOR'S DOCUMENT NUMBER (The official document number by which the document is identified by the originating activity. This number must be unique to this document.)

   DRDC Atlantic TM-2006-255

10b. OTHER DOCUMENT NO(s). (Any other numbers which may be assigned this document either by the originator or by the sponsor.)

11. DOCUMENT AVAILABILITY (Any limitations on further dissemination of the document, other than those imposed by security classification.)

   (X) Unlimited distribution
   ( ) Defence departments and defence contractors; further distribution only as approved
   ( ) Defence departments and Canadian defence contractors; further distribution only as approved
   ( ) Government departments and agencies; further distribution only as approved
   ( ) Defence departments; further distribution only as approved
   ( ) Other (please specify):

12. DOCUMENT ANNOUNCEMENT (Any limitation to the bibliographic announcement of this document. This will normally correspond to the Document Availability (11). However, where further distribution (beyond the audience specified in (11)) is possible, a wider announcement audience may be selected.)

13. ABSTRACT (A brief and factual summary of the document. It may also appear elsewhere in the body of the document itself. It is highly desirable that the abstract of classified documents be unclassified. Each paragraph of the abstract shall begin with an indication of the security classification of the information in the paragraph (unless the document itself is unclassified) represented as (S), (C), or (U). It is not necessary to include here abstracts in both official languages unless the text is bilingual.)

Splines are in widespread use as a means of interpolating or approximating discrete data. They provide an efficient means of representing fully differentiable curves. This document describes a library of C++ classes which represent splines of different types; they include one and two parameter splines with a piecewise-polynomial representation, one and two parameter B-spline curves, one and two parameter non-uniform rational B-spline (NURB) curves, one and two parameter Hermite splines, Akima splines and standard cubic splines.

The spline classes are based on the CurveLib library of C++ classes which represent fully differentiable curves of a more general nature. The spline classes inherit many convenient features of the CurveLib classes: they can be added to an arbitrary curve, scaled, composed with an arbitrary curve and inverted.

14. KEYWORDS, DESCRIPTORS or IDENTIFIERS (Technically meaningful terms or short phrases that characterize a document and could be helpful in cataloguing the document. They should be selected so that no security classification is required. Identifiers, such as equipment model designation, trade name, military project code name, geographic location may also be included. If possible keywords should be selected from a published thesaurus. e.g. Thesaurus of Engineering and Scientific Terms (TEST) and that thesaurus identified. If it is not possible to select indexing terms which are Unclassified, the classification of each should be indicated as with the title.)

splines
B-splines
NURBs
differentiable functions
CurveLib
C++
computer programs

This page intentionally left blank.

Defence R&D Canada

Canada's leader in defence
and National Security
Science and Technology

R & D pour la défense Canada

Chef de file au Canada en matière
de science et de technologie pour
la défense et la sécurité nationale

www.drdc-rddc.gc.ca