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 I: Multi-parameter differentiable functions

David Hally

Defence R&D Canada – Atlantic

Technical Memorandum
DRDC Atlantic TM 2006-254
January 2007

Canada

This page intentionally left blank.

C++ classes for representing curves and surfaces

Part I: Multi-parameter differentiable functions

David Hally

Defence R&D Canada – Atlantic

Technical Memorandum

DRDC Atlantic TM-2006-254

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

A library of C++ classes for representing multi-parameter differentiable functions is described. The principal utility of the classes lies in the ability to combine simple curves in a variety of ways to make complex curves while maintaining the differentiability of the result. This can be done using arithmetic functions, composition, vector operators (e.g. dot and cross products) and inverse methods.

The classes also include a wide variety of simple curves which can be used as building blocks, including constants, linear curves, polynomials, exponential functions, trigonometric functions, hyperbolic functions and Bessel functions.

Résumé

Nous décrivons une bibliothèque de classes C++ servant à représenter des fonctions différentiables multipraramétriques. L'utilité principale de ces classes repose sur leur capacité à combiner des courbes simples de plusieurs façons, pour représenter des courbes complexes différentiables. On y parvient en utilisant des fonctions arithmétiques, la composition de fonctions, des opérations vectorielles (produits vectoriel et matriciel) et des méthodes d'inversion.

En outre, ces classes comprennent une grande variété de courbes simples que l'on peut utiliser comme éléments de base : courbes constantes et linéaires, ainsi que fonctions polynomiales, exponentielles, trigonométriques, hyperboliques et besselliennes.

This page intentionally left blank.

Executive summary

C++ classes for representing curves and surfaces: Part I: Multi-parameter differentiable functions

David Hally; DRDC Atlantic TM-2006-254; 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. The current document describes a library of C++ classes which can be used for this purpose.

Principal results: A library of C++ classes for representing multi-parameter differentiable functions is described. The principal utility of the classes lies in the ability to combine simple curves in a variety of ways to make complex curves while maintaining the differentiability of the result. This can be done using arithmetic functions, composition, vector operators (e.g. dot and cross products) and inverse methods.

The classes also include a wide variety of simple curves which can be used as building blocks, including constants, linear curves, polynomials, exponential functions, trigonometric functions, hyperbolic functions and Bessel functions.

Significance: The library of C++ classes provides a useful tool for representing complex geometry for use in CFD programs. However, their definition is quite general, so they can be used in a wide variety of applications.

Sommaire

C++ classes for representing curves and surfaces: Part I: Multi-parameter differentiable functions

David Hally ; DRDC Atlantic TM-2006-254 ; 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 pouvoir toutefois représenter la géométrie du navire ou de l'hélice d'une manière compatible avec les logiciels de dynamique numérique des fluides. Le présent document décrit une bibliothèque de classes C++ que l'on peut utiliser à cette fin.

Résultats : Nous décrivons une bibliothèque de classes C++ servant à représenter des fonctions différentiables à plusieurs paramètres. L'utilité principale de ces classes repose sur leur capacité à combiner des courbes simples de plusieurs façons, pour représenter des courbes complexes différentiables. On y parvient en utilisant des fonctions arithmétiques, la composition de fonctions, des opérations vectorielles (produits vectoriel et matriciel) et des méthodes d'inversion. En outre, ces classes comprennent une grande variété de courbes simples que l'on peut utiliser comme éléments de base : courbes constantes et linéaires, ainsi que fonctions polynomiales, exponentielles, trigonométriques, hyperboliques et besselliennes.

Importance : La bibliothèque de classes C++ 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 leur définition est très générale, elles peuvent être appliquées à différents domaines.

Table of contents

Abstract i

Résumé i

Executive summary iii

Sommaire iv

Table of contents v

1 Introduction 1

2 Base curve classes 2

3 Exceptions 5

4 Arithmetic operators for curves 5

5 Composition of curves 7

6 Simple curves 9

6.1 Elementary functions 9

6.1.1 Trigonometric functions 10

6.1.2 Exponential functions 11

6.1.3 Hyperbolic functions 12

6.1.4 Bessel functions 13

6.2 Identity curves 14

6.3 Abs and Unit 15

6.4 Derivative curves 15

6.5 Constant parameter curves 16

6.6 Complex conjugate curves 17

6.7 Vector curves from scalar curves 17

6.8 Linear curves 19

6.9 Linear transformations of parameters 20
6.10 Polynomials 21
7 Curves made from vector-valued curves 22
7.1 Selecting a single component 22
7.2 Throwing away a vector component 23
7.3 Concatenating vector values 23
7.4 Reflection in a plane 24
7.5 Dot products 25
7.6 Cross products 25
7.7 Unit vectors 26
7.8 Projections 26
7.8.1 Projection to a plane 27
7.8.2 Projection to a sphere 28
7.8.3 Projection to a cylinder 28
8 Surfaces 29
8.1 Axi-symmetric surfaces 31
9 Interpolation between boundary curves 33
9.1 Ruled curves 33
9.2 Transfinite interpolation 34
10 Curves with parameter ranges 37
10.1 Parameter ranges 37
10.2 Range curves 38
10.3 Standard range curves 40

11 Implicitly defined curves 40
11.1 Newton-Raphson search 40
11.2 Implicit curves 41
11.3 Inverse curves 45
12 Defining a new curve 47
13 Concluding remarks 51
References 52
Annex A: Concepts 53
A.1 Arithmetic Object 53
A.2 Scalar Object 55
A.3 Comparable Scalar Object 56
A.4 Vector Object 57
A.5 Absolute Object 58
Annex B: Prototypes for VecMtx::VecN 61
B.1 Constructors 61
B.2 Other member functions 61
B.3 Other functions 62
Annex C: Prototypes for VecMtx::MtxN 65
C.1 Constructors 65
C.2 Other member functions 65
C.3 Other functions 65
Annex D: Prototypes for CurveLib::Derivs 69
D.1 Constructors 69
D.2 Other member functions 69
D.3 Other functions 70

Annex E: Prototypes for class Angle 71
E.1 Constructors 71
E.2 Static members 71
E.3 Member functions for setting and retrieving the angle 71
E.4 Trigonometric functions 72
Annex F: Prototypes for class Error 73
F.1 Prototypes for class ProgError 73
Index 74

1 Introduction

Many computer applications need to be able to represent differentiable functions of considerable complexity. This document describes a library of C++ classes for this purpose. It was originally designed for representing the geometry of complex shapes (e.g. ship hulls and propellers), so the functions are called curves. In fact, they are much more general than the name implies. Any multi-valued multi-parameter differentiable function can be represented by a 'curve'.

The principal utility of the curve classes lies in the ability to combine simple curves in a variety of ways to make complex curves. This can be done using arithmetic functions (Section 4), composition (see Section 5), vector operators (e.g. dot and cross products: see Section 7) and inverse methods (see Section 11). Each curve so constructed is fully differentiable.

The curve classes also include a wide variety of simple curves which can be used as building blocks (see Section 6). These include constants, linear curves, polynomials, exponential functions, trigonometric functions, hyperbolic functions and Bessel functions.

Because of their importance in geometric applications, two-parameter curves which return three-dimensional points (i.e. curves which define a surface in three-dimensional space) are given special treatment. They are described in Section 8.

An important application of curves is interpolation. Basic classes for interpolating from boundaries are described in Section 9. A companion document[1] describes classes which implement splines.

Section 10 describes an extension to the basic curve class in which the curves are given well-defined parameter ranges.

Section 12 describes how to define a new curve class for the cases in which it is inconvenient, or inefficient, to generate a curve from the basic curve classes. This section also contains information on how the curve classes are implemented.

There are three other documents which are companions to this one. The first[1] describes C++ classes for implementing splines. The second[2] describes classes for saving curves 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 Base curve classes

All the curve classes are encapsulated in the namespace `CurveLib`. Almost all the classes are templates. To define the attributes required of template arguments we will use the Standard Template Library[5] 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 CurveLib 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.

Every curve is derived from the template base class `Curve<N,V,F>`. It 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 type `F` must be a model of a Scalar Object: see Annex A.2. Loosely speaking, a Scalar Object is a floating point number; `float`, `double`, `std::complex<float>` and `std::complex<double>` are all models of a Scalar Object. The type `V` must be a model of an Arithmetic Object with respect to `F`: see Annex A.1. This requires it to have the arithmetic operators one normally expects from scalars, vectors or matrices.

If the template parameter `F` is omitted from `Curve<N,V,F>`, it defaults to `double` (there is one exception: see Section 6.6). Classes derived from `Curve<N,V,F>` typically allow the type of the parameters to be specified by `F`. Whenever this is the case, `F` appears last in the template parameter list and is given the default type `double`. Thus, the class `Sqrt<double>`, whose template argument is the type of both its parameter and of its returned value, can be represented more simply by `Sqrt<>`.

The list of parameters is represented by a class `Curve<N,V,F>::ParamType`. It 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 Annex A.4). In the current implementation `ParamType` is set using a `typedef` to the class `VecN<N,F>` in the namespace `VecMtx`. A list of function prototypes for `VecN<N,F>` is given in Annex B.

The number of derivatives to be taken with respect to each parameter is specified using an instance of `Curve<N,V,F>::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 `DerivType` is set to the class `Derivs<N>` using a `typedef`. A list of function prototypes for `Derivs<N>` is given in Annex D.

In all curves derived from `Curve<N,V,F>` the type `ValueType` is equivalent to the type of the returned value: i.e. it is simply an alias for `V`. For example, knowing that the class `Sqrt<F>` is derived from `Curve<N,V,F>` for some `N`, `V` and `F`, one can determine

the return type using `Sqrt<F>::ValueType` (it is equivalent to `F`).

All curves derived from `Curve<N,V,F>` in the `CurveLib` library have a default constructor: i.e. a constructor having no arguments. For some curves this is sufficient to define the curve (e.g. `Sqrt<F>`: see Section 6.1); for others the curve remains undefined if the default constructor is used. It can later be defined using specialized member functions in the derived class or by assignment to another curve. This paradigm can be useful, for example, if the curve is a member of a class but there is not enough information available to define the curve when the class is constructed. It is recommended that all classes derived from `Curve<N,V,F>` include a default constructor.

Since a curve may remain undefined if a default constructor is used, `Curve<N,V,F>` provides the following member function for determining whether the curve 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 (see Section 3).

An important property of curves is that they are polymorphic, even though the class `Curve<N,V,F>` has no virtual functions. For example, suppose a `Sqrt<>` is assigned to a `Curve<1U,double>`:

```
using namespace CurveLib;
Curve<1U,double> c;
Sqrt<> sqrtx;
c = sqrtx;
```

When evaluated with the same arguments, `c` will return the same value as `sqrtx`.

The function represented by a `Curve<N,V,F>` may be evaluated using the following two member functions:

`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, suppose that we defined a curve `sqrtx2y2` by

```
using namespace CurveLib;
FOneParamCurve<2U> x(0), y(1);
Sqrt<> sqrtx;
Curve<2U,double> sqrtx2y2 = sqrtx(x*x+y*y);
```

It represents the function $f(x,y) = \sqrt{x^2+y^2}$: see Sections 6.2 and 6.1 for descriptions of `FOneParamCurve<N,F>` and `Sqrt<F>`. To evaluate $f(0.5, 0.8)$ use

```
Curve<2U,double>::ParamType p(0.5,0.8);
double value = sqrtx2y2(p);
```

To evaluate $\frac{\partial^2 f}{\partial y^2}(0.5, 0.8)$ use

```
Curve<2U,double>::ParamType p(0.5,0.8);
Curve<2U,double>::DerivType d(0,2);
double value = sqrtx2y2(p,d);
```

One parameter curves are a special case. As well as the two prototypes for `operator()` defined above, they also have the following two member functions which are usually more convenient.

`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,

```
using namespace CurveLib;
Curve<1U,double> sqrtx = Sqrt<>();
double value = sqrtx(2.0);    // value = sqrt(2.0) = 1.414213...
double deriv2 = sqrtx(2.0,2); // deriv2 = the value of sqrt(x)
                              // differentiated twice at x = 2.0
                              // i.e. deriv2 = -1/(8*sqrt(2.0))
                              //             = -0.0883883...
```

3 Exceptions

All exceptions thrown by `CurveLib` classes and functions are derived from the base class `Error`; it is *not* in the namespace `CurveLib`. An `Error` contains a message which can be retrieved, appended to, or prepended to. The prototypes of the `Error` member functions are listed in Annex F.

It is wise, when using curves, to enclose the body of the code in a `try` block which catches an `Error`. For example:

```
try {
  ... // Code which uses CurveLib classes
}
catch (Error &e) {
  // Write the error message
  std::cerr << e.get_msg() << '\n';
}
```

Another important exception is `ProgError`, a specialization of `Error`. It is thrown when an exception occurs that can clearly be recognized as a programming error rather than a run-time error. The occurrence of a `ProgError` is an indication that the program is faulty. The prototypes of the `ProgError` member functions are listed in Annex F.1.

4 Arithmetic operators for curves

One of the most important ways of combining two curves to create a new one is by using arithmetic operators. For example, if `f` and `g` are two curves having the same number of arguments and the same return type, then `f+g` is a curve which returns their sum. Similarly, `f*g` returns their product, provided that the return type permits multiplication (however, see the description of the return type of `f*g` below).

```
using namespace CurveLib;
Cos<> cosx;
Sin<> sinx;
Curve<1U,double> one = cosx*cosx + sinx*sinx;
```

The curve `one` evaluates $\cos^2(x) + \sin^2(x)$.

Let `f` be a curve of type `Curve<N,V,F>`, `g` be a curve of type `Curve<N,V1,F>`, `v` be a constant of type `V`, and `s` be a constant of type `F`. Then the following arithmetic operators are defined:

`-f`

A curve which is the negation of `f`: i.e. for any set of parameters `x`, `(-f)(x)` is equivalent to `-(f(x))`.

`f+g`

A curve which returns the sum of `f` and `g`: i.e. `(f+g)(x)` is equivalent to `f(x)+g(x)`. The return types of `f` and `g` must be the same: i.e. `V` and `V1` are the same.

`f+v`

A curve which returns the sum of `f` and `v`: i.e. `(f+v)(x)` is equivalent to `f(x)+v`.

`v+f`

A curve which returns the sum of `v` and `f`: i.e. `(v+f)(x)` is equivalent to `v+f(x)`.

`f-g`

A curve which returns the difference of `f` and `g`: i.e. `(f-g)(x)` is equivalent to `f(x)-g(x)`. The return types of `f` and `g` must be the same: i.e. `V` and `V1` are the same.

`f-v`

A curve which returns the difference of `f` and `v`: i.e. `(f-v)(x)` is equivalent to `f(x)-v`.

`v-f`

A curve which returns the difference of `v` and `f`: i.e. `(v-f)(x)` is equivalent to `v-f(x)`.

`f*g`

A curve which returns the product of `f` and `g`: i.e. `(f*g)(x)` is equivalent to `f(x)*g(x)`. The product `V*V1` must be defined and return a `V1`. Note that this allows for the common case in which `V` is a matrix type and `V1` is a vector type, but not vice versa. Multiplication of a matrix by a vector (i.e. $\mathbf{v} \cdot \mathbf{M}$ where $\mathbf{v}$ is a vector and $\mathbf{M}$ is a matrix) must be handled by multiplying the vector by the transpose of the matrix.

Note also that if `f` returns a vector type and `g` returns a scalar type, then `f*g` will not be defined since the result of the multiplication is a vector but `f*g` returns a scalar. Therefore the products of vector-valued and scalar-valued curves must always be ordered so that the scalar-valued curve comes first.

`f*s`

A curve which returns the product of `f` and `s`: i.e. `(f*s)(x)` is equivalent to `f(x)*s`.

`s*f`
: A curve which returns the product of `s` and `f`: i.e. `(s*f)(x)` is equivalent to `s*f(x)`.

`f/g`
: A curve which returns `f` divided by `g`: i.e. `(f/g)(x)` is equivalent to `f(x)/g(x)`. The return value of `g` must be a scalar (`V = F`). There is no checking for divide by zero.

`f/s`
: A curve which returns `f` multiplied by `1/s`: i.e. `(f/s)(x)` is equivalent to `f(x)*(1/s)`. There is no checking for division by zero.

The arithmetic operators `+=`, `*=`, etc. are not defined. This is because we do not want to allow the following type of construction:

```
using namespace CurveLib;
Sqrt<> sqrtx;  // sqrtx(x) = sqrt(x)
Cos<> cosx;    // cosx(x)  = cos(x)
sqrtx += cosx; // Now sqrtx = sqrt(x) + cos(x); not allowed
```

A curve of type `Sqrt<>` should only be allowed to return $\sqrt{x}$.

Instances of the base class `Curve<N,V,F>` can also be set to return a constant value by assignment to an instance of `V`.

`f = v`
: A curve which returns the value `v` (its derivatives are identically zero).

This construction is not allowed for derived classes.

5 Composition of curves

The composition operator is another important way of combining two curves to create a new one. Let $f(x_1, \ldots, x_N)$ and $g(x_1, \ldots, x_M)$ be N and M parameter curves respectively. If g returns a vector of length N, then its value can be used as the argument list for f. Therefore we can define a new M parameter function $h(x_1, \ldots, x_M) = f\big(g(x_1, \ldots, x_M)\big)$. The value returned by h is of the same type as the value returned by f.

The composition of two functions is supported in the `CurveLib` library by the class `ComposedCurve<N,M,V,F>`. It defines types for the outer curve (f in the example above) and inner curve (g in the example above):

```
typedef Curve<M,V,F> OuterCurve;
typedef Curve<N,typename OuterCurve::ParamType,F> InnerCurve;
```

The prototype for its constructor is then

`ComposedCurve(const OuterCurve &f, const InnerCurve &g)`
: Construct the composition of curves `f` and `g`.

A composed curve is also defined for the case when the outer curve, `f`, has a single parameter and the inner curve, `g`, returns a scalar of type `F`. The class `ComposedCurve` cannot be used in this case because the returned type of `g` is not the same as the type of the parameter list of `f`. For this case the class `ComposedCurveFParam<N,V,F>` is used. It defines

```
typedef Curve<1U,V,F> OuterCurve;
typedef Curve<N,F,F> InnerCurve;
```

Its constructor then has the prototype

`ComposedCurveFParam(const OuterCurve &f, const InnerCurve &g)`
: Construct the composition of curves `f` and `g`.

In practice the use of `ComposedCurve` and `ComposedCurveFParam` is rather clumsy. Instead the class `Curve<N,V,F>` provides a much more elegant alternative with the following member function:

```
template<unsigned M>
Curve<M,V,F> operator()(const Curve<M,ParamType,F> &c) const
```
: Returns the composition of `*this` with `c`: i.e. returns a curve equivalent to `(*this)(c(p))` for any set of parameters `p`.

This function is only defined if template member functions are allowed by the compiler. If they are not, then the composition operator is allowed for the cases `M` equal to 1, 2 or 3.

For example, let $f(\mathbf{x})$ be a function of the 3-vector $\mathbf{x}$. We wish to shift the origin of the coordinates to define $g(\mathbf{x}) = f(\mathbf{x} - \mathbf{x}_o)$ where $\mathbf{x}_o$ is the new origin. This can be done as follows for arbitrary `V` and `F`:

```
using namespace CurveLib;
IdentityCurve<3U,F> x;
typename Curve<3U,V,F>::ParamType xo;
Curve<3U,V,F> f;
// ... define xo and f
Curve<3U,V,F> g = f(x-xo);
```

Here `x-xo` is a curve which takes three arguments and returns a 3-vector of type `Curve<3U,V,F>::ParamType` (Section 6.2 describes the class `IdentityCurve<N,F>`). It is composed with `f` to define the new curve `g`.

One parameter curves also have the following member function which allows compo-

sition with a curve returning a value of type F.

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

Returns a curve equivalent to (*this)(c(p)) for any parameter p.

For example, in the following code sqcosx is a curve representing the function $\sqrt{\cos(x)}$:

```
using namespace CurveLib;
Cos<> cosx;
Sqrt<> sqrtx;
Curve<1U,double> sqcosx = sqrtx(cosx);
```

6 Simple curves

The CurveLib library defines a wide variety of simple curves which can be used as building blocks for more complicated curves.

6.1 Elementary functions

The CurveLib library defines a large number of fully differentiable elementary functions. Where possible these functions are evaluated using the standard math library functions sqrt, pow, log, exp, erf, erfc, sin, cos, tan, asin, acos, atan, atan2, sinh, cosh, tanh, asinh, acosh, atanh, j0, j1, jn, y0, y1 and yn. These functions are called with arguments of type F and are assumed to return an appropriately accurate value convertible to an F. It can normally be assumed that all these functions except atan2 and the Bessel functions (j0, j1, jn, y0, y1 and yn) will be defined for F of type float, double, std::complex<float> and std::complex<double>. On some machines they may be defined for F of type long double and std::complex<long double> as well. The Bessel functions and atan2 are not normally defined for the complex types.

The first few functions raise their arguments to different powers:

Sqrt<F>

Represents $\sqrt{x}$. It is derived from Curve<1U,F,F>. It has only a default (no argument) constructor. If F represents a real number, then the value is not defined when x is negative. If F represents a complex number, $z = re^{i\theta}$ for $\theta \in (-\pi, \pi]$, the value returned is $\sqrt{r}e^{i\theta/2}$: i.e. there is a branch cut along the negative real axis.

Pow<F>

A two parameter function which represents $f(x, y) = x^y$; it is derived from

`Curve<2U,F,F>`. It has a single default constructor. If x is not positive, the result returned may vary from machine to machine or for different types `F` (for example, on some machines if $x = 0$ and $y = 2$, the value returned will be 0.0 if `F` is `double` but `NaN` if `F` is a `std::complex<double>`). It is best to assume that the value of the curve is undefined if x is non-positive.

`PowInt<F>`
: A single parameter function which represents $f(x) = x^n$ where n is an integer. The constructor for `PowInt<F>` has a single integer argument giving the value of n. For example, `PowInt<F>(-1)` is a curve representing $f(x) = 1/x$.

6.1.1 Trigonometric functions

Classes implementing the six standard trigonometric functions are defined.

`Sin<F>`
: Represents $\sin(x)$.

`Cos<F>`
: Represents $\cos(x)$.

`Tan<F>`
: Represents $\tan(x)$. Evaluation at odd multiples of $\pi/2$ is undefined.

`Csc<F>`
: Represents $\csc(x) = 1/\sin(x)$. Evaluation at multiples of π is undefined.

`Sec<F>`
: Represents $\sec(x) = 1/\cos(x)$. Evaluation at odd multiples of $\pi/2$ is undefined.

`Cot<F>`
: Represents $\cot(x) = 1/\tan(x)$. Evaluation at multiples of π is undefined.

Each of these functions has a single parameter and returns a scalar of type `F`; they are derived from `Curve<1U,F,F>`. Each has only a default (no argument) constructor.

The following inverse trigonometric functions are also defined. Each has a single parameter and returns a scalar of type `F`; they are derived from `Curve<1U,F,F>`. Each has only a default (no argument) constructor.

`ArcSin<F>`
: Represents $\arcsin(x)$. If `F` represents a real number, then the returned value is in the range $[-\pi/2, \pi/2]$. If `F` represents a complex number (i.e. it is of type

`std::complex<F1>` for some `F1`), then the returned value is computed using:

$$\arcsin(x) = -i \ln(\sqrt{1 - x^2} + ix) \tag{1}$$

with branch cuts along the real axis outside the range $[-1, 1]$. It has real part in the range $[-\pi/2, \pi/2]$.

`ArcCos<F>`

Represents $\arccos(x)$. If `F` represents a real number, then the returned value is in the range 0 to π. If `F` represents a complex number, then the returned value is computed using:

$$\arccos(x) = \frac{\pi}{2} - \arcsin(x) \tag{2}$$

and has real part in the range $[0, \pi]$.

`ArcTan<F>`

Represents $\arctan(x)$. If `F` represents a real number, then the returned value is in the range $(-\pi/2, \pi/2)$. If `F` represents a complex number, then the returned value is computed using:

$$\arctan(x) = -\frac{i}{2} \ln\left(\frac{1 + ix}{1 - ix}\right) \tag{3}$$

with branch cuts on the imaginary axis outside the range $[-i, i]$. It has real part in the range $(-\pi/2, \pi/2)$.

The branch cuts for the complex versions of these functions have been chosen to lie on the real or imaginary axis such that the curve is continuous across the portion of the real axis where the real version of the function is well-defined.

In addition, the two-argument function `ArcTan2<F>` is defined if `F` represents a real number; it is derived from `Curve<2U,F,F>`. It represents $\arctan(y/x)$, where x is the first parameter and y is the second parameter. The value returned is in the range $[-\pi, \pi]$. If both x and y are zero, the value is not defined.

6.1.2 Exponential functions

Classes implementing the following exponential functions are defined. Each has a single parameter and returns a scalar of type `F`; they are derived from `Curve<1U,F,F>`.

`Exp<F>`

Represents e^x. This class has only a default constructor.

`Log<F>`

Represents $\log_b(x)$. This class has a one argument constructor whose prototype

is: `Log(F b = F(0))`. It constructs a curve which returns the logarithm of its argument to the base `b`. For example, the code

```
using namespace CurveLib;
Log<> log10(10.0);
std::cout << log10(100.0);
```

would result in the value 2 being printed. If the base `b` is zero, natural logarithms (i.e. base e) are assumed. If `F` represents a real number, then `b` must be non-negative. The returned value is not defined if the function parameter, x, is zero or if `F` represents a real number and x is negative. If `F` represents a complex number, the imaginary part of the returned value is in the range $(-\pi, \pi]$; there is a branch cut along the negative real axis.

`Erf<F>`
: Represents

$$\operatorname{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2}\, dt \tag{4}$$

The template argument `F` must represent a real number. This class has only a default constructor.

`Erfc<F>`
: Represents $1 - \operatorname{erf}(x)$. The template argument `F` must represent a real number. This class has only a default constructor.

6.1.3 Hyperbolic functions

Classes implementing the six standard hyperbolic functions are defined.

`Sinh<F>`
: Represents $\sinh(x)$.

`Cosh<F>`
: Represents $\cosh(x)$.

`Tanh<F>`
: Represents $\tanh(x)$.

`Csch<F>`
: Represents $\operatorname{csch}(x) = 1/\sinh(x)$.

`Sech<F>`
: Represents $\operatorname{sech}(x) = 1/\cosh(x)$.

`Coth<F>`
: Represents $\coth(x) = 1/\tanh(x)$.

Each of these functions has a single parameter and returns a scalar of type `F`; they are derived from `Curve<1U,F,F>`. Each has only a default (no argument) constructor.

The following inverse hyperbolic functions are also defined. Each has a single parameter and returns a scalar of type `F`; they are derived from `Curve<1U,F,F>`. Each has only a default (no argument) constructor.

`ArcSinh<F>`
: Represents $\operatorname{arcsinh}(x)$. If `F` represents a complex number, then the returned value is computed using:

$$\operatorname{arcsinh}(x) = -i \arcsin(ix) \tag{5}$$

and has imaginary part in the range $[-\pi/2, \pi/2]$. There is a branch cut on the imaginary axis outside the range $[-i, i]$.

`ArcCosh<F>`
: Represents $\operatorname{arccosh}(x)$. If `F` represents a real number, then x must equal or exceed 1 and the returned value will be non-negative. If `F` represents a complex number, then the returned value is computed using:

$$\operatorname{arccosh}(x) = \ln\bigl(x + \sqrt{x^2 - 1}\bigr) \tag{6}$$

with a branch cut on the real axis in the range $(-1, 1)$. It has imaginary part in the range $[0, \pi]$.

`ArcTanh<F>`
: Represents $\arctan(x)$. If `F` represents a real number, then the returned value is in the range $-\pi/2$ to $\pi/2$. If `F` represents a complex number, then the returned value is computed using:

$$\operatorname{arctanh}(x) = -i \arctan(ix) \tag{7}$$

and has imaginary part in the range $[-\pi/2, \pi/2]$. There is a branch cut on the real axis outside the range $[-1, 1]$.

The branch cuts for the complex versions of these functions have been chosen to lie on the real or imaginary axis such that the curve is continuous across the portion of the real axis where the real version of the function is well-defined.

6.1.4 Bessel functions

Classes implementing the following Bessel functions are defined. Each of these functions has a single parameter and returns a scalar of type `F` where `F` represents a real number; they are derived from `Curve<1U,F,F>` where `F` is a model of a Comparable Scalar Object.

`BesselJ<F>`

Represents the Bessel function of the first kind of integer order n, $J_n(x)$. This class has the following constructor: `BesselJ(int n = 0)`. The constructor argument is the order, n. Derivatives are calculated using the relation

$$2J'_n(x) = J_{n-1}(x) - J_{n+1}(x) \tag{8}$$

`BesselY<F>`

Represents the Bessel function of the second kind of integer order n, $Y_n(x)$. This class has the following constructor: `BesselY(int n = 0)`. The constructor argument is the order, n. Derivatives are calculated using the relation

$$2Y'_n(x) = Y_{n-1}(x) - Y_{n+1}(x) \tag{9}$$

6.2 Identity curves

The curve `IdentityCurve<N,F>` simply returns its parameter as its value. Therefore both its parameter list and its return value are of type `ParamType`. It has only the default and copy constructors.

For example, the function $f(x, y, z) = x^2 + y^2 + z^2$ could be defined as follows:

```
using namespace CurveLib;
typedef IdentityCurve<3U>::ValueType VType;
IdentityCurve<3U> x;
Curve<3U,double> f = DotCurve<3U,VType>(x,x);
```

The class `DotCurve<N,V,F>` is described in Section 7.5. It returns the dot product of two vector-valued functions. Section 5 has another example of the use of an `IdentityCurve<N,F>`.

An `IdentityCurve<1U,F>` is a single argument curve which returns its argument as a `ParamType` (i.e. a vector of length 1). It is often more convenient to have a single argument identity curve which returns the argument as a `F`. The class `FIdentityCurve<F>` fulfills this role. `FIdentityCurve<F>` has only the default and copy constructors.

For example, suppose we wish to define a curve to represent the function $\sin(2\pi x)$. This can be done easily as follows:

```
using namespace CurveLib;
Sin<> sinx;
FIdentityCurve<> x;
double twopi = 2.0*3.14159265358979323846;
Curve<1U,double> sin2pix = sinx(twopi*x);
```

It is also sometimes useful to be able to extract a single parameter from a parameter list. The curves `OneParamCurve<N,F>` and `FOneParamCurve<N,F>` do this. They return the parameter as a `ParamType` (i.e. a vector of length 1) and an `F` respectively.

`OneParamCurve<N,F>` has one constructor besides the default and copy constructors.

`OneParamCurve(unsigned n)`
: The argument `n` is the index of the parameter whose value is to be returned. If `n` equals or exceeds `N` a `ProgError` exception will be thrown.

The constructors of `FOneParamCurve<N,F>` are similar.

`FOneParamCurve(unsigned n)`
: The argument `n` is the index of the parameter whose value is to be returned. If `n` equals or exceeds `N` a `ProgError` exception will be thrown.

Both `OneParamCurve<N,F>` and `FOneParamCurve<N,F>` remain undefined if the default constructor is used.

See Section 6.4 for an example of the use of `FOneParamCurve<N,F>`.

6.3 Abs and Unit

The curve `Abs<N,F>` treats its argument list as a vector and returns its magnitude: i.e. it represents the function

$$f(x_1, \ldots, x_N) = \sqrt{x_0^2 + \cdots + x_{N-1}^2} \tag{10}$$

It is a specialization of `Curve<N,F,F>` and has only the default and copy constructors. No additional member functions are defined beyond those inherited from the base class.

The curve `Unit<N,F>` treats its argument as a vector and returns the corresponding unit vector. It is a specialization of `Curve<N,ParamType,F>` and has only the default and copy constructors. No additional member functions are defined beyond those inherited from the base class. An instance of `Unit<N,F>` is equivalent to `IdentityCurve<N,F>()/Abs<N,F>()`.

6.4 Derivative curves

The class `DerivCurve<N,V,F>` represents a curve which is the derivative of another curve. It has two constructors besides the default and copy constructors:

`DerivCurve(const Curve<N,V,F> &c1, const DerivType &d)`
: Makes a curve equal to the derivatives of curve `c` as specified by the derivative specifier `d`.

`DerivCurve(const Curve<N,V,F> &c, unsigned i)`

Makes a curve equal to the single derivative of `c` with respect to parameter `i`.

For example,

```
using namespace CurveLib;
FOneParamCurve<2U> x(0), y(1);
Curve<2U,double> x2y2 = x*x+y*y;

Derivs<2U> d(0,2);                 // Two derivatives by y
DerivCurve<2U,double> c1(x2y2,d); // c1 = d^2(x2y2)/dy^2 = 2.0
DerivCurve<2U,double> c2(x2y2,1); // c2 = d(x2y2)/dy = 2.0*y
DerivCurve<2U,double> c3(c2,1);   // c3 = d(c2)/dy = 2.0
```

6.5 Constant parameter curves

It is often useful to be able to restrict a multi-parameter curve so that one of its parameters is held constant. For example, this can be done to define a curve along the edge of a surface. The class `ConstPCurve` is used for this purpose.

`ConstPCurve` has two possible template argument lists, depending on the capabilities of the compiler. If template arithmetic is allowed, `ConstPCurve` is defined as follows.

```
namespace CurveLib {
  template<unsigned N, class V, class F = double>
  class ConstPCurve: public Curve<N,V,F>
  {
  public:
    ConstPCurve();
    ConstPCurve(const Curve<N+1,V,F> &c, unsigned i, F v,
                unsigned deriv = 0);
  };
}
```

If the default constructor is used the curve remains undefined. The second constructor makes the curve from curve `c` by keeping parameter `i` fixed at value `v`. If `deriv` is non-zero, it is the `deriv`$^{\text{th}}$ derivative of `c` with respect to parameter `i` which is returned.

For example, the function $f(x) = x^{\frac{3}{2}}$ could be defined as follows:

```
using namespace CurveLib;
Curve<1U,F> sqrtx1p5 = ConstPCurve<1U,F>(Pow<>(),1,1.5);
```

If template arithmetic is not allowed, the following definition is used instead:

```
namespace CurveLib {
  template<unsigned N, unsigned NP1, class V, class F = double>
  class ConstPCurve: public Curve<N,V,F>
  {
  public:
    ConstPCurve();
    ConstPCurve(const Curve<NP1,V,F> &c, unsigned i, F v,
                unsigned deriv = 0);
  };
}
```

Here `NP1` must be equal to `N+1`. In this case, the example above would have to be modified to:

```
using namespace CurveLib;
Curve<1U,F> sqrtx1p5 = ConstPCurve<1U,2U,F>(Pow<>(),1,1.5);
```

Note that because zero parameter curves are not allowed, it is not possible to hold the parameter of a one-parameter curve constant. However, since the resulting curve would have constant value, this rarely causes a problem.

6.6 Complex conjugate curves

When a curve returns a complex value it is often useful to be able to convert the curve so that it returns the complex conjugate of the value instead. This is done by the class `ComplexConjCurve<N,F1,F>`. The template argument `N` is the number of parameters of the curve to be converted. The value returned by both curves is of type `std::complex<F1>`. The template argument `F` is the type of each parameter. The default parameter type if `F` is omitted is `std::complex<F1>`, an exception to the rule that the default type for `F` is `double`.

`ComplexConjCurve<N,F1,F>` has the following constructor in addition to the default and copy constructors and the assignment operator:

`ComplexConjCurve(Curve<N,std::vector<F1>,F> c)`
: Constructs a curve which returns the complex conjugate of `c`.

6.7 Vector curves from scalar curves

A `MultiCurve<N,M,F>` is a curve with `N` arguments which returns a vector of length `M`. Each component of the value of a `MultiCurve<N,M,F>` is obtained by evaluating a different scalar-valued curve. All the scalar curves have `N` arguments.

The type of the return value is the same as the type of a parameter list of an `M`-parameter curve: i.e. it is the same as `Curve<N,V,F>::ParamType` for any `V`. This

ensures that the returned value of a `MultiCurve<N,M,F>` can be used as the parameter list for a `Curve<M,V,F>`. Therefore a `Curve<M,V,F>` can be composed with a `MultiCurve<N,M,F>`. In the current implementation the returned value is actually a `VecMtx::VecN<M,F>`.

The type `MultiCurve<N,M,F>::CompCurveType` is defined to be equivalent to the type of each component curve (`Curve<N,F,F>`).

`MultiCurve<N,M,F>` has one constructor besides the default and copy constructors.

`MultiCurve(const std::vector<CompCurveType> &clist)`
: Makes the curve from the component curves in `clist`. If the length of `clist` is not `N` a `ProgError` exception will be thrown.

If the default constructor is used the curve is defined but each of its component curves remains undefined.

The curve used to evaluate a component of the multi-curve can be set using the following member function:

`void set_curve(unsigned i, const CompCurveType &c)`
: Sets curve `i` to be `c`. If `i` is equal to or exceeds `M` a `ProgError` exception will be thrown. `set_curve` can be called even if the curve is undefined; it will be defined when `set_curve` returns. Note, however, that by defining the curve in this way it is possible that the multi-curve will be defined but that one of its component curves will not be (for example, if `set_curve` is called only once while `M` is two or greater).

The subscript operator can be used to obtain the component curves:

```
const CurveCurveType& operator[](unsigned i) const;
CurveCurveType& operator[](unsigned i)
```
: Each of these functions returns the $\texttt{i}^{\text{th}}$ component curve.

For example, suppose we wish to represent the vector-valued curve $\mathbf{f}(r,\theta)$ with

$$f_x(r,\theta) = r\cos\theta; \qquad f_y(r,\theta) = r\sin\theta \tag{11}$$

The curve `xpolar` defined below does the trick:

```
using namespace CurveLib;
FOneParamCurve<2U> r(0), theta(1);
MultiCurve<2U,2U> xpolar;
xpolar.set_curve(0,r*Cos<>()(theta));
xpolar.set_curve(1,r*Sin<>()(theta));
```

Alternatively we could have used:

```
using namespace CurveLib;
FOneParamCurve<2U> r(0), theta(1);
MultiCurve<2U,2U> xpolar;
xpolar[0] = r*Cos<>()(theta);
xpolar[1] = r*Sin<>()(theta);
```

6.8 Linear curves

A `LinearCurve<V,F>` is a single argument curve which has constant first derivative. It can be defined by specifying two points which it must pass through, (x_1, y_1) and (x_2, y_2):

$$f(x) = \frac{(x_2 - x)y_1 + (x - x_1)y_2}{x_2 - x_1} \tag{12}$$

Alternatively it can be defined by specifying a point, (x_1, y_1), and a slope, m:

$$f(x) = y_1 + m(x - x_1) \tag{13}$$

`LinearCurve` has the following constructors besides the default and copy constructors:

`LinearCurve(F x1, const V &y1, F x2, const V &y2)`
: Constructs a curve which interpolates linearly between the points `(x1,y1)` and `(x2,x2)`. The curve guarantees that, if evaluated at a parameter exactly equal to `x1`, then the exact value `y1` will be returned: i.e. there will be no round-off error. Similarly, the exact value `y2` is returned if the curve is evaluated at `x2`.

`LinearCurve(F x1, const V &y1, const V &m)`
: Constructs a linear curve which passes through the point `(p1,v1)` (with no round-off error) and has slope `m`. In this case the curve guarantees that if the first derivative of the curve is evaluated, the exact value of `m` is returned.

The curve remains undefined if the default constructor is used. It can be defined after construction by assignment to another `LinearCurve<V,F>` or using the following member functions:

`void define(F x1, const V &y1, F x2, const V &y2)`
: Modifies the curve so that it interpolates linearly between the points `(x1,y1)` and `(x2,x2)`.

`void define(F x1, const V &y1, const V &m)`
: Modifies the curve so that it passes through the point `(p1,v1)` and has slope `m`.

A `BilinearPatch<V,F>` is a two-parameter curve which interpolates linearly between four corner points. It can also be defined as a ruled curve (see Section 9.1) between

two linear curves.

$$\begin{aligned} f(\xi,\eta) &= f(\xi_{lo},\eta_{lo})\frac{(\xi_{hi}-\xi)(\eta_{hi}-\eta)}{(\xi_{hi}-\xi_{lo})(\eta_{hi}-\eta_{lo})} + f(\xi_{lo},\eta_{hi})\frac{(\xi_{hi}-\xi)(\eta-\eta_{lo})}{(\xi_{hi}-\xi_{lo})(\eta_{hi}-\eta_{lo})} + \\ & \quad f(\xi_{hi},\eta_{lo})\frac{(\xi-\xi_{lo})(\eta_{hi}-\eta)}{(\xi_{hi}-\xi_{lo})(\eta_{hi}-\eta_{lo})} + f(\xi_{hi},\eta_{hi})\frac{(\xi-\xi_{lo})(\eta-\eta_{lo})}{(\xi_{hi}-\xi_{lo})(\eta_{hi}-\eta_{lo})} \end{aligned} \quad (14)$$

`BilinearPatch<V,F>` has a single constructor besides the default and copy constructors:

```
BilinearPatch(F xi_lo, F xi_hi, F eta_lo, F eta_hi,
              const V &v_xilo_etalo, const V &v_xihi_etalo,
              const V &v_xihi_etahi, const V &v_xilo_etahi)
```

Make a bi-linear patch between the four corner points given by `v_xilo_etalo`, `v_xihi_etalo`, `v_xihi_etahi` and `v_xilo_etahi`. The values of the parameters at the corners are given by `xi_lo`, `xi_hi`, `eta_lo` and `eta_hi`. For example, when evaluated at `(xi_lo,eta_lo)` the curve will return `v_xilo_etalo`. The curve guarantees that the values at the corner points will be exact (i.e. no round-off error). Note that the order of the value arguments is counterclockwise around the patch.

A `BilinearPatch<V,F>` remains undefined if the default constructor is used. It can be defined later by assignment to another `BilinearPatch<V,F>` or by using the following member function:

```
void define(F xi_lo, F xi_hi, F eta_lo, F eta_hi,
            const V &v_xilo_etalo, const V &v_xihi_etalo,
            const V &v_xihi_etahi, const V &v_xilo_etahi)
```

Sets the bi-linear patch to interpolate between the four corner points given by `v_xilo_etalo`, `v_xihi_etalo`, `v_xihi_etahi` and `v_xilo_etahi`. The values of the parameters at the corners are given by `xi_lo`, `xi_hi`, `eta_lo` and `eta_hi`.

6.9 Linear transformations of parameters

A `LinearParamCurve<N,V,F>` is a curve derived from another curve by applying a linear transformation to its parameters to project them to a different parameter space.

`LinearParamCurve<N,V,F>` has two constructors besides the default and copy constructors:

```
LinearParamCurve(const Curve<N,V,F> &c,
                 const ParamType &old_lo, const ParamType &old_hi,
                 const ParamType &new_lo, const ParamType &new_hi)
```

Makes the curve from curve `c`. The parameters `old_lo` and `old_hi` of `c` are transformed to `new_lo` and `new_hi`: i.e. if the new curve is `crv`, then

`crv(new_lo)` is equivalent to `c(old_lo)` and `crv(new_hi)` is equivalent to `c(old_hi)`.

```
LinearParamCurve(const Curve<N,V,F> &c,
                 F old_lo, F old_hi, F new_lo, F new_hi)
```

Makes the curve from curve `c`. The transformation `old_lo` to `new_lo` and `old_hi` to `new_hi` is applied to each argument of `c`. This constructor is mainly useful when `N` is one.

If the default constructor is used the curve remains undefined. It can be defined later by assignment to another `LinearParamCurve<N,V,F>` or by using the following member functions:

```
void define(const Curve<N,V,F> &c,
            const ParamType &old_lo, const ParamType &old_hi,
            const ParamType &new_lo, const ParamType &new_hi)
```

Defines the curve to be equivalent to `c` with parameters transformed linearly so that `old_lo` and `old_hi` are transformed to `new_lo` and `new_hi`.

```
void define(const Curve<N,V,F> &c,
            F old_lo, F old_hi, F new_lo, F new_hi)
```

Defines the curve to be equivalent to `c` with parameters transformed linearly. The transformation `old_lo` to `new_lo` and `old_hi` to `new_hi` is applied to each argument of `c`.

For example, here is an alternative method for defining the function $f(x) = \sin(2\pi x)$ (see Section 6.2 for another method).

```
using namespace CurveLib;
double twopi = 2.0*3.14159265358979323846;
Curve<1U,double> sin2pix =
  LinearParamCurve<1U,double>(Sin<>(),0.0,twopi,0.0,1.0);
```

`LinearParamCurve<N,V,F>` guarantees that when evaluated at `new_lo` or `new_hi`, the mapping to `old_lo` and `old_hi` is exact: i.e. no round-off errors. This can be very important if the range of definition of a curve is limited.

6.10 Polynomials

A `Polynomial<V,F>` is a single parameter polynomial with value of type `V`. The coefficients of the polynomial have type `F`. It is derived from `Curve<1U,V,F>`. The polynomial is of the form:

$$p(x) = \sum_{n=0}^{N-1} c_n (x - x_0)^n \tag{15}$$

where N is its order.

`Polynomial<V,F>` defines the type `CoefArray` for the array in which the polynomial coefficients are stored. It is a model of a Standard Template Library Random Access Container. In the current implementation it is defined to be `std::vector<V>`.

`Polynomial<V,F>` has the following member functions in addition to the default and copy constructors.

`Polynomial(const CoefArray &c, F x0 = F(0))`
: Constructs a polynominal with coefficients `c` centred around `x0`.

`void define(const CoefArray &coefs, F x0 = F(0))`
: Defines the polynominal by setting its coefficients to `c` and the centre point to `x0`.

`unsigned order() const`
: Returns the order of the polynomial, N.

`unsigned degree() const`
: Returns the degree of the polynomial, $N - 1$.

If the default constructor is used, the polynomial is defined to be the zero polynomial: i.e. it has order one and evaluates to zero.

For example, the following code defines the polynomial $p(x) = 1 - x^2$:

```
using namespace CurveLib;
double c[3] = { 1.0, 0.0, -1.0 };
Polynomial<double>::CoefArray coefs(c,c+3);
Polynomial<double> p(coefs);
```

7 Curves made from vector-valued curves

This section describes curves that take vector-valued curves and modify their values in some way. In this section it will be assumed that the template argument `V` is a model of a Vector Object with respect to the template argument `F`: see Annex A.4. In particular, this means that if `v` is of type `V`, then the subscript operator `v[i]` is defined and returns component `i` of vector `v`; this component has type `F`.

7.1 Selecting a single component

The curve `OneCompCurve<N,V,F>` returns a single component of type `F` from the value of a vector-valued curve; it is a specialization of `Curve<N,F,F>`. It has a single constructor besides the default and copy constructors:

`OneCompCurve(const Curve<N,V,F> &c, unsigned i)`
: Defines a curve whose value is component `i` from the value of curve `c`.

If the default constructor is used, the curve remains undefined.

For example,

```
using namespace CurveLib;
Curve<N,V,F> crv;
// ... define crv
OneCompCurve<N,V,F> crv0(crv,0); // crv0(x) equals crv(x)[0]
```

7.2 Throwing away a vector component

The curve `ReducedDimCurve<N,M,F>` has `N` parameters and returns a vector with `M` components. It is derived from another `N`-parameter vector-valued curve, `c`, having `M+1` components, by throwing away one of the components of its returned value. `ReducedDimCurve<N,M,F>` is derived from `Curve<N,VecMtx::VecN<M,F>,F>` and the curve `c` must be of type `Curve<N,VecMtx::VecN<M+1,F>,F>`.

If the compiler does not support template arithmetic, then an additional template argument must be specified: `MP1` equal to `M+1`. The type of the curve is then `ReducedDimCurve<N,M,MP1,F>`.

The type `CurveType` is defined to be equivalent to the type of the curve `c`. It will be `Curve<N,VecMtx::VecN<M+1,F>,F>` or `Curve<N,VecMtx::VecN<MP1,F>,F>` depending on whether the compiler supports template arithmetic.

`ReducedDimCurve<N,M,F>` has the following constructor in addition to the default and copy constructors.

`ReducedDimCurve(const CurveType &c, unsigned i)`
: Constructs the curve from curve `c` such that the value of the curve will be the same as the value of `c` except that component `i` is removed.

7.3 Concatenating vector values

The curve `ConcatenatedCurve<N,M1,M2,F>` has `N` parameters and returns a vector generated by concatenating the vectors returned by two other curves, `c1` and `c2`, having `M1` and `M2` components respectively. `ConcatenatedCurve<N,M1,M2,F>` is derived from `Curve<N,VecMtx::VecN<M1+M2,F>,F>` and the curves `c1` and `c2` must be of type `Curve<N,VecMtx::VecN<M1,F>,F>` and `Curve<N,VecMtx::VecN<M2,F>,F>` respectively.

If the compiler does not support template arithmetic, then an additional template

argument must be specified: `M1M2` equal to `M1+M2`. The type of the curve is then `ConcatenatedCurve<N,M1,M2,M1M2,F>`.

The type `Curve1Type` is the type of the curve `c1`: `Curve<N,VecMtx::VecN<M1,F>,F>`. Similarly, `Curve2Type` is the type of `c2`: `Curve<N,VecMtx::VecN<M2,F>,F>`.

`ConcatenatedCurve<N,M1,M2,F>` has the following constructor in addition to the default and copy constructors.

`ConcatenatedCurve(const Curve1Type &c1, const Curve2Type &c2)`
: Constructs the curve by concatenating `c1` and `c2`. The first `M1` components of the returned value are the components of the vector returned by `c1`; the components from `M1` to `M1+M2-1` are the components of the vector returned by `c2`.

7.4 Reflection in a plane

A `ReflectedCurve<N,V,F>` negates one of the components of a vector-valued curve. It has a single constructor besides the default and copy constructors:

`ReflectedCurve(const Curve<N,V,F> &c, unsigned i)`
: Constructs a curve equivalent to `c` except that component `i` of the returned value is negated.

If the default constructor is used the curve remains undefined.

For example,

```
using namespace CurveLib;
using namespace VecMtx;
Curve<2U,VecN<3U> > srf;
// ... define srf
ReflectedCurve<2U,VecN<3U> > rsrf(srf,1);
```

The curve `rsrf` is the surface `srf` reflected in the xz plane.

Since a curve argument list is a vector type, `ReflectedCurve<N,V,F>` can be used to change the sign of a single curve parameter. For example, if `crv` represents the function $f(x, y, z)$, then we can represent $f(x, -y, z)$ by the curve `rcrv` defined as follows:

```
using namespace CurveLib;
Curve<3U,V> crv;
// ... define crv
IdentityCurve<3U> xyz;
ReflectedCurve<3U,Curve<3U,V>::ParamType> negy(xyz,1);
Curve<3U,V> rcrv = crv(negy);
```

7.5 Dot products

The curve `DotCurve<N,V,F>` returns the dot-product of two vector valued curves; the returned value is of type `F`. If `f` and `g` are both of type `Curve<N,V,F>` where `V` is a model of a Vector Object, then the value returned by `DotCurve<N,V,F>(f,g)` for parameters `x` can be calculated as follows:

```
V v1 = f(x), v2 = g(x);
F sum(0);
for (int i = 0; i < v1.size(); ++i) {
  sum += v1[i]*v2[i];
}
return sum;
```

If `F` is a real type, then `DotCurve<N,V,F>(f,f)` can be interpreted as the square of the magnitude of the vector curve `f`. However, if `F` is a complex type, this interpretation as a vector magnitude does not hold since that would require the dot product of `f` with the complex conjugate of `f`.

`DotCurve<N,V,F>` has the following constructor in addition to the default and copy constructors:

`DotCurve(const Curve<N,V,F> &c1, const Curve<N,V,F> &c2)`
: Makes a curve whose value is the dot product of the values of the curves `c1` and `c2`.

`DotCurve(const V &v, const Curve<N,V,F> &c)`
: Makes a curve which returns the dot product of `v` and `c`.

`DotCurve(const Curve<N,V,F> &c, const V &v)`
: Makes a curve which returns the dot product of `c` and `v`.

If the default constructor is used the curve remains undefined.

Section 6.2 contains an example of the use of `DotCurve<N,V,F>`.

7.6 Cross products

The curve `CrossProdCurve<N,F>` returns the cross-product of two curves whose values are 3-vectors of type `VecMtx::VecN<3U,F>`. `CrossProdCurve<N,F>` also returns a `VecMtx::VecN<3U,F>`. The type `CrossProdCurve<N,F>::CurveType` is defined to be equivalent to the type of each curve: `Curve<N,ValueType,F>`.

`CrossProdCurve<N,F>` has the following constructor in addition to the default and copy constructors:

`typedef VecMtx::VecN<3U,F> ValueType`
: The type of the value of the curve.

`typedef Curve<N,ValueType,F> CurveType`
: The type of the curves whose cross-product is to be taken.

`CrossProdCurve(const CurveType &c1, const CurveType &c2)`
: Makes a curve whose value is the cross product of the values of the curves `c1` and `c2`.

`CrossProdCurve(const ValueType &v, const CurveType &c)`
: Makes a curve which returns the cross product of `v` and `c`.

`CrossProdCurve(const CurveType &c, const ValueType &v)`
: Makes a curve which returns the cross product of `c` and `v`.

If the default constructor is used the curve remains undefined.

7.7 Unit vectors

Suppose that `f` is a curve whose return value is a model of a Vector Object. Then the curve `UnitCurve<N,V,F>(f)` is equivalent to `f/Sqrt<F>(DotCurve<N,V,F>(f,f))`: i.e. its value is the value of `f` normalized using the square root of the dot product of the value of `f` with itself. If `F` is a real type, this curve can be interpreted as `f` converted to a unit vector (hence the name) since `DotCurve<N,V,F>(f,f)` can be interpreted as the square of the magnitude of the vector curve `f`. When `F` is a model of a complex number, `UnitCurve<N,V,F>` is well-defined but its interpretation as a unit vector is not valid, since `DotCurve<N,V,F>(f,f)` no longer represents the magnitude of the `f`: see Section 7.5.

`UnitCurve<N,V,F>` has the following constructor in addition to the default and copy constructors:

`UnitCurve(const Curve<N,V,F> &c)`
: Makes a curve whose value is the value of curve `c` converted to a unit vector.

If the default constructor is used the curve remains undefined.

7.8 Projections

The curves described in this section are each derived by projecting a curve onto a surface (or a hypersurface if the dimension of the returned value of the original curve is greater than three).

7.8.1 Projection to a plane

The curve `PlaneProjCurve<N,V,F>` is derived from another vector-valued curve by setting one of the components of the returned value to a constant value.

`PlaneProjCurve<N,V,F>` has the following constructor in addition to the default and copy constructors:

`PlaneProjCurve(const Curve<N,V,F> &c, unsigned i, F v)`
: Makes a curve whose value is the value of curve `c` except that component `i` is set to `v`.

If the default constructor is used the curve remains undefined.

For example,

```
using namespace CurveLib;
typedef VecMtx::VecN<3U,F> Vec3;
Curve<1U,Vec3,F> crv;
// ... define crv
PlaneProjCurve<1U,Vec3,F> proj_crv(crv,0,0.0);
```

Here `crv` is a curve in three-dimensional space. The curve `proj_crv` is the projection of `crv` onto the yz plane.

`PlaneProjCurve<N,V,F>` can only project a curve onto a plane aligned with one of the coordinate axes; however, it will work in any number of dimensions (i.e. the length of the vector value is arbitrary). `AnyPlaneProjCurve<N,F>` will project a curve onto any arbitrary plane, but it is restricted to 3-vectors. The type of the returned value, `ValueType`, is `VecMtx::VecN<3U,F>`. The type `CurveType` is defined to be equivalent to the type of projected curve: `Curve<N,ValueType,F>`.

`AnyPlaneProjCurve(const CurveType &c, const ValueType &p,`
`const ValueType &n)`
: Makes a curve whose value is the value of curve `c` projected onto the plane defined by the point `p` and normal `n`. The normal, `n`, must not be the zero-vector.

`AnyPlaneProjCurve(const CurveType &c, const ValueType &p1,`
`const ValueType &p2, const ValueType &p3)`
: Makes a curve whose value is the value of curve `c` projected onto the plane passing through the three points `p1`, `p2` and `p3`. These points must not be collinear.

7.8.2 Projection to a sphere

`SphereProjCurve<N,F>` is a curve derived from a vector-valued curve by projecting its values onto a sphere along radial lines: i.e. if the original curve is $\mathbf{f}(x)$, then the projected curve is

$$R\frac{\mathbf{f}(x)}{|\mathbf{f}(x)|} \tag{16}$$

where R is the radius of the sphere. The type of the returned value, `ValueType`, is `VecMtx::VecN<3U,F>`. The type `CurveType` is defined to be equivalent to the type of the projected curve: `Curve<N,ValueType,F>`.

`SphereProjCurve(const CurveType &c, F r)`
: Makes a curve whose value is the value of curve `c` projected onto the sphere of radius `r` centred at the origin.

If the default constructor is used the curve remains undefined.

The class `UnitCurve<N,V,F>` (see Section 7.7) could also be interpreted as a projection onto the unit hypersphere.

7.8.3 Projection to a cylinder

`CylProjCurve<N,F>` is a curve derived from a vector-valued curve by projecting its values onto a cylinder: i.e. if the original curve is $\mathbf{f}(x)$, then the projected curve is

$$\mathbf{p} + \mathbf{x}_a + \frac{r(\mathbf{x}-\mathbf{p}-\mathbf{x}_a)}{|\mathbf{x}-\mathbf{p}-\mathbf{x}_a|}; \qquad \mathbf{x}_a \equiv \big[(\mathbf{x}-\mathbf{p})\cdot\hat{n}\big]\hat{n} \tag{17}$$

where R is the radius of the cylinder and its axis has direction $\hat{n}$ and passes through the point $\mathbf{p}$. The type of the returned value, `ValueType`, is `VecMtx::VecN<3U,F>`. The type `CurveType` is defined to be equivalent to the type of the projected curve: `Curve<N,ValueType,F>`.

`CylProjCurve(const CurveType &c, F r, unsigned i)`
: Makes a curve whose value is the value of curve `c` projected onto the cylinder of radius `r` whose axis passes through the origin and is aligned with the i^{th} coordinate axis.

```
CylProjCurve(const CurveType &c, F r, const ValueType &p,
             const ValueType &n)
```
: Makes a curve whose value is the value of curve `c` projected onto the cylinder of radius `r` whose axis passes through the point `p` and is parallel to `n`.

If the default constructor is used the curve remains undefined.

8 Surfaces

Curves of type `Curve<2U,VecMtx::VecN<3U,F>,F>` are of particular importance as they can be used to represent surfaces in three-dimensional space. Therefore they are useful for describing the geometry of physical objects.

We will use the convention that the two parameters of a surface are ξ and η, represented in code by `xi` and `eta`.

Because of the importance of surfaces, `Curve<2U,VecMtx::VecN<3U,F>,F>` is treated as a special case and has extra member functions to ease its use (here `ValueType` is an alias for `VecMtx::VecN<3U,F>`):

`ValueType operator()(F xi, F eta) const`
: Returns the value of the curve at `(xi,eta)`. This is often more convenient than passing a `ParamType`.

`ValueType operator()(F xi, F eta,`
`unsigned dxi, unsigned deta) const`
: Returns the value of the differentiated curve at `(xi,eta)`. The number of derivatives to be taken with respect to each parameter is specified by `dxi` and `deta`.

`ValueType normal(const ParamType &p) const`
: Returns a unit normal to the surface at `p`.

`ValueType normal(F xi, F eta) const`
: Returns a unit normal to the surface at `(xi,eta)`.

`ValueType normal(const ParamType &p, const DerivType &d) const`
: Returns the derivatives of the normal to the surface at `p`. The number of derivatives to be taken with respect to each parameter is specified by `d`.

`ValueType normal(F xi, F eta, unsigned dxi, unsigned deta) const`
: Returns the value of the derivatives of the normal at `(xi,eta)`. The number of derivatives to be taken with respect to each parameter are specified by `dxi` and `deta`.

`void get_metrics(const ParamType &p, VecMtx::MtxN<2U,F> &g,`
`VecMtx::MtxN<2U,F> (&gamma)[2]) const`
: Returns the metric tensor and Christoffel symbols of the first kind at `p`.

  The 2×2 matrix `g` contains the covariant metric tensor (the matrix class

`VecMtx::MtxN<N,F>` is described in Annex C). If the surface is $\mathbf{X}(\xi,\eta)$ then

$$g_{00} = \frac{\partial \mathbf{X}}{\partial \xi}\cdot\frac{\partial \mathbf{X}}{\partial \xi}; \qquad g_{01} = g_{10} = \frac{\partial \mathbf{X}}{\partial \xi}\cdot\frac{\partial \mathbf{X}}{\partial \eta}; \qquad g_{11} = \frac{\partial \mathbf{X}}{\partial \eta}\cdot\frac{\partial \mathbf{X}}{\partial \eta} \tag{18}$$

The $2\times 2\times 2$ tensor `gamma` contains the Christoffel symbols:

$$\begin{aligned}
\Gamma_{000} &= \frac{\partial^2 \mathbf{X}}{\partial \xi^2}\cdot\frac{\partial \mathbf{X}}{\partial \xi}; \quad \Gamma_{010} = \Gamma_{100} = \frac{\partial^2 \mathbf{X}}{\partial \xi\partial \eta}\cdot\frac{\partial \mathbf{X}}{\partial \xi}; \quad \Gamma_{110} = \frac{\partial^2 \mathbf{X}}{\partial \eta^2}\cdot\frac{\partial \mathbf{X}}{\partial \xi} \\
\Gamma_{001} &= \frac{\partial^2 \mathbf{X}}{\partial \xi^2}\cdot\frac{\partial \mathbf{X}}{\partial \eta}; \quad \Gamma_{011} = \Gamma_{101} = \frac{\partial^2 \mathbf{X}}{\partial \xi\partial \eta}\cdot\frac{\partial \mathbf{X}}{\partial \eta}; \quad \Gamma_{111} = \frac{\partial^2 \mathbf{X}}{\partial \eta^2}\cdot\frac{\partial \mathbf{X}}{\partial \eta}
\end{aligned} \tag{19}$$

If the compiler allows partial instantiation of templates, these extra member functions will be defined for any type `F`. If not, they will be defined only when `F` is `double` or `float`.

The overloaded `normal` functions evaluate the unit normal to surface $\mathbf{X}(\xi,\eta)$ as follows:

$$\mathbf{n} = \frac{\partial \mathbf{X}}{\partial \xi}\times\frac{\partial \mathbf{X}}{\partial \eta}; \qquad \hat{\mathbf{n}} = \frac{\mathbf{n}}{|\mathbf{n}|} \tag{20}$$

However, this definition fails if the derivative of $\mathbf{X}$ with respect to ξ or η is zero. In this case a `NormalNotDefined` exception is thrown. Its error message is:

```
Normal not defined.
```

For example, the surface of a sphere can be defined as follows:

```
using namespace CurveLib;
OneParamCurve<2U> xi(0), eta(1);
Cos<> coscrv;
Sin<> sincrv;
MultiCurve<2U,3U> sphere;
sphere[0] =  sincrv(xi)*sincrv(eta);
sphere[1] =  sincrv(xi)*coscrv(eta);
sphere[2] = -coscrv(xi);
```

for ξ in $[0,\pi]$ and η in $[0,2\pi]$ (the signs have been chosen to make the normal to the sphere outward pointing). Since `MultiCurve<2U,3U>` is a specialization of `Curve<2U,VecMtx::VecN<3U> >`, the extra member functions are defined. The code

```
try {
  double pio2 = 0.5*3.14159265358979323846;
  std::cout << "Point  = " << sphere(pio2,0) << '\n';
  std::cout << "Normal = " << sphere.normal(pio2,0) << '\n';

  std::cout << "Point  = " << sphere(0,0) << '\n';
  std::cout << "Normal = " << sphere.normal(0,0) << '\n';
```

```
}
catch(NormalNotDefined &n) {
  std::cerr << n.get_msg() << '\n';
}
```

generates the following output

```
Point  =  0.000000e+00  1.000000e+00 -6.123234e-17
Normal = -0.000000e+00  1.000000e+00 -6.123234e-17
Point  =  0.000000e+00  0.000000e+00 -1.000000e+00
Normal = Normal not defined.
```

When evaluated at $(0, 0)$, a `NormalNotDefined` exception was thrown because the derivative with respect to η vanishes there. As will be seen in the following section, classes derived from `Curve<2U,VecMtx::VecN<3U,F>,F>` may avoid undefined normals even when one of the derivatives vanishes.

8.1 Axi-symmetric surfaces

Suppose

$$g(\xi) = \begin{pmatrix} g_z(\xi) \\ g_r(\xi) \end{pmatrix} \tag{21}$$

defines a one parameter curve which returns a two-vector and suppose that $g_r(\xi) \geq 0$. Then, a surface of revolution about the z axis is defined by.

$$f(\xi, \theta) = \begin{pmatrix} g_r(\xi) \sin\theta \\ g_r(\xi) \cos\theta \\ g_z(\xi) \end{pmatrix} \tag{22}$$

From Equations (20) and (22), the unit normal to the axisymmetric surface is:

$$\hat{\mathbf{n}} = \frac{g_z'(\xi)(\hat{x}\sin\theta + \hat{y}\cos\theta) - g_z'(\xi)\hat{z}}{\sqrt{\left(g_z'(\xi)\right)^2 + \left(g_r'(\xi)\right)^2}} \tag{23}$$

The unit normal is well-defined unless *both* $g_z'(\xi)$ and $g_r'(\xi)$ vanish. Note that if we had not required $g_r(\xi)$ to be non-negative, then the normal would not be well-defined when $g_r(\xi) = 0$ as the direction of the normal would flip as the value of $g_r(\xi)$ changed from positive to negative.

The class `AxisymmetricSurface<F>` represents the curve $f(\xi, \eta)$ but generalizes it so that the spine curve, $g(\xi)$, can be rotated about any one of the three coordinate axes.

`AxisymmetricSurface<F>` defines the following aliases:

`ValueType`
: Equivalent to `VecMtx::VecN<3U,F>`.

`SpineCurveType`
: Equivalent to `Curve<1U,VecMtx::VecN<2U,F>,F>`.

It has the following member functions.

`AxisymmetricSurface(unsigned a, const SpineCurveType &g)`
: A constructor makes a surface of revolution by rotating the spine curve `g` around the axis specified by `a`: 0 is the x axis, 1 the y axis, and 2 the z axis. If `a` is not 0, 1 or 2 an `AxisOutOfBounds` exception will be thrown.

`unsigned axis() const`
: Returns the axis of rotation as 0, 1 or 2.

`ValueType value(F xi, Angle<F> theta) const`
: Calculates a point on the surface for parameters (ξ, θ). The class `Angle<F>` is described in Annex E.

```
ValueType value(F xi, Angle<F> theta,
                unsigned dxi, unsigned dtheta) const
```
: Calculates a point on the surface, or its derivatives, for parameters (ξ, θ). The number of derivatives to be taken with respect to ξ is `dxi` and with respect to θ is `dtheta`.

`VecMtx::VecN<2U,F> spine_value(F xi) const`
: Return the value of the spine curve at `xi`.

`VecMtx::VecN<2U,F> spine_value(F xi, unsigned d) const`
: Returns derivatives of the spine curve at `xi`. The number of derivatives to be taken is specified by `d`.

`SpineCurveType spine_curve() const`
: Returns the spine curve.

If the value of $g_r(\xi)$ is zero (i.e. the axisymmetric surface is closed at at least one end), then the θ derivatives of the surface vanish. However, the normals of the surface are still well-defined; the `AxisymmetricSurface<F>` `normal` functions will return the correct unit normals in this case; they will not throw a `NormalNotDefined` exception.

For example, here is another definition of the surface of a sphere, with the same parameterization as the sphere defined in Section 8. However, this time the normal is defined everywhere:

```
using namespace CurveLib;
MultiCurve<1U,2U> circle;
circle[0] = -Cos<>();
circle[1] =  Sin<>();
AxisymmetricSurface<> sphere(2,circle);
```

Now the code

```
try {
  double pio2 = 0.5*3.14159265358979323846;
  std::cout << "Point  = " << sphere(pio2,0) << '\n';
  std::cout << "Normal = " << sphere.normal(pio2,0) << '\n';
  std::cout << "Point  = " << sphere(0,0) << '\n';
  std::cout << "Normal = " << sphere.normal(0,0) << '\n';
}
catch(NormalNotDefined &n) {
  std::cerr << n.get_msg() << '\n';
}
```

generates the following output

```
Point  =  0.000000e+00  1.000000e+00 -6.123234e-17
Normal =  0.000000e+00  1.000000e+00 -6.123234e-17
Point  =  0.000000e+00  0.000000e+00 -1.000000e+00
Normal =  0.000000e+00  0.000000e+00 -1.000000e+00
```

Notice that the normal is correctly evaluated at $(0, 0)$ whereas the surface defined in Section 8 threw a `NormalNotDefined` exception.

The class `AxisymmetricSurface<F>` is well-defined even if the radial component of the spine curve, $g_r(\xi)$, returns negative values. However, when $g_r(\xi) < 0$, the `normal` member functions will return normals which are opposite in sign from the normals returned by `Curve<2U,VecMtx::VecN<3U,F>,F>`.

9 Interpolation between boundary curves

This section describes classes for interpolating between two or more boundary curves.

9.1 Ruled curves

A ruled curve is a curve created by linear interpolation between two boundary curves with one fewer parameters. For example, suppose the boundary curves are $f_{lo}(x_1, \ldots, x_{N-1})$ and $f_{hi}(x_1, \ldots, x_{N-1})$. We introduce a new parameter, y, which governs the interpolation between the two boundary curves. When $y = y_{lo}$, the value

of the ruled curve is $f_{lo}(x_1, \ldots, x_{N-1})$; when $y = y_{hi}$, the value of the ruled curve is $f_{hi}(x_1, \ldots, x_{N-1})$. The parameter list for the ruled curve is the equal to $x_1, \ldots, x_{N-1}$ with y inserted at location k. Therefore

$$f(x_1, \ldots, x_{k-1}, y, x_k, \ldots, x_{N-1}) = \tag{24}$$

$$\frac{(y - y_{lo}) f_{hi}(x_1, \ldots, x_{N-1}) + (y_{hi} - y) f_{lo}(x_1, \ldots, x_{N-1})}{y_{hi} - y_{lo}} \tag{25}$$

Ruled curves are represented by the class `RuledCurve` which has two possible template argument lists, depending on the capabilities of the compiler. If template arithmetic is allowed, `RuledCurve` has template arguments `<N,V,F>` where, as usual, `N` is the number of parameters, `V` is the return type and `F` is the type of each parameter. If template arithmetic is not allowed, the template arguments are `<N,NM1,V,F>` where the unsigned value `NM1` is equal to `N-1`.

The type `BdyCurveType` is defined to be the same as the type of each boundary curve: `Curve<N-1,V,F>` if template arithmetic is allowed, `Curve<NM1,V,F>` if it is not.

Ruled curves have the following member functions in addition to the default and copy constructors:

```
RuledCurve(unsigned k,
           const BdyCurveType &bclo, const BdyCurveType &bchi,
           F ylo, F yhi)
```

Makes a ruled curve between the boundary curves `bclo` and `bchi`. The ruled curve has `N` parameters; parameter `k` interpolates between the boundary curves. When parameter `k` has the value `ylo`, the value of the curve is equal to the value of `bclo`; when parameter `k` has the value `yhi`, the value of the curve is equal to the value of `bchi`.

```
BdyCurveType boundary_curve(bool upper) const
```

Returns one of the boundary curves. If `upper` is true, the curve `bchi` is returned; otherwise `bclo` is returned.

If the default constructor is used the curve remains undefined.

9.2 Transfinite interpolation

A `TransFinite2dCurve<V,F>` is two parameter curve produced by transfinite interpolation from values on the boundaries. Let $f(\xi, \eta)$ be the curve we wish to construct. Let P_ξ and P_η be the operators

$$P_\xi(f)(\xi, \eta) \quad = \quad \frac{(\xi - \xi_{lo}) f(\xi_{hi}, \eta) + (\xi_{hi} - \xi) f(\xi_{lo}, \eta)}{\xi_{hi} - \xi_{lo}} \tag{26}$$

$$P_\eta(f)(\xi,\eta) = \frac{(\eta-\eta_{lo})f(\xi,\eta_{hi})+(\eta_{hi}-\eta)f(\xi,\eta_{lo})}{\eta_{hi}-\eta_{lo}} \tag{27}$$

Then:

$$\begin{aligned} f(\xi,\eta) &= \big[1-(1-P_\xi)(1-P_\eta)\big]f(\xi,\eta) \\ &= P_\xi(f)(\xi,\eta)+P_\eta(f)(\xi,\eta)-P_\xi(P_\eta(f))(\xi,\eta) \end{aligned} \tag{28}$$

The first term is a ruled curve between the boundary curves at $\xi=\xi_{lo}$ and $\xi=\xi_{hi}$. The second term is a ruled curve between the boundary curves at $\eta=\eta_{lo}$ and $\eta=\eta_{hi}$. The last term is a bilinear patch between the four corner points.

The type `BdyCurveType` is defined to be equivalent to the type of each boundary curve: `Curve<1U,V,F>`.

`TransFinite2dCurve<V,F>` has the following member functions in addition to the default and copy constructors:

```
TransFinite2dCurve(F xi_lo, F xi_hi, F eta_lo, F eta_hi,
                   const BdyCurveType xi_bndcrvs[2],
                   const BdyCurveType eta_bndcrvs[2])
```

Uses transfinite interpolation to make a curve between the boundary curves in `xi_bndcrvs` and `eta_bndcrvs`. The curves `xi_bndcrvs[0]` and `xi_bndcrvs[1]` are the curves for which ξ is `xi_lo` and `xi_hi` respectively. Similarly, the argument `eta_bndcrvs` defines the curves for the edges for which η is `eta_lo` and `eta_hi`.

```
BdyCurveType boundary_curve(unsigned dir, bool upper) const
```

Returns one of the boundary curves. The argument `dir` gives the direction of the curve: 0 for a boundary with constant ξ, 1 for a boundary with constant η. The argument `upper` indicates whether it is the upper or lower boundary.

If the default constructor is used the curve is undefined.

A `TransFinite3dCurve<V,F>` is a three parameter curve which uses transfinite interpolation in three dimensions. It is a natural extension of its two-dimensional counterpart, `TransFinite2dCurve<V,F>`. The boundary curves each have two parameters. Let $f(\xi,\eta,\zeta)$ be the curve we wish to construct. The curve is defined by

$$\begin{aligned} f(\xi,\eta,\zeta) &= \big[(P_\xi-1)(P_\eta-1)(P_\zeta-1)+1\big]f \\ &= \big[P_\xi+P_\eta+P_\zeta-P_\xi P_\eta-P_\xi P_\zeta-P_\eta P_\zeta+P_\xi P_\eta P_\zeta\big]f \end{aligned} \tag{29}$$

where P_ξ, P_η and P_ζ are as defined in Equations (26) and (27).

The type `BdyCurveType` is defined to be equivalent to the type of each boundary curve: `Curve<2U,V,F>`.

TransFinite3dCurve<V,F> has the following member functions in addition to the default and copy constructors:

```
TransFinite3dCurve(F xi_lo, F xi_hi,
                   F eta_lo, F eta_hi,
                   F zeta_lo, F zeta_hi,
                   const BdyCurveType xi_bndcrvs[2],
                   const BdyCurveType eta_bndcrvs[2],
                   const BdyCurveType zeta_bndcrvs[2])
```

Uses transfinite interpolation to make a curve between the boundary curves in xi_bndcrvs, eta_bndcrvs and zeta_bndcrvs. The curves xi_bndcrvs[0] and xi_bndcrvs[1] are the curves for which ξ is xi_lo and xi_hi respectively. Similarly, the argument eta_bndcrvs defines the curves for the edges for which η is eta_lo and eta_hi, and zeta_bndcrvs defines the curves for the edges for which ζ is zeta_lo and zeta_hi.

Each of the curves in the array xi_bndcrvs has arguments (η, ζ); each of the curves in eta_bndcrvs has arguments (ξ, ζ); and each of the curves in zeta_bndcrvs has arguments (ξ, η).

```
BdyCurveType boundary_curve(unsigned dir, bool upper) const
```

Returns one of the boundary curves. The argument dir gives the direction of the curve: 0 for a boundary with constant ξ, 1 for a boundary with constant η, 2 for a boundary with constant η. The argument upper indicates whether it is the upper or lower boundary.

If the default constructor is used the curve is undefined.

The transfinite curves are well-defined only if the boundary curves give similar values in the corners of their domains: for example, for a TransFinite2dCurve<V,F>, xi_bndcrv[0](xi_lo) and eta_bndcrv[0](eta_lo) should be the same. In practice, due to round-off, the best that one can expect is that these two values differ by a small amount. It would be convenient if TransFinite2dCurve<V,F> and TransFinite3dCurve<V,F> had member functions that would return the largest mismatch in the values of the boundary curves at the corners of the domain. However, to do so would require that the type V have some method of comparing two values and reducing the difference to an F. Moreover, to find the largest mismatch would require that F be a model of Less Than Comparable (defined by the Standard Template Library). Both these requirements would limit the possible instantiations of TransFinite2dCurve<V,F> and TransFinite2dCurve<V,F>.

To avoid these problems the function trans_finite_curve_mismatch is provided. It has a two-dimensional version and a three dimensional version which have arguments similar to the TransFinite2dCurve<V,F> and TransFinite2dCurve<V,F> constructors:

```
F trans_finite_curve_mismatch(
          F xi_lo, F xi_hi, F eta_lo, F eta_hi,
          const Curve<1U,V,F> xi_bndcrvs[2],
          const Curve<1U,V,F> eta_bndcrvs[2])
```
Returns the maximum mismatch of the points on each boundary curve at the corners of the domain defined by `xi_lo`, `xi_hi`, `eta_lo` and `eta_hi`.

```
F trans_finite_curve_mismatch(
          F xi_lo, F xi_hi, F eta_lo, F eta_hi, F zeta_lo, F zeta_hi,
          const Curve<2U,V,F> xi_bndcrvs[2],
          const Curve<2U,V,F> eta_bndcrvs[2],
          const Curve<2U,V,F> zeta_bndcrvs[2])
```
Returns the maximum mismatch of the points on each boundary curve at the corners of the domain defined by `xi_lo`, `xi_hi`, `eta_lo`, `eta_hi`, `zeta_lo`, and `zeta_hi`.

These functions should only be used if `V` is a model of an Absolute Object with respect to `F` (see Annex A.5) and if `F` is a Comparable Scalar Object (see Annex A.3).

10 Curves with parameter ranges

It is often convenient if a curve knows the range of parameters for which its evaluation is valid. For example, a curve representing $f(x) = \arcsin(x)$ might recognize that its parameters must lie in the range $[0, 1]$. Such curves are called *range curves* and are represented by the base class `RangeCurve<N,V,F>` where `F` is a model of a Comparable Scalar Object (see Annex A.3) and `V` is a vector object with respect to `F`.

10.1 Parameter ranges

Before defining range curves it is necessary to a have a means to represent the ranges of the parameters of a curve. The class `ParamRange<N,F>` is used for this purpose. Its template argument `N` is an `unsigned int` giving the number of parameters; the template argument `F` is the type of each parameter. If `F` is missing it defaults to `double`.

The type `ParamRange<N,F>::ParamType` is defined to be the type of the parameter list of a curve: i.e. it is equivalent to `Curve<N,V,F>::ParamType` for any `V`. In the current implementation it is a `VecMtx::VecN<N,F>`.

`ParamRange<N,F>` has the following member functions:

```
ParamRange(F xlo = F(0), F xhi = F(1))
```
Makes a `ParamRange<N,F>` in which each parameter has the range `xlo` to `xhi`.

`ParamRange(const ParamType &lo, const ParamType &hi)`
Makes a `ParamRange<N,F>` in which parameter `i` has range `lo[i]` to `hi[i]`.

`const ParamType& low() const`
Return the lower limits of the parameters.

`F low(unsigned i) const`
Returns the lower limit for parameter `i`.

`const ParamType& high() const`
Returns the upper limits of the parameters.

`F high(unsigned i) const`
Returns the upper limit for parameter `i`.

`void set_range(const ParamType &lo, const ParamType &hi)`
Sets the range so that parameter `i` has range `lo[i]` to `hi[i]`.

`void set_range(unsigned i, F lo, F hi)`
Sets the range of parameter `i` to be `lo` to `hi`.

`virtual bool in_range(const ParamType &p) const`
Returns true if `p` is in range.

`unsigned size() const`
Returns the number of parameters.

Two parameter ranges can be compared to see if they are the same using the `==` and `!=` operators. The following function is also defined:

```
template<unsigned N, class F>
ParamRange<N,F> intersection(const ParamRange<N,F> &r1,
                             const ParamRange<N,F> &r2)
```
Returns a range which is the intersection of `r1` and `r2`.

10.2 Range curves

A `RangeCurve<N,V,F>` is a curve whose parameters have set ranges. It contains a `Curve<N,V,F>` and a `ParamRange<N,F>`. The following constructors are defined in addition to the default and copy constructors:

`RangeCurve(const Curve<N,V,F> &c)`
Makes a `RangeCurve<N,V,F>` from `c`. The range for each parameter is the

default: $[0, 1]$.

`RangeCurve(const RangeCurve<N,V,F> &c, unsigned i,`
`FloatType plo, FloatType phi)`
Copies `c` but changes the range of parameter `i` to `[plo,phi]`.

`RangeCurve(const Curve<N,V,F> &c,`
`const ParamType &plo, const ParamType &phi)`
Gives curve `c` the range defined by `plo` and `phi`: i.e. the range of parameter `i` is `[plo[i],phi[i]]`.

`RangeCurve(const Curve<N,V,F> &c, const RangeType &r)`
Gives the curve `c` the range `r`.

The following member functions can be used to change the range of an existing range curve:

`void set_range(const ParamType &lo, const ParamType &hi)`
Sets the range to have minimum values in `lo` and maximum values in `hi`: i.e. the new range of parameter `i` will be `[lo[i],hi[i]]`.

`void set_range(unsigned i, F lo, F hi)`
Set the range of parameter `i` to be `[lo,hi]`.

The following member functions can be used to determine what the range of a curve is:

`const RangeType& get_range() const`
Returns the range of the curve.

`const ParamType& low_range() const`
Returns the minimum values of the range of each of the parameters.

`const ParamType& high_range() const`
Returns the maximum values of the range of each of the parameters.

`F low_range(unsigned i) const`
Returns the minimum allowed value of parameter `i`.

`F high_range(unsigned i) const`
Returns the maximum allowed value of parameter `i`.

The following two functions evaluate the curve at the low and high ends of its parameter range. It is of most use for one parameter range curves.

`V low_value() const`
: Returns the value at the low end of the parameter range.

`V high_value() const`
: Returns the value at the high end of the parameter range.

And finally, a function to indicate whether a parameter is in the range of the curve.

`bool in_range(const ParamType &p) const`
: Returns true if `p` is in the range of the curve.

10.3 Standard range curves

A `StandardRangeCurve<N,V,F>` is a range curve whose parameters all have the default range, $[0, 1]$. It inherits the `RangeCurve<N,V,F>` member functions, but has the following constructor:

`StandardRangeCurve(const RangeCurve<N,V,F> &c)`
: Converts the curve `c` so that its parameters have the standard range. This is done by applying a linear transformation to each parameter so that the parameter range of `c` is mapped to $[0, 1]$.

11 Implicitly defined curves

This section describes curves whose values are defined implicitly as the zeros of other *defining curves*. First we describe the primary search algorithm used by these functions.

11.1 Newton-Raphson search

Suppose $\mathbf{f}(\mathbf{x})$ is an N-valued function of N parameters, $\mathbf{x} = (x_1, \ldots, x_N)$. Then we can search for the parameters $\mathbf{x}$ for which $\mathbf{f}(\mathbf{x}) = 0$. Let $\boldsymbol{\xi}$ be an approximation to $\mathbf{x}$. A better approximation to $\mathbf{x}$ is:

$$x_n = \xi_n - \sum_{i=1}^{N} \mathbf{A}^{-1}_{ni} f_i(\boldsymbol{\xi}) \quad \text{for } n = 1, \ldots, N \tag{30}$$

with

$$\mathbf{A}_{ni} = \frac{\partial f_n}{\partial x_i}(\boldsymbol{\xi}) \tag{31}$$

This algorithm requires an initial guess at the value of $\mathbf{x}$.

Often there will be more than one possible solution. To ensure that one obtains the correct solution, the solution vector can be bounded: i.e. two vectors, **a** and **b** are provided such that $a_n \leq x_n \leq b_n$ for all n. The search algorithm is then modified to ensure that the solution lies in this range:

Initialization:
for all n, if $x_n < a_n$ or $x_n > b_n$ then
$x_n \longleftarrow (a_n + b_n)/2$
Search:
while $|f_n(x_m)| > \epsilon$
$\delta_n \longleftarrow \sum_{i=1}^{N} \mathbf{A}^{-1}_{ni} f_i(\xi_m)$
for all n
if $x_n - \delta_n < a_n$ then $x_n \longleftarrow (x_n + a_n)/2$
else if $x_n - \delta_n > b_n$ then $x_n \longleftarrow (x_n + b_n)/2$
else $x_n \longleftarrow x_n - \delta_n$

If a Newton-Raphson search fails to converge, a `FailingSearch` exception is thrown. It is derived from the base error class `Error` and has the following generic error message:

```
Failure in Newton-Raphson search when evaluating an implicit curve.
```

11.2 Implicit curves

Let $f_i(x_m, y_n)$ be a function of $M + N$ variables for $i = 1, \ldots, M$, $m = 1, \ldots, M$ and $n = 1, \ldots, N$. Let $g_m(y_n)$ be the value of x_m for which the function is zero:

$$f_i\big(g_m(y_n), y_n\big) = 0 \tag{32}$$

We say that the function $g_m(y_n)$ is defined implicitly by the function $f_i(x_m, y_n)$. The function $f_i(x_m, y_n)$ is called the *defining curve* for the *implicit curve* $g_m(y_n)$.

Given the N values y_n, the value of $g_m(y_n)$ can be found by a Newton-Raphson search of the M equations in Equation (32) for the M unknowns $g_m(y_n)$.

Differentiating Equation (32) by y_n one gets

$$\sum_{m=1}^{M} \frac{\partial f_i}{\partial x_m} \frac{dg_m}{dy_n} + \frac{\partial f_i}{\partial y_n} = 0 \tag{33}$$

which can be written

$$\sum_{m=1}^{M} \mathbf{A}_{im} \frac{dg_m}{dy_n} = -\frac{\partial f_i}{\partial y_n} \tag{34}$$

where

$$\mathbf{A}_{im} = \frac{\partial f_i}{\partial x_m}\big(g_m(y_n), y_n\big) \tag{35}$$

Therefore

$$\frac{dg_m}{dy_n} = -\sum_{i=1}^{M} \mathbf{A}^{-1}_{mi} \frac{\partial f_i}{\partial y_n}\big(g_m(y_n), y_n\big) \tag{36}$$

Implicit curves are represented in `CurveLib` by the class `ImplicitCurve` defined in the header file `ImplicitCurve.h`. The type of its returned value must be the same as that of a parameter list of an `M`-parameter curve. In the current implementation this is a `VecMtx::VecN<M,F>`.

If template arithmetic is allowed by the compiler, `ImplicitCurve` is defined as follows:

```
namespace CurveLib {
  template<unsigned N, unsigned M, class F = double>
  class ImplicitCurve: public Curve<N,VecMtx::VecN<M,F>,F>
  {
  public:
    typedef VecMtx::VecN<M,F> ValueType;
                // The type of the return value.

    typedef Curve<N+M,ValueType,F> DCurveType;
                // The type of the defining curve.
    ...
  };
}
```

Template argument `N` is the number of parameters in the implicit curve. The return value is a vector of length `M` and the defining curve has `N+M` arguments.

If template arithmetic is not allowed, the following definition is used:

```
namespace CurveLib {
  template<unsigned N, unsigned M, unsigned NPM,
           class F = double>
  class ImplicitCurve: public Curve<N,VecMtx::VecN<M,F>,F>
  {
  public:
    typedef VecMtx::VecN<M,F> ValueType;
                // The type of the return value.

    typedef Curve<NPM,ValueType,F> DCurveType;
                // The type of the defining curve.
    ...
  };
}
```

In this case the template argument `NPM` must equal `N+M`.

The evaluation of an `ImplicitCurve` executes a Newton-Raphson search to find the zeros of the defining curve. Parameters required to determine convergence of the iteration are specified using constructor arguments:

```
ImplicitCurve(const DCurveType &c, F acc = F(1)/F(1000000),
              unsigned itmax = 30, F frac = F(0))
```

Constructs the curve using `c` as the defining curve. The `N` parameters of the implicit curve are the last `N` parameters of `c`. Given values for these `N` parameters, the values of the implicit curve are such that when used as the first `M` parameters of `c`, `c` will return zero.

The Newton-Raphson search used when evaluating the curve is considered converged when the magnitude of the returned value of `c` is smaller than `acc`. A maximum of `itmax` iterations will be used. The default value for `acc` is 10^{-6}. The odd syntax for defining this value is because an `int` is guaranteed to be convertible to an `F`, but a `double` is not. Therefore using `F acc = 1.0e-06` is not guaranteed to compile.

If `frac` is non-zero, the derivatives of `c` used in the Newton-Raphson search will be evaluated using finite differences. In this case a range must be set to delimit the solution vector: see `set_range` below. The separation of the parameters used for the derivatives will be `frac` times the full range of the parameter.

The use of finite differences to evaluate the derivatives in the Newton-Raphson search is normally not necessary, since all defining curves should allow evaluation of their first derivatives. However, when the defining curve is very complex, evaluation of the derivative is sometimes much slower than evaluation of finite differences. Typically the loss of accuracy implicit in the use of finite differences does not affect the convergence rate significantly.

The range $[a_m, b_m]$ used to limit the search can be set using the following member function.

```
void set_range(const ParamRange<M,F> &r)
```

Sets a range used to limit the Newton-Raphson search: see Section 11.1. The range may also be used to initialize the parameters if this is the first evaluation of the curve and `initialize` has not been called. The range is also used if the derivatives of the curve are to be evaluated by finite differences.

The Newton-Raphson search requires an initial guess for the values of $g_m(y_n)$; we will denote these values by g_m^i. `ImplicitCurve` allows the values of g_m^i to be defined explicitly. At the end of a successful Newton-Raphson search, the values of g_m^i will be set to the value returned by the implicit curve: i.e. the subsequent call to the implicit curve will be initialized using the value returned by the current call. After construction of an `ImplicitCurve`, the values of g_m^i are undefined. Also, if a Newton-

Raphson search is unsuccessful, the values of g_m^i become undefined.

Alternatively, an initialization curve, $c_m(y_n)$, can be specified. If $c_m(y_n)$ is defined, it will be used to generate the initial values: $g_m^i = c_m(y_n)$. Clearly, the initialization curve should be an approximation of the implicit curve itself.

The full algorithm for initializing the Newton-Raphson search is as follows:

if $c_m(y_n)$ is defined
    for all m, $g_m^i \longleftarrow c_m(y_n)$
if g_m^i is defined
    if the range $[a_m, b_m]$ is defined
        for all m, if $g_m^i < a_m$ or $g_m^i > b_m$ then
            $g_m^i \longleftarrow (a_m + b_m)/2$
else
    if the range $[a_m, b_m]$ is defined
        for all m, $g_m^i \longleftarrow (a_m + b_m)/2$
    else
        for all m, $g_m^i \longleftarrow 0$

Initialization is controlled using the following member functions:

`void initialize(const ValueType &p)`
: Sets the values of g_m^i to `p`. However, if an initialization curve is defined, when the search is initialized, these values will be overridden by the values returned by the initialization curve. Therefore, this call is only effective when there is no initialization curve.

`void set_initialization_curve(Curve<N,ValueType,F> crv)`
: Sets the initialization curve, $c_m(y_n)$ to `crv`.

`void reset_initialization_curve()`
: Throws away the initialization curve, $c_m(y_n)$. The next evaluation of the curve will be initialized using the current values of g_m^i.

As an example of the use of `ImplicitCurve<N,M,F>`, consider the curve $h(x) = e^{ax} + bx$. Let $g(a, b)$ be the positive value of x at which $h(x)$ intersects the unit circle centred at $(0, 1)$. A defining curve for $g(a, b)$ is:

$$f(g, a, b) = g^2 + (e^{ag} + bg - 1)^2 - 1 \tag{37}$$

We can define a curve to represent $g(a, b)$ using an `ImplicitCurve<2U,1U>` as follows:

```
using namespace CurveLib;
FOneParamCurve<3U> g(0), a(1), b(2);
Curve<3U,double> hm1 = Exp<>()(a*g) + b*g - 1.0;
```

```
MultiCurve<3U,1U> f;
f[0] = g*g + hm1*hm1 - 1.0;
ImplicitCurve<2U,1U> gcrv(f);
```

We know the value of $g(a, b)$ always lies in $[0, 1]$ so we can restrict the range of the returned value:

```
ParamRange<1U,double> range(0.0,1.0);
gcrv.set_range(range);
```

Now `gcrv` can be treated as any other curve. The code

```
ImplicitCurve<2U,1U>::ParamType p(1.0,2.0);
ImplicitCurve<2U,1U>::DerivType da(1,0), db(0,1);
std::cout << "The value of g(1,2)     = " << gcrv(p) << '\n'
  << "The value of dg/da(1,2) = " << gcrv(p,da) << '\n'
  << "The value of dg/db(1,2) = " << gcrv(p,db) << '\n';
```

results in

```
The value of g(1,2)     =  3.011131e-01
The value of dg/da(1,2) = -1.109622e-01
The value of dg/db(1,2) = -8.211137e-02
```

being written to standard output.

Notice that the defining curve, `f`, must return a 1-vector, not a scalar, so it was defined using `MultiCurve<3U,1U>` rather than say

```
Curve<3U,double> f = g*g + hm1*hm1 - 1.0;
```

Similarly, `gcrv` returns a 1-vector, not a `double`. To convert it so that it returns a `double` we could use `OneCompCurve<N,V,F>`:

```
OneCompCurve<2U,VecMtx::VecN<1U> > gdouble(gcrv,0);
```

11.3 Inverse curves

Let $\mathbf{h}(\mathbf{x})$ be a function of N variables, x_n, which returns a vector of length N. Define the inverse of $\mathbf{h}$ to be the curve, $\mathbf{h}^{-1}(\mathbf{x})$, for which

$$\mathbf{h}\big(\mathbf{h}^{-1}(\mathbf{x})\big) = \mathbf{x} \tag{38}$$

The class `InverseCurve<N,F>`, defined in the header file `ImplicitCurve.h`, represents the inverse curve $\mathbf{h}^{-1}(\mathbf{x})$. The number of parameters, `N`, must be the same as the length of the vector returned; therefore, the return type is `VecMtx::VecN<N,F>`.

`InverseCurve<N,F>` is implemented as an `ImplicitCurve<N,N,F>` whose defining curve, $f_i(x_n, y_m)$, is:

$$f_i(x_n, y_m) = h_i(x_n) - y_i \tag{39}$$

If template arithmetic is not supported by the compiler, `InverseCurve` has one extra template argument: `InverseCurve<N,N2,F>` where `N2` must equal `2*N`. This is because `ImplicitCurve` requires an extra template argument if template arithmetic is not supported.

Because it is implemented using `ImplicitCurve<N,N,F>`, the member functions of `InverseCurve<N,F>` are similar to those of `ImplicitCurve<N,N,F>`.

`InverseCurve(const CurveType &c, F acc = F(1)/F(1000000), unsigned itmax = 30, F frac = F(0))`

Makes a curve which is the inverse of `c`. A Newton-Raphson search is used to evaluate the inverse curve. It is considered converged when the magnitude of the returned value of `c` is smaller than `acc`. A maximum of `itmax` iterations will be used.

If `frac` is non-zero, the derivatives of `c` used in the Newton-Raphson search will be evaluated using finite differences. In this case a range must be set to delimit the solution vector. The separation of each parameter used for the finite differences will be `frac` times the full range of the parameter.

`void initialize(const ValueType &p)`

Sets an initial guess for the next evaluation of the implicit function. However, if an initialization curve is defined, when the search is initialized, these values will be overridden by the values returned by the initialization curve. Therefore, this call is only effective when there is no initialization curve.

`void set_initialization_curve(Curve<N,ValueType,F> crv)`

Sets the initialization curve to `crv`.

`void reset_initialization_curve()`

Throws away the initialization curve. The next evaluation of the curve will be initialized using the current values of g_m^i.

`void set_range(const RangeType &r)`

Sets a range used to limit the Newton-Raphson search. The range may also be used to initialize the parameters if this is the first evaluation of the curve and `initialize` has not been called. The range is also used if the derivatives of the curve are to be evaluated by finite differences.

A `FailingSearch` exception is thrown if there is no convergence when an inverse curve is evaluated.

The class `FInverseCurve<F>` is very similar to `InverseCurve<N,F>` but handles the case when the defining curve has a single argument and returns a scalar (i.e. an `F`); `InverseCurve<1U,F>` is used if the defining curve has a single argument but returns

a 1-vector. The member functions of `FInverseCurve<F>` are very similar to those of `InverseCurve<1U,F>`.

```
FInverseCurve(const CurveType &c, F acc = F(1)/F(1000000),
              unsigned itmax = 30, F frac = F(0))
```
Makes a curve which is the inverse of `c`.

```
void initialize(F p)
```
Sets an initial guess for the next evaluation of the implicit function.

```
void set_initialization_curve(Curve<1U,F,F> crv)
```
Sets the initialization curve to `crv`.

```
void reset_initialization_curve()
```
Throws away the initialization curve.

```
void set_range(F plo, F phi)
```
Sets a range used to limit the Newton-Raphson search.

For example, the following code could be used to define a curve to represent $f(x) = \arcsin(x)$ (the curve `ArcSin<>` is much more efficient).

```
using namespace CurveLib;
Sin<> sin_crv;
FInverseCurve<> arcsin(sin_crv);
double pio2 = 0.5*3.14159265358979323846;
arcsin.set_range(-pio2,pio2);
```

12 Defining a new curve

Although the classes and functions defined in the previous sections provide powerful means for generating complex curves from simple ones, there will inevitably be situations in which you want to define a curve by making a specialization of the class `Curve<N,V,F>`. This section describes what you need to know to do that.

A `Curve<N,V,F>` is really a handle to a representation of the curve (see Stroustrup[6], Chapter 13.9 for a discussion of handles). Therefore, when defining a new curve one has two define two new classes: the handle, which is a specialization of `Curve<N,V,F>`; and the representation, which is a specialization of `CurveRep<N,V,F>`. It is the representation which actually does all the work in evaluating the curve.

A handle contains a pointer to its representation (the pointer is stored in a base class). Two different handles may share the same representation. The representation maintains a count of how many handles are sharing it. When that count drops to zero, the representation is deleted. This scheme has some implications:

- Representations must always be allocated from the heap.
- A representation should never be deleted explicitly; it will be deleted automatically when no handle is using it. Therefore one never has to worry about the persistence of a representation (i.e. whether the representation will be deleted before it is used).

The following member functions are the minimum required for the handle:

1. A constructor which allocates an appropriate representation from the heap.
2. An assignment operator which uses the assignment operator of the base class `Curve<N,V,F>::operator=()` to copy the pointer to the representation. This operator ensures that the reference count of the representation is incremented. The default assignment operator must not be used. If it were, the pointer to the representation (stored in the base classes) would be be copied without increasing its reference count. The reference count will then drop to zero prematurely causing the representation to be deleted while still being used. It is likely that an obscure memory allocation error will result.

The representation class *must* define a virtual destructor as well as the following virtual member functions.

`virtual F value(const ParamType &p) const`
: Evaluates the curve for parameters `p`. The argument type `ParamType` is an alias for `VecMtx::VecN<N,F>`: i.e. it is a vector of length `N`.

`virtual F value(const ParamType &p, const DerivType &d) const`
: Evaluates the derivatives of the curve for parameters `p`. `DerivType` is an alias for `Derivs<N>`.

`virtual Share::HandleRep* copy_self() const`
: Allocates a new instance of the representation from the heap, copies itself into the new representation, and then returns a pointer to it. This function is required by the base class `Share::HandleRep` which contains the attributes necessary for representations of handles.

For example, suppose we wish to define a curve which will evaluate $f(x) = \sin(x)/x$ even when $x = 0$. This cannot be done using the functions and classes described in earlier sections. Instead we define a new class, `SinxOx`, and its representation `SinxOxRep`. We will assume that the parameters and return value are of type `double`.

We consider the representation first.

```
using namespace CurveLib;
class SinxOxRep: public CurveRep<1U,double,double>
{
```

```
public:
  SinxOxRep() { }

  SinxOxRep(const SinxOxRep &rep)
   : CurveRep<1U,double,double>(rep)
   { }

  virtual ~SinxOxRep() { }

  virtual double value(const ParamType &p) const;

  virtual double value(const ParamType &p, const DerivType &d) const;

private:
  virtual Share::HandleRep* copy_self() const {
    return new SinxOxRep(*this);
  }
};
```

The class has a default constructor and a copy constructor (not strictly necessary since this is equivalent to the default copy constructor). The virtual destructor must be present, even though it does nothing explicit. The inherited function `copy_self` returns a new copy of the class allocated from the heap; the use of the copy constructor here is typical.

The overloaded `value` functions are used to evaluate the curve and its derivatives. They might be defined as follows using the fact that

$$\left.\frac{d^n}{dx^n}\left(\frac{\sin(x)}{x}\right)\right|_{x\to 0} = \begin{cases} \frac{(-1)^{n/2}}{n+1} & \text{if } n \text{ is even} \\ 0 & \text{if } n \text{ is odd} \end{cases} \tag{40}$$

```
double SinxOxRep::value(const ParamType &p) const
{
  if (p[0] == 0.0) return 1.0;
  else return sin(p[0])/p[0];
}

double SinxOxRep::value(const ParamType &p,
                        const DerivType &d) const
{
  if (d[0] == 0) return value(p);
  if (p[0] == 0.0) {
    if (d[0]%2 == 1) return 0.0;
    if (d[0]%4 == 0) return  1.0/(d[0]+1);
    else             return -1.0/(d[0]+1);
  }
  else {
```

```
    if (d[0] == 1) {
      double x = p[0];
      return (x*cos(x)-sin(x))/(x*x);
    }
    else {
      FIdentityCurve<> x;
      Curve<1U,double,double> d_sinxox_crv =
        (x*Cos<>()-Sin<>())*PowInt<>(-2);
      return d_sinxox_crv(p[0],d[0]-1);
    }
  }
}
```

When $x \neq 0$, the first derivative is calculated explicitly but higher derivatives are calculated by constructing a curve equal to the first derivative, then evaluating the derivatives of that. This technique for evaluating higher derivatives is typical.

These functions could be improved in several ways. In particular, it would be better to define a small region around zero in which the curve was evaluated using a Taylor expansion. Evaluating the higher derivatives would be more efficient if more cases were handled explicitly and if the curve `d_sinxox_crv` were a member.

The curve itself can now be defined as follows.

```
using namespace CurveLib;
class SinxOx: public Curve<1U,double,double>
{
public:
  SinxOx()
   : Curve<1U,double,double>(new SinxOxRep())
   { }

  SinxOx(const SinxOx &sinxox)
   : Curve<1U,double,double>(sinxox)
   { }

  ~SinxOx() { }

  SinxOx& operator=(const SinxOx &sinxox) {
    Curve<1U,double,double>::operator=(sinxox);
    return *this;
  }
};
```

The first constructor creates a new representation for the curve. It uses a protected `Curve<1U,double,double>` constructor to assign this representation to the curve. The representation *must* be allocated from the heap using the `new` operator. If it is not, obscure memory errors will result.

Notice that `SinOx` never explicitly deletes its representation. This is handled by the base classes `Share::Handle` and `Share::HandleRep` which provide the attributes for handles and their representations. The representations can be shared among different handles; this happens whenever a curve is copied. The representation stores a reference count: the number of handles which are currently sharing it. When the reference count falls to zero, the representation is deleted automatically.

The `SinOx` copy constructor and destructor are not strictly necessary, but it is usually a good idea to include them. On the other hand, as discussed above, the assignment operator *must* be defined.

13 Concluding remarks

This document has described a library of C++ classes which represent differentiable multi-parameter functions. Although originally developed for representing complex geometric shapes, these classes can be used in much wider applications.

The main utility of the classes lies in the ability to generate complex functions from simple ones using arithmetic operators, composition, vector operators and inverse methods. The resulting curve is always fully differentiable.

References

[1] Hally, D. (2006), C++ classes for representing curves and surfaces: Part II: Splines, (DRDC Atlantic TM 2006-255) 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] Standard Template Library Programmer's Guide (online), Silicon Graphics, Inc., http://www.sgi.com/tech/stl (Access Date: November 2006).

[6] Stroustrup, B. (1991), The C++ Programming Language, 2nd ed, Addison-Wesley Publishing Co.

[7] (1987), ANSI/IEEE Standard 754-1985, Standard for Binary Floating Point Arithmetic, ACM SIGPLAN Notices 22(2). Also see: http://grouper.ieee.org/groups/754.

Annex A: Concepts

A.1 Arithmetic Object

Description
Arithmetic Object is a concept that is defined in conjuction with the Scalar Object concept (see Annex A.2). We say that a type `V` is a model of Arithmetic Object with respect to the Scalar Object `F`.

Refinement of
Assignable, Default Constructible, Equality Comparable (these concepts are defined by the Standard Template Library).

Associated types
Scalar Object

Notation
`F` A type that is a model of Scalar Object.

`V` A type that is a model of Arithmetic Object with respect to `F`.

`x` Object of type `F`.

`v,w` Objects of type `V`.

Valid expressions
In addition to the expressions defined by the Standard Template Library (STL) concepts Assignable, Default Constructible, and Equality Comparable, the following expressions must be valid.

Name	Expression	Return type
Negation	`-v`	`V`
Addition	`v + w`	`V`
Subtraction	`v - w`	`V`
Scaling	`x*v`	`V`
Scaling	`v*x`	`V`
Scaling	`v/x`	`V`
In place addition	`v += w`	`V&`
In place subtraction	`v -= w`	`V&`
In place scaling	`v *= x`	`V&`
In place scaling	`v /= x`	`V&`
Setting to zero	`zero(v)`	

Expression semantics

Name	Expression	Pre-condition	Semantics
Negation	`-v`		Returns the negation of `v`
Addition	`v + w`		Adds `v` and `w`
Subtraction	`v - w`		Subtracts `w` from `v`. Equivalent to `v + (-w)`.
Scaling	`x*v`		Scales `v` by `x`
Scaling	`v*x`		Scales `v` by `x`
Scaling	`v/x`		Scales `v` by `1/x`. Not defined if `x == F(0)` is true.
In place addition	`v += w`		Adds `w` to `v` and returns a reference to `v`. Equivalent to `v = v + w`.
In place subtraction	`v -= w`		Subtracts `w` from `v` and returns a reference to `v`. Equivalent to `v = v - w`.
In place scaling	`v *= x`		Scales `v` by `x` and returns a reference to `v`. Equivalent to `v = v*x`.
In place scaling	`v /= x`		Scales `v` by `1/x` and returns a reference to `v`. Equivalent to `v = v/x`.
Setting to zero	`zero(v)`		Sets the object to zero.

Invariants

Most models of Scalar Object include *exceptional values* for which the normal arithmetic rules do not apply: examples are `NaN` and `Infinity` for `float` or `double` conforming to the IEEE Standard 754 for Floating Point Arithmetic[7]. The following invariants are required only if no component of `v` or `w` is an exceptional value.

`x*v == v*x` is true.
`v*F(0) == -(v*F(0))` is true.
`v + w*F(0) == v` is true.
`v*F(1) == v` is true.

Models

`float`, `double`, `long double`, `std::complex<float>`, `std::complex<double>` and `std::complex<long double>` are all models of Arithmetic Object with respect to themselves.

`VecMtx::VecN<N,F>` is a model of Arithmetic Object with respect to `F`.

Notes

Arithmetic Object has the arithmetic attributes required by scalars, vectors and matrices. The returned values of curves must model Arithmetic Object with respect

to the Scalar Object used for the curve parameters.

The CurveLib classes require a means of setting a returned value of a curve to zero. Unfortunately, there is no means of constructing an Arithmetic Object which ensures that its value, or the values of its components if it is a Vector Object (see Annex A.4), are not exceptional values. Otherwise it would be sufficient to construct the Arithmetic Object and multiply it by `F(0)`. Therefore it is necessary to require an extra function which handles the assignment. For use in the CurveLib library, it is sufficient that the function `zero(const V&)` be defined either in the namespace CurveLib or in the global namespace.

A.2 Scalar Object

Description
Scalar Object is a concept which reflects the attributes of a floating point number. Each parameter of a curve must be a Scalar Object.

Refinement of
A Scalar Object is an Arithmetic Object with respect to itself.

Notation
`F` A type that is a model of Scalar Object.

`x,y` Objects of type `F`.

`n` An object of type `int`.

Valid expressions
In addition to the expressions defined by the concepts Arithmetic Object, Assignable, Default Constructible and Equality Comparable, the following expressions must be valid.

Name	Expression	Return type
Conversion from `int`	`F(n)`	`F`
Square root	`sqrt(x)`	`F`
Exponentiation	`pow(x,n)`	`F`

Expression semantics

Name	Expression	Pre-condition	Semantics
Conversion from `int`	`F(n)`		Creates a Scalar Object with value equivalent to `n`
Square root	`sqrt(x)`		Returns the square root of `x`.
Exponentiation	`pow(x,n)`	`x != F(0) || n != 0`	Returns `x` to the power `n`

Invariants
The following invariants are required if `x` is not an exceptional value (e.g. `NaN` or `Infinity`).

`F(0) == -F(0)` is true.
`x*F(0) == F(0)` is true.
`x + F(0) == x` is true.
`x != F(0) && F(0)/x == F(0)` is true.
`x != F(0) && x/x == F(1)` is true.

Models
`float`, `double`, `long double`, `std::complex<float>`, `std::complex<double>`, `std::complex<long double>`

Notes
It is intended that a Scalar Object is some sort of floating point number but, due to inaccuracies caused by round-off, it is difficult to specify all the invariants that would ensure this in a mathematical sense. For example, associativity (`x*(y*z) == (x*y)*z` is true) is required to ensure that a Scalar Object is a mathematical field, but it cannot be guaranteed in most implementations of floating point numbers. Moreover, a clear requirement that `x*(y*z) - (x*y)*z` is small would require a notion of size and comparability that we cannot give to the Scalar Object concept.

It may seem odd to require the `sqrt` and `pow` functions, but no other elementary functions. The `sqrt` function is needed for determining the magnitude of vectors and the `pow` function is needed for the efficient evaluation of high order derivatives of some curves (for example, if $g(x) = f(ax)$ then $\partial^n g(x)/\partial x^n = a^n \partial^n f(ax)/\partial (ax)^n$; `pow` is used to evaluate a^n). This is not wholly satisfactory as it requires providing a Scalar Object with two functions which it may normally not require (for example, an `Angle<double>`, described in Annex E, is not a Scalar Object solely because it lacks these two functions) but has been adopted as the simplest means of providing efficient implementations of both scalar and vector valued functions.

A.3 Comparable Scalar Object

Description
Comparable Scalar Object is a concept which reflects the attributes of a floating point number for which comparison operators are defined. Complex numbers can be Scalar Objects but not Comparable Scalar Objects.

Refinement of
Comparable Scalar Object is a refinement of both Scalar Object and the STL concept Less Than Comparable. A Comparable Scalar Object is also an Absolute Object with respect to itself. It adds no additional syntax to the expressions defined by these models.

Expression semantics
Comparable Scalar Object inherits all the expressions of Scalar Object and Less Than Comparable and adds no new ones. However, it does add a pre-condition to the `sqrt` function:

Name	Expression	Pre-condition	Semantics
Square root	`sqrt(x)`	`x >= F(0)`	Returns the square root of `x`.

Invariants
`x < x` is false
`x < y` implies `!(y > x)`
`x < y && y < z` implies `x < z`
`x*x >= F(0)` is true
`x >= F(0) && abs(x) == x` is true
`x <= F(0) && abs(x) == -x` is true

Models
`float`, `double`, `long double`

A.4 Vector Object

Description
Vector Object is a concept which reflects the attributes of a vector with arithmetic operators. Each component of the vector is a Scalar Object.

Refinement of
A Vector Object is an Arithmetic Object with respect to some Scalar Object which we will denote by `F`.

Associated types
Scalar Object

Notation
`V` A type that is a model of Vector Object.

`F` The type of the components of an object of type `V`: a model of Scalar Object.

`v,w` Objects of type `V`.

`x` An object of type `F`.

`n` An `unsigned int`.

Valid expressions
In addition to the expressions defined by the concept Arithmetic Object, the following expressions must be valid.

Name	Expression	Return type
Size	`v.size()`	`unsigned int`
Subcripting	`v[n]`	`F`

Expression semantics

Name	Expression	Pre-condition	Semantics
Subcripting	`v[n]`	`n < v.size()`	Returns component `n`. If `v` is not `const`, the returned value is an l-value: i.e. it is assignable.

Invariants

The arithmetic operators act component by component. The following invariants are required if `x` is not an exceptional value (e.g. `NaN` or `Infinity`).

`(-v)[i] == -v[i]` is true.
`(v1+v2)[i] == v1[i]+v2[i]` is true.
`(v1-v2)[i] == v1[i]-v2[i]` is true.
`(x*v)[i] == x*v1[i]` is true.
`x != F(0) && (v/x)[i] == v1[i]/x` is true.

Models

`VecMtx::VecN<N,F>` is a model of Vector Object with respect to `F`.

Notes

It is intended that a Vector Object is a vector of floating point numbers but, due to inaccuracies caused by round-off, it is difficult to specify all the invariants that would ensure this in a mathematical sense. For example, associativity (`x+(y+z) == (x+y)+z` is true) is required to ensure that a Vector Object is a mathematical vector space, but it cannot be guaranteed in most implementations of floating point numbers. Moreover, a clear requirement that `(x+(y+z)) - ((x+y)+z)` is small would require a notion of size and comparability that we cannot give to the Vector Object concept.

A.5 Absolute Object

Description

An Absolute Object with respect to a Comparable Scalar Object `F` is one for which an absolute value of type `F` can be assigned. If the Absolute Object is a vector type, the absolute value is considered to be the magnitude of the vector.

Associated types

Scalar Object

Notation

`F` A type that is a model of Scalar Object.

`A` A type that is a model of an Absolute Object with respect to `F`.

`a` An object of type `A`.

Valid expressions

The following expression must be valid.

Name	Expression	Return type
Absolute value	`abs(a)`	`F`

Expression semantics

Name	Expression	Pre-condition	Semantics
Absolute value	`abs(a)`		Returns the absolute value of `a`.

Invariants

`abs(a) >= F(0)` is true.

Models

`float`, `double` and `long double` are Absolute Objects with respect to themselves.

`std::complex<F>` is an Absolute Object with respect to `F` for `F` of type `float`, `double` and `long double`.

`VecMtx::VecN<N,F>` is an Absolute Object with respect to `F` whenever `F` is a model of Comparable Scalar Object.

This page intentionally left blank.

Annex B: Prototypes for VecMtx::VecN

The parameter list of a curve is represented using the class `VecN<N,F>` in the namespace `VecMtx`. It is a vector of `N` values each of which has type `F`. `F` must be a model of a Scalar Object: see Annex A.2. If the template parameter `F` is omitted, it defaults to `double`. `N` is of type `unsigned int`.

`VecN<N,F>` is a model of a Vector Object with respect to `F` (see Annex A.4) and hence implements a full range of arithemtic operators. It is also suitable for the return value of a `CurveLib::Curve<N,V,F>`.

`VecN<N,F>` is designed for optimal storage: its size is the same as an array of type `F[N]` which is typically `N*sizeof(F)` plus any extra space required for alignment on word boundaries.

B.1 Constructors

`VecN<N,F>` has three constructors:

`VecN()`
: The values remain undefined.

`explicit VecN(const F[N] v1)`
: The values in `v1` are copied.

`VecN(const VecN<N,F> &v1)`
: Copy constructor: the values in `v1` are copied.

The classes `VecN<1U,F>`, `VecN<2U,F>` and `VecN<3U,F>` are special. They each have an additional constructor which allows its values to be specified by one, two or three values of type `F`.

`VecN(F x)`
: An extra constructor for `VecN<1U,F>` which sets its single element to `x`.

`VecN(F x, F y)`
: An extra constructor for `VecN<2U,F>` which sets its two elements to `x` and `y`.

`VecN(F x, F y, F z)`
: An extra constructor for `VecN<3U,F>` which sets its three elements to `x`, `y` and `z`.

B.2 Other member functions

`unsigned size() const`
: Returns the length of the vector.

`F& operator[](unsigned i)`
: Returns vector element `i` as an l-value. There is no bounds checking.

`F operator[](unsigned i) const`
: Returns the value of vector element `i`. There is no bounds checking.

B.3 Other functions

As a model of an Vector Object with respect to `F`, `VecN<N,F>` implements a wide range of arithmetic operators. The following functions are also defined (the class `VecMtx::MtxN<N,F>` is described in Annex C):

```
template<unsigned N, class F>
F dot(const VecN<N,F> &v1, const VecN<N,F> &v2)
```
: Returns the dot product of `v1` and `v2`.

```
template<class F>
VecN<3U,F> cross_product(const VecN<3U,F> &v1,
                         const VecN<3U,F> &v2)
```
: Cross product for 3-vectors.

```
template<unsigned N, class F>
F abs(const VecN<N,F> &v)
```
: Equivalent to `sqrt(dot(v,v))`. If `F` is a real type, `abs` returns the magnitude of `v`. If `F` is a complex type, then, since the returned value is a complex number, the returned value cannot be interpreted as the magnitude of the vector.

```
template<unsigned N, class F>
VecN<N,F>& unit(VecN<N,F> &v)
```
: Equivalent to `v/abs(v)`. If `F` is a real type, `unit` converts `v` to a unit vector and returns a reference to `v`.

```
template<unsigned N, class F>
VecN<N,F> prod_by_element(const VecN<N,F> &v1,
const VecN<N,F> &v2)
```
: Returns a vector whose elements are the product of the elements of `v1` and `v2`.

```
template<unsigned N, class F>
VecN<N,F> operator*(const MtxN<N,F> &m, const VecN<N,F> &v)
```
: Multiplication by a matrix. Returns `m*v`.

```
template<unsigned N, class F>
VecN<N,F> operator*(const VecN<N,F> &v, const MtxN<N,F> &m)
```
: Multiplication by a matrix. Returns `v*m`.

```
template<unsigned N, class F>
bool operator==(const VecN<N,F> &v1, const VecN<N,F> &v2)
```
True if the elements of `v1` and `v2` are the same.

```
template<unsigned N, class F>
bool operator!=(const VecN<N,F> &v1, const VecN<N,F> &v2)
```
True if the elements of `v1` and `v2` are different.

```
template<unsigned N, class F>
std::ostream& operator<<(std::ostream &out, const VecN<N,F> &v)
```
Writes `v` to `out` in scientific notation.

```
template<unsigned N, class F>
std::istream& operator>>(std::istream &in, VecN<N,F> &v)
```
Reads `v` from `in`.

The following functions are also defined for `VecN<N,F>` provided that the compiler allows template arithmetic.

```
template<unsigned N, class F>
VecN<N-1,F> reduce(const VecN<N,F> &v, unsigned k)
```
Makes a vector of length `N-1` by removing element `k` from `v`. `N` must exceed 1.

```
template<unsigned N, class F>
VecN<N+1,F> extend(const VecN<N,F> &v, unsigned k, F val)
```
Makes a vector of length `N+1` by inserting `val` into `v` at element `k`.

This page intentionally left blank.

Annex C: Prototypes for VecMtx::MtxN

A `MtxN<N,F>` is an `N` × `N` matrix whose elements are of type `F`. `F` must be a model of a Scalar Object: see Annex A.2. If the template parameter `F` is omitted, it defaults to `double`. `N` is of type `unsigned int`.

`MtxN<N,F>` is a model of an Arithmetic Object with respect to `F` (see Annex A.1) and hence implements a full range of arithemtic operators. It is also suitable for the return value of a `CurveLib::Curve<N,V,F>`.

`MtxN<N,F>` is designed for optimal storage: its size is the same as an array of type `F[N*N]` which is typically `N*N*sizeof(F)` plus any extra space required for alignment on word boundaries.

C.1 Constructors

Only the default no-argument constructor is defined explicitly. When it returns the matrix elements are undefined. The default copy constructor can also be used.

The classes `MtxN<1U,F>`, `MtxN<2U,F>` and `MtxN<3U,F>` are special. Their functions have been defined to be very effcient. `MtxN<1U,F>` also has an additional constructor:

`MtxN(F x)`
: An extra constructor for `MtxN<1U,F>` which sets its single element to `x`.

C.2 Other member functions

`unsigned num_rows() const`
: Returns the number of rows, `N`.

`unsigned num_cols() const`
: Returns the number of columns, `N`.

`F& operator()(unsigned i,unsigned j)`
: Returns matrix element `(i,j)` as an l-value. There is no bounds checking.

`F operator()(unsigned i,unsigned j) const`
: Returns the value of matrix element `(i,j)`. There is no bounds checking.

C.3 Other functions

As a model of an Arithmetic Object with respect to `F`, `MtxN<N,F>` implements a wide range of arithmetic operators. The following functions are also defined (the class `VecMtx::VecN<N,F>` is described in Annex B):

```
template<unsigned N, class F>
void identity(MtxN<N,F> &m, F d)
```
Sets the matrix `m` to the identity times `d`.

```
template<unsigned N, class F>
inline void identity(MtxN<N,F> &m)
```
Sets the matrix `m` to the identity.

```
template<unsigned N, class F>
MtxN<N,F>& transpose(MtxN<N,F> &m)
```
Transposes `m`, then returns it.

```
template<unsigned N, class F>
MtxN<N,F>& invert(MtxN<N,F> &m)
```
Inverts `m` using Gaussian elimination with pivoting. If the matrix is singular, a `VecMtx::SingularMatrix` exception (derived from class `Error` described in Annex F) is thrown.

```
template<unsigned N, class F>
VecN<N,F> operator*(const MtxN<N,F> &m, const VecN<N,F> &v)
```
Returns `m*v`.

```
template<unsigned N, class F>
VecN<N,F> operator*(const VecN<N,F> &v, const MtxN<N,F> &m)
```
Returns `v*m`.

```
template<unsigned N, class F>
MtxN<N,F> operator*(const MtxN<N,F> &m1, const MtxN<N,F> &m2)
```
Returns `m1*m2`.

```
template<unsigned N, class F>
MtxN<N,F>& scale_rows(MtxN<N,F> &m, const VecN<N,F> &v)
```
For each `i` from 0 to `N-1`, scales row `i` of `m` by `v[i]`.

```
template<unsigned N, class F>
MtxN<N,F>& scale_columns(MtxN<N,F> &m, const VecN<N,F> &v)
```
For each `i` from 0 to `N-1`, scales column `i` of `m` by `v[i]`.

```
template<unsigned N, class F>
inline bool operator==(const MtxN<N,F> &m1, const MtxN<N,F> &m2)
```
Returns true if `m1` and `m2` are equal.

```
template<unsigned N, class F>
inline bool operator!=(const MtxN<N,F> &m1, const MtxN<N,F> &m2)
```
Returns true if `m1` and `m2` are not equal.

```
template<unsigned N, class F>
std::ostream& operator<<(std::ostream &out, const MtxN<N,F> &m)
```

Writes `m` to `out`, one row per line, in scientific notation.

```
template<unsigned N, class F>
std::istream& operator>>(std::istream &in, MtxN<N,F> &v)
```

Reads `m` from `in`. The first row is read first, then the second row, etc.

This page intentionally left blank.

Annex D: Prototypes for CurveLib::Derivs

When a curve is evaluated, the number of derivatives with respect to each parameter must be specified. This is done using the class `Derivs<N>` in namespace `CurveLib`. `N` is of type `unsigned int`.

`Derivs<N>` is essentially an array of `unsigned int`s. It uses the standard square bracket syntax for subscripting. Element `n` specifies the number of derivatives to be taken with respect to parameter `n`.

D.1 Constructors

`Derivs<N>` has the following constructors:

`Derivs()`
: Default constructor: sets the number of derivatives for each parameter to zero.

`Derivs(const Derivs<N> &d)`
: Copy constructor.

The classes `Derivs<1U>`, `Derivs<2U>` and `Derivs<3U>` are special. They each have an additional constructor which allows its values to be specified by one, two or three values of type `F`.

`Derivs(F i)`
: An extra constructor for `Derivs<1U>` which sets its single element to `i`.

`Derivs(F i, F j)`
: An extra constructor for `Derivs<2U>` which sets its two elements to `i` and `j`.

`Derivs(F i, F j, F k)`
: An extra constructor for `Derivs<3U>` which sets its three elements to `i`, `j` and `k`.

D.2 Other member functions

`Derivs<N>& operator=(const Derivs<N> &d)`
: Assignment operator.

`unsigned size() const`
: Returns the number of parameters.

`unsigned total() const`
: Returns the total number of derivatives to be taken: i.e. the sum of the values in the array.

`unsigned& operator[](unsigned i)`

Returns the number of derivatives for parameter `i` as an l-value.

`unsigned operator[](unsigned i) const`

Returns the number of derivatives for parameter `i`.

D.3 Other functions

The following functions are also defined for `Derivs<N>` providing that template arithmetic is allowed by the compiler. If it is not, then `reduce` functions will only be defined for `N` equal to 2 and 3, and `extend` will only be defined for `N` equal to 1 or 2. Both `reduce` and `extend` are in the namespace `CurveLib`.

```
template<unsigned N>
Derivs<N-1> reduce(const Derivs<N> &d, unsigned k)
```

Makes a `Derivs<N-1>` from `d` by removing element `k`.

```
template<unsigned N>
Derivs<N+1> extend(const Derivs<N> &d, unsigned k, unsigned v)
```

Makes a `Derivs<N+1>` by inserting `v` into `d` at element `k`.

Annex E: Prototypes for class Angle

The class `Angle<F>` represents an angle. The template argument `F` must be a model of a Comparable Scalar Object (see Annex A.3).

`Angle<F>` is a model of an Arithmetic Object with respect to `F` (see Annex A.1) and the STL concept Less Than Comparable; hence, it implements a full range of arithmetic and comparison operators. It is also suitable for the return value of a `CurveLib::Curve<N,V,F>`.

Division of an `Angle<F>` by an `Angle<F>` is also defined (this is not required by the Arithmetic Object concept); it returns an `F` equal to the ratio of the angles.

E.1 Constructors

`Angle()`
: Default constructor: the value of the angle is unspecified.

`Angle(F ang_rad)`
: Makes an angle equivalent to `ang_rad` radians.

E.2 Static members

`static F pi`
: A representation of π as an `F`. It is used for converting between radians and degrees. Explicit representations are provided when `F` is a `float`, `double` or `long double`.

`static Angle<F> right_angle()`
: Returns the angle equal to $\pi/2$ radians or 90 degrees.

`static Angle<F> straight_angle()`
: Returns the angle equal to π radians or 180 degrees.

`static Angle<F> full_circle()`
: Returns the angle equal to 2π radians or 360 degrees.

E.3 Member functions for setting and retrieving the angle

`void set_degrees(F deg)`
: Sets the value of the angle in degrees.

`void set_radians(F rad)`
Sets the value of the angle in radians.

`F degrees() const`
Return the value of the angle in degrees.

`F radians() const`
Returns the value of the angle in radians.

`Angle<F>& canonical()`
Adjusts the angle to lie in $[0, 2\pi)$. Returns the new angle.

E.4 Trigonometric functions

The six trigonometric functions `sin(Angle<F>)`, `cos(Angle<F>)`, `tan(Angle<F>)`, `csc(Angle<F>)`, `sec(Angle<F>)` and `cot(Angle<F>)` are defined. They require that the `sin(F)`, `cos(F)` and `tan(F)` are defined and return an `F`.

Annex F: Prototypes for class Error

All exceptions in `CurveLib` are derived from the class `Error`. Any exceptions thrown by user-defined classes derived from `Curve<N,V,F>` or `CurveRep<N,V,F>` should also be derived from `Error`.

An `Error` stores a message which describes the error which has occurred. Member functions allow this message to be retrieved, to be appended to, or to be prepended to.

`Error(const std::string &m)`
: Constructs an error with message `m`.

`Error(const Error &e)`
: Copy constructor.

`void operator=(const Error &e)`
: Assignment operator: copies the error message.

`void append_to_msg(std::string s)`
: Appends the argument `s` to the error message.

`void prepend_to_msg(std::string s)`
: Prepends the argument `s` to the error message.

`std::string get_msg() const`
: Returns the error message.

F.1 Prototypes for class ProgError

`ProgError` is an exception thrown when an error occurs that can clearly be identified as a programming error rather than a run-time error. `ProgError` is derived from `Error` and has the following member functions in addition to those defined by `Error`.

`ProgError(const std::string &f, const std::string &m)`
: Constructs a `ProgError` for an error in function `f`. The error message is in `m`.

`const std::string& get_func() const`
: Returns the string describing the location of the error.

Index

Abs<N,F>, **15**
Absolute Object, 37, 56, **58–59**
Angle<F>, **71–72**
AnyPlaneProjCurve<N,F>, **27**
ArcCos<F>, **11**
ArcCosh<F>, **13**
ArcSin<F>, **10–11**
ArcSinh<F>, **13**
ArcTan2<F>, **11**
ArcTan<F>, **11**
ArcTanh<F>, **13**
Arithmetic Object, 2, **53–55**, 65, 71
arithmetic operators for curves, 5–7
AxisymmetricSurface<F>, **31–33**

Bessel functions, 13–14
BesselJ<F>, **14**
BesselY<F>, **14**
BilinearPatch<V,F>, **19–20**

Christoffel symbols, 29, 30
Comparable Scalar Object, 13, 37, **56–57**, 71
complex conjugate curves, 17
ComplexConjCurve<N,F1,F>, **17**
ComposedCurve<N,M,V,F>, **7–9**
ComposedCurveFParam<N,V,F>, **8–9**
composition of curves, 7–9
ConcatenatedCurve<N,M1,M2,F>, **23–24**
constant parameter curves, 16–17
ConstPCurve<N,NP1,V,F>, **16–17**
ConstPCurve<N,V,F>, **16–17**
Cos<F>, 5, 7, 9, **10**, 18, 30, 33
Cosh<F>, **12**
Cot<F>, **10**
Coth<F>, **12**
cross product curve, 25–26
CrossProdCurve<N,F>, **25–26**
Csc<F>, **10**
Csch<F>, **12**
Curve<1U,F,F>, 10, 13, 47
Curve<2U,VecMtx::VecN<3U,F>,F>, 29, 31
Curve<N,F,F>, 15
Curve<N,V,F>, 3, **2–4**, 5, 7, 8, 15, 18, 25, 27, 38, 47, 61, 65, 71, 73
CurveRep<N,V,F>, 47, 73
CylProjCurve<N,F>, **28**

defining new curves, 47–51
DerivCurve<N,V,F>, **15–16**
Derivs<N>, 2, 16, 48, **68–70**
DerivType, **2**, 3, 4, 48, 49
dot product curve, 25
DotCurve<N,V,F>, 14, 25, **25**, 26

elementary functions, 9–14
Erf<F>, **12**
Erfc<F>, **12**
exceptions
  Error, 3, 5, 41, 66, **73**
  FailingSearch, 41, 46
  NormalNotDefined, 30–33
  ProgError, 5, 15, 18, **73**
  VecMtx::SingularMatrix, 66
Exp<F>, **11**
exponential functions, 11–12

FIdentityCurve<F>, **14**
FInverseCurve<F>, **46–47**
FOneParamCurve<N,F>, 4, **14–15**, 16, 18, 44

header files
  ImplicitCurve.h, 42, 45
hyperbolic functions, 12–13

identity curves, 14–15
IdentityCurve<N,F>, 8, **14**, 15, 24
implicit curves, 40–47
ImplicitCurve<N,M,F>, **41–45**, 45
ImplicitCurve<N,M,NPM,F>, **41–45**

interpolation, 33–37
transfinite, 34–37
inverse curves, 45–47
InverseCurve<N,F>, **45**–**46**, 46

linear curves, 19–20
linear transformations of parameters, 20–21
LinearCurve<V,F>, **19**
LinearParamCurve<N,V,F>, **20**–**21**
Log<F>, **11**–**12**

MultiCurve<N,M,F>, **17**–**19**, 30, 33, 44

namespaces
CurveLib, 2, 5, 69
Share, 48, 49, 51
std, 2, 5, 9, 11, 12, 18, 22, 30, 31, 33, 54, 56, 58, 59, 63, 67, 73
VecMtx, 2, 18, 23–29, 31–33, 37, 42, 43, 45, 48, 54, 59, 61–63, 65–67
Newton-Raphson search, 40–41, 43, 44, 46, 47

OneCompCurve<N,V,F>, **22**–**23**, 45
OneParamCurve<N,F>, **14**–**15**, 30

parameter ranges, 37–38
ParamRange<N,F>, **37**–**38**, 38
ParamType, **2**, 3, 4, 8, 14, 15, 17, 20, 21, 24, 29, 37–39, 48, 49
PlaneProjCurve<N,V,F>, **27**
Polynomial<V,F>, 21, 22
polynomials, 21–22
Pow<F>, **9**–**10**, 16, 17
PowInt<F>, **10**
projection to a cylinder, 28
projection to a plane, 27
projection to a sphere, 28

Range curves, 37–40
RangeCurve<N,V,F>, 37, **38**–**40**, 40
ReducedDimCurve<N,M,F>, **23**
ReflectedCurve<N,V,F>, **24**
ruled curves, 33–34
RuledCurve<N,NM1,V,F>, **33**–**34**
RuledCurve<N,V,F>, **33**–**34**

Scalar Object, 2, **55**–**56**, 61, 65
Sec<F>, **10**
Sech<F>, **12**
Share classes
Handle, 51
HandleRep, 48, 49, 51
Sin<F>, 5, **10**, 14, 18, 21, 30, 33, 47
Sinh<F>, **12**
SinxOx, **48**–**51**
SinxOxRep, **48**–**51**
SphereProjCurve<N,F>, **28**
Sqrt<>, 3
Sqrt<F>, 2–4, 7, 9, **9**
Standard Template Library, 2, 22, 36
StandardRangeCurve<N,V,F>, **40**
std classes
cerr, 5, 31, 33
complex<double>, 2, 9, 54, 56, 58
complex<F>, 11, 59
complex<float>, 2, 9, 54, 56, 58
complex<long double>, 9, 54, 56, 58
cout, 12, 30, 33
istream, 63, 67
ostream, 63, 67
string, 73
vector<CompCurveType>, 18
vector<V>, 22
surfaces, 29–33
axisymmetric, 31–33

Tan<F>, **10**
Tanh<F>, **12**
TransFinite2dCurve<V,F>, **34**–**37**
TransFinite3dCurve<V,F>, **35**–**37**
trigonometric functions, 10–11

unit vector curves, 26
Unit<N,F>, **15**
UnitCurve<N,V,F>, **26**, 28

ValueType, 2
VecMtx classes
    MtxN<2U,F>, 29
    MtxN<N,F>, 62, **65**–**67**
    SingularMatrix, 66
    VecN<2U,F>, 32
    VecN<3U,F>, 24–29, 31–33
    VecN<N,F>, 2, 18, 23, 37, 42, 43, 45, 48, 54, 59, **61**–**63**, 65
Vector Object, 2, 22, 25, 26, **57**–**58**, 61, 62
vector valued curves, 22–28

Distribution list

DRDC Atlantic TM-2006-254

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 I: Multi-parameter differentiable functions

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.)	6a. NO. OF PAGES (Total containing information. Include Annexes, Appendices, etc.)	6b. NO. OF REFS (Total cited in document.)
January 2007	90	7

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.)	9b. GRANT OR CONTRACT NO. (If appropriate, the applicable number under which the document was written.)
11cj18	

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.)	10b. OTHER DOCUMENT NO(s). (Any other numbers which may be assigned this document either by the originator or by the sponsor.)
DRDC Atlantic TM-2006-254	

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.)

A library of C++ classes for representing multi-parameter differentiable functions is described. The principal utility of the classes lies in the ability to combine simple curves in a variety of ways to make complex curves while maintaining the differentiability of the result. This can be done using arithmetic functions, composition, vector operators (e.g. dot and cross products) and inverse methods.

The classes also include a wide variety of simple curves which can be used as building blocks, including constants, linear curves, polynomials, exponential functions, trigonometric functions, hyperbolic functions and Bessel functions.

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.)

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