Copy No. _____

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

DEFENCE R&D DÉFENSE

# C++ classes for representing propeller geometry

*David Hally*

**Defence R&D Canada – Atlantic**

Technical Memorandum
DRDC Atlantic TM 2013-177
October 2013

Canada

This page intentionally left blank.

# C++ classes for representing propeller geometry

David Hally

**Defence Research and Development Canada – Atlantic**
Technical Memorandum
DRDC Atlantic TM 2013-177
October 2013

# Abstract

To satisfy the need for high fidelity representations of propellers for use in Reynolds-averaged Navier-Stokes (RANS) flow solvers and other propeller applications, a library of C++ classes has been developed. It provides classes representing the surfaces of the propeller blades, hubs and ducts in fully differentiable form. The propeller blades can be defined using the traditional method of specifying the blade section shape, chord length, pitch, skew and rake at a series of radial sections. More general blade shapes are also possible provided that they conform to fairly loose requirements on the blade parameterization. Simple cylindrical and cigar-shaped hubs can be defined as well as more general axisymmetric shapes defined from splined offsets. Similarly, propeller ducts can be defined from commonly used duct cross-sections or more general shapes defined from offsets.



# Résumé

On a créé une bibliothèque de classes C++ afin de satisfaire le besoin de simulations à haute fidélité des hélices, aux fins d'utilisation dans des solutionneurs d'écoulement d'analyse d'équations de Navier-Stokes à moyenne de Reynolds (RANS) et dans d'autres applications relatives aux hélices. Cette bibliothèque comprend des classes qui représentent les surfaces des pales, des moyeux et des gaines des hélices sous forme entièrement différentiable. Les pales des hélices peuvent être définies à l'aide de la méthode classique de spécification de la forme, de la longueur de corde, du pas, de l'asymétrie et de l'inclinaison des pales à une série de sections radiales. Des pales de forme plus générale sont également possibles, pourvu qu'elles soient conformes aux exigences peu strictes concernant le paramétrage des pales. Des moyeux de forme cylindrique simple ou en forme de cigare peuvent être définis, ainsi que des formes plus asymétriques définies à partir de déviations cannelées. De même, les gaines d'hélice peuvent être définies à partir de sections transversales de gaine couramment utilisées ou des formes plus générales définies à partir de déviations.

This page intentionally left blank.

# Executive summary

## C++ classes for representing propeller geometry

David Hally; DRDC Atlantic TM 2013-177; Defence Research and Development Canada – Atlantic; October 2013.

**Background:** The design of propellers affects ship performance in many ways including maximum speed, fuel consumption, wear on shafts and machinery, on-board vibrations and radiated noise. The evaluation of propeller designs will be an important part of the projects to acquire the Arctic/Offshore Patrol Ship (AOPS), the Joint Support Ship (JSS) and other vessels for the Royal Canadian Navy.

**Principal results:** A library of C++ classes has been written to allow the representation of propeller geometry in applications for analyzing propeller performance. The classes allow an accurate description of propellers defined in a variety of ways including the traditional method of a series of transformed blade sections.

**Significance of results:** The classes are already in use in propeller applications developed by DRDC and international partners. These applications are being applied by DRDC to support the design of the propellers for the new ships in the Royal Canadian Navy.

**Future work:** If the need arises, the library of classes may be extended to include features of the propeller geometry that are currently not modelled: e.g. fillets, blade overhangs and the palms of controllable pitch propellers.

# Sommaire

## C++ classes for representing propeller geometry

David Hally ; DRDC Atlantic TM 2013-177 ; Recherche et développement pour la défense Canada – Atlantique ; octobre 2013.

**Contexte :** Le modèle des hélices a une incidence sur le rendement du navire en raison de différents aspects, y compris la vitesse maximale, la consommation de carburant, l'usure des arbres et des machines, les vibrations à l'intérieur, ainsi que le bruit rayonné. L'évaluation des modèles d'hélice fera partie intégrante de l'acquisition d'un navire de patrouille extracôtier de l'Arctique (NPEA), d'un navire de soutien interarmées (NSI) et d'autres navires pour la Marine royale canadienne.

**Principaux résultats :** Une bibliothèque de classes C++ a été créée, afin de permettre la présentation de la géométrie des hélices dans des applications d'analyse du rendement des hélices. Les diverses classes permettent d'obtenir une définition précise des hélices de diverses façons, y compris grâce à la méthode classique d'une série de sections de pales transformées.

**Importance des résultats :** Les classes sont déjà utilisées dans des applications relatives aux hélices mises au point par RDDC et des partenaires internationaux. Ces applications sont utilisées par RDDC afin d'appuyer la conception des hélices des nouveaux navires de la Marine royale canadienne.

**Perspectives :** Au besoin, on peut agrandir la bibliothèque de classes afin d'y inclure des caractéristiques de la géométrie des hélices qui ne sont pas modélisées actuellement : p. ex. les congés de raccordement, le surplomb des pales et les palmes des hélices à pas réglable.

# Table of contents

# List of figures

This page intentionally left blank.

# 1 Introduction

For many years DRDC has developed and used programs for the analysis of flows past propellers beginning with lifting line models, then lifting surface models, then, most recently using Reynolds-averaged Navier-Stokes (RANS) solvers. Each succeeding refinement in the method of analysis has required greater fidelity in the representation of the geometry of the propeller. RANS solvers need the geometry to be defined to the same level of accuracy used for its manufacture.

To satisfy the need for high fidelity representations of propellers to be used in RANS flow solvers and other propeller codes, a library of C++ classes has been developed. It uses the CurveLib library [1] to represent the surfaces of the propeller blades, the hub and optional ducts in fully differentiable form. The propeller blades can be defined using the traditional method of specifying the blade section shape, chord length, pitch, skew and rake at a series of radial sections. More general blade shapes are also possible provided that they conform to fairly loose requirements on the blade parameterization.

For the present, fillets, blade overhangs and palms of controllable pitch propellers are not modelled. This will cause some loss of fidelity in predicting flows near the blade roots. If, for example, it is important to predict blade root cavitation, it may be necessary to modify the classes to add these features.

The propeller surfaces can be used in any C++ application for which a representation of the propeller is needed. For example, the classes are used to represent the propeller geometry in the program Provise [2,3] developed by Cooperative Research Ships as a front end to its propeller analysis code PROCAL [4,5].

Another application developed at DRDC, `smooth-prop` [6,7], uses the propeller classes to write a propeller geometry to a file in IGES format which can then be used any of the major commercial RANS solvers to generate suitable grids for flow calculations. For cases in which the propeller blades are poorly defined near the tip, a common occurrence when they are defined by the traditional method, `prop-smooth` can also modify the geometry near the blade tip to ensure that it is of sufficient quality for the RANS solver.

# 2 Fundamentals

## 2.1 Auxiliary classes

The propeller geometry classes make use of several auxiliary classes and types that are defined in other libraries.

`Float`
: A floating point number. Defined in the header file `BasicTypes.h` to be `double`.

`Angle<Float>`
: An angle whose value is stored as a `Float`. The value of the angle can be set or obtained using degrees or radians. The class also allows a full range of arithmetic and trigonometric functions. It is described in detail in Ref. 1, Annex E.

`Str`
: A character string. `Str` is an alias for `std::string`.

`VecMtx::VecN<N,Float>`
: A vector of length `N` each of whose components is represented as a `Float`. The class allows a full range of arithmetic functions. It is described in detail in Ref. 1, Annex B.

## 2.2 Namespaces

The propeller classes and functions are included in namespace `PGeom`. In the sections that follow, if a namespace is not explicitly specified, then the class or function is in `PGeom`.

The propeller classes use several libraries of classes which also define namespaces: all classes in the CurveLib library (see Sec. 3.1) are in namespace `CurveLib`; all classes in the Airfoil library (see Sec. 3.2) are in namespace `Afoil`; all classes in the OFFSRF library (see Sec. 3.3) are in namespace `Offsrf`.

The vector class `VecN<N,Float>`, described in the previous section, is in namespace `VecMtx`. Because aliases are provided for `VecN<N,Float>` (e.g. `XYZPoint` and `CylPoint` defined in Sec. 4), it is rarely necessary to use namespace `VecMtx` explicitly.

## 2.3 Exceptions

All exceptions thrown by the propeller geometry classes and functions are derived from the base class `Error`; it is *not* in the namespace `PGeom`. An `Error` contains a

message which can be retrieved, appended to, or prepended to. The prototypes of the `Error` member functions are listed in Ref. 1, Annex F.

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

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

  ... // Code to handle the error
}
```

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 user error. The occurrence of a `ProgError` is an indication that the program is faulty. The prototypes of the `ProgError` member functions are listed in Ref. 1, Annex F.1.

## 2.4 Warnings

The propeller geometry classes may also send warning messages. These are considered less serious than exceptions and will not normally interrupt the flow of execution of the program. It is up to the programmer to decide what actions to take when a warning message is received. This is done by defining a *warning handler*, an instance of the class `WarningHandler`:

```
class WarningHandler
{
public:
  WarningHandler() { }
  virtual ~WarningHandler() { }
  virtual void handle(const Str &msg);
};
```

The member function `handle` is used to do something with the warning message which it receives in its argument `msg`. The default version writes the warning to `std::cerr` preceded by `WARNING!!`.

To set a new warning handler, use the function

```
WarningHandler& set_warning_handler(WarningHandler &wh);
```

It returns a reference to the warning handler currently in effect. Alternatively you can use a `WarningSentry`, a sentry class which set a new warning handler when it is constructed, then reverts to the previously defined warning handler when it is deleted. It has the benefit that it will reset the warning handler even when an exception is thrown. Use it to set a new warning handler, `wh`, as follows:

```
{ // New scope
  WarningSentry wsentry(wh); // Sets the warning handler to wh
  . . .
} // wsentry destroyed; revert to previous warning handler
```

For more on sentry classes, see Stroustrup [8].

The class `IgnoreWarningHandler` is a specialization of `WarningHandler` which redefines `handle` to do nothing; the warning is ignored.

## 2.5 The propeller accuracy

Some aspects of the propeller geometry cannot be calculated exactly; they can only be determined within a given tolerance: for example, the line of intersection between a propeller blade and the hub. To ensure that all such geometry is determined with consistent tolerances, a *propeller accuracy* is defined and designated by $\epsilon$. It is a floating point number normalized using the propeller diameter. If, for example, $\epsilon$ is set to $10^{-4}$, and you request a curve, $c(\xi)$, representing the blade-hub intersection curve, then a point returned by $c(\xi)$ will lie no more than $\epsilon D$ from the true intersection line.

The propeller accuracy is set or obtained using the following two functions:

```
void set_prop_accuracy(Float acc);
Float get_prop_accuracy();
```

The sentry class `PropAccSentry` has also been defined to make an exception proof method of altering the propeller accuracy. The accuracy is reset to its previous value when the sentry is deleted.

The `PropAccSentry` constructor has a single argument, the accuracy to be set:

```
PropAccSentry(Float acc);
```

For example, if you want to evaluate points along the blade-hub intersection to an accuracy of $10^{-8}D$, you could use the following:

```
using namespace PGeom;
Propeller p;
Blade blade = p.get_blade();
{
  PropAccSentry sentry(1.0e-08);
  SurfaceCurve intersection = blade(blade_param_curve_at_hub());
  int npts = 11;
  for (int i = 0; i < npts; ++i) {
    Float xi = Float(i)/Float(npts-1);
    std::cout << xi << ' ' << intersection(xi) << '\n';
  }
} // Propeller accuracy returned to original value here
  // as sentry is destroyed.
```

The function `blade_param_curve_at_hub` is described in Sec. 9.1.6. Notice that the outer level of curly brackets delimits the scope of `sentry` so that the propeller accuracy is returned to its original value once the loop is finished. This will also be the case if an exception is thrown during the evaluation of the intersection curve.

# 3 Class libraries used by the propeller classes

The propeller classes make use of three other class libraries which are described briefly in the following sections. Each has been documented fully in other DRDC reports [1,9–11].

## 3.1 The CurveLib library

The propeller geometry classes are based on the CurveLib library of classes for representing differentiable curves and surfaces [1]. A brief overview is presented here but, for a proper understanding of the following sections, it is recommended that Refs. 1 and 9 be read first.

The fundamental class of the CurveLib library is `Curve<N,V,F>`. It is a function object (a class that behaves like a function) which represents an `N` parameter function which returns an object of type `V`. Each parameter is of type `F` which, in the propeller classes, is always `Float`. Since all CurveLib library functions are in namespace `CurveLib`, in this section we will omit the namespace.

The parameter list of a CurveLib function object is a `Curve<N,V,F>::ParamType` which is simply an alias for `VecMtx::VecN<N,F>`. The function is evaluated using the operator

```
V Curve<N,V,F>::operator()(ParamType p) const;
```

For example:

```
using namespace CurveLib;
Curve<2U,Float,Float> f = Abs<2U,Float>();
Curve<2U,Float,Float>::ParamType p(1.0,2.0);
Float absf = f(p); // absf = sqrt(5.0)
```

The CurveLib function object `Abs<N,F>` considers its argument list to be a vector and returns the magnitude of that vector.

CurveLib function objects are fully differentiable in each of their parameters. The derivatives of a function can be evaluated using the operator

```
V Curve<N,V,F>::operator()(ParamType p, DerivType d) const;
```

where `DerivType` is an alias for `Derivs<N>`, a class representing a list of unsigned integers: see Ref. 1, Sec. 2. For example:

```
using namespace CurveLib;
Curve<1U,Float,Float> sin_crv = Sin<Float>();
Sin<Float>::ParamType p(0.0);
Sin<Float>::DerivType d(0);
Float dsin0 = sin_crv(p,d); // dsin0 = cos(0.0) = 1.0
```

The return type, `V`, is required to support a full set of arithmetic operators (see Ref. 1, Annex A.1). The curve classes themselves will also support most arithmetic functions: see Ref. 1, Sec. 4. Therefore, if `f` and `g` are both of type `Curve<N,V,F>`, then so is `2*f+3*g`.

CurveLib function objects can also be composed. Suppose that `f` is an instance of `Curve<N,V,F>`. If `g` is an `M` parameter function which returns a parameter list for `f` (i.e. it is of type `Curve<M,VecMtx::VecN<N,F>,F>`) then `f(g)` is an `M` parameter function which returns a `V` (i.e. it is of type `Curve<M,V,F>`). Details are provided in Ref. 1, Sec. 5.

### 3.1.1 One-parameter function objects

An important specialization of a `Curve<N,V,F>` is when `N` is 1. Then the function can be evaluated using an `F` instead of a `Curve<1U,V,Float>::ParamType`. Similarly, derivatives can be specified using a single unsigned integer rather than a `Curve<1U,V,Float>::DerivType`. For example:

```
using namespace CurveLib;
Curve<1U,Float,Float> sin_crv = Sin<Float>();
Float sin0   = sin_crv(0.0);   // sin0   =  sin(0.0) = 0.0
Float dsin0  = sin_crv(0.0,1); // dsin0  =  cos(0.0) = 1.0
Float d2sin0 = sin_crv(0.0,2); // d2sin0 = -sin(0.0) = 0.0
```

### 3.1.2 Implicitly polymorphic member functions

One aspect of the CurveLib classes that is inherited by the propeller geometry classes deserves special mention: although most member functions are not explicitly declared `virtual`, the often behave as if they are. We call this being implicitly polymorphic. For example, the class `Blade` used to represent a propeller blade has the member function `get_pitch_at_radius_curve` which returns a CurveLib function object defining the pitch distribution as a function of non-dimensional radius $r$. `Blade` has a fairly inefficient version of this function described in Secs. 6.1.4 and 6.1.5. On the other hand, the class `SectionBlade`, a specialization of `Blade`, has a much more efficient version of the function. Consider the following code in which a `SectionBlade` is first assigned to a blade, then the pitch curve is obtained and evaluated:

```
using namespace PGeom;
SectionBlade sblade;
 . . . // Code to define sblade
Blade blade = sblade;
Blade::ScalarCurveType pcrv = blade.get_pitch_at_radius_curve();
Float p = pcrv(0.5);
```

Which version of the pitch curve is used when `pcrv` is evaluated? It is the efficient version obtained from `sblade` because the function `get_pitch_at_radius_curve` is implicitly polymorphic.

## 3.2 The Airfoil library

Propeller blades are commonly defined by specifying a series of blade sections in the shape of airfoils. The propeller classes make use of the Airfoil class library to represent the blade sections. It defines classes for representing airfoils using CurveLib classes and has been documented in Ref. 10.

The principal class in the Airfoil library is `Afoil::Airfoil<F>` which represents a generic airfoil. When used by the propeller classes, the template parameter `F` representing the type of a floating point number will always be `Float`. The propeller classes also define the type `AirfoilPt` to represent a point on an airfoil; it is equivalent to a `VecMtx::VecN<2U,Float>`. `Afoil::Airfoil<Float>` is a specialization of `CurveLib::Curve<1U,AirfoilPt,Float>`: that is, it is a one-parameter function object returning a 2-vector representing a point on the airfoil surface. The point is defined in a two-dimensional Cartesian coordinate system in which the leading edge is normally (0,0) and the trailing edge (1,0).

The parameter of the airfoil function object, usually denoted $\xi$, is 0.0 at the trailing edge on the pressure side of the airfoil, 0.5 at the leading edge, and 1.0 at the trailing edge on the suction side.

There are several classes derived from `Airfoil<Float>` which allow the airfoil to be defined from offsets, from thickness distributions and mean line offset curves, or from NACA designations [12].

## 3.3 The OFFSRF library

The propeller classes allow a propeller to be defined by reading a specification from a file. The OFFSRF library is used to provide a consistent record structure to the files as well as standard C++ inserters for reading them. It is documented in Ref. 11.

# 4 Coordinate systems

Propeller blades are defined relative to a *propeller reference line* which extends radially perpendicular to the axis of rotation. The propeller is defined in a Cartesian coordinate system, $(X, Y, Z)$, having its origin where the propeller reference line meets the axis of rotation. The $Z$ coordinate increases along the axis of rotation toward the aft of the ship, the $Y$ coordinate increases outward from the axis of rotation along the propeller reference line, and the $X$ coordinate is perpendicular to $Y$ and $Z$ such that the coordinate system is right-handed: see Fig. 1. A point in the $(X, Y, Z)$ coordinate system is represented by the type `XYZPoint` which is simply an alias for `VecMtx::VecN<3U,Float>`.

A coordinate system scaled using the propeller diameter is also used. It is denoted $(x, y, z)$:

$$x = X/D; \quad y = Y/D; \quad z = Z/D \tag{1}$$

where $D$ is the propeller diameter. A point in the $(x, y, z)$ coordinate system is represented by the type `Point` which is simply an alias for `VecMtx::VecN<3U,Float>`.

A cylindrical coordinate system, $(r, \theta, z_R)$ is also used; it will often be called the hub coordinate system as it is most convenient for defining the hub geometry. It defines $\theta$ to be zero along the propeller reference line ($y = 0$) and increasing counterclockwise around the propeller axis when looking forward. This direction for $\theta$ is in accordance with the ITTC conventions for measuring propeller angles (e.g. skew angle or rake angle; see [13]). The coordinates $r$ and $z_R$ are non-dimensionalized *with respect to the propeller radius, R*; therefore

$$x = -\tfrac{1}{2} r \sin\theta; \quad y = \tfrac{1}{2} r \cos\theta; \quad z = \tfrac{1}{2} z_R \tag{2}$$

and

$$r = 2\sqrt{x^2 + y^2}; \quad \theta = -\arctan(x/y); \quad z_R = 2z. \tag{3}$$

The inconsistency in non-dimensionalization is unfortunate, but conforms to traditional methods. For example, the propeller properties pitch, rake, etc. are traditionally non-dimensionalized using the diameter while blade sections are identified using $r$: i.e. a radial location non-dimensionalized using the propeller radius.

A point in the $(r, \theta, z_R)$ coordinate system is represented by the type `CylPoint` which is simply an alias for `VecMtx::VecN<3U,Float>`. The value of $\theta$ is given in radians.

## 4.1 Classes supporting coordinate transformations

The class `CartesianToCylCoords` converts a Cartesian point, $(x, y, z)$, to cylindrical hub coordinates $(r, \theta, z_R)$ using Eq. (3). It is a specialization of the base class

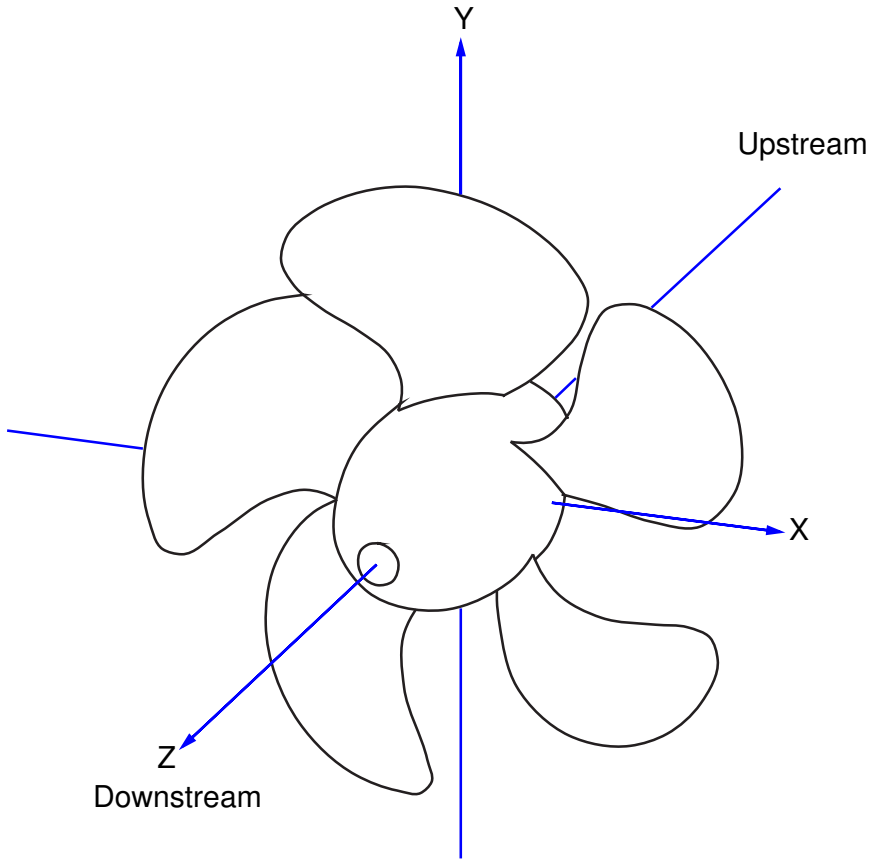



***Figure 1:*** *The Cartesian coordinate system.*

`CurveLib::Curve<3U,CylPoint,Float>` and inherits all the properties of a CurveLib function object (e.g. it is fully differentiable). It has a default constructor and a copy constructor and defines no member functions other than those inherited from its base class.

```
using namespace PGeom;
CartesianToCylCoords xyz_to_rtz;
Point xyz(1.0,2.0,3.0);
CylPoint rtz = xyz_to_rtz(xyz); // rtz = (4.472136,-0.4636476,6.0)
```

The class `CartesianToRadius` converts a Cartesian point $(x, y, z)$ to the cylindrical coordinate $r$. It is a specialization of `CurveLib::Curve<3U,Float,Float>` and inherits all the properties of a CurveLib function object. It has a default constructor and a copy constructor and defines no member functions other than those inherited from its base class.

Using `CartesianToRadius` is more efficient than using `CartesianToCylCoords` and selecting the $r$ coordinate of the result.

```
using namespace PGeom;
CartesianToRadius xyz_to_r;
Point xyz(1.0,2.0,3.0);
Float r = xyz_to_r(xyz); // r = 4.472136
```

The class `CylToCartesianCoords` converts a point in cylindrical coordinates, $(r, \theta, z_R)$, to Cartesian coordinates, $(x, y, z)$, using Eq. (2). It is a specialization of the base class `CurveLib::Curve<3U,Point,Float>` and inherits all the properties of a CurveLib function object. It has a default constructor and a copy constructor and defines no member functions other than those inherited from its base class.

```
using namespace PGeom;
CylToCartesianCoords rtz_to_xyz;
CylPoint rtz(1.0,2.0,3.0);
Point xyz = rtz_to_xyz(rtz); // xyz = (-0.4546487,-0.2080734,1.5)
```

The following function for transforming from propeller coordinates to a hull coordinate system is also defined:

```
TransformType prop_to_hull_coordinates(const XYZPoint &porigin,
                                       const XYZPoint &paxis);
```

It returns a transformation that will convert a point in propeller coordinates to a point in hull coordinates. The parameter `porigin` is the origin of the propeller coordinates in the hull coordinate system and `paxis` is an aft-facing vector in the hull coordinate system tangent to the propeller axis. `TransformType` is an alias for the class `IGES::Transformation` described in Ref. 14, Sec. 4.

This function assumes that the propeller reference line has no $Y$ component in the hull coordinate system and that the hull $Z$ coordinate increases upwards. The hull $X$ coordinate runs either from bow to stern or from stern to bow; the direction is determined from the sign of the $X$ component of the aft-facing normal. Then if $\mathbf{X}_h$ and $\mathbf{X}_p$ are points in the hull and propeller coordinate systems, respectively, then:

$$\mathbf{X}_h = \mathbf{X}_0 + \mathbf{M}\mathbf{X}_p \tag{4}$$

where $\mathbf{X}_0$ is the origin of the propeller coordinates in the hull coordinate system (the point where the propeller reference line crosses the propeller axis), $\hat{n} = (n_X, n_Y, n_Z)$ is an aft-facing unit vector tangent to the propeller axis, and

$$\mathbf{M} = \begin{bmatrix} \dfrac{-|n_X| n_Y}{\sqrt{n_X^2 + n_Z^2}} & \dfrac{-\mathrm{sgn}(n_X) n_z}{\sqrt{n_X^2 + n_Z^2}} & n_X \\ \sqrt{n_X^2 + n_Z^2} & 0 & n_Y \\ \dfrac{\mathrm{sgn}(n_X) n_Y n_Z}{\sqrt{n_X^2 + n_Z^2}} & \dfrac{|n_X|}{\sqrt{n_X^2 + n_Z^2}} & n_Z \end{bmatrix} \tag{5}$$

# 5 Surfaces

Each propeller is an aggregate of surfaces on each blade and on the hub. Each propeller surface is a function of two parameters, $(\xi, \eta)$, which returns a point in space (an $(x, y, z)$ point). The parameters are represented by the class `SurfaceParam` which is simply an alias for `VecMtx::VecN<2U,Float>`. The returned value is a `Point` (see Sec. 4). Each surface is a specialization of the class `Curve<2U,Point,Float>` from the CurveLib library: see Ref. 1, Sec. 8. For ease of use this class is given the alias `Surface` within the namespace `PGeom`.

Surfaces can be evaluated using either of the following operators:

```
Point Surface::operator()(SurfaceParam p) const;
Point Surface::operator()(Float xi, Float eta) const;
```

For example, in the following code to obtain a point on the leading edge near the tip of the reference blade,

```
using namespace PGeom;
Propeller p;
Surface blade = p.get_blade();

SurfaceParam p(0.5,0.95);
Point x1 = blade(p);
Point x2 = blade(0.5,0.95);
```

the values of `x1` and `x2` will be the same.

Derivatives of the surface with respect to its parameters can be evaluated using

```
Point Surface::operator()(SurfaceParam p, DerivType d) const;
Point Surface::operator()(Float xi, Float eta,
                          unsigned dxi, unsigned deta) const;
```

where `DerivType` (an alias for `CurveLib::Derivs<2U>`) is a 2-vector of unsigned integers given the number of derivatives to be taken with respect to $\xi$ and $\eta$. For example, you can calculate a normal to the reference blade at $(\xi, \eta) = (0.2, 0.3)$ as follows:

```
using namespace PGeom;
Propeller p;
Surface blade = p.get_blade();

SurfaceParam p(0.2,0.3);
Surface::DerivType dxi(1,0), deta(0,1);
Point norm1 = cross_product(blade(p,dxi),blade(p,deta));
Point norm2 = cross_product(blade(0.2,0.3,1,0),blade(0.2,0.3,0,1));
```

The values of `norm1` and `norm2` will be the same.

However, the following functions provide a simpler method of obtaining outward-pointing unit normal to a surface:

```
Point Surface::normal(SurfaceParam p) const;
Point Surface::normal(Float xi, Float eta) const;
```

Moreover, these functions will sometimes return a well-defined normal even when one of the surface derivatives vanishes. This is the case, for example, at the end points of a cigar-shaped hub (see Sec. 7.1.2). Therefore the preceding code is better replaced by

```
using namespace PGeom;
Propeller p;
Surface blade = p.get_blade();

SurfaceParam p(0.2,0.3);
Point norm1 = blade.normal(p);
Point norm2 = blade.normal(0.2,0.3);
```

The ranges of the surfaces parameters are stored in variables of type `RangeType`, an alias for `CurveLib::ParamRange<2U,Float>`. The `ParamRange` class is described in Ref. 1, Sec. 10.1.

Like any CurveLib function object, a surface can exist before it is properly defined. Most member functions will throw an `Error` if called before the surface is defined. The member function

```
bool is_defined() const;
```

returns `true` if the surface is defined.

## 5.1 Surface curves

Curves on a propeller surface are usually defined in the parameter space of the surface: i.e. they are one-parameter curves which return a `SurfaceParam`. Surface curves are represented by the class `CurveLib::Curve<1U,SurfaceParam,Float>`; for ease of use this class is given the alias `SurfaceCurve`.

Because they are one-parameter function objects, surface curves can be evaluated using a parameter of type `Float` and derivatives can be specified using a single unsigned integer: see Sec. 3.1.1.

For example, the intersection of the reference blade and hub can be viewed as a surface curve lying in either the blade surface or the hub surface. The `Propeller` class has member functions to return either representation (see Sec. 9.1.6). We could compare the values returned by the two representations by sampling them as follows:

```
using namespace PGeom;
Propeller p;
Blade blade = p.get_blade();
Hub hub = p.get_hub();
SurfaceCurve blade_crv = p.blade_param_curve_at_hub();
SurfaceCurve hub_crv = p.hub_param_curve_at_blade();

int npts = 11;
for (int i = 0; i < npts; ++i) {
  Float xi = Float(i)/Float(npts-1);
  SurfaceParam bparam = blade_crv(xi), hparam = hub_crv(xi);
  Point bx = blade(bparam), hx = hub(hparam);
  Float diff = abs(bx-hx);
  std::cout << xi << ' ' << diff << '\n';
}
```

A surface curve can be converted to a curve in space (i.e. a curve returning $(x, y, z)$ points) by composing it with the surface in which it lies. For example, a curve that returns the points on the hub/blade intersection could be generated using:

```
using namespace PGeom;
Propeller p;
Blade blade = p.get_blade();
SurfaceCurve blade_crv = p.blade_param_curve_at_hub();
CurveLib::Curve<1U,Point,Float> intersection_crv = blade(blade_curve);
```

If a surface curve is evaluated before it is properly defined, an `Error` will be thrown. To determine if the surface is defined use the member function

```
bool is_defined() const;
```

which returns `true` if the surface curve is well-defined.

# 6 Classes representing propeller blades

The blades of a propeller are numbered 0 to $N-1$ with blade 0 being known as the reference blade. Only the reference blade needs to be defined, the others then being generated by rotated copies. The reference blade is defined with respect to the propeller reference line in the $(x, y, z)$ coordinate system: see Sec. 4.

There are two principal classes for representing the reference blade: the base class `Blade` which has a minimal set of restrictions on the blade surface, and the class `SectionBlade` which generates the blade from a series of sections in the traditional way. They are described in the following sections.

## 6.1 The base class Blade

The class `Blade` represents the reference blade of a propeller. It is a specialization of `Surface` (see Sec. 5). Since a `Surface` returns points in the $(x, y, z)$ coordinate system, a `Blade` is non-dimensionalized using the propeller diameter. This means that its radius at the tip will normally be 0.5 (though the radius may change if the blade is rotated around the propeller reference line: see Sec. 6.1.6). The non-dimensionalization makes it easy for the propeller classes to scale the blades, for example from model scale to full scale.

### 6.1.1 Blade parameters

We will let $\mathbf{b}(\xi, \eta)$ denote the function returning a point on the blade surface (i.e. the function represented by the `Blade` class itself). For a right-handed blade (one which rotates clockwise as seen from behind looking forward), the $\xi$ parameter increases from 0.0 at the trailing edge on the downstream face, to 0.5 at the leading edge, to 1.0 at the trailing edge on the upstream face. On a left-handed blade the direction of $\xi$ is reversed (i.e. $\xi = 0$ is on the upstream side) so that the direction of the normals remains outward-pointing.

The range of the parameter $\eta$ is not defined but it is assumed to increase from the root of the blade toward the tip. However, although it is usually the case that the maximum value of $\eta$ corresponds to the tip (i.e. the point on the blade with maximal $r$), it is not required. We wish to be able to mimic a controllable pitch propeller by rotating the reference blade about the propeller reference line and in that case the tip of the rotated propeller may shift along the leading or trailing edge to a point where $\eta$ is not a maximum.

The range of the parameter $\eta$ will be denoted $[\eta^{min}, \eta^{max}]$. It can be obtained using the member function

```
void eta_range(Float &eta_min, Float &eta_max) const;
```

The range of $\xi$ is always $[0, 1]$ so no corresponding function is necessary for it.

The tip of the blade is the point on the leading or trailing edge farthest from the propeller axis. As pointed out above, in general, the $\eta$ parameter at the tip need not be $\eta^{max}$. The following member function returns the parameters at the tip:

```
Blade::ParamType parameters_at_tip() const;
```

The $\xi$ value returned will always be 0.0, 0.5 or 1.0. The parameters at the tip will be denoted $(\xi^{tip}, \eta^{tip})$.

It is common to specify points on a blade using values of $r$, so it is convenient to be able to convert between $(\xi, \eta)$ and $(\xi, r)$. The following member function returns a CurveLib function object that returns the value of $r$ given $(\xi, \eta)$

```
CurveLib::Curve<2U,Float,Float> xi_eta_to_r() const;
```

It is easily implemented as the composition of the blade with a `CartesianToRadius` (see Sec. 4.1).

Obtaining the value of $\eta$ given $\xi$ and $r$ is more complex. It requires solving

$$\mathbf{b}_x(\xi, \eta)^2 + \mathbf{b}_y(\xi, \eta)^2 = \tfrac{1}{4} r^2 \tag{6}$$

for $\eta$ which can be done using a Newton-Raphson search implemented within a `CurveLib::ImplicitCurve<2U,1U,Float>` (see Ref. 1, Sec. 11.2). However, near the tip, there may be more than one solution to Eq. (6). For example, let $r^m$ be the value of $r$ at $(\frac{1}{2}, \eta^{max})$ and suppose that the tip lies on the leading edge (i.e. $\xi^{tip} = \frac{1}{2}$); then if $r \in [r^m, r^{tip}]$, then there will be two values of $\eta$ which satisfy Eq. (6), one with $\eta < \eta^{tip}$ and one with $\eta > \eta^{tip}$.

The following member functions both return a CurveLib function object that performs the conversion from $(\xi, r)$ to $(\xi, \eta)$:

```
CurveLib::MultiCurve<2U,2U,Float> xi_r_to_xi_eta() const;
CurveLib::MultiCurve<2U,2U,Float> xi_r_to_xi_eta(Float eta_lo,
                                                 Float eta_hi) const;
```

The difference between the two is that the first places no restrictions on $\eta$, so it may not be well-defined near the tip. The second restricts $\eta$ to have values between `eta_lo` and `eta_hi`; it can be used to remove the ambiguity in $\eta$ near the tip using judicious values of `eta_lo` and `eta_hi`. Sec. 6.1.4 gives examples of its use to determine $\eta$ as a function of $r$ on the leading edge.

The class `CurveLib::MultiCurve<2U,2U,Float>` is described in Ref. 1, Sec. 6.7. Its return value is a `VecMtx::VecN<2U,Float>`. It can be composed with the blade to change its parameterization to $(\xi, r)$.

```
using namespace PGeom;
Blade blade;
... // Define blade
CurveLib::Curve<2U,Point,Float>
  blade_at_xi_r = blade(blade.xi_r_to_xi_eta());
Point x = blade_at_xi_r(0.5,0.3); // Returns the (x,y,z) point
                                  // on the LE at r = 0.3
```

However, note that, in general, the lower and upper ranges for $r$ will be dependent on $\xi$ so one must be careful when evaluating `blade_at_xi_r` near the root and tip.

Because the object returned by `xi_r_to_xi_eta` is a `CurveLib::MultiCurve`, the curve for $\eta$ as a function of $(\xi, r)$ can be obtained by selecting the second component of the curve. For example:

```
using namespace PGeom;
Blade blade;
... // Define blade
CurveLib::Curve<2U,Float,Float>
  xir_to_eta = blade.xi_r_to_xi_eta()[1];
Float eta = xir_to_eta(0.5,0.3); // Returns eta on the LE at r = 0.3.
```

### 6.1.2 Constructors

A `Blade` can be constructed from any CurveLib surface with a given parameter range: i.e. a `CurveLib::RangeCurve<2U,Point,Float>` which is described in Ref. 1, Sec. 10.2. Within the `Blade` class, this is given the alias `RangeSurface`.

```
typedef CurveLib::RangeCurve<2U,Point,Float> RangeSurface;
```

The prototype for the constructor is then

```
Blade(RangeSurface srf);
```

The first parameter of `srf` will be scaled so that $\xi$ has the expected ranges for the blade. That is, if `srf` represents the surface $\mathbf{s}(u, v)$ with $u \in [u_{lo}, u_{hi}]$ and $v \in [v_{lo}, v_{hi}]$, then the blade surface will be

$$\mathbf{b}(\xi, \eta) = \mathbf{s}\Big(u_{lo} + \xi(u_{hi} - u_{lo}), \eta\Big). \tag{7}$$

Since $\xi = \frac{1}{2}$ is identified with the leading edge, this is the line in `srf` having $u = \frac{1}{2}(u_{lo} + u_{hi})$. It is up to the calling class to ensure that the normals to `srf` are outward pointing.

The following member function also allows one to redefine the blade surface using a `RangeSurface`:

```
void define(RangeSurface srf);
```

As is recommended for all of the CurveLib classes, `Blade` also has a default constructor. When it is used, the blade will not immediately be defined: it will be necessary to define it either by assigning it to another `Blade` or by calling the member function `define`. For example, if `blade` is an instance of any class derived from `Blade`, and `srf` is a `RangeSurface`:

```
using namespace PGeom;
Blade b1, b2, b3(srf); // b1 are b2 are undefined, b3 is defined.
b1 = b3;               // b1 now defined and equivalent to b3.
b2.define(srf);        // b2 now defined and equivalent to b3.
```

The member function `is_defined()` can be used to determine if a `Blade` is defined; it will return true if it is.

### 6.1.3 Blade properties

The handedness of the blade can be determined using the member function

```
bool is_right_handed() const;
```

which returns `true` for a right-handed blade.

The trailing edge of a blade section is sharp if the unit normal at the trailing edge on the suction side differs from the unit normal on the pressure side. A blade is considered to have a sharp trailing edge if any of its sections has a sharp trailing edge. The following member function returns true if the trailing edge is sharp:

```
bool has_sharp_trailing_edge() const;
```

The section of the blade with $\eta = \eta^{max}$ might not have zero thickness; in that case the blade is said to have an open tip. This is common, for example, in blades of ducted propellers. The following member function returns true if the blade has an open tip:

```
bool has_open_tip() const;
```

If all the normals to the blade along the section with $\eta = \eta^{max}$ are the same, then the blade surface is smooth at the tip. If some of the normals differ, the blade is said to be sharp. The following member function returns true if the blade has a sharp tip:

```
bool has_sharp_tip() const;
```

The returned values of the functions `has_sharp_trailing_edge`, `has_open_tip` and `has_sharp_tip` are determined by sampling the trailing edge or tip section at a series of points, then comparing values of the points or normals to ensure that they are within the propeller accuracy. For efficiency this is done once when the blade is constructed. Derived blade classes may use more efficient and/or accurate methods for determining the values.

### 6.1.4 The leading and trailing edges

If $\eta^{tip} = \eta^{max}$, then the leading edge is simply the curve $\mathbf{b}^{le}(\eta) = \mathbf{b}(\frac{1}{2}, \eta)$ and the trailing edge is the curve $\mathbf{b}^{le}(\eta) = \frac{1}{2}\Big(\mathbf{b}(0,\eta) + \mathbf{b}(1,\eta)\Big)$; the average of the points with $\xi = 0$ and 1 keeps a single trailing edge curve well-defined even if the trailing edge is open.

However, if $\eta^{tip} \neq \eta^{max}$, then when $\eta > \eta^{tip}$ the trailing edge may include points for which $\xi = \frac{1}{2}$ and similarly for the leading edge. The member functions described in this section are provided to make it easier to evaluate points on the true leading and trailing edges.

The true leading and trailing edges can be parameterized unambiguously using $r$, since it can be assumed that it will increase monotonically along these curves from the root of the blade to the tip. The principal task is then to provide a means for calculating the blade parameters $(\xi, \eta)$ which correspond to a given value of $r$ on either the leading or trailing edge. We describe the procedure for the leading edge; the trailing edge is similar.

1. If $\eta^{tip} = \eta^{max}$ or if $\xi^{tip} = \frac{1}{2}$, then the leading edge is simply $\mathbf{b}^{le}(\eta)$ with $\eta \in [\eta^{min}, \eta^{tip}]$. Set `eta_lo` to $\eta^{min}$ and `eta_hi` to $\eta^{tip}$ and obtain the function object returned by `xi_r_to_xi_eta(eta_lo,eta_hi)[1]` (see Sec. 6.1.1); it returns $\eta$ as a function of $(\xi, r)$. Restrict it to have $\xi = \frac{1}{2}$ (see the description of the CurveLib class `ConstPCurve<N,V,F>` in Ref. 1, Sec. 6.5) to generate a function object returning $\eta$ as a function of $r$ on the leading edge.

2. Otherwise, the leading edge has two segments: one is $\mathbf{b}^{le}(\eta)$ with $\eta \in [\eta^{min}, \eta^{max}]$ and the other is $\mathbf{b}^{te}(\eta)$ with $\eta \in [\eta^{tip}, \eta^{max}]$.

   (a) Construct a function object on the first segment which solves

   $$\mathbf{b}^{le}_x(\eta)^2 + \mathbf{b}^{le}_y(\eta)^2 = \tfrac{1}{4}r^2 \tag{8}$$

   for $\eta$ restricted to the range $[\eta^{min}, \eta^{max}]$. This can be done using the class `CurveLib::FInverseCurve<Float>` described in Ref. 1, Sec. 11.3. The domain of the function object will be $r \in [r^{min}, r^{max}]$ where $r^{min}$ is the value of $r$ for the point with $(\xi, \eta) = (\frac{1}{2}, \eta^{min})$ and $r^{max}$ is the value of $r$ for the point with $(\xi, \eta) = (\frac{1}{2}, \eta^{max})$.

   (b) Construct a function object on the second segment by solving

   $$\mathbf{b}^{te}_x(\eta)^2 + \mathbf{b}^{te}_y(\eta)^2 = \tfrac{1}{4}r^2 \tag{9}$$

   for $\eta$ restricted to the range $[\eta^{tip}, \eta^{max}]$. The domain of the function object will be $r \in [r^{max}, r^{tip}]$.

(c) Combine the two function objects into a single function object over the range $r \in [r^{min}, r^{tip}]$ using a `Spline::GeneralSpline<Float,Float>` (described in Ref. 9, Sec. 5).

Once $\eta$ is known as a function of $r$ along the leading and trailing edges, it can be composed with $\mathbf{b}^{le}(\eta)$ and $\mathbf{b}^{te}(\eta)$ to generate function objects which return points on the leading and trailing edges as a function of $r$. The following two member functions return these function objects.

```
CurveLib::Curve<1U,Point,Float> leading_edge_at_radius() const;
CurveLib::Curve<1U,Point,Float> trailing_edge_at_radius() const;
```

The allowed ranges for $r$ along the leading and trailing edges will, in general be different. They can be obtained using the following member functions:

```
void leading_edge_r_range(Float &r_lo, Float &r_hi) const;
void trailing_edge_r_range(Float &r_lo, Float &r_hi) const;
```

### 6.1.5 Blade section properties

The blade section at radius $r$ is the intersection of the blade surface with a cylinder of radius $r$ whose axis is the propeller axis. The properties of any blade section—chord length, pitch, etc.—can be calculated easily provided one knows the points at the leading and trailing edges of the section. These are provided by the member functions `leading_edge_at_radius` and `trailing_edge_at_radius` described in the previous section. They return function objects representing the functions $\mathbf{b}^{le}(r)$ and $\mathbf{b}^{te}(r)$ returning the points along the leading and trailing edges as a function of $r$.

The points on the leading and trailing edges can easily be converted to cylindrical coordinates (e.g. by composing a `CartesianToCylCoords` with the function objects returned by `leading_edge_at_radius` and `trailing_edge_at_radius`). The corresponding cylindrical coordinates will be denoted $\Big(r, \theta^{le}(r), z_R^{le}(r)\Big)$ and $\Big(r, \theta^{te}(r), z_R^{te}(r)\Big)$.

The chord length, $L(r)$, of the section is the distance between the leading and trailing edges along a line on the surface of the intersecting cylinder:

$$L(r) = \tfrac{1}{2} D \sqrt{\Big(z_R^{te}(r) - z_R^{le}(r)\Big)^2 + r^2 \Big(\theta^{te}(r) - \theta^{le}(r)\Big)^2}. \tag{10}$$

The following member function returns a CurveLib function object which evaluates $L(r)$:

```
Blade::ScalarCurveType chord_at_radius_curve() const;
```

where `ScalarCurveType` is an alias for `CurveLib::Curve<1U,Float,Float>`.

The pitch, $p(r)$, is the axial distance traversed by a helix through the leading and trailing edges as it traverses one revolution around the propeller axis:

$$p(r) = \pi D \left( \frac{z_R^{te}(r) - z_R^{le}(r)}{\theta^{te}(r) - \theta^{le}(r)} \right). \tag{11}$$

The pitch angle, $\phi(r)$, is the angle the pitch helix makes with the propeller plane. It is related to the pitch by

$$\phi(r) = \arctan\left(\frac{p(r)}{\pi D r}\right); \qquad p(r) = \pi D r \tan\Big(\phi(r)\Big). \tag{12}$$

The following member functions return CurveLib function objects which evaluate $p(r)$ and $\phi(r)$:

```
Blade::ScalarCurveType pitch_at_radius_curve() const;
Blade::AngleCurveType  pitch_angle_at_radius_curve() const;
```

where `AngleCurveType` is an alias for `CurveLib::Curve<1U,Angle<Float>,Float>`.

The skew angle, $\theta_s(r)$, is the angular displacement of the centre of the section from the propeller reference line.

$$\theta_s(r) = \tfrac{1}{2}\Big(\theta^{te}(r) + \theta^{le}(r)\Big). \tag{13}$$

The following member function returns a CurveLib function object which evaluates $\theta_s(r)$:

```
Blade::AngleCurveType skew_angle_at_radius_curve() const;
```

The generator line is the locus of points where the pitch helices intersect the plane containing the propeller reference line and the propeller axis (the plane $x = 0$). The generator rake, $i_G(r)$, is the axial displacement of the generator line at $r$ from the propeller reference line. The skew-induced rake, $i_S(r)$, is the axial displacement of the centre of the section at $r$ from the generator line at $r$. The total rake, $i_T(r)$, is the axial dispacement of the centre of the section at $r$ from the propeller reference line:

$$i_T(r) = \tfrac{1}{2} D\Big(z^{te}(r) + z^{le}(r)\Big); \tag{14}$$

$$i_S(\eta) = \tfrac{1}{2} D r \theta_s(r) \tan\Big(\phi(r)\Big) = \frac{p(r)\theta_s(r)}{2\pi}; \tag{15}$$

$$i_G(\eta) = i_T(\eta) - i_S(\eta). \tag{16}$$

The following member functions return CurveLib function objects which evaluate $i_G(r)$, $i_S(r)$ and $i_T(r)$:

```
Blade::ScalarCurveType generator_rake_at_radius_curve() const;
Blade::ScalarCurveType skew_induced_rake_at_radius_curve() const;
Blade::ScalarCurveType total_rake_at_radius_curve() const;
```

It is important to note that despite not being declared `virtual`, each function returning a blade section property is implicitly polymorphic (see Sec. 3.1.2). When the blade is defined in the traditional way by specifying these properties, the properties can be evaluated much more efficiently than by the methods described here: see Sec. 6.2.

### 6.1.6 Simulating a controllable pitch propeller

To simulate a controllable pitch propeller, the blade can be rotated around the propeller reference line (the $y$ axis) using the member function:

```
void set_rotation(const Angle<Float> ang);
```

The current value of the rotation can be obtained using:

```
Angle<Float> get_rotation() const;
```

The new reference blade is determined from the non-rotated blade by

$$\mathbf{b}_{new}(\xi,\eta) = \mathbf{M}\mathbf{b}(\xi,\eta) \tag{17}$$

where $\mathbf{M}$ is the rotation matrix

$$\mathbf{M} = \begin{pmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{pmatrix}. \tag{18}$$

Note that the values of the pitch, skew, etc. will be changed by the rotation. Immediately after construction or redefinition of the blade, the rotation around the propeller reference line will be zero.

### 6.1.7 An example blade

As a simple example of a blade constructed from an arbitrary surface, we define an ellipsoidal blade:

$$L(\eta) = L_{root}\cos(\tfrac{1}{2}\pi\eta) \tag{19}$$
$$x(\xi,\eta) = L(\eta)\cos(2\pi\xi) \tag{20}$$
$$y(\xi,\eta) = \tfrac{1}{2}\sin\Big(\tfrac{1}{2}\pi\eta\Big) \tag{21}$$
$$z(\xi,\eta) = -\tfrac{1}{2}tL(\eta)\sin(2\pi\xi) \tag{22}$$

where $L_{root}$ is the chord length at the root and $t$ is the thickness of each section relative to the chord length. Note that unlike the normal method of defining sections, these

ones are defined in planes of constant $y$ rather than on the surfaces of cylinders (surfaces of constant $r$).

This blade surface can be defined using the following code (the CurveLib classes are all described in Ref. 1):

```
Float Lroot = 0.1, t = 0.25, pi = Const::Pi<Float>::value();
Float h = 0.11, z = 0.2;
CurveLib::OneParamCurve<2U,Float> xi_crv(0), eta_crv(1);
CurveLib::Cos<Float> cos_crv;
CurveLib::Sin<Float> sin_crv;
CurveLib::Curve<2U,Float,Float> Lcrv = Lroot*cos_crv(0.5*pi*eta_crv);
CurveLib::MultiCurve<2U,3U,Float> base_srf;
base_srf[0] = Lcrv*cos_crv(2.0*pi*xi_crv);
base_srf[1] = 0.5*sin_crv(0.5*pi*eta_crv);
base_srf[2] = -0.5*t*Lcrv*sin_crv(2.0*pi*xi_crv);
```

If the blade is allowed to extend all the way to the propeller axis, numerical problems may occur (for example the lowest section will not project correctly onto a hub) so it is usual to restrict the range of $\eta$ so the lowest section is above the axis but beneath the surface of the hub. Here we restrict the range to $[0.1, 1]$:

```
CurveLib::ParamRange<2U,Float> range;
range.set_range(0,0.0,1.0);
range.set_range(1,0.1,1.0);
Blade::RangeSurface srf(base_srf,range);
Blade blade(srf);
```

As defined, the blade has zero pitch. To make some we rotate the blade about the reference line:

```
Angle<Float> ang;
ang.set_degrees(60.0);
blade.set_rotation(ang);
```

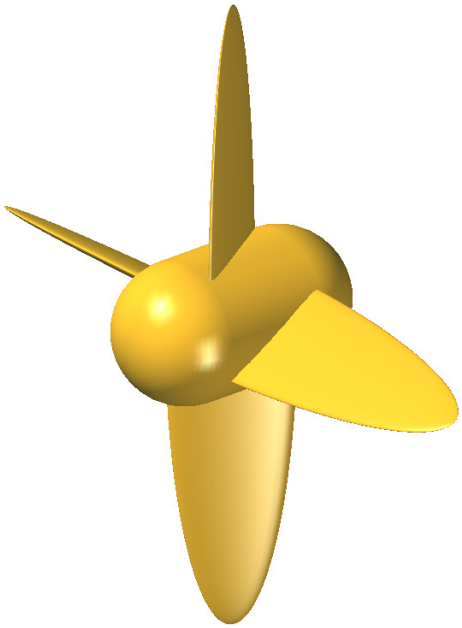


***Figure 2:*** *A propeller with ellipsoidal blades.*

Fig. 2 shows a propeller with four of these blades.

## 6.2 The class SectionBlade

Traditionally, propellers have been defined by specifying a series of blade sections (airfoils) at given radii. Each section is scaled, translated and rotated according to values of the chord length, pitch, rake and skew angle. Suppose that the blade

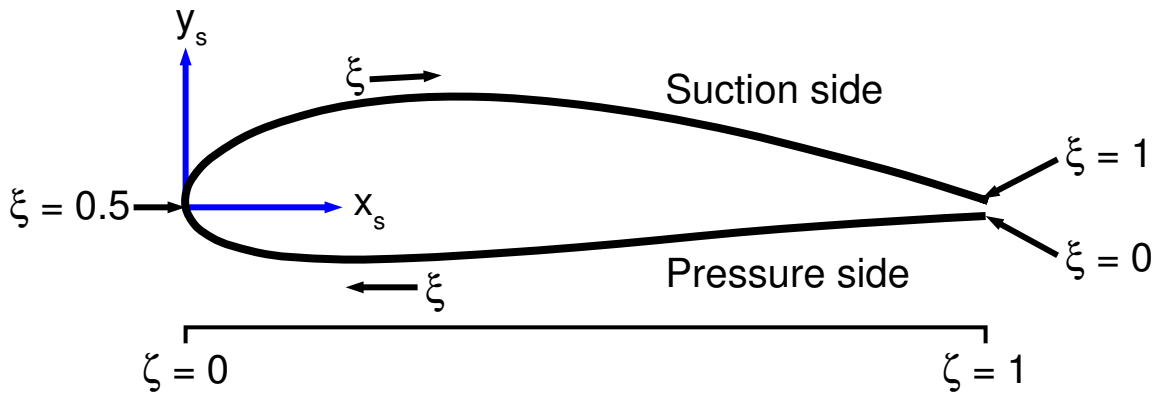



***Figure 3:*** *The coordinates on a reference section of a right-handed blade with an open trailing edge. On a left-handed blade the direction of $\xi$ is reversed.*

parameter $\eta$ is a function only of the distance to the axis, $r$. For any $\eta$, a reference section is defined as an airfoil, $\Big(x_s(\xi,\eta), y_s(\xi,\eta)\Big)$, where $x_s$ varies between 0.0 (leading edge) and 1.0 (trailing edge). In the parlance of the airfoil classes [10], the section is canonical: i.e. the leading edge is at $(x_s, y_s) = (0, 0)$ and the trailing edge is at $(1, 0)$. The coordinate $x_s$ is then equivalent to the fractional chord length along the airfoil, often denoted by $\zeta$. The airfoil coordinates are shown in Fig. 3.

The surface of a right-handed reference blade in the $(r, \theta, z_R)$ coordinate system is then defined by:

$$\theta(\xi,\eta) = \theta_s(\eta) + \frac{\Big(\Big(x_s(\xi,\eta) - \frac{1}{2}\Big)\cos\phi(\eta) + y_s(\xi,\eta)\sin\phi(\eta)\Big)}{r(\eta)}\frac{2L(\eta)}{D}; \tag{23}$$

$$z_R(\xi,\eta) = \frac{2i_T(\eta)}{D} + \Big(\Big(x_s(\xi,\eta) - \frac{1}{2}\Big)\sin\phi(\eta) - y_s(\xi,\eta)\cos\phi(\eta)\Big)\frac{2L(\eta)}{D}. \tag{24}$$

The blade in the Cartesian coordinate system can be determined using Eqs. (1) and (2). The remaining blades are determined by rotating the reference blade around the $\hat{z}$ axis in increments of $2\pi/N$.

A left-handed blade is obtained by replacing $\theta$ with $-\theta$. To ensure that $\hat{\xi} \times \hat{\eta}$ is an outward pointing normal on a left-handed blade, the parameter $\xi$ is also replaced by $1 - \xi$. Therefore, on a right-handed blade, the pressure side (face) is the region $\xi \in [0, 0.5]$; on a left-handed blade this region is the suction side (back).

The class `SectionBlade` is a specialization of `Blade` which represents the reference blade using Eqs. (23) and (24). It needs to define function objects representing the blade sections, $\Big(x_s(\xi,\eta), y_s(\xi,\eta)\Big)$, the chord length relative to the diameter, $L(\eta)/D$, the pitch relative to the diameter, $\phi(\eta)/D$, the skew angle, $\theta_s(\eta)$, and the total rake relative to the diameter, $i_T(\eta)/D$.

The function $r(\eta)$ defining the dependence of the non-dimensional radius on the parameter $\eta$ is not specified in Eqs. (23) and (24). By default, `SectionBlade` uses the following definition for $r(\eta)$:

- If the blade closes at the tip it is given by

$$r(\eta) = \sin\Big(\tfrac{1}{2}\pi\eta\Big); \qquad \eta(r) = \frac{2\arcsin(r)}{\pi}. \tag{25}$$

  This enables the blade to close smoothly at the tip despite the coordinate singularity there. The derivative of the chord length with respect to $\eta$ remains finite at the tip while its derivative with respect to $r$ is infinite.

- If the chord length at the tip is non-zero, it is simpler to define

$$r(\eta) = \eta. \tag{26}$$

`SectionBlade` allows different functional forms for $r(\eta)$ but requires that $r(\eta)$ is strictly increasing. The member function

```
ScalarCurveType radius_curve() const;
```

returns a CurveLib function object evaluates $r$ as function of $\eta$. Similarly,

```
ScalarCurveType eta_curve() const;
```

returns $\eta$ as a function of $r$. For example, with $\eta$ and $r$ defined by Eq. (25), the following code

```
using namespace PGeom;
SectionBlade blade;
 . . . // Define the blade
Blade::ScalarCurveType r_crv   = blade.radius_curve(),
                       eta_crv = blade.eta_curve();
Float eta = eta_crv(0.5);
Float r   = r_crv(eta);
std::cout << "eta = " << eta << "  r = " << r << std::endl;
```

would generate the output

```
eta = 0.333333  r = 0.5
```

### 6.2.1 The surface of reference sections

Consider the surface defined by

$$x = x_s(\xi, r); \qquad y = y_s(\xi, r); \qquad z = r. \tag{27}$$

It will be called the surface of reference sections. It resembles a fin or wing where every section has a chord length of 1.0. Eqs. (23) and (24) define the reference blade

as a surface of reference sections twisted and deformed by the chord length, pitch angle, etc.

The class `RefSectionSurface` is a base for classes which represent the surface of reference sections. It is a two-parameter function object which returns a two-vector: given $(\xi, r)$ it returns $(x_s, y_s)$. The non-dimensional radius, $r$, has been chosen as the parameter of the `RefSectionSurface` instead of the blade parameter $\eta$ because the properties of the blade sections are traditionally given at values of $r$; moreover, these properties vary smoothly with $r$ even at the tip (unlike, for example, the chord length) so that splining with respect to $r$ will almost always generate an accurate representation of the blade surface.

`RefSectionSurface` is a specialization of `CurveLib::Curve<2U,AirfoilPt,Float>` where `AirfoilPt` represents a point $(x_s, y_s)$ on an airfoil: it is an alias for the class `VecMtx::VecN<2U,Float>`. For ease of use, the base class is given the alias `SurfaceType` within `RefSectionSurface`:

```
typedef CurveLib::Curve<2U,AirfoilPt,Float> SurfaceType;
```

`RefSectionSurface` has only a default (no argument) constructor. Therefore there is little purpose in creating an instance of a `RefSectionSurface` except as the base of one of the derived classes described in Secs. 6.2.1.3, 6.2.1.1 and 6.2.1.2.

The surface of reference sections is typically defined from an array of airfoils which are splined in $r$ to generate the surface. The geometry of airfoils and C++ classes to describe them have already been documented in Ref. 10.

The splining of the airfoils can be done in two ways: using Hermite splines (see Ref. 9, Sec. 8 for a description of Hermite splines) or tensor product splines. The former has the advantage that each airfoil in the input array remains unchanged in the surface of reference sections; however, the surface of reference sections will only be $C^1$, so that its normals may have discontinuous derivatives. For applications that are sensitive to the normals, in particular calculating blade pressures with boundary element methods, Hermite splines may not work well. Surfaces of reference sections which use Hermite splines are represented by the class `HermiteRefSectionSurface` described in Sec. 6.2.1.1.

In contrast, the method of tensor product splines generates a $C^2$ surface, so the normals are continuous, but has the disadvantage that the input airfoils must be approximated before being splined with respect to the parameter $r$. Surfaces of reference section which use tensor product splines are represented by the class `BSplineRefSectionSurface` described in Sec. 6.2.1.2.

Fig. 4 shows an inheritance diagram for `RefSectionSurface` and its derived classes.

The range of $r$ for which the `RefSectionSurface` is defined can be obtained using the member function

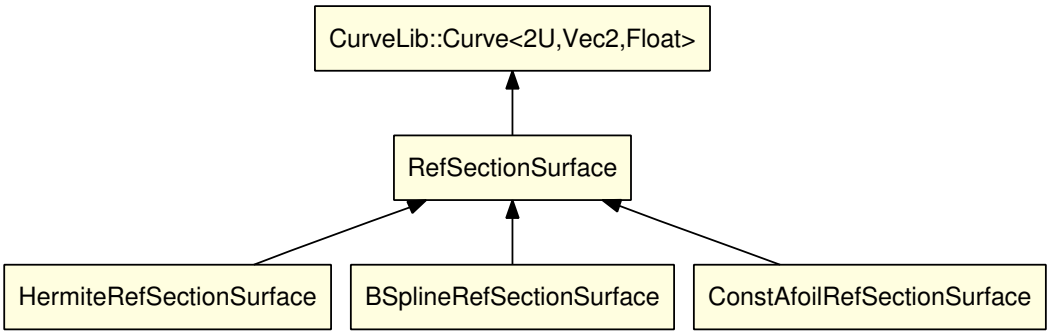



***Figure 4:*** *An inheritance diagram for the class* `RefSectionSurface`.

```
void r_range(Float &rlo, Float &rhi) const;
```

At the trailing edge of the surface of reference sections, each section can either be open or closed; the surface of reference sections is considered to have a closed trailing edge if it is closed at all its sections. If a section is closed, it can have continuous normals across the trailing edge (it is dull) or discontinuous normals (it is sharp). The trailing edge of the surface is considered to be dull only if all closed sections are dull; it is sharp if at least one closed section is sharp; if all sections are open, the sharpness is not defined. The following member functions return information about the trailing edge:

```
bool has_open_trailing_edge() const;
bool has_closed_trailing_edge() const;
bool has_sharp_trailing_edge() const;
bool has_dull_trailing_edge() const;
```

The surface of reference sections can also be open or closed at its tip (i.e. at the largest value of $r$). It is open if the uppermost section has non-zero thickness. Use

```
bool has_open_tip() const;
```

to determine whether the tip is open.

`RefSectionSurface` has been designed so that it can also be used to define the surface of a rudder or of a fin of a podded propulsor. Therefore it is allowed to have knuckles (lines of constant $r$ along which the normals to the surface are discontinuous). Since knuckles will seldom, if ever, be present on a propeller blade, the member functions implementing knuckles are not described here.

#### 6.2.1.1 Definition using Hermite splines

Suppose that for each $i \in [0, N-1]$, $\mathbf{x}_i(\xi)$ defines an airfoil at $r = r_i$. These curves can be combined to form a surface of reference sections as follows.

1. Given $\xi$ and $r$, find $j$ such that $r_j \leq r < r_{j+1}$.

2. Evaluate $\mathbf{x}_i(\xi)$ for $i \in [j-1, j+2]$.

3. The three points with $i \in [j-1, j+1]$ define a parabola whose slope can be used for the slope of the spline with respect to $r$ at $r_j$.

4. Similarly, the three points with $i \in [j, j+2]$ define a parabola whose slope can be used for the slope of the spline with respect to $r$ at $r_{j+1}$.

5. The points with $i = j$ and $j+1$, along with the two slopes, now uniquely defines a cubic polynomial with respect to $r$ in the range $r \in [r_j, r_{j+1}]$. The value of the polynomial is the value of the surface of reference sections at $(\xi, r)$.

This method of combining curves to form a surface is implemented in the class `CurveLib:HermiteExtendedCurve<N,V,F>` described in Ref. 9, Sec. 8.6. The surface of reference sections is only $C^1$ with respect to $r$ because the second derivatives at the end points of neighbouring cubic polynomial segments need not match.

In principle one could make a $C^2$ surface by sampling each airfoil at $\xi$, then forming a cubic spline through the points and evaluating the spline to get the point on the surface. In practice this is prohibitively slow, due to the many evaluations of the airfoil curves and the subsequent calculation of the spline coefficients. The Hermite splines depend only locally on the airfoil curves (i.e. only four airfoil curves are needed for each evaluation) and so are much faster to compute.

The class `HermiteRefSectionSurface`, a specialization of `RefSectionSurface`, defines a surface of reference sections using a `CurveLib::HermiteExtendedCurve` to interpolate between an array of airfoils. This representation has the property that the airfoil curves remain exact. If they interpolate offset data, the resulting surface will also interpolate those data.

`HermiteRefSectionSurface` has a default (no argument) constructor. When invoked, the surface of reference sections remains undefined; it must later be defined by copying from another `HermiteRefSectionSurface` or by using the `define` functions described below.

The following constructor makes a `HermiteRefSectionSurface` from an array of airfoils.

```
HermiteRefSectionSurface(const RadiusArray &radii,
                         const SectionArray &s,
                         bool open_tip = true);
```

Here `RadiusArray` is an alias for `std::vector<Float>` and `SectionArray` is an alias for `std::vector<Afoil::Airfoil<Float> >`. `Afoil::Airfoil<Float>` is a class representing an airfoil; it is described in Ref. 10. The array `radii` stores the value of $r$ for each section and `s` is an array of airfoils describing the section shapes. If `open_tip` is false, the surface will be closed at the tip using the algorithm described in Sec. 6.2.1.4.

For example, the following code makes a surface of reference sections from four sections (typically real propellers are specified with about ten):

```
using namespace PGeom;
RadiusArray radii;
radii.push_back(0.20);
radii.push_back(0.50);
radii.push_back(0.95);
radii.push_back(1.00);

Afoil::NACAAirfoil<Float>
  naca20("0020"), naca10("0010"), naca5("0005");
SectionArray afoils;
afoils.push_back(naca20);
afoils.push_back(naca10);
afoils.push_back(naca5);
afoils.push_back(naca5);

HermiteRefSectionSurface ref_sec_srf(radii,afoils,false);
```

The class `Afoil::NACAAirfoil<Float>` represents a NACA airfoil [12] (it is described in Ref. 10, Sec. 6.1.3), so the four sections are made from NACA 0020, NACA 0015 and NACA 0005 airfoils which have no cambre and thicknesses of 20%, 15% and 5% of chord length respectively. The four sections are located at $r = 0.2$, 0.5, 0.95 and 1.0; therefore the resulting surface of reference sections will get thinner toward the tip. It is shown in Fig. 5.

***Figure 5:*** *A surface of reference sections made from four NACA airfoils.*

Alternatively, the following constructor makes a `HermiteRefSectionSurface` from arrays of offset points at each section.

```
HermiteRefSectionSurface(const RadiusArray &radii,
                         const std::vector<OffsetArray> &offsets,
                         bool close_te = false,
                         bool sharp_te = true);
```

Here `OffsetArray` is an alias for `std::vector<AirfoilPt>`; it is an array of 2-vectors containing values of $(x_s, y_s)$ along the section. In each array of offsets, one offset must have an x-value of 0.0; this is the point at the leading edge.

The offsets are splined to create an airfoil (an `Afoil::Airfoil<Float>`) at each section, then the surface of reference sections is created by splining the array of

airfoils. The $\xi$ values for each array of offsets are determined by calculating the fractional arclength to each offset based on straight line segments between the points. These values are then adjusted so that the $\xi$ value of the point at the leading edge is 0.5 (the leading edge). Details are given in Ref. 10, Sec. 5.1.

If `close_te` is true, each airfoil will be forced to have a close trailing edge; if `sharp_te` is true the closed trailing edge will be sharp, otherwise it will be dull. The algorithms used to close the trailing edges are described in Ref. 10. The surface will have an open tip.

In addition, `HermiteRefSectionSurface` has the following functions for redefining an existing instance of the class. Each of these is equivalent to the constructor with the same arguments.

```
void define(const RadiusArray &radii,
            const SectionArray &s,
            bool open_tip = true);
void define(const RadiusArray &radii,
            const std::vector<OffsetArray> &offsets,
            bool close_te = false,
            bool sharp_te = true);
```

#### 6.2.1.2 Definition using tensor product splines

A tensor product spline of the sequence of airfoils is created by first approximating each airfoil with a spline, then splining their spline coefficients with respect to $r$. For this to work, it is necessary that each airfoil spline use the same knot sequence. The sequence of calculations used is:

1. Choose an initial knot sequence for the range $\xi \in [0, 1]$.
2. Choose the first airfoil.
3. Construct a spline on the current airfoil using the current knot sequence.
4. Sample the spline between each pair of knots and compare it with the airfoil. If it is not sufficiently accurate, insert a new knot at the centre of the knot interval.
5. Repeat steps 3 and 4 until the spline is sufficiently accurate.
6. Go to the next airfoil. Perform steps 3 to 5 until all airfoils have been approximated. The knot sequence is now sufficiently dense to approximate all the airfoils to the required accuracy.
7. Respline all the airfoils using the knot sequence.

8. Spline the coefficients of the airfoil splines with respect to $r$ to generate a two-dimensional spline with respect to parameters $(\xi, r)$. This is the surface of reference sections.

The class `BSplineRefSectionSurface`, a specialization of `RefSectionSurface`, uses a two-dimensional tensor-product B-spline curve to represent the surface defined by the reference sections of a blade. Currently, because periodic B-spline curves have not yet been implemented, the trailing edge of a `BSplineRefSectionSurface` must be sharp.

`BSplineRefSectionSurface` has a default (no argument) constructor. When invoked, the surface of reference sections remains undefined; it must later be defined by copying from another `BSplineRefSectionSurface` or by using the `define` functions described below.

The following constructor makes a `BSplineRefSectionSurface` from an array of airfoils:

```
BSplineRefSectionSurface(const RadiusArray &radii,
                         const SectionArray &s,
                         Float acc);
```

The accuracy to which the airfoils are approximated is specified using `acc`. For example, if `acc` is $10^{-4}$, then when evaluated at any $\xi$, the original airfoil and the spline approximation will return points whose separation is at most $10^{-4}$. (Since the chord length of all sections is 1.0, the accuracy can be considered to be non-dimensionalized by the chord length.) The surface will have an open tip.

Alternatively, a `BSplineRefSectionSurface` can be constructed from arrays of offsets at each section:

```
BSplineRefSectionSurface(const RadiusArray &radii,
                         const std::vector<OffsetArray> &offsets,
                         Float acc,
                         bool close_te = false);
```

The offsets are used to generate airfoils at each section as described for the similar `HermiteRefSectionSurface` constructor (Sec. 6.2.1.1); the surface of reference sections is then generated using the algorithm described above.

In addition, `BSplineRefSectionSurface` has the following functions for redefining an existing instance of the class. Each of these is equivalent to the constructor with the same arguments.

```
void define(const RadiusArray &radii,
            const SectionArray &s,
            Float acc);
```

```
void define(const RadiusArray &radii,
            const std::vector<OffsetArray> &offsets,
            Float acc,
            bool close_te = false);
```

Since the surface is represented as a tensor product spline, there is a knot sequence for each coordinate direction (see Ref. 9, Sec. 2). They can be obtained using the member functions

```
const Spline::KnotSeq<Float>& xi_knots() const;
const Spline::KnotSeq<Float>& r_knots() const;
```

The class `Spline::KnotSeq<Float>` representing a knot sequence is described in Ref. 9, Sec. 3.

The member function

```
void close_tip(bool exact);
```

can be used to close the tip using the algorithm described in Sec. 6.2.1.4. The surface will only be affected between the next-to-last and last sections. If `exact` is true, the pressure side and suction side of the closed section will match to (roughly) machine accuracy; otherwise matching is only guaranteed to within the accuracy of the reference section surface (as set by the argument `acc` in the constructor).

The member function

```
void close_trailing_edge();
```

can be used to close the trailing edge of the surface by collapsing it to the mean line of the trailing edges on the pressure and suction sides. Only the portion of the surface in the first and last $\xi$-knot interval will be affected.

#### 6.2.1.3 The class ConstAfoilRefSectionSurface

The class `ConstAfoilRefSectionSurface` represents a surface of reference sections in which all the reference sections are the same. It is not very useful, in practice, as propeller sections typically get thinner as one proceeds from the root to the tip and will also usually have varying amounts of cambre. `ConstAfoilRefSectionSurface` is derived from the base class `RefSectionSurface`.

`ConstAfoilRefSectionSurface` has a default (no argument) constructor. When invoked, the surface of reference sections remains undefined; it must later be defined by copying from another `ConstAfoilRefSectionSurface` or by using the `define` function described below.

The following constructor makes a `ConstAfoilRefSectionSurface` from an airfoil and a range of $r$ values.

```
ConstAfoilRefSectionSurface(const Afoil::Airfoil<Float> &a,
                            Float rlo, Float rhi);
```

A `ConstAfoilRefSectionSurface` can also be redefined using

```
void define(const Afoil::Airfoil<Float> &a,
            Float rlo, Float rhi);
```

#### 6.2.1.4 Closing the tip

If the chord length at the blade tip is zero, the blade will automatically be closed at the tip. In that case it is best that the uppermost section of the surface of reference sections remains open. However, if the chord length is not zero, the blade will only be closed at the tip if the surface of reference sections closes at the tip. This can be ensured in two ways:

1. by ensuring that the airfoil at the tip has zero thickness; or
2. by modifying the splines in the surface of reference sections to ensure that the surface closes.

The problem with the first method is that it will generally induce wiggles in the splines defining the surface of reference sections; the second method is preferred.

Suppose that there are $N$ airfoils with the last one, at $r_{N-1} = 1$, having non-zero thickness, and that the surface of reference sections has been constructed. When the surface is defined using Hermite splines, the cubic polynomial between $r_{N-2}$ and $r_{N-1}$ is altered: its value and slope at $r = r_{N-2}$ remain the same, but the value at $r = r_{N-2} = 1$ is changed to $\left(x_{N-1}(\xi), \frac{1}{2}[\, y_{N-1}(\xi) + y_{N-1}(1-\xi)]\right)$ and the slope there is changed to

$$\mathbf{s}_{N-1}(\xi) = \frac{\mathbf{x}_{N-1}(\xi) + \mathbf{x}_{N-1}(1-\xi) - 2\mathbf{x}_{N-2}(\xi)}{r_{N-1} - r_{N-2}} - \mathbf{s}_{N-2}(\xi). \tag{28}$$

This makes the polynomial between $r_{N-2}$ and $r_{N-1}$ parabolic with slope $\mathbf{s}_{N-2}$ at $r_{N-2}$.

If a tensor product spline is used, then:

1. The spline is first converted to a B-spline representation if it is not already.
2. Extra knots are included in the $\xi$ sequence to make it symmetric about $\xi = 0.5$: i.e. $t_j = 1 - t_{M-1-j}$ for each $j$, where $M$ is the number of $\xi$ knots. Note that this does not change the values of the splined surface, just the way they are represented. However, because of the increased number of knots, it will be somewhat less efficient to evaluate.
3. The last row of spline coefficients (i.e. the ones corresponding to $r_{N-1}$) is modified by $B_j \to \frac{1}{2}(B_j + B_{M_s-1-j})$ where $M_s$ is the number of spline coefficients ($M_s = M - 4$).

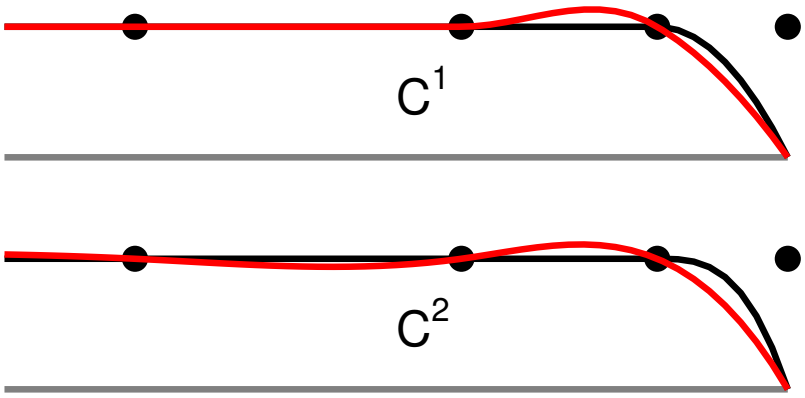



***Figure 6:*** *The thickness of the surface of reference sections after closure of the tip. The curves in red show the closure that results when the thickness of the last section is set to zero; the curves in black show the closure resulting when the splines are adjusted. The circles show where the airfoil sections were defined. The curves at the top use Hermite spline; those at the bottom use tensor product splines.*

In each of these methods, the surface of reference sections is modified only between $r_{N-2}$ and $r_{N-1}$. The airfoil at the tip is replaced by: $\mathbf{x}(\xi) = \frac{1}{2}\Big(\mathbf{x}(\xi) + \mathbf{x}(1-\xi)\Big)$.

Fig. 6 shows the thickness of a surface of reference sections near its tip after closure. The original surface had constant thickness. The sections at which the sections were defined are shown by the dots in the figure. The curves at the top are $C^1$ while those at the bottom are $C^2$. The curves in red show the closure when the thickness of the last section is set to zero. It can be seen that the thickness of the closed surface overshoots the original thickness on the second last interval between sections. For the $C^1$ case this does not propagate further, but for the $C^2$ case the wiggle extends all the way to the first section, although it becomes very small after a few sections. In contrast, the closure by adjusting the splines has no overshoot and the modification of the surface is confined to the interval between the last two sections.

***Figure 7:*** *The surface of reference sections of Fig. 5 after its tip has been closed.*

The surface of reference sections shown in Fig. 7 is the same as that shown in Fig. 5 except that this time it was constructed to have a closed tip. Notice that the geometry has changed only in the immediate vicinity of the tip: since the second last section is at $r = 0.95$, only the region $r \in [0.95, 1.0]$ has changed.

### 6.2.2 Closing the trailing edge

The airfoils used to define the surface of reference sections may need to have their trailing edges closed. There are two options:

1. use a closure that leaves the trailing edge sharp: the unit normal at the trailing edge on the suction side will differ from the unit normal on the pressure side; or
2. use a closure that makes the trailing edge dull: the two normals at the trailing edge will be the same.

The class `Afoil::Airfoil<Float>` provides the following member functions for doing this:

`void close_trailing_edge();`
: Ensures that the airfoil has a closed trailing edge; does nothing if the trailing edge is already closed. The trailing edge will be sharp.

`void make_trailing_edge_blunt(F radius);`
: If the airfoil has an open trailing edge, closes it such that it is dull; does nothing if the trailing edge is already closed. The argument `radius` is the approximate radius of curvature of the trailing edge. It must be strictly positive.

When the trailing edge is to be dull, the radius of curvature of each airfoil must be chosen carefully. Consider a plane that cuts normal to the trailing edge: see Fig. 8. We would like the cross-section of the trailing edge in this plane to be roughly semi-circular. However, the airfoil section meets the trailing edge obliquely, so that the semi-circular trailing edge in the plane is stretched into a semi-ellipse whose minor to major axis ratio is $\csc\beta$ with $\beta$ equal to the angle between the airfoil section and the line along the trailing edge. The radius of curvature of the ellipse at the trailing edge is

$$r_c = \tfrac{1}{2} d \sin\beta \tag{29}$$

where $d$ is the width of the trailing edge gap. From Eqs. (23) and (24), a tangent to the trailing edge is

$$\begin{aligned}
\mathbf{t} &= \frac{2}{D}\left[\hat{x}\frac{\partial x}{\partial r} + \hat{y}\frac{\partial y}{\partial r} + \hat{z}\frac{\partial z}{\partial r}\right]_{x_s=1/2,\, y_s=0} \\
&= (-\hat{x}\sin\theta + \hat{y}\cos\theta) \\
&\quad - (\hat{x}\cos\theta + \hat{y}\sin\theta)\left(r\frac{d\theta_s}{dr} + \frac{\cos\phi}{D}\frac{dL}{dr} - \frac{L\sin\phi}{D}\frac{d\phi}{dr} - \frac{L\cos\phi}{rD}\right) \\
&\quad + \hat{z}\left(\frac{2}{D}\frac{dR}{dr} + \frac{\sin\phi}{D}\frac{dL}{dr} + \frac{L\cos\phi}{D}\frac{d\phi}{dr}\right).
\end{aligned} \tag{30}$$

A unit tangent to the chord line at the trailing edge ($x_s = \frac{1}{2}$) is

$$\hat{a} = -(\hat{x}\cos\theta + \hat{y}\sin\theta)\cos\phi + \hat{z}\sin\phi \tag{31}$$

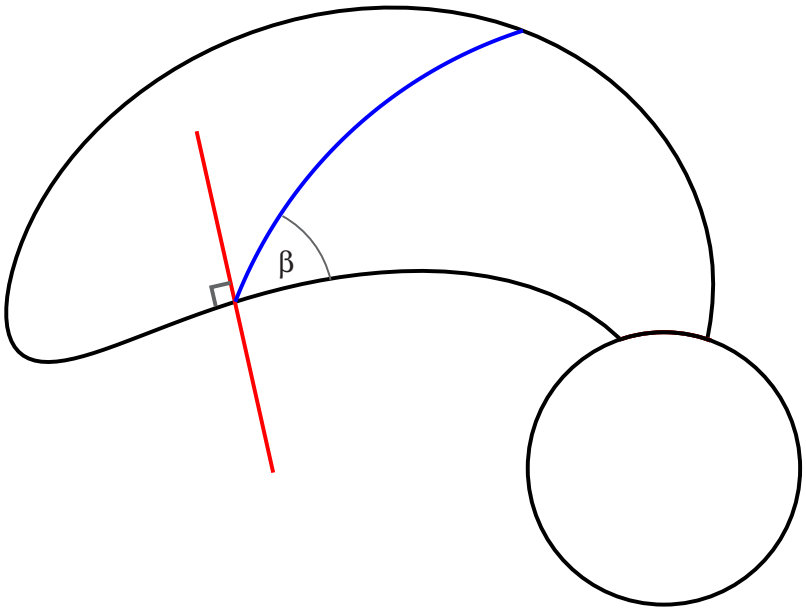



***Figure 8:*** *The red line marks a plane normal to the trailing edge of a highly skewed blade. The blue line is the section meeting the plane at the trailing edge.*

so that

$$\beta = \arccos(\hat{t} \cdot \hat{a}); \qquad \hat{t} = \frac{\mathbf{t}}{|\mathbf{t}|}. \tag{32}$$

Therefore the trailing edge at each section is closed using

$$r_c = \tfrac{1}{2} d \sqrt{1 - (\hat{t} \cdot \hat{a})^2}. \tag{33}$$

Because this method of closure requires knowledge of the blade section properties, knowledge that is not available to the classes representing the surface of reference sections, it must be implemented in `SectionBlade`, not in `RefSectionSurface`.

### 6.2.3 Defining the blade surface using a tensor product spline

A CurveLib function object used to represent a `SectionBlade` is normally a direct implementation of Eqs. (2), (23) and (24): the surface of reference sections, and function objects for the rake, skew, etc. are combined to form the blade surface. It is also possible to replace this representation with a two-parameter tensor product spline describing the blade surface. This has two principal advantages:

1. evaluating points on the surface is more efficient; and
2. the splined surface can be represented in standard file formats (e.g. IGES [15]) while the implementation using Eqs. (2), (23) and (24) cannot.

However, there are also drawbacks:

1. the spline representation is generally much more costly in memory as many spline coefficients must be stored;
2. the geometry near the tip will tend to be poorly defined as the coordinate singularity there will cause wiggles in the spline; and
3. it is only an approximation of the original blade surface so some fidelity is lost in the conversion.

The conversion of the blade surface to a tensor product spline is similar to that for the surface of reference sections in that the same knot sequence must be used on each blade section. If the surface of reference sections is a `BSplineRefSectionSurface`, its $\xi$ knot sequence can be used; otherwise a suitable knot sequence is determined using an algorithm similar to that described in Sec. 6.2.1.2. The knot sequence in the $\eta$ direction is obtained from the $r$ values of the blade sections. The original blade surface is sampled at an array of points which are then splined to generate the new representation.

## 6.2.4 Constructors

`SectionBlade` has a default (no argument) constructor. When used, the blade remains undefined and must later be defined by assignment to another `SectionBlade` or using one of the `define` functions described below. As for all CurveLib functions objects, the member function `is_defined` can be used to determine if the blade has been defined.

`SectionBlade` has three additional constructors defined in the following sections.

### 6.2.4.1 Construction from pre-defined function objects

The following constructor makes a blade from function objects representing the functions in Eqs. (23) and (24).

```
SectionBlade(RefSectionSurface rss,
             ScalarCurveType total_rake_crv,
             AngleCurveType skew_crv, ScalarCurveType pitch_crv,
             ScalarCurveType chord_crv, ScalarCurveType r_crv,
             ScalarCurveType eta_crv, bool rh = true);
```

The function object `rss` represents the functions $\Big(x_s(\xi, r), y_s(\xi, r)\Big)$; `total_rake_crv` represents the total rake relative to the diameter, $i_T(r)/D$; `skew_crv` represents the skew angle, $\theta_s(r)$; `pitch_crv` represents the pitch relative to the diameter, $p(r)/D$; `chord_crv` represents the chord length relative to the diameter, $L(\eta)/D$; `r_crv` represents the non-dimensional radius, $r(\eta)$; and `eta_crv` represents $\eta$ as a function of $r$. Notice that most of these curves are given as a function of $r$ but that the chord length

is given as a function of $\eta$ since its derivatives with respect to $r$ are infinite at the tip when, as is usual, the chord length there is zero. Requiring the specification of both $r(\eta)$ and $\eta(r)$ is not strictly necessary, as either could be derived from the other; however, inverting one or the other curve would be inefficient when a simple analytic version of the curve is available (as in Eqs. (25) and (26), for example); therefore both curves are required. If the flag `rh` is true, the blade will be right-handed; otherwise left-handed. CurveLib function objects are used to combine each of these arguments to form the blade surface according to Eqs. (23) and (24).

The range of radii is obtained using `rss.r_range(rlo,rhi)` (see Sec. 6.2.1); the range of $\eta$ can then be obtained using `eta_crv` to convert the range of $r$. It is assumed that all section properties curves are well-defined over these regions.

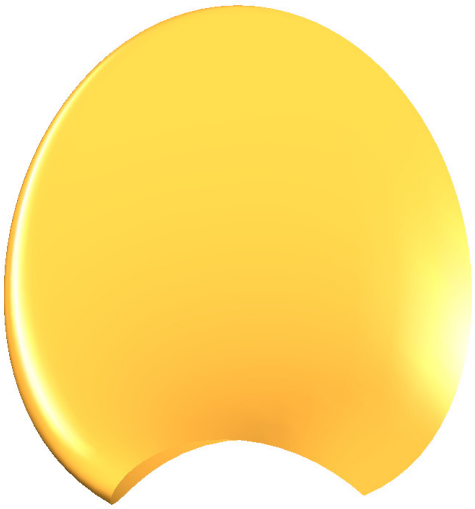


***Figure 9:*** *A blade with circular expanded outline.*

The following code defines a blade having no rake, no skew and constant pitch equal to $D$. It has a circular expanded outline: the chord length is defined as $L(r) = \sqrt{r(1-r)}\,D$. Note that the derivative of $L(r)$ at the tip is infinite; however, using the relationship between $r$ and $\eta$ given in Eq. (25) leads to $L(\eta) = \sin\left(\frac{1}{4}\pi(1-\eta)\right)\sqrt{2\sin(\frac{1}{2}\pi\eta)}$ which is well-behaved at the tip. The surface of reference sections is the same as that defined on page 29 and shown in Fig. 5. A display of the blade is shown in Fig. 9.

```
using namespace CurveLib;
Float pi = Const::Pi<Float>::value();
ConstCurve<1U,Float,Float> zero_crv(0.0), one_crv(1.0);
SectionBlade::ScalarCurveType
  total_rake_crv = zero_crv,
  pitch_crv = one_crv,
  eta = FIdentityCurve<Float>(),
  r_at_eta = Sin<Float>()(0.5*pi*eta),
  chord_at_eta = Sin<Float>()(0.25*pi*(1.0-eta))*
                 Sqrt<Float>()(2.0*r_at_eta),
  eta_at_r = (2.0/pi)*ArcSin<Float>();
CurveLib::AngleCurve<1U,Float> skew_crv(zero_crv);

SectionBlade blade(rss,total_rake_crv,skew_crv,pitch_crv,chord_at_eta,
                   r_at_eta,eta_at_r);
```

The member function

```
void define(RefSectionSurface rss,
            ScalarCurveType rake_c,
            AngleCurveType skew_c, ScalarCurveType pitch_c,
            ScalarCurveType chord_c, ScalarCurveType r_c,
            ScalarCurveType eta_c, bool rh = true);
```

can be used to redefine the blade after it has been constructed. It is equivalent to creating a new blade using the constructor described above.

#### 6.2.4.2 Construction from an array of BladeSections

The struct `BladeSection` represents a single section of a propeller blade: an airfoil with a value of non-dimensional radius, $r$, giving its location along the propeller reference line, and values for total rake, pitch, skew angle and chord length. It has the following public members.

`Float radius`
: The value of non-dimensional radius for the section.

`Float pitch`
: The pitch of the section relative to the diameter.

`Angle<Float> skew`
: The skew angle of the section.

`Float total_rake`
: The total rake of the section relative to the diameter.

`Float chord_length`
: The chord length of the section relative to the diameter.

`Afoil::Airfoil<Float> airfoil`
: An airfoil defining the reference section. The class `Afoil::Airfoil<Float>` is described in Ref. 10.

A `SectionBlade` can be constructed using an array of `BladeSection`s:

```
SectionBlade(const std::vector<BladeSection> &sections,
             bool rh = true, bool close_te = false,
             bool sharp_te = true, bool close_tip = false,
             bool sharp_tip = false, bool splined = false,
             bool c2 = true);
```

If `rh` is true, the blade will be right-handed; otherwise it will be left-handed.

If `close_te` is true, then `SectionBlade` will ensure that the trailing edges of all airfoils in the sections are closed: if `sharp_te` is true, any open trailing edge that is closed will be sharp; otherwise it will be dull. The algorithm described in Sec. 6.2.2 is used to close the trailing edges. Similarly, if `close_tip` is true and the chord length at the tip is finite, then the blade will be closed at the tip using the algorithm described in Sec. 6.2.1.4.

If `c2` is true, the blade surface will have continuous second derivatives with respect to its parameters. This is a necessary condition for some applications. However, this level of continuity may come at the price of approximating, rather than interpolating, the offset data in `p`. When `c2` is true and `p.sharp_te` is false, then the trailing edge closure of closed sections will be $C^2$ provided that all blade sections are closed.

The array of `BladeSections` provides a sequence of non-dimensional pitch values, $p_i$, at the section radii, $r_i$. A spline, $p_s(r)$, is generated using the points $(r_i, p_i)$; it returns the non-dimensional pitch as a function of $r$. If `c2` is true, a standard cubic spline is used to ensure that the curve is $C^2$; otherwise the curve will only be $C^1$. The pitch as a function of $\eta$ is then obtained using Eqs. (25) and (26):

$$p(\eta) = p_s\Big(r(\eta)\Big)D. \tag{34}$$

Curves for $i_T$ and $\theta_s$ as functions of both $r$ and $\eta$ are generated in a similar way.

If the blade tip is closed, $L$ as a function of $r$ will have infinite slope at the tip; splining $L$ with respect to $r$ will yield a curve that has finite slope at the tip resulting in a blade which is not smooth there. Because the slope of $L$ with respect to $\eta$ is finite, we can avoid this problem by converting the values $r_i$ to equivalent values of $\eta$ using Eq. (25) and splining with respect to $\eta$ to generate the curve $L(\eta)$.

If `splined` is true, the blade will first be constructed as described above, then converted to a splined surface using the algorithm described in Sec. 6.2.3. In this case the propeller section data will be approximated, not interpolated, but the blade surface will evaluate more efficiently.

The member function

```
void define(const std::vector<BladeSection> &sections,
            bool rh = true, bool close_te = false,
            bool sharp_te = true, bool close_tip = false,
            bool sharp_tip = false, bool splined = false,
            bool c2 = true);
```

can be used to redefine the blade after it has been constructed. It is equivalent to creating a new blade using the constructor described above.

#### 6.2.4.3 Construction from PropSectionData

`SectionBlade` has a constructor which defines the blade using offset data stored in a `PropSectionData` structure (defined in Sec. 9.3):

```
SectionBlade(const PropSectionData &p,
             bool rh = true, bool close_te = false,
             bool splined = false, bool c2 = true,
             bool close_tip = false);
```

A surface of reference sections is defined by splining the section offset data in `p`; similarly, splines for the chord length, rake, skew angle and total rake are defined. These are combined to make the blade surface.

The argument `p` contains arrays of offsets along the pressure and suction sides of blade sections at a number of radii, $r_i$. At each radius, the offsets are concatenated to form a single list of offsets which can be splined to generate an airfoil (an `Afoil::Airfoil<Float>`). The trailing edges of the airfoils are closed in accordance with `close_te` and `sharp_te` using the algorithm described in Sec. 6.2.2. The airfoils and the section properties in `p` can then be used to create an array of `BladeSection`s which are used to construct the blade as described in Sec. 6.2.4.2.

The remaining arguments are similar to those of the constructor described in the previous section: `rh` determines whether the blade is right-handed or left-handed; if `splined` is true, the blade will first be constructed as described above, then converted to a splined surface using the algorithm described in Sec. 6.2.3; if `c2` is true the blade surface will have continuous second derivatives with respect to its parameters; if `close_tip` is true and the chord length at the tip is finite, then `SectionBlade` will ensure that the blade is closed at the tip using the algorithm described in Sec. 6.2.1.4.

The member function

```
void define(const PropSectionData &p, bool rh = true,
            bool close_te = false, bool splined = false,
            bool c2 = true, bool close_tip = false);
```

can be used to redefine the blade after it has been constructed. It is equivalent to creating a new blade using the constructor described above.

### 6.2.5 Section properties

`SectionBlade` redefines the member functions `pitch_at_radius_curve`, etc. so that if the blade has not been rotated about the propeller reference line (see Sec. 6.1.6), they return the curves used to define the blade; these curves are usually more efficient and more accurate than those defined by the base class. In addition it provides member functions which return the pitch, total rake, etc. as a function of the blade parameter $\eta$. This can't be done in the base class as there is no guarantee that each section lies on a curve of constant $\eta$. However, note that these functions always refer to the blade properties when the blade has not been rotated about the propeller reference line. The new member functions are:

```
ScalarCurveType pitch_at_eta_curve() const;
AngleCurveType  pitch_angle_at_eta_curve() const;
ScalarCurveType chord_at_eta_curve() const;
AngleCurveType  skew_at_eta_curve() const;
ScalarCurveType total_rake_at_eta_curve() const;
ScalarCurveType generator_rake_at_eta_curve() const;
ScalarCurveType skew_induced_rake_at_eta_curve() const;
```

### 6.2.6 Defining a SectionBlade from a file

A `SectionBlade` can be defined by reading records in OFFSRF format [11] from an input file. The class `BladeReader` is used to read the file and construct the blade. `BladeReader` is derived from `Offsrf::Base` so it has an inserter defined for reading the file. It also has a public member `blade` of type `SectionBlade` to represent the blade non-dimensionalized using the diameter (i.e. its diameter will always be 1.0) as well as a public member `diameter` of type `Float` to specify the diameter. The following code reads the file `blade.dat` to define a propeller blade.

```
using namespace PGeom;
BladeReader reader;
Offsrf::IFStream in("blade.dat");
if (!in) throw Error("Could not open blade.dat");
in >> reader;
if (!in.good() && !in.eof())
  throw Error("Error when reading blade.dat.");
if (!reader.blade.is_defined()) {
  throw Error("Blade undefined after reading blade.dat.");
SectionBlade blade = reader.blade;
Float diameter = reader.diameter;
```

`BladeReader` recognizes three different records to define a blade: `DREA PROPELLER GEOMETRY`, `BLADE SECTIONS` and `SCALED BLADE SECTIONS`. They are mutually exclusive, so only one of them should be present in the file.

#### 6.2.6.1 The DREA PROPELLER GEOMETRY record

The `DREA PROPELLER GEOMETRY` record is used to read the definition of the blade from a DREA propeller geometry file [16]. The file is read into a `PropSectionData` struct which is used to construct the blade. The record has the following format:

`{DREA PROPELLER GEOMETRY:` *file-name*
  *<Records defining the blade properties>*
`}DREA PROPELLER GEOMETRY`

where *file-name* is the name of the file. The following records can be used to define the blade properties. Use

`{LEFT HANDED}` or `{RIGHT HANDED}`

to make the blade left- or right-handed. If neither is present it will be right-handed. Use

`{CLOSE TRAILING EDGE}`

to ensure that the trailing edges of all blade sections are closed. To specify whether they should be dull or sharp after closure use

`{DULL TRAILING EDGE}` or `{SHARP TRAILING EDGE}`

If neither of these is present, a sharp closure will be used. The algorithm described in Sec. 6.2.2 is used to close the sections. To ensure that the surface of reference sections is closed at the tip, use

`{CLOSE TIP}`

and

`{DULL TIP}` or `{SHARP TIP}`

to specify whether it should be dull or sharp after closure. The default is a dull tip. The records

`{C1}` or `{C2}`

are used to specify that the blade surface is $C^1$ or $C^2$. If the blade is $C^1$, the surface of reference sections will be defined as described in Sec. 6.2.1.1 and the curves defining the rake, skew, etc. will be defined using Hermite splines; if it is $C^2$ the surface of reference sections will be defined as described in Sec. 6.2.1.2 and the rake, skew, etc. will be defined using standard cubic splines. The record

`{SPLINE REPRESENTATION}`

can be used to require that the blade surface be replaced by a tensor product spline: see Sec. 6.2.3.

#### 6.2.6.2 The BLADE SECTIONS record

The `BLADE SECTIONS` record specifies a series of blade sections, each with an airfoil to define the section shape and values of $r$, $p/D$, $L/D$, $\theta_s$ and either $i_T$ or $i_G$. These are used to construct a series of `BladeSections` which are used to construct the blade. It has the following format:

```
{BLADE SECTIONS
  {SKEW IN DEGREES} or {SKEW IN RADIANS}
  {USE GENERATOR RAKE} or {USE TOTAL RAKE}
  {SECTION: r  p/D  L/D  θs  rake
    <Records to define an airfoil: see Ref. 10, Sec. 7>
  }SECTION
  ... ! More SECTION records.  There should be at least two.
  {DIAMETER: diameter }
  <Records defining the blade properties>
}BLADE SECTIONS
```

The records `SKEW IN DEGREES` and `SKEW IN RADIANS` determine whether the values of $\theta_s$ in the `SECTION` records are interpreted as angles in radians or degrees. If neither is present, the skew angle will be in degrees.

Similarly, the records `USE GENERATOR RAKE` and `USE TOTAL RAKE` determine whether the values of *rake* are interpreted as total rake, $i_G$ or generator rake $i_T$. If neither is present total rake is assumed.

The records defining the blade properties are the same as for the `DREA PROPELLER GEOMETRY` record.

#### 6.2.6.3 The SCALED BLADE SECTIONS record

The `SCALED BLADE SECTIONS` record specifies a series of blade sections all constructed from a single thickness distribution and mean line offset curve (see Ref. 10, Sec. 6.1). Each section has an associated value of $r$, $t/L$, $c/L$, $p/D$, $L/D$, $\theta_s$, and either $i_T$ or $i_G$, where $t$ is the thickness and $c$ is the cambre. The thickness distribution, mean line offset curve, thickness and cambre are used to construct an airfoil for each section. These are used to construct a series of `BladeSections` which are used to construct the blade. The record has the following format:

The `SCALED BLADE SECTIONS` record has the following format:

```
{SCALED BLADE SECTIONS
  {SKEW IN DEGREES} or {SKEW IN RADIANS}
  {USE GENERATOR RAKE} or {USE TOTAL RAKE}

  {THICKNESS DISTRIBUTION AIRFOIL
    <Records to define the thickness and cambre: see Ref. 10, Sec. 7.9 >
  }THICKNESS DISTRIBUTION AIRFOIL

  {SECTION: r  t/L  c/L  p/D  L/D  θs  rake }
   ... ! More SECTION records.  There should be at least two.

  {DIAMETER: diameter }
  <Records defining the blade properties >
}SCALED BLADE SECTIONS
```

The records `SKEW IN DEGREES`, `SKEW IN RADIANS`, `USE GENERATOR RAKE` and `USE TOTAL RAKE` affect the values of *skew-angle* and *rake* in the same way as in the `BLADE SECTIONS` record and the records defining the blade properties are the same as for the `DREA PROPELLER GEOMETRY` record.

The following records define the geometry of the David Taylor Model Basin (DTMB) P4382 propeller blade, one of the NSRDC skewed propeller series [17]. It uses the DTMB modification of the NACA 66 airfoil [18] for each of its sections and has no generator rake. A display of the blade is shown in Fig. 10.

***Figure 10:*** *The blade of DTMB P4382.*

```
{SCALED BLADE SECTIONS
  {SKEW IN DEGREES}
  {USE GENERATOR RAKE}

  {THICKNESS DISTRIBUTION AIRFOIL
    {NACA 66 DTMB(mod): 1 1 }
  }THICKNESS DISTRIBUTION AIRFOIL

  !       radius  thick   cambre pitch   chord skew(deg.) rake
  {SECTION: 0.2   0.2494  0.0389 1.4142  0.174   0.000    0.0 }
  {SECTION: 0.3   0.1562  0.0370 1.4332  0.229   4.655    0.0 }
  {SECTION: 0.4   0.1068  0.0344 1.4117  0.275   9.363    0.0 }
  {SECTION: 0.5   0.0768  0.0305 1.3613  0.312  13.948    0.0 }
  {SECTION: 0.6   0.0566  0.0247 1.2854  0.337  18.378    0.0 }
  {SECTION: 0.7   0.0421  0.0199 1.1999  0.347  22.747    0.0 }
  {SECTION: 0.8   0.0314  0.0161 1.1117  0.334  27.145    0.0 }
  {SECTION: 0.9   0.0239  0.0134 1.0270  0.280  31.575    0.0 }
  {SECTION: 1.0   0.0200  0.0100 0.9589  0.000  36.000    0.0 }

  {DIAMETER: 0.3048 }

  {RIGHT HANDED}
  {CLOSE TRAILING EDGE}
  {DULL TRAILING EDGE}
}SCALED BLADE SECTIONS
```

# 7 Classes representing hubs

The hub is represented as a spine curve,

$$\mathbf{g}(\xi_h) = \hat{r}\, g_r(\xi_h) + \hat{z}\, g_z(\xi_h), \tag{35}$$

rotated about the $z$ axis to form an axisymmetric surface. Here $g_z(\xi_h)$ is the distance along the $z$ axis and $g_r(\xi_h)$ is the distance of a point on the spine from the $z$-axis. The function $g_z(\xi_h)$ should be increasing. A point on the hub is then parameterized using $\xi_h$ and $\theta_h$:

$$\mathbf{x}(\xi_h, \theta_h) = \hat{x}\, g_r(\xi_h) \sin\theta_h + \hat{y}\, g_r(\xi_h) \cos\theta_h + \hat{z}\, g_z(\xi_h). \tag{36}$$

With these definitions we have $\theta_h = -\theta$, where $\theta$ is the angular coordinate defined in Eq. (3). The change in sign has been made to maintain outward pointing normals to the hub surface: i.e. $\hat{\xi}_h \times \hat{\theta}_h$ is outward pointing. The intersection of the propeller reference line with the hub is at $\theta_h = 0$.

## 7.1 The base class Hub

The base class Hub represents a propeller hub. It is a specialization of the CurveLib class `AxisymmetricSurface<Float>` (see Ref. 1, Sec. 8.1) which is itself a specialization of `Curve<2U,Point,Float>`, so a `Hub` is a specialization of a `Surface` as defined in Sec. 5. Like all propeller surfaces, the hub is non-dimensionalized using the propeller diameter.

Specializations of `Hub` are provided for defining cylindrical hubs with or without hemispherical end caps and for defining a hub using splined offsets. The specializations are described in Secs. 7.1.1–7.1.3. Fig. 11 is an inheritance diagram for the hub classes and Fig. 12 shows examples of hubs created using the derived classes.

`Hub` provides the alias `ZRPoint` to represent a point $(z, r)$ returned by the spine function; it is equivalent to `VecMtx::VecN<2U,Float>`. `AxisymmetricSurface<Float>` also provides the following alias for the type of the function object representing the spine of the surface:

```
typedef CurveLib::Curve<1U,ZRPoint,Float> SpineCurveType;
```

`Hub` has two constructors, the default (no argument) constructor, and

```
Hub(SpineCurveType s, Float xih_lo, Float xih_hi);
```

which makes a hub using `s` as the spine. The range for $\xi_h$ is given by `xih_lo` and `xih_hi`: i.e. the upstream end of the hub is at $\xi_h =$ `xih_lo` and the downstream end at $\xi_h =$ `xih_hi`. An existing hub can also be redefined in a similar way using

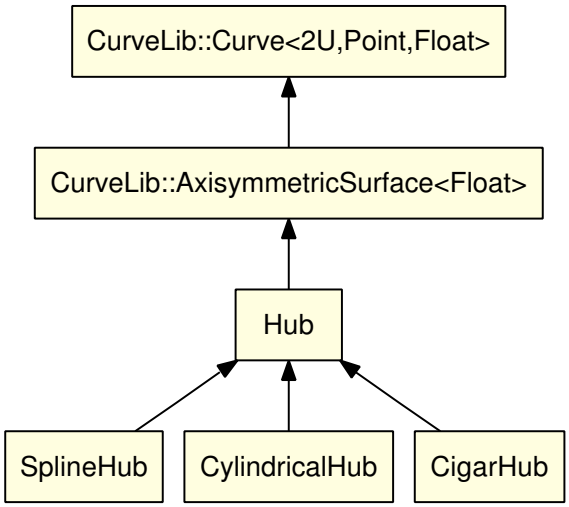



***Figure 11:*** *An inheritance diagram for the class* `Hub`.

```
void define(SpineCurveType s, Float xih_lo, Float xih_hi);
```

The function object representing the spine can be obtained using

```
SpineCurveType spine_curve() const;
```

Its range for the parameter $\xi_h$ can be obtained using

```
void spine_range(Float &xih_lo, Float &xih_hi) const;
```

The range of $\theta_h$ is unlimited. The range of $z$ values is obtained using

```
void z_range(Float &z_lo, Float &z_hi) const;
```

The spatial coordinate $z$ is a function only of $\xi_h$. The member function

```
Float z(Float xih, unsigned n = 0) const;
```

is provided as a convenience for calculating $z$ at $\xi_h$. The argument `n` specifies the number of derivatives to be taken: i.e. if `n` is positive, then the value of $d^n z/d\xi_h^n$ is returned instead. This function is usually very efficient as it only involves evaluation of $g_z(\xi_h)$ or its derivatives.

The member function

```
Float xi(Float z, unsigned n = 0) const;
```

performs the inverse calculation returning $\xi_h$ or its derivatives given a value of $z$. In general it is only an approximation calculated by using a Newton-Raphson iteration to solve $g_z(\xi_h) = z$ (but this may be changed by derived classes). Control over the accuracy of `xi` is provided by the following two member functions:

```
Float get_xi_accuracy() const;
Float set_xi_accuracy(Float acc);
```

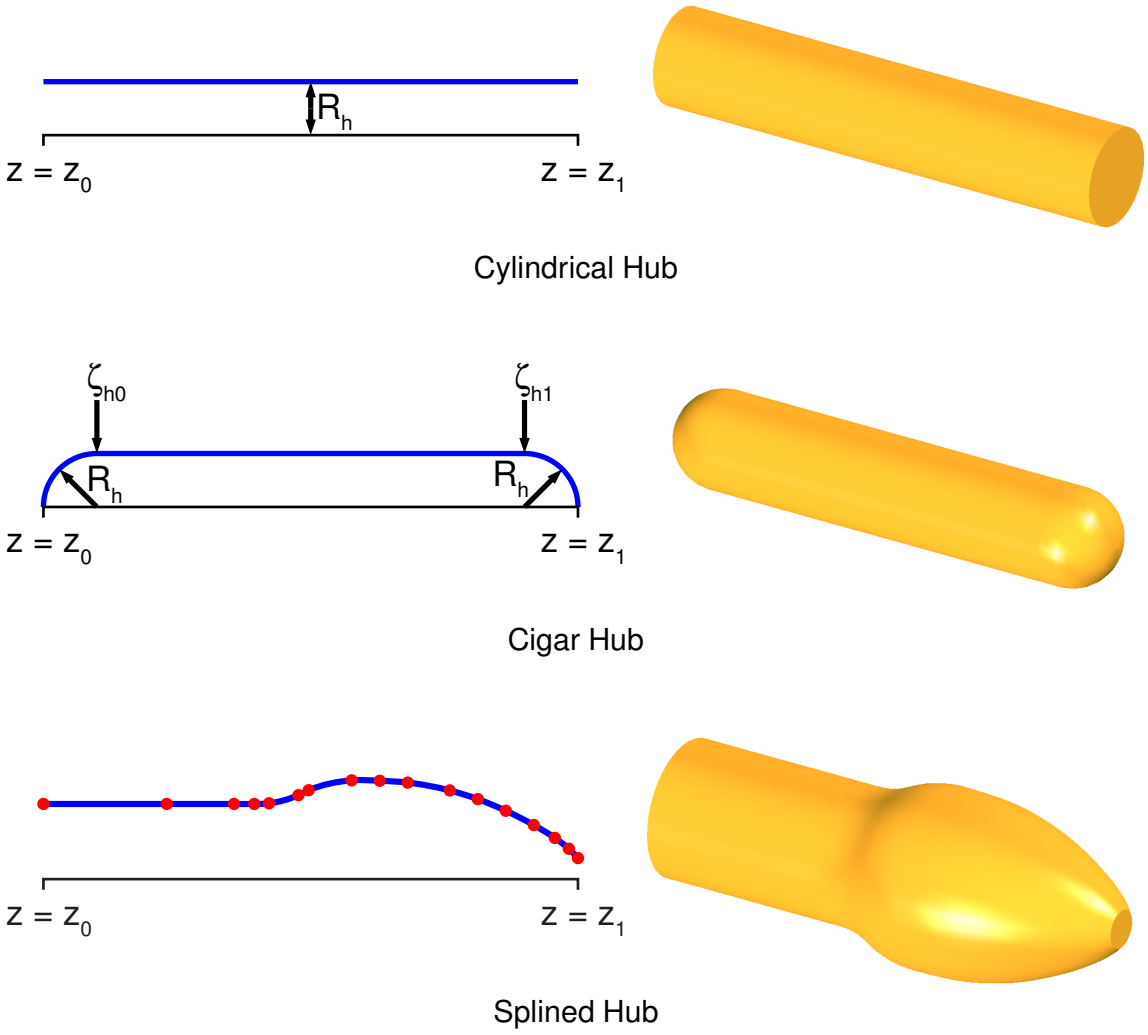



***Figure 12:*** *Hubs created by the* `CylindricalHub`, `CigarHub` *and* `SplineHub` *classes. The spine for each hub is at the left, its appearance in three-dimensions at the right. The red dots on the spine of the splined hub show the points interpolated to create the spine.*

The latter returns the previous value of the accuracy. If the accuracy of `xi` is `acc` then `|z(xi(z_val))-z_val| < acc`. By default the accuracy will be the propeller accuracy (obtained from `get_propeller_accuracy`: see Sec. 2.5) at the time the hub is constructed or defined; however, derived classes may use a different default accuracy, in particular when $\xi_h$ can be determined analytically so that the accuracy is effectively zero.

It is straightforward to evaluate points on the hub using parameters $(\xi_h, \theta_h)$ by using the standard evaluation function for a CurveLib function object. The following member functions provide a similar capability when one wants to use $z$ as the parameter instead of $\xi_h$.

```
Point value_at_z(Float zh, Angle<Float> theta) const;
```
Returns a point on the hub as a function of $z$ and $\theta_h$.

```
Point value_at_z(Float zh, Angle<Float> theta,
                 unsigned dz, unsigned dtheta) const;
```
Returns a point on the hub, or its derivatives, as a function of $z$ and $\theta_h$; `dz` derivatives with respect to $z$ are taken and `dtheta` with respect to $\theta_h$.

```
ZRPoint spine_value_at_z(Float zh, unsigned d = 0) const;
```
Returns the value of the spine or its derivatives as a function of $z$.

The member function

```
CurveLib::Curve<3U,SurfaceParam,Float> parameter_curve() const;
```

returns a function object that, when given a Cartesian point $(x, y, z)$, projects it radially onto the hub, then returns the hub parameters which generate the projected point: i.e. it will return $(\xi_h, \theta_h) = \big(\xi(z), \arctan(x/y)\big)$ where $\xi(z)$ is the function implemented by member function `Hub::xi`. The accuracy of the returned curve is given by the accuracy of `Hub::xi` when `parameter_curve` is called: i.e. if `p` is the value of the curve, and `acc` is the value returned by `hub.get_xi_accuracy`, then `hub(p)` will be within `acc` of the projected point. The $\theta_h$ value returned by the curve is always in the range $[-\pi, \pi]$. If a curve whose value is a Cartesian point is composed with the parameter curve, the resulting curve may be discontinuous at $\theta_h = \pm\pi$. The returned curve is only well-defined for values of $z$ in the range returned by `z_range` and for $r = \sqrt{x^2 + y^2} > 0$.

The member function

```
CurveLib::Curve<3U,XiThetaRPoint,Float> parameter_3d_curve() const;
```

is in all respects similar to `parameter_curve`, except that it returns a function object that returns points in the hub coordinate system $(\xi_h, \theta_h, r)$ instead of just the hub parameters $(\xi_h, \theta_h)$. `XiThetaRPoint` is an alias for `VecMtx::VecN<3U,Float>`.

The member function

```
CurveLib::Curve<3U,Float,Float> distance_above_hub_curve() const;
```

returns a function object that, given a Cartesian point $(x, y, z)$, returns the distance of the point above the hub as measured along a radial line (not along a normal to the hub): i.e. it returns $d = \sqrt{x^2 + y^2} - g_r\Big(\xi(z)\Big)$. The function object returned by `distance_above_hub_curve` is defined only for values of $z$ in the range returned by `z_range`.

To determine if a Cartesian point `x` is inside the hub, use the member function

```
int is_inside(const Point &x) const;
```

The possible returned values are

0: if `x` is inside the hub;
1: if `x` is above the hub;
2: if `x` is upstream of the start of the hub; and
3: if `x` is downstream of the end of the hub.

The hub may be split into a rotating part and a stationary part at $z = z_h$: the portion with $z < z_h$ rotates; the portion with $z > z_h$ is stationary. The inclusion of this feature was made to facilitate the representation of a podded propulsor in which the propeller is upstream of the fixed portion of the pod. The following member functions support this feature:

```
Float z_transition() const;
```
Returns the value of $z_h$.

```
void set_z_transition(Float z);
```
Sets the value of $z_h$

```
bool has_z_transition() const;
```
Returns true if a value for $z_h$ has been defined.

The size of a hub can be scaled simply by multiplying it by a `Float`:

```
Hub operator*(const Hub &h, Float s);
Hub operator*(Float s, const Hub &h);
```

Each function returns a hub equivalent to `h` scaled by `s`.

### 7.1.1 Cylindrical hubs

The class `CylindricalHub`, a specialization of `Hub`, represents a cylindrical propeller hub defined by

$$g_r(\xi_h) = R_h/D, \qquad g_z(\xi_h) = z_0 + \xi_h(z_1 - z_0), \tag{37}$$

where $R_h$ is the radius of the hub. The parameter $\xi_h$ has the range [0,1] and in terms of $z$ is given by

$$\xi_h = \frac{z - z_0}{z_1 - z_0}. \tag{38}$$

Since $\xi_h$ is related to $z$ by a simple analytic function, the member function `xi` is always accurate to machine accuracy. Therefore the implicitly virtual function `set_xi_accuracy` inherited from `Hub` does nothing.

`CylindricalHub` has a default constructor as well as the following:

```
CylindricalHub(Float radius, Float z0, Float z1);
```

where `radius` is the radius of the hub and `z0` and `z1` are the values of $z$ at the upstream and downstream ends, respectively. An existing cylindrical hub can be redefined using

```
void define(Float r, Float z0, Float z1);
```

### 7.1.2 Cigar hubs

A `CigarHub` is a cylindrical propeller hub with hemispherical end caps; it is a specialization of `Hub`. Its spine is defined by

$$g_z(\xi_h) = \begin{cases} z_0 + R_h \left[1 - \cos\left(\dfrac{\pi\xi_h}{2\xi_{h0}}\right)\right] & \text{for } \xi_h < \xi_{h0} \\ z_0 + R_h + \dfrac{(\xi_h - \xi_{h0})(z_1 - z_0 - 2R_h)}{\xi_{h1} - \xi_{h0}} & \text{for } \xi_{h0} \le \xi_h \le \xi_{h1} \\ z_1 - R_h \left[1 - \cos\left(\dfrac{\pi(1-\xi_h)}{2(1-\xi_{h1})}\right)\right] & \text{for } \xi_{h1} < \xi_h \end{cases} \tag{39}$$

$$g_r(\xi_h) = \begin{cases} R_h \sin\left(\dfrac{\pi\xi_h}{2\xi_{h0}}\right) & \text{for } \xi_h < \xi_{h0} \\ R_h & \text{for } \xi_{h0} \le \xi_h \le \xi_{h1} \\ R_h \sin\left(\dfrac{\pi(1-\xi_h)}{2(1-\xi_{h1})}\right) & \text{for } \xi_{h1} < \xi_h \end{cases} \tag{40}$$

$$\xi_{h0} = 1 - \xi_{h1} = \frac{\pi R_h}{2(\pi R_h + z_1 - z_0 - 2R_h)} \tag{41}$$

where $R_h$ is the hub radius, $z_0$ and $z_1$ are the values of $z$ at the upstream and downstream ends of the hub and $\xi_{h0}$ and $\xi_{h1}$ are the values of $\xi_h$ at which the upstream and downstream end-caps meet the cylindrical central section. The parameter $\xi_h$ has the range [0,1] and is equivalent to the fractional arclength along the spine curve. It can be represented in terms of $z$ by inverting Eq. (39):

$$\xi_h = \begin{cases} \dfrac{2\xi_{h0}}{\pi} \arccos\left(1 - \dfrac{(z - z_0)}{R_h}\right) & \text{for } z < z_0 + R_h, \\ \xi_{h0} + \dfrac{(z - z_0 - R_h)(\xi_{h1} - \xi_{h0})}{z_1 - z_0 - 2R_h} & \text{for } z_0 + R_h \le z \le z_1 - R_h, \\ 1 - \dfrac{2(1 - \xi_{h1})}{\pi} \arccos\left(1 + \dfrac{z - z_1}{R_h}\right) & \text{for } z > z_1 - R_h. \end{cases} \tag{42}$$

The implicitly virtual function `set_xi_accuracy` will do nothing since $\xi$ can be determined from $z$ to machine accuracy.

`CigarHub` has a default constructor as well as the following:

```
CigarHub(Float radius, Float z0, Float z1);
```

where `radius` is the radius of the hub and `z0` and `z1` are the values of $z$ at the upstream and downstream ends, respectively. An existing cylindrical hub can be redefined using

```
void define(Float r, Float z0, Float z1);
```

### 7.1.3 Splined hubs

A `SplineHub` is a `Hub` whose spine is represented by a Hermite spline through a set of offsets (see Ref. 9, Sec. 8 for a description of Hermite splines and the class `Spline::HermiteSpline` used to represent them). The slopes for the spline at the offset points can be calculated in different ways.

Use the constructor

```
SplineHub(const Spline::HermiteSpline<ZRPoint,Float> &s);
```

to make a hub using `s` as the spine; the range of $\xi_h$ is determined from the spline knot sequence. A `SplineHub` can also be constructed from data in a `PropSectionData`:

```
SplineHub(const PropSectionData &p, bool c2 = false);
```

If `c2` is true, the hub will be $C^2$; otherwise it will only be $C^1$. The following functions can be used to redefine an existing hub; they are equivalent to the constructors with the same arguments.

```
void define(const Spline::HermiteSpline<ZRPoint,Float> &s);
void define(const PropSectionData &p, bool c2 = false);
```

The knots used by the spine can be obtained using

```
const Spline::KnotSeq<Float>& get_spine_knots() const;
```

The class `Spline::KnotSeq<Float>` is described in Ref. 9, Sec. 3.

## 7.2 Defining a hub from a file

A `Hub` can be defined by reading records in OFFSRF format [11] from an input file. The class `HubReader` is used to read the file and construct the hub. `HubReader` is derived from `Offsrf::Base` so it has an inserter defined for reading the file. It also has a public member `hub` of type `Hub` to represent the hub. The following code reads the file `hub.dat` to define a hub.

```
using namespace PGeom;
HubReader reader(true);
Offsrf::IFStream in("hub.dat");
if (!in) throw Error("Could not open hub.dat");
in >> reader;
if (!in.good() && !in.eof())
  throw Error("Error when reading hub.dat.");
if (!reader.hub.is_defined())
  throw Error("Hub undefined after reading blade.dat.");
Hub hub = reader.hub;
```

The boolean argument of the `HubReader` constructor affects whether the hub will be dimensional or non-dimensional when it is defined by reading a DREA propeller geometry file (see the description of the `DREA PROPELLER GEOMETRY` record below). If the argument is true, the hub will be non-dimensionalized using the propeller diameter; otherwise it will have its true dimensions. When the hub is to be included as a component of a propeller, the argument should always be true.

`HubReader` recognizes four different records to define a hub: `CYLINDER`, `CIGAR`, `DREA PROPELLER GEOMETRY` and `OFFSETS`. They are mutually exclusive, so only one of them should be present in the file.

The `DREA PROPELLER GEOMETRY` record reads a file in the DREA propeller geometry format [16] and defines the hull from the hub offset data within it. The record has the following format:

`{DREA PROPELLER GEOMETRY:` *file-name* `}`

where *file-name* is the name of the file containing the description of the propeller.

The `CYLINDER` record defines a cylindrical hub; it has the following format:

`{CYLINDER:` $R_h$ $z_{min}$ $z_{zmax}$ `}`

where $R_h$ is the hub radius, and $z_{zmin}$ and $z_{zmax}$ define the range of $z$ for the hub. When the hub is to be included as a component of a propeller, these values would be non-dimensionalized using the propeller diameter.

The `CIGAR` record defines a cylindrical hub with hemispherical end caps. Its format is similar to that for the cylindrical hub:

`{CIGAR:` $R_h$ $z_{min}$ $z_{zmax}$ `}`

The total $z$ range must exceed the hub diameter: i.e. $z_{zmax} - z_{min} > 2R_h$.

The `OFFSETS` record is used to define a hub by splining a sequence of offsets. It has the following format:

`{OFFSETS`
$z_1$ $r_1$
...
$z_n$ $r_n$
`}OFFSETS`

The offsets are used to construct a $C^2$ `SplineHub`.

In addition, the following record can be used to define the $z$ value for the transition between the rotating and stationary portions of the hub.

`{Z TRANSITION:` $z$ `}`

It should appear after the record used to define the hub.

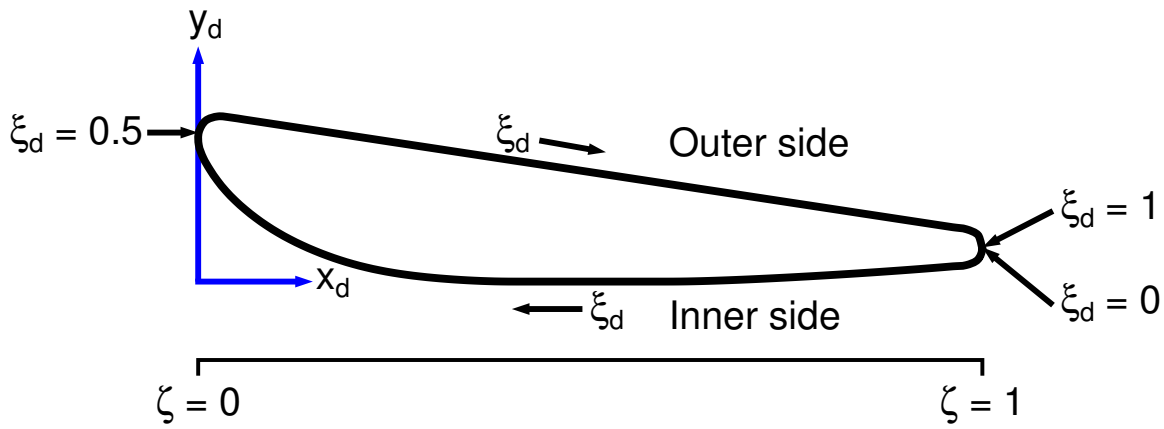



***Figure 13:*** *The coordinates for the airfoil defining a duct cross-section.*

# 8 Classes representing ducts

Like a hub, a propeller duct is represented as an axisymmetric surface created by rotating a cross-section curve,

$$\mathbf{c}(\xi_d) = \hat{r}\, c_r(\xi_d) + \hat{z}\, c_z(\xi_d), \tag{43}$$

about the $z$ axis. The cross-section is defined by scaling and translating an airfoil:

$$c_z(\xi_d) = L_d\Big(x_d(\xi_d) - \zeta_p\Big), \qquad c_r(\xi_d) = L_d y_d(\xi_d) + r_{off}, \tag{44}$$

where $(x_d, y_d)$ is the airfoil, $L_d$ is the chord length of the duct, $\zeta_p$ is the distance from the leading edge of the duct to the propeller plane, and $r_{off}$ is a radial offset.

The standard airfoil parameterization is used so that parameter $\xi_d$ increases from 0.0 at the trailing edge on the inner surface, to 0.5 at the leading edge, to 1.0 at the trailing edge on the outer surface: see Eq. 44. The coordinate $x_d$ is 0.0 at the leading edge of the duct and 1.0 at the trailing edge so that is is equivalent to the fractional chord length, $\zeta$. However, note that unlike most airfoils, $y_d$ is usually not 0.0 at the leading and trailing edges: instead, $y_d$ is 0.0 on the lowest point of the inner side of the airfoil (i.e. the side that corresponds to the inner surface of the duct).

A point on the duct is then parameterized using $\xi_d$ and $\theta_d$:

$$\mathbf{x}(\xi_d, \theta_d) = (\hat{x}\sin\theta_d + \hat{y}\cos\theta_d)\, c_r(\xi_d) + \hat{z}\, c_z(\xi_d) \tag{45}$$

With these definitions we have $\theta_d = \theta_h = -\theta$, where $\theta_h$ is the angular hub parameter and $\theta$ is the angular coordinate defined in Eq. (3). The normals to the duct are outward pointing.

## 8.1 The class Duct

The class Duct represents a propeller duct. It is a specialization of the CurveLib class `AxisymmetricSurface<Float>` (see Ref. 1, Sec. 8.1) which is itself a specialization of `Curve<2U,Point,Float>`, so a `Duct` is a specialization of a `Surface` as defined in Sec. 5. Like all propeller surfaces, the duct is non-dimensionalized using the propeller diameter.

`Duct` has three constructors in addition to the default and copy constructors.

```
Duct(Afoil::Airfoil<Float> afoil, Float chord_over_radius,
     Float zeta_prop, Float r_offset);
```

makes a duct using `afoil` to define the cross-section shape. The values of $L_d/R$, $\zeta_p$ and $r_{off}/R$ are given by `chord_over_radius`, `zeta_prop` and `r_offset` respectively.

There are several duct cross-sections that have been used widely both on real ships and as test cases to validate computations. Two of them, the MARIN 19A and the MARIN 37 ducts shown in Fig. 14, are available using the following constructor:

```
Duct(const Str &desig, Float chord_over_radius, Float zeta_prop,
     Float r_offset);
```

This makes a duct using a cross-section having a standard designation given by `desig`; currently its value must be `MARIN 19A` or `MARIN 37` but other standard ducts may be added in the future. An `Error` is thrown if the designation is not recognized. The remaining arguments are the same as in the previous constructor.

A duct can also be constructed from a `DuctData` struct.

```
Duct(const DuctData &dd);
```

`DuctData` has the following public members:

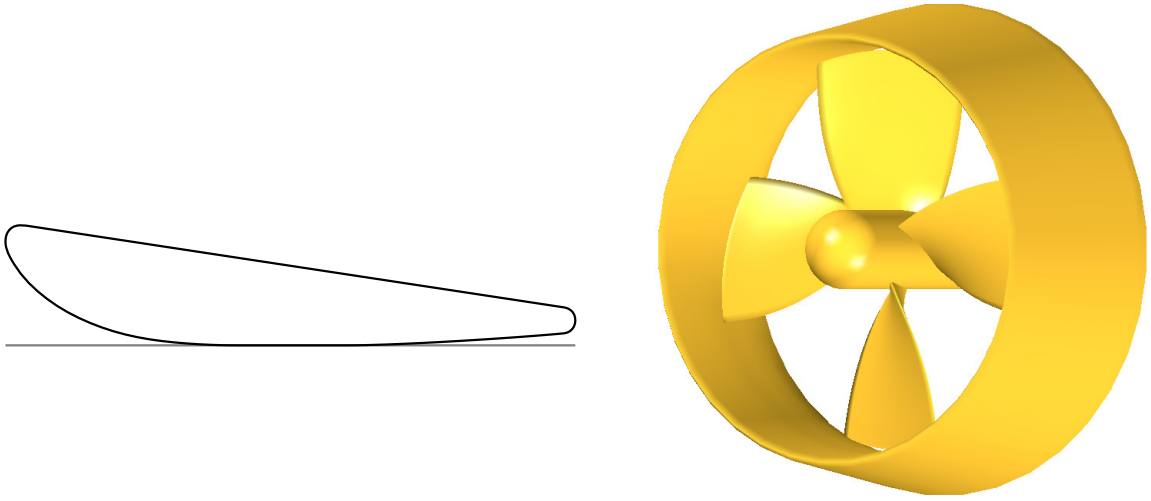
`description;`
: A description of the duct.

`Float chord_over_radius;`
: The value of $L_d/R$.

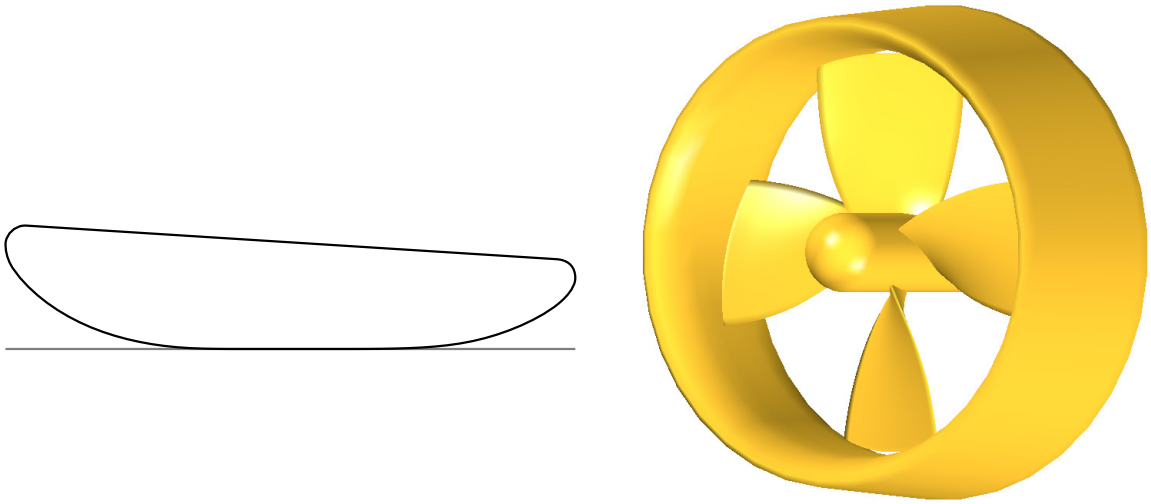
`Float z_prop;`
: The value of $\zeta_p$.

`Float r_offset;`
: The value of $r_{off}/R$.

`std::vector<AirfoilPt> xup;`
: Points on the upper surface of the airfoil defining the duct cross-section. The points are in order from the leading edge to the trailing edge. The abscissa of each point is the fractional chord length measured from the leading edge.



MARIN 19A Duct



MARIN 37 Duct

***Figure 14:*** *The MARIN 19A and MARIN 37 ducts. The duct cross-section is shown at the left, the duct in three-dimensions along with a propeller and cigar hub at the right.*

The ordinate is the distance of the point above the mean cambre line, also non-dimensionalized using the chord length.

`std::vector<AirfoilPt> xdown;`
Similar to `xup` but the points are on the inner surface.

`bool sharp_te;`
True if the trailing edge is sharp.

A function object for the duct cross-section can be obtained using

```
CurveLib::Curve<1U,ZRPoint,Float> cross_section() const;
```

Here `ZRPoint` represents a point $(c_z(\xi_d), c_r(\xi_d))$ on the duct cross-section; it is an alias for `VecMtx::VecN<2U,Float>`. Values of $c_z(\xi_d)$ or its derivatives can also be obtained using

```
Float z(Float xid, unsigned d = 0) const;
```

where `xid` is the value of $\xi_d$ and `d` is the number of derivatives to take.

The airfoil $\Big(x_d(\xi_d), y_d(\xi_d)\Big)$ used to define the cross-section is obtained using

```
Afoil::Airfoil<Float> get_airfoil() const;
```

The member function

```
int is_inside(const Point &x) const;
```

determines if the point `x` is inside the duct. The possible returned values are

0: if `x` is inside the duct;
1: if `x` is above the inner surface of the duct;
2: if `x` is upstream of the start of the duct; and
3: if `x` is downstream of the end of the duct.

The member function

```
Float distance_to_inner_surface(const Point &x) const;
```

returns the distance of the point `x` from the inner surface of the duct. A positive value means the point is below the inner surface, negative means it is above. It is useful for determining the clearance between the propeller tip section and the duct. Alternatively, suppose the function object, `crv`, represents a curve $\mathbf{x}(\xi)$ which returns a point in space as a function of $\xi$. Then the member function

```
CurveLib::Curve<1U,Float,Float> distance_to_inner_surface(
                    const CurveLib::Curve<1U,Point,Float> &crv) const;
```

returns a function object giving the distance of $\mathbf{x}(\xi)$ from the inner duct surface as a function of $\xi$.

The size of a duct can be scaled simply by multiplying it by a `Float`:

```
Duct operator*(const Duct &d, Float s);
Duct operator*(Float s, const Duct &d);
```

Each function returns a duct equivalent to `d` scaled by `s`.

## 8.2 Making the trailing edge sharp

Most duct cross-sections have blunt trailing edges to facilitate their operation when the ship is backing. However, some applications require that the trailing edge be sharp: for example, boundary element methods require a sharp trailing edge to prevent flow separation and to be able to apply a Kutta condition. A blunt trailing edge can be made sharp by specifying the location of the sharp trailing edge and two points, one on each side of the duct, where the modified geometry will be faired into the original geometry. The trailing edge point is specified in the coordinate system of the airfoil defining the cross-section denoted by $(x_d, y_d)$. The two points where the new and old surfaces are merged are specified by values of $x_d$; these values must be in (0,1).

Let $(x_d, y_d) = (x_{te}, y_{te})$ be the location of the new trailing edge and let $x_i$ and $x_o$ be the value of $x_d$ for the points on the inner and outer surfaces, respectively, where the old and new geometries are merged. Let $\xi_i$ be the duct parameter $\xi_d$ corresponding to the value $x_i$: i.e. $x_d(\xi_i) = x_i$. Similarly define $\xi_o$ so that $x_d(\xi_o) = x_o$. The new duct cross-section will then be defined as:

$$x_d(\xi) = \begin{cases} x_i + x_d'(\xi_i)(\xi - \xi_i) + \frac{1}{2}x_d''(\xi_i)(\xi - \xi_i)^2 \\ \quad + \left[x_i - x_{te} - x_d'(\xi_i)\xi_i + \frac{1}{2}x_d''(\xi_i)\xi_i^2\right]\dfrac{(\xi - \xi_i)^3}{\xi_i^3} & \text{for } \xi < \xi_i; \\ x_d(\xi) & \text{for } \xi_i \le \xi \le \xi_o; \\ x_o + x_d'(\xi_o)(\xi - \xi_o) + \frac{1}{2}x_d''(\xi_o)(\xi - \xi_o)^2 \\ \quad + \Big[x_{te} - x_o - x_d'(\xi_o)(1 - \xi_o) \\ \qquad\qquad - \frac{1}{2}x_d''(\xi_o)(1 - \xi_o)^2\Big]\dfrac{(\xi - \xi_o)^3}{(1 - \xi_o)^3} & \text{for } \xi > \xi_o; \end{cases} \tag{46}$$

$$y(\xi) = \begin{cases} y_d(\xi_i) + y_d'(\xi_i)(\xi - \xi_i) + \frac{1}{2}y_d''(\xi_i)(\xi - \xi_i)^2 \\ \quad + \left[y_d(\xi_i) - y_{te} - y_d'(\xi_i)\xi_i + \frac{1}{2}y_d''(\xi_i)\xi_i^2\right]\dfrac{(\xi - \xi_i)^3}{\xi_i^3} & \text{for } \xi < \xi_i; \\ y_d(\xi) & \text{for } \xi_i \le \xi \le \xi_o; \\ y_d(\xi_o) + y_d'(\xi_o)(\xi - \xi_o) + \frac{1}{2}y_d''(\xi_o)(\xi - \xi_o)^2 \\ \quad + \Big[y_{te} - y_d(\xi_o) - y_d'(\xi_o)(1 - \xi_o) \\ \qquad\qquad - \frac{1}{2}y_d''(\xi_o)(1 - \xi_o)^2\Big]\dfrac{(\xi - \xi_o)^3}{(1 - \xi_o)^3} & \text{for } \xi > \xi_o. \end{cases} \tag{47}$$

This curve is $C^2$.

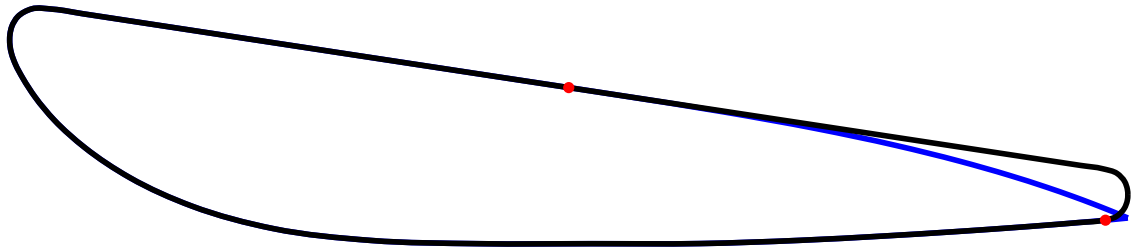


***Figure 15:*** *The MARIN 19A duct before (black) and after (blue) the trailing edge has been made sharp. The red dots show the points where the new trailing edge was faired into the original cross-section.*

The trailing edge of a duct can be made sharp using the following `Duct` member function:

```
void make_trailing_edge_sharp(const ZRPoint &te,
                              Float x_i, Float x_o,
                              Float *xi_i, Float *xi_o);
```

Here `te` specifies the new trailing edge point, `x_i` is the value of $x_i$ and `x_o` is the value of $x_o$. The values of $\xi_i$ and $\xi_o$ are returned in `xi_i` and `xi_o` respectively.

For example, a MARIN 19A duct could be closed as follows with trailing edge at $(x_d, y_d) = (1.0, 0.023)$, $x_i = 0.98$ and $x_o = 0.5$:

```
using namespace PGeom;
Duct duct19A("MARIN 19A",1.0,0.5,1.0875);
ZRPoint te(1.0,0.023);
Float xi_i, xi_o;
duct19A.make_trailing_edge_sharp(te,0.98,0.5,&xi_i,&xi_o);
```

The result is shown in Fig. 15. The blue curve is the new trailing edge. The two red dots show the locations where the new trailing edge was faired into the original cross-section.

Use

```
bool has_sharp_trailing_edge() const;
```

to determine whether the duct has a sharp trailing edge.

## 8.3 Defining a duct from a file

A `Duct` can be defined by reading records in OFFSRF format [11] from an input file. The class `DuctReader` is used to read the file and construct the duct. `DuctReader` is derived from `Offsrf::Base` so it has an inserter defined for reading the file. It also

has a public member `duct` of type `Duct` to represent the duct non-dimensionalized using the propeller diameter. The following code reads the file `duct.dat` to define a duct.

```
using namespace PGeom;
DuctReader reader;
Offsrf::IFStream in("duct.dat");
if (!in) throw Error("Could not open duct.dat");
in >> reader;
if (!in.good() && !in.eof())
  throw Error("Error when reading duct.dat.");
if (!reader.duct.is_defined())
  throw Error("Duct undefined after reading blade.dat.");
Duct duct = reader.duct;
```

The following records are used to specify the size and location of the duct:

`{CHORD LENGTH/PROP RADIUS:` $L/R$ `}`
`{RADIAL OFFSET:` $r_{off}/R$ `}`
`{Z PROP/CHORD:` $\zeta_p$ `}`

`DuctReader` recognizes three different records to define the shape of the duct cross-section: `AIRFOIL`, `DESIGNATION` and and `OFFSETS`. They are mutually exclusive, so only one of them should be present in the file.

An `AIRFOIL` record simply encloses records to define an arbitrary airfoil:

`{AIRFOIL`
  *<Records to define an airfoil: see Ref. 10, Sec. 7>*
`}AIRFOIL`

A `DESIGNATION` record defines the cross-section using a string to specify one of the pre-defined duct cross-sections:

`{DESIGNATION:` *designation* `}`

Currently *designation* must be either `MARIN 19A` or `MARIN 37`.

An `OFFSETS` record defines a series of $(x_d, y_d)$ offsets to define the airfoil shape:

`{OFFSETS`
  $x_{d1}$ $y_{d1}$
  ...
  $x_{dn}$ $y_{dn}$
`}OFFSETS`

To ensure that the trailing edge of the duct is sharp, use the following record:

{SHARP TRAILING EDGE: $x_{te}$ $y_{te}$ $x_i$ $x_o$ }

For example, the following records define the duct whose cross-section is shown in Fig. 15:

```
{DESIGNATION: MARIN 19A }
{CHORD LENGTH/PROP RADIUS: 1.0 }
{RADIAL OFFSET: 1.0875 }
{Z PROP/CHORD: 0.5 }
{SHARP TRAILING EDGE: 1.0 0.023  0.98 0.5 }
```

# 9 Classes representing propellers

There are two classes for representing propellers: `Propeller` and `SectionPropeller`. The former uses a `Blade` to represent the reference blade, the latter a `SectionBlade`.

## 9.1 The base class Propeller

A propeller is defined using three basic components: a reference blade, a hub and a duct. Although the propeller classes defined here will allow any of these components to be missing, in practice the reference blade will always be present, the hub will almost always be present, but the duct is often missing. Each component is defined independent of the others. The hub and the reference blade intersect (or at least touch) and are connected by their intersection curve. Currently, no provision is made for fillets between the blade and the hub.

`Propeller` is a base class for propellers. It contains a reference blade of type `Blade` (see Sec. 6.1), a hub of type `Hub` (see Sec. 7) hub and a duct of type `Duct` (see Sec. 8). The number of blades is also stored.

The `Propeller` class must also store the propeller diameter since the reference blade, hub and duct are all given in non-dimensional form.

### 9.1.1 Constructors

`Propeller` has the following constructor in addition to the default (no argument) and copy constructors:

```
Propeller(unsigned nb, Float d, Blade b, Hub h = Hub(),
          Duct dct = Duct());
```

It makes a propeller with diameter `d`, `nb` copies of blade `b`, a hub `h` and a duct `dct`. When the the default argument is used for the duct, it will remain undefined; the propeller is then assumed to have no duct. Similarly, if the hub is undefined, the propeller will have no hub.

The propeller can also be defined using the member function:

```
virtual void define(unsigned nb, Float d, Blade b, Hub h = Hub(),
                    Duct dct = Duct());
```

which is similar to the constructor. It is virtual so that it can be redefined by `SectionPropeller` to ensure that the blade is actually a `SectionBlade`.

If the default constructor is used, the propeller remains undefined. It must later be defined by assignment to another propeller, or by using the `define` function.

### 9.1.2 Member functions for the propeller description

For identification purposes, each propeller contains the following descriptive strings:

- an origin: the organization where the propeller originated;
- a designation: the number or designator assigned to the propeller by the originating organization;
- a description: any other other information that might be useful for identifying the propeller.

The following member functions are provided for setting and obtaining these strings:

```
void set_origin(const Str &s);
Str origin() const;
void set_designation(const Str &s);
Str designation() const;
void set_description(const Str &s);
Str description() const;
```

### 9.1.3 Member functions for obtaining the principal characteristic of the propeller

The number of blades of the propeller can be obtained using

```
unsigned number_of_blades() const;
```

and its diameter using

```
Float diameter() const;
```

Use

```
bool is_right_handed() const;
```

to determine whether the propeller is right-handed: i.e. rotates clockwise when viewed from astern.

### 9.1.4 Member functions for the propeller components

To determine which components a propeller has, you can use the member functions

```
bool has_blade() const;
bool has_hub() const;
bool has_duct() const;
```

Each returns true if that component is defined. To get a representation of one of the components, use

```
Blade get_blade() const;
Hub   get_hub() const;
Duct  get_duct() const;
```

The surfaces of blades other than the reference blade can be obtained using

```
Surface get_blade(unsigned n) const;
```

where `n` is the number of the blade you want. The numbering is from 0 to $N-1$ with 0 being the reference blade. If the surface is denoted $\mathbf{b}_n(\xi,\eta)$, then

$$\mathbf{b}_n(\xi,\eta) = \mathbf{M}_n\mathbf{b}(\xi,\eta) \tag{48}$$

where $\mathbf{M}_n$ is a $3\times 3$ rotation matrix which generates blade $n$ from the reference blade by rotating about the propeller axis:

$$\mathbf{M}_n = \begin{bmatrix} \cos(2\pi n/N) & -\sin(2\pi n/N) & 0 \\ \sin(2\pi n/N) & \cos(2\pi n/N) & 0 \\ 0 & 0 & 1 \end{bmatrix}. \tag{49}$$

To change one of the components, use

```
virtual void set_blade(Blade b);
virtual void set_hub(Hub h);
virtual void set_duct(Duct d);
```

These functions are virtual because changing a component may also require other changes in derived classes. A cylindrical hub can also be added to the propeller using

```
void make_cylindrical_hub(Float radius, Float z0, Float z1,
                          bool endcaps);
```

which makes a cylindrical hub of given radius extending from `z0` to `z1` in the Z direction. If `endcaps` is true, hemispherical end caps are added while keeping the length of the hub unchanged: in this case an `Error` is thrown if `2*radius` exceeds `z1-z0`.

The surface of the blade will normally extend into the interior of the hub so that the intersection between the blade and hub is well-defined. The member function

```
Surface blade_above_hub(unsigned n = 0) const;
```

returns a surface representing the portion of blade `n` above the hub. If there is no hub, the surface is the same as the full blade surface. The first parameter of the surface is the same as blade $\xi$. The second parameter, $\eta'$, varies from 0 at the hub-blade intersection to 1 at the tip of the blade. Let $\eta_h(\xi)$ denote the value of $\eta$ at which the blade intersects the hub (this can be obtained using the function `blade_param_curve_at_hub` described in Sec. 9.1.6). Then

$$\eta' = \frac{\eta-\eta_h(\xi)}{1-\eta_h(\xi)}; \qquad \eta = (1-\eta')\eta_h(\xi)+\eta'. \tag{50}$$

If the surface is denoted $\mathbf{b}_n(\xi, \eta')$, then

$$\mathbf{b}_n(\xi, \eta') = \mathbf{M}_n \mathbf{b}\Big(\xi, (1-\eta')\eta_h(\xi) + \eta'\Big) \tag{51}$$

where $\mathbf{M}_n$ is given by Eq. (49).

### 9.1.5 Member functions to simulate a controllable pitch propeller

In order to simulate a controllable pitch propeller, the reference blade can be rotated about the propeller reference line by a given angle. Use the following member functions:

```
void set_blade_rotation_around_generator(Angle<Float> ang);
Angle<Float> get_blade_rotation_around_generator() const;
```

When a new blade rotation is set, the `Blade` function `set_rotation` is called to rotate the blade, then a check is made to ensure that the blade/hub intersection is still well-defined. If it is not, a warning message will be written (see Sec. 2.4).

### 9.1.6 Member functions for the hub/blade intersection

To check that the intersection of the blade and the hub is defined, use

```
bool blade_hub_intersection_defined() const;
```

It will return false, for example, if either of the blade or hub are missing or if the root of the blade is above the hub surface.

The following two member functions return the intersection curve in the parameter space of the blade and in the parameter space of the hub respectively:

```
SurfaceCurve blade_param_curve_at_hub() const;
SurfaceCurve hub_param_curve_at_blade() const;
```

Each function will throw a `ProgError` if the intersection is not defined. The parameter of each of the returned curves is the same as the blade parameter $\xi$. These curves can be composed with the blade and hub, respectively, to create curves which return points on the intersection in $(x, y, z)$ coordinates:

```
using namespace PGeom;
Propeller prop;
Blade blade = prop.get_blade();
Hub hub = prop.get_hub();
SurfaceCurve b_param_crv =  prop.blade_param_curve_at_hub();
SurfaceCurve h_param_crv =  prop.hub_param_curve_at_blade();
CurveLib::Curve<1U,Point,Float>
```

```
bcrv = blade(b_param_crv),
hcrv = hub(h_param_crv);
```

The curves `bcrv` and `hcrv` should be equivalent. In practice, because the intersection curve cannot be calculated exactly, they will differ by a small amount. The accuracy of the parametric intersection curves has been set so that for any `xi` with value in [0.0,1.0], `abs(bcrv(xi)-hcrv(xi))` will not exceed 0.1 times the current propeller accuracy (as determined from the function `get_prop_accuracy`: see Sec. 2.5).

## 9.2 The class SectionPropeller

The class `SectionPropeller` is a specialization of the base class `Propeller` in which the blade is represented as a `SectionBlade` rather than the more generic `Blade` (`SectionBlade` is described in Sec. 6.2). Thus, its blades are constructed from a series of airfoils along with curves to define the total rake, skew angle, pitch and chord length. `SectionPropeller` inherits all the `Propeller` public member functions except for `Propeller::define` and `Propeller::set_blade` which are hidden to prevent a blade being assigned which is not a `SectionBlade`. The member function `get_blade` is overloaded to return a `SectionBlade`:

```
SectionBlade get_blade() const;
```

`SectionPropeller` has default and copy constructors. Use

```
SectionPropeller(unsigned nb, Float d, SectionBlade b,
                 Hub h = Hub(), Duct dct = Duct());
```

to make a propeller with diameter `d`, `nb` copies of blade `b`, a hub `h` and a duct `dct`. When the the default argument is used for the duct, it will remain undefined; the propeller is then assumed to have no duct. Similarly, if the hub is undefined, the propeller will have no hub.

A `SectionPropeller` with no duct can also be constructed from the data in a `PropSectionData` (see Sec. 9.3).

```
SectionPropeller(const PropSectionData &pdata, bool rh = true,
                 bool close_te = true, bool splined = false,
                 bool c2 = false, bool close_tip = false);
```

The arguments are passed to `SectionBlade` and `SplineHub` constructors; see Sec. 6.2 and Sec. 7.1.3 for details on how the arguments are actually used.

The following member functions can be used to redefine a `SectionPropeller`. They are similar to the constructors having the same arguments.

```
void define(unsigned nb, Float d, SectionBlade b, Hub h = Hub(),
            Duct duct = Duct());
```

```
void define(const PropSectionData &pdata, bool rh = true,
            bool close_te = true, bool splined = false,
            bool c2 = false, bool close_tip = false);
```

## 9.3 The class PropSectionData

In the early 1980s, DRDC (then DREA) developed a system for storing information concerning ship propulsion. Part of that system was a file format to specify offset data describing the geometry of a propeller [16]. The class `PropSectionData` represents the data in a DREA propeller geometry file. It can be interpolated in various ways to generate smooth representations of the propeller.

The blade sections are defined using a sequence of $(x_s, y_s)$ points to define the section shape: see Fig. 3. They are represented by the class `AirfoilPt` which, as explained in Sec. 6.2.1, is an alias for `VecMtx::VecN<2U,Float>`.

The following public members represent the data:

`Str origin;`
: The origin of the propeller.

`Str designation;`
: The designation of the propeller.

`Str description;`
: The description of the propeller.

`unsigned num_blades;`
: The number of blades on the propeller

`Float diameter;`
: The diameter of the propeller.

`Float hub_diameter;`
: The ratio of the hub diameter to the propeller diameter.

`std::vector<Float> radii;`
: The value of $r$ for each section. The radii must be strictly increasing.

`std::vector<Angle<Float> > skew_vals;`
: The values of the skew at each of the radii.

`std::vector<Float> rake_vals;`
: The values of the rake at each of the radii. These values are non-dimensionalized with respect to the propeller diameter.

`std::vector<Float> pitch_vals;`
: The values of the pitch at each of the radii. These values are non-dimensionalized with respect to the propeller diameter.

`std::vector<Float> chord_vals;`
: The values of the chord length at each of the radii. These values are non-dimensionalized with respect to the propeller diameter.

`std::vector<std::vector<AirfoilPt> > xup;`
: A sequence of $(x_s, y_s)$ points on the suction side of the blade section at each radius. The points are in order from the leading edge to the trailing edge and are non-dimensionalized using the chord length of the section. The abscissa of each point is the fractional chord length measured from the leading edge: i.e. 0 at the leading edge, 1 at the trailing edge. The ordinate is the distance of the point above the mean cambre line.

`std::vector<std::vector<AirfoilPt> > xdown;`
: Similar to `xup` but giving the points on the pressure side of the section curve at each radius. Since the points are normally below the mean cambre line, the ordinates of these points are usually negative.

`std::vector<Hub::ZRPoint> hub_points;`
: Points on the spine of the hub. Each point is a $(z, r)$ pair where $z$ is the distance along the hub non-dimensionalized with respect to the propeller diameter.

`bool sharp_te;`
: True if the trailing edge is sharp on closed sections: i.e. the normal on the suction side of the blade will not be the same as the normal on the pressure side at the trailing edge.

`bool sharp_tip;`
: True if the tip is sharp: i.e. there will not be a single well-defined normal to the blade at the tip.

The following public member functions are also defined:

`bool has_open_trailing_edge() const;`
: Returns true if the trailing edge is open: i.e. the thickness of the section at the trailing edge is non-zero so that the blade does not close.

`bool has_open_tip() const;`
: Returns true if the tip is open.

`void add_points_at_leading_edge();`
: Ensures that on every section the offsets for both sides of the blade include a point at the leading edge. If only the face has one, it will be copied to the back and vice versa. If neither has one, the two leading points on the face and the two leading points on the back will be splined and the location of the leading edge point determined from the spline. If both have a point but they are different, an `Error` is thrown.

The following functions can be used to read or write the data to I/O streams:

`std::istream& operator>>(std::istream&, PropSectionData&);`
: Reads propeller section data from an input stream in DRDC format. The trailing edge is assumed to be sharp as this information is not present in the DRDC geometry file.

`std::ostream& operator<<(std::ostream&, const PropSectionData&);`
: Writes propeller section data to an output stream in DRDC format.

`void describe(std::ostream&, PropSectionData&);`
: Writes a description of propeller section data to an output stream.

## 9.4 Defining a propeller from a file

A `SectionPropeller` can be defined by reading records in OFFSRF format [11] from an input file. The class `PropReader` can be used to read the file and construct the propeller. `PropReader` is derived from `Offsrf::Base` so it has an inserter defined for reading the file. It also has a public member `propeller` of type `SectionPropeller` to represent the propeller. The following code reads the file `prop.dat` to define a propeller.

```
using namespace PGeom;
PropReader reader;
Offsrf::IFStream in("prop.dat");
if (!in) throw Error("Could not open prop.dat");
in >> reader;
if (!in.good() && !in.eof())
  throw Error("Error when reading prop.dat.");
SectionPropeller propeller = reader.propeller;
```

The input file can contain a `DREA PROPELLER GEOMETRY` record:

`{DREA PROPELLER GEOMETRY:` *file-name*
  *<Records defining the blade properties>*
`}DREA PROPELLER GEOMETRY`

where *file-name* is the name of the file and the records defining the blade properties are the same as those described in Sec. 6.2.6.1.

Alternatively, the propeller can be defined using records to define each of its components as follows:

`{NUMBER OF BLADES:` $N$ `}`
`{DIAMETER:` $D$ `}`
`{BLADE`
  *<Any records recognized by* `BladeReader`*: see Sec. 6.2.6>*
`}BLADE`
`{BLADE ROTATION AROUND GENERATOR:` $\theta$ `} ! degrees`

```
{HUB
  <Any records recognized by HubReader: see Sec. 7.2>
}HUB
{DUCT
  <Any records recognized by DuctReader: see Sec. 8.3>
}DUCT
```

Any of these records can also be used after a `DREA PROPELLER GEOMETRY` record; the components defined in the geometry file will then be replaced by the new component defined by the records which succeed it.

The following records define the geometry of the David Taylor Model Basin (DTMB) P4119 propeller [19] often used as a test case for propeller analysis programs. It uses the DTMB modification of the NACA 66 airfoil [18] for each of its sections and has no generator rake or skew. A cigar hub is used and there is no duct. A display of the blade is shown in Fig. 16.

```
{ORIGIN: DTMB }
{DESIGNATION: P4119 }
{NUMBER OF BLADES:3}
{DIAMETER:0.3048}
{BLADE
  {SCALED BLADE SECTIONS
    {USE GENERATOR RAKE}
    {THICKNESS DISTRIBUTION AIRFOIL
      {NACA 66 DTMB(mod): 1 1 }
    }THICKNESS DISTRIBUTION AIRFOIL
    !          r/R   t/L     c/L    p/D    L/D  skew i_G/D
    {SECTION: 0.20 0.20550 0.01429 1.105 0.3200  0.0  0.0 }
    {SECTION: 0.30 0.15530 0.02318 1.102 0.3625  0.0  0.0 }
    {SECTION: 0.40 0.11800 0.02303 1.098 0.4048  0.0  0.0 }
    {SECTION: 0.50 0.09160 0.02182 1.093 0.4392  0.0  0.0 }
    {SECTION: 0.60 0.06960 0.02072 1.088 0.4610  0.0  0.0 }
    {SECTION: 0.70 0.05418 0.02003 1.084 0.4622  0.0  0.0 }
    {SECTION: 0.80 0.04206 0.01967 1.081 0.4347  0.0  0.0 }
    {SECTION: 0.90 0.03321 0.01817 1.079 0.3613  0.0  0.0 }
    {SECTION: 0.95 0.03228 0.01631 1.077 0.2775  0.0  0.0 }
    {SECTION: 1.00 0.03125 0.01175 1.075 0.0000  0.0  0.0 }

    {RIGHT HANDED}
    {CLOSE TRAILING EDGE}
    {DULL TRAILING EDGE}
  }SCALED BLADE SECTIONS
}BLADE
{CIGAR HUB: 0.1 -0.3333 0.3 }
```

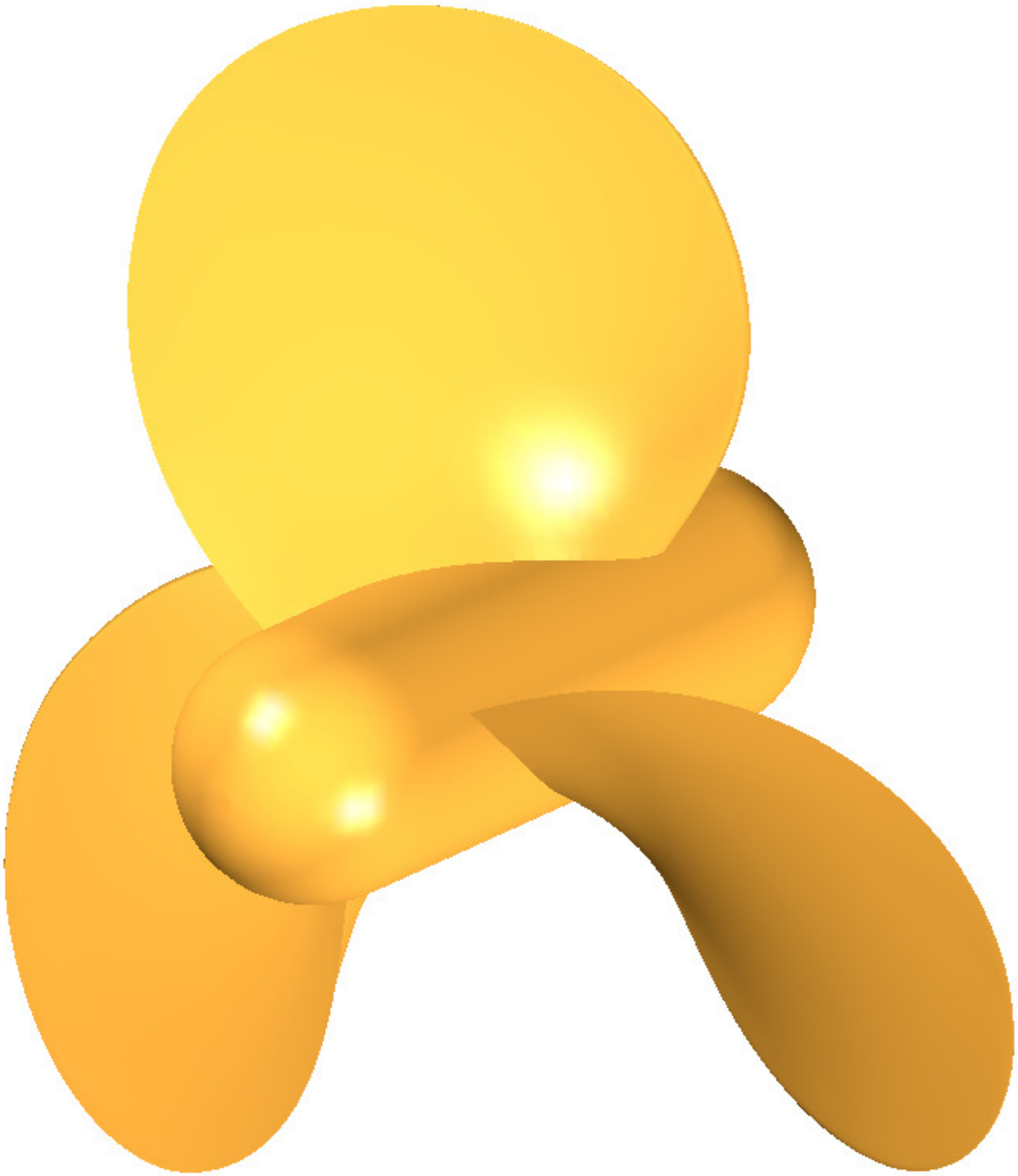


***Figure 16:*** *The propeller DTMB P4119.*

# 10 Concluding remarks

A library of C++ classes has been written to permit the accurate representation of propeller geometry. The library is based on the CurveLib and Airfoil libraries of C++ classes that have been documented previously [1,9,10].

The propeller geometry can be used in any C++ application for which a representation of the propeller is needed. They are already being used in the program Provise [2,3] developed by Cooperative Research Ships as a front end to its propeller analysis code PROCAL [4,5], and in an application for generating IGES [15] representations of propellers suitable for use in RANS solvers [6,7]. These applications are being used by DRDC to support the design of the propellers for the new ships in the Royal Canadian Navy.

# References


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

[2] Noble, D. J. and Heath, D. C. (2005), User's Guide for the PROCAL Visualization Environment, PROVISE: Final Report for Release Version 1.0, (DRDC Atlantic ECR 2005-140) Defence Research and Development Canada – Atlantic.

[3] Noble, D. J. and Heath, D. C. (2008), User's Guide for the PROCAL Visualization Environment, PROVISE: Final Report for Release Version 2.0, (DRDC Atlantic ECR 2008-255) Defence Research and Development Canada – Atlantic.

[4] Bosschers, J. (2013), User's guide PROCAL baseline code version 2.21, (MARIN Report No. 26295-1-RD) MARIN, Wageningen, The Netherlands.

[5] Bosschers, J. (2009), PROCAL v2.0 Theory Manual, (MARIN Report No. 27834-7-RD) MARIN, Wageningen, The Netherlands.

[6] Hally, D. (2013), Smoothing propeller tip geometry for use in a RANS solver, (DRDC Atlantic TM 2013-178) Defence Research and Development Canada – Atlantic.

[7] Hally, D. (2013), User's guide for smooth-prop: a program to smooth propeller tip geometry, (DRDC Atlantic TM 2013-179) Defence Research and Development Canada – Atlantic.

[8] Stroustrup, B. (1997), The C++ Programming Language, 3rd ed, Ch. 21.3.8, Addison-Wesley Publishing Co.

[9] Hally, D. (2006), C++ classes for representing curves and surfaces: Part II: Splines, (DRDC Atlantic TM 2006-255) Defence Research and Development Canada – Atlantic.

[10] Hally, D. (2010), C++ classes for representing airfoils, (DRDC Atlantic TM 2009-053) Defence Research and Development Canada – Atlantic.

[11] Hally, D. (1994), C++ Classes for Reading and Writing files in OFFSRF Format, (DREA Tech.Comm. 94/302) Defence Research and Development Canada – Atlantic.

[12] Abbott, I. H. and von Doenhoff, A. E. (1958), Theory of Wing Sections, Dover Publications Inc., New York.

[13] (1999), ITTC – Recommended Procedures: Propeller Models: Terminology and Nomenclature for Propeller Geometry. 7.5-01, 02-01. Available on-line at http://ittc.sname.org/2002_recomm_proc/7.5-01-02-01.pdf. (Access Date: August 2013).

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

[15] (1988), Initial Graphics Exchange Specification (IGES) Version 4.0, US Dept. of Commerce, National Bureau of Standards. Document No. NBSIR 88-3813.

[16] Trumpler, D. (1981), DREA Ship Propulsion Database: Report 1, Detailed System Manual, Evans Computing Applications Ltd., Halifax, NS, Canada.

[17] Boswell, R. J. (1971), Design, Cavitation Performance, and Open-water Performance of a Series of Research Skewed Propellers, (Report 3339) Naval Ship Research and Development Centre, Washington, U.S.A.

[18] Brockett, T. (1966), Minimum Pressure Envelopes for Modified NACA-66 Sections with NACA a = 0.8 Camber and BuShips Type I and Type II Sections, (Report 1780) David W. Taylor Naval Ship Research and Development Center.

[19] Jessup, S. D. (1998), Experimental data for RANS calculations and comparisons (DTMB4119), In *Proceedings of Propeller RANS/Panel Method Workshop*, 22$^{nd}$ ITTC Propulsion Committee, Grenoble, France.

# List of symbols

| | |
|---|---|
| $\epsilon$ | The propeller accuracy. |
| $\zeta_p$ | The location of the propeller plane relative to the duct leading edge as a fraction of the duct chord length. |
| $\eta_h(\xi)$ | The value of $\eta$ at the intersection of the reference blade with the hub. |
| $\eta^{max}$ | The maximum value of $\eta$ on the blade. |
| $\eta^{min}$ | The minimum value of $\eta$ on the blade. |
| $\eta^{tip}$ | The value of $\eta$ at the blade tip. |
| $\theta$ | Angular coordinate about the propeller axis: see Sec. 4. |
| $\theta^{le}$ | The value of $\theta$ for the point on the leading edge of a blade section. |
| $\theta_s$ | Skew angle. |
| $\theta^{te}$ | The value of $\theta$ for the point on the trailing edge of a blade section. |
| $(\xi, \eta)$ | Parameters for a point on the reference blade. |
| $(\xi_h, \theta)$ | Parameters for a point on the hub. |
| $\xi^{tip}$ | The value of $\xi$ at the blade tip. |
| $\phi$ | Pitch angle. |
| $c$ | Cambre of a blade section. |
| $D$ | Propeller diameter. |
| $i_G$ | Generator rake. |
| $i_S$ | Skew-induced rake. |
| $i_T$ | Total rake. |
| $L$ | Chord length of a blade section. |
| $L_d$ | Chord length of propeller duct. |
| $\mathbf{M}$ | A unitary matrix to transform points from the propeller coordinate system to the hull coordinate system. |
| $N$ | The number of blades. |

| | |
|---|---|
| $P$ | Pitch. |
| $r$ | The distance of a point from the propeller axis normalized using the propeller radius. |
| $r_{off}$ | The radial offset of the duct cross-section shape. |
| $R$ | The propeller radius. |
| $t$ | Thickness of a blade section. |
| $(X, Y, Z)$ | Cartesian coordinates aligned with the propeller axis: see Sec. 4 and Fig. 1. |
| $\mathbf{X}_0$ | The centre of the propeller disk in hull coordinates. |
| $\mathbf{X}_h$ | A point in hull coordinates. |
| $\mathbf{X}_p$ | A point in propeller coordinates. |
| $(x, y, z)$ | Cartesian coordinates aligned with the propeller axis and non-dimensionalized using the propeller diameter: see Sec. 4. |
| $(x_d, y_d)$ | Coordinates of the airfoil defining a duct cross-section: see Eq. 44. |
| $xi$, $x_o$ | Values of $x_d$ used to specify where a sharp duct trailing edge is faired into the original duct. |
| $x_{le}$, $x_{te}$ | The location of the sharp trailing edge of a duct cross-section. |
| $(x_s, y_s)$ | Coordinates of a blade section: see Fig. 3. |
| $z_h$ | The value of $z$ which splits the hub into rotating and stationary parts. |
| $z_R$ | Cylindrical coordinate along the propeller axis: $z_R = Z/R$. |
| $z_R^{le}$ | The value of $z_R$ for the point on the leading edge of a blade section. |
| $z_R^{te}$ | The value of $z_R$ for the point on the trailing edge of a blade section. |

# Index

This page intentionally left blank.

## DOCUMENT CONTROL DATA

(Security markings for the title, abstract and indexing annotation must be entered when the document is Classified or Designated.)

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 Research and Development Canada – Atlantic
PO Box 1012, Dartmouth NS B2Y 3Z7, Canada

2a. SECURITY MARKING (Overall security marking of the document, including supplemental markings if applicable.)

UNCLASSIFIED

2b. CONTROLLED GOODS

(NON-CONTROLLED GOODS)
DMC A
REVIEW: GCEC APRIL 2011

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 propeller geometry

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.) |
|---|---|---|
| October 2013 | 94 | 19 |

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 Research and Development Canada – Atlantic
PO Box 1012, Dartmouth NS B2Y 3Z7, Canada

| 9a. PROJECT OR GRANT NO. (If appropriate, the applicable research and development project or grant number under which the document was written. Please specify whether project or grant.) | 9b. CONTRACT NO. (If appropriate, the applicable number under which the document was written.) |
|---|---|
| Project 01gl08 | |

| 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 2013-177 | |

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.)

To satisfy the need for high fidelity representations of propellers for use in Reynolds-averaged Navier-Stokes (RANS) flow solvers and other propeller applications, a library of C++ classes has been developed. It provides classes representing the surfaces of the propeller blades, hubs and ducts in fully differentiable form. The propeller blades can be defined using the traditional method of specifying the blade section shape, chord length, pitch, skew and rake at a series of radial sections. More general blade shapes are also possible provided that they conform to fairly loose requirements on the blade parameterization. Simple cylindrical and cigar-shaped hubs can be defined as well as more general axisymmetric shapes defined from splined offsets. Similarly, propeller ducts can be defined from commonly used duct cross-sections or more general shapes defined from offsets.

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.)

Propellers; Computational fluid dynamics; Computer programs; C++; IGES

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

DEFENCE DÉFENSE

**www.drdc-rddc.gc.ca**